# Expressive and Efficient Model Transformation with an Internal DSL of Xtend

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

## ABSTRACT

Model transformation (MT) of very large models (VLMs), with millions of elements, is a challenging cornerstone for applying Model-Driven Engineering (MDE) technology in industry. Recent research efforts that tackle this problem have been directed at distributing MT on the Cloud, either directly, by managing clusters explicitly, or indirectly, via external NoSQL data stores. In this paper, we draw attention back to improving efficiency of model transformations that use EMF natively and that run on *non-distributed* environments, showing that substantial performance gains can still be reaped on that ground.

We present *Yet Another Model Transformation Language* (YAMTL), a new internal domain-specific language (DSL) of Xtend for defining declarative MT, and its execution engine. The part of the DSL for defining MT is similar to ATL in terms of expressiveness, including support for advanced modelling contructs, such as multiple rule inheritance and module composition. In addition, YAMTL provides support for specifying execution control strategies. We experimentally demonstrate that the presented transformation engine outperforms other representative MT engines by using the batch transformation component of the VIATRA CPS benchmark. The improvement is, at least, one order of magnitude over the up-to-now fastest solution in all of the assessed scenarios. The software artefacts accompanying this work have been approved by the artefact evaluation committee and are available at http://remodd.org/node/585.

## CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; *Specialized application languages*; *Software performance*;

## KEYWORDS

EMF, Model transformation, ATL, Expressiveness, Performance

**ACM Reference Format:**
Artur Boronat. 2018. Expressive and Efficient Model Transformation with an Internal DSL of Xtend . In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3239372.3239386

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*MODELS '18, October 14–19, 2018, Copenhagen, Denmark*
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-4949-9/18/10…$15.00
https://doi.org/10.1145/3239372.3239386

## 1 INTRODUCTION

Over the past decade the application of MDE technology in industry has led to the emergence of very large models (VLMs), with millions of elements, that may need to be updated using model transformation (MT). This situation is characterized by important challenges [26], starting from model persistence using XMI serialization with the Eclipse Modeling Framework (EMF) [31] to scalable approaches to apply model updates.

An important research effort to tackle these challenges was performed in the European project MONDO [19] by *scaling out* efficient model query and MT engines, such as VIATRA [3] and ATL/EMFTVM [46], building on Cloud technology either directly, by using distributed clusters of servers explicitly, or indirectly, by using NoSQL stores, which streamline the deployment and management of clusters. Among the first type, there are approaches that distribute model queries, e.g. IncQuery-D [32], others parallelize [40] and distribute [2] MT. Among the second type, some approaches compile model transformations to external data stores, e.g. Gremlin-ATL [15], others circumvent the XMI serialization problem by using NoSQL stores, e.g. NeoEMF [16] and MORSA [17].

In this work, we address the problem of transforming VLMs efficiently by *scaling up* core MT techniques instead. In particular, we introduce *Yet Another Model Transformation Language* (YAMTL) as an internal DSL of Xtend [18] and compare it with the core MT engines that are scaled out in the MONDO project, namely VIATRA3 (VIATRA in the rest of the paper) and ATL/EMFTVM, along two dimensions: a qualitative one based on their *expressiveness*, and a quantitative one based on their *performance*.

Regarding *expressiveness*, we discuss that the language is, at least, *as expressive as* ATL for batch MT, including declarative rules with several input/output elements, control mechanisms with lazy rules, priorities and an efficient operation *fetch*, which implements an object resolution strategy based on internal traceability links. In addition, we show that YAMTL provides support for advanced modelling constructs for reuse such as multiple rule inheritance and module composition.

To analyse *performance*, we focus on batch MTs that are executed on demand, instead of being incremental or reactive, by using the batch MT component of the VIATRA CPS benchmark [3, 43]. We demonstrate, via a controlled experiment, that YAMTL substantially outperforms the up-to-now fastest solution of the benchmark by using all of its scenarios. Furthermore, we show that YAMTL is able to perform the benchmark transformation involving more than $10M$ objects and $27.5M$ references in about $32s$. most of the times. A Github repository with all of the software artefacts used in the experiments, including the different solutions, the benchmark harness used and raw results, is available at [13] for the sake of reliability of the empirical study.

By being an Xtend DSL, YAMTL lives in symbiotic relationship with Xtend, enabling the integration of declarative batch (both out-place and in-place) MT with Xtend model-to-text transformation in memory, while benefiting from its tool ecosystem and language expressiveness. This includes Xtend's IDE, typechecker and debugger. YAMTL is available at https://yamtl.github.io.

In the rest of the paper, the presentation of algorithms has a marked functional scent, and readers accustomed to lambda expressions in programming languages with side effects, e.g. as in Java and Xtend, or closures, as in Groovy, should find no trouble. Our findings are presented as follows: a brief introduction to the benchmark transformation and YAMTL are presented in section 2; YAMTL's matching algorithm is presented in section 3; the operational semantics of rules is presented in section 4; the analysis on the language expressiveness and performance of YAMTL is given in sections 5 and 6, resp.; concluding remarks and future work are given in section 7.

## 2 YAMTL

In this section, YAMTL is introduced by providing its concrete syntax and an outline of its semantics by using the batch transformation of the VIATRA CPS benchmark [3, 43], which is available as a third-party public resource [36], including thorough documentation about the models involved and about the mapping rules. The source domain describes a generic infrastructure for cyber-physical systems (CPS) where applications (services) are dynamically allocated to connected hosts. The target domain represents system deployment configurations (DEP) with stateful applications deployed on hosts. The batch transformation derives the initial deployment model from a CPS model. Parts of this transformation are used to present YAMTL but, due to space limitations, a complete definition will not be provided and is available in the experiments repository [13]. Additional YAMTL examples are available in [11].

### 2.1 Concrete Syntax

The declaration of the YAMTL transformation starts by creating an Xtend class that extends the class `YAMTLModule`. Within its constructor, the `header()` of the transformation defines its signature, declaring its input and output models and is declared in the constructor method of the transformation class. Input models that are transformed *in place* are declared with the keyword `.inOut()` instead of `.in()`, and the clause `.out()` has to be skipped. An excerpt of the transformation for the benchmark is shown in Listing 1. In the following, we present the syntax for defining YAMTL rules[1]:

```
new Rule(RNAME)[.inheritsFrom(RNAME_LIST)][.abstract]
    [.lazy|.uniqueLazy][.transient]
    {.in(ENAME,TYPE)[.with(ENAME_LIST)]
        [(.filter(FILTER)|.derivedWith(QUERY))].build}+
    [.filter(FILTER)]
    {.using(VNAME,QUERY)}*
    {.out(ENAME,TYPE,ACTION)[.overriding()][.drop].build}+
    [.endWith(ACTION)][.priority(P)].build
```

[1]The following EBNF notational conventions have been used: () for grouping elements, | for denoting alternatives, [] for denoting optionality (0..1), {}* for denoting repetition (0..*), and {}+ for denoting repetition (1..*)

Rules are declared in a list, enclosed in `#[]`, in the `ruleStore`. A rule is declared with a name *RNAME* and it consists of: *input elements*, defining the input pattern (or left-hand side) of the rule; a `filter` condition; *output elements*, defining the output pattern (or right-hand side); `using` elements initializing *local variables* and an `endWith` block defining actions to be performed at the end of the rule execution.

```
class Cps2DepYAMTL extends YAMTLModule {
  val CPS = CyberPhysicalSystemPackage.eINSTANCE
  val DEP = DeploymentPackage.eINSTANCE
  new () { // constructor of the transformation module
    header().in('cps', CPS).out('dep', DEP)
    ruleStore(#[
      new Rule('CyberPhysicalSystem_2_Deployment')
        .in('cps', CPS.cyberPhysicalSystem).build
        .out('dep', DEP.deployment, [
           val cps = 'cps'.fetch as CyberPhysicalSystem
           val dep = 'dep'.fetch as Deployment
           out.hosts += cps.hostTypes
             .flatMap[instances].fetch as List<DeploymentHost>
         ]).build      // initialize an output element
      .build,          // initialize a rule
      // other rules
    ])
    helperStore(#[
      new Helper('waitingTransitions') [
        val Map<String,List<Transition>> map = newHashMap
          CPS.transition.allInstances.forEach[ transition |
            // initialization of map
          ]
        return map
      ].build])}
  def isWaitSignal(String action) {
    action.startsWith("waitForSignal") }
  // other helpers here...
}
```

**Listing 1: Excerpt of Benchmark Transformation in YAMTL**

There are two types of input elements: *matched*, which are initialized by means of YAMTL's matching algorithm; and *derived*, which are initialized by means of a contextual query, dependent on, at least, one matched element. Rules are declared with, at least, one matched input element, declaring the name of the object variable *ENAME* and its type *TYPE* (an `EClass` instance, where `EClass` embodies the concept class in EMF). An optional local filter condition can be defined for a matched input element by using the clause `.filter()` with a lambda expression *FILTER* with type `() => boolean`. An input element can be declared as derived with the clause `.derivedWith()`, where a *QUERY* lambda expression with type `() => EObject` is used to compute the value of the match. When several input elements are declared, a common filter condition, involving all input elements, can be declared using the clause `.filter(` *FILTER*`)`. Dependencies among matched input elements can be declared with the clause `.with()`, enabling the use of objects matched in previous input elements in the filter expression of subsequent input elements.

