Lazy Execution of Model-to-Model Transformations

Massimo Tisi, Salvador Martínez, Frédéric Jouault, and Jordi Cabot

AtlanMod, INRIA & École des Mines de Nantes, France
{massimo.tisi, salvador.martinez_perez, frederic.jouault, jordi.cabot}@inria.fr

Abstract. The increasing adoption of Model-Driven Engineering in industrial contexts highlights scalability as a critical limitation of several MDE tools. Most of the current model-to-model transformation engines have been designed for one-shot translation of input models to output models, and present efficiency issues when applied to very large models. In this paper, we study the application of a lazy-evaluation approach to model transformations. We present a lazy execution algorithm for ATL, and we empirically evaluate a prototype implementation. With it, the elements of the target model are generated only when (and if) they are accessed, enabling also transformations that generate infinite target models. We achieve our goal on a significant subset of ATL by extending the ATL compiler.

1 Introduction

Several Model-Driven Engineering (MDE) tools, when adopted in industrial contexts, show critical effeciency limitations in handling very large models (VLMs). When these tools are built around model-to-model (M2M) transformations, the efficiency of the transformation engine risks to become a performance bottleneck for the whole MDE environment. While specific M2M transformation languages and engines have been developed since several years [12,8,3], optimizing the transformation of VLMs is just becoming a compelling research task.

Lazy evaluation is one of the classical approaches that can provide, under specific conditions, a significant speed-up in program execution, especially when manipulating large data structures. When a programming language performs lazy evaluation, the value of an expression is calculated only when it is needed for a following computation (in contrast with *eager evaluation*, where expressions are evaluated as soon as they occur). This avoids the computation of unnecessary intermediate values. The useful part of large data structures is only calculated on-demand, even allowing for infinite-size data structures. For this reason lazy evaluation is a commonly used technique in several programming paradigms (for instance functional programming languages are classified in *lazy* or *eager*, depending on their evaluation strategy).

Lazy evaluation would significantly speed-up the execution of MDE tools based on M2M transformations, e.g., in cases where only part of the VLMs

involved in the transformations is actually used. Unfortunately, all the M2M transformation engines we are aware of support only eager computation of the target models. Models are always completely generated according to the transformation logic and it is not possible to automatically avoid the computation of model elements that will not be consumed afterwards.

This paper wants to provide the following contributions: 1) the study of the application of lazy evaluation to M2M transformation languages as a twofold problem, encompassing lazy navigation of the source model and lazy generation of the target model; 2) the implementation of an engine for lazy generation of the target model; 3) a practical evaluation of the lazy approach to model generation.

Our approach has been implemented in a prototype of a lazy transformation engine for the ATL [8] language, obtained by adapting the standard ATL engine. Our experimentation shows that M2M transformation languages like ATL, with an explicit representation of the transformation logic, can be naturally provided with an efficient lazy evaluation strategy.

Moreover, our approach to lazy generation allows the construction of an engine that can be plugged into existing tools consuming EMF models, without requiring modifications to the tools. The model is accessed like a normal EMF model, but its elements are computed on demand.

Finally the lazy generation approach can be naturally applied to transformations that generate an unbounded target model. Only the part of the model explicitly requested by the consumer is generated. In this way finite computations can make use of infinite intermediate models generated by transformation. This represents a significant extension of the application space of existing transformation languages.

The paper is structured as follows: Section 2 introduces the problems motivating the paper, by providing two running examples. Section 3 describes our approach to lazy execution of model transformations, in Section 4 we describe the implementation of a lazy engine for ATL and in Section 5 we experimentally evaluate its behavior; Section 6 discusses related work and, finally, in Section 7 we conclude the paper and propose further challenges.

2 Motivating Scenarios

In this section we provide two application scenarios that are the motivation for our work, running examples of the paper and subject of our experimental evaluation.

2.1 Scenario 1: Large Models

To illustrate how laziness addresses the performance problems of handling VLMs, we introduce an ideal database schema editor based on M2M transformations, whose structure is shown is Fig 1. This tool provides the user with an editor of the conceptual model of the database (in the form of a UML Class Diagram)

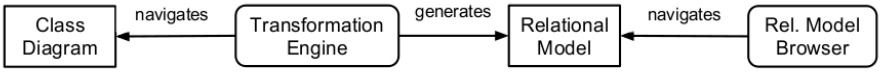

Fig. 1. A model-driven database schema editor.

