Query-driven incremental synchronization of view models*

Csaba Debreceni, Ákos Horváth, Ábel Hegedüs, Zoltán Ujhelyi, István Ráth, Dániel Varró
Budapest University of Technology and Economy,
Department of Measurement and Information Systems,
1117 Budapest, Magyar tudósok krt. 2.
{debrecenics,ahorvath,hegedusa,ujhelyiz,rath,varro}@mit.bme.hu

ABSTRACT

Views are key concepts of domain-specific modeling in order to provide specific focus of the designers by abstracting from unnecessary details of the underlying abstract model. Usually, these views are represented as models themselves (*view models*), computed from the source model. However, the efficient maintenance of views when the source model changes is challenging, as recalculation from scratch has to be avoided to achieve scalability.

In the paper, we propose an approach to define view models in a highly automated way, based on declarative model queries. The views are automatically populated in accordance with the lifecycle of regular model elements - however, their existence is entirely bound to the underlying abstract model. This means that view models are automatically and incrementally maintained. Our contribution can also be interpreted as extending the concepts of derived features to derived objects, specified and maintained by incremental queries.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Theory, Algorithms

Keywords

incremental view model synchronization, derived objects

1. INTRODUCTION

*This work was partially supported by the CERTIMOT (ERC_HU-09-01-2010-0003), MONDO (EU ICT-611125), TÁMOP-4.2.2.C-11/1/KONV-2012-0001 and a collaborative project with Embraer. The TÁMOP-4.2.2.C-11/1/KONV-2012-0001 project has been supported by the European Union, co-financed by the European Social Fund.

VAO '14 York, UK

Modern domain-specific modeling environments offer domain engineers graphical or textual editors to alter the design, as well as multiple views each of which present only a specific aspect, in order to manage complexity. Views may capture a taxonomy (e.g. class generalization), a hierarchy (e.g. containment), interconnections, etc. of the system under design, and offer advanced navigation capabilities to highlight a selected element in different related views.

These views are frequently represented as models themselves (called *view models*), which are populated and maintained automatically from an underlying model (called *source model*). View models are commonly defined by means of queries, which identify a subset of elements in the system model to be included in the view. Such a view definition can also be considered as a special unidirectional model transformation from the system model to the view model. However, similarly to database view maintenance, maintaining views when the system model changes is a challenging problem, which necessitates dedicated support in the modeling tools in order to avoid the recalculation of views from scratch (after each individual source model change).

Naturally, multiple view models can be defined for the same source model, all of which need to be maintained as the model change. Moreover, view models may also depend on other view models, i.e. they can be derived from other view models forming a chain of dependent view models. In such a case, a change in the source model needs to be propagated transitively to all dependent views. Due to these challenges, most industrial tools built using state-of-the-art technologies such as the Eclipse modeling infrastructure rely on hand coded views.

In this paper, we define a lightweight mechanism to specify view models by means of derivation rules captured in the form of annotated model queries following incremental and live graph transformations on a semantic level. As a result, after initialization, the view elements are automatically populated in accordance with the lifecycle of regular model elements - however, their existence (and non-existence) is entirely bound to the underlying source model. We propose an efficient incremental technique for view maintenance on top of the EMF-IncQuery framework, which supports multiple views over a source model and the chaining of view models.

Our concepts and implementation is presented in the context of an avionics toolchain where high-level functional architecture models are extracted from low-level Simulink models. Moreover, a graphical notation model is also derived as a view from the architecture model to demonstrate the chaining of views.

Our paper is structured as follows. A tooling challenge in an avionics context is presented as a motivating example in Sec. 2. The query language is overviewed and the derivation rules are detailed in Sec. 3. Implementation details on query based objects for incremental unidirectional view model synchronization are discussed in Sec. 4. Initial performance measurements are presented in Sec. 5. Related work is overviewed in Sec. 6 while Sec. 7 concludes our paper.

2. A MOTIVATING SCENARIO: AN AVIONICS TOOLCHAIN

Our motivating example is extracted from an ongoing industrial avionics research project where the main purpose was to define a complex, semi-automated development chain to allocate software functions onto different hardware architectures. The tool takes Matlab Simulink (SIM) block diagrams models as input, and it abstracts them into a functional architecture model (FAM), which later serves as an input for the allocation step. The FAM has to be available both in its abstract EMF representation and also in a graphical notation via the user interface of the tool (illustrated in Fig. 1). Our example is simplified by considering only subsystems collected into functions and complex block-port-signal interconnections into information links.

