Partial Evaluation of Model Transformations

Ali Razavi
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
arazavi@swen.uwaterloo.ca

Kostas Kontogiannis
Department of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
kkontog@softlab.ntua.gr

***Abstract*—Model Transformation is considered an important enabling factor for Model Driven Development. Transformations can be applied not only for the generation of new models from existing ones, but also for the consistent co-evolution of software artifacts that pertain to various phases of software lifecycle such as requirement models, design documents and source code. Furthermore, it is often common in practical scenarios to apply such transformations repeatedly and frequently; an activity that can take a significant amount of time and resources, especially when the affected models are complex and highly interdependent. In this paper, we discuss a novel approach for deriving incremental model transformations by the partial evaluation of original model transformation programs. Partial evaluation involves pre-computing parts of the transformation program based on known model dependencies and the type of the applied model change. Such pre-evaluation allows for significant reduction of transformation time in large and complex model repositories. To evaluate the approach, we have implemented QvtMix, a prototype partial evaluator for the Query, View and Transformation Operational Mappings (QVT-OM) language. The experiments indicate that the proposed technique can be used for significantly improving the performance of repetitive applications of model transformations.**

I. Introduction

Model Driven Engineering (MDE) is an emerging paradigm whose fundamental objective is to raise the level of abstraction in the software development process. The basic premise of Model Driven Engineering (MDE) is that software artifacts can be represented as collections of models, which adhere to standardized meta-modeling formalisms such as the Meta Object Facility (MOF) or the Eclipse Modeling Framework (EMF). In addition to serving as a medium for documentation, MDE aspires to promote models to first-class development artifacts. To this end, various model transformation frameworks have been proposed to facilitate bridging levels of abstraction between models, and even use models to generate executable code [4].

To be practically relevant, MDE should also support modern iterative and incremental development processes (a.g., Agile methodologies). When commencing the development of a system in MDE style, the initial models need to be transformed into multitudes of formats to realize a concrete system. The models then, being first class entities, should be refined for maintenance purposes, which necessitates their re-transformation. Unlike code-based development, simple make-like programs are, however, not adequate for effective handling of model transformations. This is mostly because models are modularized in a rather different way than source code, with a different coupling and cohesion characteristics. MDE encourages developing models that describe a certain perspective of the underlying system, and the dependencies of such models with other models in a project tend to be context dependent. For instance, when using UML to describe a system and transforming it to Java code, the details of what constitutes an underlying class in Java can be scattered across, amongst others, class diagrams and sequence diagrams. Consequently, a change made to the class diagram would need to be weaved with the sequence diagram to form the target Java class.

In this context, a common problem that arises is that in order to keep these models synchronized and consistent with each other, one has to re-apply all transformations in all models so that a complete re-generation of all model artifacts is achieved. Plain re-transformation can be intractable for large models and/or complex transformations. Therefore, *incremental* model transformations are considered to be the proper mitigation strategy for keeping the source and target of transformations synchronized [9].

Most existing MDE frameworks utilize transformations that are denoted either by a declarative style language–in the form of logical relations or functional mappings–for cases in which there exist clear correspondences between the elements of both sides of the transformations– or by imperative style languages when mutual relationships of models are less evident. As we review in the next section, the body of research work on incremental model transformation primarily addresses the issue for the declarative model transformation languages. However, incremental transformation of imperative model transformation languages, due to the familiarity of developers with this style and also the richer expressiveness they offer, are just as important. In this paper, we propose an approach to increase the performance of model transformation process based on the *partial evaluation* of transformation programs. Partial evaluation is an established methodology in programming languages research that is based on the premise that programs can

978-1-4673-1067-3/12/$31.00 © 2012 IEEE

ICSE 2012, Zurich, Switzerland

be executed on a subset of input data known *a priori*, so as to generate a *residual* program, whose expressions are, to the extent allowed by the availability of known data, are statically pre-evaluated. Thus, the residual program, when executed over the dynamic inputs, does not need to compute the parts of the code that only correspond to known inputs. Performing fewer computations at runtime, residual programs are expected to perform better than their original counterparts [10], [1], [12], [11].

The basic premise of our proposed approach is that model transformations are also a kind of software programs, which get as input instances of a metamodel (e.g., MOF or similarly Ecore). When a transformation is applied iteratively, the altered elements of the source model can be considered as dynamic data, and the invariant fragments as static. The objective is to reduce the number of computations performed when a complex transformation is invoked by pre-computing and storing in a residual transformation the expressions that are not affected by a model change. As a proof of concept, we have developed a prototype of a partial evaluator for a subset of OMG's imperative model transformation language, QVT Operational Mappings[14]. Transformations denoted in QVT, and in other model transformation languages alike, make extensive use of collection operations to manipulate the elements of input models and their containers, in order to form the target model. Therefore, our technique primarily focuses on the specialization of these types of expressions. In this respect, we have implemented a prototype partial evaluator in QVT-OM itself. This design decision has two important methodological consequences. On the one hand, it adheres to the general philosophy of model driven engineering, which strives to treat all major software component as models; in particular, the object of our partial evaluator–i.e., QVT transformation–are themselves treated as models, and are manipulated as such. On the other hand, this enables the concept of self-application, that is, specializing the partial evaluator by itself.

The rest of the paper is organized as follows. Section II discusses related work and lays the foundations for the rest of the paper. Section III presents the outline of the partial evaluation process. Section IV presents the detail of the partial evaluation technique for the QVT-OM model transformation language, explained through a running example transformation. Section V reports and comments on experimental results. Finally, Section VI concludes the paper, and points to a number of future research directions.

