Softw Syst Model
DOI 10.1007/s10270-014-0436-y

REGULAR PAPER

Query-driven soft traceability links for models

Ábel Hegedüs · Ákos Horváth · István Ráth · Rodrigo Rizzi Starr · Dániel Varró

Received: 9 June 2013 / Revised: 17 September 2014 / Accepted: 19 September 2014

Abstract Model repositories play a central role in the model driven development of complex software-intensive systems by offering means to persist and manipulate models obtained from heterogeneous languages and tools. Complex models can be assembled by interconnecting model fragments by hard links, i.e., regular references, where the target end points to external resources using storage-specific identifiers. This approach, in certain application scenarios, may prove to be a too rigid and error prone way of interlinking models. As a flexible alternative, we propose to combine derived features with advanced incremental model queries as means for soft interlinking of model elements residing in different model resources. These soft links can be calculated on-demand with graceful handling for temporarily unresolved references. In the background, the links are maintained efficiently and flexibly by using incremental model query evaluation. The approach is applicable to modeling environments or even property graphs for representing query results as first-class relations, which also allows the chaining of soft links that is useful for modular applications. The approach is evaluated using the Eclipse Modeling Framework (EMF) and EMF- INCQUERY in two complex industrial case studies. The first case study is motivated by a knowledge management project from the financial domain, involving a complex interlinked structure of concept and business process models. The second case study is set in the avionics domain with strict traceability requirements enforced by certification standards (DO-178b). It consists of multiple domain models describing the allocation scenario of software functions to hardware components.

Keywords Soft links · Incremental model queries · Derived features · Traceability

Communicated by Prof. Perdita Stevens.

This work was partially supported by the CERTIMOT (ERC_HU-09-01-2010-0003) and the Trans-IMA (industrial) projects, the TÁMOP (4.2.2.B-10/1-2010-0009) Grant and the János Bolyai Scholarship.

Á. Hegedüs (✉) · Á. Horváth · I. Ráth · D. Varró
Department of Measurement and Information Systems,
Budapest University of Technology and Economics,
Magyar tudósok krt. 2., Budapest 1117, Hungary
e-mail: hegedusa@mit.bme.hu

Á. Horváth
e-mail: ahorvath@mit.bme.hu

I. Ráth
e-mail: rath@mit.bme.hu

D. Varró
e-mail: varro@mit.bme.hu

R. R. Starr
Embraer S.A. Av Brigadeiro Faria Lima,
São José dos Campos 2170, Brazil
e-mail: rodrigo.starr@embraer.com.br

1 Introduction

Modeling frameworks serve as underlying model management infrastructure for various industrial development tools, especially in the avionics and automotive domain. These domains necessitate the handling of large models with potentially millions of model elements. For maintainability and scalability reasons, such models are not persisted in a single document, but stored as an interconnected network of model fragments where each fragment stores a certain part of the entire system model. In other application scenarios, models are complemented with external traceability models to explicitly persist traceability links between requirements models, design models, analysis models or source code,

Published online: 08 October 2014

for instance. In both scenarios, these models are frequently manipulated by several development or verification tools in complex toolchains operated by different design teams.

Unfortunately, the interconnection of complex system models imposes several technical problems due to the identification strategies of model elements in these modeling frameworks. When serializing a model, a model element is either identified by a unique identifier generated by the framework, or by a relative path of containment hierarchy in the given model fragment. These techniques are used when interconnecting models using associations, e.g., for *internal traceability* purposes [66]: the target end of the association points to an object resided in a different fragment. Such interconnections are also used in *external traceability* scenarios [31,33,36] where inter-model links are introduced from traceability metamodel elements to existing metamodels which cannot be altered.

These scenarios demonstrate various shortcomings of modeling frameworks. First, (1) the serialization (persistent storage) of interconnected model fragments with circular dependencies along associations is often not supported. Furthermore, without highly intelligent multi-fragment transaction management, (2) local changes in a model fragment may introduce broken links unless all dependent model fragments are manipulated together in working memory. Such broken links require tool-specific resolutions—with a worst case scenario of fixing the links manually by the designer using text editors (and not the modeling tool). Finally, (3) all traceability links captured by associations are explicitly persisted every time even if traceability links could be derived from existing unique identifiers.

Extending on our MODELS 2012 paper [27], we provide an approach for the soft interconnection of models based on *derived references* and *incremental model queries*. Derived references are attributes and relations of the model calculated at runtime, and their values are not stored explicitly. When using derived relations, the corresponding links only exist after the models are loaded. Therefore, model fragments can be (de)serialized in arbitrary order and warnings can be issued about broken links when certain resources are unavailable or not loaded. In order to provide an efficient and flexible handling of such soft links, we use fully incremental model query evaluation as a technical foundation. As a result, it is sufficient to identify a model element by a query evaluation instead of local or global identifiers, and less amount of information needs to be persisted for traceability purposes.

Soft interconnections can be used in query evaluation similarly to regular links; therefore, our approach also supports chaining of soft links. This allows the integration of additional model fragments to an existing application, which is important in modular model-driven software development.

We selected the EMF [53] as a modeling foundation and used the EMF- IncQuery query evaluation framework [65] for providing soft link functionality.[1] EMF is a de facto modeling standard with wide-spread use in industrial applications, while EMF- IncQuery is an open-source Eclipse project.[2]

Similar challenges, such as fragmentation of data, heterogeneous storage and distributed access are relevant in the field of databases as well. In particular, materialized views [25] defined by queries are conceptually similar to soft interconnections. Furthermore, federated (virtual) database systems can logically interconnect distinct shards by executing queries transparently over the distributed table space [28]. Consequently, our approach is based on the adaptation of virtual databases and materialized views to (graph-based) models and model query languages.

We initially developed our approach in the context of a industrial research project concerning business process modeling. The first case study of the paper (Sect. 2.1) uses the domains of the project. We have also successfully applied the approach in another industrial project in developing a complex, model-driven application in the avionics domain. Our second case study (Sect. 2.2) demonstrates how soft links were applied in this application. While the inter-model connections in the case studies are listed in this section, a high-level introduction to the concepts of soft links is provided in Sect. 3 using derived features defined by model queries.

The definition and use of inter-model soft links is discussed in Sect. 4, while soft links are applied in the context of external traceability scenarios in Sect. 5. The details for providing incremental computation and efficient maintenance of soft links are intentionally deferred until Sect. 6, which proposes an architecture and algorithms in the context of EMF models. Finally, related approaches and tools are described in Sects. 7 and 8 concludes our paper.

2 Motivation: interconnected model fragments

2.1 Modeling and managing business processes

Our first case study was conducted on a business process modeling (BPM) scenario based on a project carried out together with an industrial partner. While the actual metamodels (shown in Fig. 1) are significantly simplified here to provide better focus, they still demonstrate many practical industrial problems of interconnecting models. In the case study, semi-automatic workflows (captured as a *process model*) contain both automated and manual tasks. Architecture-level deployment decisions are captured by a

[1] Fully implemented and documented at http://incquery.net/incquery/new/examples/query-driven-soft-links.

[2] The authors affiliated with Budapest University of Technology and Economics are also committers of the project.

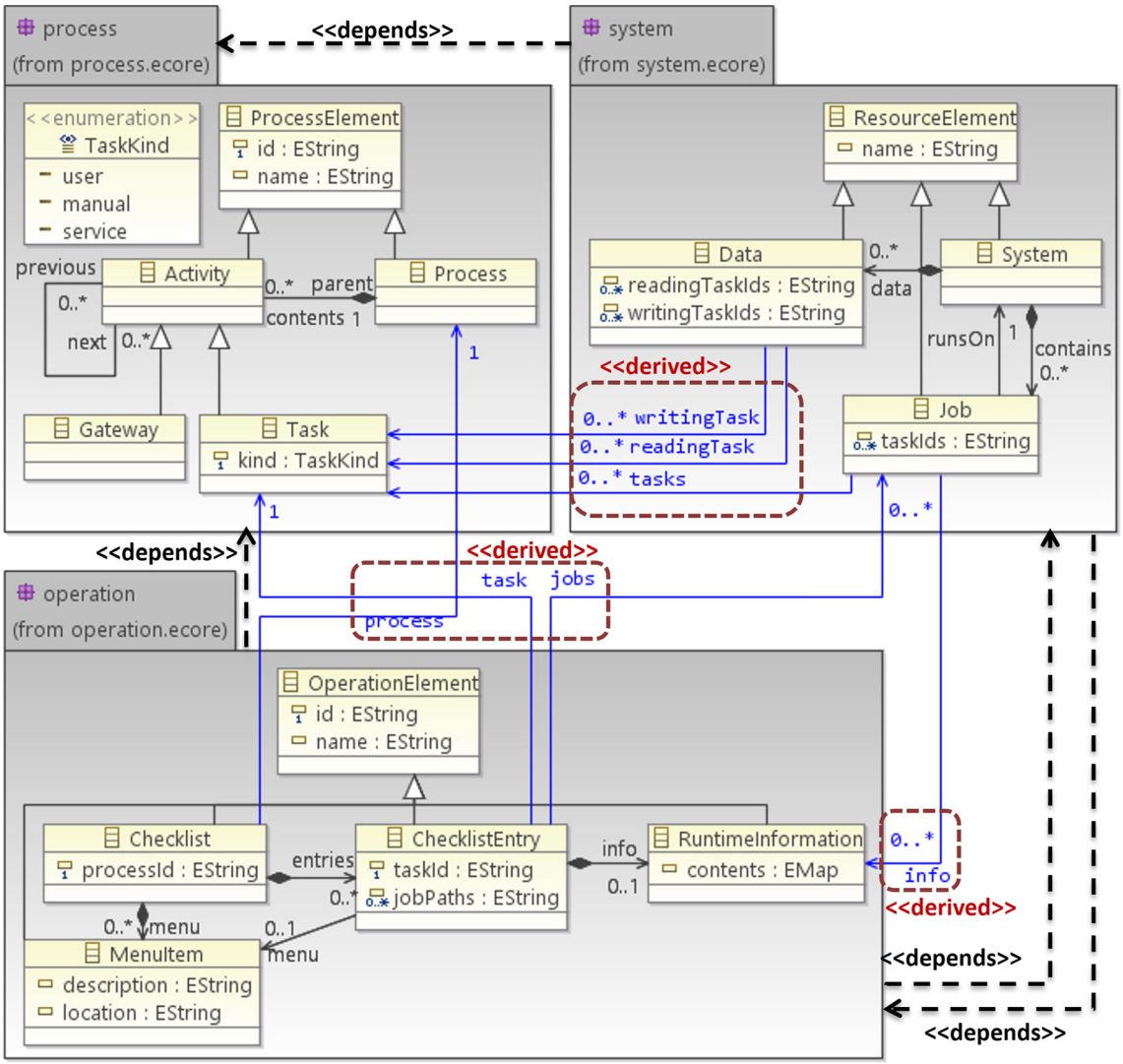


Fig. 1 The metamodels of the BPM case study

separate *system architecture model* comprising of jobs and data resources referring to tasks in the business process model. Finally, the instances of the processes managed by operators are captured in an *operation* model containing a checklist for each process with task entries assigned for each operator.

2.1.1 Overview of BPM metamodels

Business process metamodel. Business processes (process package) are defined by a fragment of the standard XPDL [73] metamodel. The ProcessElement top-level class defines id (unique identifiers) and name attributes for each element. A Process includes Activities that are either Tasks (atomic workflow steps) or Gateways (e.g., fork-join, decision, loops), while the control flow of the process is represented by the next and previous relations between activities. Based on the value of the kind attribute, tasks can be service (for automated execution through API calls), manual (where the operator initiates some job) and user (when the task itself is performed by an operator or other assigned personnel).

System architecture metamodel. The system architecture metamodel defines a top-level ResourceElement that defines a name for each element. This simplified architecture includes Systems (representing larger components), Data elements that represent application data (e.g., configuration, input or output files) that can be read or written during the execution of tasks in the processes and Jobs (e.g., scripts or one-shot programs) that run on Systems. We assume that each system must have a unique name and each job contained in the same system must have different names. Otherwise, names are not globally unique in this domain.

Operation metamodel. The operation metamodel is used for representing Checklists followed by operators when per-

forming the manual tasks in processes. The top-level OperationElement adds a name and a unique identifier for each model element. Each Checklist is related to a Process and includes a number of entries and a menu. The menu contains MenuItems that have textual descriptions and a location, where the operator can access it. The entries are ChecklistEntry elements, each corresponding to one task, an arbitrary number of jobs, and optionally to a MenuItem. Finally, each entry can contain further information (e.g., historical statistics or requirements) stored by a RuntimeInformation element using a content map.

Inter-model connections. These metamodels and thus the corresponding model instances heavily depend upon each other (see Fig. 1). The following logical interconnections are present in our example:

- Job can be linked to a Task to represent that the task includes the execution of the job.
- Process is a referenced from Checklist to indicate that the process includes the tasks that will be performed by going through the checklist
- ChecklistEntry links to both to a process Task and a system Job to indicate that the entry is active when the given task is performed and it involves the execution of the listed jobs.
- RuntimeInformation can be attached to a Job if the job is executed as part of the entry containing the information.

