Experimentation with a Big-Step Semantics for ATL Model Transformations

Artur Boronat

Department of Informatics, University of Leicester
aboronat@le.ac.uk

Abstract. Formal semantics is a convenient tool to equip a model transformation language with precise meaning for its model transformations. Hence, clarifying their usage in complex scenarios and helping in the development of robust model transformation engines. In this paper, we focus on the formal specification of a model transformation engine for the declarative part of ATL.
We present an implementation-agnostic, big-step, structural operational semantics for ATL transformation rules and a rule scheduler, which form the specification of an interpreter for ATL. Hence, avoiding a complex compilation phase. The resulting semantics for rules enjoys a compositional nature and we illustrate its advantages by reusing an interpreter for OCL. The semantics discussed has been validated with the implementation of an interpreter in Maude, enabling the execution of model transformations and their formal analysis using Maude's toolkit. We also present an evaluation of the interpreter's performance and scalability.

Keywords: ATL, big-step semantics, model transformation, Maude.

1 Introduction

Model(-to-model) transformation is a core asset in model-driven engineering (MDE) to manage complexity in scenarios such as DSL development, code generation, system interoperability and reverse engineering [5]. Demonstrating the reliability of such model transformations and of the generated software is a crucial step for MDE to succeed. Recent surveys [1,2] provide an outline of verification techniques applied to model transformations, ranging from lightweight approaches based on testing to automated and interactive theorem proving.

Our goal in this paper is to provide a formalization of ATL [9] using a big-step structural operational semantics that equips model transformation rules with a precise semantics that is independent of any implementation language. This semantics is nonetheless geared for functional programming languages with referential transparency and immutability. We focus our study on out-place ATL model transformations with one target domain and declarative rules, including OCL, `resolveTemp` expressions, matched and lazy rules. The formalization also includes a rule scheduler with small-step operational semantics. The whole approach has been validated by implementing an interpreter prototype in Maude [7]

that follows the semantics specification faithfully. The interpreter allows for the analysis of ATL model transformations using Maude's verification toolkit, such as bounded model checking of invariants. The interpreter has been validated with a representative case study of out-place model transformations [10], which is also helpful to compare its performance and scalability against other approaches.

Maude has already been used as a rewriting engine for executing declarative model transformation rules [4,14] where models are encoded as configurations of objects using an associative, commutative binary top symbol with an identity, and model transformation rules are encoded as conditional rewrite rules. That is, models become graphs when the term representing the configuration of objects is interpreted modulo associativity and commutativity, and rewrite rules operate on the whole models. As already observed in [14], this approach does not scale very well for large models ($\sim$7K$-$10K objects) in out-place model transformations.

Our semantics of ATL relies on Focussed Model Actions (FMA) [3], a DSL modelling the typical side effects found in a model transformation (object creation and destruction, and setting and unsetting structural features − which can be attribute values, and possibly bidirectional cross-references and containments) in functional contexts where models can be either represented as sets of objects or as sets of nested objects. Our semantics (FMA-ATL) allows for an alternative lower level encoding of ATL model transformations in Maude that decouples the query mechanism from the model transformation part in a model transformation rule. We analyse its performance and show that it scales better than previous Maude-based approaches in certain scenarios, allowing for the verification of a larger class of out-place ATL model transformations.

In what follows, we present: the case study that has been used to validate the interpreter; the FMA-ATL semantics, including support for OCL, matched and lazy rules, and resolveTemp expressions; performance and scalability experiments based on the FMA-ATL interpreter for the case study; a detailed comparison with a previous Maude-based formalization of ATL; and final remarks including lessons learnt from building an interpreter for ATL.

2 Case study

We have borrowed the case study [10] that transforms class diagrams into relational schemas as it is representative of out-place model-to-model transformations and it facilitates a comparison of results with other approaches that implement a semantics for ATL in Maude. The metamodels have been extended by adding the meta-classes *Package*, which contains classifiers, in the metamodel *Class*, and by adding the meta-class *Database*, which contains tables, in the metamodel *Relational*, as shown in Fig. 1. An additional rule has been included to transform packages into databases, where the tables generated for the classes in the package are contained by the generated database.[1]

[1] The other ATL rules used in the case study are available at [10] or by accessing the experiment resources at `https://github.com/arturboronat/fma-atl-experiments`.

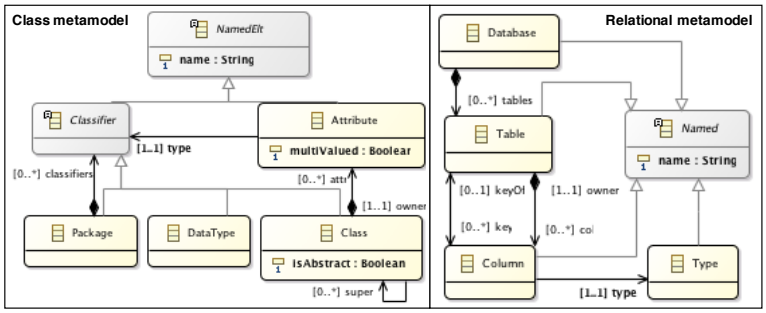


Fig. 1. Metamodels.

```
rule Package2Database { from p : Class!Package
  to out : Relational!Database ( name <- p.name,
    tables <- p.classifiers->select(c | c.oclIsKindOf(Class!Class)) -> union(
      p.classifiers ->select(c | c.oclIsKindOf(Class!Class))
      ->collect( c | c.attr )->flatten()
      ->collect(c | thisModule.resolveTemp(c,'out')))}
```

