Reactive Model Transformation with ATL

Salvador Martínez, Massimo Tisi, Rémi Douence

▶ **To cite this version:**

Salvador Martínez, Massimo Tisi, Rémi Douence. Reactive Model Transformation with ATL. Science of Computer Programming, 2017, 136, pp.1 - 16. 10.1016/j.scico.2016.08.006 . hal-01627991

HAL Id: hal-01627991
https://inria.hal.science/hal-01627991v1

Submitted on 2 Nov 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Reactive Model Transformation with ATL

Salvador Martínez[a], Massimo Tisi[a], Rémi Douence[b]

[a] *AtlanMod team (Inria, Mines Nantes, LINA), Nantes, France*
[b] *Ascola team (Inria, Mines Nantes, LINA), Nantes, France*

Abstract

Model-driven applications may maintain large networks of structured data models and transformations among them. The development of such applications is complicated by the need to reflect on the whole network any runtime update performed on models or transformation logic. If not carefully designed, the execution of such updates may be computationally expensive. In this paper we propose a reactive paradigm for programming model transformations, and we implement a reactive model-transformation engine. We argue that this paradigm facilitates the development of autonomous model-driven systems that react to update and request events from the host application by identifying and performing only the needed computation. We implement such approach by providing a reactive engine for the ATL transformation language. We evaluate the usage scenarios that this paradigm supports and we experimentally measure its ability to reduce computation time in transformation-based applications.

Keywords: Model-Driven Engineering, Model Transformations, Reactive Programming

1. Introduction

Model-driven applications, i.e. applications based on explicit structured data models, are becoming wide-spread within both the academic and industrial worlds. Modeling frameworks like EMF (the Eclipse Modeling Framework[1]) simplify the development of such applications by providing a standard representation and interface to the data structure (i.e., the model), code generation facilities, scalable serialization and interoperability with several model-driven tools. Modeling frameworks are extensively used in a wide range of scenarios[2] and even the Eclipse 4 platform is itself developed on EMF[3].

Model-driven applications manipulate models by executing *model transformations*, defined using general-purpose languages (GPLs, e.g., Java), or domain-

[1] `www.eclipse.org/emf/`
[2] `http://en.wikipedia.org/wiki/List_of_Eclipse_Modeling_Framework_based_software`
[3] `http://www.eclipse.org/e4/resources/e4-whitepaper.php`

specific languages called model-transformation languages (MTLs). Popular MTLs like the Object Management Group (OMG) QVT (Query/View/Transformation) [1] and the AtlanMod Transformation Language (ATL) [2] are supposed to provide a higher level of abstraction and expressiveness in describing the transformation as a relationship among source and target model elements. In a typical model-driven application, the application logic is separated in two parts: 1) *transformations* specify *how* to derive target models from source models, 2) the *host application* defines *what* (i.e., which model elements) to compute, *when*, and *why* (e.g., in response to which events). The application is responsible for explicitly executing the transformations, seen as black-box functions that return the computed target models.

In this paper we argue that MTLs, when provided with a specific execution environment, enable a beneficial paradigm shift for programming model-driven applications, towards reactive programming. *Reactive programming* [3] denotes a programming paradigm oriented towards event processing and change propagation through data-flows. Values change over time and, when they change, all dependent computations are automatically re-executed [4]. By taking inspiration from this concept we propose, for model-driven applications, a paradigm where a network of *reactive transformations* defines persistent data-flows among models. A *reactive transformation engine* takes care of activating only the strictly needed computation in response to *updates* or *requests* of model elements[4]. Computation is updated when necessary, in an autonomous and optimized way. The application architecture is deeply changed, since the host application does not directly control the execution of the transformations anymore, but only accesses or updates the underlying models.

We want to argue that reactive transformation systems have the following benefits:

1. The development of the host application is simplified, since the application relinquishes responsibility over *when* and *what* to transform. However, despite this loss of control, the reactive transformation engine can be used for an efficient implementation of all the application scenarios of traditional transformation engines.
2. The amount of computation to perform is automatically minimized, allowing for more efficient applications in several scenarios, w.r.t. traditional transformation systems.

We support our arguments by implementing a reactive engine for the ATL language[5], evaluating its application to different usage scenarios and experimentally assessing the performance gain w.r.t. the standard ATL engine in the same application. For implementing our reactive transformation engine for ATL we

[4]Note that differently from general reactive programming languages [4], our specific languages for reactive transformation are focused on the definition of the transformation logic, and they don't expose an explicit notion of time.

[5]The full code of `Reactive ATL` and the running case can be found at the address: `https://github.com/atlanmod/org.eclipse.atl.reactive`

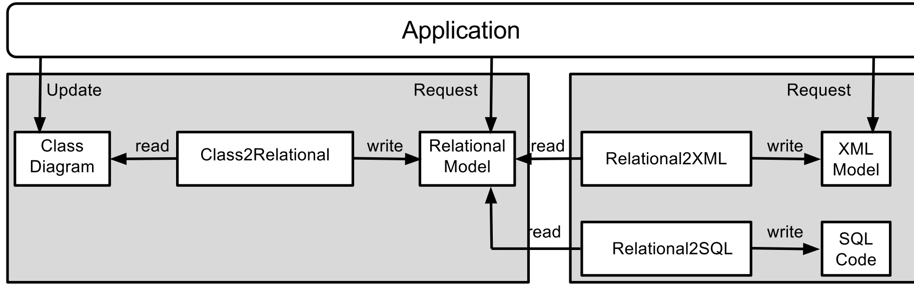


Figure 1: The transformation chain: Class2Relational + Others

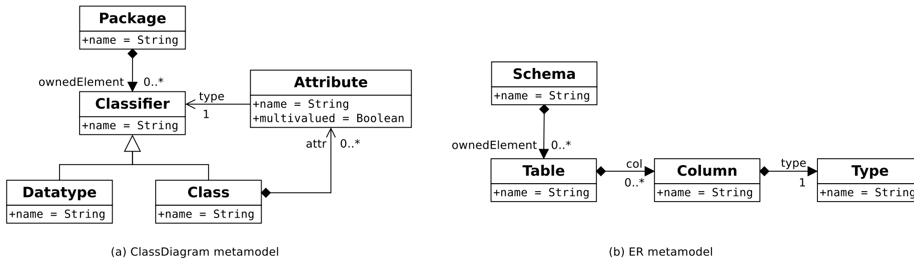


Figure 2: Class diagram and ER metamodels

build on our previous work, where we studied the possibility to apply incrementality [5] and lazy evaluation [6] to model transformations. In this paper we design a transformation system that reacts to both updates and requests, by integrating incrementality and lazy evaluation as dual aspects of reactive transformation in a new invalidate/lazy recompute evaluation strategy. In addition, our system is able to react to updates to the model transformation code itself, by identifying the computation needed to keep the whole transformation network consistent. Finally, we provide a formalization in Haskell of the semantics of the previously available ATL engines and our new reactive engine, and we automatically check their input-output equivalence.

The rest of the paper is organized as follows. Section 2 presents a running example. Section 3 describes our general architecture whereas Section 4 focuses on the advancements with respect to standard ATL. Section 5 illustrates the corresponding implementation and Section 6 shows the results of a performance evaluation. Section 7 provides a detailed insight into related works. Finally Section 8 concludes and outlines possible future work.