EXAMPLE 2.1. *Our sample* Simulink model *of Fig. 1 consists of six blocks, out of which* PilotControl Navigation, Engine Management System (EMS) *and* Flight Management System (FMS) *represent* functions *(blue rectangles tagged with "func") while* SubS1 *and* SubS2 *represent blocks which are irrelevant for allocation (green rectangles without a "func" tag). These blocks are arranged into a containment hierarchy (with* PilotControl *and* SubS2 *as top level elements). The* FAM model *contains only equivalents of the four function blocks and it also abstracts from complex block-port-signal interconnections as supported by Matlab Simulink (depicted by red rectangles for external ports of a function which within the function translates into grey spheres for representing the same ports within the function), representing them as (logical) information links between functions. Finally, the graphical* notation model *of the FAM shows only the functions connected by edges for information links.*

As only the SIM models are allowed to be modified, the abstract FAM is a view model which always reflects the current structure of the underlying SIM model. Furthermore, the graphical notation of the FAM is a view model on a different abstraction level, which is strongly tied to the abstract FAM model, thus creating a chain of views. Additionally, during the building of view models, a generic traceability model is also constructed to store traces between the objects of source and target models. The main concepts of these modeling languages are captured by corresponding metamodels in EMF presented inFig. 2.

The main *EClass* (represented by a rectangle) of the *Simulink* metamodel is the SubSystem that includes InPort and OutPort elements and PortBlocks. It has a tag *EAttribute* which stores the role of the element. Subsystems may contain Blocks along the subBlocks *EReference*. A Signal represents a communication link between one OutPort and some InPorts of different SubSystems.

The (simplified) *FAM* metamodel contains Function elements and InformationLinks between them if a communication link exists. In this context, the provider function sends data to the consumer.

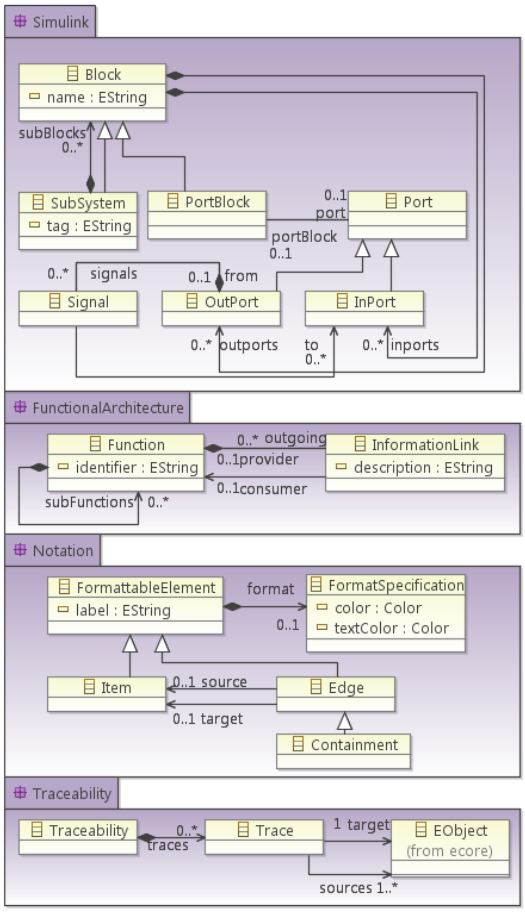


Figure 2: Metamodels of Simulink, Functional Architecture and Notation model

The *Notation* metamodel (similar to standard notation metamodels used e.g. by the Eclipse Graphical Modeling Framework) describes a uniform model format to describe graphical representations, such as lists, trees or graphs. It defines Edges or Containments between Items where the edges symbolize simple reference relations while the containments represent a hierarchical structure. Additionally, these elements can have some display information stored in FormatSpecification such as background color and text color.

A generic *Traceability* metamodel is used to define directed links between a specific set of EObject instances (common supertype for all EMF classes) from the source and an instance of target metamodels using Trace elements.

Challenges of view models.

From lessons learned while developing a complete and a feature-rich version of this tool chain, we have identified the following challenging recurring tasks: (i) the definition of view models in a reusable and preferably declarative way, (ii) facilitating efficient and incremental updates to view models when source models change, (iii) serialization support for view models taking dependency chains into consideration and (iv) chaining view model abstractions in order to reuse view transformations across several layers of abstraction.

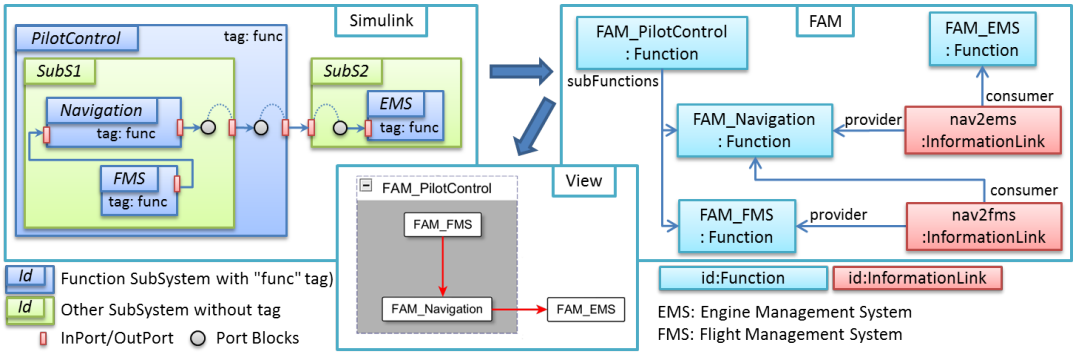


Figure 1: Simulink models, functional architecture models with graphical notation