We have developed an additional version of the Class2Relational transformation to discuss various aspects of our approach. In this version, all the matched rules are declared as lazy but for the rules that transform data types and packages. The rule that transforms packages calls the lazy rules that transform classes and multivalued attributes into tables as follows:

```
rule Package2Database { from p : Class!Package
  to out : Relational!Database ( name <- p.name,
    tables <- p.classifiers->select(c | c.oclIsKindOf(Class!Class))
      -> collect( c | thisModule.Class2Table( c ) )
      -> union( p.classifiers -> select(c | c.oclIsKindOf(Class!Class))
        -> collect( c | c.attr ) -> flatten()
        -> select( a | a.type.oclIsKindOf(Class!DataType) and a.multiValued )
        -> collect( a | thisModule.MultiValuedDataTypeAttribute2Column(a) )
      ) -> union (
        p.classifiers -> select(c | c.oclIsKindOf(Class!Class))
        -> collect( c | c.attr ) -> flatten()
        -> select( a | a.type.oclIsKindOf(Class!Class) and a.multiValued )
        -> collect( a | thisModule.MultiValuedClassAttribute2Column(a) )))}
```

The rule that transforms classes into tables calls the rules that transform single valued attributes into columns as follows:

```
lazy rule Class2Table { from c : Class!Class
  to out : Relational!Table ( name <- c.name,
    col <- Sequence {key} -> union(
          c.attr -> select( a |
            a.type.oclIsKindOf(Class!DataType) and not a.multiValued
          ) -> collect( a | thisModule.DataTypeAttribute2Column(a) )
        ) -> union (
          c.attr  -> select( a |
            a.type.oclIsKindOf(Class!Class) and not a.multiValued
          ) -> collect( a | thisModule.ClassAttribute2Column(a) )),
    key <- Set {key}),
  key : Relational!Column (
    name <- 'objectId', type <- thisModule.objectIdType )}
```

3 FMA-ATL: an Interpreter for ATL

The big-step semantics of out-place ATL transformation rules is developed by using FMA, a DSL modelling the typical side effects performed by a model transformation. Our formalization covers a subset of declarative ATL, namely: matched and (non-unique) lazy rules, one *in-pattern* element, several *out-pattern* elements, filters, OCL and resolveTemp expressions, and helpers.

Given an out-place ATL model transformation, the FMA-ATL interpreter parses matched/lazy rules and helpers, initializing the interpreter configuration. This includes the computation of attribute helpers, caching their result. It then computes all enabling matches for matched rules, by considering their *in-target* element and their filter condition. This is achieved by retrieving all the instances of the class involved in the in-pattern element and by evaluating the filter condition for each instance using an OCL select expression.[2] Then the scheduler starts the model transformation by selecting one enabling match and the corresponding ATL matched rule. The execution of a matched rule involves the interpretation of both a FMA statement representing the side effects in the target model and of a trace statement that instructs what trace links need to be created in the trace store. After these side effects are applied, the scheduler updates the pool of pending matches by disabling those matches in which the transformed object was participating and continues the execution with the next enabling match.

Our formalization covers most of the out-place ATL model transformations expressible in the original ATL but it also introduces a number of constraints trading a small subclass of ATL model transformations for better scalability: when executing a matched rule for a given object, the source model where OCL expressions, appearing in binding statements of the *out-pattern* elements, are evaluated is reduced to the root container of the matched object;[3] a lazy rule can only be called under a containment reference and the result must be a collection of objects whose type must be compatible with that of the containment reference;[4] and `resolveTemp` expressions in a lazy rule can only be resolved with objects produced by matched rules other than the one calling that lazy rule.

By using big-step structural operational semantics for ATL rules, each matched rule application corresponds to one big transition containing all the computations associated with the fine-grained side effects that are applied both to the target model and to the trace store. When verifying the correctness of model transformations, this is helpful for keeping the state space isomorphic to the one that is generated when using normal declarative rules, in a graph transformation

[2] The extension to several *in-pattern* elements does not necessarily affect the order of magnitude of the performance obtained in the experiments in Section 4. In practice, matching different in-pattern elements in the model consists of independent queries together with the evaluation of the filter condition, whose cost may become the dominating factor.

[3] This constraint does not apply to OCL expressions in the filter condition.

[4] Lazy rule recursion is conditioned by the containment structure specified in the metamodel, including infinite recursive structures, as in the composite design pattern.

sense. Next, we explain the phases used by the FMA-ATL interpreter to execute an ATL transformation after an introduction to FMA.