Many industrial tools (including the TIBCO Business Studio [62] used in our industrial project for capturing XPDL models and the AUTOSAR standard [2]) store identifiers of external (inter-model) elements using (a list of) simple string (or integer) attributes. In contrast, modeling frameworks often use references (corresponding to lazily initialized inter-object pointers) to interconnect different models (or model fragments), which are created by lazy loading upon first use.

2.2 Hardware–software allocation in Integrated Modular Avionics (IMA)

Our case study in IMA is based on an ongoing, cooperation project with a major airframer. It aims at defining a model-driven approach for the synthesis of complex, integrated Matlab Simulink [38] models (implementation model) capable of simulating the software and hardware architecture of an airplane.

The high-level overview of the IMA application workflow is depicted in Fig. 2.

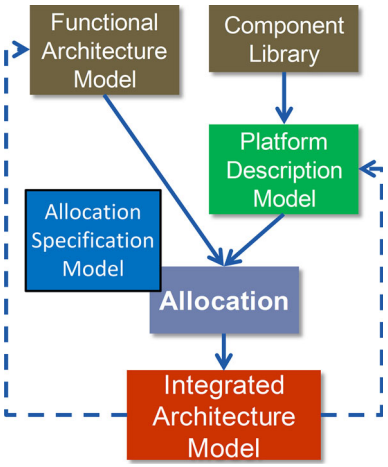


Fig. 2 High-level overview of the IMA application workflow

- The *input artifacts* for the process are the Functional Architecture Model (FAM) and the Component Library (CL) defined in Matlab Simulink as a library and a system, respectively. The goal of the FAM is to capture the description of the different software functions that are allocated to the IMA system under design, while the CL defines the available hardware elements that can be used for the definition of the HW architecture of the aircraft. These Simulink models are imported into the IMA design tool.
- Using these inputs during the development first the system architect specifies the Platform Description Model (PDM) based on the building blocks (e.g., computational unit, router, chassis, power unit, etc) defined in the Component library (e.g., a specific vendor's processor or power unit, etc).
- In the next step the system architect allocates the functions from the FAM to the underlying PDM. The *allocation* itself includes two major parts: (i) the mapping of functions defined in the FAM to underlying execution elements within the PDM and (ii) the automated discovery of available communication paths for the various information links defined between the FAM elements. The mappings and the communication paths are stored in a Allocation Specification Model.
- Finally, when the allocation is complete and fulfils the structural requirements, the Integrated Architecture Model is automatically synthesised and ready to be simulated in Simulink.

In order to support the envisioned model-driven IMA development framework, we defined the modeling architecture with the following goals in mind: (i) define a *simulation tool (vendor) independent modeling layer* for both the functional and hardware architecture, (ii) ensure *instance model reuseability* in the allocation process to support the allocation of a single FAM to multiple PDMs and (iii) provide a *complete traceability support* through all the models used in the development process.

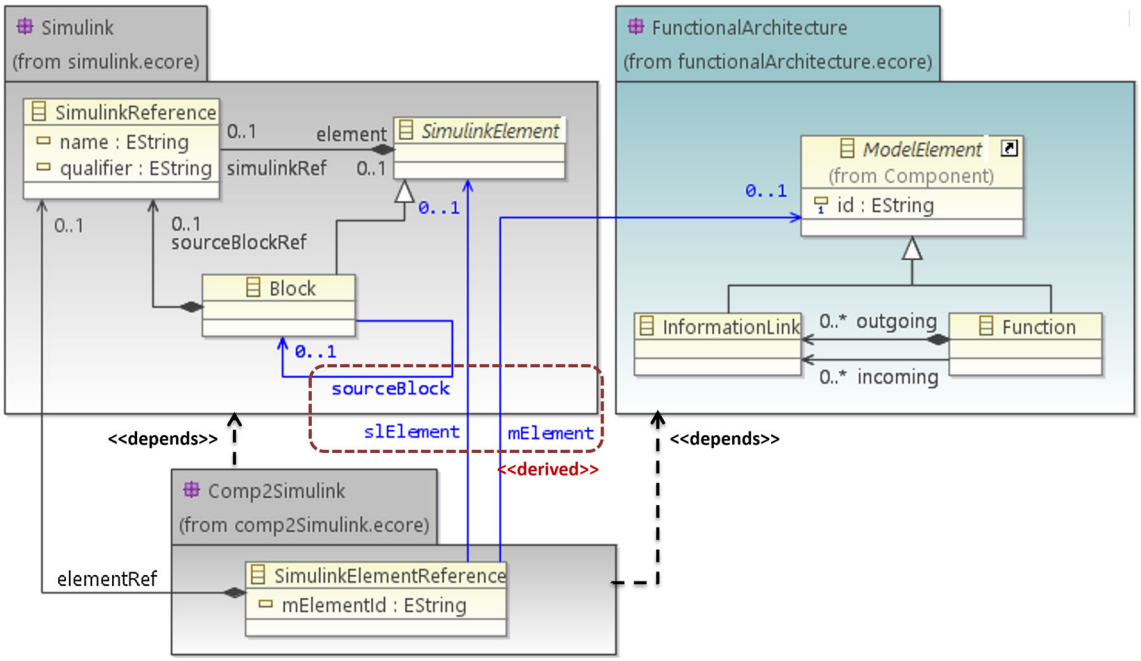


Fig. 3 The metamodels of the IMA case study

2.2.1 Overview of the IMA metamodels

The complete modeling architecture contains more than 140 classes and 200 references in seven metamodels. For demonstrating how we applied soft links in this case study, we selected an extract of the metamodels that shows how changes in either a source or target domain model can be handled without specific manipulations on references between the domain models. This fragment in Fig. 3 highlights traceability-specific challenges from the *Comp2Simulink*, *Simulink* and *FAM* metamodels.

Functional Architecture metamodel. A Function represents a functionality provided by the aircraft that is implemented either in software or hardware, while functions communicate with other functions through InformationLinks. Both are subclasses of the ModelElement class that is the superclass of all domain metamodel elements.

Simulink metamodel. All Simulink specific elements are subclasss of the SimulinkElement class, while a Block represents a Simulink block, whose sourceBlock reference points to its defining Simulink library element. Finally, the SimulinkReference class is a unique identifier for a SimulinkElement defined by a fully qualified name (FQN) constructed from a name and qualifier.

Integration metamodel. The Comp2Simulink package defines a platform-specific mapping layer between domain-specific (FAM in the current case) and Simulink models. Its SimulinkElementReference element captures which Simulink element (slElement) is represented by a certain domain-specific element (mElement).

Inter-model connections. Using this structure, we are able to handle two problems that are characteristic to this domain:

- *Handling (Simulink) libraries.* Libraries in Simulink contain re-usable blocks which can be inserted into models similarly to regular blocks. Blocks inserted from a library refer back to the library element, which allows changes in library elements to be propagated to all models that use the library. The sourceBlock soft link can flexibly handle changes if an other version of the library is loaded into the IMA tooling. For example, it is possible to have alternative libraries for scheduler simulation or failover/redundancy evaluation, where only the internal structure of the library elements are different.
- *Traceability between domain and implementation models.* During the allocation, the traceability between the FAM and its corresponding Simulink models has to be consistent. In certain situations, the underlying Simulink models must be changed without affecting its corresponding FAM (e.g., by redefining the internal structure of a function). However, when the changes are made in Matlab and the modified Simulink model is imported into the IMA tooling, the traceability links that connect the Simulink elements (slElement) to their corresponding FAM elements (mElement) should be automatically repaired without changing any FAM elements.

These problems are solved using soft links defined by queries as described in Sects. 4.4 and 5.2.

3 Toward soft traceability links

In the paper, we propose soft links for interconnecting models by combining derived features and incremental model queries. The term "query-driven soft links" refers to the fact that (1) certain model interconnections only exist at runtime and are not persisted explicitly when serializing instance models, while (2) the interconnected model elements can be accessed and navigated along derived features. Furthermore, (3) our query-driven technique allows to define complex, n-ary interconnections of several model elements, and (4) to identify model elements dynamically based on query results (instead of static unique element identifiers).

3.1 Definition of soft links

In this section, we define a formal foundations of soft links as computed interconnections between model fragments through basic formalization using graphs.

An *(edge-)labeled, directed graph* is defined as a structure $G = (N, E, L, src, trg, label)$ where N is the set of *nodes*, E is the set of *edges* and L is the set of *labels*. The functions $src, trg : E \to N$ map the edges to their source and target nodes, respectively. Finally, the function $label : E \to L$ maps the edges to their labels. We denote the values of these functions for a given edge e by $e.src$, $e.trg$, $e.lbl$.

Note that type information of nodes can be represented by labels on self-loop edges. For the sake of simplicity, we refer to the name of such labels as the type of the element but we do not define here a precise type mapping.

The graph representation can be mapped to specific modeling frameworks, such as industrial tools MetaEdit+ [63], EMF, Groove [46] or RDF/triple stores [44] and research tools Clafer [3], MetaDepth [35] or VPM [68].

Next, a *model repository MR* is a graph that satisfies the following:

- $R \subseteq N$ is a set of *resource nodes*, representing model fragments. Resource nodes are identified by res labels.
- For each node $n \in N \backslash R$, there is exactly one edge $e \in E$, where $e.src = n$ and $e.trg \in R$. We refer to $e.trg$ as the *owner* of n and denote it by $n.own$.
- $T \subseteq E$ is a set of edges called *trace links* that contains all edges l with different owners of its source and target nodes (i.e., $l.src.own \neq l.trg.own$).

Example 1 Figure 4 illustrates a model repository consisting of three resource nodes $Proc$, Sys and Op (representing a process, a system architecture and an operation model from the BPM case study in Sect. 2.1) with trace links represented with dashed arrows. For example, the nodes owned by $Proc$ are p, t_1, t_2, the source of edge c is p and its target is t_1, while the trace link l_p connects a node owned by Op to a node owned by $Proc$. Note that we denote labels representing types on Fig. 4 by the type name added after the name, e.g., p:Process for node p representing a process.

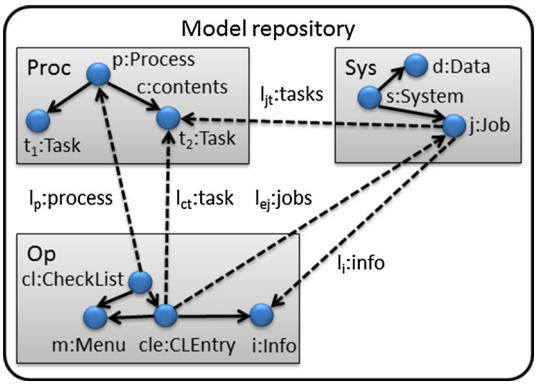


Fig. 4 Example model repository with trace links

A *query* is a tuple $q = (\overrightarrow{P}, b)$ where $\overrightarrow{P}$ is a set of named parameters and b is the body that describes what structural constraints have to be satisfied. A *match* $m : q \mapsto MR$ of a query over a model repository MR is a set of parameter assignments that satisfy the query constraints, where each assignment maps a parameter to its value, e.g., parameter p_i to node n_i in MR denoted by $m.p_i := n_i$. We denote a given match by listing the assigned values in the order of the parameters, e.g., $\{n_1, n_2, \ldots n_k\}$ for a query with k parameters. Finally, the set of matches of query q over a model repository is denoted by $M_q = \{m | m : q \mapsto MR\}$.

Example 2 A sample query may select "*each node* p_1 *with two outgoing edges where all incoming edges are trace links*" and the single parameter is p_1, then the matches of the query in Fig. 4 are $\{p\}$, $\{cl\}$ and $\{s\}$.

Note that we intentionally do not specify how the body of a query is described and how matches of a query are found in the model repository. Possible approaches include graph patterns [22] (as used later in the current paper) or OCL expressions [60]. Our motivation with this generality is to emphasize that soft links can be adopted using modeling frameworks and query techniques different from those exemplified in this paper.

Let $q_{sl} = (\overrightarrow{P}, b)$ be a model query with two parameters $\overrightarrow{P} = \{Src, Trg\}$. We can regard the matches of q_{sl} as edges between two nodes in the model repository and these edges are *calculated based on the existing nodes and edges*.

A *soft link* is an edge l which exists in a model repository if and only if a model query q_{sl} has a match with parameter assignments $\{l.src, l.trg\}$. If edge l is computed by the query q_{sl}, the type of the edge is also sl.

Example 3 In the model repository in Fig. 4, we can define the query q_{info} as "*node* p_1 *of type* Job *with incoming edge from a node of type* ChecklistEntry *that also has outgoing edge to node* p_2 *of type* RuntimeInformation". The query has a match $\{j, i\}$ and the trace link l_i can be created as a soft link typed *info*.

