A Formal Framework for Prototyping Executable Semantics in ATL

Artur Boronat[0000−0003−2024−1736]

Department of Informatics, University of Leicester, UK
aboronat@le.ac.uk, http://arturboronat.info

Abstract. ATL is a well-established model transformation language both in industry and in academia, where it is used as a reference language for studying different types of model transformations and their properties. In this paper, we discuss current limitations of ATL's in-place semantics that hamper its application for modelling and verifying systems and propose a new in-place semantics for ATL that enables it as a specification language for simulating and verifying EMF-based systems. Our approach is based on FMA-ATL, an executable specification of a large excerpt of ATL in Maude, which has been augmented with the new in-place semantics so that Maude's verification tools can then be used both to perform bounded model checking of invariants and to model check LTL formulas in the resulting system models, where appropriate. Furthermore, FMA-ATL uses ATL as front-end language and it can be *reused as-is* for verification, including its tool support.

Keywords: EMF, ATL, system specification, formal methods.

1 Introduction

ATL is a model-to-model transformation language that seeks pragmatism by ensuring that executed model transformations always produce the same result. This offloads the responsibility of ensuring those conditions from software engineers when designing a model transformation, which helps to focus on the domain problem, namely the transformation definition. Such pragmatism is implemented by using a read-only source model in which model elements are transformed only once, building an output model from scratch. There are situations where such bulk semantics is too expensive, both from a productivity point of view and from a computational point of view. For example, when endogenous model transformations involve sparse model updates in possibly large models, explicit rules need to be introduced in order to copy the model elements that are not the target of model updates and that are to be preserved.

The ATL2010 compiler addresses these concerns by emulating in-place model transformations [10] for ATL transformations in refining mode using a two-step process, which relies on an abstract language for defining updates. In a first step, rules are applied and a diff model is computed representing in-place changes as a patch of model differences. In a second step, the changes obtained from all the rules are reordered in order to apply creations first, modifications afterwards and deletions at the end, ensuring the standard properties of ATL transformations. The resulting sequence of model changes is applied as a patch to the model.

One could consider the use of in-place ATL transformations for specifying and simulating software systems by modelling system states with EMF (Eclipse Modeling Framework) models and by capturing the dynamic aspects of the system with an ATL modules. System simulation could be achieved by successive applications of the transformation to pre-states in order to produce post-states, with the amalgamated updates of several rule applications. However, ATL in-place semantics presents a number of drawbacks that hampers such an approach. By the very nature of ATL, transformations are deterministic when they are evaluated by ensuring that each source object is matched by an ATL rule only once. This means that only a subset of deterministic systems can be modelled for verification purposes. Non-deterministic systems and, hence, concurrent systems cannot be modelled with ATL transformations. In particular, there are two reasons that hinder *soundness* and *completeness* of the verification of systems modelled with ATL. First, when a transformation is applied to a source model, ATL's in-place strategy affects side-effects but not the matches of rules, which are computed up front. For example, the application of a rule that disables a pre-computed match is disregarded, yielding an incorrect result. Second, the application of rules that enable new matches are also disregarded for the same reason. This means that in-place ATL transformations may not capture all the intended behaviour of a system. That is, an in-place model transformation model, understood as the class of model transformations for all models conforming to the source metamodel, cannot be used as a system model, corresponding to execution paths between system states. This is important from a verification point of view, as the absence of errors in that case is no guarantee of the correctness of the actual system. That is, the specification is an under-approximation of the intended behaviour. These drawbacks are illustrated with the running example of section 2 in section 5.

In this work, the structural operational semantics (SOS) of FMA-ATL [2], which formalizes a large excerpt of representable ATL model transformations, has been augmented with a new in-place semantics that overcomes those problems. This semantics specification is implemented faithfully in Maude [3] yielding a scheduler for applying matched rules and an interpreter for their side effects. FMA-ATL uses the EMF as front-end for defining metamodels and models, permitting the reuse of EMF-based systems and domain-specific modelling languages (DSMLs). Furthermore, our approach reuses Maude's verification tools for analysing correctness properties in the resulting system models. Furthermore, we use the official ATL language as the front-end language for FMA-ATL, providing a new engine for ATL equipped with formal verification techniques. This last contribution facilitates the validation and verification of ATL system specifications by reusing the tool ecosystem that is already available for ATL, facilitating the collaboration between software engineers specialized in (model-driven) software development and software engineers specialized in validation and verification. The tool and examples used are available at `https://fma-atl.github.io/`.

In the rest of the paper: in section 2, ATL is presented as a specification language for EMF-based systems, using the *Concurrent Append Problem* from [9] as a running example; in section 3, ATL is used as property specification language and the different verification techniques supported in FMA-ATL are illustrated; in section 4, the integration of ATL is discussed; in section 5, the shortcomings of ATL2010's in-place semantics are illustrated with the running example and FMA-ATL features are compared against those of representative tools used for simulating and verifying EMF-based systems; and final concluding remarks are given in section 6.

