A Static Analyzer for Model Transformations

Andreza Vieira and Franklin Ramalho

Federal University of Campina Grande,
Campina Grande, Paraíba, Brazil
andreza@copin.ufcg.edu.br
franklin@dsc.ufcg.edu.br

Abstract. Adoption of the Model-Driven Architecture approach is increasing each day. As any other software development project, a MDA-based project is constantly evolving given that software requirements change along their lifecycle. Thus, changes in MDA transformations are also frequent. These changes are hard and error-prone tasks when manually accomplished. We propose a static analyzer for inspecting the source code of ATL transformations. It provides an API comprising methods to extract and handle diverse elements from ATL transformations. Therefore, the proposed static analyzer can be useful during several development tasks of MDA-based projects, such as maintenance and debugging, since it enables developers to save effort and development time by automatically identifying dependences and relations from transformation elements.

Keywords: Static analysis, transformation code analysis.

1 Introduction and Motivation

Within software engineering field new trends related to software development process are emerging and showing their benefits, such as greater automation and higher productivity. The Object Management Group (OMG) [16] has proposed an approach that works in this direction: the Model-Driven Architecture (MDA) [12]. The main goal of this approach is to shift the focus from code to models in the software development process. MDA allows the automatic generation of output models from input models by means of transformation definitions, which are transformation rules describing how input models can be transformed into output models. These transformation definitions are expressed through a transformation language, such as ATLAS Transformation Language (ATL) [3], and they are automatically performed by transformation engines.

Software development projects are constantly evolving given that software requirements change along the whole project lifecycle. Similarly, MDA-based projects evolve along their lifecycle and require changes in their transformations. However, to perform changes in large transformations is a hard task given that developers have to manually look for dependencies and relations of the transformation element being changed, which is a waste of effort and development time. For instance, to exclude a given rule named *R1* it is necessary to know if it is invoked by another rule along the whole transformation chain. Indeed, manually performing this task is error-prone because it depends on the developers skills.

An approach that can be adopted to accomplish the problems previously cited is the static analysis. It examines either the source code or the bytecode of a program without executing it [15]. Then, all information about any transformation can be captured and handled as necessary, mainly to identify elements that can be impacted with any change in a transformation. The static analysis can be adopted for different purposes within software engineering, such as for locating any particular code path that might lead to potential runtime errors or for analyzing the change impact of software. Automated tools can assist developers in carrying out static analysis. Within software engineering there are a number of static analysis tools (static analyzers) for inspecting diverse programming languages, such as FindBugs [9] for Java code and *MOPS* (MOdelchecking Programs for Security properties) [14] for C code.

On the other hand, in the MDA context there are no static analysis tools that allow the inspection of ATL transformations. Just one work [19] proposes a static analysis approach in this context. However, it is focused on detection of common errors in model transformations specified just in VIATRA2 VTCL (Viatra Textual Command Language) [4] by inspecting their source codes. VIATRA2 is a framework for model to model (M2M) transformations that uses a high level language (VTCL) combining elements of model transformations and abstract state machines. However, [19] neither handles M2M transformations specified in ATL language nor provides an API with methods for helping developers to manipulate transformation elements in a customized way, mainly for maintenance tasks.

In this work, we propose a static analyzer for M2M transformations specified in ATL. It provides an API containing a number of methods that can be used to manipulate elements enclosed by an ATL transformation, such as rules, helpers, models and metamodels. The static analyzer is very useful to help developers to accomplish several development tasks in transformations, such as maintenance and debugging, given that: (i) developers will save effort and development time by automatically finding out the dependencies and relations of the element being changed in a transformation; (ii) it is not error-prone; (iii) it can analyze even very long transformations without additional effort; and (iv) its API is very easy to understand.

