An AmmA/ATL Solution for the GraBaTs 2009 Reverse Engineering Case Study

Frédéric Jouault and Jean-Sébastien Sottet

INRIA Centre Rennes Bretagne Atlantique,
Ecole des Mines de Nantes
4, rue Alfred Kastler, 44307, France
{frederic.jouault,jean-sebastien.sottet}@inria.fr
http://www.emn.fr/x-info/atlanmod/

Abstract. This paper presents a solution to the reverse engineering case study of GraBaTs 2009 implemented using the AtlanMod Transformation Language (ATL), and the AtlanMod Model Management Architecture (AmmA). Two scalability approaches are presented: a classical one, as well as an optimization for multiple queries on the same model. The task showing the genericity of the transformation tool is also solved.

Key words: Program Comprehension, Program Query, Transformation Scalability

1 Introduction

The case study proposed in [7] consists of two program comprehension (or reverse engineering) tasks. The first one exercises the scalability of transformation tools. It has been solved with AmmA/ATL (described in Section 2). The corresponding results are presented in Section 3. The second task exercises the genericity of transformation tools. Section 4 presents its solution using AmmA/ATL.

2 About AmmA/ATL

The AtlanMod Transformation Language [3,4] (ATL) is a general-purpose model transformation language and tool [1] available from Eclipse.org. ATL is declarative, with rules matched over source models that create elements in target models. An imperative part is also present to handle cases otherwise too difficult to handle declaratively.

The AMMA [5] (AtlanMod Model Management Architecture) platform has been developed around ATL in order to support tasks in various domains such as: model-based Domain-Specific Languages implementation, reverse engineering. A new version of AMMA, called AmmA [2], introduced a number of new features including a new modeling framework called KMF (Kernel Modeling Framework). This framework has an implementation with a lower memory footprint than EMF (Eclipse Modeling Framework) in dynamic mode (i.e., when no code is generated from metamodels).

3 Task 1

The main challenge in this task is the size of source models. The overall performance of a solution strongly depends on the way source models are actually loaded. Section 3.1 discusses this issue. The solution presented here implements several distinct approaches described in Section 3.2. The two most significant approaches have been timed, and the results are given in Section 3.3.

3.1 Loading

The existing ATL engines control source model navigation: they expect the full model to be navigable in all directions at all time. The simplest way to satisfy this requirement is to load whole models in memory. This is what the currently available ATL implementations do. However, this approach has two main issues:

- **Loading time.** Because whole models need to be loaded, the loading phase takes significantly more time than the transformation phase for large models.
- **Memory footprint.** Moreover, enough memory must be available to represent whole models. Available memory depends on the operating system and other factors (see [7]).

The second issue is more critical than the first because if memory requirements exceed available memory, then the transformation cannot be executed at all. There are two main possibilities to reduce memory footprint:

- **Parser-driven loading.** In this situation, the parser sends small chunks of the source model to the transformation engine. This typically requires to parse the whole source file, but only a small window around the cursor need to be kept in memory. Model elements appearing earlier in the file are forgotten.
- **Lazy loading.** With such an approach, only model elements that have recently been navigated need to be kept into memory. A cache may keep more elements for increased performance. Elements that are not navigated by the transformations may not need to be loaded.

The parser-driven approach is incompatible with the existing ATL engines (see above) because navigation control is moved from engine to loader (i.e., the engine cannot navigate source models in all directions at all times). Therefore, we experimented with lazy loading because it can be made to work with existing ATL engines.

3.2 Implementation

The previous section described the two cases that have been experimented with AmmA/ATL:

- **Loading whole models**, which is performed by what is called a static injector[1] below. This name indicates that the injector is executed once (per model), and has completely finished its job when it returns to its caller.
- **Partially loading models**, which is what dynamic injectors do. Such an injector continuously loads parts of a model according to navigation requests coming from the transformation engine.

Lazy loading of models from XMI files requires building an index because no random access is possible in XML documents. The index requires a significant amount of memory. Moreover, the whole XML document must be parsed to build it. The advantage is that less memory is required because only some model elements are in memory at a given time. However, the gain is not as much as expected because of the two reasons mentioned above. We have experimented with this solution which is provided in the solution as the DynamicXMILoader.

Because of this issue, we designed and implemented a binary format for models called BMS (Binary Model Storage). Model elements stored in this file may be randomly accessed. Additionally, it is possible to speed up transformations significantly by storing indexes of elements by their meta-elements. This is equivalent to pre-computing the OCL *allInstances()* operation. Without these indexes, the whole model needs to be traversed. Once a model has been converted into a BMS file (e.g., from XMI), it can be processed faster than the XMI file can be. However, this improvement is achieved at the cost of a preprocessing step: the model needs to be loaded using another injector once, an serialized as a BMS file. Therefore, depending on whether a single query is to be performed or several, it may be advantageous or not to use BMS over a static XMI injector.

