JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Language Design of the NeoJoin View Definition Language

Lars König, Tobias Stickling, Alexander Kocher, Hüseyin Kemâl Çakmak, Erik Burger, Veit Hagenmeyer, Anne Koziolek, and Ralf Reussner
Karlsruhe Institute of Technology (KIT), Germany

ABSTRACT In the view-based development of cyber-physical systems, developers with different roles and tasks use specific views to access models of the system. Using projective approaches, these views are derived from consistent, underlying models. The views and transformations from and to the underlying models are defined using domain-specific languages. The problem is that existing view definition languages lack either an easy-to-use syntax or sufficiently expressive transformation operators or incremental model-view transformations. In this paper, we propose a syntax for the view definition language NeoJoin based on the concepts of model queries and meta-model descriptions. We evaluate the usability of the proposed syntax with a user study and apply it to an existing case in the domain of smart electricity grids. With the proposed syntax, we aim to provide an easy-to-use, yet universally applicable language for the definition of views in cyber-physical systems development.

KEYWORDS view-based development, model transformation, language design.

1. Introduction

In model-driven engineering, view-based approaches enable developers to use the most efficient representation of relevant aspects of a system (Bruneliere et al. 2019; Cicchetti et al. 2019). Various approaches have been proposed, providing domain-specific view definition languages to define the views available to developers. Unfortunately, to date, no approach meets the requirements of cyber-physical systems engineering (König, Stickling, & Burger 2025), which are:

- incremental, bidirectional transformations between models and views
- expressive transformation operators
- combination of information from heterogeneous meta-models
- usage both before and during development

JOT reference format:
Lars König, Tobias Stickling, Alexander Kocher, Hüseyin Kemâl Çakmak, Erik Burger, Veit Hagenmeyer, Anne Koziolek, and Ralf Reussner. *Language Design of the NeoJoin: View Definition Language.* Journal of Object Technology. Vol. 25, No. 3, 2026.
http://dx.doi.org/10.5381/jot.2026.25.3.a1

We previously presented the concept of the new view definition language NeoJoin (König, Stickling, & Burger 2025), in an effort to meet all of these requirements. In this paper, we propose a novel, textual syntax for NeoJoin. Our textual syntax integrates the view type definition with the definition of the model-view transformation in a query-style syntax based on known concepts from database engineering and object-oriented modelling. The NeoJoin implementation provides support for different transformation backends and so enables the use of incremental, bidirectional transformations. Our sole purpose here, in this paper, is to describe the textual syntax; therefore, we refer to (König, Ritz, & Burger 2025) for initial results towards an incremental, bidirectional transformation backend based on triple graph grammars.

To evaluate our contributions, we show that NeoJoin can be used in CPS engineering by applying it to a view-based outage management system for smart electricity grids (Burger, Mittelbach, & Koziolek 2016). We further show that NeoJoin can be effectively and efficiently used by developers without intensive training in NeoJoin or model-driven technologies.

In the end, NeoJoin should help developers to manage the complexity of their collaborative development processes, which remains an open challenge for academia and for industry (Feichtinger et al. 2022). This is especially important in the engi-

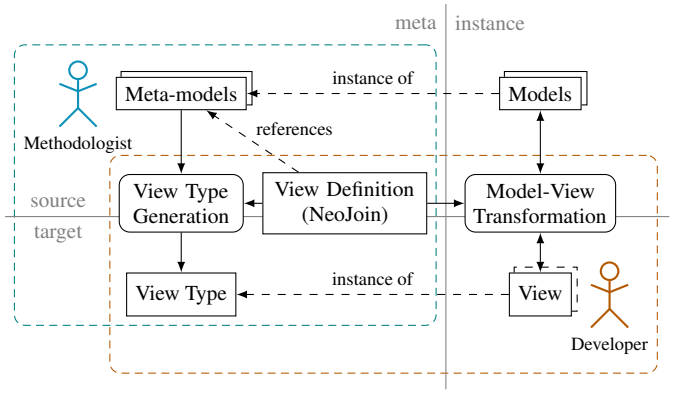

Figure 1 Overview of the view-based development process using NeoJoin. □ = artifact, ⬭ = process step, → = data flow, ⬚ = interaction scope of role with artifacts and process steps

neering of cyber-physical systems where, in addition, a high degree of variability and dependability is needed.

2. Preliminaries

Here we explain the view-based development process supported by NeoJoin (subsection 2.1), and we introduce our running example and make our problem statement (subsection 2.2).

2.1. View-based Development Process

Developers use views specific to their role and task in order to work on the models of the system under development (ISO 2011). We refer to the meta-model of a view as its *view type* (Goldschmidt et al. 2012). With *projective* approaches to view-based development, a view is defined by its view type as well as by the model-view transformation which generates the view from the models and updates the models upon changes in a view (Atkinson et al. 2015).

Figure 1 overviews the artifacts and process steps involved in the view-based development process supported by NeoJoin. We distinguish artifacts along two dimensions. Along the one dimension, we distinguish between derived artifacts that the developers work with (called *target*) and underlying artifacts that these are derived from (*source*). Along the other dimension, we distinguish between artifacts at the *meta level* and artifacts at the *instance level*. We assume a *pragmatic* approach for assembling the source artifacts (Atkinson et al. 2015), i.e., there can be multiple source meta-models and instantiating models.

NeoJoin supports development processes with different roles. In the development process of the orthographic software modelling (OSM) approach (Atkinson et al. 2010), the role of the *methodologist* is responsible for creating the meta-models, as well as the view types. The role of the *developer* is responsible for developing the system models using the views. (Burger 2013) also defines *flexible views*, which can be defined by the developer. Both of these processes are supported by NeoJoin; moreover, NeoJoin offers a flexible, combined syntax for the definition of the view type and the model-view transformation.

We refer to the combined definition of the view type and the model-view transformation as *view definition*. The view definition only references elements from the source meta-models and is therefore independent of the model instances. Due to that, it can be used to generate the view type from a view definition before model instances are created, supporting the methodologist in the OSM development process (Atkinson et al. 2010). Together with the source instance models, the view definition can then be used by the developer to create one or multiple views using one of the transformation backends supported by NeoJoin.

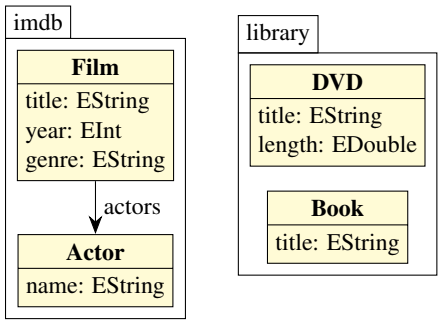

Figure 2 Class diagram of the meta-models `imdb` and `library` used as running example in this paper.

2.2. Running Example and Problem Statement

We use a toy example to illustrate the syntax for view definitions on multiple source meta-models in NeoJoin. The two meta-models `imdb` and `library`, shown in Figure 2, address concepts around media items. The `imdb` meta-model, referring to the *Internet Movie Database*, contains the two meta-classes `Film` and `Actor`, where a film has a number of actors that take part in it, represented by the reference `actors`. The `library` meta-model describes items in a library, such as DVDs and books, represented by the two meta-classes `DVD` and `Book`. In the remainder of the paper, we refer to the meta-classes of the two meta-models as well as to their attributes and references.

The overall problem to be solved by view-based approaches, is to provide developers with view types and instantiating views that contain part of the information in the models conforming to the source meta-models. Views are especially useful for analyzing large heterogeneous models and deriving multiple role- or task-specific views. In our example, a view type could contain the most recent films of an actor, stating title, genre, and length. As we show in the source meta-models of our example in Figure 2, creating views of this view type would require combining information from multiple models.

Note, the following examples for NeoJoin are not intended to construct a single view type but instead to show different syntax elements. We therefore do not present a meta-model or variants of a meta-model for the results of the presented queries.

3. Language Design

A NeoJoin view definition produces a single view type, as well as the associated model-view transformation. For that, a view definition consists of a header and a number of queries, each of which represents a meta-class in the view type. A query

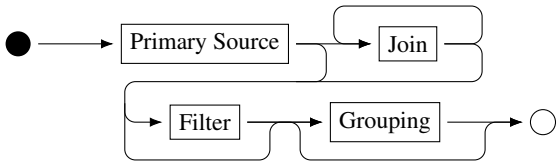

Figure 3 Syntax for the source section of a query.

comprises two parts: first, an optional source section, which selects meta-classes and instances from the source meta-models and models, and second, a mandatory target section, which defines the meta-class in the view type, including its features (i.e., attributes and references).

