Declarative Model Transformation Execution Planning

Horacio Hoyos Rodriguez[1]* and Dimitrios S. Kolovos[2]

[1] University of York
York, YO10 5DD
United Kingdom
horacio.hoyos.rodriguez@ieee.org
[2] dimitris.kolovos@york.ac.uk

Abstract. Declarative model transformation languages require that the underlying reasoning engine synthesises an execution plan that guarantees correctness while also providing reasonable performance. Optimization of these execution plans is a hard problem and for languages such as QVT Core, finding an optimal solution is still unsolved. By understanding how a brute force execution plan guarantees correctness, we can find an algorithm to construct the complete solution space of correct execution plans and from it find an optimal solution. In this paper we explore how the complete solution space of execution plans can be constructed, how data dependency analysis can be used to evaluate these plans for correctness and how their performance can be estimated. The results show that our performance estimates are correlated to the observed performance.

1 Introduction

Model to model transformations are usually expressed in dedicated domain specific languages, commonly referred to as *model transformation languages* (MTLs). MTLs express algorithms that describe how elements in a model (or models) are transformed into elements in another model (or models). Declarative MTLs (DTL) only provide logic constructs to express relations between elements in these *candidate* models and the execution engine is responsible for synthesising an *execution plan* that uses these relations to perform the model transformation.

DTLs are attractive because "particular services such as source model traversal, traceability management and automatic bi-directionality can be offered by an underlying reasoning engine" [2]. The QVT Core language (QVTc) [1] uses pattern matching as the primary logic construct. Pattern matching is done over a flat set of variables by evaluating conditions over those variables against the candidate models. Although our optimization algorithms have been developed for QVTc, the ideas presented in this paper are mostly discussed in a language-agnostic level that can be applied to most, if not all, DTLs.

* This research was supported by the ESPRC through the LSCITS initiative.

A *declarative transformation program* (DTP) is a specification written in a DTL. The relations expressed in a DTP can be considered as a set of constraints that must hold for all the candidate models. The execution plan, as a minimum, must check and enforce (by modifying if necessary) the candidate models to ensure that the constraints are satisfied. This paper focuses on the problem of computing an efficient execution plan for a DTP. The correctness problem is also considered, since the lack of correctness will result in models that do not satisfy the constraints. Finding an optimal execution plan for DTPs is a hard problem and well known for other declarative languages such as logic programs, relational query languages and triple graph grammars (TGG). On those domains several approaches have been proposed in the past.

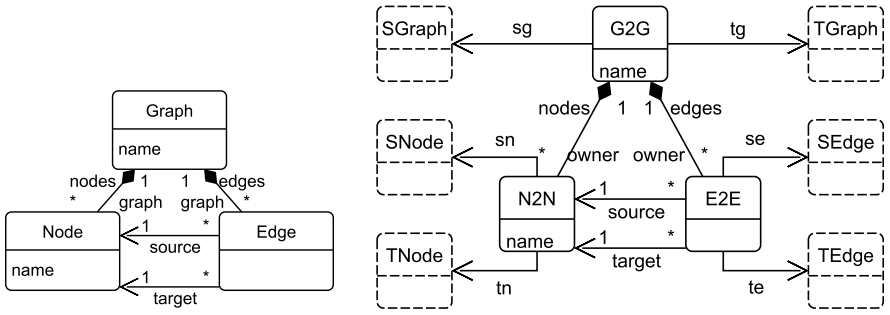
However, DTL optimization problem presents unique challenges. Mainly, the lack of explicit relations between rules and the guarantee for correctness. To understand and overcome these challenges we first propose a representation for DTPs, presented in Sect. 2. Next, Sect. 3 presents how the lack of explicit relations between rules results in solution spaces of considerable size. Following, we show how data dependency analysis can be used to find implicit relations between rules and as a result reduce the solution space by filtering incorrect plans. In Sect. 4 we discus how the execution plans can be evaluated and a cost function derived and Sect. 5 evaluates the cost function against performance results. Finally, Sect. 6 presents related work and Sect. 7 concludes with a summary and future work.

2 Execution Plan and Running Example

In this section we present how an execution plan can be represented by a model and how all possible execution plans for a given DTP can be constructed. Our execution plan model is defined for rule-based DTLs, where the relations are grouped into rules, but can be easily adjusted for other kind of DTLs. In a rule-based DTL, each rule defines a set of types of interest, and relations in the rule are restricted to relations between instances of those types. In this sense, each rule can be viewed as a procedure with a set of parameters (each with a type from the types of interest) and a set of statements. The direction of these parameters, e.g. input or output, is defined by the semantics and syntax of each particular DTL.