II. Background and Related Work

A. Background

Partial evaluation of software programs refers to the approach whereby parts of the program are pre-computed with values of known inputs so as to yield a new *residual* program which, when executed to a set of known and unknown inputs, will compute only the parts of the program that correspond to the unknown inputs. Because the residual program performs fewer computation, it is expected to run faster than the original program. More specifically, let $[\![prog]\!]_{\mathcal{L}}[in] = out$ denote that $prog$ is a program specified in language $\mathcal{L}$ and produces output out for the sequence of inputs in. Suppose that the first m inputs of the program are *known* before the execution time and they are denoted as $(k_1, ..., k_m)$, and the rest of the inputs are *unknown* and are denoted as $(u_1, ..., u_n)$. A partial evaluator for $prog$ is a program such as $\mathcal{PE}$, inputs of which are the source code of program $prog$ in language $\mathcal{L}$, and its set of known inputs. It transforms the input program to a *specialization* of it, referred to as $prog_s$, with respect to this set of known inputs. Using the same notation, we can denote $[\![\mathcal{PE}]\!][prog, k_1, ..., k_m] = prog_s$. Because some of the expressions in $prog$ are replaced by statically pre-evaluated values in $prog_s$, the latter is intuitively expected to run faster than the former. Partial evaluation is a form of program transformation as it produces another program as output. Running this program on the remaining inputs (i.e., the unknowns) results in the same output; thus $[\![[\![\mathcal{PE}]\!][prog, k_1, ..., k_m]]\!]_{\mathcal{L}}[u_1, ..., u_n] = out$.

Partial evaluation usually consists of two phases. During the first phase, *Binding Time Analysis*, the source code of the program is analyzed with respect to the set of known inputs and its expressions are annotated as either *static* or *dynamic*. Following this analysis, in the second phase, the constructs that are determined to be static are evaluated, starting from the inputs and progressively replacing each static expression with its evaluated value. Dynamic expressions in contrast are substituted with symbolic expressions that are derived from the values of static expressions and other dependent dynamic expressions. The second phase yields a program called *residual* program, which only needs the unknown subset of the inputs of the original program to run. There are two common strategies to carry out these two phases; explicitly and separately in *offline* evaluators, versus *online* evaluation by performing static analysis *on the go* along with specialization [11]. Either way, all the statically computable expressions of the original programs are replaced with pre-evaluated values in the residual program, thus will not be recomputed during runtime.

B. Related Work

An elaborate classification of various model transformation approaches is presented in [4]. A prominent model transformation framework is the OMG's Query, View and Transformation (QVT) which specifies three languages [14]. QVT Core and QVT Relations both define mappings between the two sides of transformations in a declarative fashion. Both languages are also envisioned in the standard to have built-in support for incremental and bi-directional transformation. For specifying more sophisticated transformations whose mappings are not as straightforward as those expressible by the Core and Relations languages, OMG

offers the QVT Operational Mappings (QVT-OM) language. This language provides a hybrid collection of imperative and declarative constructs, and is designed to operate in one direction with no direct support for incremental execution of transformations. As discussed, this essentially results in redundant computation of unaffected model elements, when the transformations are used iteratively. Amongst other proposed model transformation frameworks are the Atlas Transformation Language (ATL) [2], and Triple Graph Grammars (TGG) [6]. ATL programs can be used to perform syntactic or semantic model transformations and run on top of a specialized virtual machine. Triple Graph Grammar (TGG) is a graphical, declarative, incremental and bi-directional model transformation methodology based on graph transformation [16]. TGG is shown to be semantically aligned with QVT Relations [7].

In the area of model synchronization, one solution that relates to our work in the sense that it enhances existing transformations to achieve model synchronization is SyncATL, proposed by Xiong et al. in [18]. The authors have proposed an extension to the bytecode of the ATL virtual machine [2], whereby supporting automated, backward synchronization of models linked by an ATL transformation. For the forward synchronization, SyncATL relies on re-invoking the transformation and merging the results with the existing target. However, SyncATL does not address incremental synchronization, as the framework relies on re-executing the transformation in its entirety for forward change propagation. Another framework with the theme of building incremental synchronization around existing transformation engines is presented in [8]. The Tefkat [13] transformation engine, which has a declarative, logical flavor, is decorated with support for incrementality. In Tefkat, transformation rules are specified as logical predicates, which are reduced using SLD resolution. Their synchronization framework avoids redundant computations in the successive transformation of the same model by preserving the intermediate SLD trees.

There exists a vast body of research on partial evaluation. An excellent introductory resource is the book authored by Jones et al. [11], which also provides an exhaustive list of references to the existing literature. More concise entry points to the area of partial evaluation can be found in [10] and [3]. Sundaresh et al. [17] were amongst the first researchers to exploit partial evaluation for deriving incremental programs, albeit in the context of code-driven programming languages. The indexing model employed in [15] is similar to the scheme we have used for accessing the elements of cached collections. To our knowledge, however, all partial evaluation frameworks have hitherto focused on the specializing the source code written in general purpose programming languages; we are not aware of any other research work that leverages partial evaluation techniques in the context of model transformation languages. As we discuss in Section IV, there also tend to be differences in the specialization of model transformations, which rely extensively on collection manipulation, and ordinary programs, wherein collection-based caching strategies are often not as relevant.

