Towards dynamic views on heterogeneous models – the NeoJoin view definition language

Lars König, Tobias Stickling, Daniel Ritz and Erik Burger

Karlsruhe Institute of Technology (KIT), Kaiserstraße 12, 76131 Karlsruhe, Germany

Abstract

Dynamic views on heterogeneous models are an important part of a scalable, agile development process. Current solutions provide, as yet, only limited expressiveness for complex view (type) operations or only partial combination of view type and transformation syntaxes. In this paper, we present our idea of NeoJoin, an easy-to-use view definition language with a combined syntax for view types and complex, incremental transformations. While a prototype is still work in progress, we explain the language concepts, we give examples of model queries in our preliminary syntax, and we provide valuable insights into technical challenges, such as the generation of triple graph grammars for the transformations.

Keywords

view-based development, view definition, model query, language

1. Introduction

In model-driven software or systems development, different developer roles create and use various different models. Therefore, one important part of a scalable development process that supports variability is the definition and generation of views on models. Further, in an agile development process, the tasks of developers may change, and not all required types of views may be foreseeable. Therefore, developers should be able to create dynamic queries on models as needed.

Various view definition approaches [1] already provide editable views with extensive model-level transformation operations. However, no approach combines all features relevant for supporting agile, model-driven development processes. Therefore, we envision a language that combines:

R1 An intuitive syntax for the combined specification of view types and transformations

R2 Expressive transformation operators on the meta- and model-levels

R3 Support for the combination of information from heterogeneous meta-models

R4 Incremental and bidirectional transformations between models and views

R5 IDE support including visualizations and AI support

We propose *NeoJoin*, a view definition language supporting all the features required to facilitate the processes of an agile, model-driven development. Using known concepts and syntax elements from languages like SQL and Emfatic[1], NeoJoin has an intuitive syntax for the definition of queries and view type elements. Its easy-to-use syntax, however, should not limit NeoJoin in offering a comprehensive, but expressive set of query operators, combining the specification of view types and model-level transformations. To synchronize development across different views, NeoJoin uses the inherent bidirectionality and incrementality of triple graph grammars. To achieve this, our query operators are operationalized in the form of triple graph grammar rules in the transformation backend.

In this early-stage paper, we present the concepts and current state of our syntax on a small example from the Smart Home domain. Still, we provide insights into key aspects of the language pipeline

Joint Proceedings of the STAF 2025 Workshops: OCL, OOPSLE, LLM4SE, ICMM, AgileMDE, AI4DPS, and TTC. Koblenz, Germany, June 10-13, 2025

lars.koenig@kit.edu (L. König); burger@kit.edu (E. Burger)
0000-0002-1751-1291 (L. König); 0000-0003-2832-3349 (E. Burger)

[1]https://eclipse.dev/emfatic accessed at 2025-03-24 08:30

for generating the model-level transformations. We know that NeoJoin is yet in early development; therefore, we present plans for an evaluation of its expressiveness, conciseness, correctness, and usability, and we relate NeoJoin to other view definition approaches.

2. Background

2.1. Foundations

View-based Development Model-driven development describes the development of software or other systems, in which models are central development artifacts. An important aspect of models is abstraction, i.e., they reduce the complexity of system descriptions for developers [2]. As developers work on different kinds of models, which describe the same system, these are also called *views* [3]. There are different approaches to maintain consistency between the views on a system description, i.a., *projective views*, which are generated dynamically from a common source [4]. In case the common source is a *pragmatic single underlying model*, it is built from multiple models, which might conform to different meta-models, connected with explicit consistency specifications [4]. As we believe this to be the most general case for a view definition language, we will assume that view types [5], i.e., view meta-models, are defined on multiple meta-models.

Model Transformation In model-driven development, information is commonly transformed between different models, e.g., from domain models to source code. In addition to general purpose programming languages, there are domain-specific model transformation languages. Model transformations can be classified according to a number of features. For view generation, we consider model transformations that are exogenous, asymmetric, bidirectional, and incremental. *Exogenous* transformations are defined between different meta-models [6] and in case of *asymmetric* transformations, models conforming to one meta-model, the view type, can be derived completely from models conforming to the other meta-models [7]. Further, *bidirectional* transformations can be executed in forward and backward direction between the models [8], which is required for views to be editable. Lastly, we use the term *incremental* for transformations that are both source incremental, i.e., only executed for change parts of the source model, and target incremental, i.e., update the target model instead of re-generating it [8]. Model transformation languages are often divided in imperative and declarative languages or, more general for model transformation approaches, described as operational, relational, or graph-based [8], with NeoJoin classified as declarative and relational.