In order to apply the concept of soft links on existing metamodeling and model query evaluation approaches, we selected the EMF for representing the model repository and EMF- IncQuery for specifying and evaluating model queries. However, other modeling ecosystems and query evaluation techniques can also be used for the definition and application of such interconnections.

3.2 Model repository in Eclipse Modeling Framework (EMF)

The EMF is a Java framework and code generation facility for building tools and other applications based on a structured model. EMF provides a metamodel (Ecore) for describing structured models. Using these structured models, EMF provides tools and runtime support to produce a set of Java classes representing the model in Java Virtual Machine.

The model repository concept in the EMF is represented by a *resource set*, which contains a set of *resources* that are model fragments persisted in separate locations. Each EMF resource is a resource node in the model repository.

The nodes owned by resources are *EObjects* (typed model elements) or *datatype values* (numbers, character strings, boolean values etc.), while edges are associations between nodes called *settings*. The labels that represent the types are specified by *Ecore models*, which contain *EClass* labels applicable on EObjects, *EDataType* labels for datatypes and *EStructuralFeature* (feature) labels for settings. In addition, EStructuralFeatures are distinguished as *EReference* for settings between EObjects and *EAttribute* for settings with EObject source nodes and datatype value target nodes.

Derived features in EMF models represent computed information which can be calculated from other model elements. Essentially, we distinguish between *derived attributes*, which represent settings to datatype values and *derived references*, which represent "virtual" interconnections between EObjects (represented graphically by the derived stereotype in Figs. 1, 3).

In this paper, we show how incremental model queries can be combined with derived features to achieve true soft traceability links for models. Derived features for soft links will be defined by using a declarative, high-level graph-based query language (Sect. 4) and evaluated with fully incremental pattern matching [5] (Sect. 6.1) as offered by the advanced model query framework EMF- IncQuery [65] or available for OCL expressions [12].

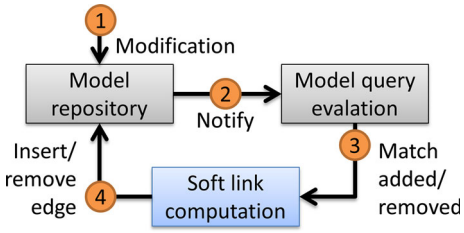


Fig. 5 Soft link life cycle

Incremental evaluation of model queries over EMF models depends on the *change notification* mechanism that is provided by EMF for regular (non-derived) features. Applications (e.g., the query evaluation engine) can observe EObjects and receive notifications any time the value of a feature is modified. Since such notifications are required to provide efficient incremental evaluation, soft links represented by derived features also send change notifications when their value is modified.

3.3 Life cycle of soft links

Figure 5 shows the overview of the life cycle of soft links illustrating how a modification on the model repository (1) leads to changes in trace links. When the model repository is modified, it notifies the model query evaluation about the change (2) and if the change affects the matches of the query, then the added or removed match is sent to the soft link computation (3).

Finally, the edge representing the soft link is added to or removed from the model repository (4). In Sect. 6.2, we describe in detail how incremental query evaluation is used to provide this life cycle for interconnected models.

3.4 Advantages of soft traceability links

Our soft interconnection technique driven by incremental query evaluation offers the following advantages, which are also aligned with the research roadmap for achieving scalability in model driven engineering [34]:

- *Handling circular trace links.* Circular trace links between model fragments can be handled easily with soft links. For instance, in the BPM case study, the models system and operation (see Fig. 4) are mutually connected to each other along the trace links jobs and info. This circularity prevents serialization using regular methods where repeated traversal of the same resource is not allowed. As soft links are not serialized, this problem no longer occurs.
- *Graceful management of broken links.* When models are manipulated by multiple, independent tools, inter-model

links can be easily broken, which result in runtime problems when the corresponding node is attempted to be accessed along a broken link. Soft links provide graceful behavior in case of broken links by issuing warnings in case of unresolved elements. Such warnings are issued based on the incremental computation as well, and broken links are corrected automatically if the model containing the target node is loaded later.

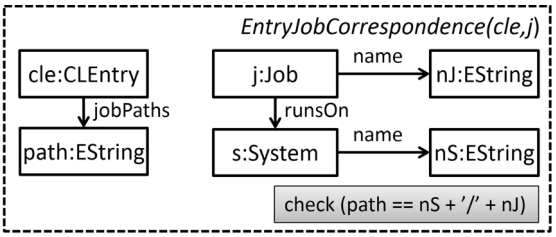
– *Improved persistence.* Whenever a model interconnection can be calculated by a query, it does not have to be explicitly persisted into traceability models. As a result, the execution time of persistence operations for complex interconnected models can be reduced.
– *Support for soft link chaining.* Soft links can be defined using existing soft links when integrating additional model fragments. Such chaining can be useful for modular model-driven software development.
– *High performance.* Due to the incremental query evaluation [7], trace links can be reevaluated very efficiently even in case of complex definitions (e.g., transitive closures [9]). As a result, the maintenance of soft links can be efficient with low memory overhead even for large models with complex traceability structures (Sect. 6.6).

4 Specification of interconnections with model queries

In order to support the definition of soft interconnections between models, a graph pattern-based model query language [8] is used as the specification language. Therefore, a brief introduction to this query language is provided first (Sect. 4.1), followed by a detailed description on how this general purpose query language is adapted to specify soft interconnections for the case studies (Sects. 4.2, 4.3). Finally, we illustrate their use on handling libraries of Simulink blocks in Sect. 4.4.

4.1 Model queries by graph patterns: an overview

Graph patterns [67] are an expressive formalism used for various purposes in model-driven development, such as defining declarative model transformation rules, capturing general purpose model queries including model validation constraints, or defining the behavioral semantics of dynamic domain-specific languages. A graph pattern (GP) represents conditions (or constraints) that have to be fulfilled by a part of the model. A basic graph pattern consists of *structural constraints* prescribing the existence of nodes and edges of a given type, as well as *expressions* to define *attribute constraints*. A *negative application condition* (NAC) defines cases when the original pattern is *not* valid (even if all other constraints are met), in the form




```
1 // ChecklistEntry.jobs link
2 pattern EntryJobCorrespondence(cle, j) {
3   Job.name(j,nJ);
4   System.name(s,nS);
5   Job.runsOn(j,s);
6   ChecklistEntry.jobPaths(cle,path);
7   check(path == nS+'/'+nJ);}
```

Fig. 6 Model query to define EntryJobCorrespondence in graphical and textual syntax

of a negative sub-pattern. A match of a graph pattern is a set of model elements that have the exact same configuration as the pattern, satisfying all positive, but no negative constraints.

4.2 Soft links by derived features and model queries

4.2.1 Sample soft link

First, we demonstrate on an example how the graph pattern EntryJobCorrespondence(cle,j) (Fig. 6) can be used to express the soft links captured by the inter-model connection jobs (connecting *ChecklistEntry* and *Job* in Fig. 1), that is, to identify those jobs that correspond to a task execution as part of the checklist entry.

This model query formulated as a graph pattern has two parameters: cle and j, denoting the source and the target end of the soft link. The query defines the designated set of jobs by checking the names of the given job element j and the system s it *runs on* (nJ and nS, respectively) and the $path$ stored in the entry. Model queries for the other soft links defined in the BPM cse study are defined similarly in Listings 1 and 2.

```
1  // Job.tasks link
2  pattern JobTaskCorrespondence(j,t){
3    Task.id(t,tId);
4    Job.taskIds(j,tId);}
5  // Data.readingTasks link
6  pattern DataTaskReadCorrespondence(d,t) {
7    Task.id(t,tId);
8    Data.readingTaskIds(d,tId);}
9  // Data.writingTasks link
10 pattern DataTaskWriteCorrespondence(d,t) {
11   Task.id(t,tId);
12   Data.writingTaskIds(d,tId);}
```

Listing 1 Resource-Process mapping

The query language also supports the following language constructs:

```
// Job.info link
pattern JobInfoCorrespondence(j,i) {
  ChecklistEntry.info(cle,i);
  RuntimeInformation.id(i,iId);
  find EntryJobCorrespondence(cle, j);}
// ChecklistEntry.task link
pattern EntryTaskCorrespondence(cle, t) {
  Task.id(t, tId);
  ChecklistEntry.taskId(cle,tId);}
// Checklist.process link
pattern ListProcessCorrespondence(cl,pr){
  Process.id(pr,prId);
  Checklist.processId(cl,prId);}
```

Listing 2 Checklist entry mapping

- check(path == nS + '/' + nJ) checks that the model element bound to variable *path* is equal to the concatenated value of nS and nJ.
- Using the find keyword, graph patterns can be composed by reusing other graph patterns. Therefore, if a soft link is defined as a model query by a graph pattern, this definition can be reused in other queries, and thus, in other soft links.

Soft links can be defined as model queries using this highly expressive graph pattern-based language [8]. Graph patterns with embedded NACs provide expression power equal to first-order logic [45], while advanced features (e.g., transitive closure [9], match counting and evaluation of expressions) further increase the usability of the language.[3] The queries are used as specifications for the soft links in the following way:

1. *Each query should have exactly two parameters* Each match of the query represents a soft link in the model. The type of the soft link is specified by the query of the match; therefore, only the source and target nodes are needed as parameters.
2. *Value of first parameter: source node* The value assigned for the first parameter (e.g., *cle* in Fig. 6) is the source node of the link.
3. *Value of second parameter: target node* The value assigned for the second parameter (e.g., *j* in Fig. 6) is the target node of the link.

4.3 Complex soft links in the IMA case study

Two of the soft links in the IMA case study use the SimulinkReference objects for identifying the target of the interconnection. In order to compare two reference objects, we introduce a helper pattern referenceEqual (see Listing 3) that will match the references sourceRef and targetRef if the name and qualifier of both elements are equal.

[3] For comparison of the query language with other formalisms, see [6,8].

```
pattern referenceQualifier(sr, qual) {
    SimulinkReference.qualifier(sr,qual);
}
// name and qualifier of a SimulinkReference
pattern simulinkReference(sr, name, qual) {
    SimulinkReference.name(sr,name);
    find referenceQualifier(sr,qual);
}
// name and qualifier same for two references
pattern referenceEqual(sourceRef, targetRef) {
    find simulinkReference(sourceRef, name, qual);
    find simulinkReference(targetRef, name, qual);
} or {
    // no qualifier
    SimulinkReference.name(sourceRef,name);
    SimulinkReference.name(targetRef,name);
    neg find referenceQualifier(sourceRef, _squal);
    neg find referenceQualifier(targetRef, _tqual);
}
```

Listing 3 Pattern for comparing Simulink references

The pattern referenceEqual that compares Simulink references includes two additional language constructs for specifying alternative pattern bodies and negative application conditions. The pattern has two alternative bodies (using the or keyword) to match (1) references with the same name and qualifier and (2) references that only have names and no qualifier. The neg find construct specifies that a qualifier cannot be found for the reference.

Listing 4 contains the definitions of the soft links for the IMA case study. The sourceBlock pattern describes that the reference stored in the sourceBlockRef feature of block (the source of the soft link) is equal to the reference of sourceBl (the target of the soft link) according to the referenceEqual pattern. Similarly, the slElement pattern specifies that the reference stored in elementRef is equal to the reference of elem. Finally, the mElement pattern is similar to the definitions in the BPM case study and uses simple attribute value equality to identify the target element.

```
// Block.sourceBlock link
pattern sourceBlock(block, sourceBl) {
    Block.sourceBlockRef(block,blRef);
    Block.simulinkRef(sourceBl,srcRef);
    find referenceEqual(blRef,srcRef);
}
// SimulinkElementReference.slElement link
pattern slElement(seRef, elem) {
    SimulinkElementReference.elementRef(seRef, ser);
    SimulinkElement.simulinkRef(elem, er);
    find referenceEqual(ser, er);
}
// SimulinkElementReference.mElement
pattern mElement(seRef, comp) {
    SimulinkElementReference.mElementId(seRef, meId);
    ModelElement.id(comp, meId);
}
```

Listing 4 Soft links in the IMA case study

4.4 Handling libraries in the IMA case study

Using soft links to define and maintain the references to the defining Simulink library elements allows flexibility to handle changes if an other version of the library is loaded into the

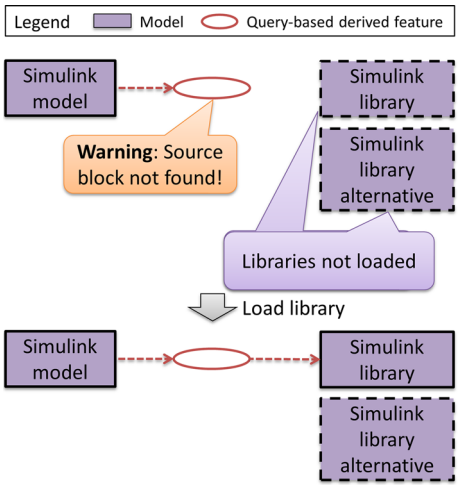


Fig. 7 Alternative libraries for a Simulink model

IMA tooling without the need to do any manual readjustment on the elements of the Simulink instance model.