3.1 Focussed Model Actions (FMA)

FMA constitutes a formalization of the typical model actions that can be found in the EMF API for manipulating models at the object level in order to implement model transformations in functional programming languages. Specifically, it constitutes an abstraction layer to implement the side effects of model transformations in declarative languages like Maude where the actions of a rewriting rule are represented in a term with variables in the right-hand side of the rule together with equationally-defined functions. Thus, one does not have to manually deal with different combinations of term patterns for each type of model action in order to make sure that the model is left in a consistent state (e.g. without dangling edges, containment integrity, etc) after the transformation.

FMA is not Turing complete and it only includes two types of statements: model actions **ActStmt** including `create`, `set`, `setCmt`, `unset`, `let`-binding, the `snapshot2` operator, sequence with the separator symbol ';' and the no-op symbol $*$; and FMA statements **Stmt**, including `create`, `delete`, sequence with the separator symbol ';', the no-op symbol (), `let`-binding and the operation `snapshot` x $\{s2\}$ for an object variable x and a model action *s2*. In FMA, the `snapshot` operator is used to focus the interpreter on an object for manipulating it locally by applying the sequence *s2* of model actions to the set of properties of the object referenced by x, reducing the amount of model traversals required to apply each model action. The `snapshot2` operator enables updates in contained objects of the object under focus without having to invoke `snapshot` again. FMA also provides support for declaring procedures and for evaluating expressions, which include values (String, Int, Bool, references to objects of a particular type), variables and `let`-binding and procedure calls. While FMA expressions evaluate to a value, FMA statements apply model actions to a model and they evaluate to the corresponding no-op symbol.

The following example is a FMA statement representing the side effects of the rule `DataType2Type`, in which an object of type `Type` is created and its attribute `name` is set to the result of evaluating the OCL expression `DT . name` where `DT` is the object variable matched in the left-hand side of the rule.

```
let T = create(Relational ! Type) in () ;
snapshot (T){
  set(name, (DSL#String) DT . name)
}
```

The semantics of the FMA interpreter consists of two main components: the interpreter configurations (the interpreter state together with the statement being evaluated) for the two types of statements and for expressions; and big-step semantic rules describing its behaviour. Fig. 2 describes the configuration classes of the FMA interpreter for FMA statements **Stmt**, for model actions **ActStmt**, and for FMA expressions. Each configuration class has a subclass describing what the final configurations are for each type of statement and expression.

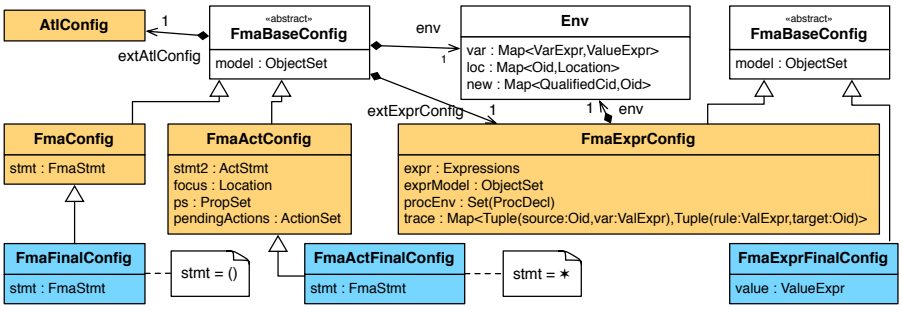
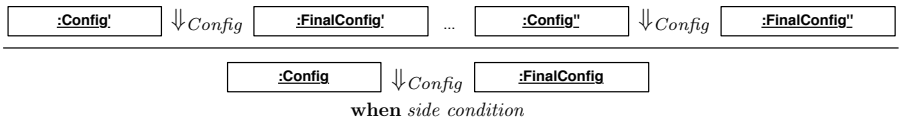


Fig. 2. FMA configurations.

Model action configurations `FmaActConfig` contain the statement `stmt2` to be interpreted, the `focus` location of the object being manipulated, the set of properties `ps` (attributes, references and containments) of the object under focus and a set `pendingActions` of pending model actions for setting non-containment opposite references. Pending actions are used for updating opposite references automatically when the other end is set. Final configurations of the interpreter for model actions are given by those states whose statement is the no-op symbol $*$. Expression states contain the model `exprModel` to be queried, an environment `procEnv` of procedure declarations, the expression `expr` to be evaluated, a trace map `exprTrace` to be used for the ATL expression `resolveTemp`, and an environment with a substitution for the free variables in the expression `expr` and with the location store attaching a location in the model `exprModel` to each object identifier. Final configurations for FMA expressions only contain a `value`.

Big-step rules specify an evaluation relation for a particular type of configurations and are defined in terms of transitions from a configuration of a particular type of statements or expressions to a final configuration of the corresponding type. Transition rules are represented using the following form:



The transition appearing in the conclusion (below the bar) is valid if the transitions appearing in the premise of the rule (above the bar) also hold and if the side conditions `when`, placed below the rule for the sake of readability, are satisfied. A side condition is a conjunction of boolean predicates, assignments `T=F(`T_1`,..,`T_n`)` involving n-ary functions F, and transitions from other evaluation relations as we will see in the next sections. Side conditions are evaluated from left to right, a boolean predicate is satisfied when there is a valid substitution for its variables, an assignment is satisfied when the function returns a value with

the shape indicated in the term `T` (usually assigning values to variables appearing in `T`)[5] and a transition is satisfied when the right-hand side of the transition can be reached (possibly initializing variables) by using transition rules for the corresponding configuration type. In the case of transitions, these will always be *one-step* transitions in practice owing to the design of the big-step rules. A rule without any premise is an axiom and simply asserts a fact, i.e. that a transition from a configuration to a final configuration holds if the side conditions are satisfied as explained above.