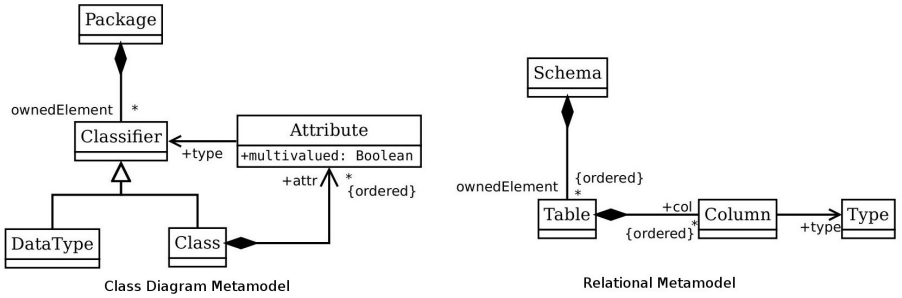

Fig. 2. Class and Relational metamodels.

and with a transformation that generates a corresponding relational model. The user can check the relational model by using a read-only model browser.

The tool uses a M2M transformation to generate the relational model from the Class diagram (the well-known Class2Relational transformation). In Fig. 2 we show the source and target metamodels of the transformation. The *ClassDiagram* metamodel represents a very simplified UML Class diagram. In this metamodel, *Packages* are containers of *Classifiers* that are either *Datatypes* or *Classes*. *Classes* can, in turn, be containers of *Attributes*, which can be multivalued. The *Relational* metamodel describes simple relational schemas. *Schema* contains *Tables* that are composed of *Columns*. Finally, *Columns* have a type that characterizes the kind of elements they can hold.

Listing 1.1 shows the main rules of the *Class2Relational* ATL transformation.

Listing 1.1. ATL Class2Relational transformation.

```
1
2  rule Package2Schema{
3    from
4      p: ClassDiagram!Package
5    to
6      out: Relational!Schema (
7        ownedElements <- p.ownedElement->
8                select(e | e.oclIsTypeOf(ClassDiagram!Class))
9      )
10 }
11
12 rule Class2Table {
13   from
14     c : ClassDiagram!Class
15   to
16     out : Relational!Table (
17       name <- c.name,
18       col <- Sequence {key}->
```

```
19              union(c.attr->select(e | not e.multiValued)),
20      key <- Set {key}
21    ),
22    key : Relational!Column (
23      name <- 'objectId',
24      type <- thisModule.objectIdType
25    )
26  }
27
28  rule DataType2Type {
29    from
30      dt : ClassDiagram!DataType
31    to
32      out : Relational!Type (
33        name <- dt.name
34      )
35  }
36
37  rule DataTypeAttribute2Column {
38    from
39      a : ClassDiagram!Attribute (
40        a.type.oclIsKindOf(ClassDiagram!DataType) and not a.multiValued
41      )
42    to
43      out : Relational!Column (
44        name <- a.name,
45        type <- a.type
46      )
47  }
48
49  rule ClassAttribute2Column {
50    from
51      a : ClassDiagram!Attribute (
52        a.type.oclIsKindOf(ClassDiagram!Class) and
53        not a.multiValued
54      )
55    to
56      foreignKey : Relational!Column (
57        name <- a.name + 'Id',
58        type <- thisModule.objectIdType
59      )
60  }
```

The ATL transformation constitutes a set of rules that describe how parts of the input model generate parts of the target model. These rules must have an *input pattern* and an *output pattern*. E.g., in the rule *ClassAttribute2Column* input model elements of type Attribute are selected to be transformed into output elements of type Column. Rules can have filters and bindings. Filters are used to impose conditions on the input elements selected by the input pattern and bindings are used to initialize values of the elements created by the output pattern. In the rule *ClassAttribute2Column*, a filter is introduced to select only Attributes that are not multivalued and whose type is *Class*. Two bindings are then used to initialize the name and type of the created Column. The rule *Class2Table* creates a Table for each Class, adds a *key* Column and initializes the list of columns with the respectively transformed Attributes. Finally, rule *Package2Schema* transforms a Package into a relational Schema and initializes the list of Tables.