Example Our running example is a transformation that creates a copy of a graph, written in QVTc, as presented in Listing 1. The complete QVTc semantics and syntax can be found in [1]. The graph domain is presented in Fig. 1a. To distinguish between the source and target models, we prepend S and T to the graph domain to indicate source and target models respectively. In QVTc, a middle (trace) model must be explicitly defined; its domain is presented in Fig. 1b. Although QVTc is a bi-directional DTL, we have written our example in a uni-directional manner for simplicity. For a bi-directional transformation the proposed approach will work by indicating for which direction the exploration

is to be performed. We have defined two model types: *sourceGraph* and *targetGraph*, and an execution direction from the former to the latter. The complete specification of the transformation is presented in Listing 1.



(a) Graph Domain

(b) Graph2Graph Domain

Fig. 1. The domains for the endogenous QVTc graph to graph transformation.

Listing 1. QVTc graph to graph transformation.

```
1  map g2g in Graph2Graph {
2   sourceGraph(g1:SGraph |) {}
3   enforce targetGraph() {
4    realize g2:TGraph |}
5   where(){
6    realize g2g:G2G |
7    g2g.sg:=g1; g2g.tg:=g2;
8    g2.name:=g1.name;}}

10 map n2n in Graph2Graph {
11  sourceGraph(g1:SGraph,
12    n1:SNode| n1.graph=g1;) {}
13  enforce targetGraph(
14    g2:TGraph |) {
15   realize n2:TNode |
16   n2.graph:=g2;}
17  where(g2g:G2G| g2g.sg=g1;
18    g2g.tg=g2;) {
19   realize n2n:N2N |
20   n2n.owner:=g2g;
21   n2n.sn:=n1; n2n.tn:=n2;
22   n2.name:=n1.name;}}

24 map e2e in Graph2Graph {
25  sourceGraph(g1:SGraph,
26    sn1:SNode,tn1:SNode,
27    e1:SEdge|
28    e1.graph:=g1;
        e1.source:=sn1;
29    e1.target:=tn1;){ }
30  enforce targetGraph(g2:TGraph,
31    sn2:TNode,tn2:TNode |) {
32   realize e2:TEdge |
33   e2.graph:=g2;e2.source:=sn2;
34   e2.target:=tn2;}
35 where(g2g:G2G,sn2n:N2N,
36    tn2n:N2N | g2g.sg=g1;
37    g2g.tg=g2;
38    sn2n.owner=g2g; sn2n.sn=sn1;
39    sn2n.tn=sn2;tn2n.sn=tn1;
40    tn2n.tn=tn2;) {
41   realize e2e:E2E |
        e2e.owner:=g2g;
42   e2e.se:=e1; e2e.te:=e2;
43   e2e.source:=sn2n;
44   e2e.target:=tn2n;}}
```

Rule *g2g* Listing 1 at line 1 defines the relation between `Graph` elements. The `TGraph` and `G2G` elements are defined as realized variables (in lines 4 and 6 respectively) so they will be created in the target and trace model if not found.

Line 8 states that the target graph must have the same `name` as the source graph. Rules *n2n* for `Node` (line 10) and *e2e* for `Edge` (24) elements, exhibit a similar pattern, with additional conditions to verify that we are keeping the correct references throughout. For example, in rule *n2n* we are interested in a `G2G` that is the trace element for the source graph and the target graph (lines 17 and 18). The predicates in the guard of the *where* (lines 35 to 40) of rule *e2e* guarantee that the source and target nodes of *e2* (target model edge) match the source and target nodes of *e1* (source model edge).

According to the semantics of QVTc all variables in guard patterns are input parameters. For *realized* variables it depends on the execution direction, i.e. realized variables in the target domain (which represents the target model) are output parameters. Realized variables in the middle model are always considered as output parameters. Thus, for example, in rule *g2g* the input parameter is: `g1:SGraph`, and the output parameters are: `g2:TGraph` and `g2g:G2G`.

2.1 Transformation Execution

We propose a definition of *correctness* based on the constraint satisfaction requirements: an execution of the DTP results in all the relations holding for all the candidate models.

Definition 1. *A transformation execution is correct, if after execution the relations stated in the DTP are satisfied by the candidate models.*

Using the procedure analogy, execution of a transformation can be accomplished by invoking all the rules it consists of. Thus, the reasoning engine must produce an execution plan that invokes the rules in such a way that the execution is correct. Aspects of the execution plan that affect correctness are: invocation order, number of invocations and how arguments are bound for invocation.

Algorithm 1 Naïve declarative transformation execution

```
1: procedure EXECUTE(Transformation)
2:     repeat
3:         forEach rule r in Rules do                          ▷ R times
4:             forEach combination of P⊙E do                   ▷ P^E times
5:                 if CANBEEXECUTED(r) then
6:                     EXECUTERULE(r)
7:                 end if
8:             End for
9:         End for
10:    until no changes in candidate models                    ▷ D times
11: end procedure
```

Naïve Execution Plan. The simplest, naïve execution plan we can conceive can be represented by Algorithm 1. Since there are no explicit relations between rules, the algorithm iteratively invokes all rules until no more changes are observed in the candidate models. If relations existed, the rules could be invoked

following these relations. For example in QVT Relations[1], the rules in the *where* clauses could be used for a top-down invocation. In line 4, all combinations of *parameters* (P) with model *elements* (E) are found. For example for rule *n2n*, all combinations of existing elements of types `SGraph`, `SNode`, `TGraph` and `G2G` would be found.