This paper is organized as follows. In Section 2, we introduce some concepts about MDA and Static Analysis. In Section 3, we explain details about our static analyzer implementation and give an overview about our approach. In Section 4, we present two examples illustrating how the proposed static analyzer can be applied to obtain information about an ATL transformation. In Section 5, we give an overview concerning related works on static analyzers within software engineering field and within MDA context. Finally, Section 6 summarizes our conclusions and gives some pointers to future work.

2 Background

2.1 MDA and ATL Transformations

MDA is an approach proposed by the OMG whose main objective is to provide an entire infrastructure for the software development in which the specification of a system is separated from its implementation details. The key of MDA is the importance attributed to the models during the software development process [12]. In this approach, the effort and time spent during the tests and implementation tasks of the software development lifecycle are shifted to modeling, metamodeling and transformations tasks.

Metamodels are pivotal elements in the MDA framework. Among other roles, they are used to define languages abstract syntax, such as process and modeling languages. For instance, UML class diagrams allow modelers to design classes, attributes, associations and other elements because they are defined by the UML metamodel.

The automatic generation of output models from input models happens by means of a transformation definition: a set of transformation rules that describe how input models can be transformed into output models. Transformation definitions are expressed through a transformation language, such as Query/View/Transformation (QVT) [17] and ATL. Finally, a tool called transformation engine is used to execute the transformation definitions.

Nowadays, ATL is a very popular transformation language employed to specify M2M transformations. By using this language, we can define three kinds of units [3]:
(i) An *ATL library*. It enables developers to specify helpers and attributes that can be imported from different kinds of ATL units, including libraries themselves. Helpers and attributes are, respectively, methods and constants defined within any ATL unit;
(ii) An *ATL query*. It enables developers to specify a query on a number of source models. It can be composed by a single query element, helpers and attributes;
(iii) An *ATL transformation module*. It enables developers to specify the way to produce a set of target models from a set of source models. It can be composed by helpers, attributes and rules.

ATL supports three kinds of rules: matched rules, called rules and lazy rules. The matched rules are declarative rules that constitute the core of an ATL transformation given that they enable developers to specify for which kinds of source elements target elements must be generated and the way the generated target elements have to be initialized. The called rules are imperative rules that can generate target model elements. However, as opposed to matched rules, they do not match any source model element and they have to be called by a matched rule or another called rule. The lazy rules are like matched rules. However, as opposed to matched rules, they have to be called by another rule.

2.2 Static Analysis

Static analysis is a mechanism that aims at inspecting either the source code or the bytecode of programs without executing it [15]. It is applied to various fields of computer science including software engineering, in which a number of well-known static analysis tools have been proposed, such as FindBugs and Checkstyle [5] for Java code, as well as *MOPS* and *UNO* [11] for C code.

In the static analysis context, there are two techniques for inspecting a program [13]: (i) by means of its source code or (ii) by means of its bytecode. A source-level analysis generally provides a great facility for handling the program structure, such as loops, since it contains more information and it abstracts machine details. However, it can be time-consuming since it involves lexical and syntactic analysis of the source code and several verifications on the structure of the code. On the other hand, a bytecode-level analysis is more faithful, given that it inspects the code that is actually executed. In addition, it is independent from source syntax, name resolution, type checking, template/generics instantiation, etc. Then, it avoids redundant work with compiler tasks already done. However, the structure of a bytecode can be hard to understand, then becoming hard to inspect it.

Within software engineering field the static analysis tools are widely used to help developers to locate any particular code path that might lead to potential runtime errors. These runtime errors are generally errors which a compiler cannot find out, such as variables that were used without initialization, variables that were not used anywhere and logical inconsistencies. It is important to emphasize that static analysis tools are susceptible to detect some errors that are not real errors (false-positives). In these cases it is necessary to have a human analysis in order to judge if the error is a false-positive.

Furthermore, the information obtained from a static analysis tool can also be applied to other purposes within software engineering, such as: (i) to verify software's properties used in safety-critical computer systems and to locate potentially vulnerable code; (ii) to detect weaknesses in the source code at the exact location; (iii) to verify if the source code conforms to coding guidelines and standards; (iv) to prove properties about a given program, for instance, its behavior matches with its specification; and (v) to analyze the change impact of a software. For instance, suppose that a software developer needs to make some changes in his project. Then, it would be interesting if he/she applied a change impact analysis technique to figure out how the change can affect his project.