3. DEFINITION OF VIEW MODELS

View models are defined by using a fully declarative, rule based formalism. Preconditions of rules are defined by model queries (building on the language of the EMF-IncQuery framework [4]), which identify parts of interest in the source model. Derivation rules then use the result set of a query to define elements of the view model. On the theoretical level, queries are defined as graph patterns while rules can be formalized as live graph transformation rules [15] and executed using the EMF-IncQuery Event-Driven Virtual Machine (EVM) [19]. Informally, when a new match appears in the result set of a query then the corresponding derivation rule is fired to create elements of the target view model. When an existing match disappears in the result set of a query, the inverse of the rule is fired to delete the corresponding view model elements.

3.1 Model queries by graph patterns

A graph pattern (GP) represents *structural constraints* prescribing the interconnection between nodes and edges of a given type. They are extended with *algebraic expressions* to define *attribute constraints* and *pattern composition* to build up new queries by reusing existing graph patterns. The called pattern is used as an additional set of constraints to meet. A *negative application condition* (NAC) describing cases when the original pattern is *not* valid. *Pattern parameters* are a subset of nodes and attributes representing the model elements interesting from the perspective of the pattern user.

A *match* of a pattern is a tuple of pattern parameters that fulfill all the following conditions: (1) have the same structure as the pattern; (2) satisfy all structural and attribute constraints; and (3) does not satisfy any NAC. When evaluating the results of a graph pattern, a pattern parameter can be used both as *input* to pre-select model elements or attribute values and *output* which are bound by a successful match of the pattern.

Example 3.1. *We demonstrate the capabilities of the* EMF-IncQuery *language using a few simple patterns described in Lst. 1. Pattern* function *enumerates all* SubSystems *that are tagged as "Func". The* functionIdentifier *pattern (Lst. 1) pairs the name of the source* Block *and the corresponding* SubSystem *where the* SubSystems *are tagged as "Func" - for this reason, it reuses the previously defined patterns using the* find *keyword. Finally,* subFunction *gathers all sub-systems for each* SubSystem.

3.2 Derivation rules by query annotations

Listing 1 Model query and derivation rule subFunction

```
@QueryBasedObject(eClass = "Function")
pattern function(subsys : SubSystem){
  SubSystem.tag(subsys, "Func");
}
@TraceLookup(src ={subsys}, trg =f, type = "Function")
@QueryBasedFeature(src = f, trg = id, feature = "id")
pattern functionIdentifier(subsys:SubSystem, id:EString){
  find functionSubsystem(subsys);
  Block.name(subsys,id);
}
@TraceLookup(src = {psys}, trg = p, type = "Function")
@TraceLookup(src = {ssys}, trg = s, type = "Function")
@QueryBasedFeature(src=p, trg=s, feature="subFunctions")
pattern subFunction(psys :SubSystem, ssys :SubSystem){
  find functionSubsystem(psys);
  SubSystem.subBlocks+(psys,ssys);
  find functionSubsystem(ssys); }
```

In our context, *view models* are conceptually equivalent to regular models. Their structure is defined by a metamodel, which thus consists of (view) classes, features (e.g., references and attributes). However, the lifecycle (i.e. existence and non-existence) of view model elements is strongly tied to certain elements of the underlying source model as being specified by query-based *derivation rules*.

In principle, an arbitrary model transformation language could be used to specify how to synthesize view models. Our approach uses *a lightweight mechanism exploiting annotations* of the previous query language to capture derivation rules, instead of relying on a full-fledged model transformation language. As a result, we obtain a declarative formalism compliant with the execution semantics of incremental and non-deleting graph transformations (GT) , where the LHS of the GT-rule is defined by the query itself and the creation rules are captured by its annotations.

Note that unlike bidirectional model synchronization approaches like triple graph grammars [14, 9], derivation rules define a unidirectional transformation from the source model to the view model, where model changes are also propagated only in this direction but not vice versa.

A derivation rule is obtained when a query is extended by one or more of the following annotations:

- `@QueryBasedObject(eClass = "ClassName")`: The QueryBasedObject annotation adds a new view object of type ClassName and creates a Trace element into the *Traceability* model where parameters of the pattern will be the source n-ary, and the created object will be in the target.

- `@QueryBasedFeature(src = source, trg = target, feature = "FeatureName"):`
  The QueryBasedFeature annotation, as defined in [16], is used to define references and attributes initialized by queries. It sets the pattern parameter trg as the value of FeatureName feature of the pattern parameter src where src has to be a view model object.

Additionally, to support the explicit traceability, another annotation is defined to select the corresponding elements from the created view model.

- `@TraceLookup(src = source1, source2, trg = target, type = "TargetClass")`
  The TraceLookup annotation is used to declare a new variable with its "trg" attribute to use in other annotations, find the proper target element in the view model which is created from the collection of "src", and select the target elements of type "TargetClass". (Note that the type is only an optional filter.)