For concrete, i.e., non-abstract, target meta-classes, the target section also defines the model-view transformation between the source models and a view. As the queries are specified entirely on the source meta-models (see subsection 2.1) they cannot reference instances in the view. References in the view are therefore specified as references to source instances. After all instances have been transformed to a view, the references are then created to reference the corresponding view instances. If there are multiple view instances of a single source instance, their type is used to distinguish them (see subsection 3.4).

In the following subsections, we describe the syntax elements of the source and target section in NeoJoin. Our focus is syntax, because we want to present our language design choices. For that reason, we explain semantics only where required for understanding the syntax and our design decisions. We intentionally chose to use syntactical elements from languages such as SQL, Emfatic[1], and Xbase[2].

3.1. Setting Up the View Type

```
export movies to "http://example.org/movies"
import "http://example.org/imdb"
import "http://example.org/library" as lib
```

Listing 1 View definition header

The header of a view type definition contains the name and URI of the created view type, and import statements for the source meta-models. In Listing 1, the created view type is named `movies` and assigned the URI `http://example.org/movies`. There can be any number of import statements. An import statement requires the URI of the source meta-model, and can optionally specify an alias for the imported meta-model. In the example, the meta-models are available via the aliases `imdb` and `lib`.

3.2. Selecting From the Sources

The source section of a NeoJoin query is responsible for selecting meta-classes and instances from one or multiple source meta-models. In general, it consists of the four parts shown in the syntax diagram in Figure 3: a primary source meta-class, optionally one or multiple joins, as well as an optional filter and grouping part. The source section itself is optional and can be omitted when creating a single instance of a new meta-class in the view type.

```
from Film film
join DVD dvd
  on film.title == dvd.title
  with film using title
where film.year > 2020
```

Listing 2 Source/join/where statement

The primary source meta-class of a query selects a single meta-class from one of the source meta-models as input for the view type, as well as for the model-level transformation. In the example of Listing 2, the source meta-class `Film` from the source meta-model `imdb` is selected, and the alias `film` is registered to reference an instance of `Film` in the remainder of the query. If there is only the primary source meta-class, i.e., the query contains no joins, and no alias is explicitly set, the implicit alias `it` is generated. If the name `Film` was not unique across the source meta-models, we could use its qualified name `imdb.Film`, including the name or alias of the source meta-model (see subsection 3.1).

To combine multiple source meta-classes, as well as instances of multiple source meta-classes when creating instances of a view tape in a view, developers can use `join` for inner joins, or `left join` for left outer joins. In addition to the primary source meta-class of a query, a join selects another source meta-class by its name, and assigns an alias to reference an instance of the selected meta-class in the remainder of the query. In Listing 2, the source meta-class `DVD` from the meta-model `library` is selected, and the alias `dvd` is registered to reference an instance of `DVD`meta.

Join conditions are used to specify which instances of the source meta-classes are combined to create an instance of the target meta-class in the view. There are two different types of join conditions: expression join conditions introduced with the keywords `on` and feature join conditions using the keywords `with` and `using`. Expression join conditions state an expression, such as `film.title == dvd.title` in the example in Listing 2, which instances must fulfill to be included in the resulting view. Feature join conditions are a shorthand to join instances on a common attribute, such as `title`. If there is only a single other source meta-class, the reference to it can be omitted (in the example, `with film`). There can be multiple joins in a query, and each join can have multiple joins conditions, which all need to be fulfilled by the combined instances. (In the example, the expression join condition and the feature join condition are equivalent, and either one could be omitted.)

While the primary source meta-class, as well as the joins, select the source meta-classes, and specify how to combine their instances, the optional filter of a query restricts the instances or tuples of joined instances that are selected. From each selected instance or each selected tuple of instances, an instance of the target meta-class of the query is created in the view. The filter is defined using the keyword `where` followed by a boolean expression, such as `film.year > 2020` in Listing 2. The expressions in NeoJoin are specified in the Xbase expression language. In the expressions, the defined aliases for instances of the source meta-classes, such as `film`, are available and give access to its

[1] https://eclipse.dev/emfatic/ (accessed 2025-11-11 12:00)
[2] https://eclipse.dev/Xtext/documentation/305_xbase.html (accessed 2025-11-11 12:00)

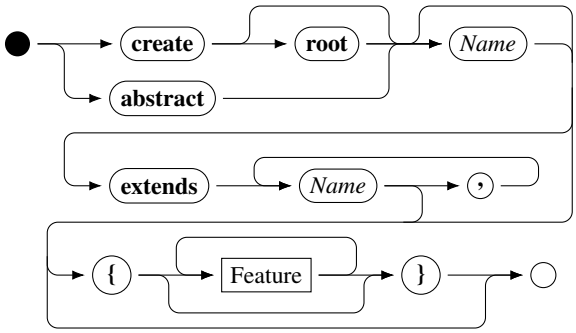

Figure 4 Syntax for the target section of a query.

features, such as `year` in Listing 2.

```
group by film.year
```

Listing 3 Group statement

Developers can create instances of the target meta-class for each single instance or tuple of joined instances from the source meta-models, but they also have the option of grouping the source instances using the statement `group by`: In the optional grouping part of a query, instances or tuples of instances are grouped by the result of the given Xbase expression, such as `film.year` in Listing 3. The expression can have any result type. Instances or tuples of instances are in the same group if the expression results in the same value. In the example, films from the same year would be grouped together. As there are now multiple instances or tuples of instances selected to create an instance of the target meta-class, the aliases for the instances of the selected source meta-classes now represent multiple instances each: instead of being of type `Film`, the alias `film` is then created as a variable of type `List<Film>`.

3.3. Creating the Target Meta-Classes

The target section specifies a meta-class in the view type, to which the instances selected in the source section are transformed, if there is a source section. The general syntax structure of the target section is shown in Figure 4. This syntax structure covers four use cases for queries: First, defining a new target meta-class and transforming instances of one or multiple source meta-classes; second, copying a single source meta-class to the target and transforming its instances; third, defining a new target meta-class with a single instance; and fourth, defining an abstract target meta-class without instances. In this subsection, we explain and illustrate the syntax for the four use cases.

```
from Film film
join DVD dvd
  using title
create Movie extends Item { /* ... */ }
```

Listing 4 Example query with inheritance.

The example in Listing 4 demonstrates the first use case: The query defines the target meta-class `Movie` by combining the source meta-classes `Film` and `DVD`. The keyword `create` indicates that target instances of the defined target meta-class are created. As the query contains a source section, a target instance is created for each tuple of joined source instances. The name of the defined target meta-class (`Movie`) is optional and defaults to the primary source meta-class. `Movie` will extend the meta-class `Item`; additional meta-classes could be added after `Item`, separated by commas, as shown in Figure 4. The extension mechanism is limited to the creation of the target meta-classes; there is no inheritance between the queries defining the meta-classes `Item` and `Movie`. The body of the target section defines the features of the target meta-class, as well as the transformation of the feature values (see subsection 3.4).

```
from Film film
where film.year > 2020
create Movie
```

Listing 5 Example query with renaming.

An example for the second use case is shown in Listing 5, indicated by the missing target section body (i.e., missing braces after the name of the target class). The query in the example defines the target meta-class `Movie` by copying all features from the source meta-class `Film` and transforming its instances to the target. The selected source instances can be limited by filters, but joins and groupings are not allowed because these would require an explicit definition of the features of the target meta-class. The name of the target meta-class is optional and if omitted, determined by the name of the source meta-class. If the source meta-class of the query references other meta-classes in its meta-model, queries are implicitly created that copy the meta-classes and transform their instances to the target. This feature allows developers to easily create views on entire models or, regarding the included meta-classes, subsets of them.

```
create root MovieDatabase {
  name := "All movies since 2020"
}
```

Listing 6 Example query defining the target meta-class `MovieDatabase`, as well as a single instance of it.