Overall, we have experimented with the following injectors:

1. **Static XMI.** This injector parses an XMI document using an XML push parser [6]: the SAX parser from the Java API[2], and creates whole models into memory.
2. **Static Pull XMI.** This injector parses an XMI document using an XML pull parser [6]: the StaX parser from the Java API[3], and creates whole models into memory.
3. **Dynamic XMI.** This injector lazily loads an XMI document, but performs a first complete traversal (without creating the model into memory) to build an index.
4. **Static BMS.** This injector parses a BMS file, and creates whole models into memory.
5. **Dynamic BMS.** This injector lazily loads a BMS file, and only traverses the part of the files that correspond to elements that the transformation engine need to navigate.

[1] An injector parses a file (e.g., in XMI format) into a model.

[2] See `http://java.sun.com/javase/6/docs/api/javax/xml/parsers/SAXParserFactory.html`.

[3] See `http://java.sun.com/javase/6/docs/api/javax/xml/stream/XMLInputFactory.html`.

We only benchmarked 2 for XMI loading because it is faster than 1 and 3, and because although it uses more memory than 3 it can work even on set4. As for BMS, we only benchmarked 5 because it uses less memory and is faster than 4. However, all injectors are available in the solution.

The transformation is a simple filtering query, which yields the classes that match the condition defined in the case study (see Listing. 1.1. The transformation itself can be decomposed into successive filtering such as selecting only static (e.isStatic), and public (e.isPublic) methods that have a return type. This is performed by the helper at lines 1-6 and by the initial part of the rule filter at lines 11-14. Finally, the transformation selects methods that return type name equal to class name (last part of rule filter at line 15-20).

Listing 1.1. Query Q1

```
1  helper context JDT!TypeDeclaration def: methodsWithReturnType : Sequence
       ↪(JDT!MethodDeclaration) =
2    self.bodyDeclarations->select(e |
3      e.oclIsKindOf(JDT!MethodDeclaration)
4    )->select(e |
5      not e.returnType.oclIsUndefined()
6    );
7
8  rule Q1 {
9    from
10     s : JDT!TypeDeclaration (
11       s.methodsWithReturnType->select(e |
12         e.isPublic
13       )->select(e |
14         e.isStatic
15       )->select(e |
16         e.returnType.oclIsKindOf(JDT!QualifiedType) or
17         e.returnType.oclIsKindOf(JDT!SimpleType)
18       )->exists(e |
19         e.returnType.name.fullyQualifiedName = s.name.fullyQualifiedName
20       )
21     )
22   to
23     v_class : VIEW!Class (
24       name <- s.name.fullyQualifiedName
25     )
26 }
```


3.3 Benchmarks

In this section, we present performance figures that have been measured on a Dell Latitude D630 with an Intel(R) Core(TM)2 CPU U7600 @ 1.20GHz, with 2GB of physical memory, and running the Windows XP 32 bits operating system. The Java Virtual Machine (JVM) version 1.6.0 13-b03 has been restarted for each measure. Reusing the same JVM for several measures would make all but the first executions slightly faster. The reason seems to be that class loading (and possibly other initializations) happens during the first execution.

The execution times have been measured with two different injectors: Static Pull XMI, and Dynamic BMS (see Section 3.2). In all cases, load time, transformation time, as well as total time are given. Total time is usually slightly greater than the sum of load and transformation times. This is mostly caused by transformation setup that occurs between load and transformation.

Using the Static Pull XMI Injector Table 1 presents the results obtained with the Static Pull XMI Injector described in Section 3.2. The first column identifies the model (from the test-cases) used in a given row. Loading time (in seconds) is given in the second column. Transformation time (in seconds) is given in the second column.

Only one execution has been performed to create this table. Multiple executions may lead to slightly smaller figures because of operating system cache. However, the JVM uses almost all available memory (the 700MB not used for the JVM heap are almost entirely used by the operating system and services). Therefore, there is not much space available for the cache.

These measures have been performed with 1300MB of maximum an minimum JVM memory heap[4]. Setting the minimum heap size to the same value as the maximum avoids heap resizing, and some garbage collections. The smaller sets (e.g., set0, set1) can work with less heap memory.