3.2.1 *Informal execution semantics of derivation rules*

The execution of derivation rules is triggered by changes in the source model. (1) When a new match of a query appears in the source model, the corresponding derivation rules are fired and new elements are created in the view model, according to the annotations above. (2) When an existing match of an annotated query disappears, the corresponding derivation rule is reverted to undo all its creation operations (thus deleting its corresponding view-model elements). (3) Traceability links are established between source and view model elements.

In fact, changes in the source model can trigger more derivation rules at the same time, these rules have to be ordered. (a) When new matches appear, the QueryBasedObject rules have higher priority than QueryBasedFeatures to firstly create the elements in the view model, and then set its features. (b) When existing matches disappear, the priorities change so firstly reset the features of elements then delete the elements from the view model.

We assume that the derivation rules are non-conflicting to guarantee that the creation of each view model element is uniquely determined by one (or more) elements of the source model. Although not detailed in the current paper, the approach can be scaled to more complex view model derivation rules by assigning rule priorities and custom execution scheduling strategies as supported by the EMF-IncQuery EVM [19].

The current approach is tailored towards interactive applications where the source model is assumed to be modified in "atomic" transactions, i.e. the user modifying the model step-by-step each time performing a simple command. Thus the system can synchronize after each such step is registered (through *model change notifications* propagated through the incremental pattern matcher, see Sec. 4) as an atomic run-to-completion step.

Example 3.2. *The patterns from Lst. 1 have derivation annotations defined that are to be read as follows. The derivation rule of* function *in Lst. 1 prescribes the derivation of a view object of type* Function *in the FAM together with a traceability link. The* functionIdentifier *sets the* identifier *attribute of the created object to the name of the corresponding Simulink block (passed by the pattern parameter* identifier*).*

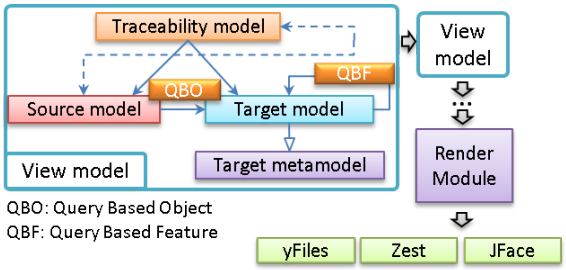


Figure 3: Correlation between view model chaining and visualizing

Finally, the derivation annotation subFunction *prescribes the creation of a* subFunctions *reference when a new match of the query between the view objects identified by* p *and* s *appears. Applying these rules on the SIM model of Fig. 1, we obtain all the (blue)* Function *nodes of the FAM model and the appropriate* subFunctions *references between them.*

3.3 Derivation rules of the case study

Let us now demonstrate how to capture the chain of view models for the motivating example of Fig. 1. As Fig. 3 shows, a graphical visualization of the FAM is obtained by chaining view models. In fact, a graphical visualization is derived as a view model where its source model is the FAM, which is a view model in itself. This graphical visualization is used as the input of one of the available renderers that displays the model using e.g. Eclipse JFacelist or table viewers, or Zestor yFiles for Javagraph visualization engines.

3.3.1 *From Simulink models to FAM*

First in Lst. 2, we show how the complex interconnections of Simulink blocks (via ports and signals) are simplified to create InformationLinks, which connect a consumer and a provider Function in the FAM. For that purpose, we define a derivation rule for query informationLink using two auxiliary queries connectedBlocks and inPortToInPortConn (where a sourceInPort is connected to a targetInPort along the block hierarchy, either from upper level to lower level or the other way round). Query connectedBlocks checks if there is a signal assigned between the two ports. To set the references and the description attribute of the new created InformationLink element, we use another derivation rule called informationLinkData that selects the Function created from the corresponding source elements.

When applying the set of rules to the SIM model of Fig. 1, we obtain the FAM as a view model also depicted in Fig. 1. Note that the real query in the industrial tool is contains several similar queries to detect different types of connections between blocks. For space considerations, we restrict the synthesis of information links to a single case when hierarchical levels are crossed.

3.3.2 *From FAM to a graphical notation model*

As visualizing parts of EMF models are a recurring task, we have also defined a set of shorthands to make rule specification more concise. The (1) @Item rule creates an Item element from the Notation metamodel (see Fig. 2) with an attribute named label. The (2) @ContainsItem and (3) @Edge rules create containment and non-containment references between selected model elements, respectively. Finally, the (4) Format annotation defines additional formatting rules to be