Local variables to be used in the evaluation of both output elements and the `.endWith()` block are initialized with elements `.using()` by declaring the name of the variable *VNAME* and a *QUERY* lambda expression with type `()=>Object`.

Output elements are declared with the clause `.out()` by providing a name *ENAME*, a type *TYPE* (an instance of `EClass`), and a lambda expression, with type `()=>void`, representing the procedure to be used to initialize/update the output element. The rule that creates a

deployment root object from a cyberphysical system object, shown in Listing 1, is defined with an input element `cps` with no filter condition and with an output element `dep`, whose lambda expression is defined between `[]`. In the lambda expression, rule elements can be accessed with the operation `fetch`. The logic of the rule is simple in this case, the `instances` belonging to all `hostTypes` in the input element `cps` correspond to `DeploymentHost` objects, which are obtained using the operation `fetch`. That is, the operation fetches the output objects to which input objects have been *mapped* using another rule. For matched rules, the user does not need to provide any hint as rules are *declarative* and applied internally by the MT engine. The operation `fetch` lays at the core of the execution semantics of YAMTL and will be presented in detail in section 4.

The clause `.endWith()` defines an optional *ACTION* lambda expression with type `()=>void`, which can refer to any element of the rule and to local variables. The clause `endWith` is not strictly required and is provided for convenience only, facilitating grouping the initialization of different output elements in one single block, which may help in achieving a concise presentation of rules.

By default, rules are matched using input elements, which may have local filter conditions, and the rule filter condition. However, a rule that is either `lazy` or `uniqueLazy` will not be matched automatically. Lazy rules can only be applied when the match for matched input elements is explicitly provided by using the `fetch` command as explained in the following section. A rule that is `uniqueLazy` is applied only if it has not been applied already for the same input match. Such constraint does not apply to a rule that is `lazy`.

A rule that is `abstract` can neither be matched automatically nor applied. A rule can specialize multiple parent rules using the clause `.inheritsFrom(`*RNAME_LIST*`)`, where the order of inheritance as specified in the list *RNAME_LIST* is important. Rules can also be annotated with the clause `.priority(`*P*`)`, where *P* is a `long` value. The YAMTL matching algorithm will apply rules with lower priority first. A rule that, in addition, is `transient` does not persist its output elements when the target model is flushed to physical storage.

On the other hand, YAMTL provides the operation `allInstances`, defined for all of the `EClass` instances of the `in` (or `inOut`) model, facilitating the creation of OCL-like queries in Xtend lambda expressions. YAMTL also allows users to define attribute helpers in transformations for computing values during the initialization of the model transformation. Such helpers are defined in the constructor of the transformation and are linked to the YAMTL module with the operation `helperStore`. An attribute helper consists of a name and a *QUERY* lambda expression. In the example of Listing 1, the attribute caches all transitions whose attribute `action` value is `waitForSignal` using a map. Operation helpers, like `isWaitSignal`, are defined using normal Xtend methods.

## 2.2 Semantics Outline

The execution semantics of YAMTL is defined by using *configurations* $C$, which form the execution context that is used to apply a transformation. YAMTL's configurations involve: the input and output models (or only one if the transformation is executed *in place*); a map *locations* associating each type involved in the transformation with its extent of objects, providing potential locations of the model where rules may be applied; an environment $\Gamma$ as a, possibly empty, set of variable assignments; a rule store as a non-empty list of rules; a helper store as a, possibly empty, list of attribute helpers; a *matchPool* with matched rules; and an *eventPool* tracking the order in which transformation steps are applied. A component $c$ of a configuration $C$ can be accessed using the prefix form $c(C)$.

When loading a YAMTL MT definition, YAMTL initializes transformation rules by inserting their abstract representation, presented in section 3.1, in the rule store. The initialization of a rule involves the computation of references that are encoded through names in the concrete syntax, e.g. the list of parent and children rule names become lists of references to the objects representing them internally. Additional derived references, which are used in the execution semantics of rules, are computed, e.g. all parents/children of a rule. In specialized rules, derived input elements whose query expression is given in a parent rule are updated with it, and output elements are updated by building the list of actions to be performed in leftmost, top-down order along the rule inheritance hierarchy for that rule.

Once the rule store is initialized, rules are typechecked both for rule inheritance, considering the constraints in [48], and for `with` dependency cycles. If no error arises, computation is performed as follows: *(1)* the map *locations* is initialized; *(2)* rules in the rule store are ordered using declared priorities; *(3)* an environment $\Gamma$ is initialized with the value of attribute helpers; *(4)* the *matchPool* is initialized with all rule matches; *(5)* the objects of the input model are transformed by applying rules to scheduled matches.

*Dispatch Semantics.* In the following, an informal description of the matching process of a rule is provided. This process is described more precisely in section 3. YAMTL finds a match for a rule by first mapping matched input elements to objects that satisfy the corresponding filter condition in the order defined in `with` dependencies – e.g. in the clause `.inElement(V1,T1).with(#[V2])`, the element with name `V2`, from the singleton list `#[V2]`, is matched earlier than `V1`. If no dependency is defined, the elements are ordered by the size of their type extent. If such a partial match exists, YAMTL attempts to complete the total match by evaluating query expressions for derived elements in the order in which they were declared. If a query cannot be resolved to an object, the match for the rule is considered invalid. If a total match is found, the satisfaction of that match is finally asserted by the rule filter condition, which is `[true]` by default. The user is to keep in mind the order of precedence declared among input elements using `with` clauses and YAMTL will throw a run-time exception when trying to fetch an input element that has not been matched yet. To use YAMTL efficiently, two design principles need to be applied: matched input elements should only be defined for matching objects that are not related to each other via references (when they should be defined as *derived elements*); and rule filter conditions should be pushed down as element filter conditions as much as possible in order to help the matching algorithm prune invalid matches as soon as possible.

*Execution Semantics.* In the following, an outline of the execution semantics of a rule is given. Such operational semantics is detailed in section 4. An output element with a new name prescribes the creation of a new instance and the expression *ACTION* is a procedure describing how to *initialize* it. On the other hand, when the

output element refers to an input element, the input element is taken as the output element – i.e. the input element is also an output element,– in which case the expression *ACTION* is a procedure describing how to *update* it but no object is created afresh. In both cases, the mapping from input match to output match is *traced* as a transformation step. *ACTION* expressions, which can refer both to input/output elements and to local variables, should only define assignments for features of output elements. That means that all objects should be created as output elements. If an ad-hoc object is created manually (by using, e.g. an object factory) in an action expression and assigned to an output object, the ad-hoc object is not traced. The only inconvenience that this creates is that non-traceable objects cannot be fetched from a different rule. Moreover, elements that are used as both input and output can be removed using the clause `.drop`, which denotes *delete cascade* semantics – both the object and its contents following containment references are removed.

*Multiple Rule Inheritance.* When using rule inheritance, rules are expected to be covariant both in input elements and in output elements with respect to inheritance relationships. The semantics of rule with respect to inheritance is as follows: in matched input elements, filter expressions are inherited using a leftmost top-down evaluation strategy w.r.t. the inheritance hierarchy defined in clauses `inheritsFrom`; in derived input elements, derivation expressions for derived input elements are overriden if declared or simply inherited otherwise; in output elements, action expressions are also inherited following a leftmost top-down evaluation strategy w.r.t. the inheritance hierarchy by default, and they can be *overriden* by using the qualifier `overriding` in the corresponding output element of a descendant rule. In a specialized rule, YAMTL expects: the elements of the parent rule to be declared in the descendant rule; abstract rules to be specialized; and concrete rules to have elements in the output pattern that are typed with concrete classes. A declaration error is thrown if those constraints are violated. When a specialized rule inherits the same output element from two different parent rules, situation known as the *diamond problem* [33], YAMTL detects the situation and warns the user but the model transformation proceeds using inheritance semantics as explained above.

*Module Composition.* A YAMTL module can be imported in other Xtend/Java classes by instantiating its main class. On the other hand, a YAMTL module can import any JVM library. Moreover, YAMTL modules can be composed by using the Xtend `extends` clause, i.e. by creating a subclass of another module. So, we distinguish between *module import*, for reusing a YAMTL MT as a library, and *module extension* for composing modules.