	Loading	**Transforming**	**Total**
set0	1.359	0.047	1.437
set1	3.844	0.078	3.969
set2	56.422	2.484	58.969
set3	122.547	6.328	129.172
set4	122.078	6.718	129.437

Table 1. Execution times with the Static Pull XMI injector (all times are given in seconds) using 1300MB of minimum and maximum memory heap.

Using the Dynamic BMS Injector Table 2 presents the results obtained with the Dynamic BMS Injector described in Section 3.2. The first column identifies the model (from the test-cases) used in a given row. BMS file computation time is given in the second (XMI loading time) and third (BMS serialization) columns (in seconds). Columns four, five, and six give the BMS loading, transformation, and total times (in seconds). Finally, the two last columns (seven and eight) respectively give the main BMS file size, and index size (stored as one file per metamodel in a separate folder). These sizes have been computed using the `ls -lh`, and `du -sh` commands respectively.

When the transformations are executed multiple times, the first execution is slower (especially for large files). The operating system cache is probably the cause. Whereas it had only a low impact with the Static Pull XMI Injector (see above), it helps significantly in this case. The reason seems to be that much less memory is used by the JVM, and therefore more is left for the operating system

[4] Using the -Xmx1300m -Xms1300m JVM options.

to use as a cache. Therefore, the benchmarks have been executed five times, and the third value has been taken for the table[5].

The Dynamic BMS Injector performs correctly with only 32MB of heap memory. However, it also requires the size of the BMS file as addressing space because it maps the file into memory. The injector performs slightly faster with 64MB of heap memory (minimum and maximum), which is the figure that we used to create Table 2.

BMS file computation has been performed with 1300MB of heap (minimum and maximum) plus 260MB (currently hardcoded because the final size is not known in advance) of addressing space. Only one execution of BMS file computation has been performed. The figure obtained for set4 XMI loading (i.e., second column, last row) is significantly off: it should be close to the time reported in Table 1 for the same task (i.e., second column, last row). It seems that an external event occured, which impacted execution time. We did not have the time to investigate this detail more precisely yet.

	Computing		**Loading**	**Transforming**	**Total**	**File size**	
	XMI loading	***Serializing***				***Main***	***Index***
set0	1.265	4.453	0.078	0.703	0.860	4.9M	907K
set1	4.172	10.125	0.062	0.765	0.906	15M	2.4M
set2	49.360	94.031	0.094	3.796	3.969	90M	27M
set3	121.250	230.375	0.062	8.687	8.859	195M	60M
set4	282.765	289.609	0.063	9.360	9.547	211M	65M

Table 2. Execution times and file sizes with the Dynamic BMS Injector (all times are given in seconds, file sizes in bytes) using 64MB of minimum and maximum memory heap. Computation of BMS files has been performed with the Static Pull XMI Injector using 1300MB of minimum and maximum memory heap.

Discussion The results presented above show that transformation time is below ten seconds in all cases. Loading time represents most of the time with the Static Pull XMI Injector (up to about two minutes), whereas it is negligible with the Dynamic BMS Injector. The fact that the Dynamic BMS Injector continues loading the model during the transformation increases transformation time by about forty percent (40%).

About four times less addressing space (i.e., heap memory plus mapped file size) is required with Dynamic BMS than with Static Pull XMI. The main reason why BMS requires less memory than a model loaded in memory is that the binary structures used by BMS have a lower overhead than Java objects. The

[5] Mean and standard deviation could have been more precise information, but would have required more executions.

main inconvenient is that the Dynamic BMS projector is (currently) read-only (i.e., dynamically loaded models can only be read, not modified).

The cost of BMS file computation is less than four times the cost of a single XMI loading (if we exclude the singularity of set4 in Table 2. This shows that using the Dynamic BMS Injector becomes interesting if a single large models is queried four or more times, which is likely to happen in program comprehension scenarios.

We have not been able to load the larger models of the test-cases with EMF in dynamic mode on the machine used for the benchmarks. The reason is that the modeling framework provided by AmmA (i.e., KMF) has a lower memory footprint than dynamic EMF. We have not attempted to load the models with Java code generated by EMF from the Java metamodel.

4 Task 2

This tasks consists in computing a Control Flow Graph (CFG) and a Program Dependence Graph (PDG) for a small Java program. There is no significant problem to implement this with ATL.

The provided solution additionally generates a GraphML representation of the CFG (see Fig. 1) that may be loaded in the Prefuse graph visualization tool[6].