The third use case is relevant primarily for creating a single root meta-class, as in Listing 6. Views can optionally have a single root meta-class with a single instance. If a single root meta-class is present, it will contain all non-contained meta-classes; if a single instance is present, it will contain all non-contained instances. In the example, the root meta-class `MovieDatabase` is defined, together with a single instance. This is indicated by the missing source section of the query. As there is no source meta-class, the name of the target meta-class is mandatory, and all features and feature values have to be specified explicitly, such as `name` in the example. We explain the syntax for defining features and their values in detail in subsection 3.4. By omitting `root`, also non-root target meta-classes with a single instance each can be defined.

```
from DVD, Book
abstract Item {
  title = Book.title
  length = DVD.length
  genre: EString
}
```

Listing 7 Example query with abstract target meta-class .

Finally, the fourth use case creates an abstract target meta-class, as shown in Listing 7. In the example, `Item` is abstract, there are

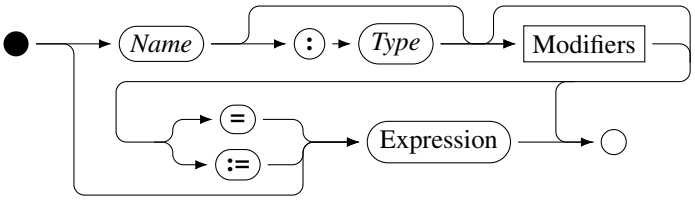

Figure 5 Syntax for a feature definition.

no direct instances created, indicated by replacing the keyword `create` with the keyword `abstract`. The features of the target meta-class can still be copied from one or multiple source meta-classes (`DVD` and `Book`). The instances of the source meta-classes are, however, not transformed by such a query. In this special case, developers can specify multiple primary source meta-classes and use the names of the meta-classes as variables to define the features of the target meta-class. An abstract meta-class can also extend other meta-classes using `extends`, as shown in Listing 4. In addition to copied features, the developer can define features by providing a name, a type, and optionally additional modifiers (see subsection 3.4). Calculating feature values is not supported. An abstract meta-class can be defined without copying features from source meta-classes, omitting the source section of the query. If the query has a single source meta-class instead, the name of the target meta-class and the body of the target section can be omitted. In this case the name or the features are copied from the source meta-class.

3.4. Defining the Features of Target Elements

The target section of a query defines its target meta-class and the transformation from the source instances or tuples of source instances to the target instances. In subsection 3.3, we describe four different use cases for creating target meta-classes, covered by the syntax for the target section. In most use cases, when the query does not copy a source meta-class to the target as-is, the developer specifies the features of the target meta-class explicitly. The feature definitions are written between the braces (`{}`) in the body of the target section, as shown in Figure 4.

The syntax diagram in Figure 5 now shows the syntax structure of feature definitions in NeoJoin. With the shown syntax, we support three different kinds of feature definitions: First, copied features from one of the source meta-classes; second, explicitly defined features; and third, abstract features. In the remainder of this section, we explain the syntax for the three kinds and show examples for each.

```
film.title
released = film.year
```

Listing 8 Copied features

Listing 8 shows two minimal examples of the first use case. In the first line, we define a feature by copying the feature `title` from the source meta-class `Film`. We do this by providing an expression resolving to a feature of one of the source meta-classes using its alias, such as `film` in this example. In the second line, we additionally rename the copied feature `year` to `released`. For this, we use the copy operator (`=`).

In case there is only a single source meta-class with the implicit alias `it`, as described in subsection 3.2, the features of the source meta-class can be accessed directly, omitting the alias `it`. Instead of `film.title` or `it.title`, we simply write `title`. This shortcut is available in all expressions, including expressions in the source section, and calculated features.

```
recent: EBool := film.year > 2020
```

Listing 9 Explicitly defined feature

In the second use case, demonstrated in Listing 9, we define a new feature by providing a name and type for it, as well as an expression for calculating the instance values. We do this using the calculate operator (`:=`). The expression for calculating the feature value can access the instances of all source meta-classes using their aliases and is, as all expressions in NeoJoin, written in the Xbase expression language. When omitting the type of a feature, NeoJoin infers it from the expression. In the example above, the type statement `: EBool` could therefore be omitted.

```
genre: EString
```

Listing 10 Abstract feature

When defining an abstract meta-class in the target, as, e.g., shown in Listing 7, developers can define the abstract features of the meta-class by specifying their name and type. In Listing 10, we create the abstract feature `genre` of type `EString`. When creating instances of a concrete meta-class extending the abstract meta-class, developers have to provide an expression for the value of the inherited feature. For the abstract feature defined in Listing 10, we could, e.g., define the inherited feature as `genre := film.genre`. Note that we use the calculate operator `:=`, as the inherited feature is already defined and, therefore, cannot be copied from a source meta-class.

```
name: EString [id, 1..1] := film.title
```

Listing 11 Modifiers

In addition to the name, type, and, if applicable, expression, a feature definition can also contain a list of modifiers, which are included in the generated view type. If the feature is defined by copying an existing feature, the explicitly specified modifiers override the copied ones. Note that it is not possible to override the modifiers of an inherited feature.

As shown in Figure 5, the modifiers are specified after the type or name, if the type is inferred, of a feature. The modifiers in a feature definition are enclosed in brackets (`[]`) and separated by commas (`,`). Listing 11 contains a feature definition, in which we define the feature `name` to have exactly one value, which is used as the identifier of the generated instances. NeoJoin supports the following feature modifiers: `unique`, `ordered`, `unsettable`, `id` (only for attributes), `containment` (only for references), `derived`, `changeable`, `volatile`, and `transient`. Every feature modifier can be used as-is or negated by prefixing it with an exclamation mark (`!`).

In addition to using modifier keywords, developers can define the multiplicity of a feature by specifying lower and upper bounds. The lower and upper bounds are integer numbers and have to be greater or equal to 0 and 1, respectively. The multiplicity of a feature is defined as part of the modifier list, using

the syntax of Emfatic, by separating the lower and upper bounds with two dots (`..`), as shown in the example above. To define a feature with an unlimited number of values, the upper bound can be replaced with a star symbol `*`. NeoJoin also supports three shorthands: `*` for the multiplicity `0..*`, `+` for `1..*`, and `?` for `0..1`.

```
actors = film.actors create Actor {
  name = it.name
}
```

Listing 12 Reference definition with a sub query

While the feature definition in NeoJoin generally works the same for attributes and references, there are a few aspects specific to references. First, as reference definitions are specified on source instances and implicitly mapped to corresponding target instances, it can happen that the target of a reference in the view is ambiguous. This can happen when a single source meta-class is used in multiple queries. To make the transformation unambiguous, NeoJoin requires an explicit type as part of the reference definition in those cases.

A feature specific to reference definitions are sub queries, which can be used to quickly create queries for referenced meta-classes. Sub queries extend a reference definition by appending the keyword `create`. The definition of a sub query is similar to the target section of a regular query, but cannot be used to create root or abstract meta-classes. It is interpreted as a regular query, with the type of the reference being the single source meta-class. In Listing 12, the reference `actor` from the source meta-class `Film` is copied to a target meta-class, including a sub query to create the target meta-class `Actor` for the referenced instances. Again, the name of the target meta-class is optional and defaults to the name of the source meta-class. The body of the target section of a query, including the opening and closing braces (`{}`), can be omitted to implicitly create queries that copy the referenced meta-class, its attributes, references, as well as all referenced meta-classes (as described above in subsection 3.3).

4. Implementation

We have implemented the syntax presented in section 3 in a prototype for NeoJoin[3]. The structure of the prototype was first presented in (König, Stickling, & Burger 2025). Our prototype consists of a parser, a meta-level transformation, as well as two model-level transformation backends. In addition, it includes two front ends: a command line interface and an extension to the IDE Visual Studio Code The extension to Visual Studio Code provides syntax highlighting, auto-completion, syntax validation, and a live visualization of the view type meta-model.

The parser module uses the Eclipse Modeling Framework (EMF) to parse a textual view definition, using the syntax presented in section 3, referencing source meta-models in the Ecore format. We implemented the parser using the Xtext parser framework[4]. The module responsible for the meta-level transformation generates the Ecore view type meta-model from source meta-models, using the parsed view definition. For the model-level transformation, we use the parsed view definition as an interface, such that we can extend our prototype with multiple transformation backends. While the parser and view type generation are dependent on the EMF framework, the parsed view definition can be serialized in any format, allowing us to integrate non-EMF backends as well. We currently support a unidirectional, EMF-based transformation backend as a reference implementation, and work on integrating a bidirectional, incremental transformation backend based on triple graph grammars (König, Ritz, & Burger 2025).