Listing 2 Defining information links

```
// inport connected to inport through hierarchy
pattern inPortToInPortConn(src :InPort, trg :InPort){
  // from upper level into subsystem
  Port.portBlock.outports.signals.to(src, trg);
} or {
  // from inside subsystem to upper level
  Port.portBlock.inports(outerOutPort, src);
  OutPort.signals.to(outerOutPort, trg);
}
// blocks are directly connected or transitively
pattern connectedBlocks(prov :Block, con :Block){
  Block.outports.signals.to(prov, consumerInPort);
  Block.inports(con, consumerInPort);
} or {
  Block.outports.signals.to(prov, innerIP);
  find inPortToInPortConn+(innerIP, consumerInPort);
  Block.inports(con, consumerInPort);
}
@QueryBasedObject(eClass = "InformationLink")
pattern informationLink(prov :SubSystem, con :SubSystem){
  find functionSubsystem(prov);
  find connectedBlocks(prov, con);
  find functionSubsystem(con);
}
@TraceLookup(src={c}, trg=cF, type = "Function")
@TraceLookup(src={p}, trg=pF, type = "Function")
@TraceLookup(src={p,c}, trg=l, typ = "InformationLink")
@QueryBasedFeature(src=l, trg=pF, feature = "provider")
@QueryBasedFeature(src=l, trg=cF, feature = "consumer")
@QueryBasedFeature(src=l, trg=desc, feature = "desc")
pattern informationLinkData(p : SubSystem, c : SubSystem,
                            desc : EString){
  find informationLinks(p,c);
  desc = eval(p.name + "->" + c.name);}
```

attached as a FormatSpecification instance to the current element.

EXAMPLE 3.3. *We illustrate the view model derivation by a set of derivation rules described in Lst. 3. The patterns* item, containment *and* edge *patterns are responsible for creating their corresponding Notation model elements using elements from the FAM metamodel and the created trace elements. In Fig. 4a we illustrate the notation model derivation process. At first, the* Item *elements are created together with their corresponding specification from the matches of the* item. *Then for all matches of the* containment *and* edge *patterns the source and target* Items *are looked up, and the corresponding notation element is created. Note that in this case,* TraceLookup *annotations do not use type filtering because every created elements are* Item. *Finally, the resulting model is added to a renderer. Fig. 4b depicts the finished visualization as created by the yFiles renderer.*

4. UPDATING VIEW MODELS BY INCREMENTAL QUERY EVALUATION

Now we outline the mechanism of handling changes in the source models and show how derivation rules are used to update the target model continuously.

4.1 Incremental evaluation of queries

The key to update any target model synchronously is incremental query evaluation, where query result changes caused by model manipulation are also provided without complete reevaluation. We use the EMF-INCQUERY framework [20] that supports incremental query evaluation over

Listing 3 Deriving graphical notation from functional architecture model

```
@Item(label = "$func.identifier$")
@Format(color = "#FFFFFF", textColor = "#000000")
pattern item(func : Function) {
  Function(func);
}
@TraceLookup(source = {p}. target = {par})
@TraceLookup(source = {s}. target = {sub})
@ContainsItem(source = par, target = sub)
pattern containment(par : Item, sub : Item) {
  Function.subFunctions(p, s);
}
@TraceLookup(source = {p}. target = {provider})
@TraceLookup(source = {c}. target = {consumer})
@Edge(source = provider, target = consumer)
@Format(color = "#FF0000")
pattern edge(provider : Item, consumer : Item) {
  InformationLink.provider(link, p);
  InformationLink.consumer(link, c); }
```

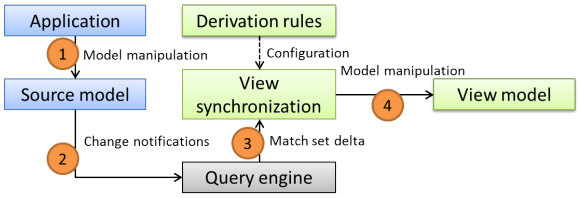


Figure 5: Overview of integration architecture

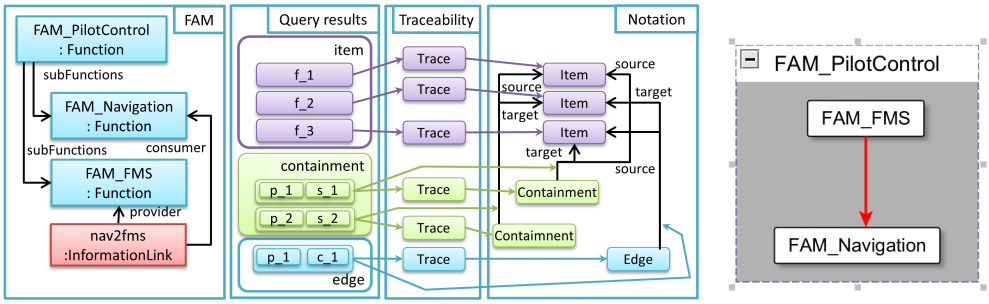
EMF instance models by constructing a *Rete* rule network that processes *change notifications* sent by the EMF notification API. This Rete network remains in operation as long as the query is needed: it continues to receive elementary change notifications and propagates them to produce *query result deltas*. Even though this imposes a slight performance overhead on model manipulation, and a memory cost proportional to the cache size (approx. the size of match sets), EMF-INCQUERY can evaluate very complex queries over large instance models very efficiently.

The framework also provides an interface to notify any observer object when a new match appears or an old one disappears. With this functionality, our query-based approach has the ability to react on these events by firing all the required derivation rules.