Triple Graph Grammars Triple graph grammars (TGGs) [9] are a formalism for specifying mutually applicable graph construction rules for two models and a connecting correspondence model. Their strength comes from the possibility to operationalize the TGG rules as incremental transformations for, i.a., forward and backward transformation between the three models. As such, they constitute a declarative model transformation language that fulfills, without explicit support for asymmetric transformations, our requirements for the operation of model-view transformations. TGG rules are specified, usually using a graphical syntax, by providing a context (black) of nodes and edges that are required for the rule to be applied, and additional nodes and edges (green) that define the construction step. A model can thus be changed by applying a construction step, with a matching context, which causes the other model and the correspondence model between them to be changed alongside. Rule applications can be further restricted by defining application conditions that must hold in the graph for the rule to be applicable [10]. Transformations operate similar to the parsing of textual grammars by finding a valid sequence of TGG rule applications for a model or changes to a model. To keep attributes consistent between a source and a target model, attribute conditions can be used to specify their relation [11].

Approach	Multiple MMs	QL Operations	VTL Operations	Bidirectional	Incremental
EMF Views [12]	✓	✓	✓	✓	✗
Kitalpha [13]	✓	✓	✓	✓	✗
ModelJoin [14]	✓	✓	✓	✓	✗
OpenFlexo [15]	✓	✓	✓	✓	✓
VIATRA Viewers [16]	✓	✓	✗	✓	✓

Table 1
Feature matrix with selected model view approaches from the survey of Bruneliere et al. [1].

2.2. Related Work

As stated in Section 1, various approaches for defining model views have been proposed, underlining the importance of the matter. We do, however, believe that a language that meets our requirements fully, does not yet exist. This is supported by the survey of Bruneliere et al. [1], from which we chose and aggregated a subset of their features based on the requirements **R2**–**R4** we presented in Section 1. We selected approaches when at most one feature was not supported by the approach. The resulting comparison is shown as a feature matrix in Table 1. The exact mapping of our features, i.e., the column headers in Table 1, to the feature names from [1], as well as to the requirements presented in Section 1, is given in Table 2.

We included the feature *Multiple MMs*, corresponding to **R3**, as we believe this is a crucial case for software / systems engineering where various kinds of models are used in the development process. The features *QL Operations* (Query Language Operations) and *VTL Operations* (View Type Language Operations) refer to a set of transformation operations we deem relevant, representing **R2**. Note that we propose a combined syntax for query and view type definition. To be able to read and edit views in a multi-developer scenario, we need *bidirectional* and *incremental* transformations between models and views. We have included these features as well, addressing **R4**.

The feature matrix in Table 1 shows that only one of the considered approaches has all desired features. Most approaches lack an incremental and bidirectional transformation between the models and the view, such as EMF Views [12], Kitalpha [13], and ModelJoin [14]. Although VIATRA Viewers [16] offers incremental, bidirectional transformations, the expressiveness of their view type definition language is inherently limited to pattern-based queries and does not support the modification or filtering of meta-classes. The Viewpoint Modeler of the model federation approach OpenFlexo [15] is most closely related to NeoJoin. In contrast to OpenFlexo, which expects explicit view type definitions (cf. [1]), we provide a combined syntax based on relational operators for the definition of the view type and the model-level transformations. We believe this to be more intuitive, which we defined as requirement **R1**, especially for on-the-fly generation of view types for new or changing development tasks.

3. NeoJoin

Here we introduce NeoJoin. To give an intuition of the capabilities of the language, we first present the concepts and current state of the syntax on a small example in Section 3.1. Next, we describe the language processing pipeline, including a detailed description of some key aspects for generating the incremental model-level transformations in Section 3.2. As NeoJoin is in an early stage of development, the shown syntax and implementation details might change in the future.

