Analyzing Flowgraphs with ATL

Valerio Cosentino Massimo Tisi
Fabian Büttner
AtlanMod, INRIA & École des Mines de Nantes, France
{valerio.cosentino, massimo.tisi, fabian.büttner}@inria.fr

This paper presents a solution to the Flowgraphs case study for the Transformation Tool Contest 2013 (TTC 2013). Starting from Java source code, we execute a chain of model transformations to derive a simplified model of the program, its control flow graph and its data flow graph. Finally we develop a model transformation that validates the program flow by comparing it with a set of flow specifications written in a domain specific language. The proposed solution has been implemented using ATL.

1 Introduction

This paper presents an ATL-based solution to the Flowgraph Case Study[2] for the Transformation Tool Contest 2013 (TTC 2013)[1]. The main task of the case study is deriving the program dependence graph (PDG) of the given source code. This graph contains both control and data flow information and is obtained through a sequence of steps: 1) creation of a simplified model for the Java program; 2) generation of the program control flow graph; 3) addition of data flow dependencies to create the PDG. A final additional task is 4) validation of the resulting PDG against a set of specifications written in the provided DSL.

The solution[2] is implemented using an ATL transformation chain. We address all the tasks of the case study by relying exclusively on the ATL declarative language, with the exception of text-to-model injectors and global orchestration. The case study shows the flexibility of ATL in handling a wide range of tasks: classical model-to-model transformation and model-to-text transformation in task 1, in-place refinement in task 2, a complex algorithm in task 3, model validation in task 4. It is also intended as a full-range example for new ATL developers.

The ATL Transformation Language (ATL) [3] is a model transformation language and tool available from the Eclipse modeling project [3]. ATL is a declarative language allowing the specification of transformation rules, that are matched over the source model to create elements in the target model. Expressions are written using the Object Constraint Language (OCL [4]). ATL contains also an imperative part allowing to handle cases whose declarative expressions would be too complex. The solution we propose in this paper makes use only of the declarative part of the language.

ATL allows the developer to decorate the input metamodel with derived attributes and operations on model elements, named *Helpers* and grouped into reusable *Libraries*. Finally the developed transformation can be applied in *normal mode*, where target models are built from scratch, or in *refining mode*, where the input model is modified in-place.

[1] http://tinyurl.com/TTC2013-HomePage
[2] The full solution is available on the SHARE server of the contest http://tinyurl.com/ATL-solution
[3] http://www.eclipse.org/m2m/atl/
[4] http://www.omg.org/spec/OCL/2.3.1/

Pieter Van Gorp, Louis M. Rose, Christian Krause (Eds.):
Sixth Transformation Tool Contest (TTC 2013)
EPTCS 135, 2013, pp. 32–36, doi:10.4204/EPTCS.135.4

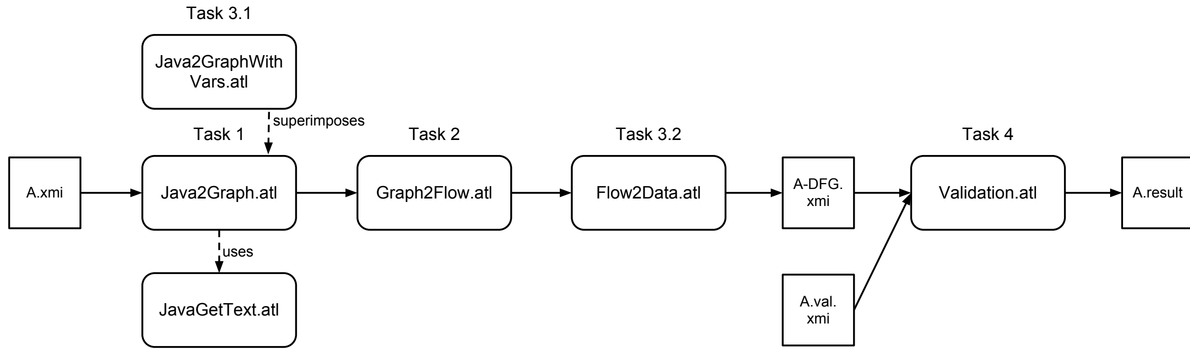

Figure 1: The chain of ATL transformations (intermediate models are omitted).