Figure 7 shows when a Simulink model is loaded to the IMA tooling; however, we would like to use two different libraries based on the simulation goals the Simulink model will be used (e.g., simulation of scheduling algorithm or evaluation of failover or redundancy).

This approach also allows model editing without loading all referenced libraries, while soft links are not broken in the sense that whenever a library will be loaded the references will be automatically repaired.

Additionally, it is possible to add validation for the soft links and thus issue a warning for the user indicating that a given soft link should be set but its target was not found. This is very useful as the user can concentrate on those references that are broken due to the changes made either to the model or the library.

```
1  @Constraint(severity = "warning", location = Block,
2    message = "Source block with reference $blRef$
3      not found for block $block$"
4  )
5  pattern sourceBlockNotFound(block, blRef) {
6      Block.sourceBlockRef(block, blRef);
7      neg find hasSourceBlock(block, _srcBl);
8  }
9  pattern hasSourceBlock(block, srcBl) {
10     Block.sourceBlock(block, srcBl);
11 }
```

Listing 5 Validation of sourceBlock soft link

Listing 5 defines a validation rule with a warning level severity and a specific message that is parametrized with the block and blRef parameters of its defining query. The sourceBlockNotFound query matches on those blocks that has no referred library element through the sourceBlock soft link. This is defined using a negative application condition on the hasSourceBlock pattern.

5 Applications in external traceability

The approach outlined in Sect. 3.3 is capable of maintaining trace links and thus provide the basic functionality needed for the features defined in the two case studies introduced in Sect. 2. In this section, we elaborate on the use of external n-ary traceability interconnections in the BPM case study and discuss the advantages of using soft interconnections in the integration model of the IMA case study.

In general, soft links may offer several advantages when applied in an *external traceability context*. Figure 8 illustrates a typical architecture applied to the BPM case study (see Sect. 2.1), where interconnections between three distinct models (belonging to the *process*, *system* and *operation* domains, respectively) are augmented with *explicit (external) traceability models* (T).

In such a scenario, traceability models T typically conform to a custom metamodel that may describe simple binary (source-target) relationships with the help of association classes that use explicit unidirectional references to point to elements of the host models. In more complex cases, T may also include ternary (or hyper-) edges that interrelate multiple elements (e.g., three element types from all three domains, as in Fig. 8).

5.1 External traceability-specific challenges

While this commonly used approach has an obvious advantage over *internal traceability/correspondence links*, namely that the external models do not require the modification of the host metamodels, it also involves a number of frequently encountered problems as mentioned in Sect. 3:

- *Fragility.* Inter-model hard links are fragile, they may break when a host model is manipulated without also loading the traceability model. Additionally, when using file-based model fragments, traceability links may even break during external operations (e.g., when the files are moved within the workspace [47]).
- *Identification of target elements.* To work around the fragility issue, traceability models may use IDs or fully qualified naming schemes (as presented in our previous examples) to store cross-references, even for external traceability models. However, such key attributes need to be present in the host models, while they also necessitate an auxiliary mechanism to enforce consistency rules (such as uniqueness) within the host domains. If these prerequisites are not met, then additional, auxiliary techniques have to be used to add identifier maintenance capabilities to host domains, as used, e.g., by EMFStore [55] and CDO [57].
- *Persistence scalability issues.* In complex system modeling scenarios, the amount of edges can grow to be

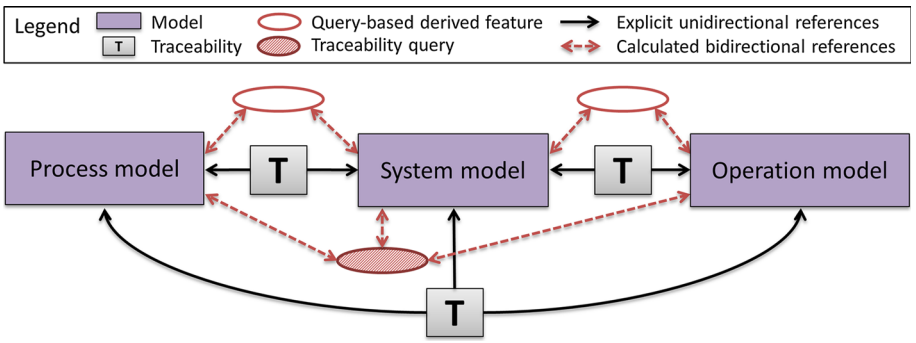


Fig. 8 External traceability modeling scenario

the dominant factor in storing the entire model repository, in terms of both in-memory and serialized persistence overhead [7]. Hence, the performance of all model management-related operations (e.g., serialization) may be severely negatively affected as the size of the model resources grow, especially when taking *fragility* into consideration (i.e., that traceability models with hard links need to be loaded and manipulated together with host model fragments).

5.2 Traceability management with bidirectional soft links

The traceability architecture (components with black outline in Fig. 8) can be augmented or even replaced with model-integrated *soft links* (symbolized by red outlined empty ovals) and *traceability queries* that can be accessed through the query evaluation framework (oval with dashed fill). Both techniques share incremental, on-the-fly evaluation as their background (which will be detailed in Sect. 6.1).

From the traceability perspective, the most important advantages of soft links are that they are (logically) *bidirectional* references that are *maintained upon model changes*. The bidirectionality is supported as the both parameters of soft link queries can be used for input or output. Thus, assuming that host metamodels are allowed to be extended, such traceability links can be added without regard for circular serialization dependencies. Therefore, it is entirely up to the language designer to specify where such traceability links are going to reside, making trace link navigation from host model elements feasible as well. The queries used for bidirectional references do not have to be bijective, as each match of the query will result in one trace link and a node can be the source or target of multiple links.

Additionally, as soft links provide graceful behavior for broken traceability references, erroneous trace records may be marked with warning markers, instead of throwing exceptions or runtime errors. These markers can then be corrected by, e.g., a user-aided, on-demand resolution process, which may be further supported by helper queries that locate the most likely target host model element (especially when non-ID keys are used to identify model elements, such as EntryJobCorrespondence in Fig. 6. In this case, a helper query may enumerate those elements whose local names are similar).

It is possible to fine-tune traceability links in a straightforward way. The designer can choose which references are stored explicitly and which ones are going to be calculated on-demand, when the models are loaded into memory, since the results of query evaluation can be both processed manually or represented by derived features. This gives the tool developer precise control over performance vs. compliance considerations (i.e., when certain traceability information is required to be stored persistently).

Finally, soft links behave exactly like normal links (send notifications), easing the integration with user interface components or on-the-fly validators.

5.2.1 *External traceability between domain and implementation models in the IMA case study*

Handling traceability references between domain and implementation model elements using soft links yields the same advantages as in the case of libraries. In addition, tools can be easily programmed to read and access individual model fragments completely separately. This is especially important where critical IP can be found in the different domain and implementation models, and only the required fragments are allowed to be disclosed to the different third-party tool providers.

Soft links can be checked with a validation query (see Fig. 9), similarly to the approach for Simulink library references (see in Sect. 4.4). However, due to the fact that a separate model element represents the traceability reference (SimulinkElementReference), two separate validation rules have to be defined in order to be able to differentiate when the domain model is not loaded (mElement) or the implementation model element is missing (slElement).

Finally, the separation of domain models from platform-specific implementation models also makes it possible to have a single component model correspond to different platform representations. For example, it is possible to create a single component library model that represents blocks in

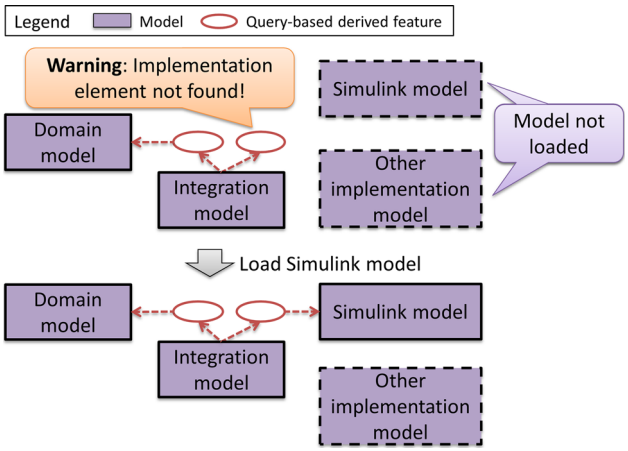


Fig. 9 Alternative implementations for domain model

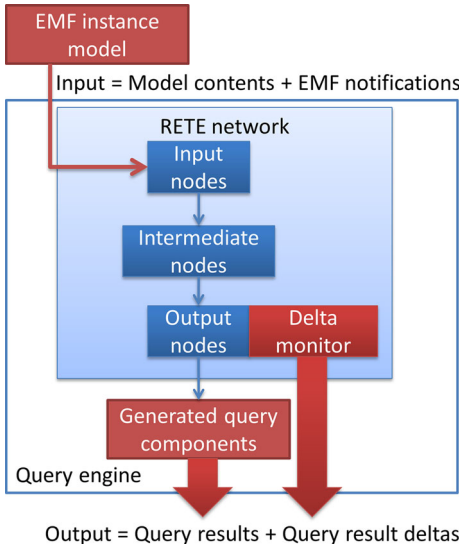


Fig. 10 The incremental query evaluation architecture

a Simulink model or components defined in a different platform (e.g., SysML [61] or AADL [49]). Depending on which platform representation is loaded at a given moment, different target models can be generated from the same PDM that uses the library.

5.3 Using traceability queries for n-ary links

If host metamodels cannot be modified, or hyperedges (multilinks, connecting three or more element types) are desired for traceability modeling, the architecture of Fig. 8 can be augmented with generic queries. In this case, a ChecklistEntry is connected to Tasks and, consecutively, to Data elements to represent the traceability information between data elements that are read by a given check list element. Such a ternary relationship may be implemented by the Ternary pattern (shown in Listing 6).

```
1 // data read by checklist entry
2 pattern Ternary(cle,t,d){
3  find ChecklistEntryTaskCorrespondence(cle,t);
4  find DataTaskReadCorrespondence(d,t);
5 }
```

Listing 6 Ternary link traceability query

The matches of the Ternary pattern in a model can be accessed using query evaluation. The parameter assignments can be read from the match, and the object or attribute values can be manipulated naturally to modify or process the model (see Sect. 9.4 for an example). It is also possible to observe the match set of the pattern and react to new or removed matches.

Note that the traceability relation between a SimulinkElementReference and its corresponding Simulink and Functional Architecture model elements is also representable as a ternary hyper-edge.

5.3.1 Summary

Soft links and traceability queries can be used to overcome the challenges presented by traceability-specific applications by complementing external traceability models and supporting incrementally maintained bidirectional links between interconnected elements.

6 Details of soft link computation

In this section, we outline how soft links can be managed using the incremental query evaluation features of the EMF-IncQuery framework. We propose an algorithm for mapping changes of query results to notifications. Our approach can be integrated to notification based applications (like EMF) in a transparent way by mapping model changes to the values of derived features using incremental evaluation.

6.1 Incremental evaluation of queries: an overview

The key to efficient evaluation and change notification for soft links is the incremental graph pattern matching infrastructure (first introduced in [48]) that uses the internal architecture shown in Fig. 10.

The input for the incremental graph pattern matching process is the EMF instance model and its Notification API where callback functions can be registered to instance model elements that receive notification objects (e.g., ADD, REMOVE, SET etc.) when an elementary manipulation operation is carried out.

Based on a query specification, the framework constructs a Rete rule network [48] that processes the contents of

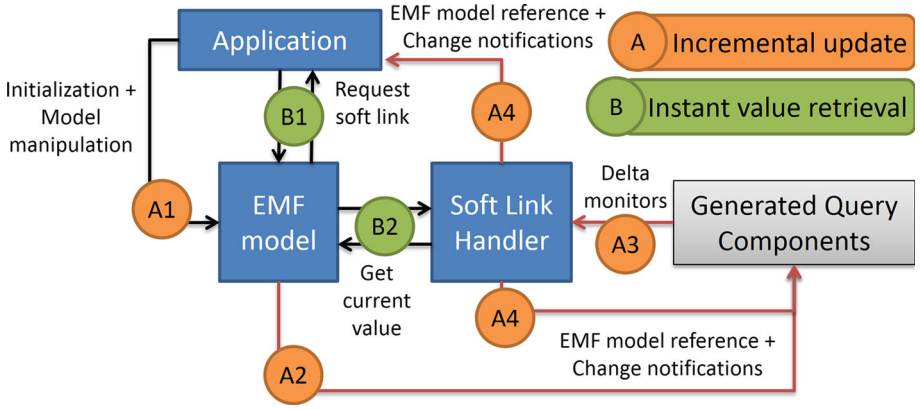


Fig. 11 Overview of the integration architecture, adopted from [43]

the instance model to produce the query result at its output node. Query results are then post-processed by *auto-generated query components* to provide a type-safe access layer for easy integration into applications. 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* through its *delta monitor* facility, which are used to incrementally update the query result. These deltas can also be processed externally, which is a key feature for the integration of derived features (Sect. 6.2).