The ATL transformation is executed in two steps. In the first step all the rules are matched creating all the corresponding target elements. Additionally, matching a rule creates, in the internal structures of the transformation engine,

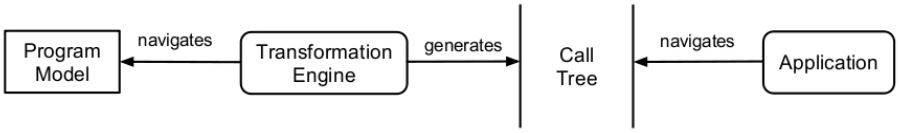

Fig. 3. A model-driven visualizator for method-call trees.

a traceability link that relates three components: the rule, the match (i.e. source elements) and the newly created target elements. In the second step, the created elements are initialized as described in the rule bindings. To perform the inizialization, ATL relies on a resolution algorithm that has been explained in details in [8].

Even with the very simple mapping of this scenario, when the Class diagram is large enough the transformation execution time can be significant. If the transformation engine has no support for change propagation (like the standard ATL engine), after each update to the Class Diagram, the user will have to wait the whole transformation processing, to see the corresponding element update in the relational model. Even a support for change propagation does not avoid the computation time for the whole initial target model.

In the following we propose a solution in which the tool offers a lazy exploration of the relational model. Transformation rules are activated only when the user requests to analyze a table, and only the necessary rules are executed. This delays the computation to the moment it is needed (at data consumption instead of data production) and strongly reduces the computation time.

2.2 Scenario 2: Infinite Models

As a second scenario we introduce a method-call hierarchy browser (similar to the one included in the Eclipse distribution) that computes this hierarchy in a model-driven way. The tool (Fig. 3) represents source code as a model conforming to the *Program* metamodel and uses a M2M transformation to generate the method-call hierarchy as a graph. Source and target metamodels are shown in Fig. 4, while Fig. 5 contains two example models. In the source model, *Programs* contain *Methods* that hold references to the other *Methods* they call. The target model is a simple tree, where *Nodes* represent method calls.

In Listing 1.2 we show an ATL transformation that performs the generation of the method-call hierarchy. Rule *Program2Root* translates the *Program* element into the root of the target tree. Then Program2Root activates the rule *Method2Node*[1] to generate a first-level node for each method. Finally *Method2Node* is a recursive rule that creates new children *Nodes* for each method call. In the

[1] *Method2Node*, in the ATL jargon is a *lazy rule*, i.e. a special kind of declarative rule that is only fired when directly called from other rules. We will omit discussing lazy ATL rules in the following sections, to avoid confusion with the concept of laziness we are promoting in this paper (i.e. the rules are not activated until their target

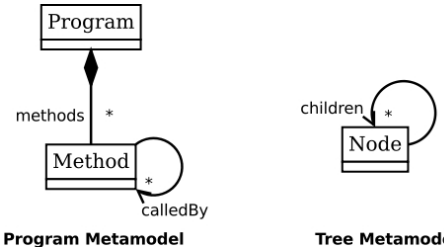

Fig. 4. Program and Tree metamodels.

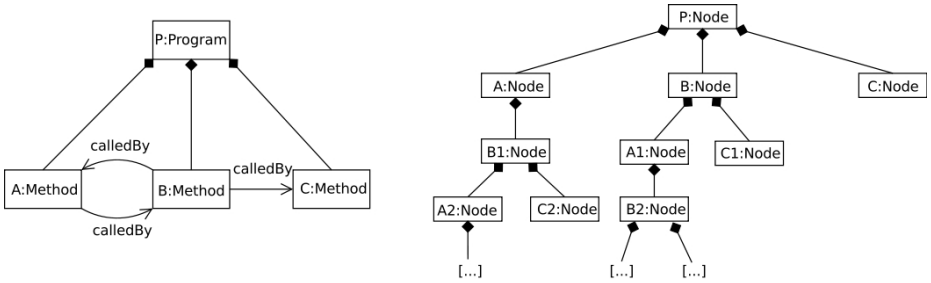

Fig. 5. A recursive program and its corresponding method-call hierarchy.

common case in which two methods call each other (as in Fig. 5), the source model will contain a loop and the target method-call tree will become infinite (the *Method2Node* rule will continue to recur).