As an example, we define the semantic rule that defines the integration of the mOdCL [11] interpreter into the FMA interpreter for evaluating OCL expressions. OCL expressions can be defined in FMA using the FMA expression `(T:Scalar) OCL:OclExp`, which indicates the type of the OCL expression being used and the OCL expression `OCLE:OclExp` using mOdCL syntax. The semantic rule `E-OclExp` in Fig. 3 evaluates such an expression by calling the mOdCL interpreter with: the OCL expression, the model `MODEL` to be queried, the variable store and the location store from the environment. For the integration to work correctly we have also had to extend some of mOdCL operations with a few equations specifying the behaviour of the `allInstances()` expression and of navigation expressions using attributes.

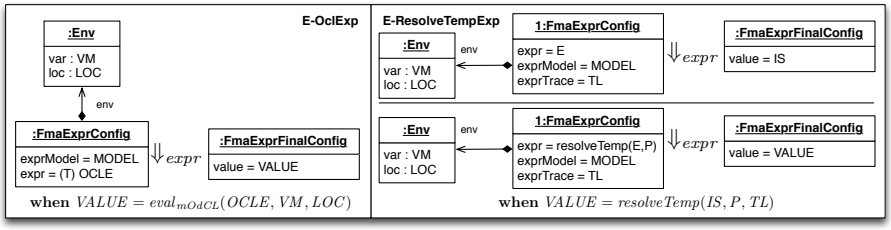


Fig. 3. FMA-ATL semantics of OCL and `resolveTemp` expressions.

In the following subsections, we describe the following extensions to FMA in order to provide support for ATL. Procedure declarations are extended with ATL helpers; FMA expressions are extended with helper calls and with `resolveTemp` expressions; and FMA statements are extended with lazy rule calls.

3.2 Initialization

To execute a model transformation, the FMA-ATL interpreter first initializes the interpreter state from the transformation declaration, loading each rule into a rule store and helpers into a helper store. Once rules are available in the store

[5] This notation is a convenient representation both for executing a function and for decomposing its results with projections. Hence, unification is not required.

it computes all matches for non-lazy rules, as ATL does, but without creating any objects. This allows FMA-ATL to work with declarative rules that have causal dependencies among them when the expression `resolveTemp` is used in the variable bindings of an ATL rule.

Rule initialization. When an ATL rule is initialized, FMA-ATL obtains an FMA statement that represents the model transformation to be applied in the target model from the variable bindings in the target pattern elements. Additionally, it also obtains a trace statement that indicates what trace links should be added to the trace store. This initialization is performed by extracting a graph from the set of target pattern elements where nodes are FMA expressions and where there are two types of named edges: reference edges and containment edges. When defining an edge, a source node is always a variable expression referring to a variable appearing in the target pattern elements of the rule.

This initialization process is illustrated in Fig. 4. For each target pattern element, FMA traverses its binding statements introducing: a) a node with a variable expression referring to the object being created and b) either a reference edge, when the binding corresponds to an attribute or to a non-containment reference, or a containment edge, when the binding corresponds to a containment reference. When defining containment edges, variables corresponding to a target element are extracted from OCL expressions so that containments to objects created in the target pattern elements are represented as containment edges between variable expressions whereas containments to objects created by other matched rules are represented as containment edges to an FMA expression.

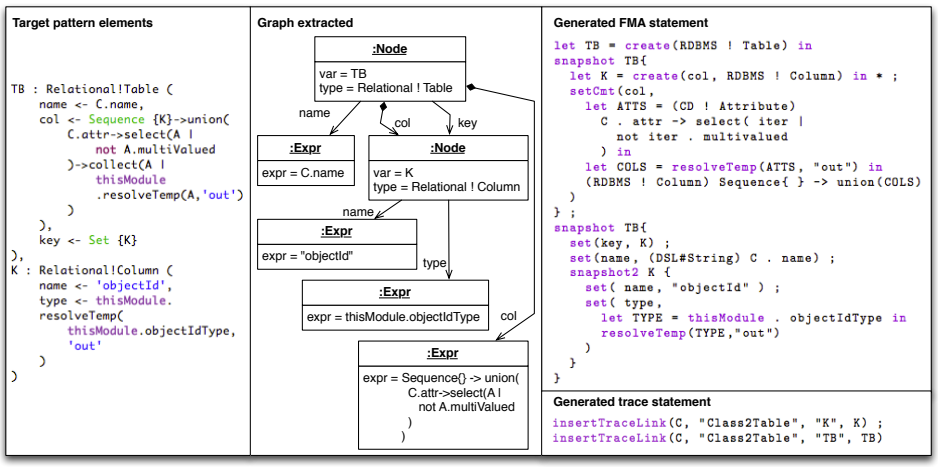


Fig. 4. Translation of Class2Table ATL target pattern elements.