2. Running Example

To illustrate the mechanism of reactive transformation, we will use through the paper a basic running scenario on the development of a database schema editor based on model transformations (see Figure 1). The developer in charge of producing such application should provide the user with an editor of the

conceptual model of the database (in the form of a UML Class Diagram) and a model transformation that generates a corresponding relational model (i.e., the *Class2Relational* transformation). The developer then provides the user with the capability of browsing the relational model in read-only mode. Additionally, the developer may also provide means for the relational model to be transformed to an XML model or to SQL code in order to facilitate interoperability, code-generation and/or report generation tasks. Thus, our ideal database schema editor may constitute a transformation chain as we can see in Figure 1 (the class diagram model, relational model, XML model and SQL code can, in turn, be part of other more complex transformation networks).

In this scenario, updates of the source class diagram and inspection of its corresponding relational model may occur often. Therefore, in order to reduce the amount of work to be performed by the transformation chain, it would be desirable to be able to: 1) propagate only the changes to the affected elements in the Relational Model when the user modifies the Conceptual UML model, 2) perform the transformation and updates w.r.t. possible changes in the Conceptual UML model, in an on-demand basis, so that unread Relational Model elements do not trigger any unneeded transformation evaluation. In other words, it would be desirable for the transformation chain to handle source updates in an *incremental* manner and target model requests in a *lazy* manner. For example, adding a *Class* should only trigger the computation of the corresponding *Table* (and then, the corresponding XML element) in the target models. Reading the *Columns* of a given table should only trigger the execution of the transformation rules for those *Columns*, e.g. ignoring the *Columns* of other *Tables* until they are requested.

In Fig. 2, we show the source and target metamodels of the *Class2Relational* 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*. *Columns* have a type that characterizes the kind of elements they can hold.

Listing 1 shows the main rules of the *Class2Relational* transformation written in ATL. The ATL transformation is made of 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. For instance, in rule *ClassAttribute2Column*, a filter is introduced to select only Attributes that are not multivalued and whose type is a *Class*. Two bindings are then used to initialize the name and type of the created Column. 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.

Listing 1: ATLClass2Relational transformation.

```
rule Package2Schema{
  from
    p:ClassDiagram!Package
  to
    out:Relational!Schema (
      ownedElements<-p.ownedElement
      ->select(e|e.oclIsTypeOf(
         ClassDiagram!Class)))
}

rule Class2Table {
  from
    c:ClassDiagram!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'
}

rule DataType2Type {
  from
    dt:ClassDiagram!DataType
  to
    out:Relational!Type (
      name <- dt.name)
}
```

```
rule DataTypeAttribute2Column {
  from
    a:ClassDiagram!Attribute (
      a.type.oclIsKindOf(
        ClassDiagram!DataType)
        and
        not a.multiValued)
    to
      out:Relational!Column (
        name <- a.name,
        type <- a.type
    )
}


rule ClassAttribute2Column {
  from
    a:ClassDiagram!Attribute (
      a.type.oclIsKindOf(
        ClassDiagram!Class)
        and
        not a.multiValued)
  to
      foreignKey:Relational!Column(
        name <- a.name + 'Id',
      )
}
```

The incremental and lazy transformation system we propose, can also react to updates to the transformation definition itself. A transformation definition may need to be updated in order to modify some target model initialization values (e.g., ids, types, etc), add new rules to take care of previously ignored elements, or to answer to any new requirement. As an example, Listing 2 shows two updated rules for the Class2Relational transformation, that add the processing of *Multi-Valued Data-Type Attributes*. The new *MVDTAttribute2Column* rule transforms such elements and the *ownedElements* binding in *Package2Schema* is updated to include the newly generated tables. Since the model transformation system is kept constantly live and active so that it can react to model updates and requests, it would be desirable if the system could react seamlessly to updates in the transformation code and propagate only the changes imposed by these updates (without needing to stop the live system and relaunch it with the new specification). Concretely, in our example, (only) the *Multivalued DataType Attributes* elements should be transformed and (only) the *ownedElements* property re-evaluated, avoiding the need for re-launching the full transformation.

In the following sections we will describe a reactive transformation engine and framework for efficiently dealing with this scenario. For simplicity, in the following sections we limit the discussion to applications based on 1-to-1 unidirectional transformations, analogous to Figure 1. This kind of application

maintains a single source model. The model is then manipulated by a transformation chain, but the generated models are read-only. However our reactive system also supports transformations with multiple source/target models.

Listing 2: Updated ATLRelational transformation.

```
rule Package2Schema{
  from
    p:ClassDiagram!Package
  to
    out:Relational!Schema (
      ownedElements <- let classes:
      Set(ClassDiagram!Class) =
      p.ownedElement
      ->select(e | e.oclIsTypeOf(
          ↪ClassDiagram!Class)) in
      classes->union((
      classes->collect(e|e.attr))
         ->flatten()
      ->select(e|e.multiValued)
      ->asSet()),
      name <- p.name
    )
}
```

```
rule MVDTAttribute2Column {
  from
    a:ClassDiagram!Attribute (
      a.type.oclIsKindOf(
          ↪ClassDiagram!DataType)
      and a.multiValued)
  to
    out:Relational!Table (
      name <- a.owner.name
              + '_' + a.name,
      col <- Sequence{id, value}
    ),
    id:Relational!Column (
      name <-
        a.owner.name.firstToLower()
        + 'Id'
    ),
    value:Relational!Column (
      name <- a.name
    )
}
```

While this example is small enough to illustrate the mechanism of reactive transformation, the motivating cases for our work come from complex transformations in industry-related projects. For instance, in the context of the MONDO EU FP7 project[6], reactive transformations have been applied to the development of synchronized views for large building information models (BIM) [7]. A BIM model is a multidisciplinary data model which describes all the information pertinent to a building and its components. It is described using the IFC (Industry Foundation Classes) specification [8], a freely available format to describe, exchange, and share information typically used within the building and facility management industry sector. The IFC model is necessarily large and complex, as it includes all common concepts used in building-industry projects, from feasibility analysis, through design, construction, and operation of a built facility. For this reason, views (similar to UML viewpoints) are defined to satisfy a particular information exchange requirement that does not concern the full IFC model. Relying on the aforementioned view definitions, view transformations in ATL have been defined. Given the size of BIM models, that can reach several millions of model elements, views are required to be computed on demand, only for the strictly necessary parts while changes in the source full IFC model should avoid the need to relaunch the full transformation (view generation).

[6] http://www.mondo-project.org/

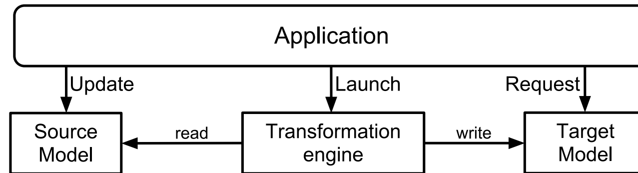


Figure 3: Common model-driven application architecture

3. The Reactive Transformation System

The typical architecture of a model-driven application, shown in Figure 3, promotes the separation of the modeling layer from the application layer (in order to simplify the discussion, we refer here to 1-to-1 model transformations). In this architecture, an application interacts with the modeling layer by a sequence of 1) updates to elements of source models, 2) requests of elements of target models and 3) direct execution of model transformations, to control in which moment to perform the transformation computation[7]. The ordering of these events depends on the concrete application logic or runtime user interaction. The organization in two layers reflects also into a separation of concerns: the modeler does not need to know the application logic and neither the language in which it is written, the developer of the application layer does not need to know about transformation logics and languages.

