Softw Syst Model (2016) 15:473–496
DOI 10.1007/s10270-014-0413-5

REGULAR PAPER

View-based model-driven software development with ModelJoin

Erik Burger · Jörg Henss · Martin Küster · Steffen Kruse · Lucia Happe

Received: 27 June 2013 / Revised: 28 March 2014 / Accepted: 2 April 2014 / Published online: 9 May 2014

Abstract Fragmentation of information across instances of different metamodels poses a significant problem for software developers and leads to a major increase in effort of transformation development. Moreover, compositions of metamodels tend to be incomplete, imprecise, and erroneous, making it impossible to present it to users or use it directly as input for applications. Customized views satisfy information needs by focusing on a particular concern, and filtering out information that is not relevant to this concern. For a broad establishment of view-based approaches, an automated solution to deal with separate metamodels and the high complexity of model transformations is necessary. In this paper, we present the *ModelJoin* approach for the rapid creation of views. Using a human-readable textual DSL, developers can define custom views declaratively without having to write model transformations or define a bridging metamodel. Instead, a metamodel generator and higher-order transformations create annotated target metamodels and the appropriate transformations on-the-fly. The resulting views, which are based on these metamodels, contain joined instances and can effectively express concerns unforseen during metamodel design. We have applied the ModelJoin approach and validated the textual DSL in a case study using the Palladio Component Model.

Keywords View-based modeling · Model-driven software development · Model transformation · Model-based query language

Communicated by Prof. Colin Atkinson.

E. Burger (✉) · J. Henss · L. Happe
Institute for Program Structures and Data Organization (IPD), Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany
e-mail: burger@kit.edu

J. Henss
e-mail: henss@kit.edu

L. Happe
e-mail: lucia.happe@kit.edu

M. Küster
FZI Forschungszentrum Informatik, Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany
e-mail: kuester@fzi.de

S. Kruse
OFFIS—Institute for Information Technology, Escherweg 2, 26121 Oldenburg, Germany
e-mail: steffen.kruse@offis.de

1 Introduction

While in early years of software development, roughly 90 % of a software was concerned with providing the primary services, today it is only about 40 %; the rest deals with platform-specific technologies, GUI, interoperability, security, and other dependability concerns [1]. In model-driven software development, processes suffer from two problems:

First, fragmentation of information across instances of different metamodels, which are used to address the specific viewpoints of the underlying domain [2]. This can lead to redundancies and inconsistencies if models share a semantic overlap and describe the same software system (e.g., an architecture model and an object-oriented class diagram). Large systems are often not represented by instances of a single metamodel, since existing metamodels and domain standards have to be respected. For example, the analysis of performance properties requires the usage of a performance-specific metamodel, while the design of the system is modeled in a general-purpose modeling language such as UML.

Second, the individual models are usually too complex to be grasped by a single person. Navigating large models is a

time-consuming and frustrating task [1]. Although detailed models of the software under development form the basis for facilitating more reasonable software project planning, the complexity of such models can lead to a poor effort distribution which is among the major causes of rework [3].

View-based approaches tackle these problems by supporting seamless access to autonomous, heterogeneous information sources eventually targeting a specific concern. *View-centric modeling* techniques [4,5] address this task by treating views as first-class entities of the modeling process: Information is stored in a central model, while users operate on custom partial views that represent only relevant information, and thus reduce the complexity for developers. From a model-driven perspective, a view is again a special model that conforms to a view type, i.e., a metamodel. To create a view, a software developer has to choose a view type or define a new one and implement model transformations from the central model to the view.

Although this concept theoretically solves the problem of fragmentation and complexity, existing view-based approaches fail to define a metamodel for the central model, since such a "super-metamodel" would have to cover all the possible viewpoints of software development. Furthermore, the definition of a view type requires the creation of a new metamodel and model transformations, which is a time-consuming and error-prone process.

We therefore propose the *ModelJoin* approach for view-based model-driven software development. ModelJoin implements the concept of dynamically created, so-called *flexible view types* [6], which can contain information from several distinct metamodels. ModelJoin features a human-readable domain-specific language (DSL) with a concrete textual syntax for the declarative description of customized views. With ModelJoin, developers can create new view types and views by writing a query in an SQL-like concrete textual syntax. The new metamodel for the view type and the transformations are generated automatically and on-the-fly by ModelJoin. This reduces the complexity of view definition and makes it possible to create customized views rapidly, since a ModelJoin query does not only define the view type, but also the actual set of elements which are contained in the resulting view.

The ModelJoin DSL is a specialized language for the definition of views over heterogeneous models. Existing query languages for metamodels serve a similar purpose, but are limited to one metamodel [7–9], require adaptations to the underlying metamodels [10] or the a priori definition of the target metamodel [11]. ModelJoin addresses these shortcomings while at the same time reducing the complexity for the creation of user-specific views. In contrast to all-purpose model transformation languages [12,13], it reduces the expressiveness for the sake of clarity and brevity and combines metamodel and transformation definition. Although related to model composition [14,15] and model merging [16–18], our approach is not based on automatic matching of structural similarities, but relies on the explicit declarative definition of semantic overlaps in the ModelJoin DSL by the developer.

With ModelJoin, existing metamodels are combined in a non-intrusive way, so developers can create views which aggregate information from loosely coupled models without having to modify the metamodels.

The contribution of this paper is a method for the specification of view types and views on heterogeneous models, including a formal definition of the ModelJoin DSL, and a prototypical implementation based on model-driven technologies such as metamodel generation and higher-order transformations. The approach is evaluated with a case study in the field of component-based software development, using the Palladio Component Model [19].

The remainder of this paper is structured as follows: Sect. 2 contains the foundations of the approach. In Sect. 3, we present the view-based development approach, followed by a scenario (Sect. 4), which will serve as a running example throughout the paper. Section 5 contains the formal definition of the ModelJoin DSL. The prototypical implementation is presented in Sect. 6, followed by a case study based on the running example (Sect. 7). The final sections contain related work (Sect. 8) and the conclusion with an outlook on future work in Sect. 9.

2 Foundations

2.1 View-based model-driven software development

The roots of view-based software development go back even before the era of object-oriented languages [20]. The first object-oriented methods like OMT [21] and Fusion [22] already featured several diagram types for structural, behavioral, and operational viewpoints. This was further extended in approaches like the 4+1 viewpoints by Kruchten [23], leading to today's standards like RUP [24] and UML [25], which contain several diagram types for the description of software architectures.

With the *Orthographic Software Modeling (OSM)* [5] approach, Atkinson et al. aim to establish views as first-class entities of a model-driven software engineering process. In their envisioned view-centric development process, all information about a system is represented in a single underlying model (SUM); thus, the SUM even transcends the function of being a model, but becomes the system itself. All access to the SUM is organized in user-specific, automatically generated views. Even executable source code is treated as only a special textual view. The views are synchronized via the central SUM and are not synchronized directly with each other. The approach has been implemented prototypically for

component-based systems in KobrA [26], which is based on UML, OCL, and the Atlas Transformation Language (ATL) [13]. The OSM approach is currently being developed further in the VITRUVIUS approach [6,27], which is based on a modular single underlying model that includes legacy metamodels and views. To retrieve information from the modular SUM, elements from multiple sub-models have to be integrated in specialized views. In the current state, these view types and transformations for the synchronization of the models and the views have to be created and maintained manually.

The terms *view* and *viewpoint*, on which the terminology in this paper is based, have been defined in the IEEE 1471/ISO 42010 standard [28,29]. It contains a definition for the terms *architecture view* and *architecture viewpoint*: The term *architecture view* is defined as a "work product expressing the architecture of a system from the perspective of specific system concerns," and *architecture viewpoint* is defined as a "work product establishing the conventions for the construction, interpretation, and use of architecture views to frame specific system concerns." These conventions may include "languages, notations, model kinds, design rules, and/or modeling methods, analysis techniques, and other operations on views."

Furthermore, the ISO 42010 standard differentiates between *synthetic* and *projective* approaches:

> In the synthetic approach, an architect constructs views of the system of interest and integrates these views within an architecture description using model correspondences. In the projective approach, an architect derives each view through some routine, possibly mechanical, procedure of extraction from an underlying repository. (from [29])

ModelJoin is suited for projective approaches, where the information about a system is represented in several models, from which the views are derived.

Model-driven software development [30] is based on models and transformations as primary artifacts. The model-driven architecture (MDA) standard, as defined by the Object Management Grouop (OMG) [31], uses the terms *view* and *viewpoint* in the sense of the ISO standard and thus contains the three viewpoints *computation independent viewpoint, platform independent viewpoint*, and *platform specific viewpoint* with the adjacent models CIM, PIM, and PSM. The OMG stack of modeling standards (MOF, QVT, OCL) and EMF [32] serves as a technical base for our implementation, which is described later in this paper (see Sect. 6).

Since the definitions of the IEEE 1471/ISO 42010 standard [28,29] give only a broad definition of the terms and do not distinguish between view types and actual view instances, we will use an extension of the definition of Goldschmidt et al. [4,33], given in Definition 1. The full terminology used in this paper is shown in Fig. 1.

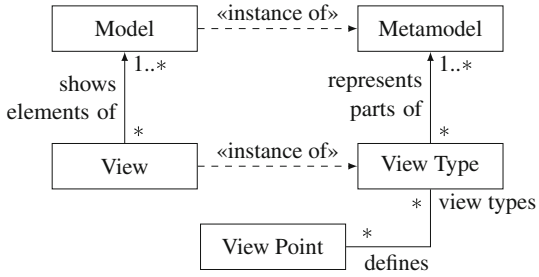


Fig. 1 Terminology for view-based modeling used in this article (adapted from [4])

Definition 1 (*View type*) A *view type* defines the set of metaclasses whose instances can be displayed by a view. It comprises a definition of a concrete syntax plus a mapping to the abstract metamodel syntax. The actual *view* is an instance of a view type showing an actual set of objects and their relations using a certain representation. A *viewpoint* defines a concern.

In UML, for example, the diagram types, such as sequence or class diagrams, are *view types*. An actual diagram containing classes *A*, *B*, and *C* is an example for a *view*. The static architecture or dynamic aspects of a system are *viewpoints* in UML.

The difference of Definition 1 to the IEEE/ISO standard is the introduction of the term *view type*. Technically, a view type is a metamodel of which actual views are instances. The view type is independent from the metamodel it represents; it may represent elements from, be a subset of, or be identical with another metamodel, but may also contain specially defined elements which are not part of any other metamodel. An actual view that is an instance of the view type can restrict, aggregate, and reorganize information from other models.

In contrast to the definition of Goldschmidt et al. [4], we have extended the definition so that a view type can represent parts of multiple metamodels. To define the elements which are represented by the view types and views, we introduce the notion of *scope* for view types.

Definition 2 (*Scope of view types*) The *projectional scope* of a view type is defined by the elements types which are part of the view type and their relation to elements on the metamodel which the view type represents (see Fig. 1). View types with a *single-metamodel projectional scope* represent only one metamodel. View types with a *multi-metamodel projectional scope* contain elements which represent elements from multiple metamodels. The *selectional scope* of a view type defines restrictions on the instance level of elements. Although the selectional scope is primarily defined in the actual views, the view type can also define instance-based limitations.

In the view type, elements such as classes, attributes, and references represent elements of the metamodels which are used to describe software systems. The projectional scope can be expressed as a mapping of the view type elements to the other metamodel elements and usually model-to-model transformations between them. The selectional scope can be defined by constraints on the view type metamodel, which is valid for all views that instatiate the view type, and by the (manual) selection of elements, which is specific for each view.

2.2 Query languages

Creating views for instances of heterogeneous metamodels is similar to the definition of views in relational databases [34], where views are defined as stored queries and can be used as virtual tables. Queries in relational databases combine data from tables with heterogeneous table schemata and define a new table schema for the result set, as well as the selection of elements in the result set. The table schema of the result set in relational databases corresponds to the view type in the model-driven world. Thus, a view in the model-driven world that offers comparable possibilities would have to comprise the definition of a view type and the selection of elements for the actual views.

Existing query languages for EMF [8,9,12] exploit the analogy to database systems and use an SQL-like textual concrete syntax, but offer only projectional operators. Thus, the result of a query contains only a subset of the original information. Thus, it is not possible to query information that is spread across instances of several metamodels and to display the result in an integrated metamodel directly. To achieve this, a target metamodel and model transformations would have to be specified, which introduces high efforts for the developer, since the metamodel and the transformations change as the information needs of the user change. Thus, the metamodel and the transformations have to be adapted manually.

3 Flexible view types for model-driven development

3.1 Motivation

Every software project that makes use of metamodeling techniques and models faces the problem that information about the system, which contains the entities of interest, is spread across instances of different metamodels. We will call such instances *heterogeneous models* in the following. Although representing an entity of interest in multiple formalisms may lead to redundancies, it may still be necessary to do so, either to represent various levels of abstraction, to describe different view points on the system, or because compatibility to legacy software requires the usage of specific metamodels.

In model-driven software development (MDSD) processes, heterogeneous models are used to describe a software system in different stages of development. For example, the system architecture may be described by instances of a component model, while the object-oriented design is described using UML class diagrams. The execution semantics of the system is defined by the implementation code using a general-purpose programming language like Java. Additionally, several domain-specific metamodels may be used to describe further aspects, such as network topologies, energy consumption, or the performance of the system.

We have identified the following problem areas that arise from the usage of heterogeneous models (illustrated with the MDSD example):

Traceability: The artifacts of heterogeneous models, such as components, classes, and code, are semantically related: Several classes implement a component, code conforms to the object-oriented design and realizes the architecture. This trace information is, however, rarely persisted: If at all, it is persisted in natural language definitions, such as the system documentation. Since there is no or little support by development frameworks, trace information is not available in the models themselves. If users have information needs that require gathering information from several different artifact types, they cannot rely on tool support, but have to aggregate this information manually.