As part of the implementation, we created unit tests. The `language` module, which contains the implementation of the parser and the view type generation, has a line coverage of 68.0 % and a condition coverage of 53.9 %, as calculated in our continuous integration pipeline. The `backend-emf` module, which contains the unidirectional reference implementation for the model-view transformation, has a line coverage of 75 % and a condition coverage of 46.7 %.

5. Evaluation

Our evaluation of the syntax of NeoJoin (presented above in section 3), using a prototypical implementation, is based on these four evaluation goals: (G1) Usability, (G2) Expressiveness, (G3) Reliability, and (G4) Conciseness.

We evaluated the *expressiveness* and *conciseness* of the proposed syntax because our overall aim is to support general view definitions with an easy-to-use syntax. Following this, we evaluated the usability of the proposed syntax by performing a user study with 13 participants (see subsection 5.1). We further applied our language to an existing case for view-based development from the domain of energy systems. The case (see subsection 5.2) applies view-based modelling to the problem of outage management in smart electricity grids (Burger, Mittelbach, & Koziolek 2016). All evaluation material is available in our replication package (König et al. 2026).

5.1. *Usability* (G1): User Study

Here, we explain the evaluation design of our user study, present the results, and discuss their validity. To set a baseline, we also included the view definition language ModelJoin (Burger, Henss, et al. 2016) in the user study. In the user study, participants were assigned either to ModelJoin or to our language, NeoJoin. A more detailed description of the user study is available in (Stickling 2025) (included in our replication package).

5.1.1. *Study Design*

All material, including the questionnaires and tasks, is available in our replication package (König et al. 2026).

Structure, Execution, and Environment We structured the user study in three phases: During the first phase, the participants filled out a pre-study questionnaire for demographic data, regarding their professional background and experience with model-driven development. During the second phase, the participants were shown a short tutorial document, introducing the syntax of the assigned language. During the third phase, the participants were asked to perform a set of tasks in the assigned

[3] https://github.com/vitruv-tools/NeoJoin (accessed 2025-12-15 14:35)
[4] https://eclipse.dev/Xtext/ (accessed 2025.12.15 14:30)

language. We performed the study unsupervised in an uncontrolled environment, using the online tool LimeSurvey[5]. The participants were able to interrupt the survey and continue again later. We encouraged participants not to use additional resources for solving the tasks.

Object of Study Our goal was to evaluate the usability of the view definition languages NeoJoin and ModelJoin. However, the two languages have different levels of IDE support in different IDEs, Visual Studio Code[6] for NeoJoin and Eclipse[7] for Model Join. As not to bias the results through the use of different IDEs and different levels of IDE support, we decided to evaluate only their textual syntax. Therefore, we used plain text input for both languages throughout the study.

Participants Our only requirement was that the participants have a background in computer science or software development. This requirement aligns with our intention to design a view definition language that is easy-to-use, re-using concepts from database query languages, for example. We used convenience sampling and assigned each of the participants to one of the two languages. As we did not know the number of participants beforehand, we randomly assigned them upon opening the survey for the first time. We excluded the name of the assigned language from all survey documents and did not inform the participants whether they were assigned the language we developed.

Tasks We developed a total of four tasks, applicable to both the ModelJoin language and the NeoJoin language. All tasks used the same two small source meta-models, which we adapted from an example in the ModelJoin wiki[8]. The first two tasks required the participants to read a view definition, i.e., queries, in the assigned language and describe the resulting view type. The description of the resulting view type should include the classes and their features, including their types. For the remaining two tasks, the participants received a natural language description of the view definition and were tasked to write the view definition in the language assigned to them. In general, we intended to cover all basic features of both languages with the tasks, including joins, attribute definitions, and sub queries. We excluded features supported by only one of the languages, such as grouping in NeoJoin or abstract classes in ModelJoin (at the time of the study). We further tried to reduce the influence of the expression languages on the results of our usability evaluation by using and requiring only simple expressions in the tasks. The expression languages used are Xbase in NeoJoin and OCL in ModelJoin.

Pilot Study While preparing the study material, we ran a pilot study with colleagues, all with a background in model-driven engineering. The results of the pilot study were used to improve the study material and to better align the time required for the study with the time intended to be required. None of the participants from the pilot study participated in the actual study.

[5] https://www.limesurvey.org/de (accessed 2025-12-08 15:45)
[6] https://code.visualstudio.com (accessed 2025-12-08 15:50)
[7] https://eclipseide.org (accessed 2025-12-08 15:50)
[8] https://sdq.kastel.kit.edu/wiki/ModelJoin/Getting_Started (accessed 2025-12-08 16:00)

NeoJoin | ModelJoin	$\bar{x}$	σ	$\bar{x}$	σ
TCR [%]	91	0.13	73	0.11
Time [minutes]	6.03	2.27	7.56	2.95
RTLX	48.33	15.91	51.25	12.50
SUS	75	13.05	65	6.12

Table 1 Overview of the mean $\bar{x}$ and standard deviation σ of all metrics collected for the view definition languages NeoJoin (left) and ModelJoin (right) as part of the user study.

Measures and Analysis Our evaluation goal (G1) *usability* can be broken down into the effectiveness, efficiency, and satisfaction of solving tasks (ISO 2018). In our evaluation we considered the properties: effectiveness, efficiency, and general usability. As a measure of effectiveness, we chose the Task Completion Rate (TCR) (ISO 2018). We calculated the TCR using the average editing distance between the solution of the participants and our expected solution across all tasks. To calculate the editing distance we ignored minor errors, such as spelling or capitalization, and deducted points for errors in the view definition. We deducted 0.5 points when an element, e.g., a feature definition, had a single error, and 1 point when an element had more than one error or could not be matched to an element of our expected solution. A detailed description of this is provided by (Stickling 2025) (included in our replication package). To measure the efficiency with which the participants solved the tasks, we employed the NASA Raw Task Load Index (RTLX) (Hart & Staveland 1988), as well as the task completion time. We used the standardized questionnaire for the RTLX, limited to the areas: mental demand, perception of own performance, effort, and frustration. For the task completion time, we instructed the participants to track the time required for solving the tasks. Finally, we used the system usability scale (SUS) (Brooke 1996) to assess the general usability, using the standardized questionnaire.

5.1.2. Results on Usability *(G1)*

We provide demographic information about our participants, then we present the results, and finally we discuss threats to the validity. We include an overview of the collected metrics in Table 1.

Sample Description We had a total of $n = 13$ participants, of which $n_{NJ} = 9$ were randomly assigned to NeoJoin, while the remaining $n_{MJ} = 4$ were assigned to ModelJoin. Most of the participants had a university degree, with 2 participants having a bachelor's degree, 7 participants having a master's degree, and 1 participant having a PhD. In addition, most of the participants have at least some work experience, with 11 participants having at least one year of experience and 5 participants having more than 5 years of experience. The participants mostly work as software engineers (4 participants) or researchers (4 participants), while one participant was a student. The remaining participants did not state their profession. The participants, who were

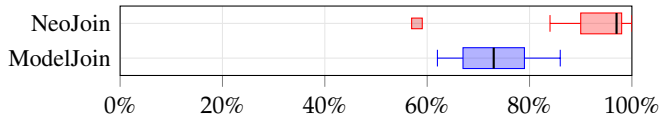

Figure 6 Average Task Completion Rate (TCR) (ISO 2018) across all reading and writing tasks.

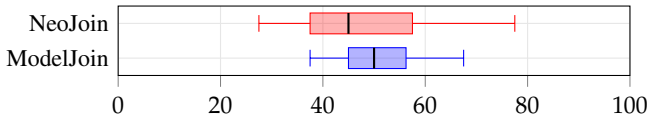

Figure 7 NASA Raw Task Load Index (RTLX) (Hart & Staveland 1988) for all tasks. Lower scores indicate a lower perceived task load.

randomly assigned to ModelJoin, had mostly school degrees, except for one participant having a master's degree, however, two of them reported 2–5 years of work experience and the other two reported more than 5 years of work experience.