2 ATL as System Specification Language

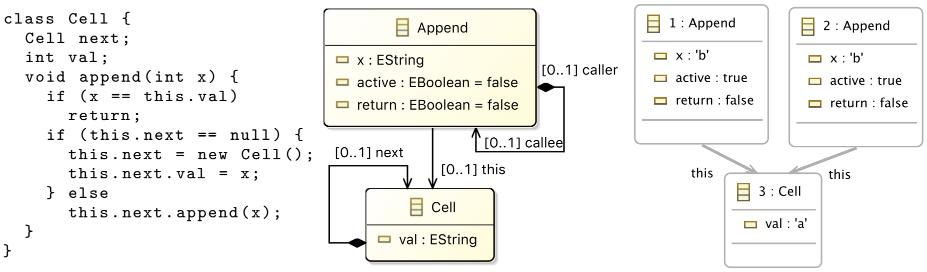
Combining refining mode and in-place transformations without control flow constraints removes the guarantees that make ATL transformations confluent and terminating. Fortunately, these are also the conditions that enable ATL as a specification language for modelling concurrent processes, which we consider in the rest of this section. We first introduce an example that cannot be modelled using the in-place semantics of ATL, explaining the syntax used to model EMF-based systems, and then show how this semantics is captured by augmenting the FMA-ATL semantics with a new scheduler rule.

2.1 The Concurrent Append Problem

Our running example is the *concurrent append problem* of the Java program of Fig. 1, adapted from [9], which implements the append method on cells, which may be arranged forming a list, of Fig. 1. Given a String value `x` as parameter, the program appends a new tail cell to the list if `x` is not contained in any of the existing cells. An example correctness criterion is that the list of cells must not contain the same value more than once. However, different threads may access cells concurrently by calling the append method, which might result in undesired race conditions without certain assumptions on atomicity.

In this paper, the example is modelled using the ATL system specification of Fig. 2, where the state model, represented as an EMF model, and an initial state are given in Fig. 1. The state model expresses that each `cell` contains a value `val` and may have a successor via the composition `next`. The append method call is represented with: an `Append` object with the argument `x` to be inserted; an `active` flag (corresponding to the program counter) indicating that the method is being executed; and a `return` flag indicating that the execution of the method is over. Recursion is modelled using the containment reference `callee`.

States are represented as nested object diagrams, where objects may be nested through containment references. Each containment reference is depicted as a labelled box, whose contents are the immediate children. The state of Fig. 1 shows two concurrent calls to `append("b")` on the singleton list `["a"]`.



```
class Cell {
  Cell next;
  int val;
  void append(int x) {
    if (x == this.val)
      return;
    if (this.next == null) {
      this.next = new Cell();
      this.next.val = x;
    } else
      this.next.append(x);
  }
}
```

Fig. 1. Java program (left), model of states (middle) and initial state (right).

In the following, we explain how to use ATL to model the `Append` method using a new in-place semantics, an defer the discussion of deviations w.r.t the current in-place semantics in subsection 5.1.

In ATL, a (matched) specification rule has: a name; a source pattern, denoted with the keyword `from`, consisting of an object variable (or element) to be matched for the rule to be enabled and a filter condition expressed in OCL where the object variable can be used; a potentially empty list of local variable initializations, enclosed by the `using` block; and a target pattern, denoted by the keyword `to`, containing a list of object variables (or elements) that refer to objects that are created, when a new variable is used, or to objects that are updated, when the name of a declared variable (either the source pattern variable or a local variable) is suffixed with `__ref`. Each target object variable encloses a list of bindings, each of which corresponds to a feature (attribute or reference) initialization, when the object is created, or to a feature update, when the `__ref` naming convention is used in the object variable name. Updates for attributes reset their value. Updates for references (including containments) either reset their value, if the upper bound is one, or append a new reference if the upper bound is not met. Updates may also be destructive for references if the suffix `__unset` is appended to the name of the reference. In that case, the corresponding reference is deleted if the lower bound is not met. A containment reference can only be deleted when the contained object is *isolated*, there are no incident references to it or to any of its contents. The deletion of a containment reference implements *cascade delete* semantics, that is the contained object and its contents vanish outright. ATL also provides the statement `drop` that can be used in the target pattern of a specification rule, denoting the deletion of the object matched by the source pattern, with the same *delete cascade* semantics.

An ATL specification can also contain lazy specification rules, which are not matched by the scheduler and have to be called from matched rules explicitly. In FMA-ATL [2], unique lazy rules are used to reduce the state space of a system specification from an initial state by amalgamating the side effects of all the lazy rule applications in one big transition step. Moreover, an ATL specification can contain helpers, which are functional operations that can be used to query the source model or to perform computations. ATL helpers will be discussed in section 3.