Listing 1.2. ATL infinite transformation

```
1
2  rule Program2Root {
3    from
4      s : Program
5    to
6      t : Node (
7          children <- s.methods->collect(e | thisModule.Method2Node(e))
8          )
9  }
10
11 lazy rule Method2Node {
12   from
13     s : Method
14   to
15     t : Node (
16       children <- s.calledBy->collect(e | thisModule.Method2Node(e))
17         )
18 }
```

Contrarily to the example in Listing 1.1, this transformation cannot be executed in the current ATL engine, since its computation will not terminate. On

element is needed). However our prototype engine includes support for ATL lazy rules.

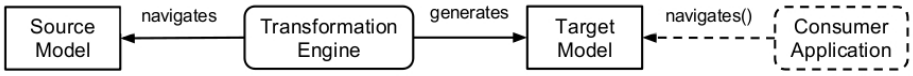

Fig. 6. Transformation and consumer.

the contrary, the lazy engine we propose can launch the transformation, and generate nodes on demand, when the user browses the tree. Moreover the consumer tool could compute in a finite time expressions on infinite method-call models (e.g., reachability of method C from method A in a given number of steps).

3 Lazy Model Transformation

Since every M2M transformation is both a producer and a consumer of models, a lazy approach to model transformation has to address both the aspects of lazy production and lazy consumption.

1) Model production (or *model generation*) by model transformation is based on an execution strategy that is built-in in the transformation engine or user-controllable as part of the transformation language. Lazy generation requires the introduction of a lazy execution strategy, driven by external model-consumption events.

2) Independently from the execution strategy, transformation engines need to analyze input models, to obtain the necessary information for controlling their execution or computing output values. In MDE we usually refer to this phase with the term *model navigation*. A lazy transformation approach involves lazy navigation of source models, possibly performed by a navigation language with lazy evaluation.

In this section we study the two aspects of lazy model generation and navigation and we argue that they are separated and orthogonal to each other. We will refer to a general schema in which the transformation is connected to a consumer application, as in Fig. 6.

3.1 Lazy Model Generation

With *lazy model generation* we indicate on-demand activation of the computation for generating a data element of the target model. No assumption is made on the strategy for extracting and evaluating data from the source model, i.e. lazy generation is independent from the navigation mechanism. Since only the required subset of the target model is computed, lazy generation can be used to address the problem of VLMs (or infinite models) when they are the *target* of the transformation.

Eager transformation languages activate rules according to internal execution strategies. *Source-driven* transformation languages base their execution strategy

on the structure of the source model. For instance ATL has a source-driven execution algorithm that associates source elements to matching rules and fires the rules in non-deterministic order [8]. *Target-driven* languages in contrast follow a predetermined production order for target elements (e.g., sequential or template-based). Lazy generation can be seen as an alternative execution strategy that differentiates from the previous ones for being driven by a special kind of external events, i.e. the consumption requests.

With respect to an eager system, a transformation system with lazy generation has to provide some additional features:

1. To initiate the lazy generation process, consumption requests on the target model have to be tracked. This requires extending the model navigation mechanism the consumer uses, to intercept the requests and activate a corresponding generation in the transformation engine. If this adaptation can be performed in a transparent way, the client system will not notice that the model is lazily built. For instance, several transformation languages use EMF [15] as their model management system. A naturally transparent extension mechanism in this case would be to re-implement the EMF API, so to provide the same interface of a standard EMF model. For performing lazy access by the EMF API, we only need to override the *eGet()* method in the implementation of model elements, in order to trigger a call to the engine operations.
2. The transformation engine has to provide the means to launch the computation of a single model element or a single property of the target model. The degree of laziness in computing the target elements is strongly dependent on the modularity of the transformation algorithm. E.g., in a transformation language natively designed to maximize independent computation of target elements, the performance of a lazy system would be optimal.
3. Finally, the lazy engine can keep track of the status of the partial transformation, and use it as a context for the execution of new computations. The stored context is exploited by the lazy system to avoid recomputations. In transformation systems this context usually includes trace links that map elements in the target model with their corresponding sources. E.g., the trace links in the current state can be used to avoid recomputing previous matches when a new value is requested. Lazy transformation engines that keep extra state information can be *live systems*, and keep their state information constantly in memory, or *offline systems*, and provide a way to freeze and restore their state.