This programming model is suited only to small applications:

- For larger systems, that maintain an elevated number of models organized in transformation networks, the application has to explicitly launch (only) the required transformations at each model update. This direct control is delicate and error-prone.
- While it is certainly possible for the application developer to design efficient solutions to the data propagation problem (e.g., a lazy computation system) the implementation of such logics is not trivial and it has to be repeated for each application.
- To explicitly activate the right propagation in the right moment, the application developer needs to have a deep understanding of the transformation network. In cases in which he wants a fine-grained control over the propagation, he needs also to know the semantics of each transformation rule, to activate the propagation only of the updated element. This breaks the separation of concerns that the two-layer architecture tries to promote.

[7] *Source* and *target* are the roles that the model takes w.r.t. the transformation. In the same application a model will generally be source of some transformations and target of others. Notable is the case of bidirectional transformations, that we simplify here as a couple of unidirectional transformations.

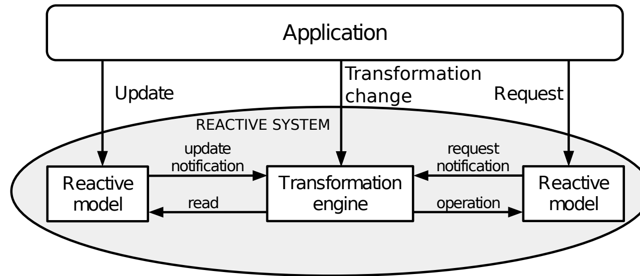


Figure 4: Reactive transformation system

We propose to address these problems by shifting the traditional programming paradigm of model transformations. Today transformations are seen as *transformation functions*, calculating an (updated) target model. We propose to program model transformations as reactive programs, managed by a event-driven reactive engine. A reactive transformation engine would autonomously activate the computation in answer to external events, taking complete charge of data propagation in the model driven system. For a model-transformation system these events are model updates and requests coming from the application layer, along with changes to the transformation itself.

The traditional advantages of reactive programming apply also to the model-transformation case:

- when dealing with complex propagation networks, a reactive engine frees the application developer from designing the propagation system; The same is true for the higher-order case, where updates occur to the transformation itself;
- the engine provides an optimized propagation mechanism, using strategies like lazy and partial evaluation, whose implementation would be too expensive for the application developer; the engine can perform the propagation at arbitrary granularity, while fine-grained propagation systems would be more difficult to implement for the application developer;
- the developer can focus on describing the rest of the application and does not have to get in contact with the transformation logic or even with the transformation language.

Our proposed architecture for reactive transformation systems is represented in Figure 4. A reactive system is made of two components: a *Reactive modeling framework* (i.e., the interface to source and target reactive models) and a *Reactive transformation engine*:

1. To be able to react to modeling events, the transformation engine has to intercept them. This task is performed by the modeling framework, that receives updates and requests from the application. Hence we have to substitute (by maintaining the same interface, so that maximum compatibility

with existing tools is maintained) a standard modeling framework (e.g., EMF) with a reactive version that communicates to the transformation engine the arrival of events.

2. The reactive transformation engine will be responsible of calculating the transformation computation that needs to be updated. Note that this includes calculating the required updates to be performed in order to synchronize an ongoing transformation operation with a new transformation definition. A reactive transformation engine for ATL will be described in Section 4.

3.1. Reactive Modeling Framework

Having a single standard modeling framework accessed by all the applications in a technical space is a crucial aspect of model-driven engineering. To maintain interoperability with the technical space we need a mechanism to allow existing applications to use the reactive modeling framework transparently, without requiring modifications. For this reason we propose a reactive modeling framework that re-implements the interface of the standard modeling framework by adding a reactive behavior to each method.

In the architecture of Figure 4 we use thus a *ReactiveObject* (see Section 5) for the elements of both, the source and target model making them reactive models. This uniform treatment of source and target events (and correspondingly of incrementality and lazy evaluation) allows us to construct chains of reactive transformations by sharing intermediate reactive models.

Our so-called *ReactiveObject* allows transformation engines to subscribe to it and be notified according to the observer pattern. As an observed object, a *ReactiveObject* is not aware of the specific transformation engine that will react to its events. In this architecture, the responsibility of the modeling framework is only to notify the transformation engine, that will have to answer accordingly by computing updated values for target elements or properties.

We add then an invalidation mechanism for model elements and properties, useful to postpone the computation of updated values until they are actually required. When a change occurs in the source model, instead of forwarding the changes to the target model, target element invalidation operations are performed. This way, the actual change propagation happens only if needed. This mechanism is implemented by including in the *ReactiveObject* a set of validity flags for 1) each property of the element and 2) the element itself.

1. *Feature Validity* tells whether a feature in the target model can be directly retrieved (it is up-to-date) or needs to be calculated (was never requested before) or re-calculated (a change on the source model may have affected its value).
2. *Element Validity* indicates whether a target element is up-to-date or is to be deleted/re-computed due to changes in the source model.

Finally, a ReactiveObject notifies its observers when an invalid element or property is accessed, to trigger their re-computation.

Table 1: Main transformation evaluation strategies

Transformation	Propagation	Description	References
eager	no	one-shot transformation	[2, 1]
lazy	no	lazy transformation	[6]
eager	yes	incremental transformation	[5, 9, 10, 11, 12, 13, 14, 15, 16]
lazy	yes	lazy transformation & propagation	-

4. Reactive ATL

4.1. Existing Execution Strategies for ATL

Reactive ATL builds on top of previous results on model transformation, implemented on the ATL transformation engine. We start this section by providing a brief introduction to the execution strategies in the standard ATL engine and related research prototypes. Table 1 summarizes the main approaches, with references to related work. Reactive ATL is meant to fill the last row of Table 1, being the first approach combining lazy transformation and change propagation in model-to-model transformation.

4.1.1. Standard ATL

The standard ATL engine works as a one-shot transformation engine [2]. The ATL execution algorithm launches, in a non-deterministic order, all the *matched rules* in the transformation over matches of their respective source patterns (beside matched rules, two other kinds of rules exist on ATL, lazy rules and called rules, that need to be called from other rules). For a given match the specified target elements are created along with an internal traceability link that stores the relation between a source element, the rule in charge of transforming it and the created target elements.

The initialization of the features of the created target elements relies on the traceability links, following what is called the *ATL resolution algorithm*. When a value is assigned to a given feature, this algorithm acts as follow: 1) if the value is a primitive type, it is directly assigned to the given feature. 2) if the value is a target model element it is also directly assigned to the feature. 3) if the value is a source model element, the corresponding target model element (w.r.t. the traceability links) is assigned to the feature.

Upon the initialization of any transformation, the ATL engine parses a compiled ATL transformation (ASM code) and creates a map of operations (including those operations meant to transform elements and initialize bindings). These operations will then be called from the engine when needed during the transformation process.

4.1.2. Incremental ATL

Incremental ATL [5] builds on the standard ATL engine and introduces the ability of incrementally propagate source model changes to the target model.

This feature is based on the tracking of the expressions that appear in the bindings and filters of ATL transformations to query and compute model elements.