The ATL specification modelling the dynamic behaviour of the `append` method, adapted from [9], is shown in Fig 2, and consists of the following rules:

Append a new cell. Rule `Append` is responsible for appending a new cell to the list if the control reaches the last cell (there is an active `Append` object pointing to the last cell) and the value stored at this last cell is not equal to the method argument.

Go to next cell. Rule `Next` checks whether the method argument is not equal to the value stored at the current (`this`) cell and makes a recursive call then for checking the next cell by generating a new `Append` object and declaring it as the `active` call, deactivating the current call.

Value found in list. Rule `Found` checks if the method argument matches the value stored at the current (`this`) cell and, if so, indicates that the computation is over by disabling `active` and by enabling `return`.

Return result. Finally, rule `Return` simply removes an append invocation object (from the stack of recursive calls) if it has already calculated the result.

```
rule Append {
  from a1 : append!Append (
    a1.active and a1.this.val <> a1.x
    and a1.this.next.oclIsUndefined()
  ) using {
    c1 : append!Cell = a1.this;
  } to a1__ref : append!Append (
    x <- '',
    active <- false,
    return <- true
  ),
  c2 : append!Cell (
    val <- a1.x
  ),
  c1__ref : append!Cell (
    next <- c2
  )
}
rule Found {
  from a1 : append!Append (
    a1.active and a1.x = a1.this.val
  ) to a1__ref : append!Append (
    x <- '',
    active <- false,
    return <- true
  )
}
```

```
rule Next {
  from a1 : append!Append (
    a1.active=true and a1.x <> a1.this.val
    and a1.callee.oclIsUndefined()
  ) using {
    c : append!Cell = a1.this.next;
  } to a1__ref : append!Append (
      active <- false,
      x <- '',
      callee <- a2
    ),
    a2 : append!Append (
      active <- true,
      x <- a1.x,
      this <- c
    )
}
rule Return {
  from a1 : append!Append (
    a1.return = true and
    not(a1.caller.oclIsUndefined())
    and a1.callee.oclIsUndefined()
  ) using {
    caller : append!Append = a1.caller;
  } to caller__ref : append!Append (
    return <- true,
    callee__unset <- a1
  )
}
```

Fig. 2. An ATL version of the method `Cell::append(x: String)`.

The FMA-ATL engine consists of a scheduler rule that is applied to engine configurations, i.e. *states* of the FMA-ATL engine. Given an ATL system specification, the FMA-ATL engine parses matched/lazy rules and helpers, producing the initial engine configuration. This includes the computation of attribute helpers, caching their result. It then computes all enabling matches for matched rules, by considering their source pattern element and its filter condition. Then the scheduler starts the system simulation 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 system state. After these side effects are applied, continues the execution with the next enabling match until no more rules can be applied. In subsequent sections, we describe the engine configurations and how the scheduler rule is used to simulate ATL system specifications from an initial system state.

2.2 FMA-ATL Configurations and Engine Initialization

The main configuration types of the FMA-ATL engine are depicted in Fig. 3. The class `AtlMatchingConfig` represents the configuration of the engine for applying matched rules: 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 domain contains a `name` that identifies the domain, a `model` referring to a collection of objects, a `loc` map with locations for the objects in the model, and a factory `new` for obtaining fresh identifiers when a new object is created.

To simulate a system specification, the FMA-ATL engine first initializes an `AtlMatchingConfig` configuration, loading each specification rule into a rule store and helpers into a helper store. A FMA-ATL rule is initialized, by generating a FMA statement that models the side-effects on the state that are represented in the bindings of the target pattern of a specification rule, as described in [2].

A FMA statement can be regarded as a sequence of typical updates that can be performed in an EMF model instance. This initialization is performed by extracting a graph of side effects from the list of bindings of each target pattern element. Nodes are target pattern object variables and expressions representing a query (used in the initialization of the binding). Named edges are defined from object variables to expressions or to other object variables representing each binding. Once the graph is generated, FMA-ATL walks through the graph twice starting from the root object and following containment edges: first, it obtains a 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.