3.1. Language Concepts and Syntax

In NeoJoin, a view definition comprises multiple meta-model imports, a name for the resulting view type, and a number of queries (see Figure 1). Each query needs exactly one source class and possibly many joined classes. Filter conditions can be added as well as optionally one aggregation statement with a grouping expression. A query always contains a projection expression, which itself can contain

Our Feature	Features from [1]	Requirements
Multiple MMs	Type Structure – Metamodel Arity – Multiple Metamodels	**R3**
QL Operations	Design-Time Feature – Query Language – QL Operations – Join & Aggregate	**R2**, **R3**
VTL Operations	Design-Time Feature – ViewType Language – VTL Operations – Modify & Filter	**R2**
Bidirectional	Runtime Feature – Runtime Consistency – Direction – Model to View & View to Model	**R4**
Incremental	Runtime Feature – Runtime Consistency – Recomputation – Incremental	**R4**

Table 2
Mapping of our features in Table 1 to the features from the survey of Bruneliere et al. [1] and the requirements presented in Section 1.

a new name for the target class and possibly many feature definitions. Feature definitions define the attributes and references of the target class, both on the meta level and on the model level. Features can either be selected or renamed, in which case they preserve additional attributes, such as multiplicity, or calculated from a given expression.

Now we show the query syntax of NeoJoin on a small example from the Smart Home domain. In the example, we have the two source meta-models *IoT* and *Automation*, shown in Figure 2. These meta-models partially describe the same concepts with `Device` (*IoT*) and `Server` (*Automation*). Based on that overlap, we define queries which combine the two classes. In this way, we achieve an overview of the functions provided as well as of the surrounding devices targetable by the functions. For example, in our smart home environment, a device may be blinds controlled automatically by a home server via short-range Bluetooth connection. The examples shown below use our preliminary syntax, to be updated as our development of the language progresses.

```
from automation.Function
create Function {
    name
}
```

The first query is an example of simply selecting a class from one of the source meta-models to be included in the view type. The query body, enclosed with braces, lists the attributes from the source meta-class to be included in the target meta-class. This query specifies the bidirectional transformation between the source models and the view while also, at the same time, defining the target class in the view type. Therefore, creating a function in a source model would also lead to the creation of a function in the view.

```
from iot.Device device
join automation.Server server
where name == id
create Device {
    x = posX
    y = posY
    name = device.name
    functions
}
```

The second query is more complex. Here, the classes `Device` and `Server` (from the source meta-models `iot` and `automation`, respectively) are joined together to create the class `Device` in the view type. In general, an arbitrary number of classes from the same or different meta-models can be joined together. In this example, devices and servers are joined based on the equivalence of their attributes `name` and `id`, each of which represents the same concept. As a first step, we plan to consider inner joins only. The join condition is an expression restricting the result of a join, for which we plan to use the Xbase expression language that is part of the Xtext framework. The resulting meta-class `Device` in the

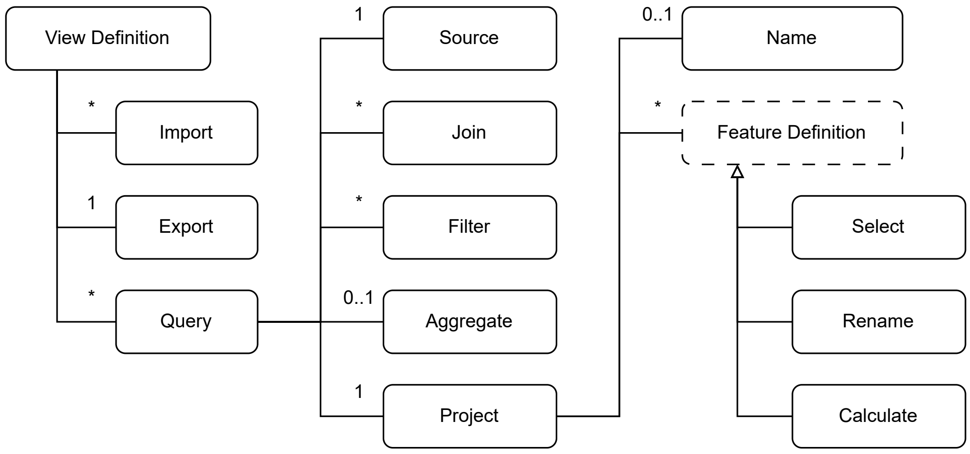

Figure 1: Overview of the language concepts and the structure of view definitions in NeoJoin. Rectangles represent syntax constructs, rectangles with dashed lines represent abstract syntax constructs for which inheriting concrete syntax constructs can be used.