By this approach, the query results (i.e., the match sets of graph patterns) are continuously maintained as an in-memory cache and can be instantaneously retrieved. 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), the query evaluation framework can evaluate very complex queries over large instance models very efficiently. These special performance characteristics, reported in [65], allow query-based derived features to be evaluated instantly after model modifications in most cases, regardless of the size of the instance model.

6.2 Derived features driven by queries

To support soft links captured as derived features, the outputs of the engine need to be integrated into the EMF model access layer at two points: (1) *query results* are provided in the getter functions of derived features, and (2) *query result deltas* are processed to generate EMF Notification objects that are passed through the standard EMF API so that application code can process them transparently. The overall architecture of our approach is shown in Fig. 11.

The application accesses both the model and the query results through the standard EMF model access layer—hence, no modification of application source code is necessary. In the background, as a novel feature, *soft link handlers* are attached to the EMF model objects that integrate the generated query components (pattern matchers). This approach follows the official EMF guidelines of implementing derived features and requires less programming effort to integrate than ad hoc Java code, or OCL expression evaluators.

When an EMF application intends to read a soft link (B1 on Fig. 11), the current value is provided by the corresponding handler (B2) by retrieving the value from the cache of the related query. When the application modifies the EMF model (A1), this change is propagated to the generated query components along notifications (A2), which may update the delta monitors of the handlers (A3). These updates are processed by the handler and turned into derived feature changes, which can trigger further changes in the result sets of other derived features (A4).

6.2.1 Illustrative example

Figure 12 illustrates a detailed elaboration of soft link handlers, which process elementary model manipulation notifications to update and generate notifications for derived features. The figure corresponds to a case where the user assigns a new Job to a ChecklistEntry through the Editor which is essentially a cle.getJobPaths().add(jobPath) method call on the Model. During the add method, the ChecklistEntry EObject sends an ADD notification to the Notification Manager, which will notify the Query Engine about the model modification. The Query Engine updates the match sets of each query and registers the match events in the Deltamonitor. Once the update of the Rete network is finished, it invokes the callback method of each IncqueryFeatureHandler. Each handler has a Deltamonitor from which it retrieves the new and lost match events since the last callback to process them. During the processing, the handler may send new notifications (e.g., the value set of the info soft link of job is updated) that is propagated to listeners. Any time the soft link value is retrieved from the model (e.g., job.getInfo()), it accesses the handler for the current value of the derived feature, which is returned instantly.

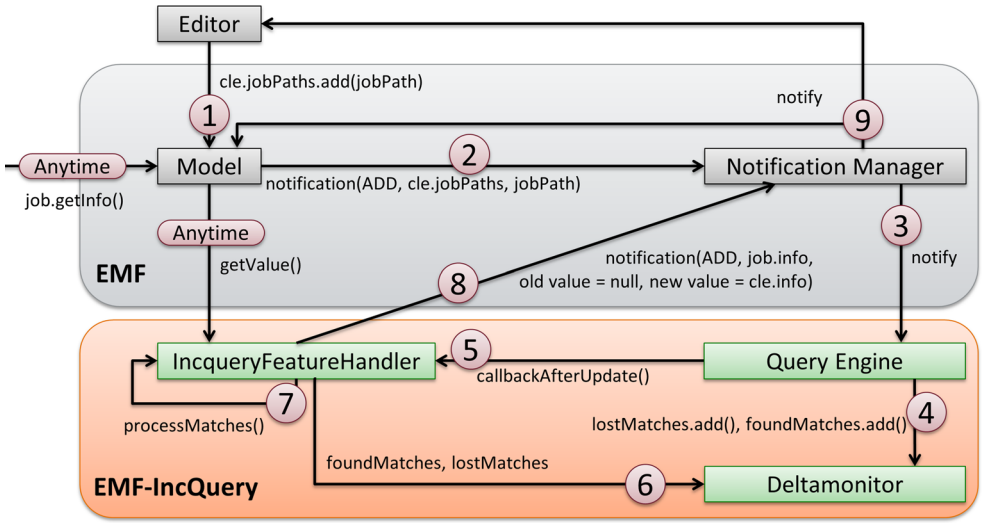


Fig. 12 Elaboration of the execution

6.2.2 *Role of incrementality*

Soft links could be built on top of a batch query evaluation technology, but there are three essential roles of incrementality for making soft links feasible: (1) the query engine notifies the soft link handler *when* the results of the query changed, so that updates can be performed, (2) it specifies *which* soft link changed; therefore, it is not necessary to evaluate all queries upon each change and (3) it provides information on *how* the results changed (appeared and disappeared matches), therefore iteration on all query results can be avoided.

6.3 From changes of match sets to notifications

We now explain the notification processing and propagation procedure in algorithmic detail (extending our previous paper [43]). For the sake of simplicity, we introduce an auxiliary discriminator variable $Kind$ whose value represents two distinct cases:

- SINGLE and MANY correspond to derived references of target multiplicity 1 and *, respectively (e.g., the soft links task and jobs of ChecklistEntry);
- More complex kinds of derived feature with an arbitrary, deterministic iteration algorithm can also be handled by the approach. For example, summing up the values of a numeric attribute in the list of matched objects.

The main part of our soft link handler algorithm is an event loop that is called by the query engine each time the underlying Rete network is updated as a result of some model manipulation (see Algorithm 1).

The algorithm is initialized with the following input variables (line 2): (1) the EObject $Source$ whose soft link is handled; (2) the derived $Feature$; (3) the $DeltaMonitor$ for the query matcher; and (4) the previously mentioned discriminator value $Kind$. Each handler stores an internal value for the feature, initialized in line 3 depending on $Kind$. Finally, the handler uses two global variables: pU for storing partial events and the set N of unsent notifications.

The event loop starts from line 5, it first resets the partial event store, then processes matches found since the last execution of the loop (line 9). These events are supplied by the delta monitor of the query and removed after processing is finished. Similarly, the matches lost since the last execution are also processed (line 14) and removed after.

When a soft link with SINGLE kind is used and only a *match-found* event occurs without a *match-lost* event, an

Algorithm 1 Main event loop

```
1:  let S ← Source, F ← Feature, DM ← DeltaMonitor,
2:  let k ← Kind                                          ▷ Input variables
3:  let (k = SINGLE)?iV ← null : iV ← ∅                   ▷ Internal value init
4:  let pU ← null, N ← ∅                                  ▷ Global variables
5:  function EVENTLOOP
6:      let pU ← null
7:                                                        ▷ First processing found events
8:      let MFE ← DM.matchFoundEvents
9:      let found ← PROCESSFOUNDMATCHES(MFE)
10:     let RFE ← DM.matchFoundEvents \ found
11:     let DM.matchFoundEvents ← RFE                     ▷ Removing events
12:                                                       ▷ Then processing lost events
13:     let MLE ← DM.matchLostEvents
14:     let lost ← PROCESSLOSTMATCHES(MLE)
15:     let RLE ← DM.matchLostevents \ lost
16:     let DM.matchLostevents ← RLE                      ▷ Removing events
17:                          ▷ If stored value not yet used, handle partial match event
18:     if partialUpdate ≠ null then
19:         let N ← N ∪ notification(SET, null, pU)
20:         let iV ← pU                                   ▷ Updating value
21:     end if
22:     while N ≠ ∅ do                                    ▷ Notification sending loop
23:         let n ← N[0]
24:         let N ← N \ n
25:         S.eNotify(n)                                  ▷ Sending notification through source
26:     end while
27: end function
```

additional processing step is required to handle the partial event (line 19). This occurs when the query did not lose any matches since the last event loop, but a new match is found. This translates to a notification representing the setting of the feature value from *null* to pU (line 20). Finally, if there are any unsent notifications (line 22), the first notification n in the list N is sent through the *Source* EObject. In order to stabilize the notification loop, the notification sending is separated from the calculation of the soft link value. Thus, new notifications caused by n are simply added to the list N, which will be depleted after all, if causal circularity between the definitions of soft links is avoided. Such circularity occurs if soft links are chained and a circle in the chain does not allow a steady state after a change.

Algorithm 2 Processing match-found events

```
1: function PROCESSFOUNDMATCHES(events)
2:     let P ← ∅
3:     for all e ∈ events do
4:         if e.source = S then
5:                                          ▷ Extracting feature target from event
6:             let target ← e.target
7:             if k = SINGLE then
8:                                          ▷ Storing value for later processing
9:                 let pU ← target
10:            else if k = MANY then
11:                let N ← N ∪ notification(ADD, null, target)
12:                let iV ← iV ∪ target                          ▷ Updating value
13:            end if
14:        end if
15:        let P ← P ∪ e
16:    end for
17:    return P
18: end function
```

New matches. The handling of match-found events is detailed in Algorithm 2. The PROCESSFOUNDMATCHES function iterates through the match-found events (line 3) and extracts the target object from the event (line 6), if the source EObject of the event equals *Source*. Depending on the *Kind* of the soft link, a notification is created and the internal value is updated (line 11 for MANY). For SINGLE features, the target object is stored for later usage (line 9). Finally, the list of processed events is returned.

Lost matches. The handling of match-lost events is similar to the processing of match-found events, see Algorithm 3. The PROCESSLOSTMATCHES function iterates through the match-lost events (line 3) and extracts the target object from the event (line 6), if the source EObject of the event equals *Source*. Depending on the *Kind* of the soft link, a notification is created and the internal value is updated (line 13 for MANY). For SINGLE kind features, the stored value of pU is used for creating the notification (line 8). Finally, the list of processed events is returned at the end of the function.

Algorithm 3 Processing match-lost events

```
1: function PROCESSLOSTMATCHES(events)
2:     let P ← ∅
3:     for all e ∈ events do
4:         if e.source = S then
5:                                          ▷ Extracting feature target from event
6:             let target ← e.target
7:             if k = SINGLE then                              ▷ Using stored value
8:                 let n ← notification(SET, target, pU)
9:                 let N ← N ∪ n
10:                let iV ← target                             ▷ Updating value
11:                let pU ← null                               ▷ Resetting stored value
12:            else if k = MANY then
13:                let n ← notification(REMOVE, target, null)
14:                let N ← N ∪ n
15:                let iV ← iV \ target                        ▷ Updating value
16:            end if
17:        end if
18:        let P ← P ∪ e
19:    end for
20:    return P
21: end function
```

	Hard link	Soft link (OD)	Soft link (IQ)
Read access	Yes	Yes	Yes
Direct manipulation	Yes	Partly[1]	Partly[1]
Change notification	Yes	Partly[1]	Yes
Inconsistent state	Error on load	Error on read	Warning message
Persisted	Yes	No	No

Fig. 13 Comparison of trace link variants (*OD* on-demand computation, *IQ* incremental queries, *1* Can be supported by additional, manual programming)

6.4 Comparison of trace link variants

We distinguished between three variants of trace links that are possible between model fragments. *Hard links* are regular edges that are created manually and stored persistently, *on-demand* (OD) soft links are computed when required, while soft links driven by *incrementally evaluated model queries* (IQ) are updated automatically. Figure 13 summarizes the differences between the three variants of trace links.

Each link variant allows regular read access to the target of the link; however, only a hard link allows direct manipulation as well. In the case of soft links, direct manipulation can be added by additional, manual programming. Furthermore, supporting proper change notifications with on-demand computation also needs manual programming for each type (as te computation uses different query), although it is very important for many applications. A major difference between the three variants is dealing with an inconsistent state, where a hard link may cause errors during loading from persistent storage if the target of the link does not exist. An on-demand soft link will not be able to compute its value or even cause errors on read, when the application requests the target of the link. While incrementally updated soft link can offer graceful degradation with warnings instead of runtime errors and exceptions in cases when the target of the link is not available (for example the resource is not in the resource set). Finally

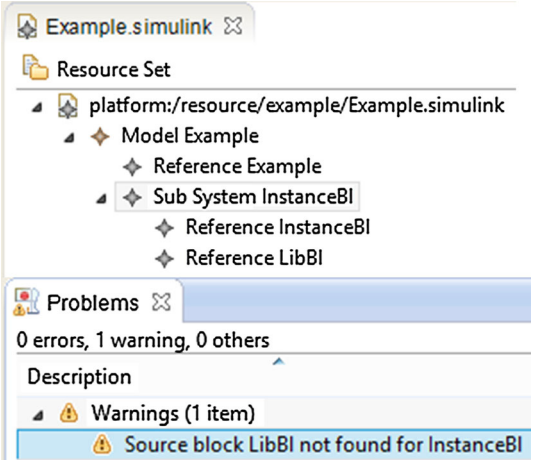


Fig. 14 Validation message of missing source block

as only hard links are persisted, using soft links can lead to smaller model sizes and reduced load time.