When other YAMTL modules are *extended*, the initialization of rules and attribute helpers starts from the leaf modules, which do not extend other modules, and proceeds along the `extends` hierarchy from parent to descendant. When a specializing module declares a rule that is already defined in the super-module, the existing rule is *redefined* by the new rule if they have *compatible* signatures. That is, the rule in the importing module only defines all of the elements of the rule in the imported module, and for each of such elements, the type of the element in the importing module must be a *subtype* of the type of the element in the imported module. Attribute helpers are simply redefined as they do not have parameters. Other helpers defined as Xtend operations follow the usual specialization semantics in Xtend. Imported rules and attribute helpers that are *redefined* cannot be accessed any longer.

## 3 DISPATCH SEMANTICS

In this section, the algorithm for matching YAMTL rules is presented. First, we introduce notation for representing input element matches and rule matches internally. Then, we provide the notion of *valid match* both for input elements and for the input pattern of a rule. We continue by describing the matching procedure to find matches for a given rule.

### 3.1 Abstract Syntax

The execution of a rule, including its matching, always requires the presence of an environment $\Gamma$ in the current YAMTL configuration $C$. During the matching process, it is used to evaluate filter conditions and derivation expressions that may refer to variables that have been initialized while evaluating other input elements. This is the reason why `with` dependencies *usually* need to be declared among input elements if they exist – `with` dependencies can be avoided if all conditions are defined in the rule filter, as opposed to be in an input element filter.

We use the vector notation $\overrightarrow{c}$ for denoting variables that range over collections, for both lists and sets, of components $c$. We will disambiguate the type of collection used when accessing its elements. For lists, given a list $L$, we will use $[]$ to denote the empty list, $h :: L$ for accessing the head of a list, and $L :: l$ for accessing the last element of a list. For sets, given a set $S$, we will use $\emptyset$ to denote the empty set, and $\{e\} \cup S$ for accessing an element from a set. Note that in expressions of the form $c :: \overrightarrow{c}$ there is no confusion: $c$ refers to the head of the list and $\overrightarrow{c}$ refers to the tail of the list. Similarly, in the expression $\{c\} \cup \overrightarrow{c}$, $c$ is an element of the set and $\overrightarrow{c}$ denotes a set variable. The imperative assignment operation $:=$ is used to update variables, and can be used in lambda expressions. The sequencing operation ; is also valid in a lambda expression.

*Definition 3.1 (Rule (abstract syntax)).* A YAMTL rule $r$ is internally declared as

$$r : (mod, abstract, trans, parents, children, \overrightarrow{in}, f, \overrightarrow{u}, \overrightarrow{out}, endWith)$$

where $r$ is the name of the rule, $mod$ is one of the modifiers in {`matched`, `lazy`, `uniqueLazy`}, *abstract* and *trans* are a boolean flags indicating whether the rule is abstract or transient, resp., *parents* is the list of parent rule names, *children* is the list of children rule names, $in \in$ `Input` is an input element, $f \in$ `(Env)=>boolean` is the global filter of the rule, $u \in$ `Using` is a variable declaration, $out \in$ `Output` is an output element, and $endWith \in$ `(Env) => void` is the procedure that is executed at the end of the rule.

Given a rule $r$, its components are accessed using the name of the component $c$ in prefix form. For example: $abstract(r)$ determines whether the rule is abstract or not. In addition, we will use the helper function $isTopRule(r)$ for checking whether the rule has no parents. As an additional remark, the type of lambda expressions used in the internal abstract representation of rules use the type `Env` explicitly, which is not required when declaring rules using YAMTL's concrete

syntax, as presented in section 2.1. Moreover, $in$ may refer either to a matched input element $in_m$ or to a derived input element $in_d$. When interested in distinguishing matched input elements $in_m$ from derived ones $in_d$, we replace $in$ with $(in_m, in_d)$ without loss of generality. On the other hand, for an object $\varsigma \in$ `EObject`, we use $type(\varsigma)$ to get its type, an `EClass` instance $c$, and $allSuperTypes(c)$ for obtaining the set of all supertypes of an `EClass` $c$.

*Definition 3.2 (Input Element (abstract syntax)).* Input elements are represented with the notation $in_m : (type, wl, f)$ for matched input elements, and $in_d : (type, wl, d)$ for derived input elements, where $in_m$ and $in_d$ are names, $type \in$ `EClass` is the type associated with the input element, $wl$ is a list of input element names corresponding to `with` dependencies, $f \in$ `(Env)=>boolean` is the local filter condition of a matched input element, and $d \in$ `(Env)=>EObject` is the derivation query of a derived input element.

*Definition 3.3 (Input Element Match).* Given a rule $r$, a variable assignment $in \mapsto \varsigma$ is a match for the input element $in \in in(r)$ if $type(in) = type(\varsigma) \lor type(in) \in allSuperTypes(type(\varsigma))$.

*Definition 3.4 (Input Rule Match).* Given a rule $r$, a match for the rule is a named list $m_r : \overrightarrow{in \mapsto \varsigma}$ of input element matches $in \mapsto \varsigma$ for each $in \in in(r)$.

*Definition 3.5 (Output Element (abstract syntax)).* Output elements are represented with the notation $out_m : (type, drop, overriding, action)$ where $out$ is a name, $type \in$ `EClass` is the type associated with the output element, $drop$ is a boolean flag indicating whether the corresponding object needs to be dropped, $overriding$ is a boolean flag indicating whether the element overrides the corresponding one in the parent rule (if any), and $action \in$ `(Env)` $\Rightarrow$ `void` is the action expression that either initializes or updates the features of an output element.

The notions *output element match* and *output rule match* for output elements are similar to those for input elements. The notation is abused by using $r$ ($in$ and $out$) both for the name of the rule (of an input element and of an output element, resp.) and for the rule itself (for the input element and for the output element themselves, resp.). The notions presented above are purely syntactic. In the following subsection, the notion of *valid match* is introduced both for input elements and for rules, characterizing the set of matches for a given rule.

In what follows, most of the algorithms are presented using an equational presentation. That is, algorithms are specified declaratively using a set of equations of the form $LHS = RHS$ *if* $Cond$, where: $LHS$ is a pattern with variables that may be used to decompose collection values using list and set constructor operators (e.g. to fetch the head of a list with an expression $h :: L$); $Cond$ is a conjunction of equational predicates (of the form $L = R$, where $L = true$ is abbreviated as $L$), which are used both to ascertain constraints and to match fresh variables (the predicates in the conjunction are evaluated from left to right), and which are defined over previously matched variables in $LHS$ or in $Cond$; and $RHS$ is the action to be performed with the variables matched earlier either in $LHS$ or in $Cond$. An equation is applied only if $LHS$ matches a term (an expression) and the condition $Cond$ is satisfied[2]. The cases that are not presented explicitly as equations, when no other equation applies, are reduced to the *no-op* expression (e.g. ; in Java) and parameters are passed by value.

[2]This notation permits the compact representation of functions and is typical in functional programming languages like Maude [14], where this notation is borrowed from. An equation can be regarded as a *case* of either a `match` statement in Haskell or Scala, augmented with a condition, or a `switch` statement in Java/Xtend, augmented with pattern matching and a condition.

## 3.2 Valid Matches

For a match $in \mapsto \varsigma$ of an input element $in$ to be *valid*, two cases are considered: *(1)* when the input element is matched, in which case the match needs to satisfy its local filter condition, which may refer to the matched object; and *(2)* when the input element is derived, in which case its derivation query must return the object included in the match. More precisely:

*Definition 3.6 (Valid Input Element Match).* Given a rule $r$ and an environment $\Gamma$, a match $in \mapsto \varsigma$ for $in \in in(r)$ and $\varsigma \in$ `EObject` is valid, denoted by

$$(in \mapsto \varsigma) \vDash_\Gamma in,$$

and defined as follows:

$$\begin{cases} f(in)(\Gamma \cup \{in \mapsto \varsigma\})) = true & \text{if } in \in in_m(r) \\ d(in)(\Gamma) = \varsigma & \text{if } in \in in_d(r) \end{cases}$$

To check whether a match is *valid* for a given rule, that is, whether it satisfies its input pattern $in(r)$, the list of input elements is traversed in sequence (input elements are ordered according to both `with` dependencies and the size of their type extent – in that order, – and matched elements are traversed earlier than derived ones), checking that for each input element there is a valid match, in equation MC1 below. Every time a valid match is found, it is inserted in the environment in order to evaluate subsequent filter and derivation expressions. When all input elements are traversed, the match is valid if the the global filter of the rule is satisfied, in equation MC0.