Redundancy/consistency: If information about the same entity is represented by instances of heterogeneous metamodels, the same piece of information (e.g., a class name) is stored redundantly in several artifacts. If tracing information is missing, changes to a single artifact may lead to inconsistencies; if a user modifies a model, he or she may be unaware that this produces an inconsistency with another artifact. If the tracing information is present, additional efforts are necessary to represent the change in all artifacts and to resolve inconsistencies.

Evolution: If metamodels are modified, existing instances have to be migrated to the new version using co-evolution techniques [35,36]. In a scenario of heterogeneous interconnected metamodels, which can evolve independently, this imposes additional complexity, since the consistency and traceability links on the instance level and their construction rules on the metamodel level also have to be updated. In the case that existing metamodels have to be used either to meet standards, or because of other conventions that are not in the responsibility of the developers, it is often not possible to modify the metamodels. To integrate information from several metamodels, non-intrusive metamodel extension techniques [37] have to be used.

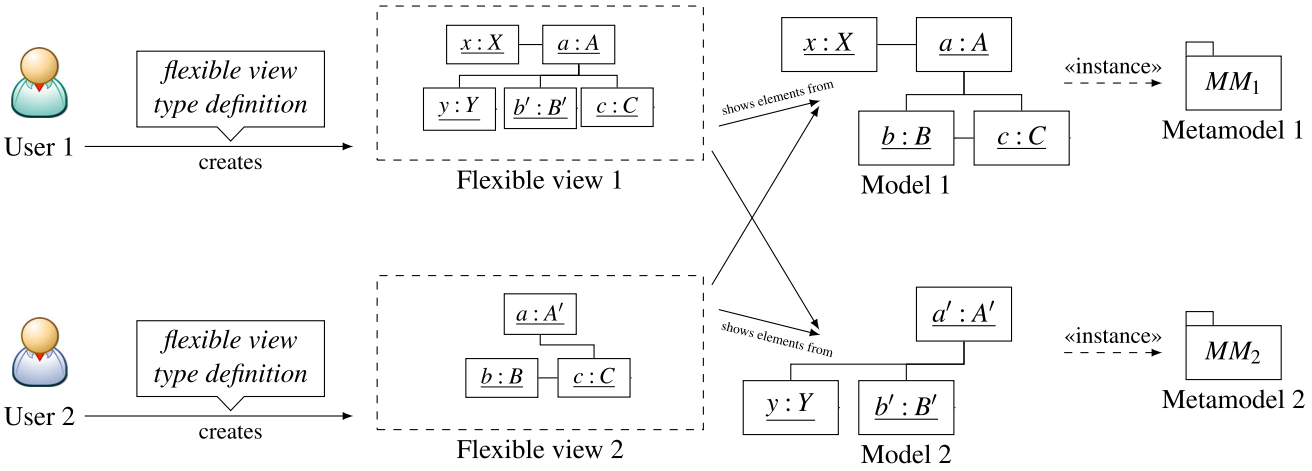


Fig. 2 *Flexible views* concept example

3.2 Flexible view types

View-based approaches try to tackle the aforementioned problems by decoupling the way a system is represented to the user from the way that it is technically represented in the infrastructure.

The available view types are usually predefined by standards or by the availability in development tools. Our concept of *flexible view types* offers custom, user-defined view types. It can be used by developers who would like to re-arrange, aggregate, or combine information that is distributed among instances of heterogeneous metamodels.

Definition 3 (*Flexible view type*) A *flexible view type* over a set of models contains:

- the definition of the *view type metamodel* and its *projectional scope*. The view type may have a single- or multi-metamodel projectional scope;
- the definition of the *selectional scope*, i.e., the selection of elements that is contained in the flexible view. This selection is based on instance properties;
- a set of rules for editability of the view and for the back propagation to the source models.

The abstract concept of flexible view types can be seen in Fig. 2: Two metamodels MM_1 and MM_2 and their respective instances *Model 1* and *Model 2* share a semantic overlap, i.e., different elements of MM_1 and MM_2 represent the same entity type in the system of interest. In the example, this is indicated by similar naming of the elements on metamodel level (A corresponds to A', etc.) and on model level (a corresponds to a', etc.). Each of the models carries additional information which is not available in the respective other model (elements x, y and the references to these elements). *User 1* wants to create a view that aggregates all information from *Model 1* and *Model 2* while at the same time displaying overlapping elements as one element. The resulting view (*Flexible view 1*) integrates information from both models and identifies the overlapping elements by a naming convention. *User 2* creates a view which shows the elements of type C in addition to the overlapping elements. The projectional and selectional scope of a flexible view type is specified in the *flexible view type definition*.

The concept of flexible views has been presented in a previous paper [6]. It has since been extended by the definition of view type scopes.

3.3 ModelJoin: a language for flexible view type definitions

ModelJoin serves as a description language for flexible view type definitions. With ModelJoin, developers can create flexible view types rapidly, having to specify neither the metamodel behind the view type nor the transformation rules needed between the source models and the view. These artifacts can be generated automatically using a ModelJoin expression as input.

A ModelJoin expression defines the projectional and selectional scopes of a flexible view type. It contains sufficient information to generate a metamodel for the view type, if such a metamodel does not exist already. Since ModelJoin supports multi-metamodel projectional scopes, users can satisfy information need on the underlying system that cannot be answered by examining the single models independently. ModelJoin does not contain a specification of editability and synchronization yet; thus, the view types created with ModelJoin are read-only. The generated view types and transformations can, however, be used as a starting point for more sophisticated, editable views. If a view type is to be extended for editability, developers can adapt the target metamodel and the transformations manually and can generate and customize editors for the target metamodel.

The problem areas which we have mentioned in Sect. 3.1 are addressed by ModelJoin in the following way: The *trace-*

ability between heterogeneous artifacts can be improved by creating custom view types which integrate information based on properties of the artifacts, such as name equality or common identifiers. This way, integrated views can be created for heterogeneous models that are not explicitly linked. *Redundancies* in the models themselves are not reduced by ModelJoin, since the approach is non-intrusive and does not change the source models or metamodels. It is, however, possible to create integrated views which are free of redundancies, and which can be used to determine if *inconsistencies* in the source models exist. In the case of metamodel *evolution*, the flexible view types created in ModelJoin coevolve automatically, since they are generated from the definition in the ModelJoin DSL. This definition may of course have to be adapted to the new version of the metamodel with respect to refactorings or renames. The effort of adapting a ModelJoin expression is, however, significantly lower than the effort of adapting the view types, instances, and transformations manually.

4 Motivating example

To illustrate the view type definition and the definition of flexible views (Definition 3), we give an example from the field of model-based performance analysis. The two main metamodels used throughout this section are as follows: First, a description of a component-based software architecture and its relevant performance influences; second, the results of a performance prediction, which are stored in a data model that defines different sensors.

Technically, these models are independent from each other since the sensors can be used for storage of various data items (beyond software performance data). The idea of our flexible view concept is to combine the information stored in the two models without explicit postprocessing of the data. We bring together information from the software architecture description, such as components and interfaces, and from the performance prediction, such as sensors, measurements, and experiments. The integrated views that are generated from the query statements can be used for specific analyses. We give examples of what types of questions can be answered by combining the two models.

4.1 Software architecture model

The *Palladio Component Model (PCM)* is a component metamodel used to predict performance and reliability of component-based software systems. It consists of several sub-metamodels which are used to model a software system. A software architecture is modeled as a set of components which provide services. The actual behaviors of services are modeled as abstract Service Effect Specifications (SEFFs). A deployment model is used to store the mapping of components to modeled hardware resource containers. The current Palladio metamodel comprises ∼100 classes and ∼200 other metamodel elements. For a more detailed description of the PCM, we refer to a journal paper [19] or the Palladio Technical Report [38].

Taking into account the following characteristics of a software system, the performance is predicted using a simulation:

1. *Usage profile:* The number of users and their access rates and the services that are used.
2. *Internal structure:* An abstraction of the internal structure of components, i.e., control- and data flow.
3. *Dependent services:* External services that are used by the system. Usually a black box view of the performance-relevant behavior.
4. *Resource environment:* Hardware- and virtual nodes and execution environments on which the components run.

The complete Palladio Component Model is too large to reproduce it here. Instead, we illustrate only those elements referenced in the example queries. Elements from three different packages are referenced (see Fig. 3). BasicComponents and CompositeComponents are the types of components that can be instantiated by an AssemblyContext. The component types are defined as part of a repository

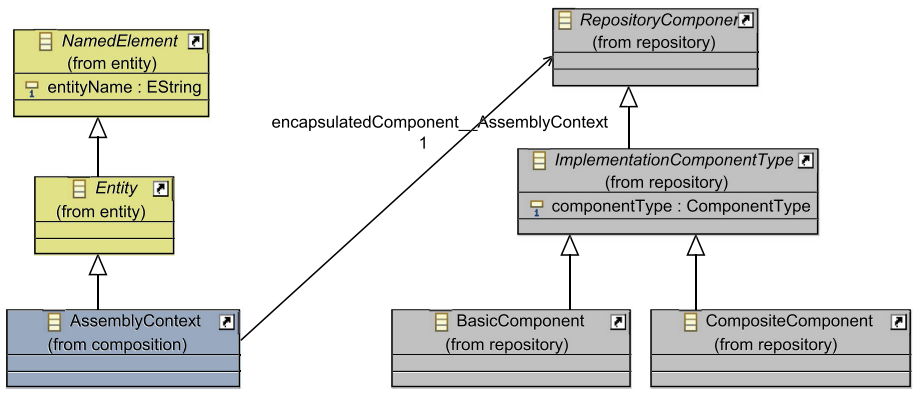


Fig. 3 Extract of the Palladio Component Metamodel, strongly simplified from the PCM technical report [38]

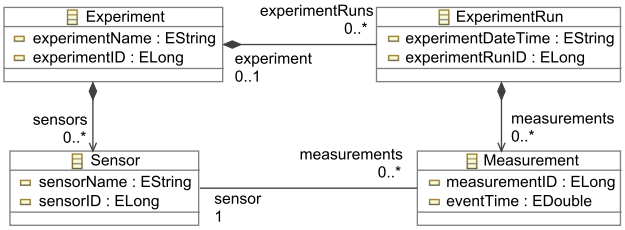


Fig. 4 Metamodel of the sensor framework

mechanism. After definition of the types, the structural model (from package *composition*) defines how they are assembled and connected. The *entity* package defines commonly used attributes such as entityName.

4.2 Performance data model

To perform a prediction on the modeled system, the model is transformed by the Palladio simulator to an executable simulation model using a model-to-text transformation. During the execution of the simulation, measurements usually have to be carried out to obtain information on the chosen quality metrics. Currently, two kinds of measurements are supported:

- TimeSpanMeasurements capture time durations, e.g., when measuring the time span between receiving a request and sending the corresponding response;
- StateMeasurements capture state changes of simulated entities, e.g., the number of active requests in a system.

Measurements are always appendant to a unique sensor that is attached to a simulated entity of interest. Furthermore, each measurement is assigned to an experiment run, as parameters of the simulation can vary throughout. The experiment runs belong to an experiment that also contains the various sensors. The simulation uses a specific model to persist the measurements from the simulation sensors. Furthermore, Experiments and ExperimentRuns can be used in this model to group measurements. Figure 4 shows the corresponding metamodel. The model used as input of the simulation and the result model were intentionally not tightly linked, to enable the analysis of results without needing the input model.

4.3 Usage scenarios

We illustrate in the following the different usage scenarios of flexible view types based on the two models discussed so far. The concept is exemplified using an integration of software architecture models with performance prediction results.

Integration of data from different models Architecture and performance prediction models can represent the same entity from different perspectives. The software architecture model represents static structures, such as components, interfaces, and their deployment to virtual or hardware nodes. When some execution semantics is added to the components as an abstraction of the performance-relevant attributes, the software can be simulated producing a (usually large) set of measurement data. These data are not directly connected to the software model, but represented as instances of the Sensor Framework metamodel (Fig. 4). The semantic overlap is given by the sensorName of a Sensor. Usually, this is the unique identifier of an assembled component. Now, several questions can be asked regarding the connection of the two models.

- *What is the simulated response time at a specific point in the architecture, e.g., for a BasicComponent in* Fig. 3? To answer this question, the data from the sensor model must be attached to the software architecture model.
- *What is the average utilization of a hardware node over multiple experiment runs?* To answer this, the information from the sensor model must be joined with the hardware node from the software architecture model.

A manual approach to these questions would be, first, to write some specialized code that attaches the data to the software model. Of course, methods for all different queries have to be written. Second, a new model can be created combining the two models (decorating the two) making explicit the semantic overlap by directly referencing the respective elements. After that, a model transformation can derive the attributes of the decorating model. This solution might suffer from scalability issues since for all combinations of model elements, new matching elements of the decorator have to be created. Writing the transformation for each new query will be cumbersome.

Instead, we propose to specify the query based on the two (meta-)models and have a derived transformation take care of the integration of the two models. This transformation is automatically generated from the query and the two metamodels.

On-the-fly generation of view types The abovementioned scenario of integrating two models that share a semantic basis, but are not connected, raises a second question. The result of a query that integrates the two models can be represented by a new metamodel. For example, one could specialize the Component element from the software architecture model by subclassing and adding attributes from the sensor model. This solution fits, however, only one specific query (asking for the response time of a component).

Flexible view types can be used to specify the elements from the two metamodels that should be combined or selected. Developers can derive the needed elements from the two models and combine them using a flexible view type definition. For example, references can be added on-the-fly

to support navigability. The view type metamodel is volatile in the sense that it is generated each time its definition is changed. It can be, however, persisted for future reuse.

5 The ModelJoin language

In this section, we will present the abstract syntax of the ModelJoin domain-specific language (DSL) and give a formal definition of the semantics of the operators.