For exemplifying one of the applications of the static analysis mechanism we present at Source Code 1 a main Java method that hides a bug. Lines 5-6 instantiate a map whose key is the name of a person (`String`) and the value is a person (`Person`). At lines 7-8, we put just an element into this map. Line 9 obtains a person from the map by the key "`Bob`". Before printing the name of the city of the obtained person (line 11), we have verified if this person was not null (line 10). At line 13, we obtained the full name of the person and printed it. This method has no compilation error, but when it is executed it generates a *NullPointerException* because line 13 tries to obtain the full name of a person that has a null value. As it is a simple example, it is very easy to identify the line that generates the exception. However, it is very hard to find errors like that into large source codes. FindBugs is a static analysis tool that proposes to detect this kind of error, among several others, without executing the program.

Source Code 1: A main Java method containing one bug.

```
1   import java.util.HashMap;
2
3   public class Main {
4       public static void main(String[] args) {
5          HashMap<String, Person> namesMap =
6              new HashMap<String, Person>();
7          namesMap.put("Paul", new Person(
8              "Paul McDonald", "Los Angeles"));
9          Person person = namesMap.get("Bob");
10         if (person != null) {
11             System.out.print(person.getCity());
12         }
13         System.out.print(person.getFullName());
14     }
15 }
```

We have applied FindBugs to the main Java method presented at Source Code 1. Then, it detected the exact code line that generated the *NullPointerException* (line 13) and reported the bug message and description presented as follows.

Bug:	Possible null pointer dereference of person.
Description:	There is a branch of statement that, if executed, guarantees that a null value will be dereferenced, which would generate a NullPointerException when the code is executed. Of course, the problem might be that the branch or statement is infeasible and that the null pointer exception can't ever be executed; deciding that is beyond the ability of FindBugs.

3 ATL Static Analyzer

The ATL static analyzer proposed in this work for M2M transformations provides an API comprising methods to manipulate the ATL transformation elements. It has been implemented using the Java programming language. The API has been made available under the General Public License (GPL) and it is available at [2]. An overview of its implementation is illustrated in Fig. 1. As we can observe, the static analyzer receives as input an ATL transformation module file and invokes a method from the ATL API [1] to parse this transformation. This parse results in an EObject [8] element, which comprises all information about the transformation. Then, our static analyzer extracts this information from the EObject element in order to instantiate the Java classes related to the ATL transformation being analyzed (input ATL transformation). These Java classes must be instantiated as Java objects to keep information about the input ATL transformation, given that they follow the same features and relationships defined by the ATL metamodel [10].

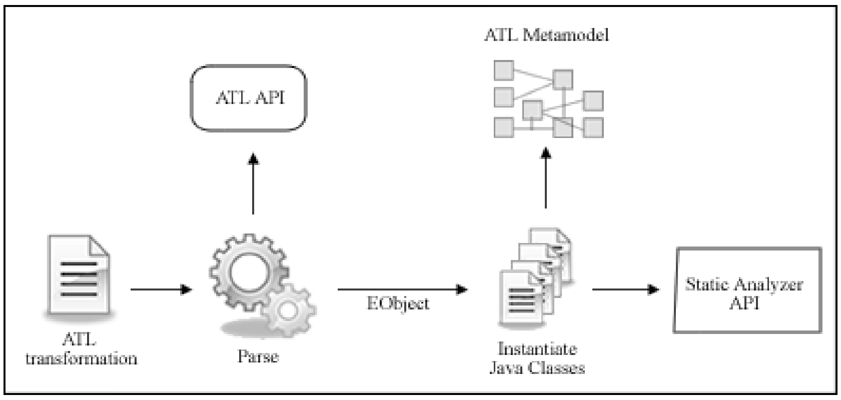

Fig. 1. Overview of our ATL static analyzer.