view type contains the attributes `posX` and `posY` from the meta-class `Device` in the source meta-model `iot`, but in the view-type `Device` they are renamed to `x` and `y`. In addition, the attribute `name` is included, which equals the attributes `name` and `id` from the join condition. The attributes from the join condition are already constrained to be equal; therefore, it suffices to assign the new attribute `name` to one of the source attributes. Finally, the reference `functions` from the meta-class `Server` is included. In the view type, `functions` will reference the meta-class `Function` created in the view type.

```
from iot.Link
create Link {
    range
    devices
    distance := distanceBetween(devices[0].posX, devices[0].posY, devices[1].posX, devices[1].
        posY)
}
```

This third query not only includes an attribute and a reference from the source meta-class but it also shows how custom functions can be used to perform arbitrary calculations on the attributes of the source models. Not every function will have an inverse. For such attributes, then, the resulting model transformation can, in general, be unidirectional only; in other words, changes in the view cannot be propagated back to the source models by such a transformation. To indicate this, the attribute should be marked as read-only in the view type.

```
from automation.Server server
where !server.functions.isEmpty()
group by _
create ServerSummary {
    avgNoFunctions := server.map([functions.size()]).average()
}
```

In the fourth and last query, instead of selecting all model classes conforming to a meta-class, we define an aggregation over all instance of the meta-class `Server`. The result, on the meta-level, is a meta-class with the name `ServerSummary`, while on the model level, a single class will be created in the view that corresponds to all instances of `Server` in the source model. If, instead, an expression is used after `group by`, then the aggregation could also result in multiple view classes. The value of aggregated attributes can be calculated using an Xbase expression; in this query by taking the average of the number of functions in each `Server` class for the calculation of the attribute `avgNoFunctions`.

3.2. Technical Concept

In this section, we give an overview of the language processing pipeline of NeoJoin. The inputs are defined by the developer using the textual syntax of NeoJoin; specifically, these inputs will be the

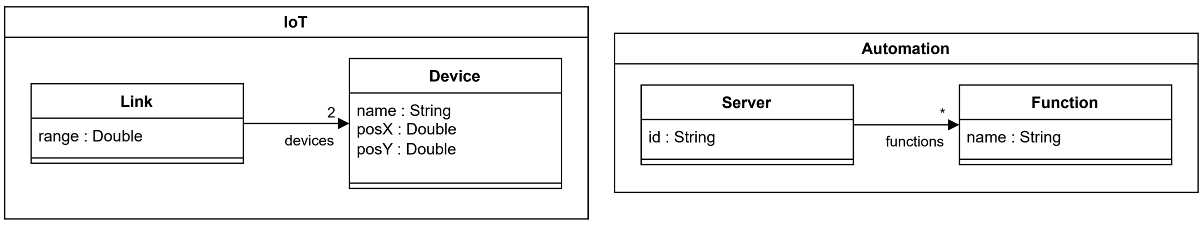

Figure 2: Source meta-models *IoT* and *Automation*.

queries (see example in Section 3.1) and the source meta-models on which the queries are defined. The pipeline, shown in Figure 3, starts by parsing the syntax to an abstract syntax tree (AST). The AST is further processed to create the query model and then is used to derive the view type as well as the source-target mapping in the language frontend. As an intermediate step, the *operator lowering* takes place. During this step, the declarative query model is transformed into an operation-oriented sequence of operator applications. The operator applications are then used in the final rule generation step to generate the triple graph grammar (TGG) rules as well as the correspondence meta-model (TGG schema). Using available TGG engines, such as eMoflon [17], these outputs can be used to bidirectionally and incrementally transform between the source models and the view.

Below we focus on critical steps of the language processing pipeline, including the challenges encountered to date as well as and the solutions we expect NeoJoin will eventually provide.