For the in-place semantics, the type graph of the graph that is used to represent the side effects of an ATL specification rule has been augmented with *update* and *drop* nodes. On the one hand, update nodes are obtained when the name of an object variable in the target pattern of a rule contains the suffix `__ref` and it coincides with the object variable used in a source object variable

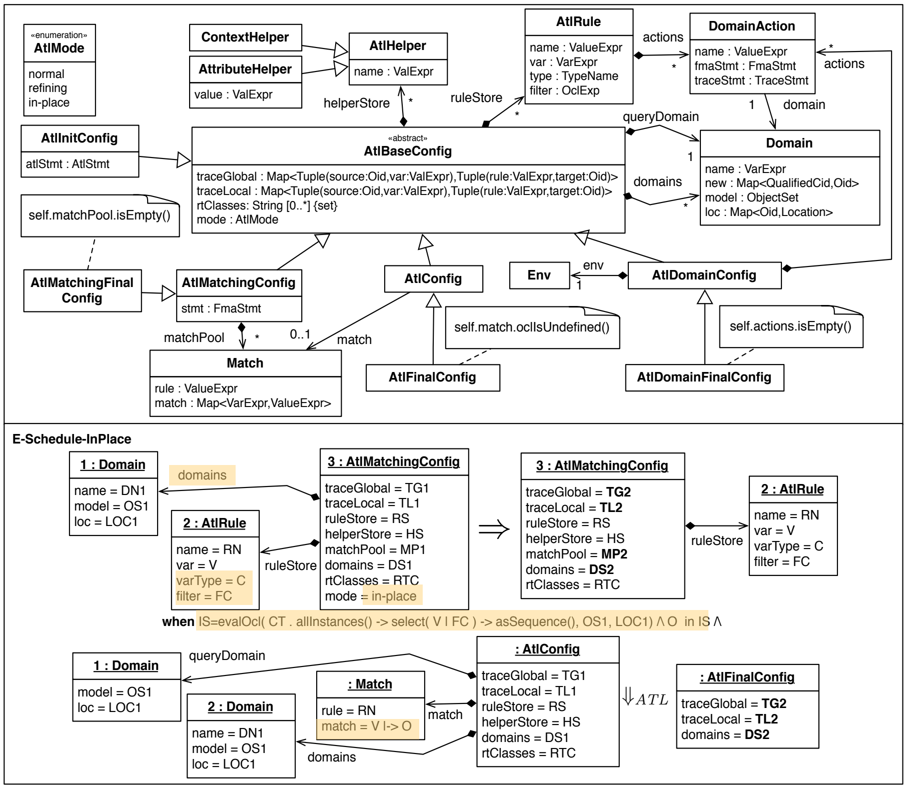


Fig. 3. FMA-ATL configuration model (top) and in-place scheduler rule (bottom).

(either the source pattern object variable or a local variable of the using block). On the other hand, drop nodes are obtained when a drop statement is parsed. When a side-effect graph is translated into a FMA procedure, update nodes correspond to free variables that are to be bound in the environment. That is, FMA-ATL does not create a new object for update nodes in the first traversal of the graph. Moreover, drop nodes correspond to object destruction by deleting the corresponding containment reference when the object is not a root one.

Moreover, binding compilation to FMA statements has also been augmented by allowing deletion of references. This is used in an ATL specification by appending the suffix __unset to a reference name in a binding of a target pattern element. Such bindings are compiled to unset model actions in FMA when the graph of side-effects is traversed. In FMA, an unset action deletes a reference if the lower bound of the reference is not met. The deletion of a containment can only be applied when the object to be deleted or any of its contents are not referenced from an external object. Such a deletion entails the deletion of the objects, including its contents.

2.3 Rule Scheduling

In-place semantics for ATL specifications is defined by using a new scheduler rule `E-Schedule-InPlace`, shown in Fig. 3, where the main differences w.r.t. the normal (out-place) scheduler rule of [2] are highlighted. This rule is introduced in order to compute matches during the execution of the transformation, avoiding the up-front computation of the matches.

A match is computed as in the computation of matches up-front when ATL is executed in normal or refining mode. Given the contextual type `C` of the variable `V` used in the in pattern element of the rule and the filter condition `FC`, the list of matches for a rule is computed by evaluating the expression

```
CT.allInstances()->select( V | FC )->asSequence()
```

over the model `OS1` with the location map `LOC1` using the operation `evalOcl`. Thereby, the scheduler considers causal dependencies between rules based on the current state. In the implementation, rules are ordered lexicographically by name and the list of matches is ordered by each object internal id, a natural number. Therefore, each rule is applied for each list of matched objects orderly in the expression `O in IS` given that the list `IS` is computed for each rule application. This is, however, a potential source of *starvation* that needs to be taken into account when specifying a system: if a rule is always enabled for a list of objects, it will always be applied to the first object, treating others unfairly.

Once a match is found for a given specification rule, the match is defined by using the variable `V` of the in pattern element, and the side-effects of the specification rule, represented as a FMA statement, are interpreted using the big-step evaluation relation $\Downarrow_{ATL}$ presented in [2]. We can regard this evaluation relation as a black-box component where the precondition involves that a match for a rule must be selected and that the system state must be in the query domain (used to evaluate OCL queries) and in the domain (used to apply rule side effects). The postcondition of the evaluation relation guarantees that the system state in the resulting domain `DS2` is well-formed after applying the rule.