Procedure `canBeExecuted` (line 5) tests if the binding process (line 4) bound an element for each of the input parameters, i.e. all elements needed by the rule are available. Procedure `executeRule` (line 6) invokes the rule and modifies the candidate models (i.e. adds or modifies elements).

The reasons this algorithm will correctly execute a QVTc transformation (and possibly any rule-based DTP) are as follows. (i) All possible invocation orders are considered (execution can start at any rule) by invoking all the rules. (ii) Since all rules are invoked indefinitely they are invoked as many times as necessary. (iii) All possible combinations of candidate model elements and rule parameters are used for argument binding. New elements (realized) by a rule execution will be available for binding in the next iteration.

Based on these reasons we introduce an additional constraint for execution plans: *completeness*:

Definition 2. *An execution plan is complete if all the rules in the transformation are invoked at least once.*

2.2 Execution Plan Model

A complete execution of the naïve algorithm can be represented by a list of rules, that starts with the first executed rule and ends at the last one, with elements in the list representing the order in which rules were executed. An execution plan is then any of such lists. We propose to model execution plans as directed graphs, adopting the representation of execution paths used in compiler optimization [2].

In an execution plan graph each node represents a rule. Directed edges are used to represent rule calls. There is a specially designated root node that represents the initial execution point. Outgoing edges of a node are ordered, such that lower order edges are called before higher order ones. As such the execution plan can be regarded as an ordered tree, rooted at the root node. This model is language agnostic, as the references from nodes to rules can be made generic in order to point to rules in any transformation language. A significant difference with the list representation, is that in the execution plan graph, rules are allowed to invoke other rules. We say that the invoked rules are *nested* in the caller rule. However, nesting in this case does not entitle containment as a rule may be invoked from many rules. This representation also allows modelling of loops, for example those from recursive invocations.

Execution plan graphs are executable, with execution semantics defined by outgoing edge order and call edge traversing, i.e. the execution follows a depth-first traversal. For this, our execution plan graph guarantees outgoing edge traversal order. As a result, the execution plan model can be easily used for interpreted execution or as a structure for code generation.

3 Solution Space Exploration

The purpose of the complete solution space exploration is to generate all possible execution plans that produce the same results as the naïve plan. Building the complete solution space is clearly impractical and not scalable. However, this exploratory setup will allow us to identify desired characteristics of execution plans that will lead to correct transformations. The next step of this research will use these characteristics to develop a heuristic approach for efficient synthesis of execution plans.

In the naïve algorithm it is impossible to statically determine how many times a rule should be invoked. Hence, any representation as an execution plan graph would require an infinite number of nodes. One possible way of constructing finite execution plans is to delegate the outer loop control to the reasoning engine and allow the execution plan graph to have loops. Loops allow invocation sequences to be repeated, making it possible to represent any execution with a finite list. Loops in the execution plan that result in infinite loops during execution can be broken by the reasoning engine. With these premises, Algorithm 1 can be represented with our execution plan model, with calls from the root to each of the transformation rules and a loop call from the root to itself after all the rule calls.

Given a DTP, the solution space is constructed by creating execution plans that (collectively) invoke all the rules in all possible orders and nestings. Note that although we can now construct finite execution plans, we still have no way of knowing how many times each rule should be invoked or in what order. This results in an infinite solution space. Next we show that by identifying optimizations in Algorithm 1 it is possible to produce a finite solution space of execution plans that produce the same outcome.

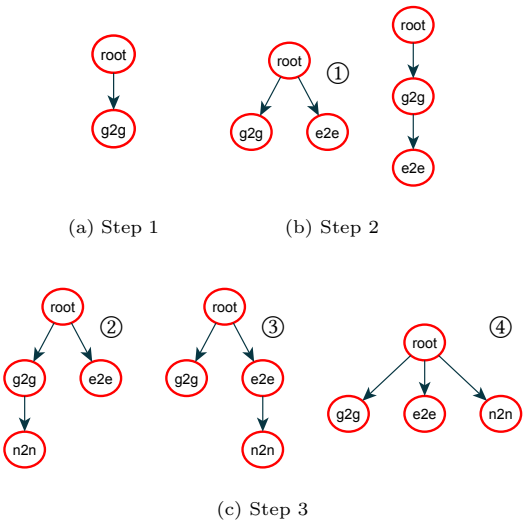
3.1 Limiting The Solution Space

There are two ways of optimizing an algorithm [8]: improve the logic component or improve the control. In our approach the reasoning engine cannot modify the DTP logic, hence optimization has to be done on the execution plan. The complexity of Algorithm 1, due to the loops at lines 2, 3 and the binding process (line 4), is $D \times R \times P^E$. The factor P^E takes into account all possible combinations of parameters and model elements. We have identified two main optimizations of this algorithm: remove the outer loop and minimize the combinations of parameters that are attempted. Next, sections Sect. 3.2, Sect. 3.3 and Sect. 3.4 show how to optimize the former and finally Sect. 3.5 discusses the latter.