In this work, we have developed our static analyzer by inspecting the source code of an ATL transformation, given that we have invoked the `parse(InputStream in)` method available in the ATL API. This method receives as parameter an input stream concerning the source code of a transformation. Then, it parses the input stream into an EObject element, which can be exploited to retrieve information such as helpers, rules and so on. Notice that we have invoked the `parse(InputStream in)` method manually just to obtain the EObject element comprising information about the transformation, but this method is automatically invoked by the ATL Virtual Machine during the compilation process, then generating the bytecode (.asm).

The static analyzer was implemented according to the ATL 2.0 metamodel in order to define the same features and relationships existent in an ATL unit. An excerpt of the ATL metamodel is illustrated in Fig. 2. As we can observe, the ATL language enables developers to define three kinds of ATL `Unit`: a transformation `Module`, a `Query` or a `Library`. Fig. 2 shows that a `Module` can be composed by zero or more elements of the type *`ModuleElement`*. These elements are a `Helper` or a *`Rule`*. A `Helper` is the definition of methods or attributes within an ATL unit. On the other hand, a *`Rule`* specifies how to generate target model elements. It can be either declarative (`MatchedRule`) or imperative (`CalledRule`). A `LazyMatchedRule` is a kind of `MatchedRule`.

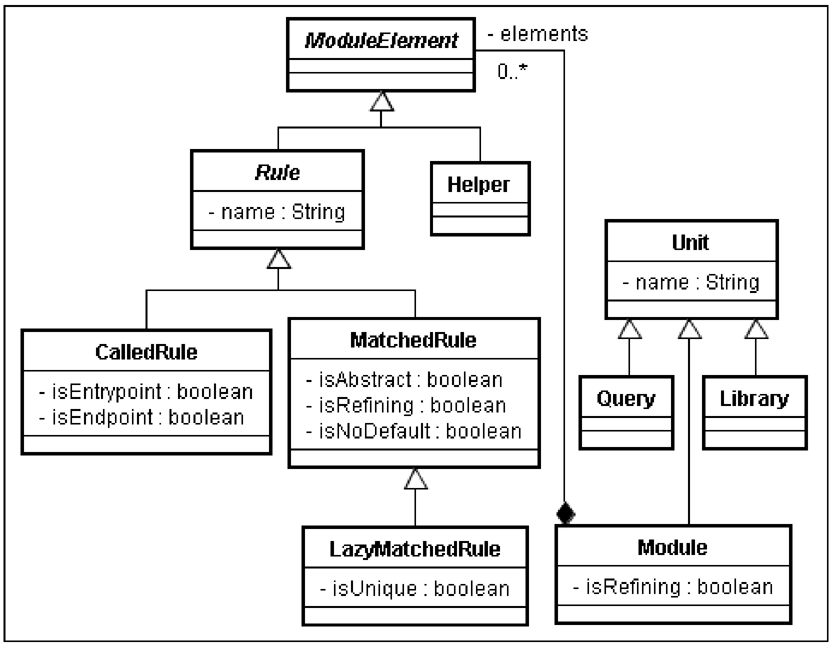

Fig. 2. Excerpt of the ATL metamodel.

In order to implement the ATL static analyzer according to the ATL metamodel, we had to create a Java class for each ATL metaclass of the ATL metamodel. For instance, according to the Fig. 2 previously illustrated, we had to create (i) an abstract Java class named *`Rule`* comprising a `name` property for the *`Rule`* ATL metaclass and (ii) two concrete Java classes named `MatchedRule` and `CalledRule` comprising their respective properties.