3.2.1. Language Frontend

The language frontend is the first part of the processing pipeline. It is responsible for handling the inputs from the developers. Central to the frontend is the definition of the grammar for the concrete syntax of NeoJoin, which we will define using the Xtext framework[2]. The Xtext framework allows us to automatically generate the Ecore meta-model for the abstract syntax tree (AST) as well as basic IDE support (e.g., syntax highlighting and auto-completion through a generated language server[3]) for VS Code[4], for example. The AST from the Xtext parser will be further processed in the *post processing* step, which may include normalization of the query structure, type inference, etc. The result will be a

[2]https://eclipse.dev/Xtext/ accessed at 2025-03-28 13:30
[3]https://microsoft.github.io/language-server-protocol/ accessed at 2025-03-28 13:30
[4]https://code.visualstudio.com/ accessed at 2025-03-28 13:30

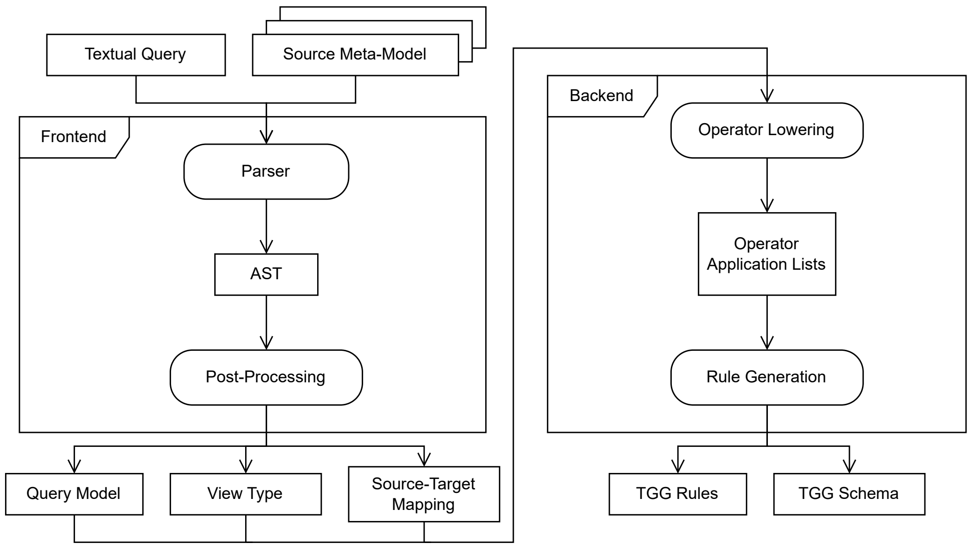

Figure 3: Overview of the language processing pipeline starting with a *textual query* and the *source meta-models* in the top left.

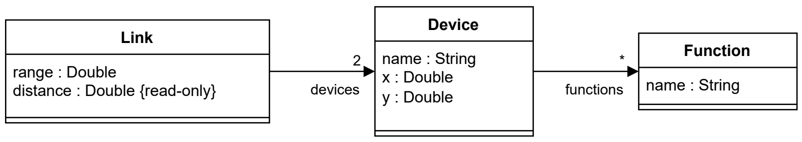

Figure 4: View type combining information from the source-meta-models *IoT* and *Automation*, presented in Figure 2.

fully-resolved query model for the next steps of the language pipeline.

In addition to the generation of the query model, the language frontend is concerned with using the source meta-model and the queries to infer the view type, i.e., the view meta-model. While this inference step is also a model-to-model transformation, it is unidirectional and can be done state-based, i.e., at query compile time. We therefore plan to implement the inference step using a general purpose programming language. For the example queries shown in Section 3.1, the resulting view type is shown in Figure 4. In addition to creating the view type, the inference step also creates traceability information between the view type and the source meta-models. Building on this traceability information, the view type inference decides which attributes can be transformed bidirectionally between the source models and the view. Attributes which cannot be transformed back to the source models (e.g., because calculated by an aggregation function without an inverse) are marked as *read-only*.

3.2.2. Language Backend

The language backend is concerned with transforming the abstract representation of the queries into executable TGG rules. To that end, we first apply, as intermediate step, the *operator lowering* step. *Operator lowering* transforms from the query model based on the abstract syntax to the underlying transformation operators of the language. The resulting operators closely resemble the language concepts introduced in Figure 1. However, the difference is that these operators may contain transformation-specific information. We call this process *lowering*, to call to mind the notion used in compiler construction. Next, in the *rule generation* step, we apply the list of operators to generate the TGG rules. The result is the definition of an incremental, bidirectional transformation in a graph-based TGG engine.

