A Graph Query Language for EMF models[⋆]

Gábor Bergmann, Zoltán Ujhelyi, István Ráth, Dániel Varró

Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
1117 Budapest, Magyar tudósok krt. 2
{bergmann,ujhelyiz,rath,varro}@mit.bme.hu

Abstract. While model queries are important components in model-driven tool chains, they are still frequently implemented using traditional programming languages, despite the availability of model query languages due to performance and expressiveness issues. In the current paper, we propose EMF-IncQuery as a novel, graph-based query language for EMF models by adapting the query language of the Viatra2 model transformation framework to inherit its concise, declarative nature, but to properly tailor the new query language to the modeling specificities of EMF. The EMF-IncQuery language includes (i) structural restrictions for queries imposed by EMF models, (ii) syntactic sugar and notational shorthand in queries, (iii) true semantic extensions which introduce new query features, and (iv) a constraint-based static type checking method to detect violations of EMF-specific type inference rules.

1 Introduction

Model queries are important components in model-driven tool chains. They are widely used in model transformations, model execution/simulation, report generation, or the evaluation of well-formedness constraints. *Global model queries* can be evaluated over the entire model to retrieve all results fulfilling the query, while *local model queries* retrieve information specific for some given input model elements. In current industrial applications based on popular modeling frameworks (e.g. the Eclipse Modeling Framework *EMF*[1]), model queries are still frequently implemented using a traditional programming language (Java), despite the availability of more advanced declarative query languages such as OCL [2] or EMF Model Query [3].

The reasons for this are two-fold. Unfortunately, as observed in tool development practice, as well as in benchmark measurements [4], the implementation infrastructure behind these high level model query languages often has *scalability issues* when large instance models are used, which may effectively rule out the application of these technologies in certain industrial applications.

Additional issues include *expressiveness and learning effort.* Simple technologies (such as EMF Model Query) are not flexible or expressive enough for advanced use cases involving complex join operations, while complex – and thus

⋆ This work was partially supported by the SecureChange (ICT-FET-231101) European Research Project.

significantly harder to learn [5] – languages such as OCL still lack important features despite their higher expressive power. Such important features include reusable query modularization, recursion and transitive closures, which are not easily accessible or not supported at all. Finally, there is also a practical need for the adaptation and extension of compile-time validation techniques, which are currently in very early stages of development.

In [4], we demonstrated how incremental model transformation techniques of the VIATRA2 framework can be adapted to efficiently support advanced model queries over large EMF models and proposed a new runtime model query framework called EMF-INCQUERY. Our initial investigations were focused on providing high performance for model queries, therefore, we reused the query language of VIATRA2. However, as the underlying (meta-)modeling foundations for VIATRA2 and EMF are different, the direct reuse of the VIATRA2 graph pattern language raised usability issues.

In the current paper, we present the query language of EMF-INCQUERY, which provides an EMF-specific dialect of the graph pattern language of the VIATRA2 transformation framework. This query language – having its roots in the graph transformation domain – shares proven concepts from languages of existing and very powerful tools (e.g. Fujaba, PROGRES, GrGEN, GReAT, VMTS, AGG) and is intended to integrate to the industry-standard EMF platform to reach a broader audience. By adapting the query language of VIATRA2, our goal is (1) to enable that EMF-INCQUERY inherits the declarative nature, conciseness, easy specification and comprehension of the VIATRA2 language, and (2) to ensure that the new query language is properly tailored to the modeling specificities of EMF.

In the paper, we report about this adaptation, which includes (i) structural restrictions for queries imposed by EMF models (e.g. lack of edge variables), (ii) syntactic sugar and notational shorthand in queries (like transitive closure along edges, simplified attribute conditions) that support more convenient query specification, and (iii) true semantic extensions to the existing VIATRA2 pattern language, which introduce new query features (like aggregation, indexing in ordered collections, arithmetic assignments). Finally, (iv) a constraint-based static type checking method is used to detect violations of EMF-specific type inference rules in the EMF-INCQUERY language. Compared to [4] (which introduced the runtime incremental matching framework for EMF-INCQUERY together with performance benchmarks), the current paper focuses exclusively on the EMF-specific query language itself.

The rest of the paper is structured as follows. In Sec. 2, we introduce a motivating case study and the existing pattern language of VIATRA2. Sec. 3 describes the novel language features of EMF-INCQUERY by elaborating the case study. We present the static type checking feature in Sec. 4 and discuss related work in Sec. 5. Sec. 6 concludes the paper with future research directions.

