A constraint language for algebraic term based on rewriting theory

François Prugniel, Pierre-Etienne Moreau, Horatiu Cirstea

A constraint language for algebraic term based on rewriting theory

François Prugniel, Pierre-Étienne Moreau, and Horatiu Cirstea

LORIA, Vandoeuvre-les-Nancy, France

Abstract

A key feature of Model Driven Engineering is the ability to define meta-models, but also constraints that have to be satisfied by their instances. Constraints are expressed in OCL (Object Constraint Language), which became a standard. OCL Constraints offer the possibility to capture properties which cannot be easily encoded in the meta-model.

In the programming language community, instead of meta-models, we generally use grammars or algebraic signatures to define the syntax of the programs or abstract syntax trees (AST) we consider. But unfortunately, even when considering rich formalisms such as many-sorted signatures with subtyping or dependent types, they are not expressive enough to encode in a simple way some subsets of terms we want to consider. For instance *arithmetic expressions which have at most two levels of plus operator*, this includes a, $a+b$, $(a+b)+(b+c)$, but not $a+(b+(c+d))$ for instance. There is a need for a constraint language dedicated to tree based data structures such as terms and AST.

In this paper we present both a language to express constraints on trees, and a compilation scheme that shows how to translate constraints into an executable formalism based on term rewriting and strategies.

1 Introduction

Since a few years, a huge need for model, model transformation and constraint checking appear, especially on critical systems domain. With his OCL [1], the Object Management Group developed a powerful language to constrain UML diagrams but not really adapted for algebraic signatures. Moreover, existing OCL cores are static for the most of them. They check constraints on a static instance of models and those which check constraints at execution are unusable on real conditions because execution time explodes with the checking of invariants at each changing state.

Starting from OCL experiences, we propose a new constraint language designed for algebraic signatures, based on rewriting pattern-matching and strategies, inspired by OCL and XPath [5]. Our aim is to provide a way to constrain

a signature and the key to check the constraints. For this, we developed a constraint language which allows structural properties checking and dynamic typing. Contrary to OCL, we choose to let the user choose when he wants to check constraints by simply calling methods. This way, he can control increased execution time due to checking.

This kind of language has several applications, particularly for compiler or model transformation. By specifying constraints at each phase of compilation or model transformation, we can check the good sequence of process. It is a major asset for debugging and a great step to obtain trustworthy compiler. Moreover, we think we can develop our theory for the models by using term-graph [3].

In this paper, we will give our motivations and the essential notations in a first part. In a second part, we will detail our grammar, with explanation about our choices and semantics. In a third part, we will show you some translation schemes through an example.

2 Background

Motivation

Our goal is to develop a constraint mechanism for algebraic term. Indeed, the model community has OCL but there are no constraint mechanism for algebraic signature. To allow verification of structural properties and dynamic typing for algebraic signature, we are working on a constraint language based on the rewriting theory which permits to specify constraints during the conception and to verify them at execution. A strength of OCL is its simple syntax which permit at each actor of a project to write and to understand constraints. As an incomprehensible language to non-expert users will never be used, we want to create a language which is expressive and easy enough to be understandable by each actor of a project. We take the good ideas from OCL that are usable with algebraic signature, like the context or the navigation. We add the predicates from XPath and some custom mechanisms to obtain a new constraint language for algebraic signature.

Notations

We introduce here the semantics of our notations in this paper.

The *names* in italic and red are non terminal, the names in blue are terminal. The '[]' notation means 'optionally', the '()*' notation means '0 or more' and the '()+' notation means '1 or more'. The TopDown strategy, which is uses to check a constraint, is a way to visit the term where we check the logical expression. It is a depth-first search. That means we traverse the tree from the root and explores as far as possible along each branch before backtracking.