The new scheduler rule of Fig. 3 allows FMA-ATL to simulate system specifications. More specifically, this rule has been faithfully been implemented in Maude as a rewrite rule and each application of the `in-place` scheduler rule coincides with one system transition, thus executing the FMA-ATL engine from an initial configuration amounts to simulating the system specification from an initial system state. Furthermore, the `in-place` scheduler rule allows to reuse Maude's toolkit to traverse the system state space for verification purposes, as explained in the following section.

The behaviour of the system specification of our running example can thus be simulated by running the FMA-ATL engine with a system specification from an initial system state. Taking the system state of Fig. 1 as initial, a valid resulting execution path is graphically depicted in Fig 4, denoting a rule application with an arrow, whose label contains the name of the arrow between system states, but for the initial state, and the identifier of the matched object (in between parenthesis).

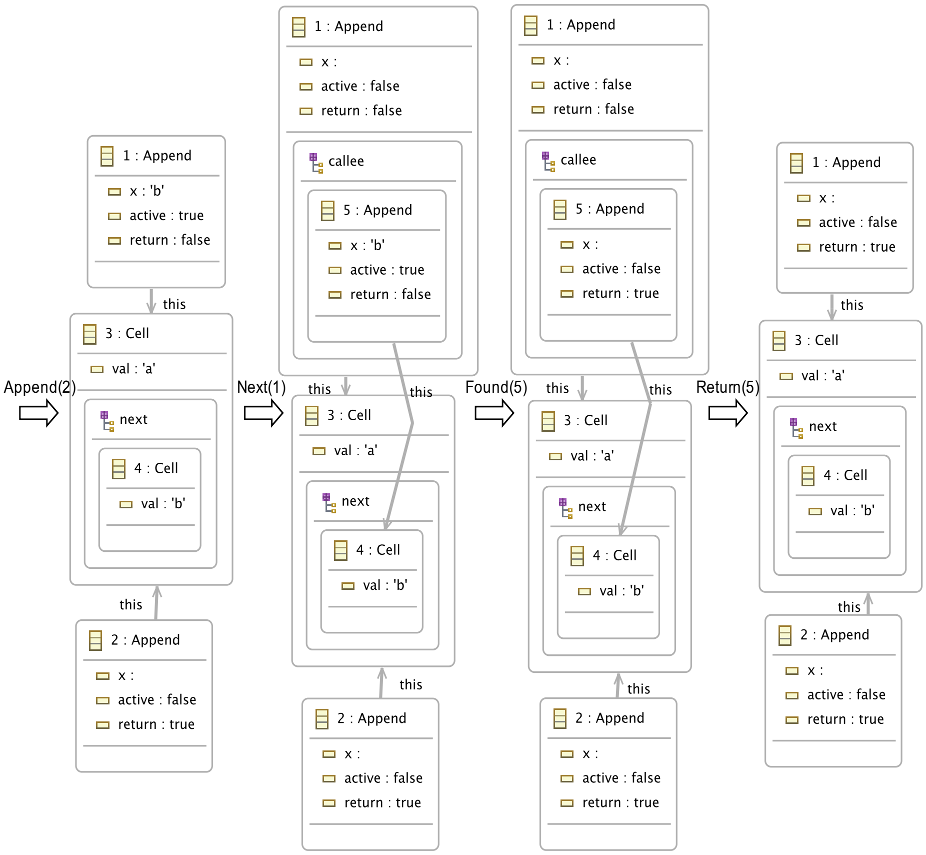


Fig. 4. Simulation from the initial system state of Fig. 1.

3 ATL as Property Specification Language

FMA-ATL is implemented in Maude so that we can reuse its LTL model checker for verifying temporal properties when the system specification models a finite state space [6], and its bounded model checker for invariants when the state space is infinite. The *model checking problem* consists in deciding whether a given correctness property holds in a specified system by systematically traversing all enabled transitions in all system states. That is, in FMA-ATL, given a system state, represented as an Ecore model instance, all possible enabled specification rules are applied and this procedure is recursively repeated on the successor states until no more matches are found.

Relevant classes of such correctness properties are *safety* and *reachability* properties. A safety property defines a desired property that should always hold on every execution path or (equivalently) an undesired situation which should never hold on any execution path. A reachability property describes, on the contrary, a desired situation which should be reached along at least one execution path. These two types of properties are interrelated in that a proof of the violation of a safety property is a witness of the reachability property defined as the negation of the safety property. Hence, if a safety property holds (or a reachability property is violated), the entire state space needs to be examined.

Such correctness properties are frequently formalized as LTL formulae built over a set of *state properties*, which either hold or not in a given system state. In FMA-ATL, the property specification is given in a separate ATL module defining the satisfaction of each state proposition using an ATL attribute of the form `helper def : P : Boolean = B ;` where `P` is the name of the state property and `B` is the boolean OCL expression that defines its satisfaction. The property specification module must share the same header with the system specification module. For example, in the listing below, the `Shared` property denotes when two distinct cells of the list contain the same value, a situation that is prohibited. Moreover, the `Isolated` property denotes a desired behaviour, a cell in the list and an append call will never share the same value in the same system state.