3.2 Permutation Solutions

Since Algorithm 1 is bound to terminate (no changes in candidate models), any execution of the algorithm can be represented by a finite set of rule invocations. If the invocation order can be statically computed or the correctness of a given order it can be proved, then the outer loop can be eliminated. Finding all the

combinations of parameters and model elements for each rule is impractical. This search can be optimized be reducing the search space, for example by limiting the search to the elements consumed and produced by a rule.



(a) Step 1 (b) Step 2

(c) Step 3

Fig. 2. Constructing solutions by permutations.

To eliminate the outer loop, we must make sure that all possible executions plans are constructed. Construction of the plans is guided by all permutations of a set of the DTP rules. Although the permutations of a finite set are also finite, the number of possible execution plans grows exponentially as the number of rules increases. We present how this can be achieved, but reiterate that in a real-world scenario this is impractical. The initial plans created will include each rule once. At a later stage, we will add missing invocations that guarantee that the plan is correct.

Example Given the rule permutation {*g2g*, *e2e*, *n2n*}, solutions for this permutation can be created as presented in Fig. 2. The first step, Fig. 2a, consists in adding an edge from the root to the first rule (*g2g*) in the permutation. For the next rule, *e2e*, we can create two different trees by adding invocations from root and from g2g, as presented in Fig. 2b. Selecting *n2n* next, new trees are created by adding edges from all existing nodes in the tree in all existing trees. For example, if we select tree ① in Fig. 2b, the three trees in Fig. 2c will be created. The process will be repeated for the next permutation of rules, e.g {*e2e*, *n2n*, *g2g*} and so on and so forth until all trees for all possible execution plans (where each rule is invoked once) are created.

3.3 Data Dependency Analysis

A rule might not be executed due to missing bindings (conditional in Algorithm 1 at line 5). The outer loop guarantees that the rule is attempted at a later time when new elements have been created and the bindings might be complete.

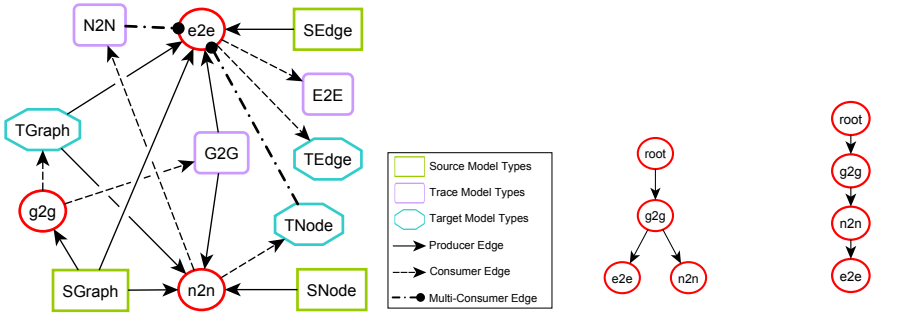
The failed attempt to execute a rule due to the lack of valid input parameters can be understood as a producer-consumer problem. Execution of a rule that consumes a set of types may fail if the rules that produce those types have not been executed. We will use the term *starvation* to refer to a rule that fails to execute due to a lack of elements of its consumed types. Next we present how data dependency analysis can be used to validate and/or complete the permutation solutions to produce execution plans that do not present starvation while ensuring that rules are called as seldom as necessary, and which will result in correct transformation executions.

The producer-consumer relations can be easily derived from the types of the input and output parameters. If a rule has an input parameter of type A, then it will have a data dependency on all rules that have output parameters of type A. These dependencies can be represented in a dependency graph. A dependency graph is a directed graph in which nodes represent rules and types involved in the transformation, and edges represent the data dependencies. Outgoing and incoming edges of a rule represent the types of the output and input parameters respectively. The dependency analysis information can be used to detect starvation in an execution plan. If there is a path in the dependency graph from rule A to rule B, then we say that *rule B is a consumer of rule A*. Conversely, *rule A is a producer of rule B*.

The outer loop of Algorithm 1 also guarantees that all elements created during execution are taken into consideration in the binding process. We introduce the concept of *thoroughness* to validate that an execution plan satisfies this constraint.