After the Java classes and their respective properties have been created, we instantiated them with information about the input ATL transformation. For obtaining these information we had to invoke the `parse(InputStream in)` method available at "`org.eclipse.m2m.atl.engine.parser.AtlParser`" class of the ATL API. This method parses a given input stream (ATL transformation) and returns an EObject element comprising information about the given input stream. In order to illustrate how the transformation was parsed, Source Code 2 shows an excerpt of the `loadMetaElements()` method available at `Transformation` Java class of the proposed ATL static analyzer. Lines 2-4 show some classes that must be imported. Lines 10-11 get an input stream from the path of the input ATL transformation. This path is informed by the user during the instantiation of the `Transformation` Java class. Then, at lines 12-13 the transformation is parsed and returned as an EObject.

Source Code 2: Excerpt of the loadMetaElements() method source code.

```
package org.atlanalyzer.main;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.m2m.atl.
    engine.parser.AtlParser;
...

public class Transformation {
    public void loadMetaElements() throws
    ATLCoreException, IOException {
       InputStream in = new BufferedInputStream(new
         FileInputStream(this.getPath().toString()));
       EObject modelObj =
         AtlParser.getDefault().parse(in);
       ...
       in.close();
    }
}
```

Since all information about the ATL transformation being analyzed has been instantiated as Java objects, the proposed static analyzer provides methods to manipulate a number of ATL transformation elements, such as rules, helpers, models and metamodels. For instance, we can obtain all matched rules of a transformation by invoking the method `getMatchedRules()`. Table 1 shows some of the methods provided by the static analyzer and a brief description of them.

Table 1. Some methods provided by the proposed ATL static analyzer.

Method signature	Description
public Rule getRule(String ruleName)	Obtains a specific rule (matched or called) by the informed ruleName;
public Set<MatchedRule> getMatchedRules()	Obtains a set of all matched rules of a transformation;
public Set<CalledRule> getCalledRules()	Obtains a set of all called rules of a transformation;
public Set<Helper> getHelpers()	Obtains a set of all helpers of a transformation;
public Helper getHelper(String helperName)	Obtains a specific helper according to a informed name;
public Set<Rule> getInvokerRules()	Obtains a set containing all rules that invoke a specific helper;
public Set<OclModel> getInputMetamodels()	Obtains a set containing the metamodel of each input model of a transformation;
public Set<OclModel> getOutputMetamodels()	Obtains a set containing the metamodel of each output model of a transformation;
public Set<MatchedRule> getChildren()	Obtains a set containing all the matched rules that extend a specific matched rule.

4 Examples of Application

This section presents two concrete examples with the aim at illustrating the application of the ATL static analyzer proposed in this work.

In the first example, the ATL static analyzer was applied to obtain a set of matched rules that extend a specific matched rule in an ATL transformation module. Source Code 3 shows an excerpt of the transformation available at the ATL User Guide [3], which is a KM3-copier: every model element from the source model is copied to the target model.

Source Code 3: Excerpt of the `Copy` ATL transformation example.

```
1  module Copy;
2  create OUT : MM from IN : MM;
3  rule CopyDataType extends CopyClassifier {
4      from s : MM!DataType
5      to t : MM!DataType
6  }
7  rule CopyEnumeration extends CopyClassifier {
8      from s : MM!Enumeration
9      to t : MM!Enumeration (literals <- s.literals)
10 }
11 abstract rule CopyModelElement extends
12 CopyLocatedElement {
13     from s : MM!ModelElement
14     to t : MM!ModelElement (name <- s.name)
15 }
16 abstract rule CopyLocatedElement {
17     from s : MM!LocatedElement
18      to t : MM!LocatedElement (location <- s.location)
19 }
20 rule CopyPackage extends CopyModelElement {
21     from s : MM!Package
22     to t : MM!Package (contents <- s.contents)
23 }
24 rule CopyClass extends CopyClassifier {
25     from s : MM!Class
26     to t : MM!Class (
27         isAbstract <- s.isAbstract,
28         supertypes <- s.supertypes,
29          structuralFeatures <- s.structuralFeatures,
30         operations <- s.operations)
31 }
32  rule CopyClassifier extends CopyModelElement {
33     from s : MM!Classifier
34     to t : MM!Classifier
35 }
```

Suppose that in the `Copy` ATL transformation module it is not necessary to copy classifiers anymore, thus we have to exclude the `CopyClassifier` matched rule (lines 32-35). However, there are some matched rules in the `Copy` transformation that extend this rule. Then, we have to know these matched rules before to exclude `CopyClassifier` in order to eliminate the dependencies.