Once the graph is generated, FMA-ATL walks it from its root objects following containment edges twice: first, it obtains an FMA statement that creates a tree of objects that initializes their containment references; second, for each object created in the first traversal, it initializes their attributes and non-containment references as mandated by the rule. Finally, FMA-ATL traverses the nodes that are variables and obtains a trace statement as a sequence of insertions of trace links into the trace store. The free variables used in the trace statement are those that are initialized in the execution of the FMA statement.

Helpers. The helper store is initialized with the helpers defined in the transformation. Helpers with context are simply copied into the helper store but attribute helpers are computed and FMA-ATL caches their value at declaration time by introducing their value together with the attribute helper.

3.3 Scheduling

In this section, we describe the behaviour of the FMA-ATL interpreter and the semantics of ATL matched rules. We start by introducing the main configuration types and we then continue by describing the main rules that specify the behaviour of the FMA-ATL interpreter and the big-step semantics of matched rules. Both the configuration model of the FMA-ATL interpreter and the rules mentioned above are depicted in Fig. 5, where small-step and big-step transition rules are denoted by $\Rightarrow$ and $\Downarrow$, respectively. In the representation of semantic rules, the notation has been simplified by collapsing references to collections of objects in an attribute whose value is a set of objects.

Configurations of the FMA-ATL interpreter have: a `ruleStore` and a `helperStore` with the set of ATL rules and the set of helpers, respectively, that are defined in the transformation; a `queryDomain` pointing to the domain that contains the source model and a set of `domains` that correspond to the different target models that are created by the transformation; a set of `resolveTempClasses` that corresponds to the type names of those classes whose objects may be involved in `resolveTemp` expressions; a `globalTrace` map including all the trace links that are generated by the model and a `localTrace` map that contains the trace links of those objects that may participate in `resolveTemp` expressions − that is, those objects that are instance of a class involved in a `resolveTemp` expression. Each domain contains a `model`, a location store `loc` giving the locations for each object in that model and a factory `new` of fresh identifiers, which is used to create new objects. In addition, there are three specialized types of configurations: `AtlMatchingConfiguration`, used in the rule `E-Schedule` and which has a pointer to a pool of enabled matches; and the corresponding type of final configurations defined as those whose pool of enabled matches is empty; `AtlConfiguration`, used in the rule `E-RuleSideEffects` and which has a pointer to a specific match; and the corresponding type of final configurations defined as those with no match; `AtlDomainConfiguration`, used in the rule `E-DomainActions` and which has an environment of variables containing the match, and a set of domain actions corresponding to those defined in the rule

that is being applied; and the corresponding type of final configurations defined as those with an empty set of domain actions.

Once the pool of matches has been computed and linked to an object `AtlMatchingConfig`, FMA-ATL starts applying ATL rules by choosing a match from this pool in the rule `E-Schedule`. This rule selects the root container of the matched object (and the corresponding locations) from the query domain model and uses it as the source domain model for executing the rule. It then applies the matched rule with the selected match via the rule `E-RuleSideEffects`. The resulting target domain and trace maps are updated with the results from executing the matched rule and the pool of enabled matches is updated by disabling all matches involving the matched object (according to the ATL semantics it cannot be transformed any longer as it appears in a trace link).

The semantics of ATL matched rules is given by two big-step semantic rules: `E-RuleSideEffects`, which executes the model actions defined for a domain in that ATL rule[6] via the rule `E-DomainActions`, creating an `AtlFinalConfig` object with updated target domain and trace maps, and deleting the match; and `E-DomainActions`, which applies the model actions for a particular domain by executing the FMA statement in the target domain and the trace statement for creating trace links in the global and local trace maps, creating an `AtlDomainFinalConfig` without any pending domain actions.

3.4 ResolveTemp Expressions and Lazy Rules Call Statements

`ResolveTemp` expressions. To consider `resolveTempExpr` expressions we have extended the set of FMA expressions with `resolveTemp(E,P)`, where `E` is an expression that evaluates to a collection of object references and `P` is the name of the target variable to be used when exploring trace links. The semantics of such an expression, defined by the big-step semantic rule `E-ResolveTemp` in Fig. 3, evaluates the expression `E` to a set `IS` of object identifiers and then, for each object identifier `SO` in `IS`, it obtains the set of target identifiers `TO` by getting the value for the key `(SO,P)` in the local trace map `TL` in the function `resolveTemp`.

Lazy rule call statements. We have modelled calls to lazy rules as statements instead of modelling them as expressions as in FMA an expression cannot perform changes in a model. Thus we have extended FMA model actions with the statement `setCmtLazyRule(P,RN,O)` where `P` is the containment reference to contain the newly created objects, `RN` is the name of the lazy rule, and `O` is the matched object to be used. In the translation from ATL rules to FMA statements illustrated in subsection 3.2, a lazy rule call expression is translated into an FMA model action. In FMA-ATL, a lazy rule is used to expand containments under an object under focus (that is, in a snapshot statement) and its semantics is that of a matched rule as described by the rules `E-RuleSideEffects` and

[6] At the moment, we only consider one target domain. To consider more than one target domain, we only need to add an additional rule to iterate over the different domain actions appropriately.