4.2 Integration architecture of view synchronization

The view models are constructed and updated incrementally by translating the match set deltas from the incremental query evaluation into model modifications. The integration architecture for our framework is shown in Fig. 5. After an initial configuration of the view synchronization using the derivation rules, the source model is loaded by EMF and the initial query results from the query engine are used for constructing the view model.

Once the view model is created, the synchronization is a reactive process with the following main steps: (1) model manipulations are carried out by an application (e.g. graphical editor or user interface) on the source model, (2) these changes are propagated to the query engine as change notifications, (3) a match set delta is sent to the view synchronization, and finally, (4) model manipulation is performed on the view model based on the derivation rules.



(a) The Derivation Process (b) Rendering (yFiles)

Figure 4: Notation Model Derivation

4.3 From match set changes to view model synchronization

4.3.1 Initial setup of derivation rules

The view models are created entirely based on the derivation rules specified as model queries with annotations (see Sec. 3.2). Additionally, the Ecore models (i.e. metamodels) for view models are also selected. These metamodels must include the EClasses and EStructuralFeatures (EAttributes and EReferences) defined by the derivation rules.

The view synchronization groups the derivation rules into two sets, based on the annotation type (query-based object, query-based feature). For each rule, a matcher for the model query is initialized in the query engine and handlers are attached to the matcher to receive deltas. These handlers are responsible for modifying the objects or feature settings in the view model.

4.3.2 Query result deltas

As described in Sec. 4.1, the Rete network processes change notifications and provides query result deltas, that can be considered as higher level notifications. A delta is a structure $Delta = (Found, Lost)$, where $Found$ is the set of new matches that appeared and $Lost$ is the set of old matches that disappeared since the last delta.

The initial construction is executed similarly as incremental updates are handled, since the initial query results can be considered as a delta containing the initial matches in $Found$.

4.3.3 Source-view traceability

During the synchronization process, internal traceability links are stored for identifying the view model elements corresponding to appearing and disappearing matches in deltas. Fig. 6 shows the Simulink source model, the query results, the traceability links and the FAM view model for the example in Fig. 1 (the EMS function is omitted).

The query results contain all matches for each model query that is used by derivation rules in the Simulink-FAM example. The matches are tuples of pattern parameter assignments, where each assignment is a direct reference to an EObject in the source model, or an attribute value. For easier readability this assignment between the FAM elements and their corresponding pattern parameter is not shown in Fig. 6 (like PilotControl assigned to f_1).

The traceability structures are created for each view model object created by object derivation rules. The traceability structure stores the source match for the view model element. This explicit traceability model is managed by the view model synchronization and is used for finding the container and target objects of view model links and attributes.

4.3.4 Incremental view maintenance

Fig. 6 also shows the order of steps performed when the FMS and Navigation subsystems in Simulink are connected by their ports: (1) a delta is received with $Found = \{(f\_2, f\_3)\}$ for the informationLink query, (2) the handler of the object derivation rule (see Lst. 2) creates a new Trace in the traceability model and (3) adds an InformationLink EObject to the view model. When all view model objects are created, (4) another delta is received with $Found = \{(f\_2, f\_3, l\_1, d\_1)\}$ for informationLinkData that leads to (5) set the consumer and provider references in the view model.

The targets of these references are selected by finding the appropriate Trace in the explicit traceability model, where the source of the link is the proper match. For example, Navigation is identifiable by the match f_2 in Simulink, and the view element for the corresponding Trace is FAM_Navigation. Therefore, the consumer reference is set to this object.

Although not shown in Fig. 6, the description attribute of the information link is set at the same time with the references in a similar way.

4.3.5 Derivation rule priority

Since the traceability links are used for identifying view elements and their corresponding matches in the delta, the order of handler execution is not arbitrary. For $Found$ matches, object derivation rules are executed first, even if there are link and attribute rules defined for the same query. Links and attributes are set after the objects are created. However, for $Lost$ matches, links and attributes are handled before objects to ensure that the source and target objects are still available. Derivation rules of a given query can depend on objects created by rules for other queries, therefore deltas are not handled separately, but combined based on both rule types (query-based object and feature) and events (found, lost).

5. PERFORMANCE EVALUATION

We carried out an initial evaluation to demonstrate the scalability of our approach on the Train benchmark case study [20] (an accepted model querying performance benchmark). It was selected as in our industrial case study we only had moderate size models containing only a few hundred elements and their execution in all cases were smaller than half a second.