We compile the constraints from our language to Tom [4], an extension of Java which adds pattern-matching and strategies for algebraic signatures. In our translation patterns, the keyword `Strategy` designates the strategy which

will check the constraint and it is follows by its name to make a method. The `visit` keyword designates node's sort where we will check the constraint. The `_` (resp. `_*`) corresponds to a generic variable (resp. list of variables). The `t[a=b]` means that we look for slot `a` in term `t` and `b` can be a variable name to take the contains of the slot or a term. The `a << b` means that `b` filters `a`. To make the difference between terms and other things, we put a ‘ before them.

Example grammar

```
1 Forest = forest(Tree*)
2 Tree   = tree(root : Node)
3 Node   = node(value : Integer, l : Node, r : Node)
4        | leaf(value : Integer)
```

Figure 1: Example grammar

In this paper, we consider a grammar (*Fig. 1*) which represents binary tree and a collection of these trees. We assume that each node on trees has an integer on label and for each node, the value of the left son is greater than the value of the right son. The root of a tree also has the highest value.

3 Partial concrete syntax

In this part, we will present the important parts of our concrete syntax. We chose not to show you our abstract syntax to focus on our effort to make a language simple but powerful for the user.

Basic structure

Constraints	::=	context *Context* : *ConstraintExpressions*
ConstraintExpressions	::=	(*DefExpr*)* (*ConstraintExpr*)+
ConstraintExpr	::=	*ConstraintName* : (*LetExpr*)* *LogicalExpr* ;

Figure 2: Basis of the concrete syntax

Fundamentally, a constraint is composed of a context and a logical expression. To factorize constraints, we allow the possibility to write several logical expressions with a same context (*Fig. 2*). Each constraint has a *ConstraintName* which is a string starting by a letter. That point permits to identify them for calling them latter. We also add two writing short-cut mechanisms : *DefExpr* and *LetExpr*. They permit to put recurring expressions on variables, valid for each constraint with the same context in the first case and only valid for the *LogicalExpr* associated for the second.

All about the context

Context ::= *Type* [:: *OperatorName* ([*Predicat*])* (. *SlotName* ([*Predicat*])*)*]

Figure 3: Concrete syntax for Context

As in OCL, each constraint need a context (*Fig. 3*). It represents the starting point in the term to check the logical expression. That means during checking we traverse the term with a TopDown strategy and at each node which matches with context, we check the constraint on the sub-term with this node as root. A context is at least a sort from the grammar to constrain. Next, we can refine context with an operator name and go down recursively with a slot name. The user must check the coherence of his context. For this, we added a predicate mechanism, which comes from XPath. For the moment, we only allow the possibility to precise the name of the constructor which interests the user for a given slot name. That means we ignore the other cases. For example, if we want to select as context the value of the left of a tree with a depth of 1, we have to write : `Tree::tree[root=node].root[l=leaf].l.value`. Here, we are looking for a constructor `tree` of sort `Tree` where the root slot is a `node` with a `leaf` as left son. We are thinking about an extension of the expressiveness of predicates. If the context is wrong, the constraint won't be checked, so we plan to warn users about contexts which never match during the checking process.

Build a logical expression

LogicalExpr ::= *LogicalExpr LogicalOperator LogicalExpr*
| *IfExpr* | ! *LogicalExpr*
| *Boolean* | *ComparativeExpr*

Figure 4: Concrete syntax for logical expression

Checking a constraint is the same as evaluating a logical expression on a sub-term. These expressions (*Fig. 4*) can be several things, like two logical expressions linked by a logical operator (equality, non equality, conjunction, disjunction or implication). We can also negate a logical expression or use boolean. We also have a conditional expression (if then else) which must ensure a boolean result, so the else part is obligatory. An other possibility is the comparison of two *Expr*. These expressions can be constants (double, string, etc.), calculation between two *Expr*, term built on grammar, path to reach a specific node or list of nodes from the current one in order to use them directly or to call a method from the API on it. In the case of a path, the keyword `self` refers to the current context. For the case of term built on grammar, we have two ways to write them. The first, with parenthesis, implies to write the whole term and the second uses the notation with brackets presented before.