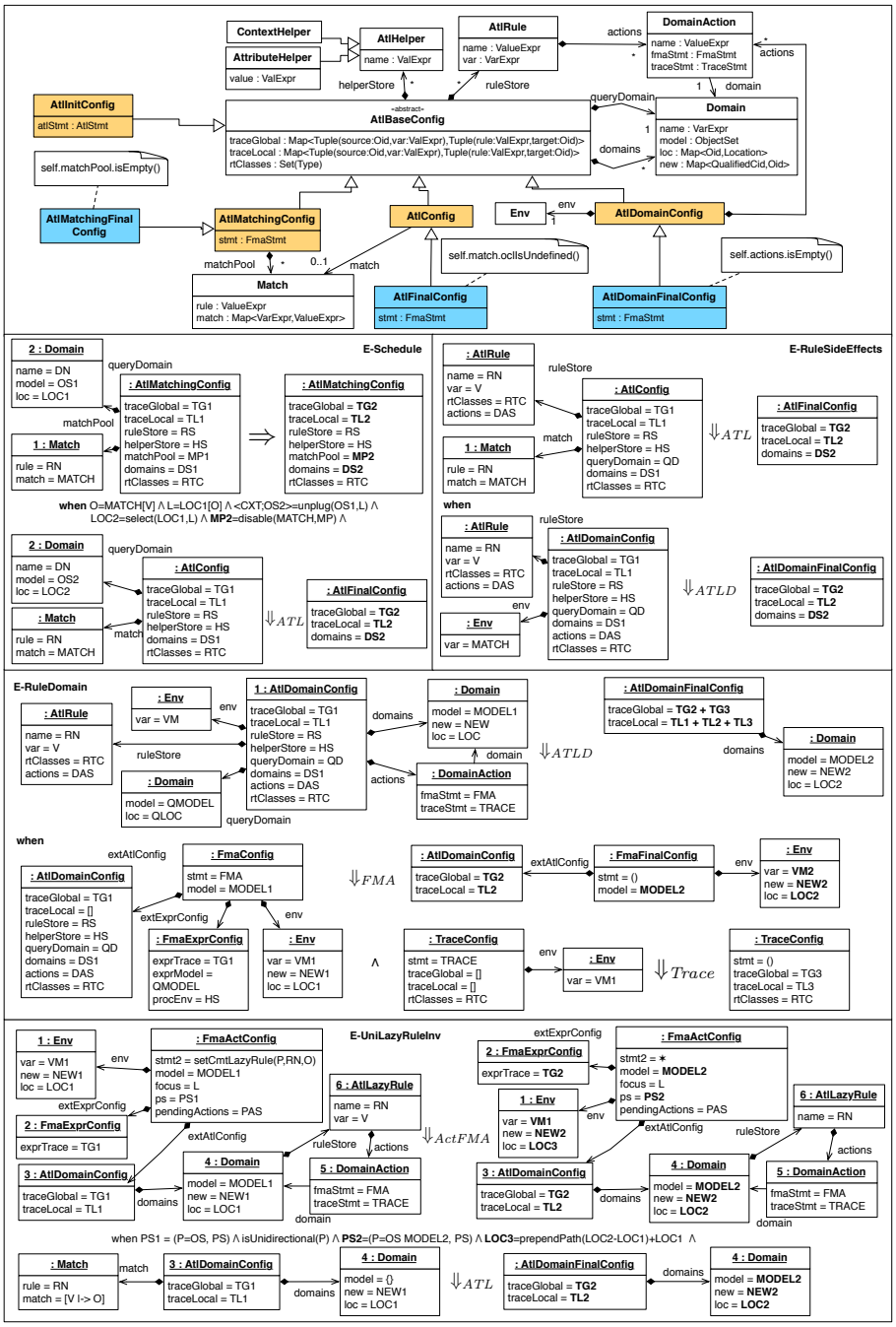


Fig. 5. FMA-ATL semantics: configurations and scheduling rules.

`E-DomainActions`. The only difference is the way in which they are invoked as the match is fixed by the call statement and not by the scheduler.

The way in which we have allowed the interaction between FMA and FMA-ATL while preserving the semantics of the former, is by means of the extension point `extFmaConfig` of class `AtlConfigs` in Fig. 2 which allows the FMA model action `setCmtLazyRule` to invoke a FMA-ATL rule with an appropriate configuration as shown in rule `E-UniLazyRuleInv` in Fig. 5 for uni-directional containment references. This rule prepares the configuration for executing the lazy rule as a matched one with an empty target model and stores the results in the extended configuration `extAtlConfig`. Bi-directional containment references are handled similarly but a new model action is added to the set of pending actions in order to update the opposite reference.