We asked the participants to state their experience with relevant technologies on a scale from 1 (no experience) to 5 (extensive experience). The participants reported mixed experience with model-driven technologies: on average, model-driven software development received a rating of 2.7 and EMF/Ecore receiving a rating of 2.0. These ratings make for a valid sample to test the usability of NeoJoin, as its intended users may not have a background in model-driven software development. The participants assigned to NeoJoin on average reported a higher experience with model-driven technologies (3.222) and EMF/Ecore (2.333) than the participants assigned to ModelJoin (1.5 and 1.25, respectively). Only one participant reported any experience with ModelJoin, rating his or her experience with 2, and was randomly assigned to the NeoJoin group. We therefore assume that previous experience with ModelJoin has not significantly influenced our results. All participants reported at least some experience with general software development technologies, with SQL receiving an average rating of 3.2 and Java receiving an average rating of 4.1. Thereby, they fulfill our requirements for developers working with NeoJoin, as we use concepts from database query languages and meta-model description languages, which share concepts with object-oriented languages, such as Java.

Effectiveness We used Task Completion Rate (TCR) (ISO 2018) to evaluate participants' effectiveness at solving the tasks in the language assigned to them. The results are shown in Figure 6. All participants were able to solve all four tasks, including two reading tasks and two writing tasks (detailed in subsubsection 5.1.1). While the TCR is high for both languages, most participants using NeoJoin achieved a higher TCR than the participants using ModelJoin. The second lowest TCR for NeoJoin was 84 %, while the highest TCR for ModelJoin was 86 % – however, there was one participant assigned NeoJoin who had difficulties with the reading tasks (indicated by the outlier at 58 %). For the reading tasks alone, the median TCR for NeoJoin was 100 %, while the median TCR for ModelJoin was 86 %. Our results suggest that a developer using NeoJoin to complete typical development tasks will do so more effectively than a developer using ModelJoin.

Efficiency We used Raw Task Load Index (RTLX) (Hart & Staveland 1988) to evaluate efficiency as a measure of the load handled by a participant on task (results shown in Figure 7); and we used Task Completion Time to evaluate efficiency as a measure of the time spent by a participant on task (results shown in Figure 8.) We calculated the RTLX values from the standardized questionnaire, filled in by participants after task completion. The median values for NeoJoin (45) and ModelJoin (50) are close to the empirical baselines for *Cognitive Activities* (46) and *Computer Activities* (54), as reported by (Grier 2015). We calculated the average task completion time as the average time required to complete a single task across all tasks. As shown in Figure 8, the participants spent a median of 6 minutes per task using NeoJoin and a median of 7.75 minutes using ModelJoin. Overall, NeoJoin achieved values similar to those achieved by the established view definition language ModelJoin; still, NeoJoin achieved smaller median values for both measures. Hence, developers using NeoJoin should be entirely capable of developing view definitions with an acceptable task load and within a reasonable time. However, our results cannot definitively demonstrate that NeoJoin improves the efficiency of solving tasks over ModelJoin, because our sample size is small, and the resulting value ranges overlap.

Usability We used the System Usability Score (SUS) (Brooke 1996) to evaluate the general usability of the two languages. We used the standardized questionnaire, filled in by participants after task completion (results shown in Figure 9). To interpret the results, we consulted the adjective rating scale from (Bangor et al. 2009). The median SUS score for NeoJoin (77.5) falls between the mean values for the adjectives *Good* (71.4) and *Excellent* (85.5), while the median score for ModelJoin (66.25) falls between the mean values for *OK* (50.9) and *Good* (71.4). Although the worst score for NeoJoin (52.5) is lower than the worst score for ModelJoin (57.5), the results of our study suggest that NeoJoin offers good usability and is more usable than the comparable view definition ModelJoin.

5.1.3. Threats to Validity

Overall, our small and unbalanced sample is a threat to the *conclusion validity* of our study. We are therefore only able to draw very limited conclusions.

We identified two threats to the *internal validity* of our study: the selection of tasks and the quality of the provided material. As the tasks we gave to the participants do not cover all features of the two languages, the selection of tasks and therefore features

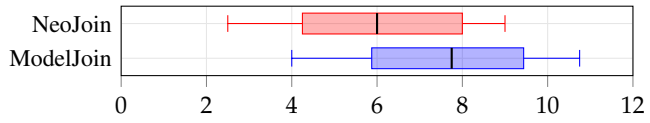

Figure 8 Average Task Completion Time (in minutes) across all tasks.

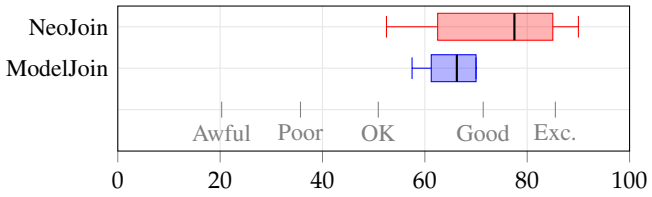

Figure 9 System Usability Score (SUS) (Brooke 1996) for all tasks. Higher scores indicate a higher usability. The values assigned to the adjectives indicate the mean score in the respective category, as reported by (Bangor et al. 2009).

could have had an influence on the results. We mitigated this by choosing tasks that cover the basic features shared by the two languages. Another influence may be the quality of the prepared material with which we introduced the assigned language to the participants. Especially for ModelJoin, our experience with the language is limited, which might have led to a worse explanation of its syntax and features. We tried to mitigate this by following a similar structure for the language tutorials and re-using (where possible) text blocks between the documents. In addition, an author of this work was involved in the development of the view definition language ModelJoin (Burger, Henss, et al. 2016), against which we compared NeoJoin in this user study. As this could pose a major threat to the validity of our user study, the mentioned author did not participate in the creation of survey material, the collection of survey results, or their analysis.

We identified the following threats to the *external validity* of our study. We chose the participants using convenience sampling, therefore our sample might not be representative for the larger population, i.e., software developers working with models. As shown in subsubsection 5.1.2, the demographic data of our participants indicates that they are from the population we consider relevant. We used random assignment to reduce the influence of previous knowledge of the participants on the results. This did, however, lead to an uneven distribution between the groups, including unbalanced experience levels in the two groups. Finally, our study required the participants to solve the tasks using plain text code in a web browser. We chose to do this to limit the influence of different IDEs and levels of IDE support, as described in subsubsection 5.1.1. It is, however, an artificial situation when developing code, including view definitions. Consequently, this may limit the generalizability of our results.

5.2. *Expressiveness* (G2), *Reliability* (G3), and Conciseness (G4): Case Study

We applied NeoJoin to the case created by (Burger, Mittelbach, & Koziolek 2016). This case applies view-based modeling to improve the outage management in smart electricity grids. The authors derived meta-models and models from publicly available, real-world data, and implemented view definitions in ModelJoin (Burger, Henss, et al. 2016). In this section, we describe the case in subsubsection 5.2.1, explain our evaluation setup in subsubsection 5.2.2, and present the results in subsubsection 5.2.3, subsubsection 5.2.4, and subsubsection 5.2.5. We finally discuss threats to the validity of our study in subsubsection 5.2.6.

5.2.1. Outage Management Case

We use the extended outage management system case described by (Burger, Mittelbach, & Koziolek 2016). They describe operating modern electricity grids as an increasingly complex task. Challenges arise from the integration of more renewable energy sources, decentralized energy generation and storage, as well as difficult to predict loads, e.g., due to the charging of electric vehicles. These challenges affect the reliability, stability, and efficiency of electricity grids and their operation. To overcome these challenges, grid operators rely on smart grids, which complement the traditional electricity grid with hardware and software for communication, metering, repair, and control. An important task supported by smart grids is outage management, i.e., the detection and prevention of grid outages. However, the software systems used for outage management cover different domains and, as such, rely on heterogeneous standards for representing data. Therefore, communication and data integration is difficult. To improve on this, Burger, Mittelbach, & Koziolek created the extended outage management system, consisting of a unified data model and multiple view types, specific to the different stakeholders and their tasks. They build their unified model by combining meta-models representing the different standards with explicit consistency preservation rules. The standards either directly support an EMF-based representation or a data format which can be mapped to an EMF meta-model.

The extended outage management system (Burger, Mittelbach, & Koziolek 2016) integrates three standards. The IEC 61970/61968 standards (IEC 2025b, 2024) defines the *Common Information Model (CIM)*, which includes "physical components, measurement data, control and protection elements, and the SCADA [supervisory control and data acquisition system] system" (Burger, Mittelbach, & Koziolek 2016). Complementary, the IEC 61850 standard (IEC 2025a) contains definitions "supporting interoperability of intelligent electronic devices in substation automation systems" (Burger, Mittelbach, & Koziolek 2016). We refer to this model as *Substation Standard*. Lastly, the IEC 62056 standard (IEC 2023) defines the *Companion Specification for Energy Metering (COSEM)*, which describes the "data exchange for meter reading, tariff and load control" (Burger, Mittelbach, & Koziolek 2016).