III. System Architecture and Overall Process

A. System Architecture

The architecture of QvtMix, as depicted in Figure 1, is structured as a two-layer system underpinning a static analysis pipeline. The first layer is an OCL expression evaluator module serving the Partial Evaluator pipeline in the second layer, which accepts as input a QVT program and a set of known input data, and produces a new partially evaluated (or specialized) OCL program. Below, we discuss in more detail the major components of the proposed architecture:

- *OCL Evaluator*: The OCL Evaluator constitutes the first layer of the architecture and serves to evaluate the results of expressions as they are "statically" reduced by the specializer. The OCL Evaluator forms the basis of a meta-circular interpreter, that is, it is implemented in the same language it interprets, i.e., QVT-OM. It maintains a lexcially scoped environment to store partially evaluated values for static variables and an expandable function environment to store residual functions generated during the specialization phase.
- *OCL Parser*: This component allows for the parsing and linking of OCL expressions and emits their abstract syntax. As our partial evaluator operates on Ecore elements, the generated abstract syntax is represented as a Ecore compliant model.
- *Binding Time Analyzer*: This component is responsible for tagging variables and expressions as STATIC or DYNAMIC. The BTA algorithm infers the binding time of expressions from the subset of input data for which the subject program is being specialized. For the case of model transformations, we partition the input model elements into fixed and variable, by annotating their corresponding elements in the input *metamodel* with a change specification. More specifically, the change specification designates for each class, container and attribute the possibility of modification (by tagging them as VAR) or lack thereof–by tagging them as FIXED. The BTA propagates the VAR and FIXED tags through the expressions using a set of inference rules. The result is the abstract syntax (in Ecore format) annotated with binding-time tags on each expression node.
- *Specializer*: This component accepts the abstract model of the transformation and produces a model that denotes the partially evaluated program. The Specializer operates in a syntax-directed fashion; it basically acts as a visitor that traverses the annotated abstract syntax tree, and applies a certain production rule to each

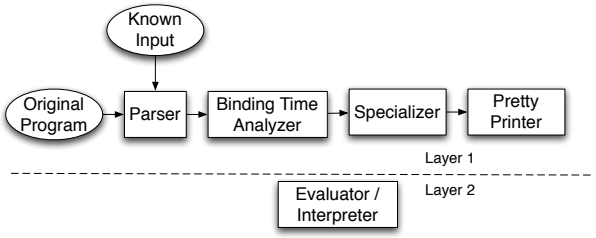

Figure 1. System Architecture Diagram

expression. It uses the BTA annotations to pinpoint the possibilities of static evaluation, passes such expressions with appropriate values to the OCL Evaluator, and creates a cache of the results in the program. The static expressions are either replaced with constants or subsequently reduced to reference the static values. The transformation's abstract syntax model is modified in such a way to incorporate the cache of pre-evaluated values and also residual expressions.

- *Pretty Printer*: This component accepts the abstract model of the partially evaluated transformation and produces source code text executable by the QVT-OM engine.

B. Partial Evaluation Process

The overall process for the partial evaluation of model transformations is depicted in Figure 2. The initial step is parsing the subject transformation and obtaining its Abstract Syntax Tree (AST). Along with the transformation, the framework also loads the input and output metamodels, and the change annotation information–that is, the annotations that tell the user which elements of the source models are fixed, and which ones are variable. The BTA framework accepts user annotations of the input meta-model to guide the partial evaluator about the possible classes of changes and locality thereof, according to which the AST of the input transformation is specialized. In the implementation, we use the 'eAnnotation' fragment of Ecore to augment the meta-models with the FIXED and VAR meta-attributes, which are accessible to the QVT-OM partial evaluator.

Our intuition is that this semi-automatic approach is often congruent with the practice of software development. Just like when programming, developers' interaction with models adheres to some pattern of locality and temporarily, that is, the most immediate changes are most likely to occur in the most recently modified part of the model, and are likely to be of the same type of the most recent modification. In programming, a developer is most likely to continue modifying a class after its inception, by adding several methods to it for example. Likewise, the model developer will likely add further operations to a UML class in a sequential manner. Therefore, the kind of guidance required can be semi-automatically inferred from the user's behavior and the strategy adopted for specialization can be carried out in a tractable fashion.

After the BTA, the partial evaluator traverses the AST of the transformation, and for each transformation rule found in the AST it performs the following actions. The statically evaluated expressions whose results can be fully evaluated at this time from the input model are evaluated and their results are either stored in a table or inlined in the code. More specifically, for scalar values, the result of the yielding expressions is replaces the expressions, whereas for collections the index up to which the input elements are processed, is kept, and the values are stored in a static cache outside the scope of the expression.

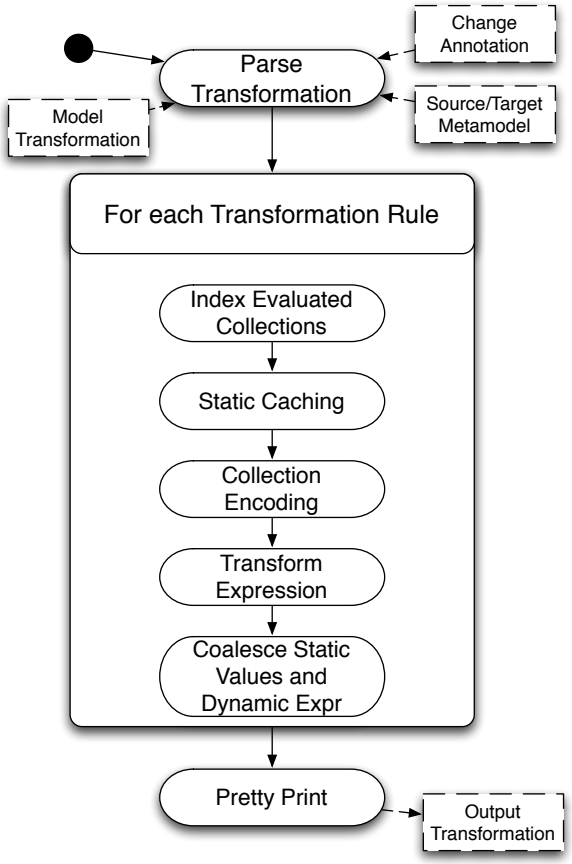

Figure 2. Overall Process of Partial Evaluation

The static caching step involves encoding the values of the expression table created in the previous step in the appropriate points in the AST. This is typically in the form of local variables for each rule. More specifically, for each variable (or interim value) in each *OclCallExpr* a local dictionary variable corresponding to its statically evaluated result is created and populated with the static result of the

expressions. Embedding these values sometimes requires minor manipulation in the AST. The next two steps deal with partitioning the collection expressions that appear in the statements in the body of the rule where the output element's values are populated. Each expression is partitioned into a static part, encoded as a dictionary or list in the previous step, and the dynamic part that has to be evaluated. The expressions need to be transformed into a new form that operate only on the modified parts of the input. This is generally done by utilizing the information stored in the table created in the second step. Finally, the expression has to merge the statically computed results with those evaluated dynamically.

