Automated abstraction in model-driven engineering

Csaba Debreceni
Budapest University of Technology and Economy,
Department of Measurement and Information Systems,
1117 Budapest, Magyar tudósok krt. 2.
debrecenics@mit.bme.hu

***Abstract*—In software development, the model-driven engineering (MDE) paradigm envisions a process starting from high-level system models and through several well-defined abstraction and refinement steps automatically derive source code, documentation or configuration artifacts.**
In MDE the concepts of view-models are widely used to provide specific focus for the designers by abstracting away unnecessary details of the underlying system model to highlight relevant domain specific information (e.g., reliability model, networking architecture, etc.). However, maintaining these abstract views upon changes in the system model is cumbersome, especially in order to avoid the recalculation of the complete view and provide instantaneous view-model updates.
In this paper, I propose an approach to incrementally synchronize abstract view-models defined by declarative model queries. These view-models are automatically populated with derived objects in accordance with the lifecycle of regular model elements. The approach is evaluated on an ongoing avionics research project built on top of the Eclipse Modeling Framework (EMF).

Keywords: model-driven engineering, derived object, view-models, abstraction

I. INTRODUCTION

Modeling is considered as one of the most elementary discipline in engineering. Its purpose is to overcome complexity, increase understandability or ease design. In software development, the model-driven engineering paradigm follows this concept by envisioning a process starting form high-level system models and through several well-defined abstraction and refinement steps automatically derive source code, documentation or configuration artifacts.

In Model-driven Engineering (MDE), the concept of view-models (e.g., reliability model, hardware architecture etc.) automatically derived from detailed system models to highlight relevant, domain specific information is a widely used and accepted approach (model abstraction). Synchronization and maintenance between these system (source) models and their different view-models are usually done by batch transformations. Unfortunately, as these transformations lack of incremental execution whenever a change happens in the corresponding system model the view-models have to be cleared and then all the transformations have to be executed to rebuild them

In this paper, I propose an approach that can incrementally synchronize abstract view-models defined by a set of queries. These view-models consist of derived objects automatically materialized from the source model tightly following its lifecycle, resulting in full featured (view-)model.

An initial prototype framework is built upon the Eclipse Modeling Framework (EMF) [12] considered as the de facto industry standard for modeling, and EMF-IncQuery [11], an incremental graph pattern matching framework with declarative pattern definition capabilities. Finally, the approach is evaluated on an ongoing avionics research project.

The rest of the paper is structured as follows. Section II provides a motivating example captured from avionics domain. Section III gives an overview of the approach where the main challenges are also mentioned. Section IV describes the methodology to define view-models with model queries. The concrete working mechanism is presented in Section V while in Section VI, related works are assessed. Finally, Section VII concludes this paper.

II. A MOTIVATING SCENARIO: MODEL-ABSTRACTION IN AN AVIONICS SYSTEM

The motivating example of this paper is extracted from an 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 diagram models as input, and it automatically abstracts them into a functional architecture model (FAM), which later serves as an input for the allocation step. The example is simplified by considering only subsystems collected into functions and complex block-port-signal interconnections into information links.

A sample Simulink model depicted in Figure 1. , consists of five blocks, out of which *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). The abstracted FAM model contains only equivalents of the three `function` blocks and it also abstracts from complex block-port-signal interconnections, representing them as (logical) links between functions where the provider function sends data to the consumer.

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.

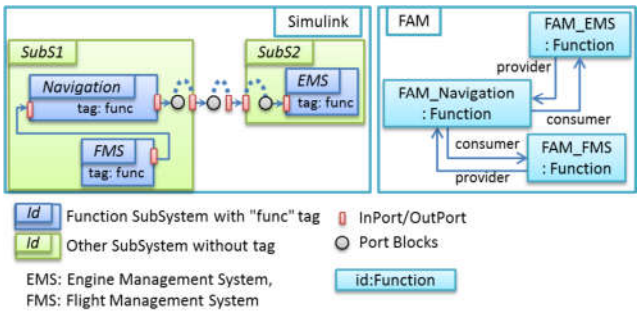

Figure 1. Simulink model example and its proper FAM representation

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.