The application of the proposed ATL static analyzer to the `Copy` ATL transformation is illustrated at Source Code 4, which shows a main Java method. First, we had to instantiate an ATL transformation specifying its path (lines 10-11). At line 12, the method `loadMetaElements()` was invoked to instantiate the Java classes with the information obtained from the transformation. Then, the module was obtained from the transformation (line 13). At lines 14-15, the matched rule that will be excluded was obtained from the module. Finally, all matched rules that extend `CopyClassifier` were obtained at lines 16-17.

Source Code 4: Example 1 of the application of the proposed static analyzer.

```
1   package org.atlanalyzer.main;
2   import org.atlanalyzer.metaelements.atl.Module;
3   import org.atlanalyzer.metaelements.
4       atl.MatchedRule;
5   ...
6
7   public class Main {
8       public static void main(String[] args)
9       throws ATLCoreException, IOException {
10          Transformation transf =
11              new Transformation("atl/copy.atl");
12          transf.loadMetaElements();
13          Module module = transf.getModule();
14          MatchedRule rule = (MatchedRule)module.
15              getRule("CopyClassifier");
16          Set<MatchedRule> children =
17            (Set<MatchedRule>)rule.getChildren();
18      }
19  }
```

In the second example, the ATL analyzer was applied to detect all rules (both matched rules and called rules) that invoke a specific helper. Then, if it is necessary to change or delete a specific helper defined in a transformation it is important to know the rules that reference this helper. Source Code 5 shows an excerpt of the Class to Relational transformation available at [6], which transforms a relational model from a class model.

Suppose that we have to change the return type of the `objectIdType` helper (lines 3-5). Then, all rules that invoke this helper must be known and analyzed before any change. Source Code 6 illustrates how to use the ATL static analyzer to obtain such information.

At Source Code 6, lines 1-12 are similar to the previous example: we have just added an import to the `Helper` java class and changed the ATL transformation file name. At lines 13-14 we obtain the helper which we have to make some change. Finally, we obtain all rules that invoke this helper at lines 15-16.

Even though we have illustrated small examples, it is important to emphasize that if the ATL static analyzer was applied to larger ATL transformations, developers would save development time given that looking for information about transformation elements requires much effort and sometimes it is unviable if the transformation is too long. In addition, this task is usually error-prone when manually accomplished.

Source Code 5: Excerpt of the `Class to Relational` ATL transformation example.

```
module Class2Relational;
create OUT : Relational from IN : Class;
helper def: objectIdType : Relational!Type =
Class!DataType.allInstances()->select(e | e.name =
    'Integer')->first();
rule Class2Table {
    from c : Class!Class
    to out : Relational!Table (
        name <- c.name,
        col <- Sequence {key}->union(c.attr->
            select(e | not e.multiValued)),
         key <- Set {key}
    ),
    key : Relational!Column (
        name <- 'objectId',
        type <- thisModule.objectIdType)
}
rule DataType2Type {
    from dt : Class!DataType
    to out : Relational!Type (name <- dt.name)
}
rule ClassAttribute2Column {
    from a : Class!Attribute (a.type.oclIsKindOf(
        Class!Class) and not a.multiValued)
    to foreignKey : Relational!Column (
        name <- a.name + 'Id',
        type <- thisModule.objectIdType)
}
```