5.1 Language design

The ModelJoin DSL is inspired by relational algebra, where queries can be used on different relational schemata. When a query on a relational database is executed, the result set contains relations that instantiate a new relation schema which depends on the query itself. In SQL, for example, the table schema of the result of a query is dependent on the columns chosen in the `SELECT` clauses, renaming operators (`AS`), `JOIN` operators, and other constructs. If we transfer this concept to the model-driven world, it means that a query defines a new metamodel for the results. The result set is a model that instantiates this metamodel. It combines information from heterogeneous models into a single result model.

Existing query languages for Ecore-based models offer, however, either only projectional operations on instances of a single metamodel (e.g., EMFQuery [8]), thus making queries on heterogeneous models impossible, or require a fixed predefined result metamodel to execute a query on heterogeneous models (e.g., QVT [12], ATL [39]). To overcome this limitation, ModelJoin includes operators for selection, projection, and joining of elements from heterogeneous models. In contrast to the aforementioned existing query languages and engines, the result of the evaluation of a ModelJoin expression contains elements which are instances of a new target metamodel that is generated during execution of the query.

In the following subsections, we will define the ModelJoin operators, which are semantically similar, but not equivalent, to *Selection*, *Projection*, and *Join* of relational algebra.

A ModelJoin expression takes at least two models as input, called *source models* in the following, which conform to the *source metamodels*. The evaluation of a ModelJoin expression returns a result set, called the *target model*, which conforms to the *target metamodel* (see Fig. 5).

To distinguish between the elements on the metamodel level and on the model level, we will refer to the elements of the metamodel as *classes*, *attributes*, and *references*. The elements on the model level, i.e., the instances, will be denominated as *objects*, *attribute values*, and *links*, respectively.

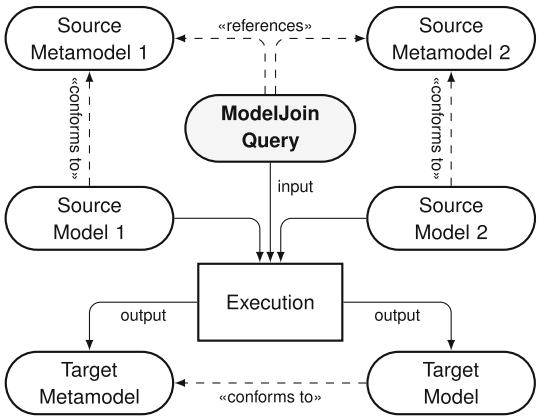


Fig. 5 ModelJoin target and source models

5.2 Set notation of metamodels

Our approach is based on the Ecore metamodel of the Eclipse Modeling Framework [32]. For the formal definitions of the semantics of ModelJoin, we will use the set notation of the OCL specification [40, Appendix A] for metamodel entities and instances, which is based on complete MOF [41]. Since our approach is based on Ecore, the definition listed in the following does not contain concepts such as object-valued attributes, associations with more than two ends, and role names, which are part of MOF, but not of Ecore.

5.2.1 Set notation of Ecore

Here, we list the subset of the OCL specification which we will use for the definition of the ModelJoin abstract syntax. We only give a brief description for the elements here; for a full definition of the elements, see [40, Appendix A].

- $\textsc{Class}$ is the set of class names with a generalization hierarchy $\prec$
- $\mathcal{N} = \textsc{Class} \cup \mathcal{T} \cup \textsc{Att} \cup \textsc{Ref}$ is the set of named elements
- $\mathcal{T}$ is the set of type names where $t_c \in \mathcal{T}$ is the type of a class $c \in \textsc{Class}$
- $\mathcal{T}_B \subset \mathcal{T}$ is the set of hard-coded basic (=primitive) type names; $\mathcal{T}_B = \{UnlimitedNatural, Integer, Real, Boolean, String\}$
- $\textsc{Att}$ is the set of all attribute signatures. The set of attribute signatures $\textsc{Att}_c$ of a class $c \in \textsc{Class}$ is defined as $a : t_c \to t; t \in \mathcal{T}_B$
- $\textsc{Ref}$ is the set of reference names. $\textsc{Ref}_c$ is the set of references of a class $c \in \textsc{Class}$. A reference $r \in \textsc{Ref}_c$ has a signature $associates(r) = \langle c, c' \rangle \in \textsc{Class} \times \textsc{Class}$
- The cardinality of attribute signatures and references is expressed by function $multiplicities(a) = N$ which assigns each attribute or reference a non-empty set $N \subseteq \mathbb{N}_0$ with $N \neq \{0\}$

- $\sigma_{\text{CLASS}}, \sigma_{\text{ATT}}, \sigma_{\text{REF}}$ are snapshot functions that return a set of all instances of a given class, attribute or reference, together forming the *system state*
- $I(c) = \{\underline{c}_1, \underline{c}_2, \ldots\}$ is the set of possible instances of a class $c \in \text{CLASS}$
- $L(r)(\underline{c}_1) \subseteq I(c_2)$ is the set of instances that are linked to $\underline{c}_1 \in I(c_1)$ via a reference r with the signature $associates(r) = \langle c_1, c_2 \rangle$

The term *system state* in the OCL definition denotes the set of instances of the metamodels.

Definition 4 (*Metamodel*) A *metamodel* is a structure

$m := (\text{CLASS}, \text{ATT}, \text{REF}, associates, multiplicites, \prec)$

To distinguish between source and target metamodels, we will write $M_{source} = \{m_{source_1} \cup m_{source_2} \cup \ldots\}$ and M_{target} with the respective sets CLASS_{source}, CLASS_{target}, ATT_{source}, ATT_{target}, and so on, which together form $\mathcal{N}_{source}$ and $\mathcal{N}_{target}$.

Note that the elements $c, c' \in \text{CLASS}$ represent class names; hence, $c = c'$ expresses that the (simple) names of two elements are identical, but not object identity. The identity of classes is expressed by the identity of their types: $t_c = t_{c'}$.

5.2.2 Type system

The OCL standard contains a generalization hierarchy $\prec$ with the reflexive extension $\preccurlyeq$. The sets ATT_c and REF_c contain the attributes and associations for a class c. The sets ATT_c^* and REF_c^* additionally contain the attributes and associations inherited from all superclasses of c.

The OCL standard contains the primitive types *UnlimitedNatural*, *Integer*, *Real*, *Boolean*, and *String*. The operator $=_t$ only allows the comparison of elements of the exact same type, so it is not possible to compare, e.g., integers with real numbers. We weaken this requirement for ModelJoin and allow coercion for the number types *UnlimitedNatural*, *Integer*, and *Real*.

5.3 Abstract syntax

The ModelJoin language is declarative, so the ModelJoin expressions describe the desired properties of source and target elements after the execution of the query. In the formal definition, we express this as relations between the source and target sets. A ModelJoin expression is a relation between a set of n source metamodels and one target metamodel:

$$q \in \mathcal{Q} = \langle m_{source_1}, m_{source_2}, \ldots, m_{source_n}, m_{target} \rangle \in M^{n+1}$$

Since the result of the execution of a ModelJoin query contains not only the metamodel, but also instances, the relation is twofold. The ModelJoin expressions will be described as such in the following: First, the signature of the expressions and the properties of the elements of the target metamodel are defined; second, the properties of the system state (i.e., the instances) are defined. We will define the effects on the system state as boundary conditions of the target metamodel. The properties of the target metamodel only depend on the source metamodel and the query, but not on its instances, so the target metamodel can always be computed via static analysis of a ModelJoin expression and the source metamodels. Thus, the same ModelJoin expression can be used with different instances, using the same target metamodel.

There are four kinds of ModelJoin expressions:

- *join* expressions $\bowtie \in \mathcal{J}$
- *keep* expressions $\kappa \in \mathcal{K}$
- *selection* expression $\varsigma \in \mathcal{S}$
- *rename* expressions $\rho \in \mathcal{R}$

Thus, $\mathcal{Q} = \mathcal{J} \cup \mathcal{K} \cup \mathcal{R} \cup \mathcal{S}$. These expressions are described in detail in the following subsections. We will define them as relations over the sets of classes, since they are not functions in the mathematical sense. The element of the target class represents the "return value" of the expression.

Definition 5 (*Mapping relation*) The target metamodel and model contain elements that *represent* elements in the source metamodels and models (cf. Fig. 1). To express this, we introduce a *mapping relation* both on metamodel and on model level. A named element $e \in \mathcal{N}_{source}$ is *mapped* to a named element $e' \in \mathcal{N}_{target}$ with the uniquely defined relation

$$\sim_{\bowtie} = \{\langle e, e' \rangle \in \mathcal{N} \times \mathcal{N} \mid e \text{ is mapped to } e'\}$$

The elements in $\mathcal{N}$ are on metamodel level. Thus, the mapping relation $\sim_{\bowtie}$ is also defined on metamodel level. For possible instances $I(c), I(c')$ of classes $c \in \text{CLASS}_{source}, c' \in \text{CLASS}_{target}$, the *instance mapping relation* is defined on model level as

$$\sim_{\underline{\bowtie}} = \{\langle \underline{c}, \underline{c}' \rangle \in I(c) \times I(c') \mid \underline{c} \text{ is mapped to } \underline{c}'\}$$

5.3.1 Join expressions

The core concept of our approach is the joining of model elements from heterogeneous models. This may be used for elements which represent the same concept in two different metamodels (cf. the example in Fig. 2). The *join operators* are defined over two metamodel classes as input and return a target class, which is newly created in the target metamodel with a specified name. Join operations are ternary relations over two source metamodels and one target metamodel. It is,

however, also possible to join more than two source metamodels by cascading join operations, so that the target metamodel of one join operation serves as the source metamodel of another join operation. Furthermore, classes can be joined with themselves.

In analogy to relational algebra, we define a *natural join* operator, which joins classes based on identically named attributes that have a compatible type. We call these attributes *join-conforming*. If two classes are joined with a natural join, join-conforming attributes in both of the classes are added to the resulting class in the target metamodel, similar to the columns of the result table of a relational join operation. On instance level, two objects are joined if they have the same values in the join-conforming attributes, and a corresponding object is created in the target model.

Definition 6 (*Join conformity*) Let $a_1 \in \text{ATT}_{c_1} : t_{c_1} \rightarrow t_1$ and $a_2 \in \text{ATT}_{c_2} : t_{c_2} \rightarrow t_2$ be attributes of classes $c_1, c_2 \in$ CLASS. *Join conformity* is a property on metamodel level. It is given if two attributes have the same name, type (allowing coercion), and multiplicities:

$$\cong_{\text{ATT}} = \{\langle a_1, a_2\rangle \in \text{ATT}_{c_1} \times \text{ATT}_{c_2} \mid (a_1 = a_2) \wedge (t_1 = t_2) \wedge (multiplicities(a_1) = multiplicities(a_2))\}$$

For two classes c_1 and c_2, all possibly joinable attributes are contained in the set of *join-conforming attribute pairs*:

$$A^{\bowtie}_{c_1,c_2} = \{\langle a_1, a_2\rangle \in \text{ATT}^*_{c_1} \times \text{ATT}^*_{c_2} \mid a_1 \cong_{\text{ATT}} a_2\}.$$

On the instance level, two objects $\underline{c}_1 \in I(c_1)$, $\underline{c}_2 \in I(c_2)$ fulfill *instance-conformity* if they carry equal values in their join-conforming attributes:

$$\cong_I = \{\langle \underline{c}_1, \underline{c}_2\rangle \in I(c_1) \times I(c_2) \mid \forall \langle a_1, a_2\rangle \in A^{\bowtie}_{c_1,c_2} \left(\sigma_{\text{ATT}}(a_1)(\underline{c}_1) = \sigma_{\text{ATT}}(a_2)(\underline{c}_2)\right)\}$$

With these helper sets and relations, we will now define the join operations.

Definition 7 (*Natural join*) For two classes $c_1 \in \text{CLASS}_{source_1}$, $c_2 \in \text{CLASS}_{source_2}$ and a target class $c' \in \text{CLASS}_{target}$, the *natural join* is defined as

$$\bowtie = \langle c_1, c_2, c'\rangle \in \text{CLASS}_{source_1} \times \text{CLASS}_{source_2} \times \text{CLASS}_{target}$$

where the target class and its instances have the following properties:

- The mapping relation holds for each of the source classes and the target class: $(c_1 \sim_{\bowtie} c') \wedge (c_2 \sim_{\bowtie} c')$
- For each of the join-conforming attribute pairs in the source classes, an attribute of the same name and type exists in the target class: $\forall \langle a_1, a_2\rangle \in A^{\bowtie}_{c_1,c_2} \exists\, a' \in \text{ATT}_{c'}\left((a' : t_{c'} \rightarrow t_1) \wedge (a_1 = a') \wedge (a_1 \sim_{\bowtie} a') \wedge (a_2 \sim_{\bowtie} a')\right)$
- For all instance-conforming pairs in the source models, an instance that has the same attribute values in the join-conforming attributes exists in the target model: $\forall \langle \underline{c}_1, \underline{c}_2\rangle \in \sigma_{\text{CLASS}}(c_1) \times \sigma_{\text{CLASS}}(c_2)\left(\underline{c}_1 \cong_I \underline{c}_2 \Rightarrow \exists\, \underline{c}' \in \sigma_{\text{CLASS}}(c')(\underline{c}' \cong_I \underline{c}_1)\right)$

When executing a natural join expression, the target class c' is always created and contains only the common attributes. If no common attributes exist, the target class is generated without attributes. This is different to the natural join in relational algebra in two ways: Firstly, the natural join in ModelJoin does not add the other non-join-conforming attributes to the target class; secondly, it does not degenerate to the cartesian product if no common attributes exist. To add attributes to a class in the target metamodel, the *keep attributes* expression is used (see Sect. 5.3.2). In contrast to the projectional approach in relational algebra, we use a constructive way of building the target metamodel. The name of the target class is by default set to the c_1 and can be changed with the *rename* expression.

ModelJoin furthermore provides a *outer join* operator, which also creates instances of the target metamodel for unmatched instances of elements in one of CLASS_{source}.