The main concepts of these modeling languages are captured by corresponding metamodels in EMF presented in Figure 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, where a function can be connected to another function as a representation of information links. 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 target metamodels using `Trace` elements.

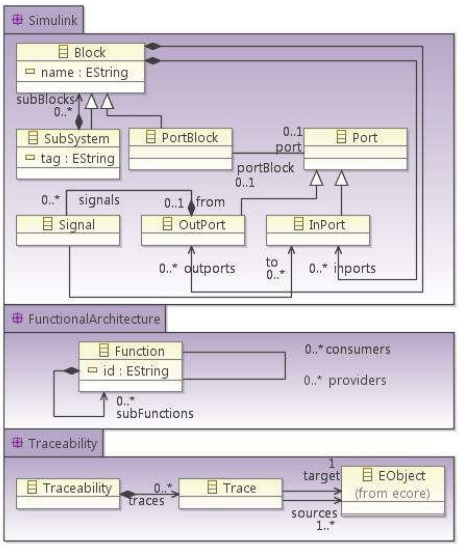

Figure 2. Simulink and Functional Architecture metamodels

III. Overview of the Approach

Main purpose of this approach is to create regular models where the elements of them are synchronized immediately and incrementally when the corresponding source model is modified. As a process (depicted on Figure 3.), the approach uses *source model*, *target metamodel* and *derivation rules* defined by model queries (*item, edge, attribute*) as input whereas the output is a *target model* that realizes the *target metamodel* with the previously defined properties.

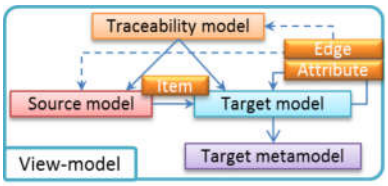

Figure 3. Overview of the approach

Referring to the motivating example, Simulink model is interpreted as *source model*, FAM metamodel represents the *target metamodel*, while the concrete FAM model will be the *target model* (derivation rules are explained in Section IV).

An initial prototype implementation is built upon the EMF-IncQuery and its Viewers [10] technology. EMF-Incquery is an incremental model query framework that allows to define queries via its pattern language and incrementally get notifications when a new match appears for a match or an old disappears. The Viewers technology provides an API to display query results for example in a tree-view or a graphical Zest view. It works with a similar terminology as we need such as item or edge. With some modification, I also extended this layer to be able to use the "attribute" expression. Theoretically, the prototype is a Viewers contribution.

During the implementation, I faced with the following challenges: *How to define derivation rules and view-models by derivation rules?* Derivation rules, in this solution, can be given by model queries where a match can be interpreted as a precondition for a rule and the execution is defined as additional information to the specified pattern. *How to trace the created elements during the synchronization?* An important task is to get the ability to trace all view-model elements from the source objects. It allows the approach to notify only the necessary part of the view-models about a change from the source model. *How to execute these derivation rules?* At the same time, more matches can appear during the modification of the source model, but the approach has to solve the conflicts between the rules and in all cases, the output has to be deterministic.

IV. Definition of Viewmodels

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 [1]), 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. 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.

The capabilities of the EMF-IncQuery language is demonstrated in Listing 1. using a few simple patterns. Pattern

`function` enumerates all `SubSystems` that are tagged as "`func`". The `trace` pattern lists all possible pairs of elements that are connected via a `Trace` model element. Note that the types of the parameters are undefined which enables to use the same pattern for arbitrary model elements connected by the traceability metamodel. The `functionId` pattern pairs the name of the source `Block` and the corresponding `Function` - for this reason, it reuses the previously defined patterns using the find keyword. Pattern `inPortToInPort` finds all inport pairs connected to each other. The `connectedBlocks` lists the block pairs linked through several inports. Finally, pattern `link` select block pairs that have to be linked in the FAM model.

```
pattern trace(src, trg) {
  Trace.sources(trace , src);
  Trace.target(trace , trg); }

@Item(eClass = "Function")
pattern function(subsys : SubSystem) {
  SubSystem.tag(subsys , "func"); }

@Attribute(source = func, target = id, eAttribute = "id")
pattern functionId(func : Function, id : EString) {
  find functionSubsystem(subsys);
  Block.name(subsys ,id);
  find trace(subsys , func); }

pattern inPortToInPortConn(src: InPort , trg: InPort) {
  Port.portBlock.outports.signals.to(src , trg); } or {
  Port.portBlock.inports(outer, src);
  OutPort.signals.to(outer, trg); }

pattern connectedBlocks(prov : Block , cons : Block)
  Block.outports.signals.to(prov , consumerInPort);
  Block.inports(cons , consumerInPort); } or {
  Block.outports.signals.to(prov , innerIP);
  find inPortToInPortConnection+(innerIP , consumerInPort);
  Block.inports(cons, consumerInPort); }

@Edge(source = prov, target = cons, eReference = "consumers")
pattern link(prov : Function , cons : Function) {
  find functionSubsystem(p);
  find connectedBlocks(p,c);
  find functionSubsystem(c);
  find trace(p, provider); find trace(c, consumer); }
```