Concretely, an ATL transformation definition uses OCL [17] as an expression language to query and compute model elements. In ATL all the OCL expressions are evaluated on the source model. Any time a source model property changes, OCL expressions used within an ATL rule could be affected. To avoid the recalculation of every OCL expression when these changes happen we need to track which source model properties are used for computing each rule filter or binding. This tracking is achieved by constructing a key-value map at execution time. When an OCL expression is evaluated, we associate it with the source model properties accessed during the evaluation. Then, this map is reversed to ease finding the affected expressions, given the updated property.

4.1.3. Lazy ATL

Fine-grained, on-demand evaluation of ATL transformations was introduced in [6]. In order to generate target models on demand the ability of launching individual rules is needed. However, model-management frameworks allow for a fine-grained access to models, where every call can request a single property of a model element. Exploiting this access pattern, in Lazy ATL we provide fine-grained lazy computation, at the property level. In practice, a mechanism for individual calculation of bindings was included. In summary two new operations were added to the ATL engine:

- *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 explicitly called by subsequent calls to the operation initProperty.

- *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 evaluated 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 trace links (i.e., it has not been transformed), a transformation on that element is launched by a call to transformElement.

4.2. Reactive Transformation Strategy

We build our reactive engine by importing the techniques developed in [5] and [6] and by leveraging on the trace links produced by the transformation at runtime to minimize the amount of work to be performed by the engine. From [5] we take *expression tracking* as a means to perform the dependency tracking needed to efficiently propagate updates. From [6] we take the mechanism of

fine-grained computation activation for launching individual rules and calculate individual bindings.

Once expression tracking and fine-grained computation activation are available, we can introduce a reactive evaluation strategy. In Reactive ATL we propose an **invalidate/lazy recompute** evaluation strategy. The objective is trying to delay computation as much as possible, to avoid unnecessary calculations. For this reason after a source update we: 1) immediately detect and invalidate impacted target elements/properties, operation that can be performed at minimal cost relying exclusively on trace information; 2) re-compute invalid target elements/properties only when explicitly requested by the application.

Elementary updates on elements of the source model are handled as follows:

- *Element modification* triggers the execution of the novel operation *propertyChanged(sourceElement, property)*. When a source element property is changed, two ATL transformation elements can be affected, *bindings* and *filters*. In the case of bindings, the operation reads the *property* $\rightarrow$ *expression* map and sets the affected target element features as invalid. In the case of filters, the corresponding target elements (i.e., the elements created by that rule and retrieved thanks to the traceability links) are set as invalid.

- *Element creation* affects the transformation as it modifies the containment feature of the element intended to hold it, thus, any element creation event is managed as an element modification event on the containment feature.

- *Element deletion* triggers the execution of a newly added operation *elementDeleted(sourceElement)* on the corresponding source element. The operation invalidates the corresponding target elements and recursively all its contained elements.

On the other side, requests on the elements of the target model are handled in the following way:

- *Requesting a valid feature of a valid element.* The feature is simply returned and no computation is activated.

- *Requesting an invalid feature of a valid element.* When the application requests an invalid feature from a target model element the proper transformation engine operations (i.e., the binding computation and resolution) are launched to recalculate the updated values as if the feature had never been computed. Then the property is revalidated.

- *Requesting a feature of an invalid element.* If the application tries to access the features of an invalid model element an exception is thrown. From an invalid element we only allow the application to request its container, to leave the possibility to reach a valid element by traversing upwards the containment tree.

4.3. Input/Output Equivalence

Given a sequence of update and request operations from the user application, requests return in the reactive execution strategy the same results as in the three execution strategies in previous work. We say that the four execution strategies are *input/output equivalent*. More precisely:

1. If no updates are performed to the source model, then any sequence of valid requests on the target model will return the same sequence of results in any of the four execution strategies.
2. If updates are performed to the source model, then after such updates the Incremental and Reactive execution strategies will respond with the same sequence of results to any sequence of valid requests on the target model. The Standard and Lazy execution strategies will also respond with the same results, if the transformation is relaunched after the updates (and before the requests).

In other words, beside avoiding the necessity to manually relaunch the transformation after updates, the reactive strategy computes the same target model elements of the previous alternatives. Of course such computation is delayed until needed, or not performed at all if never requested.

While we do not currently have a formal proof of such equivalence in our engines, we verify it by formalizing the four execution algorithms described in the previous sections and automatically checking the equivalence properties over a large set of automatically generated inputs. In practice, we represent the execution strategies as concise functions in Haskell. Properties are then verified by using Haskell QuickCheck[8], a library for random testing of program properties. QuickCheck provides combinators to define properties, observe the distribution of test data, and define test data generators. We provide QuickCheck with the specification of properties (1) and (2). Then QuickCheck tests that the properties hold in a large set of randomly generated cases.

We provide details about the structure of our Haskell formalization and a link to the full, freely available verification setup, in Appendix A.

4.4. Reactive Higher Order Transformations

Building on top of the reactive facilities, we provide support for efficiently handling the modification of the transformation specification (i.e. a higher-order transformation, a transformation of transformations) in a live transformation system. The transformation code is internally represented as a reactive EMF model (i.e. the so-called *transformation model*). Notifications coming from this model represent transformation changes[9]. Our *reactive framework* responds to

[8] `https://hackage.haskell.org/package/QuickCheck`

[9] When transformation updates come from a textual editor, since we do not have an incremental ATL parser, the updated ATL code is fully parsed at every edit. Updates to the transformation model are then computed by differencing the old and updated models. Given the limited size of the transformation code this does not pose performance issues.

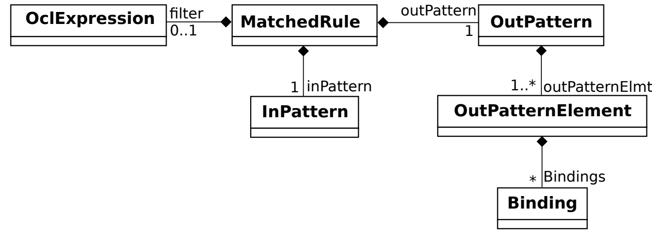


Figure 5: ATL Metamodel Excerpt

these notifications by performing three actions: *change analysis*, *transformation adaptation*, and *change propagation*.

- *Change analysis.* Once a change notification is received, the updated transformation model is analyzed so that we obtain a list of changes classified as *Binding* or *Rule* changes.
- *Transformation adaptation.* After detecting what has changed from one version of the transformation to another, the running live transformation is adapted so that new transformation operations (rule and binding calculations) conform to the updated transformation specification.
- *Change propagation.* Once the running transformation has been adapted to integrate the changes introduced in the source code, a change propagation step is required in order to keep the target model consistent w.r.t. the transformation specification. The propagation operations depend on the kind of the transformation change to be propagated.

We show in Figure 5 an excerpt of the ATL metamodel where the possible changing elements are depicted. Basically, a rule is composed of an *Input Pattern*, a (optional) *Filter* and an *OutputPattern*. The *OutputPattern* is in turn composed of *OutputPatternElements* containing *Bindings* used to initialize target element features. Any change directly affecting the *Input Pattern*, *Filter* or *OutputPattern* elements (or the rule itself, e.g., when adding or removing a rule) is classified as a rule change, as changes in these elements will modify either the set of matched source elements (including their type) or the type of the output element. Conversely, changes affecting a *Binding* element are classified as binding changes as they only change the expression used to initialize a given feature of the output element created by the rule without affecting its type.