*Definition 3.7 (Rule Match).* Given a rule $r$ and an initial environment $\Gamma$, a match $m_r : \overrightarrow{in \mapsto \varsigma}$ is valid, denoted by

$$\overrightarrow{in \mapsto \varsigma} \vDash_\Gamma in(r)$$

as defined by the following equations:

$$\overrightarrow{in \mapsto \varsigma} \vDash_\Gamma [\,] \ = \ f(r)(\Gamma) \qquad \text{(MC0)}$$

$$\{(in \mapsto \varsigma)\} \cup \overrightarrow{in \mapsto \varsigma} \vDash_\Gamma in :: \overrightarrow{in} \ = $$
$$(in \mapsto \varsigma \vDash_\Gamma in) \land \overrightarrow{in \mapsto \varsigma} \vDash_{\Gamma \cup \{(in \mapsto \varsigma)\}} \overrightarrow{in} \qquad \text{(MC1)}$$

To present YAMTL's matching procedure in the next subsection, we need to introduce the notion of *transformation step* that is used for scheduling rule matches. A transformation step, denoted by $r : \overrightarrow{in \mapsto \varsigma} \to \overrightarrow{out \mapsto \varsigma}$ consists of a labelled pair of two matches, the match for the input pattern of the rule, which enables its application, and the match for the output pattern of the rule, with the objects that result from applying the rule. Now that the notion of a valid match for a rule, without considering rule inheritance, has been presented, we are ready to present how YAMTL finds them in the following subsection.

### 3.3 Matching Procedure

The YAMTL MT engine keeps a rule store *rs* in memory in the form of a non-empty list of rules that are ordered according to their priorities, the one with lower priority is served first. The matching procedure *match* produces matches and schedules them for deferred execution, by being parameterized with a lambda expression $\lambda_s$ that dictates how to schedule a match. For now, let's assume that, given an environment $\Gamma$ and a rule match $\overrightarrow{in \mapsto \varsigma}$, the lambda expression $\lambda_s$ records the match in the *matchPool* for posterior processing. This lambda expression will be discussed in the following subsection.

In the following we define YAMTL's matching procedure *match*. Two helper procedures are used, $match_m$ for processing matched input elements, and $match_d$ for processing derived input elements.

Given a rule $r$, an environment $\Gamma$ and a scheduling lambda expression $\lambda_s$, the procedure *match* starts by processing the rule if it is both a matched rule (i.e. not lazy) and a top rule (i.e. with no parents) in Eq. M1.

$$match(r, \Gamma, \lambda_s) = match_m(r, in_m(r), \Gamma, \lambda_s)$$
$$if\ \ mod(r) = \texttt{matched} \wedge isTopRule(r) \quad \text{(M1)}$$
$$match(r :: rs, \lambda_s) = match(r, \lambda_s) \wedge match(rs, \lambda_s) \quad \text{(M2)}$$

Matched input elements $in_m(r)$ are processed by the operation $match_m$, which first traverses the list of matched input elements in Eq. M4 in order, trying to form a match with each object $\varsigma$ in the extent of the type of the input element. Each object $\varsigma$ is traversed in Eq M6. If, for a particular object $\varsigma$, the match $in_m \mapsto \varsigma$ is valid for the input element $in_m$, the procedure proceeds to assess the next matched input element after inserting the valid match in the environment, as defined in Eq M5, until no more elements are found. At that point, the procedure $match_m$ starts by computing derived matches in Eq. M3. When a match is found to be invalid in Eq. M5, the search for the completion of that match stops by doing *nothing*.

$$match_m(r, [], \Gamma, \lambda_s) = match_d(r, in_d(r), \Gamma, \lambda_s) \quad \text{(M3)}$$
$$match_m(r, in_m :: \overrightarrow{in_m}, \Gamma, \lambda_s) =$$
$$match_o(r, in_m, type(in_m).\texttt{allInstances}, \overrightarrow{in_m}, \Gamma, \lambda_s) \quad \text{(M4)}$$
$$match_o(r, in_m, \varsigma, \overrightarrow{in_m}, \Gamma, \lambda_s) = match_m(r, \overrightarrow{in_m}, \Gamma \cup \{in_m \mapsto \varsigma\}, \lambda_s)$$
$$if\ \ (in_m \mapsto \varsigma) \vDash_\Gamma in_m \quad \text{(M5)}$$
$$match_o(r, in_m, \varsigma :: ol, \overrightarrow{in_m}, \Gamma, \lambda_s) =$$
$$match_o(r, in_m, \varsigma, \overrightarrow{in_m}, \Gamma, \lambda_s)\ \wedge\ match_o(r, in_m, ol, \overrightarrow{in_m}, \Gamma, \lambda_s) \quad \text{(M6)}$$

Derived input elements are processed orderly, in Eq M8, by checking whether the input element derivation expression produces a valid match, in which case the next derived input element is assessed, after inserting the valid match in the environment, until no more derived input elements are left. At that point, Eq. M7 applies the scheduling lambda expression if the rule filter $f(r)$ is satisfied with the environment $\Gamma$.

$$match_d(r, [], \Gamma \uplus \overrightarrow{in \mapsto \varsigma}, \lambda_s) = \lambda_s(r, \overrightarrow{in \mapsto \varsigma})$$
$$if\ f(r)(\Gamma \uplus \overrightarrow{in \mapsto \varsigma}) = true \quad \text{(M7)}$$
$$match_d(r, in_d :: \overrightarrow{in_d}, \Gamma, \lambda_s) =$$
$$match_d(r, \overrightarrow{in_d}, \Gamma \cup \{in_d \mapsto d(in_d)(\Gamma)\}, \lambda_s)$$
$$if\ \ (in_d \mapsto \varsigma) \vDash_\Gamma in_d \quad \text{(M8)}$$

Therefore the procedure *match* finds all matches for the top rules, i.e. with no parents, in the rule store of the MT engine. In the next subsection, the matching procedure is extended with multiple rule inheritance in the scheduling lambda expression $\lambda_s$.

### 3.4 Matching with Multiple Rule Inheritance

For each selected top rule and valid match, Eq. M7 applies $\lambda_s$, which then triggers a downward search, along the rule inheritance hierarchy, for the most concrete rules that are applicable to the given match using the function *concretize*. At the end of this search the resulting rule set *rs* must be a singleton set in order for YAMTL to proceed with the execution of the transformation, as there would be several rules applicable to the same match otherwise. The match is then scheduled for the resulting concrete rule if it is not abstract, by inserting a transformation step with the rule match and an empty output match in the *matchPool*, which indexes transformation steps by their rule match, as defined in Eq. SCH[3].

$$\lambda_s(r, \overrightarrow{in \mapsto \varsigma}) =$$
$$matchPool(C) := matchPool(C) \uplus \{(r : \overrightarrow{in \mapsto \varsigma} \to \emptyset)\}$$
$$if\ \{cr\} = concretize([r], \overrightarrow{in \mapsto \varsigma}, \emptyset)\ \wedge\ not(abstract(cr)) \quad \text{(SCH)}$$

The parameters of the function *concretize* are: a list *rl* of rules, the match $\overrightarrow{in \mapsto \varsigma}$, and a set *rs* of concrete rules and it is defined as follows.

Eqs. C2 and C3 traverse the given list *rl* of rules, and Eq. C3 merges the resulting sets obtained from different search branches. In Eq. C1, for each rule $r$ for which the given match $\overrightarrow{in \mapsto \varsigma}$ is valid, the search process continues with the descendant rules $children(r)$ after removing the parent rule from the set *rs* of selected rules while piggybacking the selected rule $r$.

$$concretize(r, \overrightarrow{in \mapsto \varsigma}, \Gamma, rs) =$$
$$concretize(children(r), \overrightarrow{in \mapsto \varsigma}, \Gamma, (rs \backslash \{parents(r)\}) \cup \{r\})$$
$$if\ \overrightarrow{in \mapsto \varsigma} \vDash_\Gamma in(r) \quad \text{(C1)}$$
$$concretize([], \overrightarrow{in \mapsto \varsigma}, \Gamma, rs) = rs \quad \text{(C2)}$$
$$concretize(r :: rl, \overrightarrow{in \mapsto \varsigma}, \Gamma, rs) =$$
$$concretize(r, \overrightarrow{in \mapsto \varsigma}, \Gamma, rs) \cup concretize(rl, \overrightarrow{in \mapsto \varsigma}, \Gamma, rs) \quad \text{(C3)}$$

By the end of the finite search process, the most concrete rule that matches the object will be selected as the candidate rule for

[3]The assignment is valid in a lambda expression because we are updating the feature of the immutable object $C$ – more specifically, of an object $C$ whose reference is immutable.

the match if it is not abstract. If, by the end of this process, several rules are selected, a run-time error is thrown halting the execution as the same object can be matched by several rules. This algorithm imposes a leftmost, top-down strategy, w.r.t. the rule inheritance hierarchy, as defined in clauses `inheritsFrom`, for evaluating filter conditions. Note that, the search strategy, which is guided by depth and not by breadth, does not result in *invalid* matches as all branches need to be explored anyway and the search is forced to be confluent. Hence, once a unique concrete rule is selected, YAMTL ensures that the match satisfies the input pattern of all parent rules.

Finally, that rule will be the one applied for the match by YAMTL in the lambda expression $\lambda_s$. Note that $\lambda_s$ has not been presented as a pure lambda expression, as it stores the match in the *matchPool*, which is not passed as argument. For a complete functional presentation, all of the $match_X$ expressions (both in *LHS* and in *RHS*), in Eqs. M1-M8, need to be augmented with the *matchPool* as a parameter.