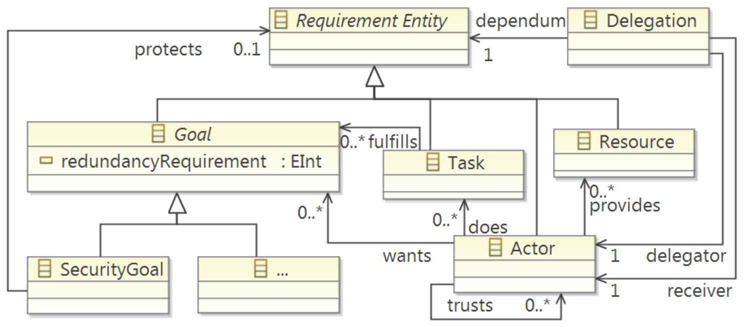

Fig. 1. Simplified security requirements metamodel

2 Background and Case Study

2.1 Case Study

Problem Domain. Our motivating scenario is from the domain of security requirements engineering in case of evolving models, inspired by the Air Traffic Management case study of the SecureChange European FP7 project. A requirements model assists security engineers to capture security related aspects of the system, to analyze the security needs of the stakeholders, and to argue about potential security threats. The concepts of a security requirements modeling language such as SecureTropos [6] typically include *actors*, *resources* (e.g. security-critical information assets) provided by actors, *goals* (functional, security, etc. requirements) wanted by actors, and *tasks* performed by actors. Relationships include tasks to fulfilling goals; trust relationships between actors; and delegation of responsibility over resources, goals or tasks. See Fig. 1 for the most important elements of the SecureChange requirements metamodel.

An important role of security requirement models is to support reasoning about security properties by formal or informal argumentation techniques [7] in an early phase of system development. To formalize static security properties, we use graph patterns as a query language in the scope of the project. On the tooling level, the EMF-IncQuery framework provides efficient, incremental constraint evaluation and feedback for the engineers already in the early stages of requirements modeling.

Analysis Tasks. Early-stage analysis of requirements models is carried out by (local and global) model queries in the case study of this paper. Support can range from finding violations of structural semantic constraints that represent security properties of the model, to generating reports that guide the engineer to fix these problems.

One challenge where early-stage analysis is beneficial is detecting violations of the *trusted path* property. The context is the following: a valuable data asset is provided by one actor, and is eventually delivered to a recipient actor, through potentially unreliable intermediate actors. A security goal requires the protection of the integrity and confidentiality of this data resource. The trusted path property states that either a trusted actor has to perform an action that explicitly fulfills the goal (e.g. time-stamping, digital signature and encryption), or else the entire data path must be trusted; indirect trust is permitted. The challenge is to formulate a query which finds the violations of this security property.

A second application of model queries is related to the *redundancy* property. Redundancy is important for resilience against failures and attacks, and is therefore an integral part of security; thus requirements often have a minimal degree of redundancy associated with them. For example, the availability requirement of a service task or data asset can be augmented with the demand of triple modular redundancy, i.e. 3 replicas of data / service must be available. A goal with the `redundancyRequirement` attribute set must be fulfilled by at least this number of separate tasks (performed by trusted actors). We will formulate two queries associated with this property: (a) one to find goals whose redundancy requirement is not met, and (b) a second one that computes an actor-centric progress indicator that informs of the total number of missing replicas for all the goals wanted by a given actor.

2.2 Case Study Solution Using the Original VIATRA2 Language

VPM Models. VPM [8], the model representation of the VIATRA2 transformation framework [9], has a very generic graph structure, similarly to the concept of clabjects [10] and ontologies [11]. A VPM model is a containment hierarchy of entities (nodes) with interconnecting relations (edges), and some special relationships such as instantiation. Models and metamodels are represented uniformly in a very flexible multi-typed, multi-level metamodeling paradigm. Nodes and edges have likewise an identity of their own, with a locally unique name, and possibly attributes. A model entity, however, can only store a single (unnamed) value, and therefore multiple attributes are represented by separate relation types and local *wrapper* model entities.

Graph Patterns and Model Queries. The VTCL language [9] was originally designed to support model transformation over VIATRA2's model representation VPM. In particular, it offers graph patterns as a mechanism for querying VPM models; a graph pattern is basically a typed graph-like structure which is matched against a large model graph. The VTCL language defines graph patterns by specifying *graph pattern constraints* over *pattern variables*. The pattern variables here are the nodes and edges of the graph pattern, some of which can be made externally accessible as *symbolic parameters* of the pattern. The constraints assert the graph structure and types of the pattern (as well as some other properties). A *match* of the pattern is a mapping of variables to VPM entities and relations so that all constraints are satisfied (analogously to a morphism into the model graph).

Graph patterns support parametrizable queries by evaluating the entire match set of a pattern globally in the model, or by binding one or more pattern parameters as input elements and only retrieving the local matches of the pattern.

Listing 1 Violations of the trusted path property, original syntax

```
1 shareable pattern noTrustedPath(ConcernedActor,SecGoal,Asset,UntrustedActor)={
2   Actor.wants(WantsEdge,ConcernedActor,SecGoal);
3   SecurityGoal(SecGoal);
4   SecurityGoal.protects(ProtectsEdge,SecGoal, Asset);
5   Actor.provides(ProvidesEdge,ProviderActor,Asset);
6   find transitiveDelegation(ProviderActor,UntrustedActor,Asset);
7   neg find transitiveTrust(ConcernedActor,UntrustedActor);
8   neg find trustedFulfillment(ConcernedActor,AnyActor,AnyTask,SecGoal);
9 }
```

We now present a solution using graph patterns in the VTCL language to address the trusted path property, as the redundancy property heavily relies on arithmetic computations that are not expressible in the original VTCL language.