6.5 Soft links in EMF

In traditional EMF-based tools, references between separate model fragments (called *resources* in EMF terminology) are materialized in the in-memory object model as *proxies*. Such proxies contain URIs that are resolved either when a resource is loaded or when the proxied reference is traversed (by calling its getter function). If the proxy resolution fails (e.g., because of a broken link due to changed file paths), an exception is thrown and the proxied reference remains in a (read only) error state until it can be successfully resolved. As the resolution is not supported by any built-in automatism, it is up to the tool developer to programmatically trigger proxy resolution (e.g., by calling `EcoreUtils.resolve()`) whenever applicable (e.g., when a new resource that may contain unresolved proxy endpoints is loaded by the user).

In addition to the functionality issues, multifile model management with EMF proxies also has a performance impact, as proxy resolution is a costly operation. Thus, depending on the nature and cardinality of cross-resource references, the order in which such resources are loaded can have a significant influence on the total time needed for loading models into memory (since resolution is performed in an eager fashion by default).

In contrast, soft links are resolved automatically at the earliest possibility, regardless of the triggering event (e.g., whether a resource has been loaded or the model manipulated). Additionally, the resolution of all soft links in a resource is performed in a single traversal, as all soft link endpoints are already indexed by EMF- IncQuery. Finally, soft links that cannot be resolved are marked with validation error messages (see Fig. 14) and are editable so that they can be fixed easily. The fixing of broken soft links may also be aided by advanced features such as quick fixes (backed by a heuristics-based search [26]). The message on Fig. 14 is created by the validator described in Listing 5 and warns that the source block LibBl was not found, although it is referenced by InstanceBL. While technologically feasible, we opted not to integrate soft links to EMF as custom proxies since many modeling tools are not prepared to handle proxy resolution notifications as well as standard ADD/REMOVE/SET notifications for EReferences.

Case	Model	Nodes	Edges
Sparse	Domain	9	13
	Simulink	42	44
	Library	42	42
Dense	Domain	179	403
	Simulink	558	809
	Library	460	648

Fig. 15 Model sizes used in the performance evaluation

6.6 Performance evaluation

Navigating on edges in a graph is one of the most used primitives in building model-driven applications. Therefore, the performance of soft links (both on-demand or incrementally evaluated) is important as they must behave transparently as regular edges for applications.

We present runtime measurements[4] on two scenarios described for the IMA case study with two different model sets:

- The first scenario is loading a library for a Simulink model as discussed in Sect. 4.4. Before loading the model fragment of the library, the source block references of blocks in the initial Simulink model are not created. After loading the library, the source blocks are found and the soft links are created.
- The second scenario is loading a Simulink model as an implementation model for a domain model as described in Sect. 5.2. Before loading the model fragment of the Simulink model, the traceability soft links from the initial domain are not created. After loading the Simulink implementation model, the links are created automatically.

The number of nodes and edges in the used models are listed in Fig. 15. In the Sparse case, the number of nodes and edges are approximately equal in the models, while in the Dense set the number of edges is 1.5–2 times higher than the number of nodes.

[4] For detailed performance evaluation of EMF- IncQuery as a query evaluation framework see [65].

Scenario	Simulink and Library		Domain and Simulink	
Case	Sparse	Dense	Sparse	Dense
Matches	29	881	34	942

Fig. 16 Total number of matches in the scenarios

Figure 16 lists the number of matches relevant to soft links after loading both models used in the scenario, this also includes the matches of the helper queries.

The model sizes and number of matches are computed using the metrics `countNodes`, `countEdges` and `countMatches` as defined in [29]. Based on our previous experience, these metrics are relevant for scenarios including query initialization and incremental evaluation.

6.6.1 Evaluation environment and method

We measured the following steps for both cases and both model sets:

1. Load primary model: The primary model (Simulink or Domain) is loaded into EMF without query evaluation.
2. First soft link access: The value of the soft link is requested from the model, causing the initialization of the queries and building the Rete network.
3. Load secondary model: The secondary model (Library or Simulink) is loaded into EMF. The values of soft links are updated incrementally by the query engine during the loading process as the model is constructed in the memory.
4. 10k soft link accesses: The value of the same soft link is requested again. The request is executed 10,000 times since one request only takes nanoseconds that cannot be reliably measured.

Additionally, we also measured baseline performance, where the Load primary model and Load secondary model phases are executed sequentially. This corresponds to loading the models into EMF without any soft link access, and thus, no query initialization or evaluation is performed. Therefore, these results are used to measure the overhead of our approach.

The results of these measurements are shown in Fig. 17 for each scenario, case and model set.[5]

The memory overhead of incremental evaluation is around 1 MB for the sparse cases and 4 MB for the dense cases. This includes the memory overhead of the Rete network of EMF-IncQuery and the soft link handlers.

6.6.2 Evaluation of results

We made the following observations based on the measurement results:[6]

- *No overhead on value requests.* The incremental query evaluation ensures that once the value of the soft link is computed, it can be requested without additional overhead. In our results, the 10,000 requests took 0.1–0.2 ms while even basic on-demand computations could take as long for each request. Therefore, applications developed over models containing soft links will have an acceptable performance.
- *Fast initialization and incremental update.* The computation of soft links upon first request, including the initialization of match sets (First soft link access), and later incremental computation after model changes (Load secondary model) is fast. The query initialization and model traversal took between 35 and 90 $ more than simply loading the primary model. The overhead of the query evaluation depends on the size of the change, not the complete model, while the overhead of soft link computation depends only on the size of appearing and disappearing matches (called *match set deltas*).
  Specifically, the differences in the Load secondary model values in the two cases (up to 36 % in our measurements) are the overhead of incremental query evaluation for processing the contents of the secondary model. Furthermore, in regular model-driven applications, where typically the number of value requests outweigh the number of (small) model modifications, both the initialization and the incremental update overheads are compensated by the instant value requests (i.e., case soft link accesses).

Summary. The combined use of incremental pattern matching and notification processing allows query-driven soft links (and derived features) to behave exactly as regular features of EMF instance models. This behavior ensures that user interfaces, model validators etc. can safely depend on soft interconnections built on soft links, without on-demand querying.[7]

[5] All measurements were carried out on a PC with Intel Core i5-2410M 2.3 GHz processor, Windows 8, Java 1.6 64 bit with 512 MB heap space (DDR3 memory), wall-time measurement with nanotime precision. The measurements are taken after a warm-up phase in order to eliminate other factors such as class loading, JIT compiler, disk access. The final values are taken as the average of at least 10 measurements.

[6] The numerical results of the measurements are included in Fig. 18 in the Appendix.

[7] Details on integration with existing EMF-based technologies are discussed in Sect. 9.

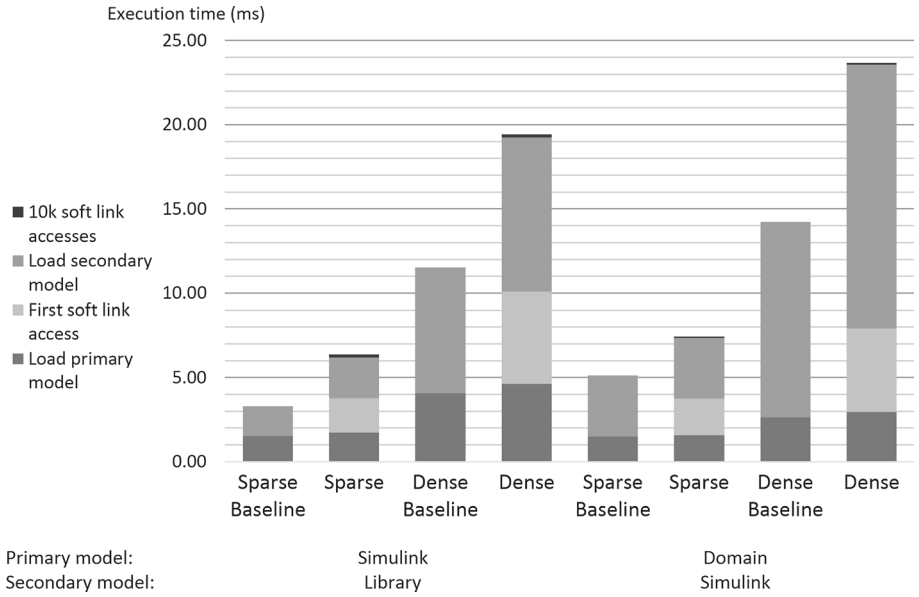


Fig. 17 Performance evaluation results

7 Related work

In this section, we first give an overview of existing approaches and tools that deal with interconnection between models, then we briefly describe other model query techniques for EMF. Finally, we list approaches that rely on derived features and therefore may take advantage of our incremental evaluation techniques.

7.1 Previous results

This paper is based on our MODELS 2012 paper [27], extended with modeling framework independent foundations. Furthermore, we elaborate on an additional industrial case study from the avionics domain. In a previous paper [43], we offer an algorithm for incremental evaluation of derived features and present technical details on the integration of existing native implementations. The current paper provides details on applying incremental queries for soft interconnections by using computed references and advanced traceability use cases.

7.2 Traceability modeling

Traceability in software modeling scenarios has been the topic of many papers (e.g., [13,18,19,42,71]). Recently, [37] emphasized the use of *weaving models* as a special kind of correspondence models to semi-automatically derive model transformation rules for model synchronization. The authors present a metamodel-based method that exploits metamodel data to automatically produce weaving models in the AMW System. The weaving models are then derived into model integration transformations. In the current paper, we use traceability concepts in the classical sense, i.e., they are used to link logically corresponding representations of identical concepts expressed in different languages and/or abstraction levels. However, a crucial difference of the current paper to these previous works is that we do not use explicit (internal or external) traceability models, but rely on traceability (correspondence) information derived by model queries and represented as soft links. In addition, our soft links are dynamic in the sense that incremental, on-the-fly query evaluation automatically ensures that traceability links are always consistent with the current state of models.

7.3 Interconnecting EMF models

In [32], correspondences between models are handled by matching rules defined in the Epsilon Comparison Language, where the application conditions (called guards) use queries similarly to our approach. Additionally, Epsilon also manages model integrity between EMF models using the novel Concordance framework [47]. It is able to handle intermodel links when models are moved/renamed and helps in correcting invalid models caused by metamodel changes. Anwar [1] introduces a rule-driven approach for creating merged views of multiple separate UML models and relies on a correspondence metamodel and OCL expressions to support model merging and composition. VirtualEMF [14] 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 [74], an ATL-based method is presented for automatically synchronizing source and target models of a

given transformation, based on the definition of the transformation.

Compared to them, the main distinctive features of our approach is (1) the fully incremental evaluation of queries for model interconnections and (2) flexible support for query-based, computed soft links. In the future, we plan to investigate how to combine the advantages of our approach with the benefits of existing solutions.

7.4 Model query approaches

OCL [60] is a standardized navigation-based query language, applicable over a range of modeling formalisms. Taking advantage of the expressive features and wide-spread adoption of OCL, the project Eclipse OCL [20] through its *Essential OCL* language provides a powerful query interface that evaluates OCL expressions over EMF models. Additionally, it also supports the definition of invariants, operations and derived features to enrich the Ecore metamodel using either the *Complete OCL* [72] or the *OCLinEcore* [21] languages. Balsters [4] presents an approach for defining database views in UML models as derived classes using OCL. The derived classes in this case are the result set of queries, which is similar to the match sets provided by EMF- IncQuery.

There are several technologies for providing declarative model queries over EMF, e.g., EMF Model Query 2 [54] and EMF Search [56]. EMF Model Query 2 provides query primitives for selecting model elements, but the expressive power of its language is limited compared to EMF-IncQuery. However, it can evaluate queries over indexes without loading the model used for creating the index. EMF Search is a framework for executing simple textual searches over EMF resources using a controllable scope. More advanced search capabilities have to be provided by metamodel-specific means, while the query engine of EMF-IncQuery is metamodel independent. Other graph pattern-based techniques like [10,23] have been successfully applied in an EMF context, but these focus on graph transformations related challenges, while EMF- IncQuery provides a unique query development, integration and efficient evaluation framework. To sum up, none of the above mentioned query technologies support incremental evaluation of queries and thus cannot be effectively used for inter-model soft links.

Cabot and Teniente [12] present an algorithm for incremental runtime validation of OCL constraints and uses promising optimizations; however, it works only on boolean constraints. An interesting model validator over UML models [24] incrementally re-evaluates constraint instances whenever they are affected, but relies on environments that support the recording of read-only access to the model, unlike EMF. Additionally, general purpose model querying is not viable.