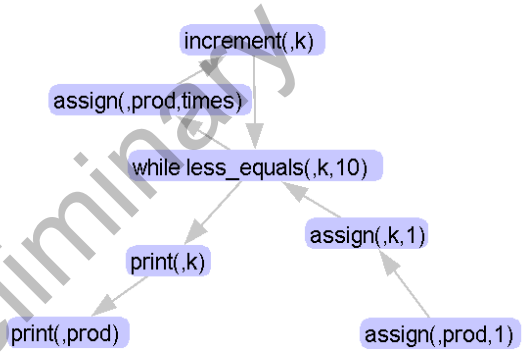

Fig. 1. Visual representation of the CFG using Prefuse

4.1 Abstract Syntax Tree to Control Flow Graph

The main issue in JDT Abstract Syntax Tree (AST) to CFG computation is the structural difference between metamodels. AST is a tree of simple lists (children

[6] See `http://prefuse.org/`.

of a node are not linked to each other), whereas CFG, as a directed graph, links neighboring statements to each other. The problem at hand is a slightly more complex version of the simple list to linked list problem. From the tree decomposition we build a map of statements <current, next>. This map gives the following statement of every statement. See Fig. 2 for transformation principle from tree structure to directed graph structure. The discontinued line on the left-hand side corresponds to the *next* helper. In the original model (without the discontinued line) there are no edges between the Si nodes.

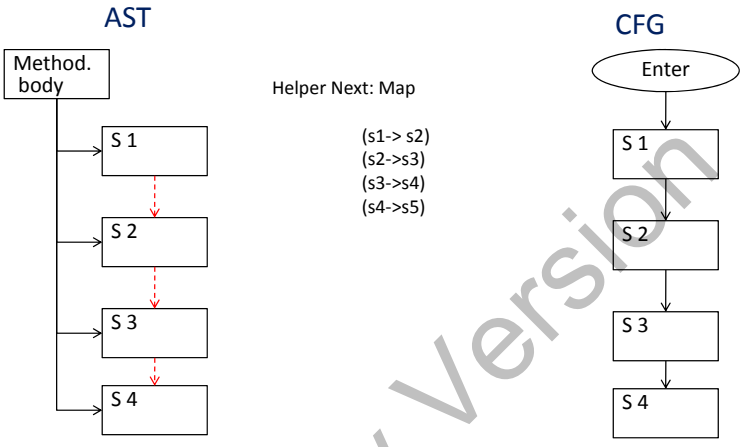

Fig. 2. AST to CFG transformation principle for non-branching statements

The conditional pattern for a simple if (i.e., containing a single statement, not a block) is described in Listing 1.2. There is no particular difficulty for computing such a view. The filter (line 3) selects simple if statements (i.e., when body, which represents a block, is undefined). The second part (lines 6-7) sets the statement if true and statement if false.

Listing 1.2. CFGNodetoPDGNode

```
1 rule SimpleIf extends Statement2Node {
2   from
3     se : JDT!IfStatement(se.body.oclIsUndefined())
4   to
5     te : CFG!ConditionalNode (
6          trueConditionNode <- se.thenStatement,
7          falseConditionNode <- se.elseStatement
8     )
9 }
```

The control flow of non-branching statements is established thanks to the *next* map. But we also face two kinds of branching statements: conditional nodes (e.g., if, switch) and iterative nodes (e.g., while, do-loop). Fig. 3 shows two patterns for branching nodes. The easiest way is to treat these two kinds of nodes independently to ease the task of further transformations (e.g., computation of

PDG). Loop recognition may be a complex task that we avoid with the creation of specific iterative nodes.

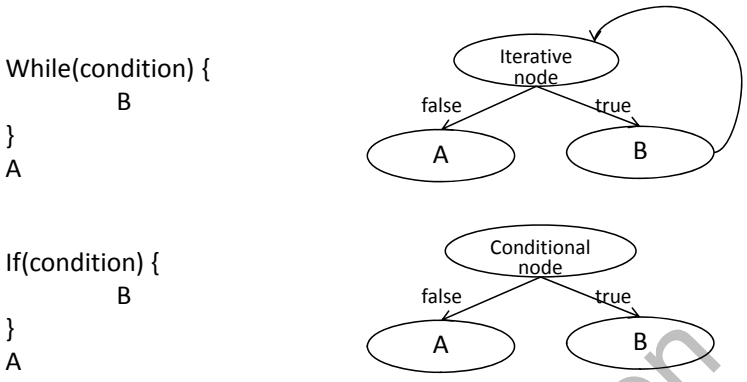

Fig. 3. CFG iterative and conditional patterns

We put Java Expression in the CFG directed graph. The CFG metamodel provides a prefixed notation for expressions. For example, expression $i = 1$ is written as: $= i, 1$. We recursively define inner expressions, for example: $prod = prod{*}k$ is written as: $= prod, {*}prod, k$. The next section presents the computation of PDG from CFG.