IV. Partial Evaluation

The partial evaluator is, in essence, a higher order transformation written in QVT; it is implemented as a visitor that uses a certain morphism strategy to evaluate static expressions and mix them with dynamic expressions in the form of residual code. The general algorithm, thus, dispatches these specific transformations for each AST node type. The AST is also represented as models compliant to the QVT metamodel depicted in Figure 3, which allows us to denote the partial evaluator in QVT-OM.

The most principal ones for QVT model transformations are the AssignExp and VariableInitExp nodes. These two are the syntactic constructs in QVT that are meant for establishing relationships between source and target elements. They are used in the ordinary sense of the assignment (that is, for assigning values to variables), and also for establishing relationships between source and target model elements. The specialization is further narrowed for specific cases of IteratorExp, which take into account the semantics of some of the standard library functions. Figure 4 presents the simplified outline of the specialization process. The specializer is basically a QVT transformation which has two mapping rules per AST node: mix and reduce. The latter processes fully static expressions and replaces them with inline static values. The former strategy, on the other hand, deals with dynamic expressions and tries to mix their dynamic part with their reduced static sub-expressions. It traverses the abstract syntax tree according to the semantic information of the language. As it is illustrated in Figure 4 under the mapping rule for OperationalTransformation (line 14-19), the structure of the QVT model is traversed in a recursive fashion. The dispatching of *mix* mapping rules is carried out polymorphically according to the object oriented rules of the QVT-OM language, and the inheritance hierarchy of the input meta-model, i.e., the QVT metamodel itself which is loaded on line 1.

The code also provides a feel of how higher-order transformations are denoted in QVT. In Figure 5, the mixer algorithm for one of the nodes of the AST, i.e., VariableInitExp is presented. This portion of the mix algorithm for this

```
01: modeltype QVT uses
    qvtoperational::expressions('http://...');//...

02: transformation QvtMix(
      in inModel : QVT, out residue : QVT
    );

03: property qvtSrc :
      OperationalTransformation = null; //...

04: intermediate class BTA {};
05: intermediate class STATIC extends BTA {};
06: intermediate class DYNAMIC extends BTA {};

07: helper ocl::ecore::OCLExpression::bta() : BTA
      {//...}

08: main() {
09:     qvtSrc := inModel.rootObjects()
          [OperationalTransformation]->
            asOrderedSet()->first();
10:     qvtSrc.map mix();
11: }

12: mapping ASTNode::mix() : ASTNode
      { init {} population {result := self;}}
13: mapping ASTNode::reduce() : ASTNode
      {Init {} population {result := self;}}

14: mapping OperationalTransformation::mix() : ASTNode
    {
15:   init {
16:     self.eOperations->
17:       select(op |
            op.oclIsKindOf(MappingOperation))->
18:         oclAsType(MappingOperation)->map mix();
//...
19: }

20: mapping MappingOperation::mix() : ASTNode {
21:     init {
22:         self.body.map mix(); //...
23: }
24: mapping MappingBody::mix() : ASTNode {
25:     init {
26:         self.initSection->map mix();
27:         self.content->map mix();
28:         self.endSection->map mix();//...
29: } //...
```

Figure 4. QvtMix Transformation

AST node shows how it is possible to programmatically alter the abstract syntax tree of the transformations. More specifically, the mapping rule checks whether the variable that is being initialized has the type of a collection (e.g., OrderedSet or List), and whether its binding time is dynamic. If so, it adds a dictionary as a global property to the subject transformation, corresponding with the name of the variable. This dictionary is used for book-keeping the indices of the statically evaluated and cached values. The reduce mapping is subsequently called to perform applicable reduction rule on the initExpression part (line 18). This transforms the expression to a residue expression that in lieu of computing the collection again, simply looks up the cached pre-computed values and merges them with the new, dynamic inputs.

Calls to select collection expressions are reduced in two ways depending on the type of their condition predicate. This distinction is made to facilitate the population and lookup of dictionaries used for static caching. For the predicates that select models based on constant literals or variables of primitive types such as String, the value of the indexer in the predicate is used as the key for the dictionary. On the