Outside the Eclipse ecosystem, the Resource Description Framework (RDF [44]) is developed to support the description of instances of the semantic web, assuming sparse, ever-growing and incomplete data. Semantic models are built up from triple statements, which can be queried using the SPARQL [70] graph pattern language with tools like Sesame [52] or Virtuoso [41]. In addition, SPIN [69], the standard SPARQL Inferencing Notation combines concepts from object oriented languages, query languages, and rule-based systems to describe object behavior on the web of data. One of the basic ideas of SPIN is to link class definitions with SPARQL queries to capture constraints and rules that formalize the expected behavior of those classes. Unfortunately, while some advanced semantic tools such as TopBraid [64] have some preliminary support for the incremental execution of simple SPIN rules (e.g., for the calculation of derived attribute values), comprehensive incremental execution of arbitrarily complex queries (as in our approach) is not supported.

To sum up, several approaches provide possible alternatives to implement model queries; thus, they can potentially be used for providing soft links. However, many of them lack incremental evaluation support or require significantly more integration effort to enable their use for soft links.

7.5 Application of derived features

The PROGRES language [51] allows the rule-based programming of graph rewriting systems and uses derived attributes for encoding dynamic semantics. ConceptBase.cc [30] 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, the latter could be applied in models as derived features representing data stored in the ConceptBase.cc DB. Neither tool has adopted EMF up to our best knowledge.

In [16] Diskin describes a formal framework for model synchronization that uses derived references for propagating changes between corresponding models. A recent work by Diskin et al. [17] 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; however, it offers algebraic specification, while our approach might serve as an implementation for this generic theoretical framework.

The MOF 2.0 tool in [50] allows the definition of derived features using OCL. It handles derived attributes and operations as custom code provided by the user and redirects calls using reflection. The FUJABA [40] tool suite also supports derived edges by path expressions. Both tools work in a non-incremental way.

JastEMF [11] is a semantics-integrated metamodeling approach for EMF. It uses derived features as side effect free operations (i.e., queries) and refers to them as the static semantics of the model. Therefore, our query-based approach could be integrated with JastEMF without any problems.

8 Conclusion

Interconnections between model fragments of complex EMF models are usually represented as regular associations and persisted using storage-specific URIs. This approach proves to be rigid and error prone in many application scenarios.

We proposed to use derived features as a flexible alternative to provide soft interlinking between model fragments and demonstrated an approach for incremental evaluation of soft links with the use of model queries on two industrial case studies. Our approach supports circular dependency between models, graceful handling for unresolved links and is implemented using EMF- INCQUERY, which provides efficient evaluation capabilities for incremental model queries. Further advantages of the approach are the straightforward integration capabilities to existing domains and applications and the fine-tunability for external traceability use cases and bidirectional and n-ary soft links. The approach also supports composition of models containing soft links, since these links appear as regular edges in the graph and queries can be specified to match on such edges.

Finally, the concept of representing query results as first-class edges in graph-based models (thus supporting soft link chaining) and the algorithm for inserting and removing such edges is applicable to modeling environments and query techniques other than EMF and EMF- INCQUERY.

As a primary direction for future work, we plan to integrate traceability queries into the EMF model layer by constructing *derived classes* whose instances behave like EObjects but their life cycles are managed by an underlying incremental query. Such constructs could be used to create n-ary traceability models that are automatically kept in-sync, retaining the graceful handling of soft links. Our initial results are published in [15].

Acknowledgments We would like to thank the anonymous reviewers for their valuable comments.

9 Appendix 1: Integration of soft links with EMF-based tools

9.1 Integration with Ecore

In the prototype implementation of our proposal, we integrated our approach to the EMF tooling by a code generator that supports the automatic derivation of integration code for our components (EMF- INCQUERY *soft link feature handlers*). The input of code generation is a generator model (referencing the EMF genmodel for the domain) that crosslinks derived features with EMF- INCQUERY query specifications (stored as EMF models thanks to the Xtext-based tooling [59]).

```
/**
 * Handler for query-based feature
 */
private IncQueryFeatureHandler sourceBlockHandler;

/**
 * @query-based getter created by EMF-IncQuery
 * for query-based feature sourceBlock
 */
public Block basicGetSourceBlock() {
  if (sourceBlockHandler == null) // on-demand init
    sourceBlockHandler = IncQueryFeatureHelper
      .getIncQueryFeatureHandler(
        // source EObject and feature from metamodel
        this, SimulinkPackage.getBlock_SourceBlock(),
        "simulink.sourceBlock", // query FQN
        IncQueryFeatureKind.SINGLE_REFERENCE // kind
      );
  return (Block) sourceBlockHandler
    .getSingleReferenceValue(this);
}
```

Listing 7 Generated code for integrating soft links

The generated integration code (Listing 7) consists of (a) the instantiation of derived feature handlers (in the constructor of EObjects), which ensures that their lifecycle is tied to the hosts, to enable their garbage collection together with the instance model itself; (b) getter implementations that delegate calls to the appropriate function of the feature handler object, and wrap the result in unmodifiable ELists to ensure that any attempt to write to derived features will result in a runtime exception.

9.2 Integration with legacy derived features

In practice, a complete refactoring of an EMF-based tool to exclusively use EMF- INCQUERY-based derived features might not be realistic, as derived features have been extensively implemented using Java code or OCL-based approaches such as OCLinEcore [21] supported by Eclipse OCL [20]. As these implementations provide only getter functionality and do not provide change notifications, we implemented an additional *derived feature adapter* [43] as a lightweight add-on component for EMF model plugins,[8] which can be used to augment existing derived feature implementations (regardless of whether Java or OCL is used).

The basic concept motivated by a suggestion in the Eclipse FAQ[9] is analogous to the previous discussion. The language

[8] For usage details , see http://incquery.net/incquery/new/examples/derivedfeatures.

[9] http://wiki.eclipse.org/EMF/Recipes#Recipe:_Derived_Attribute_Notifier.

Scenario	Case	Load primary model (ms)	First soft link access (ms)	Load secondary model (ms)	10k soft link accesses (ms)
Primary model: Simulink Secondary model: Library	Sparse Baseline	1.51		1.79	
	Sparse	1.74	2.02	2.43	0.18
	Dense Baseline	4.07		7.44	
	Dense	4.62	5.48	9.15	0.18
Primary model: Domain Secondary model: Simulink	Sparse Baseline	1.50		3.63	
	Sparse	1.59	2.17	3.58	0.10
	Dense Baseline	2.64		11.58	
	Dense	2.95	4.96	15.66	0.10

Fig. 18 Performance measurement results

engineer can add a few lines of Java code to the generated EMF model plugin: these derived feature adapters attach listeners (through the EMF Notification API) to the (explicitly specified) features a derived feature depends on, and receive notifications when model changes are registered. These notification objects are then processed and converted into new notification objects for the derived feature, propagating through the manager to application code.

This approach has additional key advantages: (1) notification support can be added – with a small implementation effort – to "legacy" derived features, without having to re-write them in EMF- IncQuery; (2) queries specified in EMF- IncQuery (whether for derived features, or on-the-fly validation purposes, or within model transformations) can reference derived features seamlessly. However, performance implications need to be taken into consideration if the original getter (Java code or OCL expression) involves complex logic, or if the derived feature computation has many dependencies but they change only rarely – in those cases, rewriting the Java code or OCL expression into EMF- IncQuery may be a better option.

9.3 Integration with Xcore

Xcore [39] is a textual syntax extension to Ecore, based on Eclipse Xtext. Xcore provides textual constructs for all (static) Ecore concepts such as EClasses, EAttributes and EReferences, and it also can be used to define the dynamic aspects of the modeling language using EOperations and derived features that contain imperative code segments written in Xbase expression language.

As both the pattern specification language of EMF- IncQuery and Xcore are based on Xtext, they can be integrated in a straightforward way. Listing 8 illustrates an extended Xcore syntax that includes some new language elements so that query-based soft links can be seamlessly integrated into the language definition.

```
1 import-incqueries simulink.eiq;
2
3 class Block {
4   contains SimulinkReference elementRef;
5
6   incquery-derived Block sourceBlock
7     spec simulink.sourceBlock;
8 }
```

Listing 8 Xcore integration code

In this case, the definition of the EClass *Block* imports the query definitions of Listing 4, which is indicated by the `import-incqueries` keyword, and a new query-based soft link *sourceBlock* is defined using the `incquery-derived` keyword, by specifying a reference to the `simulink.sourceBlock` pattern definition of Listing 4. These concepts can be applied to derived scalar attributes (conforming to EDatatypes) in a straightforward way.

9.4 Processing n-ary traceability queries

In Sect. 5.2, we introduced the query Ternary that represents a traceability hyper-edge between data elements that are read by the task of a given check list element (see Listing 6). Such a query can be used in an application for processing such hyper-edges. This approach shares the functional benefits of soft links, with the one exception that it is not integrated into the EMF model layer and as such, it is not API-transparent to EMF-based tools. Instead, the query results can be accessed through an additional API provided by EMF-IncQuery (illustrated in Listing 9 using Xtend [58]).

```
1  // initialize matcher on resource set
2  val engine = IncQueryEngine::on(resourceSet)
3  val matcher = TernaryMatcher::on(engine)
4  int matchNum = matcher.countMatches(cle, null, null)
5  println("Found " + matchNum + " matches " +
6    " for entry " + cle.name)
7  // iterate on all existing matches
8  matcher.forEachMatch(cle, null, null)[match |
9    println("Entry reads data " + match.data.name +
10   " through task " + match.task.name)]
11 // observe match changes
12 val obs = IQObservables::observeMatchesAsSet(matcher)
13 obs.addSetChangeListener[
14   // called when the match set changes
15   diff.additions.forEach[ addition |
16     val match = addition as TernaryMatch
17     println("New ternary edge: " + match.entry.name +
18     " reads data " + match.data.name +
19     " through task " + match.task.name)]
20   // similarly for diff.removals
21 ]
```

Listing 9 Ternary link processing code

Here, the results of the Ternary pattern are processed using a generated TernaryMatch data transfer class (which is the class of match) and the MatchProcessor visitor interface (that has a single process method inlined in the forEachMatch method). The EMF- INCQUERY API also exposes *match update* facility (Sect. 6.1) that allows to track the changes in the result of such a query. The second part of Listing 9 shows that a change listener is registered and added (or removed) matches can be handled.