The operator lowering step generates, for each query, ordered lists of operator applications. These lists represent the execution of the query. The operators are executed in that order, and each execution takes into account the result of the previous execution of the operators in the same list.

```
Source(Device)
Join(Server, (device, server) -> device.name == server.id)
Project(Device.name, (device, server) -> device.name)
```

Listing 1: List of operator applications generated from the join between the classes `Device` and `Server` introduced in Section 3.1.

Listing 1 shows a list of operator instances representing the same query as the example introducing the `join` syntax in Section 3.1. First, the meta-class *Device* from the source meta-model *IoT* is selected. Then, the meta-class *Server* from the other source meta-model *Automation* is joined using a condition on attribute of both included meta-classes. Finally, the attribute *name* is created in the meta-class *Device* in the view type. The definition of an attribute projection contains the expression for calculating its value as part of the transformation between models and views, in this case resulting in the name of the device from the *IoT* source model.

A special case is the transformation of reference projections in the queries, which are expressed as Xbase expressions in the concrete syntax of NeoJoin. Since this has a direct impact on the structure of the generated TGG rules, NeoJoin will, in future, use separate operators for different types of operations on the reference structure. An additional step will be employed to derive these from the Xbase expressions.

The resulting operator application lists are then used in the *rule generation* step to generate TGG rules. A single list of operator applications can always be transformed to a single TGG rule. Since

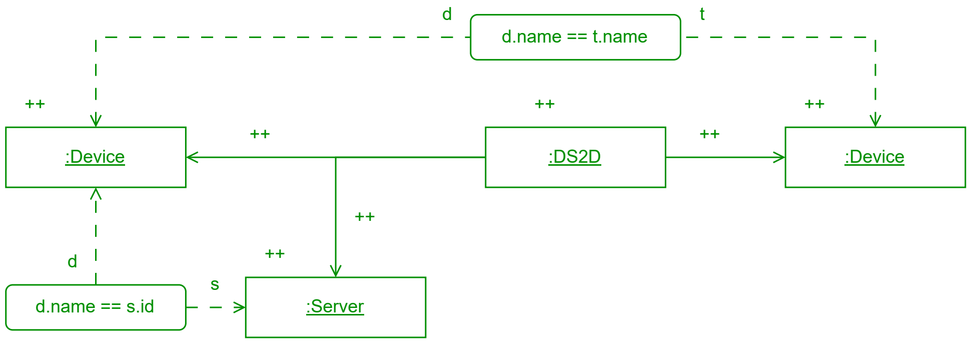

Figure 5: Example of a TGG rule for the join between the classes `Device` and `Server` introduced in Section 3.1. The rectangles represent nodes, while rounded rectangles with dashed arrows represent attribute conditions (top) and application conditions (bottom) on the referenced classes.

the correspondence graph in TGGs is a typed graph, the rule generation step is also concerned with deriving the correspondence meta-model (called TGG schema) from the generated rules. The model-level transformations are then performed by executing the resulting rules in a TGG engine, which takes care of reevaluating the changed model/view parts to enable an incremental transformation of changes.

Figure 5 shows an example of a TGG rule resulting from the join query shown, in textual syntax, in Section 3.1 and, as operator list, in Listing 1. Selecting the `Device` class in line 1 results in the green *Device* nodes being added in the source graph (left) and target graph (right), connected with the green node in the correspondence graph (center). When executing this rule as forward/backward transformations in a TGG engine, this means that whenever a device is created in the source model, a corresponding device is created in the view (forward) and vice versa (backward). The `Join` operator, parameterized with the class `Server`, then adds in the source graph both the green *Server* node and the application condition between the *Device* and *Server* nodes. Finally, the `Project` operator adds the attribute condition between the *Device* nodes in both source and target graphs, representing the bidirectional attribute calculation.

4. Planned Evaluation

We plan the evaluation of NeoJoin in alignment with our requirements for a view definition language, stated in Section 1 and detailed in Section 2.2. To evaluate the *expressiveness* and *conciseness* of our syntax, as well as the *correctness* of the generated transformations, corresponding to **R2**, **R3**, and **R4**, we plan on re-building cases from model view definition literature, such as the smart grid study from Burger et al. [18]. We also plan to create new cases from the domains of software and cyber-physical systems (CPS) engineering to demonstrate and evaluate the features of NeoJoin. In addition, artificially generated models will be used to validate correctness and *scalability* of the generated transformations.