## 4 EXECUTION SEMANTICS

Declarative matched and lazy rules are executed by applying the operation `fetch` to the input match of a rule. The operation `fetch` is totally defined for rule matches that have been scheduled using $\lambda_s$ in the matching process and defines a *check-enforce semantics* for YAMTL rules: when the `fetch` operation is used, it will trigger the creation of objects in the output match of the corresponding transformation step, if they are not present. This operation is overloaded. We start by discussing its most general variant, $\overrightarrow{in \mapsto \varsigma}$`.fetch(r,`$out$`)`, where $r$ is a rule name, – $out$ is the name of an output element of rule $r$. The overloaded variants are discussed at the end of the section.

Given a rule match, the operation `fetch` accesses its corresponding transformation step in the *matchPool*, if it exists, and updates it, with the assignment operator :=, by applying the *reduce* operation, as defined in Eq. FETCH-MATCHED. In this equation, the expression $in_m(\overrightarrow{in \mapsto \varsigma})$ restricts the match $\overrightarrow{in \mapsto \varsigma}$ to its matched variables. Then, it returns the value of the element *out* of its output match $\overrightarrow{out \mapsto \varsigma}$ in the expression $out(\overrightarrow{out \mapsto \varsigma})$.

$$\begin{aligned}
\overrightarrow{in_m \mapsto \varsigma}\,.\,fetch(r, out) \;=\; & \\
& matchPool(C) := \{reduce(r : \overrightarrow{in \mapsto \varsigma} \to \overrightarrow{out \mapsto \varsigma})\} \uplus matchPool; \\
& out(\overrightarrow{out \mapsto \varsigma}) \\
\textit{if}\ & \{(r : \overrightarrow{in \mapsto \varsigma} \to \overrightarrow{out \mapsto \varsigma})\} \uplus matchPool = matchPool(C) \\
& \wedge\ \overrightarrow{in_m \mapsto \varsigma} = in_m(\overrightarrow{in \mapsto \varsigma}) \qquad \text{(FETCH-MATCHED)}
\end{aligned}$$

$$\begin{aligned}
\overrightarrow{in_m \mapsto \varsigma}\,.\,fetch(r, out) \;=\; & \\
& match_d(r, in_d(r), \Gamma \cup \overrightarrow{in_m \mapsto \varsigma}, \lambda_s); \\
& \overrightarrow{in_m \mapsto \varsigma}\,.\,fetch(r, out) \qquad \text{(FETCH-LAZY)}
\end{aligned}$$

For lazy rules, when the input match cannot be found in the *matchPool* the Eq. FETCH-LAZY initializes a transformation step, with the given input match $\overrightarrow{in \mapsto \varsigma}$, in the *matchPool* and then applies Eq. FETCH-MATCHED, to obtain the corresponding object. The *matchPool* is accessed by using matched input elements, and derived matches are optional when invoking an operation `fetch`, as they are uniquely identified by matched input elements.

The operation *reduce* is used to perform the actual transformation step, if it has not been done already. Eq. R1 returns the transformation step as is when the output match is already present. Without considering rule inheritance yet, Eq. R2 performs the actual transformation step by applying *reduce* on the rule match, according to the following algorithm:

1. The output match $\overrightarrow{out \mapsto \varsigma}$ is initialized by inserting a new element match $out \mapsto \varsigma$ for each element *out* of the output pattern of the rule. The value $\varsigma$ of the new element match will be a new instance of type $type(out)$ by default, unless the out element refers, by name, to an input element. In that case, the output element match is initialized with the value of the corresponding input match.
2. The initial environment $\Gamma$, where values of attribute helpers are cached, is updated by inserting: *(a)* the rule input match $\overrightarrow{in \mapsto \varsigma}$, *(b)* the recently created output match $\overrightarrow{out \mapsto \varsigma}$, and *(c)* assignments for local variables (with derivation queries defined in the `using` block of the rule $r$).
3. For each element *out* in the output match, the lambda expression $action(out)$ is applied to the environment $\Gamma$, performing an update if the output element refers to an input one, or an initialization of features otherwise. When the output element defines a *drop*, only possible if the output element refers to an input element, the object is deleted together with its contents.
4. Finally, the lambda expression $endWith(r)$ is applied to the environment $\Gamma$.

$$reduce(r : \overrightarrow{in \mapsto \varsigma} \to \emptyset) = (r : \overrightarrow{in \mapsto \varsigma} \to (reduce(r, \overrightarrow{in \mapsto \varsigma}))) \tag{R1}$$

$$reduce(t) = t \quad otherwise \tag{R2}$$

In the following, we discuss overloaded forms of the operation `fetch`. The operation `fetch` is used for obtaining objects that have been created in output elements of other rules, either matched or lazy. In the case of matched rules, it can be used either without arguments, when the output pattern of the rule only contains one element, or with the name of the output element to be retrieved otherwise. In the case of lazy rules, in addition, the operation `fetch` needs to know which rule to apply as the match for the rule needs to be computed. When the input pattern of the rule contains a single matched element – it could have several derived elements – the operation `fetch` also works for object references, and the match is inferred automatically. An outline of the different possibilities is summarized in Table 1. Although not included in the table, `fetch` is also overloaded for lists of objects (and matches), mapping each input object (or match) to an output object. Furthermore, `fetch` is also overloaded for *fetching* variable values from the environment $\Gamma$ in lambda expressions with access to it, e.g. in an action expression.

## 5 EXPRESSIVENESS ANALYSIS

In this section, the MT languages involved in the performance analysis are discussed from an expressiveness point of view, facilitating *fairness* in the comparison. For a more general discussion on the

**Table 1: Variants of the Operation `fetch`**

| $\lvert in_m(r) \rvert$ | $\lvert out(r) \rvert$ | rule type | fetch expression |
|---|---|---|---|
| 1 | 1 | matched | $o\,.\,fetch$ |
| 1 | >1 | matched | $o\,.\,fetch(out)$ |
| 1 | 1 | lazy | $o.fetch(r)$ |
| 1 | >1 | lazy | $o\,.\,fetch(r,\ out)$ |
| >1 | 1 | matched | $\overrightarrow{in \mapsto \varsigma}\,.\,fetch$ |
| >1 | >1 | matched | $\overrightarrow{in \mapsto \varsigma}\,.\,fetch(out)$ |
| >1 | 1 | lazy | $\overrightarrow{in \mapsto \varsigma}\,.\,fetch(r)$ |
| >1 | >1 | lazy | $\overrightarrow{in \mapsto \varsigma}\,.\,fetch(r,\ out)$ |

expressiveness of MT languages, the reader is referred to [24] for a recent survey.

VIATRA [3] provides support for incremental and reactive MT [3, 44] that uses incremental graph queries [4, 41]. A declarative MT is defined as a rule specification using an Xtend internal DSL, where each rule consists of model queries, defined in EMF-IncQuery, and actions, defined in Xtend. While VIATRA's IDE consists of Xtext-generated tooling, YAMTL benefits from Xtend's own IDE. Both of them use Xtend for defining MT.

The ATL virtual machine [25] was the first to provide low level execution primitives to support MT. It has been extended to support lazy evaluation of OCL expressions and (target) incremental MT in the ReactiveATL MT engine [30]. EMFTVM [46] is an EMF-native MT engine, with a simpler architecture, that provides a low-level language supporting *mapping* MT in ATL and *rewriting* transformations in SimpleGT, a simple graph transformation language.

Expressiveness of MT, as supported by the aforementioned transformation engines, is summarized in Figure 1 along the following dimensions: *paradigm*, referring to the type of MTs supported (on-demand, incremental, reactive); *query pattern*, describing the language used for defining rule input patterns and some features that are normally used when defining such patterns (capturing multiobjects – or collections of objects,– recursion and negative application conditions); type of *rules*, including the cardinality of input and output patterns, how *traceability* is supported, and *control* mechanisms; advanced constructs for *reuse*, at the MT level (module import/extension, higher-order transformation – HOT), at the rule level, and at the expression level (normally referring to the value resulting from the evaluation of the expression).

Currently, YAMTL does not support HOTs and the language does not prevent the user from modifying the source model in an output action, which could introduce inconsistencies. In a MT, several source and target models can be used but only one source model and one target model can be traced implicitly by rules. From the usability point of view, ATL offers a more concise syntax for developing MT than YAMTL does. However, the Xtend tool ecosystem is more mature than that of ATL. Below, we provide a brief outline of the dispatch and execution semantics in the MT engines under study.