Basic Pattern Elements. Pattern `noTrustedPath` in Lst. 1 captures the violations of the trusted path security property using the original VTCL syntax. The symbolic parameters of the pattern are `ConcernedActor`, `SecGoal`, `Asset`, `UntrustedActor`; there are also local variables `ProtectsEdge`, `ProviderActor`, etc. The pattern constraints include a *node constraint* on line 3 that asserts that variable `SecGoal` must be mapped to a node of type `SecurityGoal`. Lines 2,4 and 5 are *edge constraints*; e.g. the first of them asserts that the variable `WantsEdge` be an edge of type `Actor.wants` (i.e. the `wants` reference of class `Actor`), leading from `ConcernedActor` to `SecGoal`. As demonstrated by line 5, edges can be navigated in both directions.

To conform to a typical engineer's intuition, patterns are normally *injective*, i.e. object variables within a pattern will be matched to different model elements; unless the pattern is declared `shareable`, when two pattern variables may store / share the same model element, or explicit variable assignment (see Line 6) is used. Here the `shareable` keyword is used in the definition of `noTrustedPath`, as the pattern must allow the special case where `ConcernedActor` and `ProviderActor` are the same.

Pattern Composition. An important type of pattern constraint is pattern composition or *pattern call*, denoted by the `find` keyword. A pattern call reuses a *called pattern* inside the body of the *calling pattern* (possibly recursively). Line 6 of Lst. 1 provides an example that asserts that the tuple (`ProviderActor`, `UntrustedActor`, `Asset`) must be a match of a pattern called `transitiveDelegation`.

The three patterns called from `noTrustedPath` are defined in Lst. 2. The *disjunctive* pattern `trustedFulfillment` finds trusted actors that fulfill a given goal. The *recursive* pattern `transitiveTrust` captures the transitive closure of the `Actor.trusts` edges. Pattern `transitiveDelegation` is also recursive, in a more complex way.

Negation. Node and edge constraints, as well as pattern calls can be prefixed by the `neg` keyword to express negation, resulting in a *negative application condition* (NAC). A match of the enclosing pattern is considered invalid if the NAC is satisfiable. Lines 7 and 8 of Lst. 1 provide examples: the pattern `noTrustedPath` matches only if there is no transitive trust between `ConcernedActor`

Listing 2 Trusted path helper patterns

```
pattern trustedFulfillment(TrustingActor,FulfillerActor,Task,Goal)={
  find actorFulfillsGoal(FulfillerActor,Task,Goal);
  find transitiveTrust(TrustingActor,FulfillerActor);
} or {
  find actorFulfillsGoal(FulfillerActor,Task,Goal);
  TrustingActor = FulfillerActor; // explicit variable assignment
}
pattern transitiveTrust(TrustingActor,Trustee)={
  Actor.trusts(TrustEdge,TrustingActor,Trustee);
} or {
  find transitiveTrust(TrustingActor,MiddleMan);
  Actor.trusts(TrustEdge,MiddleMan,Trustee);
}
pattern transitiveDelegation(Delegator,Receiver,Dependum)={
  find directDelegation(Delegator,Receiver,Dependum);
} or {
  find transitiveDelegation(Delegator,MiddleMan,Dependum);
  find directDelegation(MiddleMan,Receiver,Dependum);
} or {
  find transitiveDelegation(Delegator,Receiver,SuperDependum);
  find decomposeDirect(SuperDependum,Dependum);
}
```

and `UntrustedActor`; and there is no such `AnyActor` and `AnyTask` that `ConcernedActor`, `AnyActor`, `AnyTask`, `SecGoal` would form a match of pattern `trustedFulfillment`.

NACs do not *define* new variables in their header arguments, they are either *input or quantified*. Input variables of a NAC are those arguments that are defined somewhere else (e.g. at a positive edge or node constraint); the rest of the variables are non-existentially quantified, and are not allowed to be referenced anywhere else. The NAC states that no substitution of the quantified variables can satisfy a match of the NAC, given the value of the input variables. In the previous example, `AnyActor` and `AnyTask` were quantified variables, and the other two were obtained as input argument of the NAC.

3 The Language of EMF-IncQuery

EMF-IncQuery is a framework with a *language for defining declarative local and global queries over EMF models*, and a *runtime engine for executing them efficiently without manual coding*. The query language of EMF-IncQuery reuses the concepts of graph patterns Viatra2 as a concise and easy way to specify complex structural model queries. However, while in [4], we simply restricted the VTCL language, this paper provides a more systematic design of a graph query language for EMF models to provide high level of expressiveness but also to overcome language usability issues we experienced in [4].