Once the engine provides this infrastructure, a generic lazy generation algorithm works in three corresponding steps:

1. the consumer requests a new target element, and the call gets intercepted by the navigation interface of the target model;
2. the navigation interface (e.g., the lazy model) requests the engine to generate the single requested property or element;
3. the engine determines the computations to activate, based on the current status of the transformation.

In Section 4 we describe an implementation of this approach for the ATL language.

3.2 Lazy Model Navigation

Model navigation can be a more or less clearly separated phase in the transformation execution. In several M2M approaches a different language is used specifically for model navigation. For instance, popular languages like QVT/R [12], ATL [8] and Kermeta [3] use OCL [13] to write expressions on the source models.

Adding a lazy evaluation strategy to the model navigation mechanism allows the engine to 1) delay the access to source model elements to the moment in which this access is needed by the transformation logic and, by consequence, 2) reduce the number of source model elements accessed during navigation, by skipping the unnecessary model elements. For this reason, lazy model navigation can be used to address the problem of VLMs when they are the *source* of transformation. For instance, in Scenario 1, lazy navigation would speed-up the evaluation of expressions on big Class diagrams.

On the other hand, the problem of lazy navigation does not only exist in transformation systems, as navigation mechanisms and languages are commonly used outside of transformations. Fig. 6 shows that the consumer application needs to navigate the generated target model. Lazy target navigation by the consumer is in principle not different from lazy source navigation by the transformation engine. In the case in which transformation and consumer use the same navigation language, a lazy implementation can be re-used for both phases.

Navigation languages can be generally augmented with a certain degree of laziness. For instance, in the case of functional navigation languages, the research problem of implementing a lazy strategy for an existing language is already deeply studied (e.g., in [6]). In the task of adding laziness to OCL some work has already been carried out in [1] and in [2]. Hence, our prototype engine only focuses on lazy generation. However, in building our implementation we maintain a clear decoupling among navigation and generation to allow for independent development of both parts.

Finally, an issue tightly coupled to lazy navigation, is on-demand physical access to the source model elements, i.e. lazy loading. For lazy loading of models for transformation we refer the reader to [9].

4 A Lazy Engine for ATL

To demonstrate the feasibility and performance of lazy generation we implemented a prototype engine that activates ATL rules on demand.[2]

[2] The full code of the prototype is available at the following address: `http://www.emn.fr/z-info/atlanmod/index.php/Lazy_ATL`

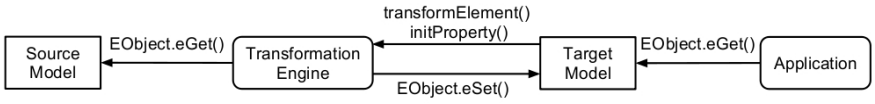

Fig. 7. Adapted lazy transformation engine

4.1 Transformation Engine

Our implementation consists of an extension of the standard ATL compiler[3] and an adaptation of the EMF *EObject* class. We don't modify the syntax of the ATL language and we reuse the standard ATL Virtual Machine.

The prototype implements the three features discussed in Section 3.

1. To intercept consumption requests we provide an adaptation of EMF *EObject*, called *LazyEObject*. *LazyEObject* implements the EMF interface and overrides only the method *eGet(EStructuralFeature eFeature)* used to request for a model feature from normal EMF model elements. In the new implementation, *eGet()*: a) checks that the requested feature is still not initialized and b) calls the *initProperty* operation of the compiler. The computation of the requested property and its physical storage in the target model for future reuse is delegated to the transformation engine.
2. On-demand computation of model elements and attributes is implemented by refactoring the standard ATL execution algorithm described in Section 2. Once new elements or properties have been computed, the transformation engine explicitely stores their value in the target model by calling a standard *EObject.eSet()* (i.e., data is *pushed* by the transformation engine, and not *pulled* by the lazy model). The fact that the client system leaves to the transformation engine the responsibility to explicitly fill the target model, allows us to keep an execution semantics for atomic initialization as similar as possible to the standard ATL engine. This simplifies the lazy engine implementation, as well as the subsequent maintenance in parallel with the standard engine. Practically, the lazy engine has been refactored to expose two new operations, additional to the standard ones:

   transformElement(source: EObject) The operation *transformElement* performs on-demand transformation of single elements, by activating the ATL rule that matches a given source element and creating the corresponding target. The properties of the newly created elements are not computed in this phase, but they have to be explicitely filled by subsequent calls to the operation *initProperty*. In ATL, once a rule is matched, more elements are generated at once (output pattern). The matching phase has a much higher cost than the creation of new empty elements in the target. For this reason in our implementation *transformElement*, together with the target element that has been requested, generates all