Definition 3. *A transformation execution is* thorough *if all the created elements of a rule are consumed by the rule's consumers.*

Example The dependency graph of the QVTc graph to graph transformation of Listing 1 is presented in Fig. 3a. It is important to note that we have made a distinction for *multi-consumer* edges to represent consumer edges for types that are used by more than one parameter. The data dependency analysis is used to evaluate the plans to the right in Fig. 3. The plan in Fig. 3b will present starvation when rule *e2e* is called from rule *g2g* as rule *n2n* would not have been executed yet and hence there would be no existing elements of types `SNode` and `N2N` to bind rule *e2e* parameters. On the other hand, the plan in Fig. 3c presents starvation due to *multi-consumer* relations: the call from *n2n* to *e2e* will starve because in the first execution rule *n2n* will have only produced one `SNode` and one `N2N`, and rule *e2e* consumes two of each. Regarding thoroughness, the plan in Fig. 3b is not thorough as elements produced by rule *n2n* are not consumed by any rule despite *e2e* being a consumer.



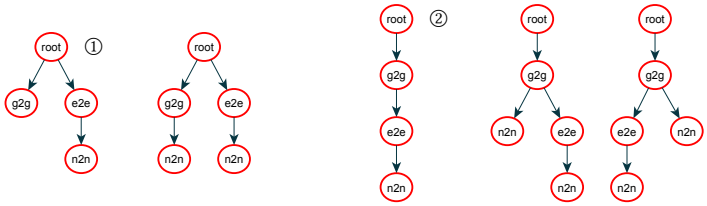
(a) Dependency Graph for the graph to graph transformation. (b) Case A (c) Case B

Fig. 3. Validate starvation and thoroughness.

3.4 Producer-Consumer Solutions

The set of permutations plans is filtered to eliminate plans that result in starvation and the remaining plans are amended to guarantee that they are thorough. The amendment process involves adding missing calls for all the producer-consumer relations. As a result, the solution space of all correct execution plans for a given transformation is constructed.

Example Rule *n2n* is a consumer of rule *g2g* and hence plans ① and ② in Fig. 4 are missing invocations from rule *g2g* to rule *n2n*. In Fig. 4a the invocation is added to plan ① resulting in the plan to the right. For plan ② since invocation order matters, the invocation to *n2n* can be either done before or after the existing invocation to *e2e*. For one invocation we can modify the existing plan, for the other(s) we need to create an additional plan. The resulting plans are shown to the right of Fig. 4b.



(a) Modify (b) Modify and create

Fig. 4. Completing solutions with thoroughness considerations.

3.5 Input Variable Relations and Binding Solutions

Finally, we consider the case of binding during invocation. If *n2n* is being invoked, we need to bind four input variables. In practice, the rule guards define precise restrictions on these variables. For example, the predicate $n1.graph = g1$ indicates

that $n1$ and $g1$ are related through the $SNode :: graph$ attribute. Hence, if we have one it is possible to derive the other. Due to the expressive power of OCL we limit our analysis to predicates of the form $< var > . < property >=< var >$. From this, it is possible to identify primary and secondary variables, where secondary variables are derived from primary ones. In most cases it is possible to pick one primary and derive the rest from this one. By using the context information and exploiting derivations we are reducing the number of possible input element combinations that are attempted for each rule.

Consider the case in which rule *n2n* is being invoked from *g2g*. In this case, we can either choose `g2:TGraph` or `g22:G2G` as the primary variable. This results in two versions of the invocation. By adding all possible binding alternatives for all rules, we can produce a new set of solutions that represent all possible plans with all possible bindings.

As the number of rules and the number of edges in the dependency graph increases, the proposed approach to generating the complete execution space becomes infeasible. This issue can be resolved by applying search-based algorithms to find a good enough solution without requiring to do the exhaustive execution space search. This is a topic for future research.

4 Execution Plan Evaluation

To express starvation and multi-consumer requirements we introduce the concept of *feasibility* and from it we can define execution plan *correctness*:

Definition 4. *An execution plan is feasible if all producers of a rule have been called before the rule is called, and if all rules with multi-consumer relations are not invoked from the producer of the multi-consumer relation.*

Definition 5. *An execution plan is correct if it is complete and feasible and results in a thorough execution.*

Notice that we express thoroughness as a property of the result and not the plan itself. The reason for this is that, in practice, thoroughness cannot be evaluated on a plan unless details of the particular DTL and modelling technology are known.

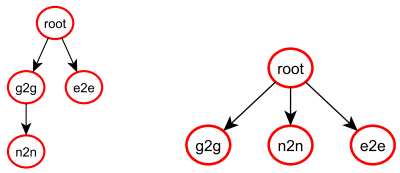
This definition of execution plan correctness overlaps with the correctness and completeness definitions formulated in [11] for triple graph grammar execution algorithms. This definition, however, allows validation of plans (algorithms) for declarative, rule-base, transformation languages.