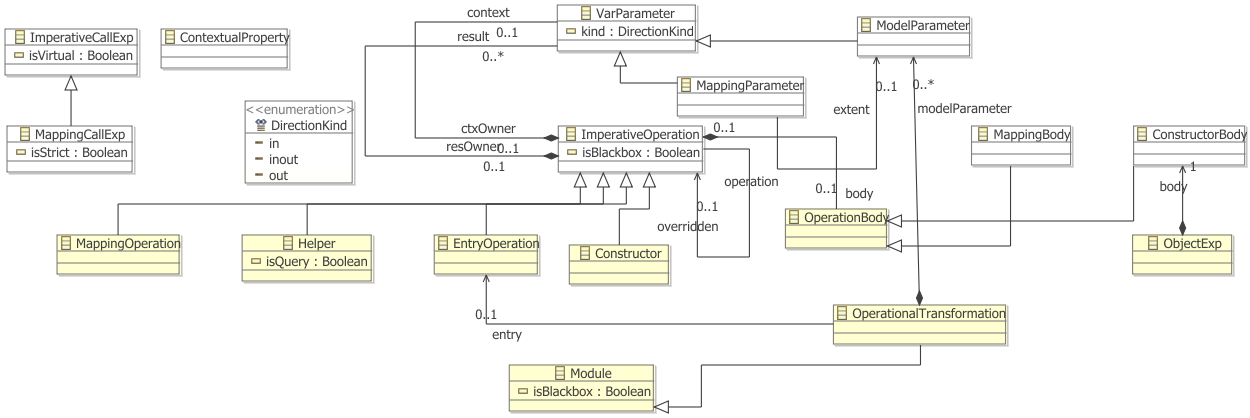

Figure 3. Simplified Metamodel of QVT Operational Mappings

```
01: mapping VariableInitExp::mix() : ASTNode {
02:   init {
03:         var expr := self.referredVariable.initExpression;
04:       var res := object VariableInitExp{};
05:       result := object VariableInitExp{};
06:   }
07:   population {
08:   if self.eType.oclIsKindOf(CollectionType) and
         self.bta().oclIsKindOf(DYNAMIC)        then
      {
09:      qvtSrc->eStructuralFeatures->append(
10:               object EAttribute {
11:                     name := '__' + self.name + '_inds';
12:                     eType := object DictionaryType {
13:                     name := 'Dict(Integer,Integer)';
14:                     keyType :=
                              object PrimitiveType {
                                    name := 'Integer'
                              };
15:                     elementType :=
                              object PrimitiveType {
                                    name := 'Integer'
                              }.oclAsType(EObject);
16:               }
17:      );
18:      res.referredVariable.initExpression :=
               self.referredVariable.initExpression.map reduce().
                     oclAsType(OCLExpression);
19:   }
20:   endif;
21:         //..
22:   }
23: }
```

Figure 5. VarInitExp mixer

```
01: modeltype BOOK uses 'http://book/1.0';
02: modeltype LIB  uses 'http://lib/1.0';
03: transformation Book2Lib(in bm : BOOK, out lm : LIB);
04: configuration property root : OrderedSet(Book);
05: main() {
06:  root := bm.rootObjects()[Root]
07:               ->selectOne(true).allBooks;
08:  root->location->map createLib();
09:}
10: mapping String::createLib() : Library {
11: init {
12:  var ownedBooks := root->select(b|b.location = self);
13: }
14: population {
15:    object result : Library {
16:          name := self;
17:          nbBooks := ownedBooks->size();
18:          pubs := ownedBooks->map Book2Pub();
19:    }
20: }
21:}
22: mapping Book::Book2Pub() : Publication {
23: init {
24:    var samebooks := root->xselect(b|b.name =
25:                        self.name)>asOrderedSet();
26: }
27: title := self.name + '_' +
28:          samebooks->indexOf(self).toString() +
29:          "_of_" + samebooks->size().toString();
30: type := PubKind::Book;
31: nbPages := self.chapters->nbPages->sum();
32:}
```

Figure 6. BooksToLibrary Transformation in QVT-OM

other hand, for predicates that depend on model elements, the position of elements are used for lookup. Due to this distinction, the generated residual code look different as one is direct addressing, while the other uses Compute expressions. They, nonetheless, have similar semantics. This case is exemplar for partially evaluating dynamic collections that can grow as a result of adding more elements. For the case of summation, the result of summation is cached along with the index of the last elements up to which the sum had been calculated. The residual code simply adds this value and the sum of the elements in the collection from the cached index to the end.

A. Illustrative Example

Figure 7 depicts two example metamodels. The left metamodel, BOOK, declares two classes: Chapter and Book where the instances of the latter comprise those of the former. The metamodel on the right denotes the domain of libraries with Publication and Library and the composition association between them. Figure 6 lists a transformation in QVT-OM that transforms an instance of the BOOK metamodel to an instance of the LIB metamodel. An instance of a Library is created for all the books that share the same

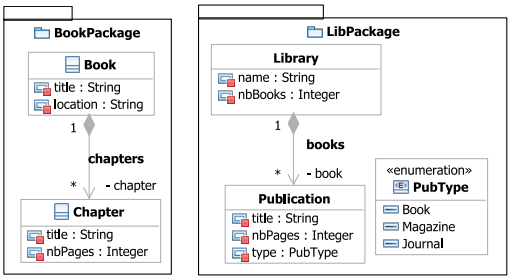

Figure 7. LIB and BOOK Metamodels

location values (defined in createLib, line 10). Each Book in the source model maps to a Publication whose type is marked as PubType::Book (defined in Book2Pub), and placed in an instance of Library that corresponds to its location. Line 17 calculates the number of books in each library by inquiring the size() of the collection ownedBooks, i.e., books that share the same location.

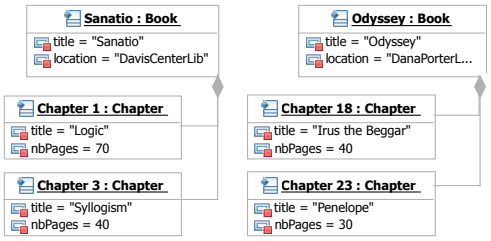

Figure 8. Instance of the Book Model

The two metamodels, albeit conceptually similar, have distinct semantic variations that have to be translated during transformation. In particular, attribute nbPages in the BOOK domain belongs to the Chapter class, whereas each instance of Publication in the LIB domain denotes its total number of constituting pages as an attribute. The transformation rule in line 31 of Figure 6 handles the translation between these two different semantics by aggregating the nbPages attribute of all the Chapter instances for each Book, into those of their corresponding Publication elements. Figure 8 illustrates an instance of the BOOK metamodel. It incorporates two books in two different locations each of which comprising two chapters. Figure 9 shows the result of the BooksToLibrary transformation applied on this instance of the BOOK metamodel.