4 Experimentation

In this section, we compare the efficiency of our interpreter against ATL 3.6 using two experiments: one for the transformation without lazy rules and one for the transformation with lazy rules. For the experiments we have developed a model with one package containing 5 classes, each of which containing 4 attributes of each particular type in order to exercise all the rules in the transformation.[7] The results are summarized as follows:

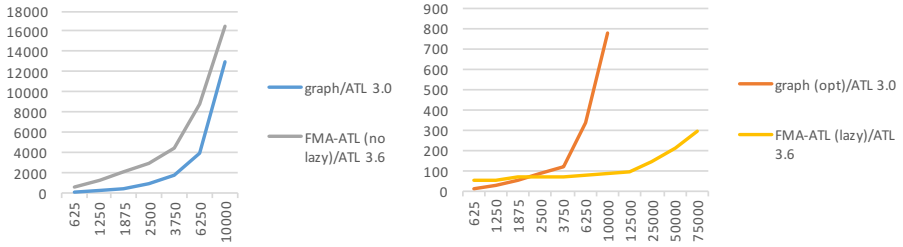
		Flat models with Maude rules			Structured models with FMA-ATL			
		Without lazy rules [14]			Without lazy rules		With lazy rules	
classes	attributes	ATL 3.0	Maude	optimized	ATL 3.6	FMA-ATL	ATL 3.6	FMA-ATL
125	500	0.3”	15”	4”	0”	35.6”	0”	3.3”
250	1000	0.5”	1’ 37”	15”	0.1”	2’ 38.2”	0.1”	7.1”
375	1500	0.8”	5’ 53”	40”	0.1”	6’ 12”	0.1”	10.9”
500	2000	1.1”	16’ 9”	1’ 37”	0.2”	11’ 44.1”	0.2”	14.9”
750	3000	2”	58’ 28”	4’ 2”	0.3”	28’ 5.3”	0.3”	22.6”
1000	4000				0.5”	54’ 3.8”	0.4”	31.6”
1250	5000	3”	3h 16’ 49”	16’ 37”	0.6”	1h 32’ 51.8”	0.5”	41.4”
2000	8000	5”	17h 57’ 15”	1h 4’ 19”	1”	4h 46’ 17.6”	0.8”	1’ 20”
2500	10000				1.3”		1.1”	1’ 43.9”
5000	20000				2.7”		2.2”	5’ 14”
10000	40000				6.6”		4.8”	17’ 16.7”
15000	60000				11”		7.5”	36’ 43.2”

In both experiments, ATL shows better performance and scalability. However, simulating transformations in Maude has an additional advantage as it provides analysis techniques that can be used for reasoning about ATL model transformations, such as bounded model checking of invariants.

The scalability of ATL model transformations in FMA-ATL using structured models and plain configurations of objects with graph-based rewriting rules has been compared by using the experiment results presented in [14]. Their formalization has two variants: an original one [13] and an optimized one [14]. The comparison of the scalability of the FMA-ATL interpreter for the ATL transformations in the case study with the implementations of the transformation in

[7] The case study and the resources for the experiments performed can be found at `https://github.com/arturboronat/fma-atl-experiments`.

Maude presented in [14] has to be considered with the following validity threats in mind: the authors used ATL 3.0, an older machine, and the input models that were used are different − e.g. ours include one additional package per five classes. Given that both Maude implementations are less efficient than ATL, the comparison is based on the unitary additional cost of executing an ATL transformation in each approach with respect to the time obtained in the corresponding experiment. In this way, we mitigate the two first threats to validity. In the graphs below we show the model size (without counting packages) on the X axis and the additional unitary cost on the Y axis for ATL transformations with matched rules only (left) and with lazy rules (right).



On the one hand, the FMA-ATL interpreter shows a worse performance when dealing with model transformations without lazy rules. However, its performance is actually comparable to that of the transformation (without optimization) on flat models. The main reason for FMA-ATL to perform worse in the first transformation is that every containment that is set to an object created by a separate matched rule requires searching the object in the model and its performance worsens as the size of the output model grows.

On the other hand, the execution of the transformation with lazy rules in FMA-ATL scales better than the optimized ATL transformation with unstructured models.[8] When using lazy transformations in FMA-ATL, the whole composite object (database) is created in one rule (Package2Database) and all containments are created under the corresponding container object without having to perform any search in the output model − drastically improving the scalability of the approach. This performance could be improved by a constant factor by making the compilation Maude-specific. That is, by using model patterns to implement side effects and it is done in the right-hand side of a rewrite rule in order to avoid the computation performed by FMA operations. However, this would lose the advantages of the layer of abstraction that FMA provides.

5 Related work

In [8], the AMMA platform was extended with abstract state machines using XASM for specifying the semantics of DSLs, including the semantics of ATL.

[8] Note that the authors also showed how to implement lazy rules in [14] as functions although there are no experimental results. However, such semantics for lazy rules differs from that of FMA-ATL where the semantics of a lazy rule is exactly the same as that of a matched rule, including the production of trace links.