After a brief introduction to EMF, we present the syntax of EMF-IncQuery step-by-step. The new language introduces some significant semantic extensions, as well as syntactic sugar for conciseness. The two main areas where EMF-IncQuery differs from the original VTCL syntax are the structural/navigational language elements and the handling of attributes and arithmetic expressions.

Listing 3 Violations of the trusted path property, EMF-specific syntax

```
1 shareable pattern noTrustedPath(ConcernedActor,SecGoal,Asset,UntrustedActor)={
2   Actor.wants(ConcernedActor,SecGoal);
3   SecurityGoal(SecGoal);
4   SecurityGoal.protects(SecGoal, Asset);
5   Actor.provides(ProviderActor,Asset);
6   find transitiveDelegation(ProviderActor,UntrustedActor,Asset);
7   neg Actor.trust*(ConcernedActor,UntrustedActor);
8   neg find trustedFulfillment(ConcernedActor,AnyActor,AnyTask,SecGoal);
9 }
```

3.1 Background Technology: the Eclipse Modeling Framework

The *EMF* ecosystem provides automated code generation and tooling (e.g. notification, editor) for model representation in Java. EMF models consist of a containment hierarchy of model elements (*EObjects*) with cross-references – some of which may only be traversed by programs in one direction (unidirectional references). Objects also have a number of attributes (primitive data values).

EMF uses *Ecore* metamodels to describe the abstract syntax of a modeling language. The main elements of Ecore are *EClass* (graphically depicted as a box), *EReference* (depicted as an edge between boxes), and *EAttribute* (depicted within the middle compartment of a box). EClasses define the types of EObjects, enumerating EAttributes to specify attribute types of class instances and EReferences to define association types to other EObjects. EReferences and EAttributes can be multi-valued and ordered. Some EReferences can additionally imply containment. Inheritance may be defined between classes (depicted by an arrow ending in a hollow triangle), which means that the inherited class has all the properties its parent has, and its instances are also instances of the ancestor class, but it may further define some extra features. Each instance EObject has exactly one EClass type, and the metamodel is well-separated from instance models. The ECore diagram of the casestudy metamodel is depicted in Fig. 1.

3.2 Structural Constraints

We now demonstrate the structural pattern constraints of the EMF-IncQuery language using the trusted path property, captured in Lst. 3. The graph pattern based query language of EMF-IncQuery references EClasses as node types, EReferences and EAttributes as edge types. Pattern variables will be mapped to EObjects of the instance model or attribute values. For example in Lst. 3, line 3 seeks for an EObject of type `SecurityGoal` and stores the corresponding element in variable `SecGoal`. Line 2 navigates from the `ConcernedActor` (also appearing as a symbolic parameter) along an EReference of type `wants`, and the EObject reached that way should be the one stored by variable `SecGoal`.

Two major limitations of the core EMF API are the lack of (i) efficient enumeration of all instances of a class regardless of location, and (ii) backwards navigation along uni-directional references. As seen here, the structural graph constraints of EMF-IncQuery can provide these missing features.

Listing 4 Helper pattern dependent on EMF edge ordering

```
pattern directDelegation(Delegator,Receiver,Dependum)={
  Delegation(Delegation);
  Delegation.elements[0](Delegation,Delegator);
  Delegation.elements[1](Delegation,Receiver);
  Delegation.elements[2](Delegation,Dependum);
}
```

Binary Edge Constraints. Edge constraints in EMF-IncQuery (lines 2,4,5 in Lst. 3) look more simple than in Viatra2; *binary edge constraints* only use the variables representing the source and the target of the edge, and edge variables are altogether omitted from the pattern. This language design choice was made to reflect that EMF, does not assign edges an identity of their own. Instance model edges are characterized only by their source object, their EReference type (defined or inherited by the EClass of the source object) and the target object, but the reference itself does not have a corresponding EObject on instance-level.

Transitive Closure. The most frequent use case of recursion in queries is to capture *transitive closure*. For more convenient definition of queries, a concise syntax is proposed for the transitive closure of an edge type: by postfixing the type name by an asterisk (`*`), its transitive closure can be used without defining a recursive pattern for it. Similarly, the closure of a binary pattern can be defined in the same way, by putting an asterisk between the pattern name and arguments in a `find` clause. Such a binary pattern emulates a pseudo-edge of the graph, while encapsulating an arbitrarily complex relationship between its pseudo-source and pseudo-target. Line 7 of Lst. 3 applies the transitive closure operator to the edge type `Actor.trust`, eliminating the need for the separate pattern `transitiveTrust` (given that `trustedFulfillment` is also modified this way). However, as `transitiveDelegation` captures a more complex recursive relationship, it cannot be expressed using this shorthand.