Example Consider the plan in Fig. 5a. Depending on the modelling technology it is possible that not all `SNode` elements are contained in a `SGraph`, in which case this plan will not be thorough, i.e. there are missing calls from other places where *SNodes* can exist. Indeed, a particular language may allow elements to be created outside rules, e.g. helper methods, in which case we would need to analyse such methods and add additional calls to consume those elements.

4.1 Performance Evaluation

From the set of correct execution plans, we are interested in selecting the best plan in terms of performance. In particular, we are interested in execution time since it is one of the key performance aspects noticed by users. The key factors that affect execution time are the number of times a rule is invoked and the binding process.

Consider the plans in Fig. 5, both of which are correct. We are interested in evaluating their performance, in order to select the optimal one.



(a) Case A (b) Case B

Fig. 5. Validate starvation and thoroughness.

When a rule is invoked from the root, the binding process must query the candidate models for all elements of each of the input types and produce all possible combinations. For P parameters and E elements for each parameter type, each rule is invoked P^E times. When a rule is invoked from a rule different from the root, it obeys a producer-consumer relation and thus a model search is not required for elements of the types of the producer-consumer relation. For example, if rule A and rule B have a producer consumer relation via type X, and A produces one element of type X, then when invoking B from A, we only need to invoke it once: to consume the new element.

Inspecting the plans in Fig. 5 we can then say that the plan in Fig. 5a has a total of $2 \times P^E$ invocations, because rule *n2n* is only invoked once per invocation of rule *g2g*. On the other hand, the plan in Fig. 5b has a total of $3 \times P^E$ invocations. From this it is possible to conclude that the plan in Fig. 5b would have a worse execution time than the plan in Fig. 5a.

"The experience with relational system has shown that the main purpose of a cost model is to differentiate between good and bad executions, in fact, it is known, from the relational experience, that even an inexact cost model can achieve this goal reasonably well." [9]. Thus, our cost function is derived intuitively and results show that, to this point, further detailed analysis of execution is not required. We define the cost of executing a rule r as $\phi(r) = \phi_e(r) + \phi_i(r)$, where ϕ_e is the execution cost and ϕ_i is the invocation cost. ϕ_e depends on the number of statements and particular semantics of the DTL. However, since all rules are written in the same language and all rules are invoked in all the plans, we can assume this cost to be constant: $\phi_e(r_i) = 1$. The invocation cost, $\phi_i(r)$,

depends on the number of times each rule is invoked and the number of outgoing calls from the rule ($O(r)$):

$$\phi_i(r) = X(r) \times \sum_{r' \in O(r)} \Big(\phi_\beta(r') + \phi(r') \Big) \tag{1}$$

Where $X(r)$ is the number of times a rule is invoked and $\phi_\beta(r)$ is the cost of binding the parameters of the invoked rule. In general $X(r) = P^E$, but for static analysis we can select an arbitray value, e.g. 100, that is sufficiently larger that the binding cost to discourage the use of loops as we know they affect performance adversly. The binding cost $\phi_\beta(r)$ accounts for the binding of input types that don't belong to the producer-consumer relation. For example, when *n2n* is invoked from *g2g*, `g2:TGraph` can be bound directly from the produced TGraph, but `n1` and `g1` must be bound separately. If any of the secondary variables is derived via a multi-value property, then the binding will also include the cost of a loop. For static analysis we consider property loops less expensive than all instances loops (P^E) and have defined them as an order of magnitude cheaper. Finally, the binding cost depends on the underlying modelling technology. For example in EMF, navigating an undefined opposite relation is more expensive than a direct one. Since the cost of a rule depends on all the rules it invokes, the cost of a plan is $\phi(root)$.

5 Results

The described approach has been implemented in Java, using the JGraphT[3] graph library for modelling the Dependency Graph and the Execution Plan Graph. For the Graph2Graph transformation the complete solution space consists of 240 possible execution plans. We took a sample of 20 plans and executed the transformation using a test model with 100k elements. For this evaluation, each graph has the same number of nodes and edges and the model has as many graphs as a graph has nodes. The idea was that loops for either all instances or multi-value properties where comparable.

Figure 6 shows the execution time for the 20 sample plans. The series represent the groups identified by the Honest Significant Differences (HSD) using the statistical software R. Since the ANOVA[12] test showed that the executions times are significantly different, the HSD results show a correlation between the cost function and the execution times.

It is important to note that the difference in execution times between Group *c* and the others is around two orders of magnitude. By inspecting the plans with longer executions times we found that those plans where the cases in which the binding resulted in derivations that used multiplicity attributes, for example deriving nodes from a graph. This sort of derivations insert additional loops during execution, which translate into higher costs, i.e. higher $X(r)$. The results show that our cost function is good enough to differentiate good and bad executions.