Definition 8 (*Outer join*) The *outer join* operator is equivalent to the *natural join* operator in its type signature and the constraints on the target metamodel. Deviating from the natural join, the result set $\{\underline{c}'_1, \underline{c}'_2, \ldots\}$ contains a respective instance for each instance in the source model, regardless of instance-conformity:

$$\forall \underline{c}_1 \in \sigma_{\text{CLASS}}(c_1) \exists\, \underline{c}' \in \sigma_{\text{CLASS}}(c')\left(\underline{c}' \cong_I \underline{c}_1\right) \wedge \forall \underline{c}_2 \in \sigma_{\text{CLASS}}(c_2) \exists\, \underline{c}' \in \sigma_{\text{CLASS}}(c')\left(\underline{c}' \cong_I \underline{c}_2\right)$$

In addition to the general outer join, there is a *left outer join* and *right outer join* operator which only creates instances for the left class (c_1) and right class (c_2), respectively.

The natural and outer joins are specialized operators for the most common cases of attribute equality in heterogeneous classes, when attributes have the same name and a compatible type. The join condition can, however, be generalized from join conformity to arbitrary logical conditions (depending on the actual language used in the implementation) on the instances of the source metamodels. In analogy to relational algebra, we call this operator *theta join*.

Definition 9 (*Theta join*) For source classes $c_1 \in \text{CLASS}_{source_1}$, $c_2 \in \text{CLASS}_{source_2}$, a target class $c' \in \text{CLASS}_{target}$, and a logical expression $\theta = I(c_1) \cup I(c_2) \rightarrow true, false$, the *theta join* is defined as

$$\bowtie_\theta = \langle c_1, c_2, c'\rangle \in \text{CLASS}_{source_1} \times \text{CLASS}_{source_2} \times \text{CLASS}_{target}$$

where the target class c' has the following properties:

– The mapping relation holds from the source classes to the target class: $(c_1 \sim_{\bowtie} c') \wedge (c_2 \sim_{\bowtie} c')$
– For all pairs in the source models for which the join condition θ holds, an instance exists in the target model: $\forall \langle \underline{c}_1, \underline{c}_2 \rangle \in \sigma_{\text{CLASS}}(c_1) \times \sigma_{\text{CLASS}}(c_2) \big(\theta(\underline{c}_1, \underline{c}_2) \Rightarrow \exists\, \underline{c}' \in \sigma_{\text{CLASS}}(c') \big((\underline{c}_1 \sim_{\underline{\bowtie}} \underline{c}') \wedge (\underline{c}_2 \sim_{\underline{\bowtie}} \underline{c}')\big)\big)$

The target class c' does not have to contain any attributes from the source classes; if desired, they have to be added manually by a *keep attributes* statement. (This behavior is different to the theta join of relational algebra, where all columns are added to the result table.)

The *theta join* is the most general of join operators, since it can contain an arbitrary join condition. In the prototypical implementation, these conditions can be expressed in OCL. Natural and outer joins can theoretically be expressed by a theta join operator where the θ-expression contains the join conformity constraints, and appropriate *keep attribute* expressions, which add the join-conforming attributes.

5.3.2 Keep expressions

The *keep* operator defines additional structural features (i.e., attributes and references) and supertype relations of the target model which are not defined by the join operators. It serves a purpose which is similar to the projection operator in relational algebra, but unlike projection, the keep operator is constructive: if there is no explicit keep statement, no attributes (apart from those which are added because they are part of a join condition) or references are included in the target metamodel. The rationale behind this behavior is to avoid the potentially high number of attributes and references inherited from superclasses. Keep operators can be applied to classes which have been mapped by join operators or other keep operators.

There are different operators for the inclusion of attributes, references, and supertype relations. For the definition of references in set notation, we assume the existence of functions $associates()$ and $multiplicities()$ that express the respective properties of a reference as described in [40].

Definition 10 (*Keep attributes*) Let $a : t_c \to t \in \text{ATT}_c^*$ be an attribute of class $c \in \text{CLASS}_{source}$ (directly defined or in one of its superclasses) and $c' \in \text{CLASS}_{target}$ a class in the target metamodel with $c \sim_{\bowtie} c'$. The *keep attributes* operator is then defined as

$$\kappa_{att} = \langle a, a' \rangle \in \text{ATT}_c \times \text{ATT}_{c'}$$

where the target attribute a' has the following properties:

- a' is an attribute of the target class c': $a' : t_{c'} \to t \in \text{ATT}_{c'}$
- a' has the same name and multiplicity as the attribute a in the source class c. In case, the target class c' was created by an outer join, it is necessary to allow empty values for the attribute, so the lower boundary of the multiplicity of a' must always be 0:

$$(a \sim_{\bowtie} a') \wedge (a = a') \wedge (multiplicities(a') = multiplicities(a) \cup \{0\}).$$

- The instances of c' carry the same attribute values as those instances of c that they are mapped to. For unmapped instances, the attribute value is null ($\perp$):

$$\forall \underline{c}' \in \sigma_{\text{CLASS}}(c') : \sigma_{\text{ATT}}(a')(\underline{c}') = \begin{cases} \sigma_{\text{ATT}}(a)(\underline{c}) & \text{if } \exists \underline{c} \in \sigma_{\text{CLASS}}(c) \mid \underline{c} \sim_{\underline{\bowtie}} \underline{c}' \\ \perp & \text{else} \end{cases}$$

Attributes can also be defined as an *aggregation* of values in the source model, similar to the SQL feature `GROUP BY`.

Definition 11 (*Aggregation function*) An aggregation function is defined as $f_\alpha : S \to \mathbb{R}, S \in \mathbb{R}^n$

ModelJoin supports the following five arithmetic aggregation functions:

– sum: $f_{\text{sum}}(s_1, \ldots, s_n) = \sum_{i=1}^{n} s_i$
– average: $f_{\text{avg}}(s_1, \ldots, s_n) = \frac{1}{n} \sum_{i=1}^{n} s_i$
– maximum: $f_{\text{max}}(s_1, \ldots, s_n) = \max(s_1, \ldots, s_n)$
– minimum: $f_{\text{min}}(s_1, \ldots, s_n) = \min(s_1, \ldots, s_n)$
– size: $f_{\text{size}}(s_1, \ldots, s_n) = n$

The aggregation operator groups elements by a certain reference through which they are linked to the source class. The result of the aggregation is then persisted in an attribute in the target class.

Definition 12 (*Aggregation*) Let $r \in \text{REF}_{source}$ be a reference between classes $c, \hat{c} \in \text{CLASS}_{source}$, i.e., the reference signature is $associates(r) = \langle c, \hat{c} \rangle$, and let $\hat{a} : t_{\hat{c}} \to t \in \text{ATT}_{\hat{c}}^*$ be an attribute of class $\hat{c}$ which is of a numeral type $t \in \{UnlimitedNatural, Integer, Real\}$, and let $c' \in \text{CLASS}_{target}$ be a class in the target metamodel with $c \sim_{\bowtie} c'$, and let f_α be an aggregation function. The *aggregation* operator is then defined as

$$\alpha = \langle r, \hat{a}, a' \rangle \in \text{REF}_c \times \text{ATT}_{\hat{c}} \times \text{ATT}_{c'}$$

where the aggregate result a' has the following properties:

– a' is an attribute of the target class c' with type t: $a' : t_{c'} \to t \in \text{ATT}_{c'}$
– The instances of c' carry an attribute value in a' that is determined by an aggregation function f_α those instances of c that they are mapped to. For unmapped instances, the attribute value is null ($\perp$):

$$\forall \underline{c}' \in \sigma_{\text{CLASS}}(c') : \sigma_{\text{ATT}}(a')(\underline{c}') = \begin{cases} f_\alpha \left(\bigcup_{\hat{\underline{c}} \in L(r)(\underline{c})} \sigma_{\text{ATT}}(\hat{a})(\hat{\underline{c}}) \right) & \text{if } \exists \underline{c} \in \sigma_{\text{CLASS}}(c) \mid \underline{c} \sim_{\underline{\bowtie}} \underline{c}' \\ \bot & \text{else} \end{cases}$$

Depending on the type of the aggregation function, the aggregation operator is written as α_{sum}, α_{avg}, α_{max}, α_{min}, or α_{size}.

To generalize the aggregation operation, ModelJoin also allows generic calculated attributes in the target model. The values of these attributes are derived from arbitrary values of source instances. They are, however, called *calculated* and not *derived* in ModelJoin, since the term *derived attribute* is defined in EMF as an attribute that is derived from other properties of the same instance. A calculated attribute in ModelJoin depends on properties of source instances which are not linked to the target instance in any way.

Definition 13 (*Calculate attribute*) Let $a' : t_{c'} \rightarrow t \in \text{ATT}^*_{c'}$ be an attribute of type t in a target class $c' \in \text{CLASS}_{target}$ and $\phi = \mathcal{N}^n \rightarrow t$ a function over arbitrary elements with the return type t. The *calculate attribute* operator is defined as

$$\delta_\phi = \langle e_1, \ldots, e_n, a' \rangle \in \mathcal{N}^n_{source} \times \text{ATT}_{c'}$$

where the attribute values of a' are defined by the function ϕ over instances $\underline{e_1}, \ldots, \underline{e_n} \subseteq \{\sigma_{\text{CLASS}} \cup \sigma_{\text{ATT}} \cup \sigma_{\text{REF}}\}$:

$$\forall \underline{c}' \in \sigma_{\text{CLASS}}(c') : \sigma_{\text{ATT}}(a')(c') = \phi(\underline{e_1}, \ldots, \underline{e_n})$$

The operator for calculated attributes is the most general way of defining attributes in the target model. The keep attributes operator and the aggregations of Definition 12 can also be expressed by calculated attributes. In actual use cases, the function ϕ will very likely (but not necessarily) be over classes c_1, c_2 with $c_1 \sim_\bowtie c'$, so that the calculation is based on instances $\underline{c}_1, \underline{c}_2$ that have been mapped to the target instance by another operator. In the prototypical implementation, we use the general-purpose language OCL [40] for the definition of calculated attributes.

The *keep references* operator defines which references exist in the target metamodel. It can only be applied to classes that have already been mapped, either at the start point of the reference (*keep outgoing*) or at the end point of the reference (*keep incoming*). If the class at the other side of the reference has not been mapped yet by another join or keep operator, it is generated in the target metamodel.

Definition 14 (*Keep references*) Let $r \in \text{REF}_{source}$ be a reference between classes $c, \hat{c} \in \text{CLASS}_{source}$, i.e., the reference signature is $associates(r) = \langle c, \hat{c} \rangle$, and $c' \in \text{CLASS}_{target}$ a class in the target metamodel with $c \sim_\bowtie c'$. The *keep references* operator is defined as:

$$\kappa_{ref} = \langle r, r' \rangle \in \text{REF}_{source} \times \text{REF}_{target}$$

where the target reference r' has the following properties:

- r' is defined between the classes that are mapping targets of the classes of r. $associates(r') = \langle c', \hat{c}' \rangle \wedge (\hat{c} \sim_\bowtie \hat{c}')$; $\hat{c}' \in \text{CLASS}_{target}$
- Since there may be target instances where the reference is not set, the multiplicity of r' is extended by 0: $multiplicities(r') = multiplicities(r) \cup \{0\}$
- For every instance pair of c and $\hat{c}$ that is linked by r, a mapped instance pair of c' and $\hat{c}'$ also exists that is linked by r': $\forall \langle \underline{c}, \hat{\underline{c}} \rangle \in \sigma_{\text{CLASS}}(c) \times L(r)(\underline{c}) \exists \langle \underline{c}', \hat{\underline{c}}' \rangle \in \sigma_{\text{CLASS}}(c') \times L(r')(\underline{c}') \mid \underline{c} \sim_{\underline{\bowtie}} \underline{c}' \wedge \hat{\underline{c}} \sim_{\underline{\bowtie}} \hat{\underline{c}}'$

The evaluation of κ_{ref} creates the class $\hat{c}'$, if it does not exist in the target metamodel, and creates the reference r' reference between c' and $\hat{c}'$. In the prototypical implementation, the keep references operator is differentiated into a *keep outgoing* and *keep incoming* operator, depending on the direction of the reference.

If several classes are created in the target metamodel that have a common superclass in the source model, common attributes or references have to be created in each of the single classes if they should be included in the target metamodel. To avoid this redundancy, it is also possible to include super- or subtype relations of the source metamodels in the target metamodel with the *keep supertype* and *keep subtype* operators. Again, ModelJoin does not automatically create any of these inheritances. They have to be made explicit by the author of the ModelJoin expressions. If the respective super- or subclass is not present in the target metamodel, it is created during the execution of the ModelJoin expression.

Definition 15 (*Keep supertype*) Let $c, \hat{c} \in \text{CLASS}_{source}, c' \in \text{CLASS}_{target}$ be classes with $c \prec \hat{c}$ and $c \sim_\bowtie c'$. The *keep supertype* operator is defined as:

$$\kappa_{super} = \langle c, c' \rangle \in \text{CLASS}_{source} \times \text{CLASS}_{target}$$

$$\text{with } \exists \hat{c}' \in \text{CLASS}_{target} \mid (c' \prec \hat{c}') \wedge (\hat{c} \sim_\bowtie \hat{c}')$$

Definition 16 (*Keep subtype*) Let $c, \hat{c} \in \text{CLASS}_{source}, c' \in \text{CLASS}_{target}$ be classes with $c \succ \hat{c}$ and $c \sim_\bowtie c'$. The *keep subtype* operator is defined as:

$$\kappa_{sub} = \langle c, c' \rangle \in \text{CLASS}_{source} \times \text{CLASS}_{target}$$

$$\text{with } \exists \hat{c}' \in \text{CLASS}_{target} \mid (c' \succ \hat{c}') \wedge (\hat{c} \sim_\bowtie \hat{c}').$$

It should be noted that the $\kappa_{super}/\kappa_{sub}$ operators do not automatically alter the signature of attributes or references, i.e., it does not move attributes or references to a superclass. This has to be made explicit in the respective κ_{att} and κ_{ref} operators. The keep super-/subtypes operators have no effect on the system state of the target metamodel.