## 5.1 Dispatch Semantics

EMF-IncQuery uses both local search-based pattern matching and incremental pattern matching. While the first starts the matching process from a single node and then completes the match step-by-step with the neighbouring nodes and edges following a *model-sensitive search plan* [45], the second one [4, 41] relies on caching results sets of queries, providing an order of magnitude faster re-evaluation time [42]. The query language allows the choice of query strategy for each defined pattern. YAMTL's matching algorithm is based on local search where the search plan is programmed with *(1)* dependencies `with`, *(2)* local/global filters and *(3)* rule inheritance, as explained below. In addition, it is model-sensitive by considering the size of the type extent in each input pattern of a rule.

EMFTVM computes all matches up-front and deals with rule inheritance by applying top rules first, and then the matching process continues with the descendants of the matched rules, recursively in *horizontal slices* corresponding to levels of children rules. When a descendant rule matches the same object, the match of the parent rule is removed. Hence, rule inheritance is used as an optimization technique where child rules are matched only if their parent rules are matched, effectively representing a RETE network [46]. YAMTL follows a similar approach but takes a *vertical slice* search strategy centered on an object by finding the most concrete rule that can be applied. This allows YAMTL to start the matching process from a single match using only those rules that are relevant. On the other hand, in EMFTVM matches corresponding to abstract rules are stored but not applied, whereas YAMTL discards matches corresponding to abstract rules.

## 5.2 Execution Semantics

The VIATRA solution, which is the fastest solution in the benchmark up to now, uses EMF-IncQuery patterns to activate the actions that perform the transformation, which are defined in plain Xtend. Hence, the execution semantics is that of an Xtend program. The execution semantics in EMFTVM is more involved as requires dealing with inheritance of feature assignment expressions, which are implicitly *overriden* in child rules and *inherited* otherwise. YAMTL uses an *inherits* strategy by default, evaluating all inherited actions for a particular output element in the order in which they were declared, with the option of overriding them manually. In YAMTL, an output element can refer to an input element defining an update. In ATL, this is achieved by declaring MTs in refining mode.

The operation `fetch` in YAMTL resembles that of `resolveTemp` in ATL although their semantics differ considerably. When used in an action expression (binding in ATL) of an output element, both operations resolve target object references, which may have been created in other rules, from input pattern matches in order to initialize references in the output element. However, whereas `fetch` implements a *check-enforce semantics* enabling access to features of target objects from the context of the calling output element, object references obtained from `resolveTemp` cannot be accessed, as they may not have been initialized yet. That is, `resolveTemp` can only be used after the matching phase is completed and, hence, access to traceability links is possible. This decouples object creation from binding initialization. While, in the current version, `fetch` is also enabled after the initial matching phase, it can be used as a control structure for executing of rules as it triggers a transformation step if it has not been performed yet. This combines both object creation and binding initialization in one single step. To dissipate concerns about decidability, note that the existence of recursive cycles is forbidden thanks to the use of the internal *matchPool*, which enforces that matches are unique.

| Feature | | ATL | | VIATRA | YAMTL |
|---|---|---|---|---|---|
| | | ATL2010 (EMFVM) | EMFTVM | | |
| Semantics | | on-demand, mapping, reactive, (source and target) incr. | as in EMFVM, rewriting with SimpleGT | on-demand, mapping/ rewriting, reactive, (source) incr. | on-demand, mapping, rewriting, (source) incr. |
| Query pattern | language | OCL (eager) | OCL (lazy) | IncQuery | Xtend |
| | multiobject | select-collect, forAll | select-collect, forAll | native | filter-map, forall |
| | recursion | helpers, rules | helpers, rules | helpers | helpers, rules |
| | negative | OCL (negation) | OCL (negation) | native | Xtend (negation) |
| Decl. rules | elements | n-m | n-m | n-m | n-m |
| | traceability | implicit/explicit | implicit/explicit | implicit/explicit | implicit/explicit[1] |
| | control | (unique) lazy rules | (unique) lazy rules | control schemas | priorities, (unique) lazy rules, fetch |
| Reuse | trafo | libraries, HOT, module import | libraries, HOT, module import | HOT | import/extension |
| | rule | single inheritance, with binding overriding | multiple inheritance: filter and bindings (overrides) | "inheritance" of patterns via pattern reuse | multiple inheritance: filter and actions (inherits/ overrides) |
| | values | attributes | attributes | genericity | attributes |

[1]Developed ad-hoc for the traceability metamodel of the benchmark.

**Figure 1: Analysis of language features.**

# 6 PERFORMANCE ANALYSIS

In this section, the performance of the YAMTL MT engine is compared against the MT approaches discussed in the previous section with the batch MT component of the VIATRA CPS benchmark [36]. The software artefacts used in the evaluation and the results obtained are publicly available [13]. The experiments were run on a MacBookPro11,5 Core i7 2.5 GHz, with four cores and 16 GB of RAM. For the experiments the following software was used: ATL/EMFTVM (4.0.0); ATL SDK (4.0.0); CPS metamodels (0.1.0); Eclipse (4.7.3); EMF SDK (2.13.0); JRE (build 1.8.0_72-b15); VIATRA SDK (1.7.2); and Xtend SDK (2.13.0).

In the following subsections, we describe the part of the VIATRA CPS benchmark selected for the experiments, the methodology followed, and conclusions drawn from the experiments.

## 6.1 VIATRA CPS Benchmark Adaptation

A new benchmark harness was developed considering the best practices recommended by the VIATRA team [20] among others. This allowed us both to fine-tune measurements with VLMs and to crosscheck results for the sake of both consistency and fairness.

For the study, existing batch MTs were extracted from the benchmark framework as independent Java projects. Classes implementing transformations were kept intact in the new projects, including their namespaces, so that errors were not introduced due to lack of expertise. From the benchmark, the following transformations were considered: *EMF-IncQuery batch transformation* (EIQ) [34], which was the fastest as a batch MT up to now ; and the optimized version of the *Xtend batch MT* (Xtend) [39], which is defined in plain Xtend using a cache to store traceability information – a feature present in YAMTL, ATL2010 and EMFTVM and therefore of relevance. Incremental variants were not considered as they show worse performance than EIQ when used for batch MT.

The YAMTL solution [12] comprises four phases: first, an attribute helper is used to cache transitions with an action `waitForSignal` indexing them by its signal; second, `HostInstances` and `ApplicationInstances` are transformed using matched rules, `StateMachines` are transformed using a lazy rule, thus creating a deep copy of the target object for each rule application, and `States` and `Transitions` are transformed using unique lazy rules within the context created by the lazy rule applied to their containing `StateMachine`; third, a transient matched rule with low priority is used to compute triggers, once all transitions have been transformed; fourth, the traceability model is extracted from YAMTL's internal model of traceability links (*eventPool*), which comes with the tool out of the box. This last construction process has been considered part of the transformation whereas the actual storage process that flushes contents to a physical file through the EMF persistence API is considered to be external. The YAMTL solution passes the CPS benchmark sanity tests, with some minor adaptations as reported in [12].

In addition, two new MTs were defined in ATL, one with matched rules (decl.) [5] and one with lazy rules (imp.) [6], and executed with ATL2010 [7, 8] and with EMFTVM [10]. These MTs map each CPS element to its DEP counterpart using either implicit object resolution, when using matched rules, or explicit rule invocation, when using lazy rules. An attribute helper is used to cache all transitions with action `waitForSignal` at the start, and these are used for setting `triggers` in `BehaviorTransitions` in the DEP model when CPS transitions are processed. Hence, there is a rule match for each element in the CPS model. These solutions are, however, not fully correct: a unique target behaviour is created per source state machine; triggers are always created for the first instance of the application type disregarding dynamic host instance allocation; and the traceability model is not extracted. Moreover, some anomalies were detected when inspecting their results: triggers are not set when using ATL2010, and the feature `to` of `BehaviorTransition` is not set consistently in the MT with lazy rules with EMFTVM. Bearing in mind these caveats, these transformations are still useful for indicative purposes.

## 6.2 Scenarios and Methodology

Among the scenarios provided as part of the VIATRA CPS benchmark [38], we discuss our analysis with *client-server*. Results with the other scenarios for the YAMTL and EIQ solutions are available online [13]. We have used the CPS model generator [37] to randomly obtain models whose size depends on a logarithmic scale factor,[4] obtaining the sizes reported in Figure 2 in terms of both number of nodes (objects) and number of edges (references). The CPS generator could not generate models twice as big as the last model used in the experiment, giving an out of memory error (oom), probably due to the EMF persistence API and not to the generation process itself. However, the biggest models considered consist of millions of nodes and edges and can be classified as VLMs.