Thus, the management of these two kinds of changes, summarized in Table 2, is performed as follows:

- *Binding change propagation.*

  First, starting from the rule containing the modified binding, all model elements of the type of the target model element holding the modified property are retrieved from the stored traceability links. Notice that this

Change	**Propagation Action**
Binding change	Property Invalidation: the updated property is invalidated for each output model element of the modified rule.
Rule change	For *InputPattern, Filter and OutputPatter changes or Rule deletion:* • Element Invalidation: each output element produced by the rule is invalidated. • Cross-references invalidation: references to the type of the invalidated elements are invalidated. For *Rule addition:* Cross-references invalidation: references to the type of the newly created elements are invalidated.

Table 2: Handling transformation changes

operation in not expensive, as the traceability link set class stores maps allowing to retrieve the traceability links produced by a given rule. Then, for each retrieved target model element, the feature flag corresponding to the modified property is set to invalid.

- *Rule change propagation.*

  InputPattern, OutputPattern and Filter changes are processed by using the traceability links to retrieve affected target model elements and setting their validity flag to false. Furthermore, we find target elements that reference elements of the type of the invalidated target element and set their referencing feature to invalid, so that the feature is recalculated in the next access.

  Rule deletion causes the invalidation of the target elements of the deleted rule and potential incoming references, analogously to what we have described above for the other rule changes. The reference re-computation happens by the standard reactive mechanism when the target model is navigated.

  Rule Addition does not require invalidation of existing elements. We only invalidate the references in the target model that may point to elements of the output types of the new rule.

5. Tool Support

In this section we illustrate our reactive implementations of EMF and ATL, namely `Reactive ATL`[10], that compose our reactive transformation system.

As a standard solution for reactive models under Eclipse, we propose a reactive version of the EMF modeling framework, built by extending the EMF EObject class. This so-called *ReactiveEObject* class allows transformation engines to

[10]https://github.com/atlanmod/org.eclipse.atl.reactive

subscribe to it and be notified according to the observer pattern. Standard EMF already implements a notification mechanism for model-element updates. We extend this behavior to model-element requests, so that subscribers get notified when model elements are requested.

Summarizing, we extend EMF in the following way:

- *Validity flags.* We modify the EObject to make it hold the validity flags described in Section 3.
- *Notifications.* The EMF *EObject* already provides notifications for events related to the setting of properties (setting features, adding elements to containment features or removing elements). However, our approach requires the modelling framework to notify for two other kinds of event: 1) requests of *EObject* features and 2) updates to the validity flags. Being notified about flag modifications makes it possible to implement an invalidation strategy whereas tracking feature requests permits to react to user model exploration on-demand.

  We extended the EMF *Notification* class to represent the two new kinds of notifications and we implemented in *ReactiveEObject* the production of these notifications.

Note that developers can directly use the notification facilities, i.e, to start listening to the mentioned events, by implementing the EMF *EContentAdapter* class providing the handling of the notification events (and thus providing a customized behaviour) and add it to the root of the models to attach it to all the model elements.

The reactive version of the ATL transformation language, has been developed without changing the ATL syntax, but adding reactivity to the language semantics by building on top of expression tracking and fine-grained computation activation features described in 4. For this reason, in building `Reactive ATL` we reused most of the default ATL engine.

6. Evaluation

6.1. Performance Evaluation

We design and evaluate a set of tests in order to compare the computational cost of Reactive ATL w.r.t. three well-known alternative evaluation strategies (i.e., different ways to react to changes and consume the target models): One-Shot (Standard) ATL [2], Incremental ATL [5], Lazy ATL [6].

To clearly highlight the difference in computation time among the different scenarios we simulated an application interaction made of four stages: 1) *Write*, sequential creation, element by element, of a source model; 2) *Read*, traversal of all the correspondent target model, 3) *Update*, sequential update of a single property in every element of the source model; 4) *Read*, second traversal of the full target model. In all the phases we navigate the model following a depth-first traversal of the model containment tree. The first two stages are

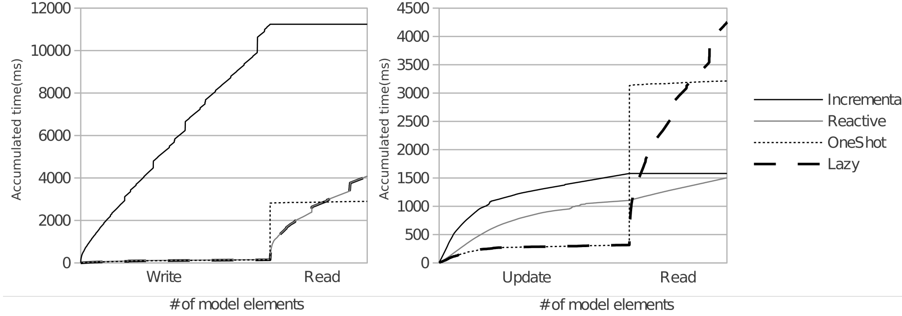


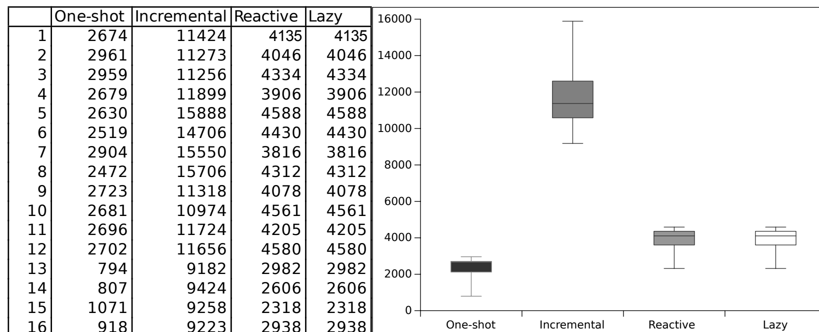
Figure 6: Performance comparison for source model 1

designed to evaluate on-demand computation while the second two stages focus on incremental propagation.

We perform the full experiment 16 times, variating the source model. As source models for the 16 iterations we use randomly generated class diagrams of different sizes. Models are generated by I) creating a random number of packages with a uniform probability distribution between 1 and 30, II) creating in each package a random number of classes and datatypes with a uniform probability distribution between 1 and 120, III) creating in each class a random number of attributes with a uniform probability distribution between 1 and 120. Names of elements are filled sequentially (e.g., Package1, Package2, ...). Attributes are multivalued or not with a uniform probability. Finally attributes are typed with a randomly selected class or datatype. The random generation process produces a set of class models with size up to 48000 model elements. The results of the evaluation[11] are depicted in Figures 6 and 8.

Fig. 6 illustrates the results of executing the four applications on the first random source model. The figure contains one chart for the write and read stages and another one for the update and second read stages. In the first chart, we can see that the *Incremental* scenario consumes a considerable amount of time during the write phase due to dependency calculation, element invalidation, notification and generation. Since the first two phases of the experimentation do not involve updates, *Lazy* and *Reactive* perform identically. In the read stage, the *Lazy* and the *One-shot* scenario not surprisingly perform similarly to [6]. At the end of the first two phases, standard ATL has slightly better computation time, w.r.t. the lazy solutions (reactive or not). However the curves show that the lazy versions perform better when the program needs to access a small part of the target model (see the read sections in Figure 6, showing that the curve for lazy computation crosses the curve of one shot after reading a certain number

[11] The experimentation has been performed in the Eclipse environment using the following hardware and software setting: Ubuntu 11.10, Linux kernel v3.0.0-13-generic-pae, Intel Core i7 processor (2,67 GHz) with HDD.