Our choice of case-study and running scenario, albeit simple, is not simplistic, as it captures the essence of what makes the partial evaluation of imperative model transformations interesting. On the one hand, MOF and OCL (and likewise EMF) are designed to provide a generic notation for the specification of models. QVT transformations, insofar as operating on MOF or EMF compliant instances, are agnostic to the semantics of what is being specified, and as such,

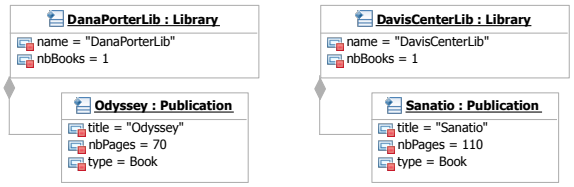

Figure 9. Result of the transformation

the difference between our pedagogical metamodels and a real-world metamodel can very well be only lexical (i.e., one can replace Book with JavaClass and Chapter with JavaMethod to obtain a representation of a very common UML model for Java, often used as a standard example in the research literature to showcase various model transformation technologies). On the other hand, the presented transformation exemplifies a particular class of model transformations, expression of which require an elaborate transformation language (e.g., QVT-OM). For example, the folding operation `chapters->nbPages->sum()` used in the example transformation is a *catamorphism*–i.e., a function that collapses a structure into a singular value, thereby loses information. Information losing transformations cannot be expressed in many commonplace transformation frameworks (e.g., most declarative ones) as they primarily denote *isomorphism* relations between the corresponding parts of source and target models. Making such transformations incremental using relational languages is straightforward. Another source of complexity in the presented transformation is engendering target objects according to an attribute value (i.e., the location attribute diffuses Library objects in the target).

B. Application of Partial Evaluation

The transformation presented in Section IV-A exemplifies some of the common characteristics of model transformations specified in such hybrid languages as QVT-OM. In this language, primarily due to its OCL heritage, several transformation rules operate on collections of model elements that are selected based on context-dependent criteria. The set of input elements in the source model is divided into a set of **FIXED** elements whose values and relationships are known and will not change, and variable elements, labeled as **VAR**. Such information about model elements are annotated in the source metamodel, as Figure 10 depicts.

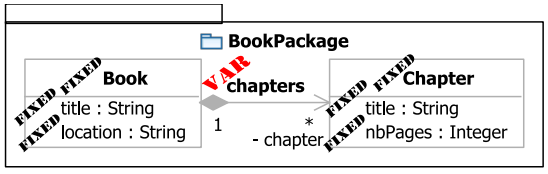

Figure 10. Annotations for static analysis

This particular annotation labels the Book class as **FIXED**, which indicates that neither any new instances of this class will be added, nor will any of the existing ones be removed from the input model. In contrast, the chapters composition link is labeled as **VAR** to declare that the future iterations of this model will have new instances of Chapter (For simplicity, we only consider addition here). All attributes of classes have **FIXED** annotations which means their values are invariant.

In the partial evaluation of BooksToLib, since Book is annotated with **FIXED** label, lines 12, 17, 24 and 27 of Figure 6 can be completely evaluated using static data and cached subsequently. The partial evaluator puts the specialized mapping rule depicted in Figure 11 in lieu of the original main. Static caches are declared as configuration properties so as to make them globally visible (line 01-07), and are populated inside the residual main mapping (line 10, 14 and 19). The one created for variable ownedbooks is indexed by string literals (since it is yielded by a select expression that depends on a string value). In contrast, the cache for samebooks is indexed by the position of each book in the root of the containment hierarchy of the source model.

```
01: configuration property root : OrderedSet(Book);
02: configuration property __samebooks_inds :
03:   OrderedSet(Tuple(i : Integer, l : OrderedSet(Integer)));
04: configuration property __ownedBooks_inds :
05:   OrderedSet(Tuple(s : String, l : OrderedSet(Integer)));
06: configuration property __nbPages_eval :
07:  OrderedSet(Tuple(ind:Integer,ev:Integer,last:Integer));
08:
09: main() {
10:  __ownedBooks_inds := OrderedSet {
11:     Tuple{s='DanaPorterLib', l=OrderedSet{1}},
12:     Tuple{s='DavisCenterLib', l=OrderedSet{2}}
13:  };
14:  __samebooks_inds := OrderedSet {
15:     Tuple{i=1, l=OrderedSet{1}},
16:     Tuple{i=2, l=OrderedSet{2}}
17:  };
18:
19:  __nbPages_eval := OrderedSet{
20:     Tuple{ind=1,ev=70,last=2},
21:     Tuple{ind=2,ev=110,last=2}
22:  };
23:  root := bm.rootObjects()[Root]->selectOne(r|true).allBooks;
24:  root->location->map createLib();
}
```

Figure 11. Partially Evaluated Main

The result of partial evaluation of the createLib mapping is presented in Figure 12. The computed expression (line 03) is generated for looking up the value of the select expression used on the right hand side of ownedBooks, as explained in the previous section. Here, there is another computation that can potentially be statically determined, namely nbBooks, originally evaluated in line 17 of Figure 6. Our framework however does not statically evaluate it, as it depends on a collection that is statically cached, namely, ownedBooks. Because our QVT implementation maintains the sizes of collections internally, it can readily be obtained at runtime.