4.2 Control Flow Graph to Program Dependence Graph

A Program Dependence Graph is composed of control flow and data dependence relationships.

Control Flow Computation There is no significant difficulty for computing PDG control flow from CFG graph. The decomposition of control flow in PDG is hierarchical, every branching node (i.e., with a condition) represents a hierarchical decomposition (See Fig. 4)

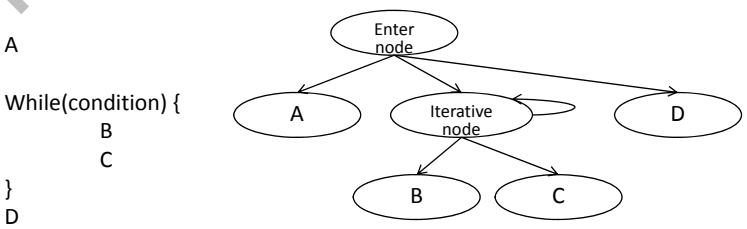

Fig. 4. PDG control flow Hierarchical decomposition pattern

Data Dependence Computation In order to understand Data Dependence computation, we first define a function $Succ(a)$ (where a is a CFG node) that returns the set of nodes that are after a in the CFG flow. Let us also consider a.mVar and a.uVar which return respectively variables modified (e.g., $x = 1$) and variables used (e.g., print(x)) in the expression supported by node a. A node x is said data dependent of node y if:

- x ∈ Succ(y)
- x.uVar ∈ y.mVar

In ATL this is translated into three helpers for:

- computing the Succ function on standard and conditional nodes. This helper calls, in appropriate cases, the second helper.
- computing the Succ function on iterative nodes (simplification that avoids complexity in successor computation).
- computing equality of qualified names on both used variable of successors expression, and modified variable by the current expression.

Theses helpers are called sequentially to filter CFG nodes for building the data dependence relationship of a PDG node. In Listing 1.3, we first select from all CFG nodes (by using .allInstancesFrom('IN') where IN represent the input model) nodes that are after the current matched node (e.isAfter(cfgnode)). Then, from this set of nodes, we select only those that are data dependent from the current node (e.isDataDependent(cfgnode)).

Listing 1.3. CFGNodetoPDGNode

```
1  rule  CFGNodetoPDGNode {
2
3    from cfgnode : CFG!Node
4
5    to pdgnode : PDG!ExpressionNode(
6                      expression<-cfgnode.expression,
7                      dataDependenceNodes<-CFG!AbstractNode.
                         ↪allInstancesFrom('IN')
8                      ->select(e| e.isAfter(cfgnode))
9                      ->select(e| e.isDataDependent(cfgnode)))
10 }
```


5 Conclusion

This paper has presented solutions to both tasks of the reverse engineering case study of GraBaTs 2009. ATL has been shown to be scalable: able to handle all sets including the larger ones on a 32 bits machine with only about 1300MB of available heap space. Advanced model loading techniques have been briefly presented that enable fast querying of large models at the cost of a preprocessing phase. Moreover, ATL has been shown to be able to solve more complex transformation problems such as CFG and PDG computation, as well as rendering to an external display surface (by targeting a visualization tool).

Acknowledgments. The authors would like to thank all the AtlanMod team, and more particularly Guillaume Doux, and Jean-Bézivin. This work has been partially supported by the Lambda project.

References

1. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl: a model transformation tool. *Science of Computer Programming*, 72(3, Special Issue on Second issue of experimental software and toolkits (EST)):31–39, 2008.
2. F. Jouault, J. Bézivin, and M. Barbero. Towards an advanced model driven engineering toolbox. *Innovations in Systems and Software Engineering*, 5(1):5–12, 2009.
3. F. Jouault and I. Kurtev. "transforming models with atl,". In *in Proc. Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, 2005.
4. F. Jouault and I. Kurtev. On the interoperability of model-to-model transformation languages. *Science of Computer Programming*, 68(3, Special Issue on Model Transformation):114–137, 2007.
5. I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez. Model-based dsl frameworks. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 602–616, New York, NY, USA, 2006. ACM.
6. A. Slominski. Design of a pull and push parser system for streaming xml. Technical Report TR550, Department of Computer Science, Indiana University, 2001.
7. J.-S. Sottet and F. Jouault. Program comprehension. In *Proceedings of the 5th International Workshop on Graph-Based Tools (GraBaTs 2009)*, 2009. To appear.