All measurements were executed on a developer PC with a 3.5 GHz Core i7 processor, 16 GB RAM, Windows 7 and

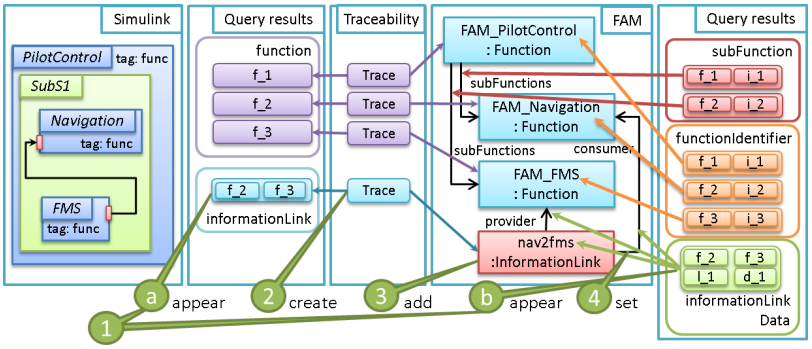
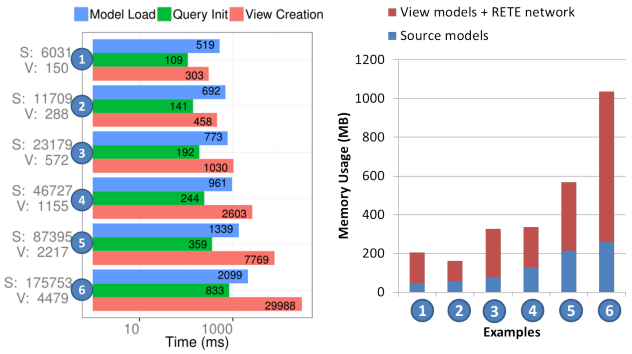


Figure 6: Internal traceability from base models to view models



(a) Runtime performance (b) Memory Usage

Figure 7: Measurement Results

Java 7. Measurement were repeated ten times, and the time and memory results were averaged. To avoid interference between different executions, each time a new JVM was created with a 2 GB heap limit. The time to start up and shut down the JVM was not included in the measurement results.

Based on the Train benchmark we defined 3 queries including advanced features such as transitive closure computation to define the view models. For the different cases we selected 6 model sizes for the source model ranging between 6000–175000 elements that produced view models of 150–4500 elements.[1]

Fig. 7a depicts the runtime results: each row shows the time required to (1) load a source model, (2) initialize the query engine and (3) derive the view model. Additionally, for each instance model, the number of model elements of the source (S) and view (V) models are presented.

Fig. 7b highlight the overall memory usage for the different model instances, where the blue section shows the memory consumption of the source model alone and the (upper) red part depicts the memory needed for both the Rete network of the cached queries and the generated view-models. As expected the overall memory consumptions for the view models are 2-3 times larger than the source models themselves due to the Rete caching mechanism [20]. However, this intensive caching provides the fast incremental update in case of small source model changes [20].

To sum up, we were able to demonstrate that our view model based approach is capable of scaling up to source models with more than 100 000 elements within acceptable timeframe and memory consumption, which is in line with our previous experiments [20]. As future work we plan to execute further evaluations to provide (i) comparison with similar approaches and (ii) performance characteristics for different complexity view models.

[1]More details are available at `https://incquery.net/publications/query-based-synchronization`

6. RELATED WORK

Defining database views by derived classes and model queries captured in OCL was first proposed in [2], but without support for incremental view calculation. This approach could exploit recent results for incremental support of OCL as in [5, 10]. Obviously, any other model query languages can be used for defining database or model views, our paper uses of a graph based query language [4].

Formal frameworks for bidirectional model synchronization like [7, 18] are more general on the theoretical level than unidirectional view maintenance by query-based derived objects, but no implementation is proposed in these papers. There are several model transformation and synchronization approaches and tools providing certain levels of incrementality, such as triple graph grammars (TGGs) [14, 9], ATL [12, 21]or QVT [17]. Fully incremental approaches guarantees minimal number of processing steps upon change, which is achieved so far only by RETE-based approaches [15, 8]).

View maintenance by incremental and live QVT transformations is used in [17] to define views from runtime models. The proposed algorithm operates in two phase, starting in check-only mode before an enforcement run, but its scalability is demonstrated only on models up to 1000 elements.

VirtualEMF [6] allows the composition of multiple EMF models into a virtual model based on a composition metamodel, and provides both a model virtualization API and a linking API to manage these models. The approach is also able to add virtual links based on composition rules. In [21] an ATL-based method is presented for automatically synchronizing source and target models. In [13] correspondences between models are handled by matching rules defined in the Epsilon Comparison Language, where the guards use queries similarly to our approach, although incremental derivation is not discussed.

A recent work by Diskin et al. [7] proposes a theoretical background for model composition based on queries using Kleisli Categories, in their approach derived features are used for representing features merged from different metamodels. The conceptual basis is similar to our approach in using query-based derived features and objects, however, it offers algebraic specification, while our approach might serve as an implementation for this generic theoretical framework.

View models can also be obtained by means of model composition. Anwar [1] introduces a rule-driven approach for

creating merged views of multiple separate UML models and relies on external traceability and OCL expressions to support model merging and composition. ConceptBase.cc [11] is a database (DB) system for metamodeling and method engineering and defines active rules that react to events and can update the DB or call external routines, which can be used for incremental view calculation, but the tool itself is not EMF compliant.