Some of the operations are not entirely computable during partial evaluation. However, using proper semantics for collections, some segments of the collections upon which

```
01: mapping String::createLib() : Library {
02:    init {
03:       var ownedBooks := compute(bs : OrderedSet(Book)
04:          = OrderedSet{}) {
05:             __ownedBooks_inds->
06:              selectOne(t|t.s=self).l->forEach(i) {
07:                   bs := bs->append(root->at(i));
08:              }
09:       };
10:    }
11:    population {
12:       object result : Library {
13:           name := self;
14:           nbBooks := ownedBooks->size();
15:           pubs := ownedBooks->map Book2Pub();
16:       }
17:    }
18: }
```

Figure 12. Specialized createLib

such operations are performed can be partitioned into static and dynamic segments. Consequently, the computations that pertain to the static segment can be partially evaluated and aggregated with residual code that operates on the rest of the collection at runtime. An example of such operations is the calculation of Publication::nbPages by summing Chapter::nbPages for all the chapters of each individual book, as denoted in line 15 of Figure 6. The **VAR** annotation on the chapters association entails that the value of the Chapter::nbPages attribute depends on chapters that will be added later on. Nevertheless, by specializing this transformation for addition of input elements, the partial evaluator can infer the result for the existing chapters. This is shown in Figure 13.

```
01: mapping Book::Book2Pub() : Publication {
02:  init {
03:   var __self_index = root->indexOf(self);
04:   var __self_samebooks := __samebooks_inds->at(__self_index).l;
05:   var __nbPages_tup := __nbPages_eval->at(__self_index);
06:   var __nbPages_new := 0;
07:  }
08:  title := self.name + '_' +
09:         __self_samebooks->indexOf(__self_index).toString() +
10:         "_of_" + __self_samebooks->size().toString();
11:  type := PubKind::Book;
12:  var i := __nbPages_tup.last + 1;
13:  while (i <= self.chapters->size()) {
14:   __nbPages_new :=__nbPages_new+self.chapters->at(i).nbPages;
15:   i := i + 1;
16:  };
17:  nbPages := __nbPages_tup.ev + __nbPages_new;
18:}
```

Figure 13. Specialized Book2Pub

V. Experiments and Discussion

In this section, we present and discuss the results of experiments we have conducted utilizing the proposed partial evaluation framework. We have used the same transformation, i.e., BookToLibrary described in the previous section over a range of models of various sizes. Our experiments were performed on a Windows XP (Service pack 3) based PC featuring Intel Core™2 CPU clocked at 2.1Ghz, 2GB of physical memory, running Eclipse M2M project's QVT Operational Mapping implementation on top of Eclipse 3.5.

Our first set of experiments involved applying the transformation on models progressively growing in the number of elements, and thus in model size. Figure 14 illustrates how the performance of the original transformation compares to the one of its specialized version. The results indicate that the difference is negligible for small input models. This has to do with I/O being the dominant factor during the transformation of these models, which is comparable for both transformations. However, as we apply partial evaluation on larger models where the processing is the most time-consuming part and the I/O effect is amortized (i.e., models with more than 100 elements), the performance advantage of partial evaluation becomes noticeable. The detailed values of this experiment are reported in Table I.

The input models were generated by a QVT transformation. The reported elements are the ones for which the BookToLibrary transformation was partially evaluated. In this set of experiments, we consider the change to be the addition of just one chapter to the first book of the first library. The original transformation requires to transform all other non-affected elements, whereas the partially evaluated one exploits its static cache to expedite the reconciliation of the source and target models. Although at first this minimal size of change for empirical analysis can be called into question, it is in fact representative of a common practical scenario. Developers often have to manipulate bits and pieces of very large models, and no matter how small the change, the full re-transformation of these models may become untractable for practical purposes. In fact, this is the standard approach taken in most existing modeling tools. In this regard, partial evaluation provides a viable solution for such transformation scenarios.

Table I
THE RESULTS OF THE FIRST SET OF EXPERIMENTS.

N_T	N_L	N_B	N_C	T_{B2L}	T_{mixB2L}
3	1	1	1	0.12ms	0.11ms
12	1	1	10	0.12ms	0.11ms
111	1	10	10	0.27ms	0.25ms
1111	1	100	10	135.6ms	1.75ms
11011	1	1000	10	10602ms	47ms

N_T: total elements
N_L: no. of Lib
N_B: no. of Book per Lib
N_C: no. of Chapter per Book
T_{B2L}: Exec. time of BookToLibrary
T_{mixB2L}: Exec. time of mixBookToLibrary

In the second set of our conducted experiments, we applied a fixed input model to both transformations, each time instructing the partial evaluator to use a fraction of input for specializing the transformation. This is in effect as same as having the rest of the elements added on the second execution of the transformation. This experiment aims to assess whether specializing more expressions inside the program leads to better performance of the specialized program [1]. In particular, in this set of experiments we focused on the summation of nbPages in the program (line 31 and line 17 of Figure 6 and Figure 13, respectively) which was reduced by the specializer. The multiple-valued variables were completely cached for each invocation of the transformation. The results of these experiments are projected in Figure 15. We triggered the transformation with a model comprised of 10 libraries, 100 books and 10 chapters, the transformation of which took 10703ms[2] by the original BookToLibrary transformation. We then varied the percentage of input elements used as FIXED and calculated the time of execution of the transformation specialized for those model elements. As expected, the more input elements were being involved in the partial evaluation, the fewer dynamic computations was need to be performed during re-execution, and thus, the transformation took less time. We started from treating all VAR elements (i.e., instances of Chapter) as dynamic (they can be considered as new elements added after the initial transformation), and reduced this by 10% in each step, which resulted in more reduction of transformation time. The full partial evaluation performs more than twice as fast as having no partial evaluation.