We have two kinds of method which can be called on *Expr* : the *Operation* and the *IteratorOperation*. This distinction comes from OCL. The *Operation* rule represents usual methods from the most of programming languages, which take some parameters to return a result. The second kind, *IteratorOperation* rule, is specific to the lists and we have to determine one or more iterators to go through them. Two examples of *IteratorOperation* are `exists` and `forall`, which are really useful in constraints.

If for example we want to make a logical expression which means that if, in the context of a `Node`, the current node is a `node`, its value is greater than the ones of its children and in the other case, its value is greater than 0, we will have this logical expression :

```
if self=‘node[] then self.getValue()>l.getValue() &&
self.getValue()>r.getValue() else self.getValue()>0 fi
```

Navigate through the tree

Navigation is a powerful notion in OCL which allows to reach any classifier linked to the context's classifier. We adapted this concept for algebraic signature. We use a similar concept to go down in nodes from a context. The '.' is used to reach a son of the current node by its slot name. It also permits to invoke method on a node. The '->' is used to invoke method on a list. This notation is not essential but we choose to use it to make the constraints reading more comfortable. The third notation, ':', was added because in algebraic signature, each sort can have several constructors with slots. Using slot name only to navigate on tree is insufficient, that is why we added the possibility to precise that we are interested by only one kind of constructor on a slot. To use this notation, we have to add ':' followed by a constructor name after a slot name. That means that we just ignore the other cases where the logical expression will not be evaluated. The last notation, '...', allows to reach a node under an other one by skipping some levels between them. We call it in-depth navigation. This concept will be translates on a TopDown strategy and we consider that we treat each result alone while strategy takes care of the application at each concerned node. Regarding this last notation, its semantics is not totally fixed. We are thinking about cases where we have to prohibit it and about the interest to precise how much levels we can skip.

To illustrate these notations, let use imagine we are on the context of `Tree::tree`. We want to build a path to reach each leaf under the current node (`tree`) if the root is a node. We will have this : `self.root:node...leaf`.

4 Some translation pattern by examples

Since our semantic is not completely fixed, we don't have all translation patterns yet from our language to rewriting theory and strategies. In this part, we will present a few constraints examples with their translation. The general idea is that we don't want to know if a constraint is true but where it is false, without

forgetting to verify if a constraint was checked at least one time. The second point is not visible in our example but this will be a feature of our checker.

Natural constraints

We chose 3 constraints on our example grammar (*Fig. 1*) to expose some possibilities of our language and some translation patterns.

- For a node of sort Node, the value of the left son is higher than the one of the right son and if a son of a node is a leaf, both are leaf. (1)
- We want that all trees have a depth greater than or equals 1. So, for all trees in Forest, the root is a node. (2)
- For a tree of sort Tree, the value of the root is the highest. We can obviously ignore the case where the root is a leaf. (3)

Constraints and translations

In translations each strategy is executed on a TopDown way and has the name of the linked constraint. We do not want to rewrite our term, we want to use the power of pattern-matching and strategies to detect mistakes with constraints to report them to users. So, for each place where the constraint is broken, we call the method `error()`.

```
1  Context Node::node
2    leaf : l.getValue() > r.getValue() &&
3           if l = `leaf[]` then r = `leaf[] else r = `node[] fi;
4
5  Strategy leaf
6    visit Node {
7      node[l=var_l, r=var_r] -> {
8           if (!(var_l.getValue() > var_r.getValue())) { error();}
9        match {
10         leaf[] << var_l && !(leaf[] << var_r) -> { error(); }
11         !(leaf[] << var_l) && !(node[] << var_r) -> { error(); }
12       }
13     }
14   }
```