For each transformation (and tool), the same Java process was used to run all of the experiments. For each model size, twelve experiments were performed, measuring performance time (in *ms*) in four phases: model load, engine initialization, transformation and

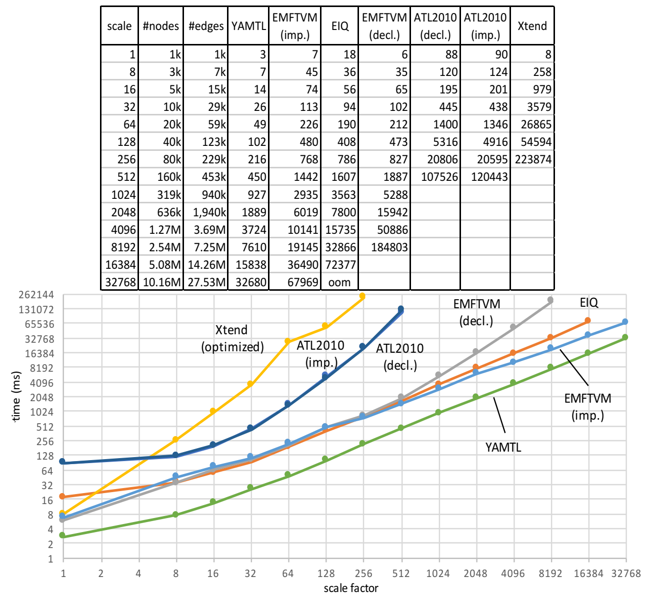
[4]While the live VIATRA CPS benchmark generates such models for each experiment, our experiments rely on fixed source models, used to evaluate all of the solutions. Although this does not affect the validity of the results, the results in the live benchmark may vary and the author is currently collaborating with the maintainers of the VIATRA CPS benchmark to integrate the YAMTL solution in the live benchmark [9].

model storage. The engine initialization phase involves the instantiation of a fresh engine instance, avoiding interference between experiments as caches are not reused. Only engine initialization and transformation times have been considered in the quantitative analysis. Transformations were executed with the VM arguments `-Xmx12288m -XX:+UseConcMarkSweepGC`. For each model size, the list of results obtained for the twelve experiments were ordered and the minimum and maximum values were deleted. Then the *median* of the remaining list was considered for the analysis of the transformation for the corresponding input model, as shown in Figure 2. For each solution, an initial iteration for the smallest model size was executed to warm up the JVM, and excluded from the analysis.

*Threats to Validity.* Regarding the DEP models produced by the different tools, EMFTVM, Xtend and YAMTL produced syntactically identical models. However, the ATL transformation is not fully correct as explained in section 6.1. On the other hand, EIQ produces highly similar models but for `trigger` references of `BehaviorTransition` objects, which appear shuffled even though they are defined as *ordered* in the CPS deployment metamodel.

### 6.3 Evaluation of Experimental Results

The results obtained, displayed in Figure 2 using logarithmic scales both for time (ms.) and for sizes, are reasonably consistent with those available in the VIATRA CPS benchmark [35] for the Xtend (optimized) and EIQ solutions. According to our findings, YAMTL is the fastest solution – in all of the experiments performed [13]. In the client-server scenario, both YAMTL and ATL (imp., i.e. with lazy rules) on EMFTVM are able to process transformations involving 10.16M objects and 27.53M references (client-server), while VIATRA gives an out of memory error (oom). YAMTL did so with a median of 32.7$s$. (including transformation initialization and execution time, and excluding EMF persistence times) for the largest models. Note, however, that the experiments are biased towards ATL as considered in threats to validity. In the experiments, EMFTVM showed a substantial improvement when using lazy rules, i.e. with non-declarative MT, which was not perceivable in ATL2010 (EMFVM). On the other hand, it can be observed that the abstractions that YAMTL provides to Xtend for declarative MT use computational resources efficiently, vastly outperforming the Xtend solution.



| scale | #nodes | #edges | YAMTL | EMFTVM (imp.) | EIQ | EMFTVM (decl.) | ATL2010 (decl.) | ATL2010 (imp.) | Xtend |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1k | 1k | 3 | 7 | 18 | 6 | 88 | 90 | 8 |
| 8 | 3k | 7k | 7 | 45 | 36 | 35 | 120 | 124 | 258 |
| 16 | 5k | 15k | 14 | 74 | 56 | 65 | 195 | 201 | 979 |
| 32 | 10k | 29k | 26 | 113 | 94 | 102 | 445 | 438 | 3579 |
| 64 | 20k | 59k | 49 | 226 | 190 | 212 | 1400 | 1346 | 26865 |
| 128 | 40k | 123k | 102 | 480 | 408 | 473 | 5316 | 4916 | 54594 |
| 256 | 80k | 229k | 216 | 768 | 786 | 827 | 20806 | 20595 | 223874 |
| 512 | 160k | 453k | 450 | 1442 | 1607 | 1887 | 107526 | 120443 | |
| 1024 | 319k | 940k | 927 | 2935 | 3563 | 5288 | | | |
| 2048 | 636k | 1,940k | 1889 | 6019 | 7800 | 15942 | | | |
| 4096 | 1.27M | 3.69M | 3724 | 10141 | 15735 | 50886 | | | |
| 8192 | 2.54M | 7.25M | 7610 | 19145 | 32866 | 184803 | | | |
| 16384 | 5.08M | 14.26M | 15838 | 36490 | 72377 | | | | |
| 32768 | 10.16M | 27.53M | 32680 | 67969 | oom | | | | |



**Figure 2: Median performance times in *ms* (client-server).**

## 7 CONCLUSION AND FUTURE WORK

In this paper, we have introduced YAMTL, an internal DSL of Xtend that is both as *expressive* as ATL for batch MTs and more *efficient* than current MT engines, which are used *in the large* in the project MONDO, in the batch MT component of the VIATRA CPS benchmark. Two important lessons are sustained by our experimental results: neither *(1)* virtual machines that work with low level MT instructions nor *(2)* incremental queries may be the most efficient approaches for batch MTs. YAMTL tackles the efficient MT of VLMs, increasing the amount of work that can be performed in a single-threaded environment, thus *scaling up* processors without employing additional physical resources.

On the one hand, regarding *expressiveness*, the semantics of the operation `fetch` has similarities with the *check/enforce* semantics at the core of *synchronization* languages, like QVT-Relations [47] or MOFLON [1], and extending YAMTL with support for incremental MT is feasible. On the other hand, while performing the experimental evaluation of the tools, we experienced the *XMI serialization problem* in EMF [26], which yielded long latency times and eventual out of memory errors. This is a clear Achilles' heel for the application of YAMTL (and other EMF-native MT tools) to VLMs, which could be addressed by using external NoSQL stores, as done in NeoEMF [16]. The performance analysis of section 6 focussed on one single case study and we aim at researching YAMTL's characteristics with further case studies and benchmarks.

Advantages of internal DSLs for MT w.r.t. external DSLs were discussed in [23, 28] and several of them have been proposed – e.g. in Scala, SIGMA [29], and in $C^\sharp$, NTL [21–23]. SIGMA is EMF native and its MT rules are defined as Scala methods, which are internally processed using reflection, thereby achieving a more concise syntax than that of YAMTL. Its performance was analysed in [27], where it was found to be as efficient as raw Java. NTL is not EMF-native and provides efficient support for incremental and bidirectional MT. A comprehensive comparison with these MT languages, which are research prototypes under development, and others is out of scope in this work and could be done collaboratively, possibly in the context of the Tool Transformation Contest.


## ACKNOWLEDGMENTS

The author would like to thank: Dr. Frédéric Jouault for insightful discussions on the use of advanced features of ATL; Théo Le Calvar, Nils Weidmann and Dr. Stefan Sauer for their improvement suggestions on the software artefacts accompanying this paper; and the anonymous reviewers for their constructive feedback.

# REFERENCES


[1] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. 2006. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *ECMDA-FA*, Arend Rensink and Jos Warmer (Eds.), Vol. 4066. LNCS, 361–375.

[2] Amine Benelallam, Abel Gómez, Massimo Tisi, and Jordi Cabot. 2015. Distributed Model-to-model Transformation with ATL on MapReduce. In *SLE*. ACM, 37–48.

[3] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. 2015. Viatra 3: A Reactive Model Transformation Platform. In *ICMT*, Vol. 9152. LNCS, 101–110.

[4] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. 2010. Incremental Evaluation of Model Queries over EMF Models. In *MoDELS*, Vol. 6394. LNCS, 76–90.

[5] Artur Boronat. 2018. ATL Batch M2M Transformation for the VIATRA CPS Benchmark. https://github.com/yamtl/viatra-cps-batch-benchmark/blob/master/m2m.batch.data/atlFiles/Cps2Dep.atl.

[6] Artur Boronat. 2018. ATL Batch M2M Transformation for the VIATRA CPS Benchmark (variant with lazy rules). https://github.com/yamtl/viatra-cps-batch-benchmark/blob/master/m2m.batch.data/atlFiles/Cps2Dep_lazy.atl.

[7] Artur Boronat. 2018. ATL2010 (EMFVM) Runner for Batch M2M Transformation for the VIATRA CPS Benchmark. https://github.com/yamtl/viatra-cps-batch-benchmark/tree/master/m2m.batch.cps2dep.atl2010.

[8] Artur Boronat. 2018. ATL2010 (EMFVM) Runner for Batch M2M Transformation for the VIATRA CPS Benchmark (lazy variant). https://github.com/yamtl/viatra-cps-batch-benchmark/tree/master/m2m.batch.cps2dep.atl2010_lazy.