5.3.3 Select

The selection operator restricts the result set to a subset of elements for which a logical predicate is fulfilled. Since only

the set of instances is reduced by this operator, selection does not have any impact on the generated metamodel.

Definition 17 (*Selection*) Let $I(c)$ be a set of instances of a class $c \in \text{CLASS}$ and $\varphi = J \rightarrow \{true, false\}$, $J \subseteq I(c)$ be a logical expression over instances $\underline{c} \in I(c)$. The *selection* operator is defined as an unary operator

$$\varsigma_{\varphi}(I(c)) = \{\underline{c} \in I(c) \mid \varphi(\underline{c})\}$$

Our prototypical implementation uses OCL statements for the logical expressions, since it is based on QVT. In general, any other language could be used.

5.3.4 Rename

Since the *join* and *keep* operations take the entity names from the first of the source elements as the name of the target element, a *rename* operator is needed to specify the entity names in the target metamodel.

Definition 18 (*Rename*) Let $e, e' \in \mathcal{N}$ be a names. *Rename* is an unary operation $\rho_{e'}(e) = e'$.

In the prototypical implementation, naming conflicts which arise from rename operations are detected and presented in the editor, so that the user can resolve them manually.

5.4 Conformity between metamodels

The target metamodel can be completely derived from the ModelJoin query. By default, it is created during the execution of the query. If a compatible target metamodel already exists, it is, however, desirable to use this existing metamodel, since further tools and processes, such as graphical editors or model transformations, may already be defined for this metamodel and can be reused.

To determine whether an existing metamodel is compatible to a ModelJoin expression, we will use the following conformity relation on metamodel level:

Definition 19 (*Metamodel conformity*) Let m_1, m_2 be metamodels and $I(m_1)$, $I(m_2)$ the sets of all possible instances of m_1 and m_2. Metamodel conformance is defined as

$$conforms(m_1, m_2) \Leftrightarrow I(m_1) \subseteq I(m_2)$$

Metamodel conformity expresses that all instances of one metamodel are also valid instances of another metamodel. This means that if a ModelJoin expression q_1 defines the target metamodels m_{target_1}, then a metamodel m_{target_2} can be used for both expressions if $conforms(m_{target_1}, m_{target_2})$ holds. The metamodel m_{target_2} may be defined by a ModelJoin expression q_2, but can in general be an arbitrary metamodel. The conformance relation between two metamodels can be checked using our approach for state-based conformance checking [42].

5.5 Assumptions/limitations

ModelJoin poses no assumptions on the structure or other properties of the source metamodels. It is, however, necessary that the semantic overlap between the source models can be explicitly specified, so that meaningful flexible view types can be defined on heterogeneous models. In simple cases, such as identically named identifiers, the natural join operators can be used to combine elements. In more complex cases, the generic operators (theta join, calculate attribute) with their OCL predicates can be used to specify the semantic connection between the source models. Thus, ModelJoin is only limited to the expressiveness of OCL.

ModelJoin cannot be used to create arbitrary view types, since the target metamodel can only contain classes and features that are derived from those in the source metamodels. It is possible to use existing metamodels as target metamodels, if the conformity relation (Definition 19) holds between the generated and the existing target metamodel. ModelJoin is, however, neither suitable as a textual definition language for metamodels nor as a general-purpose transformation language.

It is a design rationale of ModelJoin that the developer controls the elements that are included in the view type by explicitly specifying the relations between them and the source models. Thus, our approach does not contain techniques for identifying structural similarities of different metamodels. ModelJoin assists the developer in the rapid definition and creation of views, but not in determining the semantic overlap of heterogeneous models.

6 Technical aspects

We have implemented the ModelJoin language defined in Sect. 5 as an extension to the popular Eclipse modeling platform.[1] Thus, a seamless integration with existing Eclipse-based modeling tools is achieved.

The abstract syntax of the ModelJoin DSL has been defined as an Ecore metamodel with an Xtext textual syntax. Based on this textual syntax, a textual editor was developed. This makes it easy for the user to specify queries in a natural, easy-to-read way. Features such as auto-completion of metamodel elements, syntax-highlighting and on-the-fly validation, help the user to write queries on models fast and elegantly.

For transforming the models, i.e., executing the query, a target metamodel is synthesized from the ModelJoin query

[1] http://www.eclipse.org/modeling/emf/.

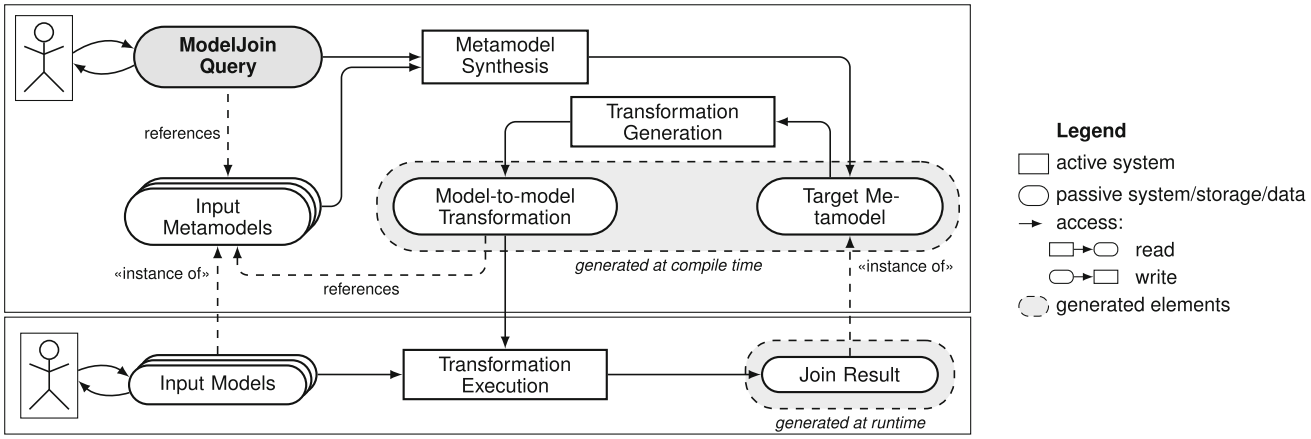


Fig. 6 Model workflow (in FMC notation [43])

and the corresponding input metamodels. It does not contain any direct references to the source metamodels. This generated metamodel is transient in the sense that it changes when the query is changed. The model-to-model transformation from the source models to the target model is generated as QVT-Operational code, using Xtend2 textual templates. Both these artifacts, the target metamodel and the transformation executing the query, are generated at runtime, usually after saving the ModelJoin query. This workflow is depicted in Fig. 6.

As running a query usually means executing several steps, we chose to employ the Eclipse Modeling Workflow Engine 2 (MWE2) [44] in complement to the editing support. MWE2 eases loading of metamodels, handling of model slots, invoking transformations, and the orchestration of workflow steps. The sources of our prototypical ModelJoin implementation are available on the ModelJoin wiki page.[2]

In the following, we outline the different steps needed to execute a query.

6.1 Concrete textual syntax

The grammar for ModelJoin queries has been specified in Xtext. Developers can use the generated Eclipse editors to create queries interactively or can use the API to create queries programmatically. The full grammar definition can be found in [45] and is not reproduced here. Instead, we will use the example in Listing 1 to explain the textual syntax.

The query in Listing 1 realizes the model-based performance analysis scenario from Sect. 4. It defines a flexible view type that integrates information from the component model and the performance results. The query header (lines 1–3) defines the source and target metamodels m_{source_1}, m_{source_2}, and m_{target}. The semantic overlap in the models lies in the classes TimeSpanSensor from the Sensor Framework metamodel and AssemblyContext from the Palladio Component Model (PCM): A time span sensor contains the simulation results for a component in a certain assembly context. The overlap can be determined by the identifier attribute id of an assembly context, which is a substring of the attribute sensorName in the corresponding sensor. In the query, this is realized in line 6 as a theta join $\bowtie_\theta$ (see Definition 9) with an OCL condition in the `WHERE` clause. The `AS` clause determines the name AssemblyContext of the class in the view type and realizes the rename operator ρ. The names of the assembly context as well as the name of the sensor are included with a keep attributes operator κ_{att} in line 7. The component which is bound by the assembly context is included with a κ_{ref} operator (lines 8–12) and is named Component. To distinguish between basic and composite components in the view type, the two subtypes of Component are included by κ_{sub} operators in lines 11–12. The statistic information of the sensor framework is specified in lines 13–20. In addition to the name and identifier of the sensor, the actual results are aggregated with α operators to display the size, average value, and extrema of the time spans. These values are represented as one object per experiment run in the source model, but reduced to a numerical attribute in the resulting view.

In the concrete syntax of ModelJoin, curly braces are used for nesting of operators. This improves the readability of the expressions since the target class of keep statements need not be specified explicitly every time, and since the statements are grouped by the elements in the target metamodel.

6.2 Metamodel synthesis

The metamodel synthesis module is used to generate the target metamodel. It uses an algorithm that, based on a parsed model representation of a ModelJoin query, creates the result classes, attributes, and references accordingly.

The implemented generation algorithm can be run in two different modes: validation and generation. In the former

[2] http://sdqweb.ipd.kit.edu/wiki/ModelJoin.

mode, only warnings and errors of the provided query, leading to an incomplete target metamodel, are collected. The actual synthesis and persistence of the target metamodel is then only performed, using the latter mode, when no errors were detected priorly. As a result, the algorithm is usually executed twice.

In the following, a coarse description of the metamodel generation algorithm will be given. For a detailed description of the algorithm, we refer the reader to the ModelJoin technical report [45].

```
1  import "platform:/plugin/de.uka.ipd.sdq.pcm/model/pcm.ecore"
2  import "platform:/plugin/edu.kit.ipd.sdq.mdsd.sensormodel/model/Sensor.ecore"
3  target "http://sdq.ipd.kit.edu/mdsd/ComponentSpeed/0.2"
4
5  theta join Entities.TimeSpanSensor with pcm.core.composition.AssemblyContext
6  where "TimeSpanSensor.sensorName.indexOf(AssemblyContext.id)␣>␣0" as jointarget.AssemblyContext {
7         keep attributes pcm.core.entity.NamedElement.entityName, Entities.Sensor.sensorName
8         keep outgoing pcm.core.composition.AssemblyContext.encapsulatedComponent__AssemblyContext as type
              jointarget.Component {
9               keep attributes pcm.core.entity.NamedElement.entityName
10              keep subtype pcm.repository.BasicComponent as type jointarget.BasicComponent
11              keep subtype pcm.repository.CompositeComponent as type jointarget.CompositeComponent
12        }
13        keep incoming Entities.Experiment.sensors as type jointarget.Experiment {
14              keep attributes Entities.Experiment.experimentName, Entities.Experiment.experimentID
15              keep outgoing Entities.Experiment.experimentRuns as type jointarget.Run {
16                    keep attributes Entities.ExperimentRun.experimentRunID, Entities.ExperimentRun.
                          experimentDateTime
17                    keep aggregate size(Entities.ExperimentRun.measurements) as jointarget.Run.
                          measurementCount,
18                      avg(Entities.TimeSpanMeasurement.timeSpan) over Entities.ExperimentRun.
                            measurements as jointarget.Run.avgTime,
19                      min(Entities.TimeSpanMeasurement.timeSpan) over Entities.ExperimentRun.
                            measurements as jointarget.Run.minTime,
20                      max(Entities.TimeSpanMeasurement.timeSpan) over Entities.ExperimentRun.
                            measurements as jointarget.Run.maxTime
21              }
22        }
23 }
```

Listing 1 Response Time ModelJoin Example

During the execution of the algorithm, the metamodel synthesis component keeps track of already mapped classes from the source metamodel in a trace model, realizing the relation $\sim_{\bowtie}$ of Definition 5. It uses this information to map further references to the source classes to the corresponding target classes accordingly. The trace model is embedded in the generated metamodel using the Ecore annotation mechanism.

The algorithm builds upon the set of metamodel operations introduced by Hermansdörfer et al. [36]. Required operations for creating the metamodel are extracted from the query by recursively traversing the statement-tree. Each kind of operation is stored in a separate set, similar to the command pattern described by Gamma et al. [46].

As the first step of the algorithm, a *create package* operation is added matching the target base package definition in the query. This package is later on used as default container for the target metamodel classes. As next step, the join statements are processed in the sequence of their occurrence in the query. For each statement, a *create class* operation, resembling the joined target class, is added.

For every natural join statement, the given classes are analyzed for join-conforming features, i.e., attributes having the same name and types being type compatible. Type compatibility is defined here according to the definition in Definition 5.2.2. For each join-conforming feature identified, corresponding creation commands are added as well.

As next step, operations specifying required annotations (see Sect. 6.3) for both joined classes and attributes are added. Moreover, the OCL join conditions are added for every theta join.