To evaluate the *usability* of NeoJoin, we plan on conducting user studies with model transformation and domain experts. For the first study, the **participants** will come from academia, but additional studies with industry partners should be conducted to assess the applicability in real-world contexts. To lower the effort and encourage participation, we plan on conducting the study in an uncontrolled **environment**, using the hardware of the participants in their usual work setting. The **process** of the user study will include:

1. a pre study questionnaire for demographic data, professional background, level of proficiency, previous experience, etc.
2. working on multiple tasks with varying complexity and answering standardized questions for quantitative assessment
3. a post study questionnaire for a qualitative assessment of NeoJoin, including open questions for possible improvements

The tasks given to the participants in step 2 will be designed to reflect the core functionality of NeoJoin, ranging from simple queries to more complex view definitions. We will ask the participants to define views for different scenarios, as well as to interpret pre-defined queries. If possible, we will reuse parts of the available cases from our evaluation on expressiveness, conciseness, and correctness. To collect quantitative data during the study, we plan to use the standardized **measures** task completion rate (TCR) [19], system usability scale (SUS) [20], and NASA's Raw Task Load Index (RTLX) [21], including the associated questionnaires.

5. Conclusion

We presented NeoJoin, a view definition language with an easy-to-use declarative syntax and a bidirectional, incremental transformation backend that leverages triple graph grammars. NeoJoin is still work in progress; therefore, we used just a small example to present the language concepts and the preliminary syntax. On the other hand, NeoJoin is in advanced enough stages for us to explore intriguing facets of the language processing pipeline and also key steps for generating the model-view transformations.

The capabilities of NeoJoin will likely enable more agile model-driven development processes as well as facilitate closer collaboration between developers across different representations, variants, and tools. Especially NeoJoin's dynamic views, created by developers on-demand, promise to allow handling the growing size, complexity, and variability of models. In this age of AI generated models, we consider this to be one of NeoJoin's strategic capabilities.

Our future plans include first and foremost to solve the conceptual and technical challenges of NeoJoin and our presented prototype. To give further details of our implementation, we provide a replication package [22] with its current state. We will also evaluate NeoJoin with cases from software and systems engineering, as well as with the user study outlined in Section 4. Building on our research, we plan to investigate the application of views with bidirectional, incremental model-view transformations for cross-domain model consistency.

Acknowledgments

This work was supported by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF) and funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1608 – 501798263. Thanks to our textician.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] H. Bruneliere, E. Burger, J. Cabot, M. Wimmer, A Feature-Based Survey of Model View Approaches, Software & Systems Modeling 18 (2019) 1931–1952. doi:10.1007/s10270-017-0622-9.
[2] D. C. Schmidt, Guest Editor's Introduction: Model-Driven Engineering, Computer Society 39 (2006) 25–31. doi:10.1109/MC.2006.58.
[3] ISO, 42010 — Software, Systems and Enterprise — Architecture Description, 2022.
[4] C. Atkinson, C. Tunjic, T. Moller, Fundamental Realization Strategies for Multi-view Specification Environments, in: 19th International Enterprise Distributed Object Computing Conference, IEEE, Adelaide, Australia, 2015, pp. 40–49. doi:10.1109/EDOC.2015.17.
[5] T. Goldschmidt, S. Becker, E. Burger, Towards a Tool-Oriented Taxonomy of View-Based Modelling, in: E. J. Sinz, A. Schürr (Eds.), Proceedings of the Modellierung 2012, volume P-201 of *Lecture*

Notes in Informatics (LNI), GI, Bonn, Germany, 2012, pp. 59–74. URL: https://dl.gi.de/handle/20.500.12116/18148.

[6] T. Mens, P. Van Gorp, A Taxonomy of Model Transformation, Electronic Notes in Theoretical Computer Science 152 (2006) 125–142. doi:10.1016/j.entcs.2005.10.021.

[7] M. Hofmann, B. Pierce, D. Wagner, Symmetric Lenses, in: 11th Symposium on Principles of Programming Languages (POPL), volume 46, ACM, Austin, Texas, US, 2011, pp. 371–384. doi:10.1145/1925844.1926428.