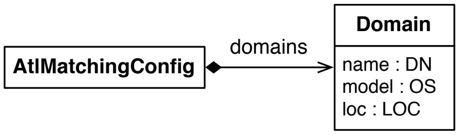
```
helper def: Shared : Boolean =
append!Cell.allInstances()->exists(c1 |
  append!Cell.allInstances()->exists(c2 | c1<>c2 and c1.val=c2.val )
);

helper def: Isolated : Boolean =
append!Cell.allInstances()->forAll(c |
  append!Append.allInstances()->forAll(a | a.x<>c.val )
);
```

To use Maude's model checker, the following components need to be characterized: the type of *states*, by defining a subsort of the sort `State`; the set of *state predicates* to be used as invariants or as atomic propositions in LTL formulas, by declaring them as subsort of the sort `Prop`; and finally the *satisfaction* of such state predicates, by providing equations for the operation

`op _|=_ : State Prop -> Bool`

In FMA-ATL, the set of states is defined by the class `AtlMatchingConfig` of our interpreter in Fig. 3. In that way, system states included in domains are wrapped by additional constructs that are used to specify the operational semantics. FMA-ATL declares a state property for each of the helper attributes defined in an ATL property specification module and defines its satisfaction using equations of the form[1]



$\models$ `P = evalOcl(B, OS, LOC)`

[1] Internally FMA-ATL works with a term representation of engine configurations and system states, which is depicted graphically for the sake of presentation.

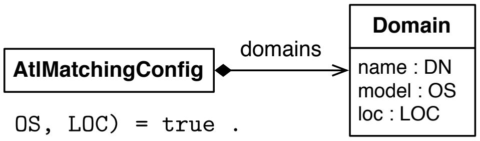
Then we can verify that the system satisfies the property that all cells in a given list will always contain unique values after a set of `append` calls, which may or may not contain the same values, with the following command:

```
red modelCheck( initialConfig , []~Shared ) .
```

where `~` denotes *not*, and `[]` is the LTL operator *always* (□) meaning that the property must hold in all future states, and `initialConfig` is the term resulting from the engine initialization phase as explained in section 2.2.

Given an ATL system specification and property specification modules, an initial system state and the name `P` of an state property with body expression `B`, in the property specification module, FMA-ATL can verify invariants, such as `Isolated`, by traversing the state space using a breadth-first strategy with Maude's search command[2]

```
search initialConfig =>*
```



```
such that evalOcl(not B, OS, LOC) = true .
```

That is, the command searches for a configuration containing a system state where the expression `B` is violated. If such configuration is not found, the refutation process ends unsuccessfully and the invariant is satisfied because the state space is finite, in the example. For systems where the state space is infinite, an upper bound can be used for the analysis trading completeness for decidability.

4 Integration with ATL

FMA-ATL is available[3] as an EMF-based standalone library that can be used to execute a substantial excerpt of ATL model transformations. It enables formal verification of systems where the specification language, both for systems and for state properties, is ATL itself.

The execution of out-place model transformations, which was presented in [2], has been augmented with new functionality developed for this work: *(1)* integration with the ATL language; *(2)* simulation of model-based systems using ATL as specification language (with in-place matched rules); *(3)* bounded model checking of invariants, which are specified in ATL, when the state space of the specified system is infinite; and *(4)* software verification using LTL model checking, where state properties are specified in ATL, when the state space of the specified system is finite.

[2] Using `=>*` the search will be performed along zero or many simulation steps. However, other strategies that can be used are `=>!` for run to completion semantics, `=>1` for one step, `=>+` for at least one step.

[3] `https://fma-atl.github.io`

The front-end language for defining metamodels and system state models is EMF (Ecore) and the language for specifying model transformations, system specifications and property specifications is ATL. To implement the integration with ATL, FMA-ATL reuses parts of AnATLyzer [4] to infer types from ATL expressions and extends its ATL serializer to serialize ATL transformations to FMA-ATL. In FMA-ATL, expressions that are specific to ATL and extraneous to OCL, like `resolveTemp`, are evaluated independently of OCL expressions so that a Maude implementation of OCL, `mOdCL` [8], can be reused. This means that ATL expressions have to be transformed in order to extract ATL specific expressions (`resolveTemp` expressions, invocation of helpers and attributes, invocation of lazy rules) from OCL expressions, which requires transforming local variables (iterator variables) into global variables (FMA variables) while using unique names and remembering the scope where they are used.

5 Related Work

In this section, we analyse our contribution w.r.t. related work by looking at the differences with ATL in-place semantics in detail and, then, by providing a broader view of the features of FMA-ATL.