Accessing Ordered Edges by Index. The `directDelegation` pattern called by `transitiveDelegation` hides a further layer of complexity. Delegation is a ternary relation; the metamodel includes a separate `Delegation` class with references to the delegator, the receiver and the dependum. Although Fig. 1 shows these as three distinct EReferences, in practice often there is only a single ordered EReference, and the three instance-level links are distinguished according to their order in the list. Such a solution may not be elegant, but it is still frequent (industrial) practice (e.g. with a metamodel generated from a grammar).

For this purpose, EMF-IncQuery offers an optional *ordering index* qualifier to capture the position of the edge within the ordered collection at the source object. Pattern `directDelegation` in Lst. 4 takes advantage of this feature and binds the index to the constant values 0, 1, and 2, respectively to capture a direct delegation relation.

Listing 5 Violations of the redundancy property

```
1  pattern redundancyViolated(CA,RG)={
2    find redundantReplicas(CA,RG,DegreeOfRedundancy,RequiredRedundancy);
3  //  check (toInteger(value(DegreeOfRedundancy)) <
4  //     toInteger(value(RequiredRedundancy))); -- Old VTCL syntax
5    check (DegreeOfRedundancy < RequiredRedundancy);
6  }
7  pattern redundantReplicas(CA,RG,DegreeOfRedundancy,RequiredRedundancy)={
8    Actor.wants(CA,RG);
9    Goal.redundancyRequirement(RG,RequiredRedundancy);
10   let DegreeOfRedundancy = count with find
11     trustedFulfillment(CA,AnyFulfillerActor,AnyTask,RG);
12 }
```

3.3 Attribute and Arithmetic Constraints

We now demonstrate the attribute and arithmetic constraints of EMF-IncQuery by formalizing the redundancy property of the case study. As defined in Fig. 2.1, the redundancy property states that a goal `RG` with a `redundancyRequirement` attribute must be fulfilled at least as many times (by trusted actors) as specified by the attribute value. The pattern `redundancyViolated` described in Lst. 5 identifies violations of this property, using a helper pattern. The secondary challenge is to provide an actor-centric indicator report on the number of missing replicas; the solution is shown in Lst. 6. In the sequel, we will gradually explain these patterns when introducing the new language features.

Scalar Variables and EAttribute Edges. EMF attribute values are stored directly within an EObject, as a separate member variable for each attribute type defined or inherited by the EClass. Therefore edge constraints representing EMF EAttributes immediately point to the raw data value. In case of asserting the equality of two attributes, the attribute edge constraints may simply share the same target variable. Such raw data values that are not model elements (EObjects) will be referred to as *scalar variables*, while variables that will be substituted with EObjects are *object variables*, which distinction was not originally part of VTCL. See line 7 of Lst. 5 as an example edge constraint extracting an attribute value from an EObject; `RequiredRedundancy` will therefore be a scalar variable (as well as `DegreeOfRedundancy`, `MR` and `TMR`).

VTCL includes a special type of pattern constraint denoted by `check()` that checks *arithmetic conditions* based on attribute values; see line 3 in Lst. 5 or line 9 in Lst. 6 for example usage.

Scalar variables are not subject to injectivity checks and can be mapped to the same value by default. This semantic distinction of scalar and object variables introduces additional static type inference challenges as well, which will be discussed later in Sec. 4.

Arithmetic Evaluation and Assignment. As scalar variables were not available previously, the pattern language of VTCL only allowed arithmetic expressions within `check()` pattern constraints. With the introduction of the class of scalar patterns, a straightforward semantic extension is the introduction of *arithmetic expression evaluation* constraints. An `eval` constraint evaluates an

Listing 6 Missing replicas per actor and goal

```
 1 pattern totalMissingReplicas(CA,TMR)={
 2   Actor(CA);
 3   let TMR = sum(MR) with find
 4     missingReplicas(CA,AnyGoal,MR);
 5 }
 6 pattern missingReplicas(CA,RG,MR)={
 7   find redundantReplicas(CA,RG,DegreeOfRedundancy,RequiredRedundancy);
 8   let MR = eval(RequiredRedundancy - DegreeOfRedundancy);
 9   check (MR > 0);
10 }
```

arithmetic expression based on referenced scalar variables, and assigns the result to a scalar variable. Line 8 in Lst. 6 provides an example, where the difference between two integer scalar variables is assigned to variable `MR`. The result scalar variable can be used just as freely as an attribute value, e.g. it can appear as a symbolic parameter to be used externally. Note however that EMF-IncQuery is not an equation solver, so circular references in expressions are disallowed.

Aggregation. The language EMF-IncQuery also includes an *aggregation* pattern constraint similar to e.g. aggregation functions of SQL. An aggregation is a pattern call that aggregates (counts, sums, etc.) matches of the called pattern with some given input parameters. More precisely, except for match counting, an arithmetic expression based on the match of the called pattern is aggregated over the matches. The resulting aggregate value is either specified in the calling pattern as a numeric constant, or captured in a scalar variable. For example, lines 8-9 in Lst. 5 count the number of trusted fulfillments of the goal, and store the result in `DegreeOfRedundancy`; likewise lines 3-4 in Lst. 6 assign `TMR` to the sum of the computed `MR` values for each goal of the actor.