	One-shot	Incremental	Reactive	Lazy
1	2674	11424	4135	4135
2	2961	11273	4046	4046
3	2959	11256	4334	4334
4	2679	11899	3906	3906
5	2630	15888	4588	4588
6	2519	14706	4430	4430
7	2904	15550	3816	3816
8	2472	15706	4312	4312
9	2723	11318	4078	4078
10	2681	10974	4561	4561
11	2696	11724	4205	4205
12	2702	11656	4580	4580
13	794	9182	2982	2982
14	807	9424	2606	2606
15	1071	9258	2318	2318
16	918	9223	2938	2938

Figure 7: Write & Read total time

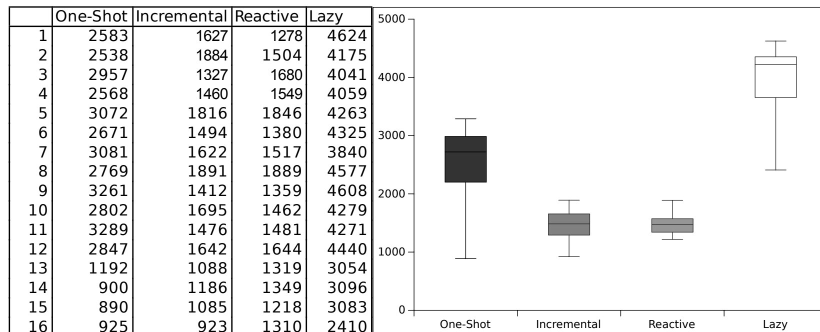
of elements of the target model): for this experiment the threshold is at 55.9% of the target model.

In the second chart of Fig. 6, the update stage modifies a single feature (the name) in every element of the source model. *Lazy* and *One-shot* do not execute any computation during this phase, and the computation time is due only to standard EMF. In the read-again stage, *Lazy* and *One-shot* behave again as in [6]. *Incremental* has good performances, by executing all the computation during the update phase. At the end of these two phases the *Reactive* approach has an excellent result requiring less computation time then all the alternatives, and consistently less computation time than the incremental option. When updates are involved in the model-application, the *Lazy* strategy performs better than *One-shot* until a significative percentage of target model exploration, whereas *Reactive* and *Incremental* (that do not need to re-launch the whole transformation) perform better even when all the source model elements are updated.

Figures 7 and 8 show the results obtained by replicating the experiment with different random source models. Fig. 7 illustrates the total computation times of the Write&Read phase. Fig. 8 summarizes the total computation times of the Update&Read phase.

In absence of updates Fig. 7 shows that *Incremental* is 4 to 12 times slower than the average one-shot application in performing a complete Write&Read. *Reactive* and *Lazy* have identical performance, up to 4 times slower than the average *one-shot* for the complete traversal. However in this case most tests are less than 2 times slower, and we already discussed that these engines are faster for partial traversals. The variability depends on the input model size, with an increase more than linear, especially for *Incremental*. When source updates are considered, as in Fig. 8, *Incremental* and *Reactive* are generally faster than *One-shot*. *Lazy* is up to 4 times slower in the global traversal.

In summary a reactive strategy outperforms other strategies in most situations. The notable exception where the standard engine is faster is navigating a big part of a newly generated model, without source updates. However, also in this case the total computation times are similar.



	One-Shot	Incremental	Reactive	Lazy
1	2583	1627	1278	4624
2	2538	1884	1504	4175
3	2957	1327	1680	4041
4	2568	1460	1549	4059
5	3072	1816	1846	4263
6	2671	1494	1380	4325
7	3081	1622	1517	3840
8	2769	1891	1889	4577
9	3261	1412	1359	4608
10	2802	1695	1462	4279
11	3289	1476	1481	4271
12	2847	1642	1644	4440
13	1192	1088	1319	3054
14	900	1186	1349	3096
15	890	1085	1218	3083
16	925	923	1310	2410

Figure 8: Update & Read total time

6.2. Supported ATL subset and prototype limitations

In this paper we focus on model-to-model declarative ATL, so we postpone to future work the study of a reactive semantics for the parts of ATL that do not fit in this paradigm.

First of all, *Refining mode*, in charge of performing in-place transformations is not supported. Reactivity (i.e., automatically coordinated incrementality and laziness) for in place transformations, as a specific case of model transformation, constitutes a research subject on its own. Nevertheless, note that in place transformations can be simulated by just creating a normal ATL transformation that copies the input model to an output model conforming to the same metamodel.

Secondly, the imperative constructs, i.e., *Called rules* and *do* sections in rules lay out of the scope of the present paper. Reactivity in imperative languages have been largely studied and thus, the results in literature may be integrated to our approach. Moreover, the preferred style while writing ATL transformation is the declarative one, leaving the use of imperative constructs to only specific problems that are too difficult to solve following a declarative style.

Finally some features of standard ATL, while not posing any conceptual difficulties, are either not supported or only partially supported (i.e., they support lazyness but not incrementality) by the current version of our engine. Helpers, Rule Variables, Metamodel-specific operations and the AllInstances() operation can be defined and used as in the standard engine for lazy evaluation scenarios. However, as they are not included in the expression tracking mechanism, it is not safe to use them in scenarios with source updates as they can lead to wrong results. Support for Reflection, Resolution of specific target elements, Rule Inheritance and Multiple Source Elements is, for the moment, not included in the Reactive compiler and therefore, rules and transformations including those features can not be executed.

Our reactive prototype can be used to efficiently handle the development of many application contexts. However, to increase the range of supported scenarios we are investigating the following extensions:

Target editing. As for now, our reactive engine allows editing only in the source

model, while the target model is read-only. This applies also for models participating in a chain, where the middle models gets modified only by the updates of the source model and not directly by users editing it. It would be interesting to provide support for the direct editing of the target model. A complete solution may support the propagation of some changes to the sources models, thus requiring bidirectionality.

Transactional updates. The granularity of changes or requests supported by `Reactive ATL` is at the level of model element or model element property. Hence complex updates are represented as sequences of atomic updates, that may pass by inconsistent intermediate states. While this does not hamper the correctness of the final result, it may trigger unneeded (re-)computation in intermediate steps. It would be interesting to record a set of changes in a transaction, so that they can be safely applied (or rolled back) at once.

7. Related Work

7.1. Outside Model-Driven Software Engineering

Reactive programming is a subject largely studied out of the modelling research community. As an example, the ESTEREL [18] synchronous language, designed to program complex reactive systems and Reactive-C [19], an extension of the C programming language following the same paradigm, reached a high level of popularity. Reactive programming has also been studied in relation to functional languages. [20] studies the semantics of functional reactive programming systems. More recently, the current status of the research on reactive applications is analyzed in [4] and [21] while the latter also provides a roadmap to overcome the limitations of the approach when applied on object-oriented programming languages. Similar to our approach, in [22] the authors present an approach to efficiently evaluate complex queries on object oriented databases by providing an incremental algorithm over collections where updates can be lazily propagated. In a similar way, in [23] the authors present the λ_{ic}^{cdd} calculus and corresponding implementation (in OCaml) where among other features, lazy computation semantics are applied to incremental computation so that input changes are not eagerly propagated. Finally, challenges in applying the reactive paradigm to applications working in distributed environments are analysed in [24].