As far as we are aware, there was no empirical evaluation of the performance of the code generated from XASM specifications for ATL. We have focussed on the use of FMA for specifying a structural operational semantics of the ATL transformation language by following a more declarative approach. That is, in our specification there is no explicit code to control the firing of semantic rules (e.g. choose and for all statements). In addition, transition rules in FMA-ATL faithfully reflect their representation in rewrite theories in Maude, enabling the automated analysis of ATL (out-place) model transformations.

In [12], ATL model transformations are translated to DSLTrans enabling the verification of the correctness of model transformations using pre-/post-condition contracts via a symbolic-execution property prover. DSLTrans transformations are graph transformations represented as declarative rules whereas FMA programs only represent the application of model transformations, decoupling them from the query mechanism. At present, the verification of ATL transformations in FMA-ATL is based on bounded model checking against contracts, which can be written in OCL, from a given source model. Focussing on other bounded model checking approaches, [6] the semantics of ATL transformations is captured in ATL transformation models using OCL constraints and reduces the verification of their partial correctness to a satisfiability problem in Alloy.

A rewriting logic semantics of ATL using Maude has been proposed in [13,14], as discussed from a quantitative point of view in Section 4. From a qualitative point of view, their representation of models is based on sets of plain objects. ATL model transformation rules are represented using conditional rewrite rules using Maude as a high-level declarative rule-based language. Their formalization requires a compiler from ATL to their encoding of ATL transformation rules, which may require non-trivial program transformations for the optimized version, as acknowledged by the authors. Our approach is based on an interpreter for ATL that reuses FMA to encode the model transformation actions of each rule, encapsulating all the cases that would need to be implemented in the compiler in the other approach. This design decision allows us to represent the semantics of ATL in a small set of rules, implementing an interpreter, while retaining the analysis facilities that Maude's toolkit provides. Therefore, these two approaches differ in the way they deal with abstraction when capturing ATL semantics in rewriting logic. Both approaches use mOdCL [11] as their OCL interpreter.

6 Conclusions

In this paper, we have presented a big-step structural operational semantics for out-place ATL model transformation rules and the specification of a simple ATL interpreter. The conditional rewrite rules implementing the FMA-ATL interpreter in Maude facilitate the verification of ATL model transformations.

The formalization and design of the interpreter has helped us detect that calls to lazy rules in ATL expressions cannot be modelled as FMA expressions since they do not simply evaluate to a value. They also update other components of the interpreter configuration and have to be considered as FMA statements.

FMA-ATL enables the execution − and, thus, verification − of a larger class of out-place ATL model transformations than previous approaches (increasing the size of models from 10K objects to 78K objects) when working with models that consist of objects with containments and lazy rules aggregating the side effects of several dependent matched rules. The formalization in [14], based on compilation from ATL to Maude, is more appropriate for models without containments or when dealing with many interdependent matched rules. In future work, we are going to apply the lessons learnt from transformations with lazy rules for making ATL transformations with matched rules more scalable. On the other hand, we plan to address in-place model transformations and unique lazy rules in order to cover a larger gamut of ATL model transformations.

References

1. Ab. Rahim, L., Whittle, J.: A survey of approaches for verifying model transformations. Software & Systems Modeling 14(2), 1003–1028 (2015)
2. Amrani, M., Combemale, B., Lucio, L., Selim, G.M.K., Dingel, J., Traon, Y.L., Vangheluwe, H., Cordy, J.R.: Formal Verification Techniques for Model Transformations: A Tridimensional Classification. JOT 14(3), 1:1–43 (2015)
3. Boronat, A.: Well-Behaved Model Transformations with Model Subtyping. CoRR abs/1703.08113 (2017)
4. Boronat, A., Heckel, R., Meseguer, J.: Rewriting Logic Semantics and Verification of Model Transformations. In: FASE. pp. 18–33. LNCS 5503 (2009)
5. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers (2012)
6. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL Transformations Using Transformation Models and Model Finders. In: ICFEM'12. pp. 198–213. LNCS 7635 (2012)
7. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude. LNCS 4350 (2007)
8. Di Ruscio, D., Jouault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs, Laboratoire d'Informatique de Nantes-Atlantique. LINA (2006)
9. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. 72(1-2), 31–39 (2008)
10. Lawlew, M., Duddy, K., Gerber, A., Raymond, K.: Language Features for Re-Use and Maintainability of MDA Transformations. In: OOPSLA & GPCE Workshop (2004), adapted in `https://www.eclipse.org/atl/atlTransformations/#Class2Relational`
11. Manuel Roldán and Francisco Durán: The mOdCL evaluator: Maude + OCL. `http://maude.lcc.uma.es/mOdCL/` (2013), online; accessed 3 March 2016.
12. Oakes, B.J., Troya, J., Lucio, L., Wimmer, M.: Fully verifying transformation contracts for declarative ATL. In: ACM/IEEE MoDELS'15. pp. 256–265 (2015)
13. Troya, J., Vallecillo, A.: Towards a Rewriting Logic Semantics for ATL. In: ICMT'10. pp. 230–244. LNCS 6142 (2010)
14. Troya, J., Vallecillo, A.: A Rewriting Logic Semantics for ATL. Journal of Object Technology 10, 5: 1–29 (2011)