The following incrementally maintainable aggregator functions are supported:

- `count` that returns the number of matches of the called pattern (with the given values of its referenced variables),
- `sum(expr)` that computes the sum of the evaluations of expression `expr` over each match,
- `avg(expr)` that returns the average of `expr`,
- `min(expr)` and `max(expr)` that respectively return the minimal and maximal value among evaluations of `expr` over the matches.

If the called pattern has no matches, `count` and `sum(expr)` return a default value of 0, while the other aggregation constraints will not be satisfied (i.e. the enclosing pattern will fail to match).

Similarly to NACs, aggregations do not *define* the variables in their arguments (except for the aggregate result), they are either input or quantified. It is worth noticing that NAC is a special case of match counting, with the aggregate value bound to 0; and match counting itself is a special case of summation, where a constant 1 is being summed over the matches.

Intent	Feature	EMF	VPM	Origin
enumeration	node constraint	Yes	Yes	Originally available
navigation	edge constraints	Binary	Ternary	EMF adaptation
	edge indices	Yes	No	EMF adaptation
	graph structure	Yes	Yes	Originally available
	recursion	Yes	Yes	Originally available
	transitive closure	Yes	Yes	Syntactic sugar
filter	negative application condition	Yes	Yes	Originally available
	attribute checks	Scalars	Wrappers	EMF adaptation
computation	arithmetic evaluation	Scalars	Wrappers	Semantic Extension
	counting and aggregation	Yes	Yes	Semantic Extension
reuse	pattern composition	Yes	Yes	Originally available

Table 1. Language features

The preceding discussion of the case study solution presented some aspects of the language design of EMF-IncQuery. Table 1 shows an overview of some important language features.

4 Static Type Checking for the Query Language

As the EMF-IncQuery query language uses complex structures (such as path expressions or transitive closures), it is possible to write erroneous queries that may lead to unexpected runtime behavior or exceptions. These mistakes can be detected by using static analysis techniques without the costly execution of the transformation. In the current work, we focus on *type errors* (e.g. using a pattern variable with incompatible types) that are hard to detect manually, because they may not cause runtime errors during execution, but rather result in an empty match set being returned. In our experience, such mistakes are very common (e.g. calling a pattern with invalid parameters, or parameters in an invalid order).

For example, the `CA` and `AnyGoal` parameters of the `missingReplicas` pattern call are switched in Line 5 in Lst. 7. In this case, the pattern is called with its `RG` parameter bound to a variable with the `Actor` type, and as a result, the pattern will never match (and the variable `TMR` will always contain the integer scalar 0).

To detect such problems, we propose a constraint-based static type checking framework for graph patterns, adapting a type checking approach for partially typed graph transformation programs [12]. For pattern constraints (see Table 1) expressing graph structure (e.g. in Line 2), the type information that has to be checked for consistency is always available. However, in the case of pattern composition, as pattern parameters are dynamically typed, *pattern parameter type inference* is necessary (see the call of `missingReplicas` pattern in Line 5). We encode type inference as constraint satisfaction problems and apply a constraint solver to propagate the available type information to calls where type inference

Listing 7 Erroneous version of missing replica counter pattern

```
 1 pattern totalMissingReplicas(CA, TMR)={
 2   Actor(CA);
 3   //Error: CA and AnyGoal parameters switched during pattern composition
 4   let TMR = sum(MR) with find
 5     missingReplicas(AnyGoal,CA,MR);
 6 }
 7 pattern missingReplicas(CA,RG,MR)={
 8   find redundantReplicas(CA,RG,DegreeOfRedundancy,RequiredRedundancy);
 9   let MR = eval(RequiredRedundancy - DegreeOfRedundancy);
10   check (MR > 0);
11 }
```

Line	Type Constraints	Comments
Line 8	$\mathit{typeOf}(CA) = \mathsf{Actor}$	Inferable from pattern call, not detailed here
	$\mathit{typeOf}(RG) = \mathsf{Goal}$	
	$\mathit{typeOf}(DegreeOfRedundancy) = \mathsf{int}$	
	$\mathit{typeOf}(RequiredRedundacy) = \mathsf{int}$	
Line 9	$\mathit{typeOf}(MR) = \mathsf{int}$	`eval` with integer operation
	$\mathit{typeOf}(RequiredR) = \mathsf{int}$	
	$\mathit{typeOf}(DegreeOfR) = \mathsf{int}$	
Line 10	$\mathit{typeOf}(MR) \in \{\mathsf{int}, \mathsf{double}\}$	Number comparison

Table 2. Type inference in the missing replicas pattern

is needed. To improve performance, the type constraints are generated and evaluated for each pattern separately (by the generating type contracts), and these partial results are combined to generate the type constraints for pattern calls.