After all join statements have been processed, the keep statements, associated with each join, are translated to metamodel operations. As keep statements can be nested inside other keep statements, with the exception of κ_{att} statements, a recursive descent is performed. Every keep statement is evaluated in the context of its parent keep or join statement:

– For all *keep supertype* and *keep subtype* statements (κ_{super}, κ_{sub}), *create class* operations are added for the

Table 1 Operations used for the metamodel synthesis

Structural primitives	$\bowtie$	$\kappa_{super/sub}$	κ_{ref}	κ_{att}, δ_{ϕ}, α
Create package	x	x	x	–
Create class	x	x	x	–
Create attribute	x	–	–	x
Create reference	–	–	x	–
Create data type	–	–	–	x
Create enum	–	–	–	x
Non-structural primitives				
Add super type	–	x	–	–

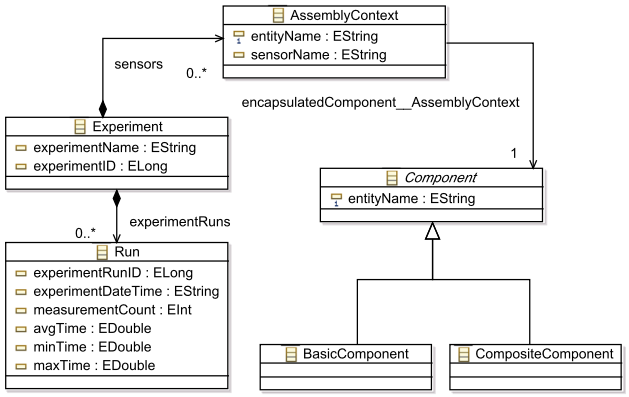


Fig. 7 Generated target metamodel for component speed example

super or subclass the statement refers to. Furthermore, *add super type* operations are created accordingly.
- For all *keep reference* statements (κ_{ref}), *create class* operations are added either for the target of the reference, in the case of keep outgoing, or the source of the reference for keep incoming statements. Moreover, *create reference* operations are created, referring to the mapped class.
- For all *keep attribute*, *calculate attribute* and *aggregation* statements (κ_{att}, δ_{ϕ}, α), *create attribute* operations are added. If an attribute refers to a non-standard data type, a corresponding *create data type* or *create enum* operations is added.

If any class to be created is situated in a sub-package of the target package, *create package* operations are added correspondingly. An overview on the created operations is shown in Table 1. In addition to these operations, annotations specifying the provenance of classes, references, and attributes are added.

After all metamodel operations have been extracted, they are executed in a specific sequence:

- First, all *create package* and *create class* operations are performed. Furthermore, data types, enums, and literals are created.
- Second, the *add super type* operations are performed to define the hierarchy of classes.
- Last, based on the hierarchy, references and attributes are added. Starting from the topmost set of classes, it is checked which references and attributes can be added to the class. When an operation is encountered that is meant to create an attribute or reference already present in a superclass, it is discarded as it is subsumed by the existing.

By first collecting all create operations, it is easy to detect synthesis conflicts. A naming conflict is usually detected when one of the operation sets already contains an equal create operation resulting from another statement. In the case of conflicts, e.g., when a class already has a structural feature with the requested name, the corresponding statement is ignored and a warning is thrown. The user is then required to take manual care of found naming conflicts by using renaming operators.

During the execution of operations, errors can occur, e.g., when referred entities of an operation are missing in the target metamodel. These errors are then traced back to the conflicting statement and indicated to the user.

The generated metamodel for the running example can be seen in Fig. 7.

6.3 Annotated target metamodel

Since both the transformation generation and the metamodel generation need to know the relations from the input metamodels to the target metamodel, the ModelJoin query would have to be parsed twice to calculate the relation. Since this would lead to duplicate code and redundant parsing of the query, we decided to annotate the target metamodel with tracing information.

During the generation of the target metamodel, the metamodel generator extracts information from a ModelJoin expression. The generator determines which elements of the source metamodel were joined to an element of the target metamodel, which was the attribute for the join condition, etc. The information is stored in the target metamodel using EAnnotation elements which reference the elements in the source metamodels directly. Thus, the generated target metamodel will contain references to the source metamodels, but only inside EAnnotation elements. EAnnotations have a name (called source in EMF), an element that they refer to (reference), and can contain additional information details in key/value pairs. Since EAnnotations cannot be differentiated through subtypes, we have introduced naming conven-

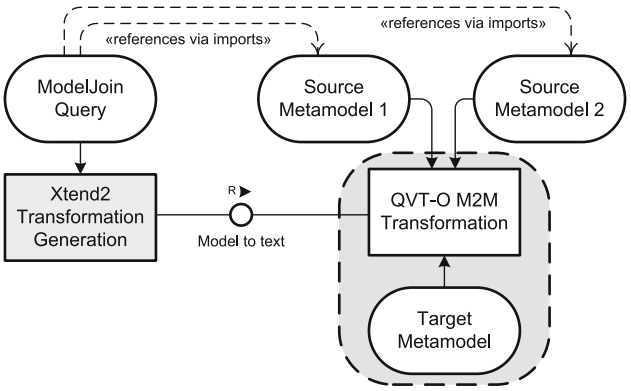


Fig. 8 Transformation generation

tions for the different annotation types. The target metamodel can be reused to create several target models.

6.4 Transformation generation

A ModelJoin query is executed as a model-to-model transformation taking the input models and resulting in a model that conforms to the generated joint metamodel (as outlined in Sect. 6.2). In order to generate the transformation, we used a model-to-text (M2T) approach based on Xtend2 (see Fig. 8), which generates the transformation directly from templates.

We chose QVT-O [12] as transformation language because of its stability, debugging support, and Eclipse integration.

The following steps are necessary when automatically generating the transformation in the Xtend2 Transformation Generation step of Fig. 8.

1. Create the OCL expression to filter joinable elements. In accordance with Definition 6, joinable means that they have the same name and are type compatible.
2. Convert the (optional) *where*-clause to an OCL selection on the joined elements.
3. Create mappings to transfer attributes from source to target model (only those marked as "keep").
4. Create mappings to transfer references from source to target model (only those marked as "keep"). Note that this usually includes setting values to the newly created meta classes in the target model.
5. Combine the single fragments generated in the preceding steps 1.-4. to one transformation.

Let us consider the exemplary template shown in Fig. 9. The theta join of FirstClass and SecondClass means that the model elements are joined using a condition expressed as an OCL constraint. The resulting QVT-O script will have two mappings for realizing the join, as QVT-O has no support for n-to-1 mappings. While the first mapping is used for checking the condition of the theta join and instantiating the `TargetClass`, the second mapping is used solely for storing tracing information. The tracing information is later on required for resolving the source instances of a target instance. To be more specific, they are put to use in the mappings created for transferring attributes and references.

```
theta join FirstClass with SecondClass where "OCL-condition"
as TargetClass

mapping FirstClass:
    thetaJoin_FirstClass_SecondClass_To_TargetClass
    (rightElement : SecondClass) : TargetClass
when {
    OCL-condition
}
{ -- create the target instances
    end {
        rightElement.map thetaJoin_update_SecondClass(result);
    }
}
mapping SecondClass::thetaJoin_update_SecondClass(rightElement
    : TargetClass) : TargetClass {
    init {
        result:=rightElement;
    }
}
```

Fig. 9 The QVT template for a theta join

In addition to the shown code, the `main` method of the QVT-O script is extended to call the generated mappings for the cartesian product of the sets of instances of FirstClass and SecondClass. Mappings for natural joins are created in a similar manner by using the joinable features, identified during synthesis, as join criteria. For realizing the left outer join functionality, the sets of instances that the mappings are called with are adjusted accordingly.

6.5 Transformation execution

As last step of the ModelJoin query evaluation process, the generated QVT-O transformation is executed to automatically create instances for the synthesized metamodel (step Model-to-model transformation in Fig. 6). Currently, our workflow uses the QVT-O engine that is part of the Eclipse M2M project. For every source metamodel, a corresponding model can be defined in the workflow all of which are then used as input models for the transformation engine. After the transformation workflow has been executed, the resulting model can be visualized.

Transformation execution can be repeated for multiple instances of the source metamodels without having to create the target metamodel and transformations again, thus creating new views for the generated view type.

6.6 Reuse of target metamodels

Since the formal definition of the ModelJoin operators only demands the existence of a target metamodel with certain

properties, target metamodels can be reused if the generated target models are valid instances of the target metamodels.

Multiple execution of the same ModelJoin expression on varying instances of the same source metamodels is the trivial case of target metamodel reuse: Since the target metamodel is identical, it can always be reused. In the prototypical implementation, this is achieved by executing the generated QVT-O transformation on the varying instances.

If the ModelJoin expression is modified, the target metamodel of the original expression can be reused if the metamodel conformity (Definition 19) holds between the new and the original target metamodel. We use our state-based conformance checking of [42] to determine the conformance relation. This allows us to use arbitrary existing metamodels, even if they were not originally generated by ModelJoin. Since the conformance check and the transformation generation require an (annotated) target metamodel as input, the metamodel generation step cannot be omitted even if an existing target metamodel is used. The reuse of the target metamodel still has the advantage that existing graphical editors and further transformations which are based on this metamodel can be used for the results of the query execution.

7 Case study

7.1 Example

For the evaluation of the ModelJoin language, we have conducted a case study using the Palladio Component Model and the Sensor Framework metamodel, as presented in Sect. 4. We used the MediaStore example from [19] and simulated performance properties of the MediaStore system. The performance measurements were persisted as instances of the Sensor Framework Metamodel. Since these metamodels are only loosely coupled, i.e., do not contain direct references to each other, the mapping of simulation results to the originating components and their elements has to be performed manually. We used ModelJoin to create a custom partial view type that only displays components, their name, and the assembly context in which they reside. In addition, the adjacent time span measurements are displayed, which have the attributes *measurement count*, and the three statistical values *mean, standard deviation*, and *variance*.

7.2 Empirical study

The envisioned benefit of ModelJoin is the reduced effort for the definition of flexible views, in comparison with the manual creation of target metamodels and transformations, which are generated automatically by the implementation of ModelJoin. Since this generated code is usually less readable and more complex than code that has been written manually, the benefit of ModelJoin in comparison with (its own) generated artifacts is of little interest here. Instead, we compared ModelJoin to a manual implementation of the same view type. The ModelJoin implementation generates Ecore metamodels and QVT-O transformation code, so we chose these languages for the comparison.

Table 2 Metrics for the manual implementation of the component speed example

Metric	P_1	P_2	P_3	P_4	*Gen*	*MJ*
Metamodel						
Classes	4	8	5	5	6	
Attributes	4	14	0	5	7	
References	3	6	7	2	2	
Inheritance	1	8	2	2	3	
Transformation						
Source LoC	46	76	53	50	179	18
# Operations	6	10	10	5	24	9

We asked researchers with experience in transformation design to implement the view type and the transformation using QVT-O. Four participants ($P_1 - P_4$) were asked to create a partial view which integrates information from the Palladio Component Model and the Sensor Framework Result Model. The participants were familiar with the Palladio Component Model and the transformation language QVT-O. The measured sensor statistic values (mean, standard deviation, and variance) should be assigned to the corresponding elements in the component model.

The task was given to the participants as a description in natural language. No detailed information was provided on where the desired information can be found in the respective metamodels. The task sheet can be downloaded from the ModelJoin wiki page.[3]

7.3 Evaluation

We used the M2M quality measurement framework[4] to analyze the QVT-O implementations. The results are depicted in Table 2. Column *Gen* contains the values of the generated artifacts of ModelJoin; column *MJ* contains the values of the expression in the ModelJoin textual syntax.

Although such a small sample can deliver only limited statistically significant results, it gives us a starting point for the evaluation. Table 3 shows the statistics of the case study. S is the standard deviation of the sample, $\bar{P}$ contains its average.

[3] http://sdqweb.ipd.kit.edu/wiki/ModelJoin/Experiment.

[4] http://code.google.com/p/m2m-quality/.

Table 3 Statistical evaluation of the empirical results

Metric	$S(P_{1..4})$	$\bar{P}$	*Gen/MJ*	T	Significance level
Cl.	1.732	5.50	6	−0.577	–
Attr.	5.909	5.75	7	−0.423	–
Ref.	2.380	4.50	2	**2.100**	80 %**
Inh.	3.202	3.25	3	0.156	–
LoC	13.475	56.25	18	**5.677**	99 %*
# Ops	2.630	7.75	9	−0.951	–

∗ one-tailed test $(1-\alpha)$
∗∗ two-tailed test $(1-\alpha/2)$

T is the test statistic calculated by $\sqrt{n}(\bar{P}-\mu_{MJ})/S$. $1-\alpha$ (one-tailed test) or $1-\alpha/2$ is the significance level at which the null hypothesis can be rejected.

We applied Student's one-sample t test [47] in order to analyze the data. For the case of the transformation, we calculated the significance levels using the one-tailed t statistic as we assumed that the experimental results would be larger than the ModelJoin query. The metamodels were analyzed using a two-tailed test. This is due to the fact that we did not know in advance if the experimental results would have larger or smaller metamodels than the automatically generated ones.

The results clearly indicate that the effort of manually creating views on heterogeneous models is high and that the specialized ModelJoin DSL reduces this effort. The null hypothesis H_0 stating that there is no difference between the length of the definition with ModelJoin and the hand-written QVT-O transformation can be rejected at a significance level of 99 %. This significance level results from the alternative hypothesis H_1 stating that the average number of lines of code of the hand-written QVT-O transformations is higher than the reference given by the ModelJoin query, i.e., $\mu > \mu_{MJ}$. There is no significant difference in the number of operations between hand-written and automatically generated transformation code.

Analyzing the sizes of the manually defined metamodels and the metamodel generated from the ModelJoin query, we got significant results only for the case of the number of references. The alternative hypothesis that there is a difference, i.e., $\mu \neq \mu_{MJ}$, only holds for the number of references in the metamodel which is significantly higher in the experimental setup than in the automatically generated case. From this observation, we conclude that the participants of the case study have over-engineered their solution and added more to the model than needed by the (very precise) task specification. The numbers of classes, inheritances, and attributes which are comparable to the reference solution can be explained by the task specification. A simple noun-extraction approach led to a solution in line with the reference. To avoid over-engineering of the solution, and to save time and effort, a ModelJoin approach answering the question was beneficial in the outlined case.

Summarizing, we draw the following conclusions from the result of the case study:

1. ModelJoin offers a compact way for the definition of flexible views, which can be seen by the number of modeled artifacts and lines-of-code of the manual QVT-O transformations and metamodel definitions in comparison with the size of the ModelJoin expression.
2. Manually implemented solutions for the creation of custom views tend to differ in the number of references used. This promotes the usage of a domain-specific language for view definition to get a minimal but sufficient solution for the requested query.