Listing 1. Model queries and derivation rules

The presented pattern language has the ability to add additional information to patterns with different annotations appearing on the top of specific patterns started with "@" character. Instead of relying on a full-fledged model transformation language, I extended the EMF-IncQuery framework to interpret the annotations as derivation rules. As a result, we obtain a declarative formalism compliant with the execution semantics of incremental and non-deleting graph transformations while the language engineering environment of EMF-IncQuery requires very little further adaptation. A derivation rule is obtained when a query is extended by one or more of the following annotations:

a) @Item(eClass = "ClassName"):

The `Item` annotation is used to add a new view object of type "ClassName" from target metamodel.

b) @Edge(source = src, target = trg, eReference = "ref"):

The `Edge` annotation is used to set a reference of source to the value of target.

c) @Attribute(target = trg, value = val, eAttribute = "attr"):

The `Attribute` annotation is used to set an attribute of target to the value of val.

The patterns from Listing 1. have derivation annotations defined that are to be read as follows. The derivation rule of function prescribes the derivation of a view object of type `Function` in the FAM together with a traceability link. The `functionId` sets the id attribute of the created object to the name of the corresponding Simulink block (passed by the pattern parameter identifier). Finally, the edge annotation of link pattern connects two function elements in the view-model. Applying these rules on the SIM model of Figure 1. , we obtain all the `Function` nodes of the FAM model and the appropriate `consumers` references between them. (Because of the EMF-specific *EOpposite* reference, the `providers` reference is also set automatically.)

V. POPULATE VIEWMODELS WITH DERIVED OBJECTS

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 [1] that supports incremental query evaluation over EMF instance models by constructing a Rete rule network [9] that processes change notifications sent by the EMF notification API. It caches the current query results and propagates changes of them to the Viewers layer of the framework.

Even though the construction and storage of Rete-network 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.

In this approach, the previously mentioned derivation rules are interpreted as follows when they are fired if a new match of the annotated patterns is appeared or disappeared:

a) @Item:

Appeared: an `EObject` is created with the specified type defined in the annotation and `Trace` object is also appeared where its target is the new object and its sources are the parameters of the match.

Disappeared: when a match disappears that previously defined an object in the view-model, it is possible to find the target object through the traceability model and delete it from the view-model.

b) @Edge:

Appeared: the annotation defines which parameters of the match act as source and target objects. In this case, the framework decides whether the multiplicity of reference is many or single and according to this, adds or sets the target of8 the reference.

Disappeared: after a match disappears, the framework decides again whether the multiplicity of the reference is many or single and according to this, removes the target to

from the reference or sets its value to the default (which usually means the "null" value).

c) *@Attribute:*
Appeared: the annotation defines which parameters of the match act as target object and value. In this case, the framework tries to convert the value to the corresponding type and then it sets to the specified attribute.

Disappeared: after a match disappears, the framework sets the value of the specified attribute to the default one.

Because the edge and attribute typed derivation rules refer to derived objects in view-models, the item typed derivation rules are fired at the highest priority. If two or more item typed or edge/attribute typed derivation rule can be fired, arbitrary order can be defined. This means that two derivation rules that can fired at the same time are never in conflict.