To give an overview of the analysis framework, we demonstrate its capabilities using the erroneous patterns in Lst. 7.

The type constraints generated from the *missing replicas pattern* in Table 2. The first column selects a line from the pattern, the second describes the generated type constraints, while the third shows related comments. Aggregating the type constraints shows no contradiction and the following result is calculated for the pattern parameters: $\mathit{typeOf}(CA) = \mathsf{Actor} \wedge \mathit{typeOf}(RG) = \mathsf{Goal} \wedge \mathit{typeOf}(MR) = \mathsf{int}$.

The generated type constraints for the *total missing replicas pattern* are listed in Table 3. However, the aggregation detects contradicting constraints for the variable `CA`, as there is no type that is both an `Actor` and `Goal`. As a result, an error is detected and reported to the developer.

5 Related Work

Model Queries over EMF. OCL [13] is a navigation-based query language, applicable over a range of modeling formalisms. OCL is more expressive in certain cases than EMF-IncQuery, considering e.g. the `iterate` construct. On the other hand it lacks query compositionality (helper operations can work around

Line	Type Constraints	Comments
Line 2	$typeOf(CA) = \mathsf{Actor}$	Pattern condition
Line 4	$typeOf(TMR) \in \{\mathsf{int}, \mathsf{double}\}$ $typeOf(MR) \in \{\mathsf{int}, \mathsf{double}\}$ $typeOf(TMR) = typeOf(MR)$	Scalar variables are summed
Line 5	$AnyGoal \to CA \Rightarrow typeOf(AnyGoal) = \mathsf{Actor}$ $CA \to RG \Rightarrow typeOf(CA) = \mathsf{Goal}$ $MR \to MR \Rightarrow typeOf(MR) = \mathsf{int}$	Variable assignments in pattern composition

Table 3. Type inference in the total missing replicas pattern

Intent	Feature	Model Query [3]	Xpand[15]	EOL [14]	MDT-OCL [2]
enumeration	node constraint	Yes	Yes	Yes	Yes
navigation	edge constraints	Yes	Yes	Yes	Yes
	edge indices	No	Yes	Yes	Yes
	graph structure	Tree only	Yes	Yes	Yes
	recursion	No	No	Well-founded	Well-founded
	transitive closure	No	No	No	Non-standard
filter	NAC	Yes	Yes	Yes	Yes
	attribute checks	Single node only	Yes	Yes	Yes
computation	arithmetic evaluation	No	Yes	Yes	Yes
	counting and aggregation	No	Yes	Yes	Yes
reuse	pattern composition	Yes	No	Operation	Operation

Table 4. Comparison of query language features

this); only well-founded recursive queries are supported this way (e.g. transitive closure of non-DAG graphs, such as the network of trust between actors, is not expressible); and the language is arguably less declarative than that of graph patterns. A precise comparison of expressivity is left as non-trivial future work.

There are several technologies that support model querying over EMF, see Table 4 for a comparison showing whether features of EMF-IncQuery can be replicated by a given tool. The Epsilon Object Language (EOL) [14], disregarding its Javascript-like imperative features, can be considered very similar to OCL. M2T Xpand's Expressions [15] is also roughly equivalent to OCL, but it does not contain any method of reuse. MDT-OCL [2] is the canonical OCL implementation for EMF. While general recursion is still not supported, the `closure` construct is provided as a non-standard extension to OCL, which is essentially a least fix point operator capable of expressing certain recursive queries such as transitive closure. *Model Query* [3] has significantly lower expressivity than EMF-IncQuery or any of the above: it cannot capture graph-like (circular) relationships, or compare attribute values; however, it can be extended by OCL.

There are also several tools [16,17,18] that adapt graph transformation concepts to EMF, although for model transformation, not as a query language. These approaches do not include any of the rich language features of EMF-IncQuery such as composition, recursion, aggregation or edge indices. Furthermore, there is a wide range of existing graph transformation tools (e.g. Fujaba, PROGRES, GrGEN, GReAT, VMTS, AGG) which offer some of the advanced features of EMF-IncQuery without supporting queries directly over EMF models.

Although not aimed at model-driven purposes, SPARQL [19] is an important query language. Comparison and benchmarking is planned for a future paper.

Incrementality. From a performance viewpoint, *incremental query evaluation* has a significant impact on the scalability of technologies that build on queries (model transformations, well-formedness validators, simulators etc.). In [4], we demonstrated that the supporting infrastructure of EMF-IncQuery scales up to provide instantaneous results for queries over large models with millions of model elements. Related work on other incremental evaluators (for OCL and other query/transformation languages) is also discussed in detail in [4].

Language Specialization. An important challenge of the current paper was to adapt a general-purpose transformation language to a restricted technological domain, focusing on a well-defined feature subset, retaining the best characteristics, and maximizing usability in practical applications. While such language reusability engineering practices are well known in the domain-specific language engineering community [20] (e.g. based on software product line techniques [21]), to our best knowledge, no adaptation experience across such different modeling platforms has been reported yet for model transformation languages.