This work is also inspired by similar efforts in the attribute grammar research community. In [25], the authors describe incremental evaluations through caching visit sequence functions. Moreover, instead of explicitly representing the tree, it is represented through a set of visit functions corresponding to its successive visits. Then, they only visit the parts of the tree that need to be read. This can be regarded as similar to the lazy model presented in this work. In [26], visit functions are used instead of visit sequences in order to achieve efficient incrementality. More recently and for a specific kind of attribute grammars, in [27], the authors present an approach for incremental evaluation based, as in the present work, on cache invalidation. They achieve incrementality by keeping

pointers from a place in the tree where a field is written to the place in the tree where the object being written was created. Then pointers from the creation point to the places where the object are read are also kept. If a change happens, these pointers are used to invalidate the cache.

[28] follows a lazy approach for the evaluation of XSLT. 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.

The implementation of a lazy evaluator for functional (navigation) languages is a subject with a long tradition [29]. We refer the reader to [30] for an example based on Lisp.

7.2. Within Model-Driven Software Engineering

This paper builds upon the two works: [5] and [6], by the same authors, where incrementality and on-demand execution are studied. Here we contribute the integration of those previous features in a single engine, a new invalidate/lazy recompute evaluation strategy and an evaluation of its performance. We also extend previous work to cover the scenario where changes are introduced to the transformation specification itself. Additionally, we provide a formal specification in Haskell showing the input-output equivalence of our reactive approach w.r.t. to the previous existing ones.

Model incremental synchronization has been extensively studied in the modeling community. [9] proposes an automatic way to synchronize the source and target models of an ATL transformation offline. Incrementality is implemented by interfacing with existing differencing tools for calculating changes to the models and propagating them bidirectionally. With respect to their work, the approach followed in [5] requires only limited changes to the ATL compiler and no change to the ATL Virtual Machine (VM), whereas they rely on several VM modifications. We follow here the approach proposed in [5] for tracking OCL expressions in order to calculate the affected target elements of a source model update.

In an alternative approach, Hearnden et al. [10] synchronize two models incrementally, by using a declarative logic-based transformation engine. The approach records a transformation execution and links each transformation step to the correspondent changes in the source model. This information is then used for change propagation.

Live and offline incrementality has been already implemented with Graph Transformations techniques, for example in [12]. Especially the work in [31] implements live incrementality, based on the Rete algorithm, a well-known technique in the field of rule-based systems. These graph transformation approaches focus on incremental pattern-matching to improve the performances of the transformation. In opposition to these graph-based systems, the proposal we follow does not directly apply in-place transformations, but it could be extended for that purpose. In this sense the proposal we are building on is more similar to [13], that employs Triple Graph Grammars for incremental offline model synchronization in both directions.

Regarding on-demand generation and fine grained control of transformation execution, the Stratego [32] system allows user-defined execution strategies for transformation rules, whereas VIATRA evaluates lazily the matchings of connected rules to avoid unnecessary computation, as described in [33]. While user-defined strategies have been used to implement target-driven approaches [34] and some limited lazy evaluation has been provided, the activation of rules as answer to external consumption has been only addressed in [6]. In that mentioned work target model request tracking and caching support is implemented in an ad-hoc way. Here, we tackle the support for these two features in a more generic way and we add support for cache invalidation.

More recently, in [14], [15] and [16] complex event processing (CEP) is explored over the VIATRA transformation framework to 1) give support to the streaming of model transformations by mixing it with incremental queries 2) facilitate the integration between different model-driven operations (transformation, query, etc.) in a source incremental execution scheme directed by trigger events.

Lazy evaluation has been explored for OCL in [35], [36] and in [37] where performance measures are presented. The topic of evaluating OCL expression incrementally has been investigated by Cabot [38], especially for detecting if a modification to a UML model violates constraints that were satisfied before.

8. Conclusions and Future Work

In this work we focused in implementing and evaluating an experimental prototype of reactive transformation engine that can cover a wide range of application scenarios. Our work shows that the use of a declarative MTL like ATL, besides benefits in expressiveness and abstraction, allows developers to easily build autonomous data-flow systems that react to application events, with advantages in terms of development and computation time.

We envision several lines for future work:

Extended support for ATL. Our prototype does not yet implement a complete reactive paradigm for ATL: currently we don't support all the ATL features and we don't have any support for transactional or concurrent updates. Moreover, our reactive engine works in a synchronous way, i.e., when a source model element change is produced or a target model element is requested, the client application must wait until the transformation engine has finished its computation. Conversely, asynchronous computation is likely to be useful when dealing with updates affecting a big subset of the target models or when the request for a feature involves heavy computations. We plan to study these aspects in future work.

Extensions to the paradigm. In the context of model transformations, the reactive paradigm can be further extended with advanced features like back-propagation and support for target updates (the first steps to provide a bidirectional engine for ATL transformations are already in an advanced state)

and retainment rules. Furthermore, while our reactive engine already supports transformation chains, we plan to study the possibility of extending it for the case of complex transformation networks. Transformation networks with several input and output models and with possibly different execution paths for any given task are challenging to manage efficiently and thus, we believe they can benefit from our reactive engine.

Infinite and streaming models. A reactive approach opens the way to scenarios based on infinite intermediate models generated on demand, or streaming models propagating from inputs to outputs. This research could widen the application space of the model-driven approach.

Acknowledgements

This work is partially supported by the MONDO (EU ICT-611125) project.

References

[1] OMG, MOF QVT Final Adopted Specification, Object Management Group (2005).

[2] F. Jouault, I. Kurtev, Transforming Models with ATL, in: MoDELS Satellite Events, 2005, pp. 128–138.

[3] D. Harel, A. Pnueli, On the development of reactive systems, Springer-Verlag New York, Inc., New York, NY, USA, 1985, pp. 477–498.

[4] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, W. d. Meuter, A survey on reactive programming, ACM Computing Surveys (CSUR) 45 (4) (2013) 52.

[5] F. Jouault, M. Tisi, Towards incremental execution of ATL transformations, in: L. Tratt, M. Gogolla (Eds.), ICMT, Vol. 6142 of LNCS, Springer, 2010, pp. 123–137.

[6] M. Tisi, S. Martínez, F. Jouault, J. Cabot, Lazy execution of model-to-model transformations, in: Model Driven Engineering Languages and Systems, Models'11, Springer, 2011, pp. 32–46.

[7] D. Smith, An introduction to building information modeling (BIM), Journal of Building Information Modeling 2007 (2007) 12–14.

[8] ISO, Industry Foundation Classes (IFC) for data sharing in the construction and facility management industries, ISO 16739:2013, International Organization for Standardization, Geneva, Switzerland (2013).

[9] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, H. Mei, Towards automatic model synchronization from model transformations, in: IEEE/ACM international conference on Automated software engineering ASE'07, ACM, 2007, pp. 164–173.

[10] D. Hearnden, M. Lawley, K. Raymond, Incremental model transformation for the evolution of model-driven systems, in: Model Driven Engineering Languages and Systems, Models'06, Springer, 2006, pp. 321–335.