7.4 Biases and threats to validity

In this section, we describe threats to validity of the case study and limitations of the findings.

Internal validity The case study does not suffer from typical construct validity problems as is did not treat a population. Rather, the participants served as reference candidates implementing a typical model integration problem. The selection of the candidates was aligned with the expertise they had in model-driven engineering. All four participants are researchers with 2–4 years of experience in modeling, model transformations, and view definitions. They all had experience with the Palladio Component Model, none was familiar with the Sensor Framework. The time to solve the task was not limited. The non-random sampling of individuals could lead to an overestimation of similarities, and thus an underestimation of the variance, of the population.

External validity Since the participants are few and not representative, it is unclear whether the study results can be generalized to other situations. Reasons for that include the fact that the population is very small and not representative. The task, however, that we defined for the experiment, is typical for model-driven or model-based software engineering. From the experience with earlier projects, we collected use case scenarios that went into the definition of flexible views and the development of ModelJoin.

8 Related work

Many of the problems that are encountered in view-based modeling have counterparts in database research. Relational databases offer the possibility to create views which may also be editable. A relational view defines a schema of its own, just like a view in MDSD has its own metamodel. The query

mechanism in relational databases serves the function of a model-to-model transformation in MDSD. If data in the partial view is manipulated, the *view update problem* [48,49] arises, which is a central issue in relational databases which is well understood, but mainly unsolved [50]. The process of reintegrating changes on a partial view into the underlying database is called *translation*, and it has been shown that such a translation does not always exist for any kind of view update, and that it is undecidable whether a unique translation exists. The problem can be alleviated by carefully designing the views, so that every edit operation of a user in a certain view can also be reverted in that same view without losing information in the underlying database. In recent research, the view-update problem has also been investigated for tree-like structures [51], which can be applied to model transformations using graph structures.

Integrating heterogeneous metamodels and instances bears similarities to the well-known problem of schema integration of heterogeneous databases [34,52]: A semantic understanding of both domains is necessary to define the mapping of elements; hence, it cannot be fully automated. Furthermore, a global database schema is used to express data from various sources. The equivalent technique in metamodeling terms is the creation of integrated view types for heterogeneous metamodels, which also needs human interaction, and for which ModelJoin offers support through its textual DSL.

ModelJoin is closely related to other approaches in the area of multi-view modeling. For instance, Cicchetti et al. present an approach for view creation from a base metamodel with a hybrid of synthesis and projection attributes [53]. A view metamodel is created from a base metamodel (with some restrictions) and a number of transformations and tools are generated to work with view-conformant models and synchronize between the view and the base model. Restrictions on the view creation cannot be avoided to be able to offer bidirectional synchronization between instances. This approach differs to ModelJoin primarily in the goal and the resulting restrictions; the hybrid approach creates a view from a single-base metamodel with synchronization, while ModelJoin joins two or more different metamodels for a combined and read-only view. By forgoing reverse synchronization, ModelJoin initially places fewer restrictions on the view creation, with other restrictions imposed by enabling joining. The suitability of each approach depends on the application case. In broad terms, ModelJoin can be seen as a first step to investigating the reverse approach of the hybrid views of Cicchetti et al., in that we derive a "base" metamodel from two distinct views. The authors mention the possibility of investigating the reversing of their approach in the future.

Extensive work has been done in the area of model-driven software development on discovering relationships between different models, synthesizing models based on other models, and deriving information on models through querying, which are central to ModelJoin.

In general, the join operator represents a model transformation and is (as it is declarative) related to declarative transformation languages like QVT Relations [12] and ATL [13]. Yet in contrast, it is specifically tailored to easily and quickly define views on two similar metamodels. This imposes restrictions (see Sect. 5.5) but in turn no predefined target metamodel is required as is needed for general-purpose transformation languages. The general-purpose languages should be used for cases too complex for ModelJoin; here, ModelJoin can still serve as a good starting point.

Approaches for collaborative modeling make extensive use of model synthesis. For example, model synthesis is used in version control systems for models. The aim is to calculate the difference between versions of models and to merge models of different versions—both for MOF-based models [54] and for EMF, like the diff and merge algorithms of EMF Compare [55]. Another common task for model synthesis is the handling of metamodel evolution. Here, models are synthesized or "updated" to restore syntactic or semantic conformance [36,56]. These approaches differ from ModelJoin in that the treated models are related in a predefined manner, by either stemming from the same base model or conforming to different versions of the same metamodel. Furthermore, ModelJoin only provides read-only views and the change propagation back into a shared model for collaboration is not supported.

The Epsilon Merging Language [57] supports the merging of models from different metamodels. In contrast to ModelJoin, it requires the target metamodel to be created manually before merging rules can be defined.

The VirtualEMF project [11] introduces virtual models as a run-time solution for adapting one model or potentially merging models from numerous sources. While the merged models are created on the fly and on demand at run-time, in contrast to ModelJoin, the merged metamodel and a weaving model have to be defined beforehand and are not generated. We see our approach to be complementary, as both artifacts could be generated using our approach.

The EMF Facet project [7] provides a mechanism to extend an existing metamodel and conforming models with new elements, without changing the original artifacts. The approach is thus related to both the synthesis of metamodels and models. In contrast to ModelJoin, however, it does not integrate two different metamodels.

The EMF- INCQUERY framework [10] tackles the problem of interconnecting heterogeneous models without setting hard links between their metamodels. Instead, incremental queries are executed to calculate derived features of EMF models. The approach also features a caching mechanism but is not non-intrusive, since the source metamodels have to be modified by adding the derived features.

In the joining of models, the ModelJoin approach is related to the field of model composition [14,15] and aspect-oriented modeling (AOM), which also include view-based modeling techniques [58]. Some approaches in model composition specialize on the joining of models of the same arbitrary metamodel (like Kompose [16]), or a specific metamodel (like the Glue Generator Tool (GGT) [17] for the UML). ModelJoin differs in that two or more different metamodels and conforming models are joined. The Atlas Model Weaver (AMW) facilitates model composition through the creation of weaving models, which describe different links between models, and that of weaving metamodels, to define link types. AMW has been used for a variety of applications, some of which entail the generation of higher-order transformations between linked models [59]. ModelJoin can be seen as a very specific use case for model composition in the joining aspect, yet it differs in that view-based approaches require a view type, which is generated by ModelJoin in tandem.

The synthesis of the view metamodel is one important step of ModelJoin, and we recognize that many other approaches use metamodel synthesis either directly or as part of a method. Specifically, Kühne et al. [60] generate a metamodel for the pattern specification part of model transformations to be able to offer better support for the creation of transformation rules. They further propose the use of three distinct steps (relaxation, augmentation, and modification) to derive at a tailored pattern-metamodel from a generic metamodel for the input or output languages. We took a different route in our approach as our purpose is different; to arrive at a metamodel of a joint view, we need to provide the means for the user to specify which parts of two distinct metamodels are meant to join as opposed to tailoring a single metamodel. While the task of creating views over models can be tackled by a large number of approaches (like general-purpose transformation languages), we believe that ModelJoin is beneficial in providing a single DSL and deriving everything necessary from it.

Other approaches for the management of heterogeneous models use a central, fixed metamodel as a hub; bidirectional transformations have to be specified for all metamodels that are to be supported: The OSM approach [5] has been implemented in KobrA [26]. More tool-driven approaches include ModelBus [61], which is focused on the interoperability of heterogeneous modeling tools, and DuALLy [62], which uses higher-order transformations based on ATL for architectural description languages.

9 Conclusion and future work

The method for the definition of custom views presented in this paper offers an intuitive way and reduces the effort to create views on heterogeneous models. With the ModelJoin DSL, developers in model-driven software development projects can define recurring or singular information needs using a textual syntax. ModelJoin implements the concept of *flexible views*, which combine view type and view definition to provide means for a rapid creation of custom, user-specific views. The metamodel which defines the view type, the instances which represent the view, and the transformations between the source models and the view type are generated from a ModelJoin query, and also coevolve if the query is modified. Developers do not have to specify the view types and transformations manually and do not have to maintain the conformance between the M2M transformations and the metamodels in case of a change to the flexible view.

With ModelJoin and the view-based development approach, developers of model-driven software projects gain permanent access to consistent, up-to-date and complete information about the system under development, tailored to the information needs of different developer roles, such as domain experts, system architects, or software developers. Views that are defined with ModelJoin can be customized to the needs of an individual developer and can be defined by the developers in textual form using the ModelJoin DSL. We expect that ModelJoin speeds up the development process since manual efforts like the creation of metamodels and transformations are avoided. The evaluation of ModelJoin indicates that definition of flexible views with ModelJoin is more compact than the implementation with other, state-of-the-art modeling technologies.

ModelJoin supports view-centric software development processes like OSM [5] or VITRUVIUS [27], which heavily rely on the possibility to create and modify varying view types and views. The ability of ModelJoin to create views on heterogeneous models without the need of changing the involved metamodels forms the basis for non-intrusive approaches, which combine legacy metamodels to form a modular single underlying model.

To fully support view-centric development processes, a flexible view should also be configurable to be editable. The view type definition has to include rules for the manipulation of elements and the transformation back to the underlying models. We plan to extend the ModelJoin DSL by constructs that describe the editability of view types and the policies for the synchronization with the source metamodels. In a view-centric development process, the source models can only be manipulated through views and require bidirectional transformations to keep the views synchronized with the models.

The implementation prototype presented in this paper is based on the EMF technologies QVT, Xtext, and Xtend2. Future implementations of ModelJoin could be performed, e.g., via an object/relational mapping using the Eclipse CDO [63] model persistency framework, which is expected to perform better on very large models, or using a graph-oriented representation of the models with a graph query language,

e.g., the Cypher Query Language [64], which supports transactionality and editability.

Our first experiences with performance properties of the ModelJoin prototype indicate that the execution time of the ModelJoin algorithms (transformation and metamodel generation) scales linearily with the size of the query. The size of the input metamodels and models only influence the loading times of the EMF framework and the execution time of the QVT-O transformations. This is, however, not a ModelJoin-specific performance issue, but concerns all EMF-based applications and is the subject of other research in model-driven development [65]. As future work, we plan a performance evaluation of ModelJoin using a test framework that creates synthetic metamodels, instances, and matching queries, which is currently under development. To systematically explore the degrees of freedom in the input data, we are planning to conduct tests using the Software Performance Cockpit [66].

The flexible views that are created with ModelJoin can already be persisted and reused on varying instances of the input metamodels. It could also be helpful to reuse the view type of such a flexible view for new ModelJoin queries, so that the resulting views are instances of the same view type. Currently, a state-based conformance check [42] can be used to determine if an existing metamodel is an apt metamodel for a specific query. Future versions of the implementation could assist the developer in adapting a query so that the result conforms to this existing metamodel.

To increase the meaningfulness of our case study, we plan to re-evaluate the tool with a larger sample of developers.

References

1. France, R., Rumpe, B.: Does model driven engineering tame complexity? English. Softw. Syst. Model. **6.1**, 1–2 (2007). ISSN: 1619-1366. doi:10.1007/s10270-006-0041-9
2. Bendix, L., Emanuelsson, P.: Requirements for practical model merge–an industrial perspective. In: Schürr, A., Selic, B. (eds.) Model Driven Engineering Languages and Systems, vol. 5795. LNCS, pp. 167–180. Springer, Berlin (2009). ISBN: 978-3-642-04424-3. doi:10.1007/978-3-642-04425-0_13
3. Yang, Y., et al.: Phase distribution of software development effort. In: Proceedings of the Second ACMIEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '08), pp. 61–69. ACM, Kaiserslautern, Germany (2008). ISBN: 978- 1-59593-971-5. doi:10.1145/1414004.1414016
4. Goldschmidt, T., Becker, S., Burger, E.: View-based modelling–a tool oriented analysis. In: Proceedings of the Modellierung 2012, Bamberg (2012)
5. Atkinson, C., Stoll, D., Bostan, P.: Orthographic software modeling: a practical approach to view-based development. In: Maciaszek, L.A., González-Pérez, C., Jablonski, S. (Eds.) Evaluation of Novel Approaches to Software Engineering, vol. 69, Communications in Computer and Information Science, pp. 206–219. Springer, Berlin (2010). ISBN: 978-3-642-14819-4
6. Burger, E.: Flexible views for view-based model-driven development. In: Proceedings of the 18th International Doctoral Symposium on Components and Architecture (WCOP '13), pp. 25–30. ACM, Vancouver, BC, Canada (2013). ISBN: 978-1-4503-2125-9. doi:10.1145/2465498.2465501
7. EMF Facet. http://www.eclipse.org/facet/
8. EMF Model Query. http://www.eclipse.org/modeling/emf/?project=query
9. EMF Query 2. http://wiki.eclipse.org/EMF_Query2Home
10. Hegedüs, Á., et al.: Query-driven soft interconnection of EMF models. In: France, R., et al. (eds.) Model Driven Engineering Languages and Systems, Vol. 7590. Lecture Notes in Computer Science, pp. 134–150. Springer, Berlin (2012). ISBN: 978-3-642-33665-2. doi:10.1007/978-3-642-33666-9_10
11. Clasen, C., Jouault, F., Cabot, J.: VirtualEMF: a model virtualization tool. In: De Troyer, O., et al. (eds.) Advances in Conceptual Modeling. Recent Developments and New Directions, vol. 6999. LNCS, pp. 332–335. Springer, Berlin (2011). ISBN: 978-3-642-24573-2. doi:10.1007/978-3-642-24574-9_43
12. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Object Management Group (2011). http://www.omg.org/spec/QVT/1.1/
13. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Satellite Events at the MoDELS 2005 Conference, pp. 128–138. Springer, Berlin (2006). http://www.springerlink.com/index/7143g735r4j59463.pdf
14. Herrmann, C., et al.: An algebraic view on the semantics of model composition. In: Akehurst, D., Vogel, R., Paige, R. (eds.) Model Driven Architecture-Foundations and Applications, Vol. 4530. Lecture Notes in Computer Science, pp. 99–113. Springer, Berlin (2007). ISBN: 978-3-540-72900-6. doi:10.1007/978-3-540-72901-3_8
15. Bézivin, J., et al.: A canonical scheme for model composition. In: Rensink, A., Warmer, J. (eds.) Model Driven Architecture-Foundations and Applications, Vol. 4066. LNCS, pp. 346–360. Springer, Berlin (2006). ISBN: 978-3-540-35909-8. doi:10.1007/11787044_26
16. Fleurey, F., et al.: A generic approach for automatic model composition. In: Giese, H. (ed.) Models in Software Engineering, vol. 5002. LNCS, pp. 7–15. Springer, Berlin (2008). ISBN: 978-3-540-69069-6. doi:10.1007/978-3-540-69073-3_2
17. Atlas Model Weaver. http://www.eclipse.org/gmt/amw/
18. Bouzitouna, S., Gervais, M.-P., Blanc, X.: Model reuse in MDA. In: Proceedings of the International Conference on Software Engineering Research and Practice (SERP'05). Las Vegas, USA (2005)
19. Becker, S., Koziolek, H., Reussner, R.: The Palladio Component Model for model-driven performance prediction. J. Syst. Softw. **82**, 3–22 (2009). doi:10.1016/j.jss.2008.03.066
20. Finkelstein, A., et al.: Viewpoints: a framework for integrating multiple perspectives in system development. Int. J. Softw. Eng. Knowl. Eng. **2**(1), 31–57 (1992)
21. Rumbaugh, J., et al.: Object-Oriented Modeling and Design. 1. Prentice Hall, Englewood Cliffs (1991).
22. Coleman, D., et al.: Object-Oriented Development: The Fusion Method. Prentice Hall, Englewood Cliffs (1994)
23. Kruchten, P.B.: The 4+1 view model of architecture. In: Software IEEE **12.6**, 42–50 (1995). ISSN: 0740-7459. doi:10.1109/52.469759
24. Kruchten, P.: The Rational Unified Process: An Introduction, 3rd ed., 6. pr. Addison-Wesley Object Technology Series. Addison-Wesley, Upper Saddle River (2007). ISBN: 0-321-19770-4
25. OMG Unified Modeling Language (UML). Object Management Group (2011). http://www.omg.org/spec/UML/2.4.1/
26. Atkinson, C., et al.: Modeling components and component-based systems in KobrA. In: Rausch, A., et al. (eds.) The Common Component Modeling Example, vol. 5153. Lecture Notes in Com-