5.1 Differences with ATL In-Place Semantics

From a system specification point of view, when using ATL2010 in refining mode, transformation rules can match several objects by means of the `using` block, and several objects can be added to the model but only the object matched by the source pattern element can be updated. That is, ATL does not support the update those objects matched in the `using` block. However, the matched object (and its contents) can be deleted using the statement `drop` (in the output pattern of a rule). Moreover, the naming conventions `__ref` and `__unset` and their semantics for applying updates to source object variables are ignored by ATL. These naming conventions are used by FMA-ATL to unset references, which cannot be done in ATL.

Regarding system verification, ATL in-place semantics is not sufficient for system specification, as explained in the introduction. To illustrate the unsoundness and the incompleteness of an ATL specification using ATL2010 in-place semantics (w.r.t. the intended behaviour of the system), we consider the scenario of the running example where the same element `'b'` is inserted twice in the singleton list of Fig 1. We have modified the rule `Append` as a workaround for the problems stated above, that is to help ATL apply updates to the matched object only. The main change in the state model of Fig. 1 involves the declaration of a reference `previous` as opposite to `next`. The new rule `Append`, shown in Fig. 5, captures the intended behaviour of the original rule: a new cell is appended to the list if it has not been found (the appender is marking the last element of the list, which has a different value). To apply the transformation, ATL computes the matches, enabling the rule for both objects `Append`, with ids `1` and `2`. When

```
rule Append {
  from a1 : append!Append (
    a1.active and
    a1.this.next.oclIsUndefined()
    and a1.this.value <> a1.x
) to a1__ref : append!Append (
    active <- false,
    return <- true,
    x <- ’’
  ),
  c2 : append!Cell (
    value <- a1.x,
    previous <- a1.this,
    list <- a1.this.list
  )
}
```

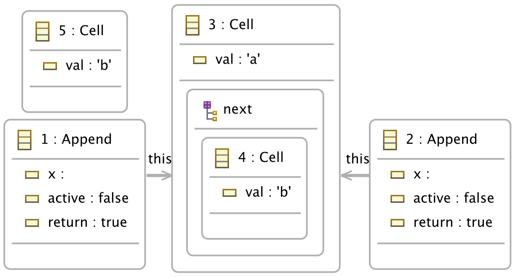


Fig. 5. Modified rule `Append` (left) and resulting state (right) from state of Fig. 1.

the transformation is executed, the first application of the rule `Append` inserts the new cell with id `4`. This should disable the match of the rule for object `1`. However, as the engine is blindly applying the pre-computed matches, a new cell with id `5` is inserted, leaving cell `4` dangling because of the upper bound of the reference `next`, as shown in the resulting state in Fig. 5. Hence, violating the state property `Shared`.

To consider a witness of incompleteness of an ATL2010 specification using in-place semantics w.r.t. the system behaviour, we look at the rule `Return`, which is enabled for object `1` after the first application of `Append`. However, the execution path of Fig. 4, where rule `Return` is applied before the application of `Append` to object `1`, is obliterated for the same reason and this behaviour is not captured by the ATL2010 in-place semantics of the system specification.

5.2 Comparison

Table 1 shows a comparison of features of ATL2010 in-place semantics with two ATL-based specification languages, namely SimpleGT [12] and FMA-ATL. To give a broader view of the contributions, we have also included Henshin[1], Groove [7] and e-Motions [11,5], which are also based on EMF (either directly or indirectly) and provide rule-based languages both for modelling and for verifying EMF-based systems.

We classify our comparison under two main dimensions: specification and verification. Question marks are inserted wherever definite information could not be found to sustain the claim. From a *system specification* point of view, we consider the type *concrete syntax* used for specifying systems, that is using the ATL language, abstract syntax (object diagrams or similar), domain-specific modelling language (DSML) or other; the language for specifying queries; their support for *negative-application conditions* (NACs); whether *updates* can be applied to *several objects* matched in the query part of the rule; *control* mechanisms to handle the application of rules, for example application of rules as long as possible (*alap*), only one match per rule *unique*, *arbitrary* selection of the rule to be applied, *rule priorities*, a dedicated *control language*, or other scheduling policies;

Features	FMA-ATL in-place	ATL2010 in-place	SimpleGT	Groove	Henshin	e-Motions
System specification						
Concrete syntax	textual (ATL)	textual (ATL)	textual (other)	graphical (abstract)	graphical (abstract)	graphical (DSML)
Query language	mOdCL	ATL-OCL	SimpleOCL	graph patterns	graph patterns	graph patterns, mOdCL
NACs	OCL (filter)	OCL (filter)	✓	✓	✓	✓
Updates	✓	✗	✓	✓	✓	✓
Control	alap	unique	alap? unique	arbitrary priorities control lang	alap priorities	round-robin
Amalgamation	✓	✗	✗	✓	✓	✗
Rule inheritance	✗	✓	✓	✗	✗	✗
Non-determinism	✓	✗	✓	✓	✓	✓
State-space generation	BFS	✗	✗	DFS, BFS linear	BFS?	BFS
Property specification and verification						
Language	ATL helpers	✗	✗	graphs	OCL	Maude
Model checking	bound. inv, LTL	✗	✗	bound. inv, LTL, CTL	bound.? inv, qualitative probabilistic	bound. inv, LTL statistical