[11] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, G. Varró, Incremental pattern matching in the VIATRA model transformation system, ACM Press, New York, New York, USA, 2008. doi:10.1145/1402947.1402953.

[12] G. Bergmann, I. Ráth, D. Varró, Parallelization of graph transformation based on incremental pattern matching, Electronic Communications of the EASST 18.

[13] H. Giese, R. Wagner, From model transformation to incremental bidirectional model synchronization, Software & Systems Modeling 8 (1) (2008) 21–43. doi:10.1007/s10270-008-0089-9.

[14] I. Dávid, I. Ráth, D. Varró, Streaming model transformations by complex event processing, in: Model-Driven Engineering Languages and Systems, Springer, 2014, pp. 68–83.

[15] G. Bergmann, I. Dávid, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, D. Varró, Viatra 3: A reactive model transformation platform, in: Theory and Practice of Model Transformations, Springer, 2015, pp. 101–110.

[16] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, Road to a reactive and incremental model transformation platform: three generations of the viatra framework, Software & Systems Modeling (2016) 1–21.

[17] OMG, 2.0 OCL specification, Adopted Specification (ptc/03-10-14).

[18] G. Berry, G. Gonthier, The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation, Science of Computer Programming 19 (2) (1992) 87–152.

[19] F. Boussinot, Reactive c: An extension of c to program reactive systems, Software: Practice and Experience 21 (4) (1991) 401–428.

[20] Z. Wan, P. Hudak, Functional reactive programming from first principles, in: ACM SIGPLAN Notices, Vol. 35, ACM, 2000, pp. 242–252. doi:http://doi.acm.org/10.1145/358438.349331.

[21] G. Salvaneschi, M. Mezini, Reactive behavior in object-oriented applications: An analysis and a research roadmap, in: Proceedings of the 12th annual international conference on Aspect-oriented software development, ACM, 2013, pp. 37–48.

[22] H. Nakamura, Incremental computation of complex object queries, in: ACM SIGPLAN Notices, Vol. 36, ACM, 2001, pp. 156–165.

[23] M. A. Hammer, K. Y. Phang, M. Hicks, J. S. Foster, Adapton: Composable, demand-driven incremental computation, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, ACM, New York, NY, USA, 2014, pp. 156–166. doi:10.1145/2594291.2594324.

[24] A. Margara, G. Salvaneschi, We have a dream: distributed reactive programming with consistency guarantees, in: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems, ACM, 2014, pp. 142–153.

[25] M. Pennings, S. D. Swierstra, H. Vogt, Using cached functions and constructors for incremental attribute evaluation, in: M. Bruynooghe, M. Wirsing (Eds.), PLILP, Vol. 631 of Lecture Notes in Computer Science, Springer, 1992, pp. 130–144.

[26] H. Vogt, S. D. Swierstra, M. F. Kuiper, Efficient incremental evaluation of higher order attribute grammars, in: PLILP, 1991, pp. 231–242.

[27] J. Boyland, Incremental evaluators for remote attribute grammars, Electr. Notes Theor. Comput. Sci. 65 (3) (2002) 9–29.

[28] S. Schott, M. L. Noga, Lazy XSL transformations, in: ACM Symposium on Document Engineering, ACM, 2003, pp. 9–18.

[29] P. Hudak, J. Hughes, S. L. P. Jones, P. Wadler, A history of Haskell: being lazy with class, in: HOPL, ACM, 2007, pp. 1–55.

[30] P. Henderson, J. H. Morris, Jr., A lazy evaluator, in: Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages, POPL '76, ACM, 1976, pp. 95–103.

[31] I. Ráth, G. Bergmann, A. Okrös, D. Varró, Live model transformations driven by incremental pattern matching, Theory and Practice of Model Transformations 5063 (2008) 107–121.

[32] E. Visser, Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9, in: Domain-Specific Program Generation, Vol. 3016 of LNCS, Springer, 2003, pp. 216–238.

[33] G. Taentzer, K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, D. Varró, S. Varró-Gyapay, Model transformation by graph transformation: A comparative study, in: Proc. Workshop Model Transformation in Practice, 2005.

[34] J. V. Wijngaarden, E. Visser, Program transformation mechanics: A classification of mechanisms for program transformation with a survey of existing transformation systems, Tech. rep., UU-CS (2003).

[35] O. Beaudoux, A. Blouin, O. Barais, J.-M. Jézéquel, Active operations on collections, in: MoDELS, Vol. 6394 of LNCS, Springer, 2010, pp. 91–105.

[36] M. Tisi, R. Douence, D. Wagelaar, Lazy evaluation for ocl, in: OCL 2015–15th International Workshop on OCL and Textual Modeling: Tools and Textual Model Transformations Workshop Proceedings, 2015, p. 46.

[37] M. Clavel, M. Egea, M. A. G. de Dios, Building an efficient component for OCL evaluation, ECEASST 15.

[38] J. Cabot, E. Teniente, Incremental evaluation of OCL constraints, Lecture Notes in Computer Science 4001 (2006) 81.

Appendix A. Haskell formalization

In this appendix we describe the general structure of our Haskell formalization of the four execution algorithms described in the paper (namely, Standard, Incremental, Lazy and Reactive ATL) and its input/output equivalence verification with QuickCheck.

The full Haskell code of the formalization is freely available[12].

- A Model is defined as:

  ```
  type Model = (Element,SetOf Element,SetOf Link)
  ```

  including a root element, a set of model elements and a set of links, with

  ```
  type Link = (Element,Element)
  ```

  This definition of Model is a significant simplification w.r.t. an EMF model. In practice we focus only on the graph structure of the model to verify that different semantics behave equivalently with any model topology.

- Independently from the execution strategy we formalize a Transformation system as a set of two models and a transformation. For instance:

  ```
  newtype TransformationReactive =
     TransformationReactive (Model, Transformation, Model)
  ```

  where a transformation is a binary relation:

  ```
  type Transformation = SetOf (Element,Element)
  ```

  The functions provided by a Transformation System are grouped in a type class:

  ```
  class TransformationI ts where
      apply :: ts -> ts
      addElementToSource :: Element -> ts -> ts
      addLinkToSource :: Link -> ts -> ts
      getFromTarget :: ts -> Element -> (SetOf Element,ts)
      ...
  ```

  Functions like `addElementToSource` and `addLinkToSource` represent updates to the source model while functions like `getFromTarget` represent requests from the target. All functions in the class return an updated state for the system, where the side effects of updates or requests have been applied. For instance, the class includes the `apply` function that given a transformation system in input returns an updated state for the system, where the transformation has been applied from the source model to the target model.

[12] https://github.com/atlanmod/org.eclipse.atl.reactive.haskell_semantics

- A function `bindingApplication` represents the computation of expressions and its calls are the key of each execution strategy:
  - in a `TransformationStrict` system the function is called by an iteration in the `apply` function.
  - in a `TransformationIncremental` system the function is called by each execution of the `-ToSource` functions.
  - in a `TransformationLazy` system the function is called by each execution of the `-FromTarget` functions.
  - in a `TransformationReactive` system the function is called by the `-FromTarget` function, but only when the requested link is invalid (otherwise the existing link is simply returned).

Finally we let QuickCheck validate the equivalence of the specifications by generating instances of `Model`, `Transformation`, and sequences of `Element` and `Link` to use in updates and requests.