puter Science, pp. 54–84. Springer, Berlin (2008). doi:10.1007/978-3-540-85289-6_4
27. Kramer, M.E., Burger, E., Langhammer, M.: View-centric engineering with synchronized heterogeneous models. In: Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO '13), pp. 5:1–5:6. ACM, Montpellier, France (2013). ISBN: 978-1-4503-2070-2. doi:10.1145/2489861.2489864
28. ISO/IEC Standard for Systems and Software Engineering-Recommended Practice for Architectural Description of Software-Intensive Systems. In: ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15 (July 2007), pp. c1–c24. doi:10.1109/IEEESTD.2007.386501
29. ISO/IEC/IEEE Std 42010:2011: Systems and Software Engineering-Architecture Description. IEEE, Los Alamitos (2011)
30. Stahl, T., Völte, M.: Model-Driven Software Development. Wiley, New York (2006)
31. OMG Model Driven Architecture. http://www.omg.org/mda/
32. Eclipse Modeling Framework. Version 2.7. http://www.eclipse.org/modeling/emf/
33. Goldschmidt, T., Becker, S., Uhl, A.: Incremental updates for textual modeling of large scale models. In: Proceedings of the 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2010)–Poster Paper. IEEE (2010)
34. Reddy, M., et al.: A methodology for integration of heterogeneous databases. IEEE Trans. Knowl. Data Eng. **6.6**, 920–933 (1994). ISSN: 1041-4347. doi:10.1109/69.334882
35. Burger, E., Gruschko, B.: A change metamodel for the evolution of MOF-based metamodels. In: Engels, G., Karagiannis, D., Mayr, H.C. (eds.) Proceedings of Modellierung 2010, vol. P-161. GI-LNI. Klagenfurt, Austria (2010). http://sdqweb.ipd.kit.edu/publications/pdfs/burger2010a.pdf
36. Herrmannsdörfer, M., Vermolen, S.D., Wachsmuth, G.: An extensive catalog of operators for the coupled evolution of metamodels and models. In: Proceedings of the Third International Conference on Software Language Engineering (SLE'10), pp. 163–182. Springer, Berlin (2011). ISBN: 978-3-642-19439-9. http://www4.in.tum.de/herrmama/publications/SLE2010_herrmannsdoerfer_catalog_coupled_operators.pdf
37. Happe, L., et al.: Completion and extension techniques for enterprise software performance engineering. In: Brunetti, G., et al. (eds.) Future Business Software-Current Trends in Business Software Development. Progress in IS. Springer, New York (2014). ISBN 978-3-319-04143-8. doi:10.1007/978-3-319-04144-5
38. Reussner, R., et al.: The Palladio Component Model. Tech. rep. Karlsruhe: KIT, Fakultät für Informatik (2011). http://digbib.ubka.uni-karlsruhe.de/volltexte/1000022503
39. Atlas Transformation Language. http://www.eclipse.org/atl/
40. OMG Object Constraint Language (OCL): Object Management Group (2012). http://www.omg.org/spec/OCL/2.3.1/
41. Meta Object Facility (MOF) Core: Object Management Group (2011). http://www.omg.org/spec/MOF/2.4.1/
42. Burger, E., Toshovski, A.: Differencebased conformance checking for ecore metamodels. In: Proceedings of Modellierung 2014, Vol. 225. GI-LNI. Vienna, Austria (2014). http://sdqweb.ipd.kit.edu/publications/pdfs/burger2014a.pdf
43. Knöpfel, A., Gröne, B., Tabeling, P.: Fundamental Modeling Concepts: Effective Communication of IT Systems. Wiley, New York (2006). ISBN 978-0-470-02710-3
44. The Modeling Workflow Engine 2 (MWE2). http://www.eclipse.org/Xtext/documentation/2_0_0/118-mwe-in-depth.php
45. Burger, E., et al.: ModelJoin. A Textual Domain-Specific Language for the Combination of Heterogeneous Models. Tech. rep. 1. Karlsruhe Institute of Technology, Faculty of Informatics (2014). http://digbib.ubka.uni-karlsruhe.de/volltexte/1000037908
46. Johnson, R., et al.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995). ISBN: 9780201633610
47. Mendenhall, W., Beaver, R.J., Beaver, B.M.: Introduction to Probability and Statistics, 12th edn. Cengage Learning, Stamford (2005). ISBN 9780534418700
48. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Trans. Database Syst. **6.4**, 557–575 (1981). ISSN: 0362-5915. doi:10.1145/319628.319634
49. Codd, E.F.: The Relational Model for Database Management: Version 2. Addison-Wesley/Longman, Boston (1990). ISBN 0-201-14192-2
50. Lechtenbörger, J.: The impact of the constant complement approach towards view updating. In: Proceedings of the Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '03), pp. 49–55. ACM, New York, NY, USA (2003). ISBN: 1-58113-670-6. doi:10.1145/773153.773159
51. Nathan Foster, J., et al.: Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. SIGPLAN Not. **40.1**, 233–246 (2005). ISSN: 0362-1340. doi:10.1145/1047659.1040325
52. Sheth, A.P., Larson, J.A.: Federated database systems for managing distributed, heterogeneous, and autonomous databases. ACM Comput. Surv. **22**(3), 183–236 (1990). ISSN: 0360-0300. doi:10.1145/96602.96604
53. Cicchetti, A., Ciccozzi, F., Leveque, T.: A hybrid approach for multi-view modeling. In: Electronic Communications of the EASST Recent Advances in Multi-paradigm Modeling (MPM2011), vol. 50 (2011). ISSN: 1863-2122. http://journal.ub.tu-berlin.de/eceasst/article/view/738
54. Alanen, M., Porres, I.: Difference and union of models. In: Stevens, P., Whittle, J., Booch, G. (Eds.) Proceedings of the "UML 2003"—The Unified Modeling Language, Modeling Languages and Applications 6th International Conference, San Francisco, CA, USA (October 20–24, 2003), Vol. 2863. LNCS, pp. 2–17. Springer, Berlin/Heidelberg (2003). ISBN: 978-3-540-20243-1
55. Brun, C., Pierantonio, A.: Model differences in the eclipse modelling framework. In: UPGRADE the European J for the Informatics Professional IX.2, pp. 29–34 (2008). http://www.cepis.org/upgrade/files/2008-II-pierantonio.pdf
56. Cicchetti, A., et al.: Automating co-evolution in model-driven engineering. In: 12th International IEEE Enterprise Distributed Object Computing Conference, pp. 222–231. IEEE (2008). doi:10.1109/EDOC.2008.44. http://www.computer.org/portal/web/csdl/doi/10.1109/EDOC.2008.44
57. Kolovos, D., Paige, R., Polack, F.: Merging models with the epsilon merging language (EML). In: Nierstrasz, O., et al. (Eds.) Model Driven Engineering Languages and Systems, Vol. 4199. Lecture Notes in Computer Science, pp. 215–229. Springer, Berlin (2006). ISBN: 978-3-540-45772-5. doi:10.1007/11880240_16
58. Kienzle, J., Al Abed, W., Klein, J.: Aspect-oriented multi-view modeling. In: Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development (AOSD '09), pp. 87–98. ACM, Charlottesville, VA, USA (2009). ISBN: 978-1-60558-442-3. doi:10.1145/1509239.1509252
59. Didonet Del Fabro, M., Valduriez, P.: Towards the efficient development of model transformations using model weaving and matching transformations. Softw. Syst. Model. **8**(3), 305–324 (2008). ISSN: 1619-1366. doi:10.1007/s10270-008-0094-z. http://www.springerlink.com/index/10/s10270-008-0094-z
60. Kühne, T., et al.: Explicit transformation modeling. In: Proceedings of the 2009 International Conference on Models in Software Engineering (MODELS' 09), pp. 240–255. Springer, Denver, CO (2010). ISBN: 3-642-12260-4-642-12260-6. doi:10.1007/978-3-642-12261-3_23

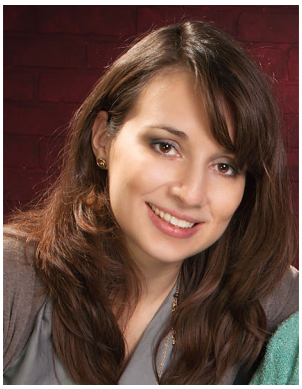
61. Hein, C., Ritter, T., Wagner, M.: Model-driven tool integration with ModelBus. In: Workshop Future Trends of Model-Driven Development (2009)
62. Malavolta, I., et al.: Providing Architectural Languages and Tools Interoperability Through Model Transformation Technologies. Tech. rep. 1, pp. 119–140 (2010). doi:10.1109/TSE.2009.51
63. Eclipse Connected Data Objects (CDO). http://wiki.eclipse.org/CDO
64. Cypher Query Language. http://docs.neo4j.org/chunked/stable/cypher-query-lang.html
65. Amstel, M., et al.: Performance in model transformations: experiments with ATL and QVT. In: Cabot, J., Visser, E. (Eds.) Theory and Practice of Model Transformations, Vol. 6707. Lecture Notes in Computer Science, pp. 198–212. Springer, Berlin, Heidelberg (2011). ISBN: 978-3-642-21731-9. doi:10.1007/978-3-642-21732-6_14
66. Westermann, D., et al.: Automated inference of goal-oriented performance prediction functions. In: Proceedings of the 27th IEEE/ACM International Conference On Automated Software Engineering (ASE 2012). Essen, Germany (2012)

Erik Burger studied Computer Science at the University of Karlsruhe. After completing his diploma thesis at SAP, Walldorf, he joined the Software Design and Quality (SDQ) group at Karlsruhe Institute of Technology in 2009, where he is currently (2014) completing his PhD thesis. His main research interests cover model-driven software development, metamodel evolution, and view-based modeling. He is currently involved in the development of the view-centric Vitruvius approach, which is based on a virtual single underlying model and the automatic generation of view types and views.

Jörg Henss studied Computer Science at the University of Karlsruhe. After completing his diploma thesis at Fraunhofer, Karlsruhe, he worked as a freelance software developer and joined the Software Design and Quality (SDQ) group at Karlsruhe Institute of Technology in 2009, where he is currently (2014) completing his PhD thesis. His main research interests cover simulation interoperability and model-driven engineering.

Martin Küster is a researcher at the Software Engineering group of FZI Research Center for Information Technology. Since 2010, he has been working with models extensively. Early work included textual syntaxes for models and models for performance prediction of real-time embedded software. More recently, he designed domain-specific languages for the model-driven development of mobile applications. The usage of models for validation and traceability of design decisions with linkage to architectural models and code is his current research focus, in which he is pursuing his Ph.D.

Steffen Kruse is a researcher at the Architecture Engineering and Interoperability group at the OFFIS—Institute for Information Technology in Oldenburg, Germany. After receiving his Diploma in Computer Science from the University of Oldenburg, he joined OFFIS in 2008 where he is currently completing his PhD thesis. His research interests include the evolution of models and model transformations and the practical visualisation of structured information.



Lucia Happe is a researcher at the Karlsruhe Institute of Technology (KIT) in the group of Software Design and Quality (SDQ). She received her Ph.D. in computer science in 2011 from the KIT as well. In 2008, she was awarded a Ph.D. scholarship from the Deutscher Akademischer Austauschdienst (DAAD). She got her diploma in computer science from the Technical University of Kosice in Slovakia. The usage of model-driven techniques in complex systems engineering, quality properties of software artifacts in model-driven development and model-based quality prediction, with focus on model-driven security, is her current research focus.