References

1. Anwar, A., Ebersold, S., Coulette, B., Nassar, M., Kriouile, A.: A rule-driven approach for composing viewpoint-oriented models. J. Object Technol. **9**(2), 89–114 2010. doi:10.5381/jot.2010.9.2.a1. http://www.jot.fm/contents/issue_2010_03/article1.html
2. AUTOSAR Consortium: The AUTOSAR Standard. http://www.autosar.org/ (2012)
3. Bąk, K., Czarnecki, K., Wąsowski, A.: Feature and meta-models in clafer: mixed, specialized, and coupled. In: 3rd International Conference on Software Language Engineering, Eindhoven, The Netherlands (2010). doi:10.1007/978-3-642-19440-5_7
4. Balsters, H.: Modelling database views with derived classes in the UML/OCL-framework. In: Stevens, P., Whittle, J., Booch, G. (eds.) ≪ UML ≫ 2003—The Unified Modeling Language. Modeling Languages and Applications, LNCS, vol. 2863, pp. 295–309. Springer, Berlin (2003)
5. Bergmann, G.: Incremental model queries in model-driven design. Ph.D. dissertation, Budapest University of Technology and Economics, Budapest (2013)
6. Bergmann, G.: Translating OCL to graph patterns. In: Dingel, J., Schulte, W. (eds.) ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems, MODELS 2014. Springer, Valencia (2014)
7. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental evaluation of model queries over EMF models. In: MODELS'10, Springer, LNCS, vol. 6395 (2010)
8. Bergmann, G., Ujhelyi, Z., Ráth, I., Varró, D.: A graph query language for EMF models. In: Proceedings of ICMT'11, Springer, Berlin (2011)
9. Bergmann, G., Ráth, I., Szabó, T., Torrini, P., Varró, D.: Incremental pattern matching for the efficient computation of transitive closures. In: Sixth International Conference on Graph Transformation, Bremen, Germany (2012)
10. Biermann, E., Ermel, C., Taentzer, G.: Precise semantics of EMF model transformations by graph transformation. In: MoDELS'08, Springer, Berlin (2008)
11. Bürger, C., Karol, S., Wende, C., Aßmann, U.: Reference attribute grammars for metamodel semantics. In: Malloy, B., Staab, S., van den Brand, M. (eds.) Software Language Engineering, LNCS, vol. 6563, pp. 22–41. Springer, Berlin (2011). doi:10.1007/978-3-642-19440-5_3
12. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. J. Syst. Softw. **82**(9), 1459–1478 (2009). doi:10.1016/j.jss.2009.03.009
13. Champeau, J., Rochefort, E.: Model Engineering and Traceability. In: SIVOES-MDA Workshop, UML 2003 Conference (2003).
14. Clasen, C., Jouault, F., Cabot, J.: Virtual composition of EMF models. In: 7èmes Journées sur l'Ingénierie Dirigée par les Modèles (IDM 2011), Lille, France. http://hal.inria.fr/inria-00606374 (2011)
15. Debreceni, C., Horváth, Á., Hegedüs, Á., Ujhelyi, Z., Ráth, I., Varró, D.: Query-driven incremental synchronization of view models. In: Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling, ACM, ACM, York, UK (2014). doi:10.1145/2631675.2631677
16. Diskin, Z.: Model synchronization: mappings, tiles, and categories. In: Fernandes, J., Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering III, LNCS, vol. 6491, pp. 92–165. Springer, Berlin (2011). doi:10.1007/978-3-642-18023-1_3
17. Diskin, Z., Maibaum, T., Czarnecki, K.: Intermodeling, queries, and kleisli categories. In: FASE 2012, Springer, Tallinn, Estonia (2012)
18. Drivalos, N., Kolovos, D., Paige, R., Fernandes, K.: Engineering a DSL for software traceability. In: Gaševic, D., Lämmel, R., Van Wyk, E. (eds.) Software Language Engineering, LNCS, vol. 5452, pp. 151–167. Springer, Berlin/Heidelberg (2009). doi:10.1007/978-3-642-00434-6_10
19. Drivalos-Matragkas, N., Kolovos, D.S., Paige, R.F., Fernandes, K.J.: A state-based approach to traceability maintenance. In: Proceedings of the 6th ECMFA Traceability Workshop, ACM, New York, NY, USA, ECMFA-TW'10, pp. 23–30 (2010). doi:10.1145/1814392.1814396
20. Eclipse Model Development Tools Project: Eclispe OCL website. http://www.eclipse.org/modeling/mdt/?project=ocl (2011)
21. Eclipsepedia: MDT/OCLinEcore. http://wiki.eclipse.org/MDT/OCLinEcorel (2012)
22. Ehrig, H. (ed.): Handbook on Graph Grammars and Computing by Graph Transformation, vol. 2. World Scientific, Singapore (1999)
23. Giese, H., Hildebrandt, S., Seibel, A.: Improved flexibility and scalability by interpreting story diagrams. In: Proceedings of GT-VMT 2009, ECEASST, vol. 18 (2009)
24. Groher, I., Reder, A., Egyed, A.: Incremental consistency checking of dynamic constraints. In: FASE 2009, Springer, LNCS, vol. 6013 (2010)
25. Gupta, A., Mumick, I.S.: Maintenance of materialized views: problems, techniques, and applications. IEEE Data Eng. Bull. **18**(2), 3–18 (1995)
26. Hegedüs, Á., Horváth, Á., Ráth, I., Branco, M.C., Varró, D.: Quick fix generation for DSMLs. In: IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2011, IEEE Computer Society (2011). doi:10.1109/VLHCC.2011.6070373
27. Hegedüs, Á., Horváth, Á., Ráth, I., Varró, D.: Query-driven soft interconnection of EMF models. In: ACM/IEEE 15th International Conference on Model Driven Engineering Languages and Systems, Springer, Innsbruck, Austria, LNCS, vol. 7590 (2012). doi:10.1007/978-3-642-33666-9_10
28. Heimbigner, D., McLeod, D.: A federated architecture for information management. ACM Trans. Inf. Syst. **3**(3), 253–278 (1985). doi:10.1145/4229.4233
29. Izsó, B., Szatmári, Z., Bergmann, G., Horváth, Á., Ráth, I.: Towards precise metrics for predicting graph query performance. In: 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013), pp. 412–431 (2013)
30. Jeusfeld, M.A., Jarke, M., Mylopoulos, J.: Metamodeling for Method Engineering. The MIT Press, Cambridge (2009)
31. Jouault, F.: Loosely coupled traceability for atl. In: Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany, vol. 91 (2005)
32. Kolovos, D.S.: Establishing Correspondences between Models with the Epsilon Comparison Language. In: Proceedings of the 5th European Conference on Model Driven Architecture—Foundations and Applications, Springer, Berlin, ECMDA-FA'09, pp. 146–157 (2009). doi:10.1007/978-3-642-02674-4_11

33. Kolovos, D.S., Paige, R.F., Polack, F.A.: On-demand merging of traceability links with models. In: Proceedings of 2nd EC-MDA Workshop on Traceability, Bilbao, Spain (2006, July)
34. Kolovos, D.S., Rose, L.M., Matragkas, N., Paige, R.F., Guerra, E., Cuadrado, J.S., De Lara, J., Ráth, I., Varró, D., Tisi, M., Cabot, J.: A research roadmap towards achieving scalability in model driven engineering. In: Proceedings of the Workshop on Scalability in Model Driven Engineering, ACM, New York, NY, USA, BigMDE'13, pp. 2:1–2:10 (2013). doi:10.1145/2487766.2487768
35. Lara, J., Guerra, E.: Deep meta-modelling with metadepth. In: Vitek, J. (ed.) Objects, Models, Components, Patterns. Lecture Notes in Computer Science, vol. 6141, pp. 1–20. Springer, Berlin (2010). doi:10.1007/978-3-642-13953-6_1
36. Limón, A.E., Garbajosa, J.: The need for a unifying traceability scheme. In: ECMDA Traceability Workshop (ECMDA-TW), pp. 47–56 (2005)
37. Del Fabro, Marcos Didonet, Valduriez, Patrick: Towards the efficient development of model transformations using model weaving and matching transformations. Software and Systems Modeling Special section paper. Springer, Berlin (2008). doi:10.1007/s10270-008-0094-z
38. MathWorks: Matlab Simulink. http://www.mathworks.com/products/simulink/ (2013)
39. Merks, E.: Xcore: Ecore meets Xtext. http://wiki.eclipse.org/Xcore (2013)
40. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: Proceedings of ICSE 2000, pp. 742–745 (2000)
41. OpenLink Software: Virtuoso Universal Server. http://virtuoso.openlinksw.com/ (2013)
42. Paige, R., Drivalos, N., Kolovos, D., Fernandes, K., Power, C., Olsen, G., Zschaler, S.: Rigorous identification and encoding of trace-links in model-driven engineering. In: Software and Systems Modeling pp. 1–19 (2010). doi:10.1007/s10270-010-0158-8
43. Ráth, I., Hegedüs, Á., Varró, D.: Derived features for EMF by integrating advanced model queries. In: Vallecillo, A., Tolvanen, J.P., Kindler, E., Störrle, H., Kolovos, D. (eds.) Modelling Foundations and Applications, LNCS, vol. 7349, pp. 102–117. Springer, Berlin (2012). doi:10.1007/978-3-642-31491-9_10
44. RDF Core Working Group: Resource Description Framework (RDF). http://www.w3.org/RDF/ (2004)
45. Rensink, A.: Representing first-order logic using graphs. In: International Conference on Graph Transformations (ICGT), LNCS 3256, pp. 319–335. Springer, Berlin (2004)
46. Rensink, A.: Time and space issues in the generation of graph transition systems. In: Mens, T., Schürr, A., Taentzer, G. (eds.), Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004), Electronic Notes in Theoretical Computer Science, vol. 127, pp. 127–139 (2005)
47. Rose, L., Kolovos, D., Drivalos, N., Williams, J., Paige, R., Polack, F., Fernandes, K.: Concordance: a framework for managing model integrity. In: Kühne, T., Selic, B., Gervais, M.P., Terrier, F. (eds.) Modelling Foundations and Applications, LNCS, vol. 6138, pp. 245–260. Springer, Berlin (2010). doi:10.1007/978-3-642-13595-8_20
48. Ráth, I., Bergmann, G., Ökrös, A., Varró, D.: Live model transformations driven by incremental pattern matching. In: Theory and Practice of Model Transformations, Springer, Berlin, LNCS, vol. 5063, pp. 107–121. doi:10.1007/978-3-540-69927-9_8 (2008)
49. SAE International: Architecture Analysis and Design Language (AADL). http://standards.sae.org/as5506a/ (2009)
50. Scheidgen, M.: On Implementing MOF 2.0-New Features for Modelling Language Abstractions (2005)
51. Schürr, A.: Introduction to PROGRESS, an attribute graph grammar based specification language. In: Nagl, M. (ed.) Graph-Theoretic Concepts in Computer Science, LNCS, vol. 411, pp. 151–165. Springer, Berlin (1990). doi:10.1007/3-540-52292-1_11
52. Sesame: RDF API and Query Engine. http://www.openrdf.org/ (2013)
53. The Eclipse Project: Eclipse Modeling Framework. http://www.eclipse.org/emf (2012)
54. The Eclipse Project: EMF Model Query 2. http://wiki.eclipse.org/EMF/Query2 (2012)
55. The Eclipse Project: EMFStore. http://www.eclipse.org/emfstore/ (2012)
56. The Eclipse Project: EMFT Search. http://www.eclipse.org/modeling/emft/?project=search (2012)
57. The Eclipse Project: The CDO Model Repository. http://www.eclipse.org/cdo/ (2012)
58. The Eclipse Project: Xtend: Modernized Java. http://www.eclipse.org/xtend/ (2013)
59. The Eclipse Project: Xtext: Language development made easy! http://www.eclipse.org/Xtext/ (2013)
60. The Object Management Group: Object Constraint Language, v2.0. http://www.omg.org/spec/OCL/2.0/ (2006)
61. The Object Management Group: OMG System Modeling Language (SysML). http://www.omg.org/spec/SysML/index.htm (2010)
62. TIBCO Developer Network: TIBCO Business Studio. http://developer.tibco.com/business_studio/ (2012)
63. Tolvanen, J.P., Rossi, M.: MetaEdit+: defining and using domain-specific modeling languages and code generators. In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM, New York, NY, USA, OOPSLA'03, pp. 92–93 (2003)
64. TopQuadrant Inc: SPIN support in TopBraid. http://www.topquadrant.com/products/SPIN.html (2013)
65. Ujhelyi, Z., Hegedüs, Á., Bergmann, G., Horváth, Á., Ráth, I., Varró, D.: EMF-IncQuery: an integrated development environment for live model queries. Sci. Comput. Program. (2014). doi:10.1016/j.scico.2014.01.004
66. Vanhooff, B., Berbers, Y.: Supporting modular transformation units with precise transformation traceability metadata. In: ECMDA-TW Workshop, SINTEF, pp. 15–27 (2005)
67. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Program. **68**(3), 214–234 (2007)
68. Varró, D., Pataricza, A.: VPM: a visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. J. Softw. Syst. Model. **2**(3), 187–210 (2003)
69. W3C: SPARQL Inferencing Notation. http://spinrdf.org/ (2013)
70. W3C: SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/ (2013)
71. Walderhaug, S., Johansen, U., Stav, E., Aagedal, J.: Towards a Generic Solution for Traceability in MDD. In: 5th ECMDA workshop on traceability, ECMDA conference (2006)
72. Willink, E.D.: Aligning OCL with UML. ECEASST 44 (2011)
73. Workflow Management Coalition: XML Process Definition Language, v2.1. http://www.wfmc.org/xpdl.html (2008)
74. 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, ACM, New York, NY, USA, ASE'07, pp. 164–173 (2007). doi:10.1145/1321631.1321657

Ábel Hegedüs is a research associate at the Budapest University of Technology and Economics and currently finishing his PhD under the supervision of Dániel Varró. His research focuses on the application of model-driven engineering for back-annotation, design space exploration and traceability. He is a core developer of the EMF-IncQuery project and a regular contributor for the VIATRA2 project. He has participated in multiple industrial and EU FP7 research projects. He won the ACM Distinguished Paper Award in 2011.

Ákos Horváth is a research fellow at the Budapest University of Technology and Economics. His main research interest is model-driven systems engineering with a special focus on model transformations and its application for the design of embedded systems. He is a major contributor for the VIATRA2 model transformation framework, and was responsible for the model-driven development framework in the avionics related DIANA European Union research project.

István Ráth is a research fellow at the Budapest University of Technology and Economics. His main research interest is model-driven systems development with a special focus on domain-specific languages and model transformations. He is the chief technological architect of the VIATRA2 model transformation framework, and a regular contributor of the SENSORIA and MOGENTES European Union research projects.

Rodrigo Rizzi Starr was born in Brazil in 1980. He received his engineering degree from the University of São Paulo in 2004 and his M.Sc. from the University of Santa Catarina in 2006. Since 2007, he is an engineer at Embraer S.A, working both on research and development and product design. His main interests are in flight control systems and automation of systems engineering activities.

Dániel Varró is an associate professor at the Budapest University of Technology and Economics. His main research interest is model-driven systems and services engineering with special focus on model transformations. He regularly serves on the programme committee of various international conferences in the field like MODELS, ASE, FASE and ICMT. He was a programme committee co-chair of FASE 2013 and ICMT 2014 conferences. He delivered a keynote talk at IEEE CSMR 2012 conference. He is a founder of the VIATRA2 and the EMF-IncQuery open-source Eclipse projects, and the principal investigator at his university of the SENSORIA, DIANA, SecureChange and MONDO European Projects. He is a three time recipient of the IBM Faculty Award. Previously, he was a visiting researcher at SRI International, at the University of Paderborn and twice at TU Berlin. In 2014, he was a visiting professor at McGill University and Université de Montréal.