[9] Artur Boronat. 2018. Discussion on the integration of the YAMTL solution into the VIATRA CPS Benchmark. https://github.com/viatra/viatra-cps-benchmark/issues/23.

[10] Artur Boronat. 2018. EMFTVM Runner for Batch M2M Transformation for the VIATRA CPS Benchmark. https://github.com/yamtl/viatra-cps-batch-benchmark/tree/master/m2m.batch.cps2dep.emftvm.plugin.

[11] Artur Boronat. 2018. Repository with Examples of YAMTL Model Transformations. https://github.com/yamtl/examples.

[12] Artur Boronat. 2018. YAMTL Batch M2M Transformation for the VIATRA CPS Benchmark. https://github.com/yamtl/viatra-cps-batch-benchmark/tree/master/m2m.batch.cps2dep.yamtl#benchmark-cps2dep-variant-yamtl-batch.

[13] Artur Boronat. 2018. YAMTL Evaluation Repository with the batch component of the VIATRA CPS Benchmark. https://github.com/yamtl/viatra-cps-batch-benchmark.

[14] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. 2007. *All About Maude*. LNCS 4350.

[15] Gwendal Daniel, Frédéric Jouault, Gerson Sunyé, and Jordi Cabot. 2017. Gremlin-ATL: A Scalable Model Transformation Framework. In *ASE*. IEEE Computer Society, 462–472.

[16] Gwendal Daniel, Gerson Sunyé, Amine Benelallam, Massimo Tisi, Yoann Vernageau, Abel Gómez, and Jordi Cabot. 2017. NeoEMF: A multi-database model persistence framework for very large models. *Sci. Comput. Program.* 149 (2017), 9–14. https://doi.org/10.1016/j.scico.2017.08.002

[17] Javier Espinazo-Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. 2013. A repository for scalable model management. *Software and System Modeling* 14, 1 (2013), 219–239. https://doi.org/10.1007/s10270-013-0326-8

[18] The Eclipse Foundation. 2018. Xtend (official web page). http://www.eclipse.org/xtend/.

[19] EU FP7. 2018. The MONDO Project: Scalable Modelling and Model Management on the Cloud. http://www.mondo-project.org/.

[20] Dénes Harmath and István Ráth. 2016. VIATRA/Query/FAQ: Performance optimization guidelines. https://wiki.eclipse.org/VIATRA/Query/FAQ#Performance_optimization_guidelines.

[21] Georg Hinkel. 2015. Change Propagation in an Internal Model Transformation Language. In *ICMT*, Vol. 9152. LNCS, 3–17.

[22] Georg Hinkel and Erik Burger. 2017. Change propagation and bidirectionality in internal transformation DSLs. *Softw Syst Model* (2017).

[23] Georg Hinkel, Thomas Goldschmidt, Erik Burger, and Ralf Reussner. 2017. Using internal domain-specific languages to inherit tool support and modularity for model transformations. *Soft. Syst. Model.* (2017).

[24] Edgar Jakumeit, Sebastian Buchwald, Dennis Wagelaar, Li Dan, Ábel Hegedüs, Markus Herrmannsdörfer, Tassilo Horn, Elina Kalnina, Christian Krause, Kevin Lano, Markus Lepper, Arend Rensink, Louis Rose, Sebastian Wätzoldt, and Steffen Mazanek. 2014. A survey and comparison of transformation tools based on the transformation tool contest. *Science of Computer Programming* 85 (2014), 41 – 99. https://doi.org/10.1016/j.scico.2013.10.009 Special issue on Experimental Software Engineering in the Cloud(ESEiC).

[25] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. 2008. ATL: A model transformation tool. *Sci. Comput. Program.* 72, 1-2 (2008), 31–39.

[26] Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos Matragkas, Richard F. Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan de Lara, István Ráth, Dániel Varró, Massimo Tisi, and Jordi Cabot. 2013. A research roadmap towards achieving scalability in model driven engineering. In *BigMDE@STAF*. 2.

[27] Filip Krikava. 2015. Solving the TTC'15 Train Benchmark Case Study with SIGMA. In *Transformation Tool Contest@STAF 2015 (CEUR Workshop Proceedings)*, Vol. 1524. 167–175.

[28] Filip Krikava, Philippe Collet, and Robert B. France. 2014. Manipulating models using internal domain-specific languages. In *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*. 1612–1614.

[29] Filip Krikava, Philippe Collet, and Robert B. France. 2014. SIGMA: Scala Internal Domain-Specific Languages for Model Manipulations. In *MoDELS (LNCS)*, Vol. 8767. Springer, 569–585.

[30] Salvador Martínez Perez, Massimo Tisi, and Rémi Douence. 2017. Reactive model transformation with ATL. *Sci. Comput. Program.* 136 (2017), 1–16. https://doi.org/10.1016/j.scico.2016.08.006

[31] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework 2.0* (2nd ed.). Addison-Wesley Professional.

[32] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. 2014. IncQuery-D: A Distributed Incremental Model Query Framework in the Cloud. In *MoDELS*, Vol. 8767. LNCS, 653–669.

[33] Antero Taivalsaari. 1996. On the Notion of Inheritance. *ACM Comput. Surv.* 28, 3 (Sept. 1996), 438–479. https://doi.org/10.1145/243439.243441

[34] VIATRA Team. 2016. Simple Xtend and Query M2M Transformation. https://github.com/viatra/viatra-docs/blob/master/cps/Simple-Xtend-and-Query-M2M-transformation.adoc.

[35] VIATRA Team. 2016. VIATRA CPS Benchmark (batch use case performance results). https://github.com/viatra/viatra-cps-benchmark/wiki/Performance-evaluation#runtime.

[36] VIATRA Team. 2016. VIATRA CPS Benchmark (CPS to Deployment Transformation). https://github.com/viatra/viatra-docs/blob/master/cps/CPS-to-Deployment-Transformation.adoc.

[37] VIATRA Team. 2016. VIATRA CPS Benchmark (model generator). https://github.com/viatra/viatra-docs/blob/master/cps/Model-Generator.adoc.

[38] VIATRA Team. 2016. VIATRA CPS Benchmark (scenario specification). https://github.com/viatra/viatra-cps-benchmark/wiki/Benchmark-specification#cases.

[39] VIATRA Team. 2016. Xtend Optimized Batch M2M Transformation. https://github.com/viatra/viatra-docs/blob/master/cps/Simple-and-optimized-Xtend-batch-M2M-transformation.adoc/#optimized-batch-m2m-transformation.

[40] Massimo Tisi, Salvador Martínez Perez, and Hassene Choura. 2013. Parallel Execution of ATL Transformation Rules. In *MODELS (LNCS)*, Vol. 8107. Springer, 656–672.

[41] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. 2015. EMF-IncQuery: An integrated development environment for live model queries. *Sci. Comput. Program.* 98 (2015), 80–99. https://doi.org/10.1016/j.scico.2014.01.004

[42] Zoltán Ujhelyi, Gábor Sz oke, Ákos Horváth, Norbert István Csiszár, László Vidács, Dániel Varró, and Rudolf Ferenc. 2015. Performance comparison of query-based techniques for anti-pattern detection. *Information and Software Technology* 65 (2015), 147 – 165. https://doi.org/10.1016/j.infsof.2015.01.003

[43] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. 2016. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software & Systems Modeling* 15, 3 (Jul 2016), 609–629. https://doi.org/10.1007/s10270-016-0530-4

[44] Dániel Varró and András Pataricza. 2002. Metamodeling Mathematics: A Precise and Visual Framework for Describing Semantics Domains of UML Models. In *UML*, Vol. 2460. LNCS, Berlin, Heidelberg, 18–33.

[45] Gergely Varró, Frederik Deckwerth, Martin Wieber, and Andy Schürr. 2015. An algorithm for generating model-sensitive search plans for pattern matching on EMF models. *Software & Systems Modeling* 14, 2 (May 2015), 597–621. https://doi.org/10.1007/s10270-013-0372-2

[46] Dennis Wagelaar, Massimo Tisi, Jordi Cabot, and Frédéric Jouault. 2011. Towards a General Composition Semantics for Rule-Based Model Transformation. In *MoDELS*, Vol. 6981. LNCS, 623–637.

[47] Edward D. Willink. 2017. The Micromapping Model of Computation; The Foundation for Optimized Execution of Eclipse QVTc/QVTr/UMLX. In *ICMT*, Vol. 10374. LNCS, 51–65.

[48] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Dimitris S. Kolovos, Richard F. Paige, Marius Lauder, Andy Schürr, and Dennis Wagelaar. 2012. Surveying Rule Inheritance in Model-to-Model Transformation Languages. *Journal of Object Technology* 11, 2 (2012), 3: 1–46. https://doi.org/10.5381/jot.2012.11.2.a3