[3] Available online, `http://jgrapht.org` (viewed April 2016)

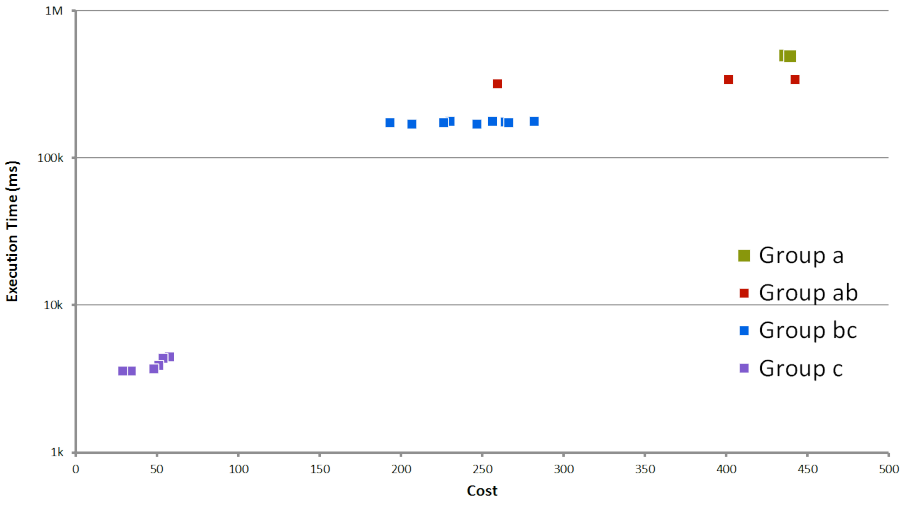


Fig. 6. Execution times for the execution plans sample.

Although the results are statistically significant, the HSD results also suggest that bigger sample sizes are needed. This is the objective of future work.

6 Related Work

Data dependency analysis has been used for instruction scheduling, constant propagation [14], parallel processing [16], and others, in compiler optimization, but most of the literature is related to imperative languages. Control synthesis for declarative languages is usually made on a statement by statement level [4, 3] following the constraints imposed by the language semantics and observing data dependencies. However, these approaches do not explore alternative plans in order to compare them with respect to performance.

In the case of other DTL such as ETL [7] and ATL [6][4] the algorithm presented in [6] is roughly what is implemented in both execution engines. Rules are executed in the order they are defined in the DTP in two passes. Once, to create all instances of output parameters and a second time where the logic of the rules is executed. In either case there is no existing analysis or optimizations. In Prolog, optimizations such as intelligent backtracking[10] use data dependence analysis to tag previous clauses and provide smarter backtracking choices. However, this type of optimizations are done at runtime and we are interested in providing a compile time solution.

[4] In their general form ETL and ATL are hybrid transformation languages, i.e. they mix declarative and imperative semantics, but it is possible to use them purely declarativly.

Data dependency analysis is also used in the QVTc compiler[15] of the Eclipse Project. Their use of dependency analysis is more detailed as they also consider the dependencies generated from the access and modification of attributes. However, they do not do a solution space search and correctness is not solely achieved by the generated schedule (execution plan) but also by runtime facilities that, for example, allow rules to be queued for later execution. Our plans do not require these runtime support. However, we are aware that without property analysis there are some transformations for which our plans will not be correct.

An important difference with functional languages is that since there exists an explicit relation between functions (i.e. one function is computed by invoking a set of other functions), there is an initial invocation structure and the optimization problem is related to finding the correct set of function invocations to obtain a result as opposed to finding an order in which to invoke these functions. The same can be said for declarative queries languages. In the case of DryadLINQ, a declarative queries system, in [13] these relations are exploited in order to fuse them into optimized loops during code generation.

As stated previously, we are interested in providing optimizations based on static analysis where the *execution plan* is determined at compile time. In [9] and optimization for the Logic Data Language (LDL) is presented. The focus on optimization for the Horn clause queries by first, identifying the solution space and second, deriving a cost model. In our approach we follow a similar methodology. Their solution space consists of binary expression trees that represent the Horn clauses and thus it is not directly applicable to DTLs where there is no logic relation between rules. For the cost function, they assume that each operation there is a predefined list of methods to calculate the cost. The reason for this is that these costs have been previously determined in the domain of relational query languages. Part of our research is related to deriving such cost functions.

Data dependency analysis is also used in [5] for deriving test cases for complete coverage testing in TGG. However, the data dependencies are per pattern, as opposed to individual types. In our case dependencies are per type, and thus one rule will depend on rules that produce all or some of the consumed types. This results in more complex dependency graphs.

In the domain of graph transformations, [11] presents an optimization to TGG execution called Precedence TGGs. They identify data dependencies between rules, but use this information to topologically sort the nodes in a source graph (model) as opposed to defining an execution order for the rules. This is due to the fact that TGG execution is defined by source graph traversal as opposed to rule traversal. Our complete solution space exploration shows that there are different alternatives to execute the same transformation and our approach can be used as a base to identify the optimal one. In [11] there is no mention of the different execution alternatives and if the proposed algorithm results in an optimal (sufficiently good execution) for a given TGG.