[3] On the Eclipse CVS: /modeling/org.eclipse.m2m/atl/dsls/ATL/Compiler/ATL.acg

the output pattern at once. This optimization is invisible to the user, and can be easily disabled.

initProperty(target: EObject, propertyName: String) The operation *initProperty* performs on-demand generation of target properties by computing the corresponding ATL bindings. If the property is an attribute its value is computed and stored in the target model. If the property is a reference, the ATL binding is into a set of source elements, the trace links of these elements are navigated to retrieve the corresponding targets (as it happens for the standard ATL resolution algorithm). If a source element has no associated traceability links (which means that it has not been transformed), a transformation on that element is launched by a call to *transformElement*.

3. As in standard ATL, the state of the current transformation is stored as trace links that relate source elements with target elements and their connecting rule. The set of traces in the lazy engine is initially empty, it gets initialized when the lazy transformation is started by a call to *transformElement*, and then it grows monotonically while the user navigates the target model (activating calls to *initProperty* and *transformElement*). For simplicity we implemented our system as a *live transformation system* that keeps its state information in memory, but we plan in future to exploit the serialization of the trace link information provided by the ATL engine to implement an offline behavior.

4.2 Considered ATL Subset

Our prototype supports a well-defined and fully functional subset of the ATL language. While the following advanced features are not supported yet, they do not pose a significant research problem and are included in our future plans:

- **Resolution of specific target elements (resoveTemp operation)** should be extended to launch the correct rule in case the element to be resolved has not been created yet.
- **Rule inheritance** could be natively handled in a future version of the lazy engine. However the inheritance tree in ATL can always be eliminated by copying the inherited features.
- **Multiple source pattern elements** would require to extend the logic to get, using trace links, source elements from target ones.

Adding laziness to other aspects of ATL instead would not be trivial:

- **Reverse bindings** are a means to set the incoming references of the target element. To detect if a reference is modified by reverse bindings, all of them have to be computed in an eager way.
- **Refining mode**: the engine for in-place transformations in ATL first computes the set of changes to apply and then executes them on the source model. Lazy generation and application of changes in ATL has yet to be studied.

– **Imperative constructs**, whose use should be avoided in ATL whenever possible, create and modify target elements without producing corresponding traces: this is not compatible with our approach in its current state.

5 Approach Evaluation

For the experimentation phase, we built a consumer program that performs controlled sequences of accesses to the target model and records the evaluation times, both in lazy and eager modes.

When executed in lazy mode, the consumer uses the Eclipse infrastructure to initiate the process. An extended Eclipse EMF editor allows the user to visualize the source model, select a starting source element and launch the lazy transformations from the source element. The rule matching the selected source element is immediately activated, initial target elements are generated, and the consumer starts browsing the output model by navigating the references of the target model elements. In eager mode the consumer simply launches programmatically an ATL transformation and then performs the same sequence of accesses as in the lazy mode. We implemented different navigation strategies for the target model (e.g. depth-first, random), and the experimentation results do not show significant variations in this respect.

To evaluate the behavior in Scenario 2, the transformation in Listing 1.2 is launched and a constraint is programmatically checked (e.g., reachability of method C from method A in a given number of steps). The computation does not terminate in eager mode and generates an immediate result in lazy mode.

The performance hit of the lazy approach in Scenario 1, is illustrated in Fig. 8. Four sets of tests have been executed, each one characterized by a different source model. The four source models, ordered by increasing model size (respectively of 8020, 16020, 25220 and 50420 elements), originated the four graphs in figure. In all the tests, we applied the transformation of Listing 1.1 in lazy and eager mode, and we navigated a fixed number of target elements[4]. The graphs in Fig. 8 are obtained by varying the length of the navigation and marking the correspondent computation time.