In addition to creating a unified data model, Burger, Mittelbach, & Koziolek defined 14 view types for specific stakeholders and tasks. They consider as stakeholders (a) control center operators and (b) the smart grid software systems (Burger, Mittelbach, & Koziolek 2016). The view types, each defined on a subset of the included meta-models, are shown in Figure 10.

5.2.2. Setup

For our evaluation, we recreated the view definitions for the extended outage management system (Burger, Mittelbach, & Koziolek 2016). We used the data available in the replication package for the master's thesis of (Mittelbach 2015) (included in our replication package (König et al. 2026)), which includes meta-models for the three standards introduced in subsubsection 5.2.1, the view definitions in the ModelJoin language (Burger, Henss, et al. 2016), the resulting view type

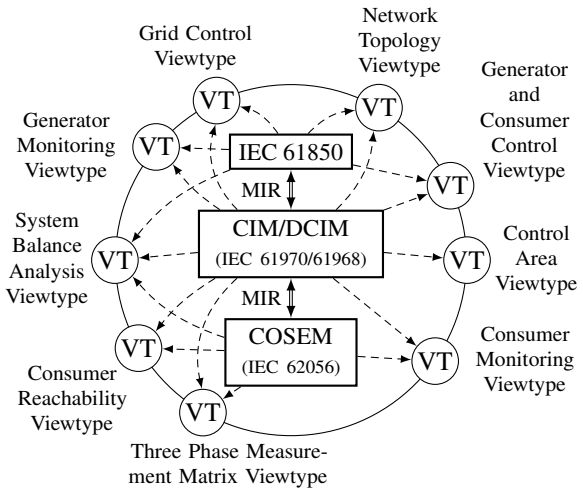

Figure 10 Meta-models and view types of the extended outage management system (Burger, Mittelbach, & Koziolek 2016). Ⓥ = view type, -▸ = source meta-models, ◂▸ = consistency preservation specification

meta-models, as well as some view instances.

When translating the ModelJoin queries to NeoJoin, we closely followed the structure of the ModelJoin queries. We used shorthands in the NeoJoin syntax where possible, e.g., omitting the alias in queries with a single source class (described above in subsection 3.2). The ModelJoin queries appear to be designed similarly, using shorthands in the ModelJoin syntax where possible. We used sub queries (described above in subsection 3.4) for referenced classes, as in the ModelJoin queries, but not for sub or super class references, as this is not supported by sub queries in NeoJoin. As ModelJoin does not support creating an alias for a class name, we used the unqualified class name in lower camel case as an alias to enable a fair comparison of the conciseness of the queries. All evaluation data, including the meta-models, view definitions in both languages, resulting view types, and views are available in our replication package.

We validate that our NeoJoin queries describe the same view type as the ModelJoin queries, using our implementation (described above in section 4) to generate the view types as Ecore files from NeoJoin view definitions. The view types for the ModelJoin queries were already available as Ecore files in the replication package of (Mittelbach 2015) (included in our replication package). We removed the `eAnnotations` elements from the ModelJoin view types using a small, self written tool, and compared the two Ecore files using another, self-written tool. For the comparison, we matched elements with the same name. We compared all `EClass` elements, including their super types, attributes, and references. For attributes and references, we compared their type and options. All detected differences were written to diff files, which are available in our replication package, together with source code of the tool we used for the comparison.

In an effort to validate the *reliability* (G3) of our implementation and, by extension, the semantic equivalence of view definitions in NeoJoin and ModelJoin (Burger, Henss, et al. 2016), we used our unidirectional, state-based reference implementation to create view instances from the available source model instances and our NeoJoin view definitions. As the view instances included in the replication package of (Mittelbach 2015) are incomplete, we only validated that the view instances were created successfully. In addition, we created a description of the expected view instance of a single view type by hand and compared it to the view instance generated by NeoJoin.

To evaluate *conciseness* (G4), we employed three metrics: *line count*, *character count*, and *level of indentation*. We determined the line count using the standard tool cloc[9] with custom language configurations for excluding lines that only contain comments. Empty lines are excluded by cloc by default. We counted non-whitespace characters using the regex `[^\s]` with the default search engine in Visual Studio Code. We counted the level of indentation manually.

5.2.3. *Results on* Expressiveness *(G2)*

We were able to recreate all 14 view types of the extended outage management system with NeoJoin, with only minor differences in the resulting view types. Here we present and explain the differences between the view types generated from the view definitions in ModelJoin and NeoJoin. We refer to the view types of the extended outage management system by their number, as defined in our replication package.

One general difference between the view types generated by NeoJoin and ModelJoin is the presence of a single root element in NeoJoin view types. As this is a design choice rather than a difference in expressiveness, we ignore it in this section.

In view type 1, ModelJoin assigns the type `EString` to the attribute `State` in the meta-class `GeneratingUnit`, while NeoJoin assigns the type `GeneratorStateKind`. Similarly, in view type 10, ModelJoin assigns the type `EInt` to the attribute `TapChange` in the meta-classes `ARCO` and `ATCC`, while NeoJoin assigns the type `BSControlKind`. Both difference are expected, as NeoJoin does not support converting Enum values to String or Integer values. Support could be added by registering extension functions that would be available in the Xbase expressions used for the feature value calculation, which we expect to be easy to implement.

View type 2 in the extended outage management system is defined in multiple steps, where the later steps add additional calculated attributes. With NeoJoin, we decided to recreate only the first step, but we assume the later steps to work in the same way. In the first step, however, there is a difference between the view types generated by ModelJoin and NeoJoin. In the ModelJoin view type, the meta-class `NonConformLoadGroup` refers to the meta-class `ConformLoad`, while it refers to the meta-class `NonConformLoad` in the NeoJoin view type. As the source meta-model *CIM* contains a reference to the meta-class `NonConformLoad` in this place, we assume this to be an error in the ModelJoin view definition.

Another difference between the view types occurs in view type 3 where the attribute `Type` in the meta-class `GeneratingUnit` is missing from the view type generated by NeoJoin. This is expected, as in the ModelJoin view definition, it is defined using instance-of checks in the expression calculating the value of the

[9] https://github.com/AlDanial/cloc (accessed 2025-12-14 14:00)

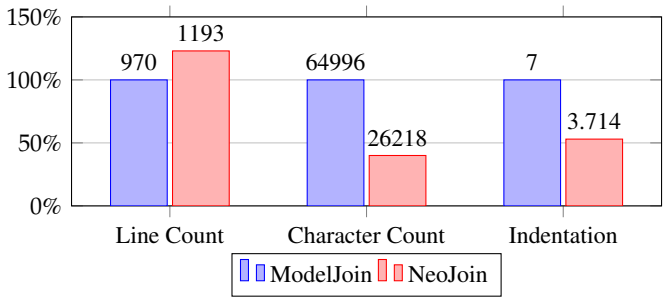

Figure 11 Comparison of our conciseness metrics *line count*, *character count*, and *level of indentation* for ModelJoin (left) and NeoJoin (right). The *level of indentation* shows the average over the maximum level of indentation per view definition. We use the ModelJoin values as a baseline at 100 % and show the absolute values above the bars.

attribute, which are not supported in NeoJoin. For simple cases, this could be resolved by creating explicit queries for all classes used in the instance-of checks. Supporting this for all cases would, however, require an explicit implementation of instance-of checks in the Xbase runtime model used by NeoJoin, which seems possible, although we did not investigate this further.

Finally, in view types 7, 8, 9, and 14, there are super classes missing in multiple places in the view types generated by NeoJoin. However, the super classes do not occur in the ModelJoin view definitions. We investigated the issue with ModelJoin developers and assume this to be an error in the view types generated by ModelJoin.

5.2.4. *Results on* Reliability *(G3)*

Using our unidirectional, state-based model-view transformation, described in section 4, we were able to successfully generated view instances for 12 of 14 view types of the extended outage management system (Mittelbach 2015), shown in Figure 10. The remaining two view types contained small errors in the view definition, which we detail in this section. In addition to generating the view instances, we also validated view type 4 by comparing the generated view instance against a description of the expected view instance, which we created by hand, and found no differences. One overall difference between the view instances generated by NeoJoin and ModelJoin (Burger, Henss, et al. 2016) seems to be that, as NeoJoin lifts sub queries to regular queries, all instances of the respective source meta-classes are transformed, instead of only the referenced instances.