Compared to previous work of the authors, we extend [16] by providing support for derived objects (in addition to derived attributes and relations), and improve performance by smart traceability links for (a subclass of) change driven transformations [3]. Furthermore, it offers a fully declarative transformation language for incremental live transformation with explicit traceability unlike [15].

We believe that our contribution is unique in providing a combination of fully incremental, unidirectional synchronization of view models allowing chaining of views by means of explicit traceability links and derived objects, which depend on the match set of the precondition of derivation rules.

7. CONCLUSION AND FUTURE WORK

In the paper, we presented an approach for deriving and synchronizing non-persistent (virtual) view modes in an incremental (but uni-directional) way that supports the maintenance of multiple views over the same source model and also the chaining of view models along derivation rules. To support the definition of such view models we extended the EMF-IncQuery framework with: (i) the ability to define derivation rules using annotations over graph queries, (ii) an explicit tracing mechanism that and (iii) an EVM based runtime environment to automatically maintain and synchronize the view models. Finally, we provided initial performance evaluation of our approach on a model validation performance benchmark over models with more than 100 000 elements.

As future work we plan to (i) provide a domain specific language for derivation rule definition with more precise lifecycle management capabilities and (ii) extend our approach to serve as a possible runtime environment for Kleisli category based model transformations.

8. REFERENCES

[1] A. Anwar, S. Ebersold, B. Coulette, M. Nassar, and A. Kriouile. A rule-driven approach for composing viewpoint-oriented models. *Journal of Object Technology*, 9(2):89–114, Mar. 2010.

[2] H. Balsters. Modelling database views with derived classes in the UML/OCL-framework. In *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, volume 2863 of *LNCS*, pages 295–309. Springer Berlin / Heidelberg, 2003.

[3] G. Bergmann, I. Ráth, G. Varró, and D. Varró. Change-driven model transformations - change (in) the rule to rule the change. *Software and System Modeling*, 11(3):431–461, 2012.

[4] G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró. A graph query language for EMF models. In *Proc. International Conference on Model Transformation*, LNCS 6707, pages 167–182. Springer, 2011.

[5] J. Cabot and E. Teniente. Incremental integrity checking of UML/OCL conceptual schemas. *J. Syst. Softw.*, 82(9):1459–1478, 2009.

[6] C. Clasen, F. Jouault, and J. Cabot. Virtual Composition of EMF Models. Lille, France.

[7] Z. Diskin, T. Maibaum, and K. Czarnecki. Intermodeling, queries, and Kleisli categories. In *FASE 2012*, Tallinn, Estonia, 2012. Springer.

[8] A. H. Ghamarian, A. Jalali, and A. Rensink. Incremental pattern matching in graph-based state space exploration. *ECEASST*, 32, 2010.

[9] H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling (SoSyM)*, 8(1), 3 2009.

[10] I. Groher, A. Reder, and A. Egyed. Incremental consistency checking of dynamic constraints. In *FASE 2009*, volume 6013 of *LNCS*. Springer, 2010.

[11] M. A. Jeusfeld, M. Jarke, and J. Mylopoulos. *Metamodeling for Method Engineering*. The MIT Press, 2009.

[12] F. Jouault and M. Tisi. Towards incremental execution of ATL transformations. In *Proc. of ICMT'10, 3rd Intl. Conference on Model Transformation*. Springer, 2010.

[13] D. S. Kolovos. Establishing correspondences between models with the epsilon comparison language. In *5th European Conference on Model Driven Architecture*, pages 146–157, Enschede, The Netherlands, 2009.

[14] M. Lauder, A. Anjorin, G. Varró, and A. Schürr. Efficient model synchronization with precedence triple graph grammars. In *Graph Transformations - 6th International Conference, ICGT 2012, Bremen, Germany, 2012*, LNCS, pages 401–415, 2012.

[15] I. Ráth, G. Bergmann, A. Ökrös, and D. Varró. Live model transformations driven by incremental pattern matching. In *Theory and Practice of Model Transformations*, LNCS 5063, pages 107–121. Springer, 2008.

[16] I. Ráth, A. Hegedüs, and D. Varró. Derived features for EMF by integrating advanced model queries. In *Modelling Foundations and Applications*, LNCS 7349, pages 102–117. Springer, 2012.

[17] H. Song, G. Huang, F. Chauvel, W. Zhang, Y. Sun, W. Shao, and H. Mei. Instant and incremental QVT transformation for runtime models. In *Model Driven Engineering Languages and Systems*, LNCS 6981, pages 273–288. Springer, 2011.

[18] P. Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software and System Modeling*, 9(1):7–20, 2010.

[19] The Eclipse foundation. EMF-IncQuery Event-Driven Virtual Machine, 2014. `http://wiki.eclipse.org/EMFIncQuery/DeveloperDocumentation/EventDrivenVM`.

[20] Z. Ujhelyi, G. Bergmann, A. Hegedüs, A. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, 2014. In press.

[21] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *Automated Software Engineering (ASE'07)*, pages 164–173. ACM, 2007.