Source Code 6: Example 2 of the application of the proposed static analyzer.

```
package org.atlanalyzer.main;
import org.atlanalyzer.metaelements.atl.Module;
import org.atlanalyzer.metaelements.
    atl.Helper; ...
public class Main {
    public static void main(String[] args)
    throws ATLCoreException, IOException {
        Transformation transf =
            new Transformation(
            "atl/classToRelational.atl");
        transf.loadMetaElements();
        Module module = transf.getModule();
        Helper helper =
         module.getHelper("objectIdType");
        Set<Rule> invokerRules =
         helper.getInvokerRules(); }}
```

5 Related Works

Nowadays, the software engineering field offers a number of tools and frameworks that propose the static analysis of code written in diverse programming languages, such as: (i) *FindBugs* [9] and *Checkstyle* [5]. Both are open source projects for Java code static analysis. *FindBugs* is a popular tool that examines a Java class or JAR files looking for potential problems by matching bytecodes against a list of bug patterns. On the other hand, *Checkstyle* is a tool that checks if Java source code adheres to a coding standard. Also, it provides some functionality to verify errors, such as the verification of duplicated code and bug patterns like double checked locking;(ii) *MOPS* [14] and *UNO* [11]. Both are projects for C code static analysis. *MOPS* is a free tool used for finding security bugs and for verifying conformance to rules of defensive programming. It focuses on identifying rules of safe programming practices and then verifying whether these safety properties are being obeyed within a program. On the other hand, *UNO* is a research/academic project that is able to detect the three most common types of software defects: use of uninitialized variable, nil-pointer references and out-of-bounds array indexing.

In addition, there exist some tools and frameworks that are multi-language, which means that they support the static analysis of code written in different languages, such as *Yasca* (Yet Another Source Code Analyzer) [20] and *Coverity Prevent* [7]. *Yasca* is an open source tool that provides support for source code written in Java, C, C++, HTML, JavaScript, ASP, ColdFusion, PHP, COBOL, .NET, and other languages. *Coverity Prevent* is a commercial tool that provides support for source code written in C, C++, C# and Java.

On the other hand, there is a lack of tools and frameworks to perform static analysis in the MDA context. [18] presents a number of proposals to facilitate the definition of Higher-Order Transformations (HOTs) in ATL, where each proposal is focused on a specific transformation class. These proposals are based on the analysis of a set of 42 freely available transformations in ATL. For instance, this work proposes a HOT library composed by those helpers that were recurrently used within the set of the 42 analyzed HOTs. Despite of [18] offers, among other proposals, an API with several helpers, its focus is not on allowing the navigation by ATL elements in order to obtain information about all ATL elements of a model transformation before its execution.

For the best of our knowledge, there exists just one work [19] that proposes a static source code analysis framework to detect some common errors in model transformation programs specified particularly in VIATRA2 VTCL, which is a Graph Transformation Language that represents transformations as abstract state machines. Transformation programs specified in this language are internally stored as an EMF model. As opposite to this work, the static analyzer that we propose is focused on ATL transformations. In addition, it has the purpose of aiding developers during several development tasks of MDA-based projects, such as maintenance, while [19] is focused on detection of common errors in transformations specified just in VIATRA2 VTCL. Also, our approach provides an API with methods from which developers can obtain information about the whole ATL transformation.

Therefore, nowadays there is no work focused on static analysis for ATL transformations. It means that it is an incipient and potential research field, in which our work is very promising given that it is the only one that offers a static analyzer that provides an API to extend and handle diverse elements from ATL transformations.

6 Conclusions

In this work, we have proposed a static analyzer for M2M transformations specified in ATL language. It provides an API containing a number of methods that can be used to manipulate elements of an ATL transformation module, such as rules, helpers, models and metamodels. The ATL static analyzer has been implemented in Java and it is available for the ATL community under a GPL license. For illustrating how it can be applied to obtain information from ATL transformations we have shown two examples.