The first view definition, which caused transformation errors, belongs to view type 2 of the extended outage mangement system. In our view definition, we transform all instances of the meta-class `ControlAreaGeneratingUnit` using a sub query, include their references to instances of the meta-class `GeneratingUnit`. However, instances of the meta-class `GeneratingUnit`, as well as its sub classes, are transformed using a join with the meta-class `ZGEN`, for which the available model contains no instances. As a result, the referenced instances in the source model could no be resolved in the view instances. In our replication package, we provide two alternative fixes for the view definition, removing either the reference or the join with the `ZGEN` meta-class.

An additional problem in the view definition for view type 2 was the transformation of instances of concrete super classes and their sub classes. This was caused by the view definition containing queries for the meta-classes `EnergyArea` and `GeneratingUnit`, as well as queries for their sub classes, which led to the instances of the sub classes being transformed twice. We fixed this by explicitly excluding instances of the sub classes from being transformed by the query of the super class in our transformation, but this requires a more systematic solution.

The other view definition, which caused transformation errors, belongs to view type 5. Here, the attribute `ID` is calculated by following the reference `ServiceDeliveryPoint`. However, as the reference is `null` for some instances in the source model, a null-check is required, which was either missing or implicitly applied in the ModelJoin view definition. After adding the null-check, we were able to generate the view instance.

5.2.5. *Results on* Conciseness *(G4)*

In our evaluation of the conciseness of NeoJoin, we considered three metrics (results shown in Figure 11). While the metrics *line count* and *character count* show the total count over all view definitions, the metric *level of indentation* shows the average over the maximum level of indentation per view definition.

As can be seen in Figure 11, the line count in the NeoJoin view definitions is 23 % higher than the line count in the ModelJoin queries. We attribute this primarily to the fact that ModelJoin contains sub queries for transforming sub and super classes from within a query for a meta-class. We chose not to provide support for this because we consider sub queries for sub and super class difficult to understand. On the other hand, not providing support for sub queries for sub and super classes is the main reason why NeoJoin view definitions have a 47 % lower maximum level of indentation on average. NeoJoin view definitions for the extended outage management system have a maximum level of indentation of 4 (instead of as many as 10 in ModelJoin view definitions). In conclusion, we have more queries in NeoJoin view definitions, which require fewer levels of indentation. Hence, we consider the language easier to read *and* to understand.

```
1 // ModelJoin:
2 keep calculated attribute "substationStandard::
      LNNodes::LNGroupY::YPTR.PwrRtg.setMag.f" as
      PowerTransformer.PowerRate:Double
3 // NeoJoin:
4 PowerRate := yptr.PwrRtg.setMag.f as double
```

Listing 13 Examples of attribute definitions in ModelJoin and NeoJoin. The examples are taken from the definition of the meta-class `PowerTransformer` in view type 1.

Overall, the number of characters required for a view definition in NeoJoin is 60 % lower than the number of characters required for a view definition in ModelJoin. We attribute this mainly to the support for aliasing and unqualified meta-class names in NeoJoin and the overall less verbose syntax. An example of this is shown in Listing 13. In the example, we see

that keywords in ModelJoin, such as `keep calculated attribute`, are realized as operators in NeoJoin, `:=` in the example. Further, instances are referenced with an alias, `yptr` in the example, while ModelJoin uses the qualified name of the meta-class instead. In places where meta-classes are referenced, such as the source section of a NeoJoin query, NeoJoin supports the use of qualified or unqualified meta-class names. In conclusion, view definitions in NeoJoin are more concise than view definitions in the established view definition language ModelJoin. Hence, we believe NeoJoin view definitions to be more readable and understandable, which benefits usability as a whole.

5.2.6. Threats to Validity

We identified two threats to the *internal validity* of our case study: limited semantic comparison and factors beyond the syntax definition influencing the conciseness of view definitions. First, while ModelJoin and NeoJoin support the transformation of model instances, i.e., the generation of views, we did not compare the generated views, as the view instances in the replication package of (Mittelbach 2015) (included in our replication package) are incomplete. To mitigate this and provide at least a limited evaluation of the semantic equivalence, we consulted a ModelJoin developer to validate our interpretation of the ModelJoin view definitions and validated a single, generated view instance against a description of the expected view instance created manually by us. Second, for our comparison of the conciseness of the syntax of the two languages, we used code-based metrics to compare the view definitions. However, the source code and therefore the used metrics depend not only on the syntax definition of the used language, but also on code style, formatting, naming conventions, etc. We mitigated this by following the structure and code style of the ModelJoin view definitions closely (described above in subsubsection 5.2.2).

One threat to the *external validity* is the selection of the evaluation case. We chose a case that is already implemented in an established view definition language, in order to have a baseline for our evaluation. On account of that, we were, however, limited in the case studies available for our evaluation. While the extended outage management system is not itself used in industry, it works in a realistic setting, using standards and data formats from industry.

6. Related Work

In their survey, (Bruneliere et al. 2019) give an overview of model view approaches, including their query and view type languages. (König, Stickling, & Burger 2025) identified approaches related to NeoJoin and compared selected features. In this section, we present the related approaches and check them against our requirements. While the focus of this paper is the language design of NeoJoin, we mention the transformation capabilities of the related approaches as well, as they are part of our overall requirements.

EMF Views (Bruneliere et al. 2015) is an EMF-based DSL for the textual description of view types. Similar to NeoJoin, it provides relational operators to derive a view type from source meta-models. The model-view transformation is derived from the view type definition. However, EMF Views only provides a limited set of transformation operators, as the view type elements and, consequently, the view elements are only proxies to model elements. It further provides only limited support for editing views and no support for incremental transformations.

The *Kitalpha* approach (Langlois et al. 2014) has a more general notion of views, which includes a concrete syntax and tools for developers using the views. For that, they provide specialized DSLs for the different aspects of views. However, they only provide a limited set of transformation operators and no support for incremental transformations.

The EMF-based *ModelJoin* (Burger, Henss, et al. 2016) view definition language is closely related to NeoJoin. It provides relational operators to define view types and model-view transformations with a similar expressiveness as NeoJoin. However, as we show in subsection 5.1, the syntax of NeoJoin provides a more effective and usable way of defining views. In addition, ModelJoin does not offer support for incremental model-view transformations.

OpenFlexo (Golra et al. 2016) is a model federation approach, mainly focused on the integration of data from different technical spaces, such as EMF, OWL, Excel, etc. They provide a language to define view types on heterogeneous meta-models with expressive transformation operators. In addition to having different views on a set of models, they also support different groups of models having the same view to keep their data consistent. In contrast to NeoJoin, OpenFlexo requires the use of different DSLs for the definition of view types and model-view transformations. This restricts its use case and makes OpenFlexo more cumbersome to use by developers and during the development of systems. In contrast, we describe the view-based development process supported by NeoJoin in subsection 2.1.

VIATRA Viewers (Debreceni et al. 2014) is an approach for model queries that can be executed incrementally. Instead of providing a DSL for view type definitions, they use annotations to the queries to define a view type. Due to that, they offer only a limited set of transformation operators, which excludes, e.g., the modification of meta-classes from the source meta-models.

In addition to DSLs for view-based development, there exist several languages concerned with the textual definition of meta-models or models, as well as model transformation languages. Generic textual DSLs include Emfatic[10] as a meta-model DSL, as well as DSLs for models, such as HUTN[11] and FlexMI (Kolovos & de la Vega 2023). While we have considered the design choices made in these languages, especially the meta-class notation of Emfatic, these languages are intended for developers to directly specify the resulting meta-models or models, while view definitions in NeoJoin specify how to derive view type meta-models and models from source meta-models and models. Similarly, model transformation languages are capable of transforming between source and target models. However, as the source and target meta-model of a model transformation can be arbitrarily different, as long as they share some semantics, model transformations are, in general, more complex to specify than view definitions, for which the target meta-model and the target models are projections of the source. We designed

[10] https://eclipse.dev/emfatic/ (accessed 2025-11-11 12:00)
[11] https://www.omg.org/spec/HUTN (accessed 2026-02-24 16:00)

NeoJoin to use this advantage to make view definitions easy to create.