[8] K. Czarnecki, S. Helsen, Feature-Based Survey of Model Transformation Approaches, IBM Systems Journal 45 (2006) 621–645. doi:10.1147/sj.453.0621.

[9] A. Schürr, Specification of Graph Translators with Triple Graph Grammars, in: E. W. Mayr, G. Schmidt, G. Tinhofer (Eds.), Graph-Theoretic Concepts in Computer Science, Springer, Berlin, Heidelberg, 1994, pp. 151–163. doi:10.1007/3-540-59071-4_45.

[10] N. Weidmann, R. Oppermann, P. Robrecht, A Feature-Based Classification of Triple Graph Grammar Variants, in: Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, ACM, New York, NY, USA, 2019, pp. 1–14. doi:10.1145/3357766.3359529.

[11] L. Lambers, S. Hildebrandt, H. Giese, F. Orejas, Attribute Handling for Bidirectional Model Transformations: The Triple Graph Grammar Case, in: Bidirectional Transformations 2012, volume 49 of *Electronic Communications of the EASST*, 2012. doi:10.14279/tuj.eceasst.49.706.

[12] H. Bruneliere, J. G. Perez, M. Wimmer, J. Cabot, EMF Views: A View Mechanism for Integrating Heterogeneous Models, in: P. Johannesson, M. L. Lee, S. W. Liddle, A. L. Opdahl, Ó. Pastor López (Eds.), Conceptual Modeling, Springer, Cham, 2015, pp. 317–325. doi:10.1007/978-3-319-25264-3_23.

[13] B. Langlois, D. Exertier, B. Zendagui, Development of Modelling Frameworks and Viewpoints with Kitalpha, in: Proceedings of the 14th Workshop on Domain-Specific Modeling, DSM '14, ACM, New York, NY, USA, 2014, pp. 19–22. doi:10.1145/2688447.2688451.

[14] E. Burger, J. Henss, M. Küster, S. Kruse, L. Happe, View-Based Model-Driven Software Development with ModelJoin, Software & Systems Modeling 15 (2016) 473–496. doi:10.1007/s10270-014-0413-5.

[15] F. R. Golra, A. Beugnard, F. Dagnat, S. Guerin, C. Guychard, Addressing Modularity for Heterogeneous Multi-Model Systems Using Model Federation, in: Companion Proceedings of the 15th International Conference on Modularity, MODULARITY Companion 2016, ACM, New York, NY, USA, 2016, pp. 206–211. doi:10.1145/2892664.2892701.

[16] C. Debreceni, Á. Horváth, Á. Hegedüs, Z. Ujhelyi, I. Ráth, D. Varró, Query-Driven Incremental Synchronization of View Models, in: Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling, VAO '14, ACM, New York, NY, USA, 2014, pp. 31–38. doi:10.1145/2631675.2631677.

[17] A. Anjorin, M. Lauder, S. Patzina, A. Schürr, eMoflon: Leveraging EMF and Professional CASE Tools, in: Tagungsband Der INFORMATIK 2011, volume 192, Gesellschaft für Informatik (GI), Berlin, 2011, p. 281.

[18] E. Burger, V. Mittelbach, A. Koziolek, View-based and Model-driven Outage Management for the Smart Grid, in: 11th International Workshop on Models@run.Time (MRT), volume 1742 of *CEUR Workshop Proceedings*, RWTH Aachen, 2016, pp. 1–8.

[19] ISO, 9241-11:2018 — Ergonomics of Human-System Interaction — Usability: Definitions and Concepts, 2018.

[20] J. Brooke, SUS: A 'Quick and Dirty' Usability Scale, in: Usability Evaluation In Industry, volume 189, CRC Press, 1996, pp. 4–7.

[21] S. G. Hart, L. E. Staveland, Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research, in: P. A. Hancock, N. Meshkati (Eds.), Advances in Psychology, volume 52 of *Human Mental Workload*, North-Holland, 1988, pp. 139–183. doi:10.1016/S0166-4115(08)62386-9.

[22] L. König, T. Stickling, D. Ritz, E. Burger, Replication Package for "Towards Dynamic Views on Heterogeneous Models with NeoJoin", 2025. doi:10.5281/zenodo.15195389.