7 Conclusions

In this paper we proposed a novel model to represent execution plans of declarative model transformations and presented how the complete solution space of execution plans for a given transformation program can be constructed. Further, we introduced a method for determining the correctness of execution plans based on data dependency analysis. We showed how the correctness criteria can be used to synthesise efficient execution plans that guarantee correct transformation execution while minimizing the number of rule invocations. Finally, we presented how predicted runtime performance could be evaluated by computing the cost of the execution plans and validated the results by execution a sample of the plans in the execution space. The ideas for constructing the complete solution space can be incorporated into a search-based algorithm that would result in a sufficiently good solution without requiring a complete solution space exploration. These contributions were presented in a language-agnostic level that facilitates its incorporation to other declarative model transformation languages.

Acknowledgments

The authors would like to acknowledge the contributions of Edward Willink in the establishment of some fundamental concepts of this work.

References

1. Object Management Group (OMG): Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification. OMG Document Number: formal /2015-02-01. `http://www.omg.org/spec/QVT/1.2/`. (2015).
2. Aho, Alfred V. Lam, Monica S. Sethi, Ravi, Ullman, Jeffrey D. Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)
3. Antoy, Sergio, Hanus, Michael: "Frontiers of Combining Systems: Third International Workshop, FroCoS 2000, Nancy, France, March 22-24, 2000. Proceedings". In: ed. by H. Kirchner and C. Ringeissen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. Chap. Compiling Multi-paradigm Declarative Programs into Prolog, pp. 171–185.
4. Halbwachs, Nicolas, Raymond, Pascal, Ratel, Christophe: Generating efficient code from data-flow programs. In: Programming Language Implementation and Logic Programming, pp. 207–218 (1991)
5. Hildebrandt, Stephan, Lambers, Leen, Giese, Holger: "Complete Specification Coverage in Automatically Generated Conformance Test Cases for TGG Implementations". In: Theory and Practice of Model Transformations: 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings. Ed. by K. Duddy and G. Kappel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 174–188. ISBN: 978-3-642-38883-5.
6. Jouault, Frédéric, Kurtev, Ivan: Transforming Models with ATL. In: Satellite Events at the MoDELS 2005 Conference. Ed. by J.-M. Bruel, pp. 128–138. Springer Berlin Heidelberg(2006)

7. Kolovos, DimitriosS. Paige, RichardF. Polack, FionaA.C. The Epsilon Transformation Language. In: Theory and Practice of Model Transformations. Ed. by A. Vallecillo, J. Gray, and A. Pierantonio, pp. 46–60. Springer Berlin Heidelberg(2008)
8. Kowalski, Robert: Algorithm = Logic + Control. Commun. ACM 22(7), 424–436 (1979)
9. Krishnamurthy, Ravi, Zaniolo, Carlo: "Optimization in a logic based language for knowledge and data intensive applications". In: Advances in Database Technology—EDBT '88: International Conference on Extending Database Technology Venice, Italy, March 14–18, 1988 Proceedings. Ed. by J.W. Schmidt, S. Ceri, and M. Missikoff. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 16–33. ISBN: 978-3-540-39095-4.
10. Kumar, Vipin, Lin, Yow-Jian: A data-dependency-based intelligent backtracking scheme for PROLOG. The Journal of Logic Programming 5(2), 165–181 (1988)
11. Lauder, Marius, Anjorin, Anthony, Varró, Gergely, Schürr, Andy: Bidirectional Model Transformation with Precedence Triple Graph Grammars. In: Modelling Foundations and Applications. Ed. by A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Störrle, and D. Kolovos, pp. 287–302. Springer Berlin Heidelberg(2012)
12. Mead, R. Gilmour, S. G. Mead, A. Statistical Principles for the Design of Experiments. Cambridge University Press (2012)
13. Murray, Derek Gordon, Isard, Michael, Yu, Yuan: Steno: Automatic Optimization of Declarative Queries. SIGPLAN Not. 46(6), 121–131 (2011)
14. Pingali, Keshav, Beck, Micah, Johnson, Richard, Moudgill, Mayan, Stodghill, Paul: Dependence flow graphs: an algebraic approach to program dependencies. In: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '91, pp. 67–78. ACM, Orlando, Florida, USA (1991)
15. Willink, Edward D. "Optimized declarative transformation First Eclipse QVTc results". In: Scalable Model Driven Engineering: 3rd Workshop, BigMDE 2016, Vienna, Austria, July 9, 2016. Proceedings. Ed. by D. Kolovos, D. Di Rusco, N. Matragkas, J. Sánchez Cuadrado, I. Rath, and M. Tisi. To appear. CEUR-WS, 2016.
16. Wolfe, Michael, Banerjee, Utpal: Data dependence and its application to parallel processing. International Journal of Parallel Programming 16(2), 137–178 (1987)