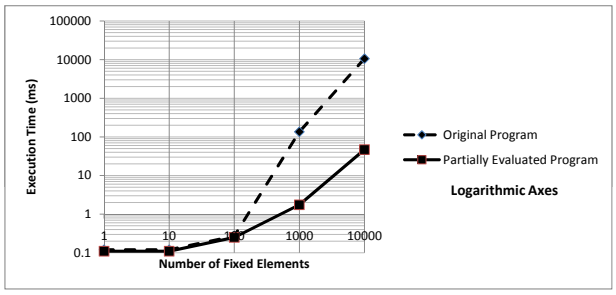

Figure 14. Execution time of the original and specialized transformation for growing input sizes

VI. CONCLUSION

In this paper, we discussed a methodology for leveraging the concept of partial evaluation for the purpose of increasing the performance of iterative and repeated applications of transformations, especially when these are applied on complex interdependent models. As a case study, we presented a partially evaluated residue of a QVT-OM transformation, and consequently demonstrated how transformation programs

[1]It should be observed that, although this proposition may appear intuitively obviously, it need not be generally true as for certain cases the overhead of parsing and interpreting residual expressions and static caches may outweigh any performance gained from avoiding the computation of expressions they replace. The experiment falsifies this possibility for the presented example

[2]The reason for being so slow is that line 12 in createLib executes in $O(N_B^2)$ in the original transformation

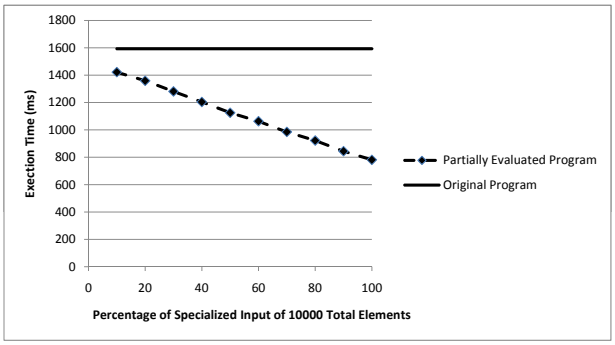

Figure 15. Execution time of the original and specialized transformation based on the utilization of input elements for partial evaluation

and computations can be simplified. This simplification of rules and programs by replacing expressions with pre-evaluated static data is, in effect, similar to transformations that are inherently incremental. Our experiments indicated a significant reduction in the re-transformation time as the percentage of utilized input model elements increases.

Our partial evaluator prototype requires the reification of OCL and QVT expressions at specialization time. This reification is not supported by the current implementations of QVT-OM, therefore substantial effort was exerted to bootstrap a QVT "self"-interpreter in QVT itself, so as to evaluate the intermediate results. The specialization part then uses this evaluator to transform the AST. The infrastructure needed for making this work (e.g., QVT self-interpretation or pretty printer) supports a comprehensive subset of the language (which has a copious grammar as it incorporates the entire OCL meta-model as well). Various specialization strategies for different AST nodes can be plugged into this core then. As far as partial evaluation is concerned, we only focused on what we thought was unique to QVT and was novel in are approach. More specifically, the static memoization of intermediate results of ImperativeIterateExpr and IterateExpr in the form of dictionaries which operate on collection-based constructs such as *xselect*, *select*, *collect* and various other set operations are handled. There exists a large class of well-known partial evaluation methodologies that are more specific to general purpose programming languages (e.g., constant propagation, call unfolding) that we skipped in our first implementation but plan to incorporate later. In summary, two major aspects that make QVT-OM unique in this respect is a) having collection operators as first class entities and b) being, in a sense, homoiconic (like the Lisp family of languages), that is, the ability to treat QVT transformations as input models. Our goal has been to showcase the combined power of these two. Thus, a natural extension would be broadening its scope of support for the rest of the language. This is not a trivial task though, as adding support for more constructs of the language requires to assure compatibility with the existing ones used for specialization. Our partial evaluation framework is both *for* and *in* the QVT Operational Mappings language. This is by design: so as to allow for self- applicability, which, according to Futamura projections [5], facilitates a multitude of potential applications which we intend to explore as future work.

This work was supported by IBM and Natural Sciences and Engineering Research Council of Canada and was conducted in collaboration with IBM Center for Advanced Studies (CAS) at the IBM Toronto Laboratory.

REFERENCES

[1] M. S. Ager, O. Danvy, and H. K. Rohde. Fast partial evaluation of pattern matching in strings. In *In Proceedings of PEPM'03*, pages 3–9, New York, NY, USA, 2003. ACM.

[2] ATL. *Specification of the ATL Virtual Machine version 0.1*. LINA and INRIA, Nantes, France, 2005.

[3] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 493–501, New York, NY, USA, 1993. ACM.

[4] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621 – 45, 2006/07/.

[5] Y. Futamura. Parital evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers and Controls*, 2(5):45–50, 1971.

[6] H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06)*, pages 543–557. Springer Verlag, 2006.

[7] J. Greenyer and E. Kindler. Reconciling TGGs with QVT. In *Model Driven Engineering Languages and Systems (MoDELS07)*, LNCS, pages 16–30. Springer Verlag, Nashville, TN, Oct. 2007.

[8] D. Hearnden, M. Lawley, and K. Raymond. Incremental model transformation for the evolution of model-driven systems. In *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06)*, volume 4199 NCS, pages 321 – 335, Genova, Italy, 2006.

[9] S. Johann and A. Egyed. Instant and incremental transformation of models. *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 362–365, Sept. 2004.

[10] N. D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503, 1996.

[11] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[12] J. L. Lawall. Faster fourier transforms via automatic program specialization. In *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*, pages 338–355, London, UK, 1999. Springer-Verlag.

[13] M. Lawley and K. Raymond. Implementing a practical declarative logic-based model transformation engine. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 971–977, New York, NY, USA, 2007. ACM.

[14] OMG. MOF QVT final adopted specification, 2008. Object Management Group document `ptc/07-07-08`.

[15] J. G. Park and M.-S. Park. Using indexed data structures for program specialization. In *In Proceedings of ASIA-PEPM '02*, pages 61–69, New York, NY, USA, 2002. ACM.

[16] A. Schürr. Specification of graph translators with triple graph grammars. In *20th Internation Workshop of Graph Theoretic Concepts in Computer Science, WG'94*, volume 903 of *LNCS*, pages 151–163, Herrsching, Germany, June 1994.

[17] R. S. Sundaresh. Building incremental programs using partial evaluation. In *In Proceedings of PEPM'91*, pages 83–93, New York, NY, USA, 1991. ACM.

[18] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 164–173, New York, NY, USA, 2007. ACM.