The main objective of the proposed static analyzer is to inspect ATL transformations in order to help developers to automatically find out information about any element enclosed by these transformations. Thus, for instance, when a developer needs to change any element in a transformation, it just has to invoke the appropriate methods of the static analyzer API to obtain information about dependencies and relations of this element. As we can note, the proposed static analyzer is very helpful along the development process of ATL transformations, such as during maintenance and debugging tasks. It is important to emphasize that, when the transformation is very large, it is unviable for developers to manually look for dependencies and relations of an element in this transformation. By using the ATL static analyzer, they can accomplish this task with reduced effort.

As an ongoing work, we are developing a change impact analysis technique that reuses our ATL static analyzer in order to provide a precise impact analysis on ATL transformations before any change has been done.

As a future work, we intend to extend the ATL static analyzer to allow users to detect bugs before running their transformations as well as to optimize them. Also, we intend to extend the ATL static analyzer in order to provide a user interface. It would be beneficial to offer the static analysis not only as an API but also as an integration with the current Eclipse based UI of ATL, which would be easier manipulated by the beginners.

Acknowledgments. We would like to acknowledge the CAPES for the financial support as well as Dalton Guerrero and João Brunet for their valuable discussions.

References

1. ATLAS Transformation Language API, http://dev.eclipse.org/viewcvs/viewvc.cgi/org.eclipse.m2m/org.eclipse.m2m.atl/?root=Modeling_Project
2. ATL Static Analyzer: A Static Analyzer Tool for ATL Transformations, http://code.google.com/p/atl-static-analyzer

3. ATL User Guide - ATLAS Group (INRIA & LINA), http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language
4. Balogh, A., Németh, A., Schmidt, A., Rath, I., Vágó, D., Varró, D., Pataricza, A.: The VIATRA2 Model Transformation Framework. In: European Conf. on Model Driven Architecture (ECMDA'05) (2005).
5. Checkstyle: A Static Analyzer Tool for Java Code, http://checkstyle.sourceforge.net
6. Class to Relational ATL Transformation Example, http://www.eclipse.org/m2m/atl/atlTransformations/#Class2Relational
7. Coverity Prevent: A Static Analyzer Tool for C/C++, C# and Java Languages, http://coverity.com/products/static-analysis.html
8. EObject API, http://download.eclipse.org/modeling/emf/emf/javadoc/2.6.0/org/eclipse/emf/ecore/EObject.html
9. FindBugs: A Static Analyzer Tool for Java Code, http://findbugs.sourceforge.net
10. Frédéric Jouault, ATL Metamodel, http://www.emn.fr/z-info/atlanmod/index.php/Atlantic#ATL_2.0
11. Holzmann, G.: UNO: Static Source Code Checking for User-defined Properties. Bell Labs Technical Report, Bell Laboratories, Murray Hill, NJ (2002).
12. Kleppe, A., Warmer, J., Bast, W.: MDA Explained. The Model-Driven Architecture: Practice and Promise, Addison-Wesley Profession (2003).
13. Logozzo, F., Fähndrich, M.: On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis. In: Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction. Budapest, Hungary (2008).
14. MOdelchecking Programs for Security properties (MOPS): A Static Analyzer Tool for C Code, http://www.cs.berkeley.edu/~daw/mops
15. Nielson, F., Riis, H., Hankin, C.: Principles of Program Analysis. Springer (2010).
16. Object Management Group (OMG), http://www.omg.org
17. Object Management Group (OMG): Query/View/Transformation Specification, document number 2008-04-03, version 1.0, April (2008), http://www.omg.org/spec/QVT/1.0/PDF
18. Tisi, M., Cabot, J., Jouault, F.: Improving Higher-Order Transformations Support in ATL. In: Proceedings of the International Conference on Model Transformation, ICMT2010, 2010.
19. Ujhelyi, Z., Horváth, A., Varró, D.: A Generic Static Analysis Framework for Model Transformation Programs. Technical Report TUB-TR-09-EE19, Budapest University of Technology and Economics (2009).
20. Yet Another Source Code Analyzer (Yasca): A Static Analyzer Tool for Several Languages, http://www.scovetta.com/yasca.html