6 Conclusion

We have introduced a graph pattern based query language for EMF-IncQuery, a technology for model queries over EMF models, with use-cases ranging from model validation to on-the-fly model synchronization. The proposed language is derived from the graph pattern fragment of VTCL and tailored to the task of querying EMF models, with additional significant semantic extensions to its predecessor. The query language is complemented by a static type inference mechanism that is necessary to guide the interpretation of queries in some cases; additionally it can detect certain classes of developer errors (and can also provide valuable information to the code generator component of EMF-IncQuery).

The language extends a core graph pattern formalism (with nested negation) by rich attribute handling and aggregation. Query capabilities also include recursion and transitive closure, which is frequently needed but (in the general case) inexpressible in many query languages. The expressivity of the language is complemented by the beneficial performance characteristics discussed in [4].

As future work, we are planning to provide streamlined integration of the query system into a fully featured model validation framework. We also envision declarative support for incremental model transformation driven by query results, using the graph transformation formalism. Finally, we plan to extend the scope of queries from a single model state to the evolution of the model, in order to support change-driven transformations [22].

References

1. The Eclipse Project: Eclipse Modeling Framework. `http://www.eclipse.org/emf`.
2. Eclipse Model Development Tools Project: MDT-OCL website (2011) `http://www.eclipse.org/modeling/mdt/?project=ocl`.
3. Eclipse Modeling Project: EMF model query website (2011) `http://www.eclipse.org/modeling/emf/?project=query`.
4. Bergmann, G., et al.: Incremental evaluation of model queries over EMF models. In: Model Driven Engineering Languages and Systems, 13th Int. Conf., MODELS'10, Springer, Springer (10/2010 2010) Acceptance rate: 21%.
5. Gilles, O., Hugues, J.: Validating requirements at model-level. In: Ingénierie Dirigée par les modèles (IDM'08), Mulhouse, France (2008) 35–49
6. Mouratidis, H., et al.: A natural extension of Tropos methodology for modelling security. In: Agent Oriented Methodologies Workshop. Object Oriented Programming, Systems, Languages (OOPSLA), Seattle-USA, ACM (2002)
7. Tun, T.T., et al.: Model-based argument analysis for evolving security requirements. Secure System Integration and Reliability Improvement **0** (2010) 88–97
8. Varró, D., Pataricza, A.: VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. Journal of Software and Systems Modeling **2**(3) (October 2003) 187–210
9. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Program. **68**(3) (2007) 214–234
10. de Lara, J., Guerra, E.: Deep meta-modelling with metadepth. In: Proceedings of the 48th Int. Conf. on Objects, models, components, patterns. TOOLS'10, Berlin, Heidelberg, Springer-Verlag (2010) 1–20
11. W3C OWL Working Group: OWL 2 Web Ontology Language. Technical report, W3C (2009) `http://www.w3.org/TR/owl2-overview/`.
12. Ujhelyi, Z.: Static type checking of model transformation programs. In Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A., eds.: Graph Transformations. Volume 6372 of LNCS. Springer Berlin / Heidelberg (2010) 413–415
13. Object Management Group: Object Constraint Language, Version 2.2. (Feb. 2010)
14. Kolovos, D., Paige, R., Polack, F.: The Epsilon transformation language. In Vallecillo, A., Gray, J., Pierantonio, A., eds.: Theory and Practice of Model Transformations. Volume 5063 of LNCS. Springer Berlin / Heidelberg (2008) 46–60
15. Eclipse Modeling Project: Xpand wiki (2010) `http://wiki.eclipse.org/Xpand`.
16. Biermann, E., et al.: Precise semantics of EMF model transformations by graph transformation. In: MoDELS '08: Proceedings of the 11th Int. Conf. on Model Driven Engineering Languages and Systems, Springer-Verlag (2008) 53–67
17. Giese, H., Hildebrandt, S., Seibel, A.: Improved flexibility and scalability by interpreting story diagrams. In Magaria, T., Padberg, J., Taentzer, G., eds.: Proceedings of GT-VMT 2009. Volume 18., Electronic Communications of the EASST (0 2009)
18. Arendt, T., et al.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: MoDELS'10. Volume 6394 of LNCS., Springer (2010) 121–135
19. W3C: SPARQL Query Language for RDF. (Jan. 2008)
20. Cleenewerck, T., et al.: Evolution and reuse of language specifications for DSLs (ERLS). In: Object-Oriented Technology. ECOOP 2004 Workshop Reader. Volume 3344 of LNCS. Springer Berlin / Heidelberg (2005) 187–201
21. White, J., et al.: Improving domain-specific language reuse with software product line techniques. Software, IEEE **26**(4) (July-Aug 2009) 47–53
22. Ráth, I., et al.: Change-driven model transformations. In: Proc. of MODELS'09. Volume 5795/2009 of Lecture Notes in Computer Science. (2009) 342–356