Figure 5: Constraint (1) and its translation

The first constraint (*Fig. 5*) highlights three points. In the `visit` part, we have to know all the needed variables to write the rules. Inside the rule of the strategy, we can see two parts, one for each conjunction of the logical expression. As we want that the two conditions of the constraint are true, we can check them alone. Finally, we can see the need of an API with the `getValue()`. Indeed, `l.value` and `r.value` are node in our tree and we can't compare them, so we have to extract the numeric values to do it. To check if the second part of the constraint is broken, we negate the two cases. On the first hand, if `var_l` filters `leaf` constructor but `var_r` does not, we break the first case. On the second

hand, if `var_l` does not filter `leaf` constructor and `var_r` does not filter node constructor, we break the second case.

```
Context Forest::forest
  forall_depth_1 : self->forall(t : Tree | t == `tree[root=node[]]`);

Strategy forall_depth_1
  visit Forest {
    forest(_*, t, _*) -> {
      match {
        !(tree[root=node[]] << t) -> { error; }
      }
    }
  }
```

Figure 6: Constraint (2) and its translation

In the second constraint (*Fig. 6*), we want to ensure that all trees in forest have a depth of at least 1, that means the root of each tree is a node. All elements of the list are concerned, so we have to check all of them and for each tree in the forest, check if the root is a node or not. For this, we use pattern-matching which explores all trees in forest. If we put the `exists` method instead of the `forall`, we have to check if we have a list which does not has an element according to the constraint. So, in this case, we don't check all elements of the list but the list itself.

```
Context Tree::tree[root=node]
  highest_value : self.getValue() > self...leaf.getValue()
    && self.getValue() > self...node.getValue()

Strategy highest_value
  visit Tree {
    tree[root=self@node[]] -> {
      TopDown(highest_value_1(self)).visit(self);
      TopDown(highest_value_2(self)).visit(self);
    }
  }
Strategy highest_value_1(Node node)
  visit Node {
    leaf@leaf[] -> {
      if (node.getValue() > leaf.getValue()) { error(); }
    }
  }
Strategy highest_value_2(Node node)
  visit Node {
    node2@node[] -> {
      if (node.getValue() > node2.getValue()) { error(); }
    }
  }
```

Figure 7: Constraint (3) and its translation

In this third constraint (*Fig. 7*), we want to ensure that the value of the root is the highest. For this, we have to present the alias notation from Tom in our translation : `node2@node[]` means we make a variable `node2` which contains the `node[]` term. The use of a predicate on the context appears on the rule of

the strategy highest_value : we don't want all the root but only the ones which are `node`. We also use the in-depth navigation twice. Each one is translated on a strategy, calls on a TopDown way on `self`. We also have to give them the root node, i.e `self`, to do the comparison. We can point out that our language is clearer than the translated version.

5 Conclusion and Future work

We presented a language which allows to write constraints on algebraic signatures. We picked up ideas from OCL and XPath to propose a language easy to come to grips with, even for users who don't know rewriting. Our current aim is to finish formalization of our grammar, its semantics and translation patterns. When this is done, we will include our language in Tom [4]. Tom is a way to make programs based on algebraic signatures, especially compiler. With this extension, we plan to constrain the Tom compiler itself.

As said in introduction, we think we can use works on term-graph [3] to adapt our theory to graphs and models. This kind of extension will allow an alternative to OCL, with several similarities but a lower complexity and less ambiguities [2]. Furthermore, it will be applicable to model transformation, in order to propose a way to check if a transformation is correct. This should be really useful for critical systems domain which makes a lot of model transformations to prove correctness of their applications.

References

[1] Object constraint language omg available specification version 2.0, 2006.

[2] D. Akehurst, P. Linington, and O. Patrascoiu. Ocl 2.0 - implementing the standard for multiple metamodels. *Proceedings of the UML'03 workshop*, Electronic Notes in Theoretical Computer Science:19, November 2003.

[3] Emilie Balland and Paul Brauner. Term-graph rewriting in tom using relative positions. *Electr. Notes Theor. Comput. Sci.*, 203(1):3–17, 2008.

[4] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on java. In *RTA*, pages 36–47, 2007.

[5] Michael Kay. *XSLT 2.0 and XPath 2.0 Programmer's Reference (Programmer to Programmer)*. Wrox Press Ltd., Birmingham, UK, UK, 2008.