Table 1. Comparison of system specification languages for EMF-based systems.

whether rule application *amalgamation* is supported by using mechanisms that group several transitions in one single transition; strategies available to explore the *state space*, usually depth-first search (DFS), breadth-first search (BFS) or linear; and whether the specification language can model *non-determinism.*

Regarding *verification*, we focus on the *language* used to specify state properties and on model checking techniques supported. Additionally: Groove provides mechanisms for symmetry reduction; Henshin[4] provides support for qualitative model checking with CADP and mCRL2, and stochastic and probabilistic model checking with PRISM; and e-Motions system specifications can model both real-time systems and stochastic systems, the latter class of models can be analysed with statistical model checking using PVeStA.

FMA-ATL, Groove and Henshin support amalgamation mechanisms to reduce the state space. In particular, FMA-ATL achieves this by using lazy rules [2]. However, these tools, together with e-Motions, do no provide support for rule inheritance. By using ATL as front-end language in FMA-ATL, the ecosystem of tools available both for developing ATL transformations (e.g. IDE support, parser) and for analysing them can be reused for facilitating the correct definition of ATL transformations/specifications. Conversely, our tool contributes to that ecosystem as well.

6 Conclusions

Verification of model-based software systems have normally been studied with in-place graph transformation and, up to now, ATL has not been used for this purpose. In this work, we have discussed several drawbacks that hamper the

[4] `http://wiki.eclipse.org/Henshin/State_Space_Tools`

use of the current ATL in-place semantics for modelling and verifying EMF-based systems. In particular, we illustrated why such system specifications are potentially incomplete and unsound w.r.t. the intended behaviour of a system for verification purposes by using a representative example form the literature.

We presented a new in-place semantics for ATL by augmenting FMA-ATL's semantics with a new scheduler rule, by enabling ATL as its front-end language and by linking Maude's verification techniques to ATL. FMA-ATL thus enables the use of ATL for specifying, simulating and verifying both deterministic and non-deterministic systems.

Acknowledgements. The author would like to thank Frédéric Jouault and Massimo Tisi for insightful discussions on the semantics of ATL, and the anonymous reviewers for their observations, which helped improve this work greatly.

References

1. T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *MODELS*, volume 6394 of *LNCS*, pages 121–135. Springer, 2010.
2. A. Boronat. Experimentation with a Big-Step Semantics for ATL Model Transformations. In *ICMT*, volume 10374 of *LNCS*, pages 3–18. Springer, 2017.
3. M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude*. LNCS 4350, 2007.
4. J. S. Cuadrado, E. Guerra, and J. de Lara. Uncovering Errors in ATL Model Transformations Using Static Analysis and Constraint Solving. In *ISSRE*, pages 34–44. IEEE Computer Society, 2014.
5. F. Durán, A. Moreno-Delgado, and J. M. Álvarez-Palomo. Statistical Model Checking of e-Motions Domain-Specific Modeling Languages. In *FASE*, volume 9633 of *LNCS*, pages 305–322. Springer, 2016.
6. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker. *Electr. Notes Theor. Comput. Sci.*, 71:162–187, 2002.
7. A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova. Modelling and Analysis Using GROOVE. *STTT*, 14(1):15–40, Feb 2012.
8. Manuel Roldán and Francisco Durán. The mOdCL evaluator: Maude + OCL. `http://maude.lcc.uma.es/mOdCL/`, 2013. Online; accessed 3 March 2016.
9. A. Rensink, Á. Schmidt, and D. Varró. Model Checking Graph Transformations: A Comparison of Two Approaches. In *ICGT*, volume 3256 of *LNCS*, pages 226–241. Springer, 2004.
10. M. Tisi, S. Martínez, F. Jouault, and J. Cabot. Refining Models with Rule-based Model Transformations. Research Report RR-7582, Mar. 2011.
11. J. Troya, J. E. Rivera, and A. Vallecillo. Simulating domain specific visual models by observation. In *SpringSim*, page 128. SCS/ACM, 2010.
12. D. Wagelaar, M. Tisi, J. Cabot, and F. Jouault. Towards a General Composition Semantics for Rule-Based Model Transformation. In *MODELS*, volume 6981 of *LNCS*, pages 623–637. Springer, 2011.