This paper is structured as follows: Section 2 illustrates our solution and finally Section 3 concludes the paper.

2 An ATL solution

The ATL solution for the Flowgraph Case Study is composed by a chain of ATL transformations orchestrated by an ANT file. The structure of the chain is illustrated in Fig. 1 and detailed in the following.

2.1 Task 1: Structure Graph

The main transformation in Task 1 is Java2Graph.atl that implements a simple mapping between elements in the JaMoPP and FlowGraph metamodels. The mapping is illustrated in Table 1. Each line of the table is encoded as a simple ATL rule. For instance in Listing 1 we show the rule that translates while loops. Rules define model elements to match in the source model (*WhileLoop*), model elements to generate in the target (*Loop*), and values to assign to target properties. The rule in Listing 1 states that the *expr* and *body* references have to be filled with the result of the translation of the *condition* and *statement* of the matched element *s*. In each of the rules of Java2Graph.atl a *txt* attribute is filled with the concrete textual syntax of the element, calculated by calling a *getText* attribute helper. The *getText* helpers are defined in an ATL library, JavaGetText.atl, that is referenced by Java2Graph.atl.

Listing 1: A rule from Java2Graph.atl.

```
rule WhileLoop2Loop {
  from
    s : JAVA!WhileLoop
  to
    t : GRP!Loop (
      expr <- s.condition,
      body <- s.statement,
      txt <- s.getText
    )
}
```

The library JavaGetText.atl contains a set of *getText* attribute helpers, one for each metamodel element, that implement the model-to-text transformation task of the case study. In Listing 2 we show an excerpt of JavaGetText.atl, to illustrate its structure. Each helper is an OCL expression on the source

Java Entities	**Flow Entities**
Methods	
ClassMethod	Method, Exit
Statements	
Block	Block
Condition	If
Return	Return
WhileLoop	Loop
Jump	JumpStmt
JumpLabel	Label
Continue	Continue
Break	Break
Other statements	SimpleStmt
Expressions	
EqualityExpression	Expr
RelationExpression	Expr

Table 1: Java2Graph mapping

model and the helpers call each other to construct complex concrete syntaxes. The excerpt in Listing 2 contains the necessary code to compute the textual syntax of an assignment of the form *a=1;*.

Listing 2: The model-to-text transformation JavaGetText.atl (excerpt).

```
helper context JAVA!ExpressionStatement def : getText : String =
  self.expression.getText + ';';

helper context JAVA!AssignmentExpression def : getText : String =
  self.child.getText + ' ' + self.assignmentOperator.getText + ' '
    + self.value.getText;

helper context JAVA!LocalVariable def : getText : String =
  self.name;

helper context JAVA!DecimalIntegerLiteral def : getText : String =
  self.decimalValue;
```

2.2 Task 2: Control Flow Graph

Task 2 is implemented in the transformation Graph2Flow.atl. The transformation uses the *refining mode* of ATL, allowing the developer to specify only the refinement part. In this case a set of rules adds the *cfNext* reference that encodes control flow edges. All these rules have the structure shown in Listing 3: elements are matched and a *cfNext* reference is added by calling a *getNext* OCL helper. The logic for deriving control flow edges, detailed in the case study description, is encoded in the set of OCL *getNext* helpers.