To reduce perturbations, each test has been repeated ten times, with exactly the same conditions. Each point in the graphs of Fig. 8 represents the average value of ten identical tests (actually the first iteration was discarded, to avoid any initialization overhead).

As expected, for a small number of accesses to VLMs, the lazy approach results much faster than the eager one. However, when the number of accesses is close to the size of the model the lazy approach is notably slower. This performance drawback is due to the overhead introduced by the lazy execution, as extra operations have to be performed everytime an element is generated (to find the source model from the trace link, check guards, etc). Nevertheless, it's

[4] The experimentation has been performed in the following hardware and software setting: Eclipse 3.5.2, Ubuntu 10.04, Linux kernel v2.6.32, Dell Latitude E6410, Intel Core i7 processor (2,67 GHz).

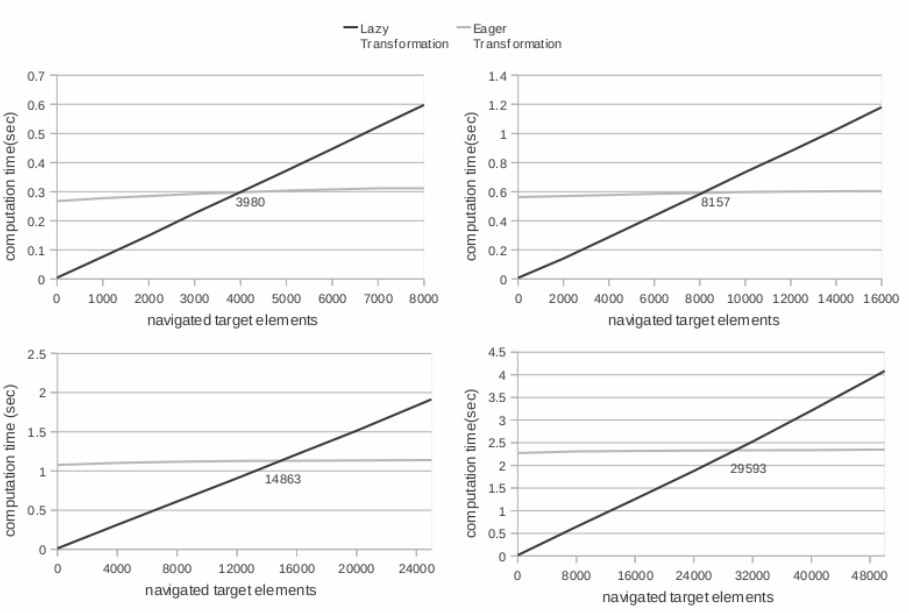

Fig. 8. Experimental results

interesting to observe that the lazy approach keeps better performance than the eager one until a significant percentage of target model navigation (from 48% in the smallest case to 58% in the biggest). Moreover, the evident similarity among the plots, with a nearly linear increase of computation time in the lazy case, shows that we have an approximately constant speed-up with the increase of model size.

The performance gap would be much wider for models big enough to exceed the computer memory. In this cases a lazy approach avoids, for limited navigations, the performance drop caused by memory management mechanisms.

6 Related work

As we said, we are not aware of any transformation tool with a lazy generation strategy in MDE. The Stratego [18] system allows user-defined execution strategies for transformation rules. While user-defined strategies have been used to implement target-driven approaches [19], the activation of rules as answer to external consumption has not been addressed. VIATRA, despite not implementing on-demand transformation, evaluates lazily the matchings of connected rules to avoid unnecessary computation, as described in [16].

Outside the MDE domain, [14] follows an approach similar to ours. The authors provide an interpreter for XSLT that allows random access to the transformation result. They also show how their implementation enables efficient pipelining of XSLT transformations. We discuss laziness for generic model transformation languages (generally not XSLT-like), proposing a modularization of the problem and focusing on the integration of lazy model transformation in existing tools.

The implementation of a lazy evaluator for functional (navigation) languages is a subject with a long tradition [7]. We refer the reader to [6] for an example based on Lisp. This subject has been explored in [1] and in [2] where performance measures are presented.