In our running example, the view-model is built up in the following sequence (depicted in Figure 4.): (1) a new match appears for the `function` pattern. (2) A new `Trace` object is created in traceability model and (3) a new `Function` element is added to view-model. When the necessary functions exist in the view-model (4) new matches appear for the `functionId` pattern and (5) the corresponding id values of the specified functions are set according to the matches.

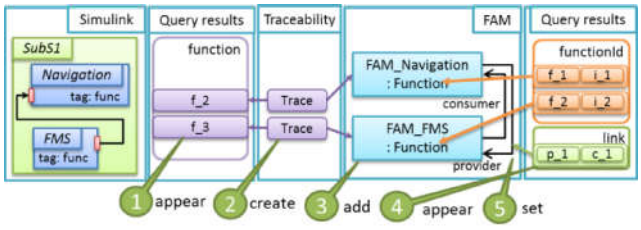

Figure 4. Internal traceability and working mechanism

VI. Related work

Defining view-models and synchronizing them with the sources is a common problem in MDE. Using derived classes and model queries captured in OCL was first proposed in [2] but without support for incremental view calculation. This solution can be further developed using recent results in incremental support of OCL topic as in [3][4].

View maintenance by incremental and live QVT transformations is used in [5] 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 [7] an ATL-based method is presented while in [8] correspondences between models are handled by matching rules defined in ECL, where the guards use queries similarly to our approach, although incremental derivation is not discussed.

I believed that this contribution is unique in incremental synchronization of view-models that can be used to create automatic abstraction from detailed models.

VII. Conclusion and future work

In the paper, an approach is presented for deriving and incrementally synchronizing non persistent view model. To achieve this functionality with the help of the EMF-IncQuery team, I extended the framework to (i) support derivation rule definitions by annotations, (ii) provide an implicit traceability between the source and view model elements and (iii) manage derivation rule execution based on the changes of their match set (disappear, appear).

A main direction for future work, I am planning to support (i) the chaining of view-models along derivation rules and (ii) improve the semantics of derivation rules to define more complex queries. Finally, to provide a deeper evaluation of the scalability of the approach as currently it was only tested on models with no more than 1000 elements.

References

[1] Ujhelyi, Z., Bergmann, G., Hegedüs, A., Horváth, A., Izsó, B., R´ ath, I., Szatmári,Z., Varró, D.: EMF-IncQuery: An integrated development environment for livemodel queries. Science of Computer Programming (2014) In press.

[2] Balsters, H.: Modelling database views with derived classes in the UML/OCL-framework. In Stevens, P., Whittle, J., Booch, G., eds.: UML 2003 - The UnifiedModeling Language. Modeling Languages and Applications. Volume 2863 of LNCS.Springer Berlin / Heidelberg (2003) 295–309

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

[4] Groher, I., Reder, A., Egyed, A.: Incremental consistency checking of dynamic constraints. In: FASE 2009. Volume 6013 of LNCS., Springer (2010)

[5] Song, H., Huang, G., Chauvel, F., Zhang, W., Sun, Y., Shao, W., Mei, H.: Instant and incremental QVT transformation for runtime models. In Whittle, J., Clark, T., Khne, T., eds.: Model Driven Engineering Languages and Systems. Volume 6981 of LNCS., Springer Berlin Heidelberg (2011) 273–288

[6] Clasen, C., Jouault, F., Cabot, J.: Virtual Composition of EMF Models. In: 7` emes Journ´ ees sur l'lng´ enierie Dirig´ ee par les Mod` eles (IDM 2011), Lille, France (2011)

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

[8] Kolovos, D.S.: Establishing correspondences between models with the epsilon com-parison language. In: Proceedings of the 5th European Conference on Model DrivenArchitecture - Foundations and Applications. ECMDA-FA '09, Berlin, Heidelberg, Springer-Verlag (2009) 146–157

[9] Charles L. Forgy. A network match routine for production systems. Working paper, 1974.

[10] Zoltán Ujhelyi, Tamás Szabó, István Ráth, and Dániel Varró. Developing and visualizing live model queries. In Proceedings of the First Workshop on the Analysis of Model Transformations (AMT '12). ACM (2012.), New York, NY, USA, 35-40.

[11] EMF-IncQuery, https://www.eclipse.org/incquery/

[12] Eclipse Modeling Framework, https://www.eclipse.org/modeling/emf