Listing 3: A rule from Graph2Flow.atl

```
rule SimpleStmt {
  from
    s : GRP!SimpleStmt
  to
```

```
    t : GRP!SimpleStmt (
      cfNext <- s.getNext
    )
}
```

2.3 Task 3: Data Flow Graph

2.3.1 Subtask 3.1

The construction of the data flow links requires to keep information, through the whole transformation chain, about variable uses and definitions. For this reason, the transformation in Task 1 has to be extended to avoid the loss of this information. We use the *superimposition* mechanism to extend the Java2Graph.atl transformation in Task 1 with a set of additional rules and helpers. The rules of the superimposed transformation, Java2GraphWithVars.atl are executed together with the rules of Java2Graph.atl by the ATL virtual machine. Rules with the same name are overridden by the superimposed transformation (but this case does not apply to our scenario). Listing 4 contains the only two rules of Java2GraphWithVars.atl, that respectively create variables and parameters. A set of OCL helpers are called by *getDefiners* and *getUsers* to fill the definition and usage references. The set of helpers find uses and definitions by analyzing the position of the variable reference in the program tree. For instance a variable definition is detected whenever the variable reference *isInLeftInAssignment* or *isInUnaryModificationExpression*.

Listing 4: Rules from Java2GraphWithVars.atl

```
rule LocalVariableStatement2Var {
  from
    s : JAVA!LocalVariable
  to
    t : GRP!Var (
      txt <- s.getText,
      definers <- Sequence{s.getLocalVariableStatement}->
        union(s.getDefiners),
      users <- s.getUsers
    )
}

rule OrdinaryParameter2Var {
  from
    s : JAVA!OrdinaryParameter
  to
    t : GRP!Param (
      txt <- s.getText,
      definers <- Sequence{s.getMethod}->union(s.getDefiners),
      users <- s.getUsers
    )
}
```

2.3.2 Subtask 3.2

For the generation of data-flow links we implemented a variation of the algorithm in [1] as a set of OCL helpers. The resulting iterative algorithm calculates for each flow instruction the set of definitions that the program needs when arriving to that point. It proceeds backwards by starting from variable uses, analyzing the successors of each flow instruction and propagating back the need for definitions.

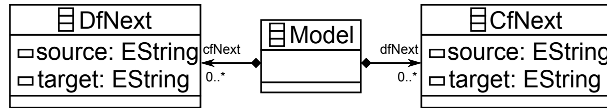

Figure 2: Metamodel for the Validation DSL.

2.4 Task 4: Validation

We implemented the validation task of the case study by an ATL model-to-text transformation (Validation.atl) that takes two models as input: the program dependence model generated by Task 3 and a user-provided specification model (Fig. 2). A set of OCL helpers iterate on the specifications and check that the correspondent dependency exists in the model. Vice versa, they also iterate on the dependency models to check that all the dependencies belong to the specification file. A textual list of missing links and false links is generated in output.

3 Conclusion

The case study shows the applicability of ATL to complex transformation scenarios in program analysis. Table 2 presents size information on the implemented transformations[5]. The transformations, beside intrinsic algorithmic complexity, look fairly readable.

Transformation	Task	LOC	Rules	Helpers
JavaGetText	1	214	0	60
Java2Graph	1	133	12	0
Java2GraphWithVars	3.1	183	2	19
Graph2Flow	2	324	7	28
Flow2Data	3.2	92	2	6
Validation	4	59	0	9

Table 2: Transformation size

The problem can be modularized in a transformation network and concisely represented by using exclusively declarative transformation rules and helpers. All the phases are handled by the same transformation language: model-to-model, model refinement, model-to-text, validation. The case study represents an interesting illustration of the ATL application space.

References

[1] Alfred V. Aho, Ravi Sethi & Jeffrey D. Ullman (1986): *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley.

[2] Tassilo Horn (2013): *The TTC 2013 Flowgraphs Case*. In: *Sixth Transformation Tool Contest (TTC 2013), EPTCS* this volume.

[3] Frédéric Jouault & Ivan Kurtev (2005): *Transforming Models with ATL*. In: *MoDELS Satellite Events*, pp. 128–138, doi:10.1007/11663430_14.

[5] Performance evaluation of the proposed ATL solution can be found at `http://docatlanmod.emn.fr/TTC/Result.pdf`