Other optimization techniques has been explored in model transformation engines. Lazy loading [9] is a complementary subject to lazy navigation, when dealing with models that do not fit into the memory of the transformation engine. [11] presents methods to evaluate pattern matches of different rules in an overlapped way, to increase performance. In [5] transformation context is preserved to efficiently perform incremental updates whereas in [17] and [4] strategies for the problem of graph pattern matching optimization are investigated. Finally this paper follows the opposite direction of [10], that adds forward change propagation to ATL. The study of the possible combination of laziness and incrementality in ATL and other M2M languages will be part of our future work.

7 Conclusions and future work

Our experimentation showed that lazy execution of transformation rules is a specific feature of transformation languages that can provide a remarkable performance gain and can extend the application space of transformation languages to infinite data structures. We envision several possible future extensions:

Complete coverage of ATL. The implementation presented here covers a significant and functional subset of ATL but in the future we plan to extend the support to the complete declarative part of the ATL language.
Optimization of the lazy engine. As in other lazy approaches, we want to evaluate the possibility to store intermediate expression values to avoid intermediate recomputations in target generation. We plan to add a lazy OCL evaluator to address the sub-problem of lazy source navigation (which is especially relevant when working with big source models).
Incrementality. We plan to study the interaction between forward change propagation and laziness in M2M languages and provide a combined engine for ATL.
Transformation chains. Finally we want to study the concatenation of lazy transformations and the possibilities of pipelining.

References

1. O. Beaudoux, A. Blouin, O. Barais, and J.-M. Jézéquel. Active operations on collections. In *MoDELS*, volume 6394 of *LNCS*, pages 91–105. Springer, 2010.

2. M. Clavel, M. Egea, and M. A. G. de Dios. Building an efficient component for OCL evaluation. *ECEASST*, 15, 2008.
3. Z. Drey, F. Fleurey, D. Vojtisek, C. Faucher, and V. Mahé. *Kermeta Language, Reference Manual*, 2009.
4. R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In *ICGT*, volume 4178 of *LNCS*, pages 383–397. Springer, 2006.
5. D. Hearnden, M. Lawley, and K. Raymond. Incremental model transformation for the evolution of model-driven systems. In *MoDELS*, volume 4199 of *LNCS*, pages 321–335. Springer, 2006.
6. P. Henderson and J. H. Morris, Jr. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, POPL '76, pages 95–103. ACM, 1976.
7. P. Hudak, J. Hughes, S. L. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *HOPL*, pages 1–55. ACM, 2007.
8. F. Jouault and I. Kurtev. Transforming models with ATL. In J.-M. Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 128–138. Springer, 2005.
9. F. Jouault and J. Sottet. An AmmA/ATL Solution for the GraBaTs 2009 Reverse Engineering Case Study. In *5th International Workshop on Graph-Based Tools, Grabats*, 2009.
10. F. Jouault and M. Tisi. Towards incremental execution of ATL transformations. In *Theory and Practice of Model Transformations*, pages 123–137. Springer, 2010.
11. T. Mészáros, G. Mezei, T. Levendovszky, and M. Asztalos. Manual and automated performance optimization of model transformation systems. *STTT*, 12:231–243, 2010.
12. OMG. *MOF QVT Final Adopted Specification*. Object Management Group, 2005.
13. OMG. *Object Constraint Language Specification, version 2.0*. Object Management Group, June 2005.
14. S. Schott and M. L. Noga. Lazy XSL transformations. In *ACM Symposium on Document Engineering*, pages 9–18. ACM, 2003.
15. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework (2nd Edition) (The Eclipse Series)*. Addison-Wesley Professional, 2008.
16. G. Taentzer, K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, D. Varró, and S. Varró-Gyapay. Model transformation by graph transformation: A comparative study. In *Proc. Workshop Model Transformation in Practice*, 2005.
17. G. Varró, K. Friedl, and D. Varró. Adaptive graph pattern matching for model transformations using model-sensitive search plans. *Electr. Notes Theor. Comput. Sci.*, 152:191–205, 2006.
18. E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 216–238. Springer, 2003.
19. J. V. Wijngaarden and E. Visser. Program transformation mechanics: A classification of mechanisms for program transformation with a survey of existing transformation systems. Technical report, UU-CS, 2003.