7. Conclusion

In this paper, we presented an expressive, yet easy-to-use syntax for the view definition language NeoJoin (König, Stickling, & Burger 2025). The proposed syntax is based on known concepts from database engineering and object-oriented design, as to lower the entry barrier for developers without a background in model-driven engineering. We described our language design in detail and illustrated it through examples. We gave insight into a prototypical implementation and then used this implementation to evaluate NeoJoin on usability, expressiveness, reliability, and conciseness. The results from our user study provide a first indication that developers can use NeoJoin to effectively and efficiently create view definitions. Although the number of study participants was too low for us to be able to draw any final conclusions, average task completion rates and usability scores were higher for NeoJoin than for the established view definition language ModelJoin (Burger, Henss, et al. 2016). In our view-based case study from cyber-physical systems engineering, we recreated the 14 view types of the extended outage management system (Burger, Mittelbach, & Koziolek 2016) from the energy systems domain. We found only minor differences in expressiveness between NeoJoin and ModelJoin. Further, NeoJoin achieved more concise view definitions, reducing the character length by 60 % compared to ModelJoin.

Overall, we believe that our proposed syntax enables developers and engineers to use NeoJoin in flexible development processes, providing expressive transformation operators and an easy-to-use syntax, even for engineers without a strong background in model-driven engineering. Beyond the contributions of this paper, NeoJoin is designed to integrate transformation backends which support incremental, bidirectional transformations, which are crucial for the distributed development of cyber-physical systems but which currently are insufficiently supported by related view-based approaches. In future work, we therefore plan to integrate our work-in-progress TGG-based transformation backend for NeoJoin (König, Ritz, & Burger 2025) and apply NeoJoin to more cases from cyber-physical systems engineering. This will include cases with variability in the development artifacts, which (Feichtinger et al. 2022) identified as a central challenge for the development of cyber-physical production systems. We further plan to research how common or connected views can support consistency preservation between models.

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. This research was edited by our textician, Daniel Shea.

References

Atkinson, C., Stoll, D., & Bostan, P. (2010). Orthographic Software Modeling: A Practical Approach to View-Based Development. In L. A. Maciaszek, C. González-Pérez, & S. Jablonski (Eds.), *Evaluation of Novel Approaches to Software Engineering* (Vol. 69, pp. 206–219). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-14819-4_15

Atkinson, C., Tunjic, C., & Moller, T. (2015, September). Fundamental Realization Strategies for Multi-view Specification Environments. In *2015 IEEE 19th International Enterprise Distributed Object Computing Conference* (pp. 40–49). Adelaide, Australia: IEEE. doi: 10.1109/EDOC.2015.17

Bangor, A., Kortum, P., & Miller, J. (2009). Determining what individual sus scores mean: Adding an adjective rating scale. *Journal of usability studies*, *4*(3), 114–123.

Brooke, J. (1996). SUS: A 'Quick and Dirty' Usability Scale. In *Usability Evaluation In Industry*. CRC Press.

Bruneliere, H., Burger, E., Cabot, J., & Wimmer, M. (2019, June). A feature-based survey of model view approaches. *Software & Systems Modeling*, *18*(3), 1931–1952. doi: 10.1007/s10270-017-0622-9

Bruneliere, H., Perez, J. G., Wimmer, M., & Cabot, J. (2015). EMF Views: A View Mechanism for Integrating Heterogeneous Models. In P. Johannesson, M. L. Lee, S. W. Liddle, A. L. Opdahl, & O. Pastor Lopez (Eds.), *Conceptual Modeling* (pp. 317–325). Cham: Springer International Publishing. doi: 10.1007/978-3-319-25264-3_23

Burger, E. (2013, June). Flexible views for view-based model-driven development. In *Proceedings of the 18th international doctoral symposium on Components and architecture* (pp. 25–30). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2465498.2465501

Burger, E., Henss, J., Küster, M., Kruse, S., & Happe, L. (2016, May). View-based model-driven software development with ModelJoin. *Software & Systems Modeling*, *15*(2), 473–496. doi: 10.1007/s10270-014-0413-5

Burger, E., Mittelbach, V., & Koziolek, A. (2016, October). View-based and Model-driven Outage Management for the Smart Grid. In S. Götz (Ed.), *11th International Workshop on Models@run.time (MRT) co-located with 19th International Conference on Model Driven Engineering Languages and Systems (MODELS)* (Vol. 1742, pp. 1–8). Saint Malo, France: RWTH Aachen. Retrieved from https://st.inf.tu-dresden.de/MRT16/papers/MRT16_paper_1.pdf

Cicchetti, A., Ciccozzi, F., & Pierantonio, A. (2019, December). Multi-view approaches for software and system modelling: a systematic literature review. *Software and Systems Modeling*, *18*(6), 3207–3233. doi: 10.1007/s10270-018-00713-w

Debreceni, C., Horvath, A., Hegedüs, A., Ujhelyi, Z., Rath, I., & Varro, D. (2014, July). Query-driven incremental synchronization of view models. In *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling* (pp. 31–38). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2631675.2631677

Feichtinger, K., Meixner, K., Rinker, F., Koren, I., Eichelberger, H., Heinemann, T., ... Schmid, K. (2022, September). Industry Voices on Software Engineering Challenges in Cyber-Physical Production Systems Engineering. In

2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA) (pp. 1–8). doi: 10.1109/ETFA52439.2022.9921568

Goldschmidt, T., Becker, S., & Burger, E. (2012, March). Towards a Tool-Oriented Taxonomy of View-Based Modelling. In E. J. Sinz & A. Schürr (Eds.), *Proceedings of the Modellierung 2012* (Vol. P-201, pp. 59–74). Bonn, Germany: Gesellschaft für Informatik (GI). Retrieved from https://dl.gi.de/handle/20.500.12116/18148

Golra, F. R., Beugnard, A., Dagnat, F., Guerin, S., & Guychard, C. (2016, March). Addressing modularity for heterogeneous multi-model systems using model federation. In *Companion Proceedings of the 15th International Conference on Modularity* (pp. 206–211). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2892664.2892701

Grier, R. A. (2015, September). How High is High? A Meta-Analysis of NASA-TLX Global Workload Scores. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, *59*(1), 1727–1731. doi: 10.1177/1541931215591373

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

IEC. (2023). *IEC 62056 electricity metering data exchange – The DLMS/COSEM suite.*

IEC. (2024). *IEC 61968 application integration at electric utilities – System interfaces for distribution management.*

IEC. (2025a). *IEC 61850 communication networks and systems for power utility automation.*

IEC. (2025b). *IEC 61970 energy management system application program interface (EMS-API).*

ISO. (2011). *ISO/IEC/IEEE 42010:2011(E) systems and software engineering – architecture description.*

ISO. (2018). *ISO 9241-11:2018 ergonomics of human-system interaction – Usability: Definitions and concepts.*

Kolovos, D., & de la Vega, A. (2023, August). Flexmi: a generic and modular textual syntax for domain-specific modelling. *Software and Systems Modeling*, *22*(4), 1197–1215. doi: 10.1007/s10270-022-01064-3

König, L., Ritz, D., & Burger, E. (2025). Transforming relational model queries to triple graph grammars. In *2025 acm/ieee 28th international conference on model driven engineering languages and systems companion (models-c)* (p. 752-761). doi: 10.1109/MODELS-C68889.2025.00098

König, L., Stickling, T., & Burger, E. (2025). Towards dynamic views on heterogeneous models – the NeoJoin view definition language. In *Joint Proceedings of the STAF 2025 Workshops.* (ISSN: 1613-0073) doi: 10.5445/IR/1000188774

König, L., Stickling, T., & Burger, E. (2026, March). *Replication Package for "Language Design for the View Definition Language NeoJoin".* Zenodo. doi: 10.5281/zenodo.18985073

Langlois, B., Exertier, D., & Zendagui, B. (2014, October). Development of Modelling Frameworks and Viewpoints with Kitalpha. In *Proceedings of the 14th Workshop on Domain-Specific Modeling* (pp. 19–22). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2688447.2688451

Mittelbach, V. (2015). *Model-driven Consistency Preservation in Cyber-Physical Systems* (Unpublished master's thesis). Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany.

Stickling, T. (2025). *Designing a Model Transformation Language for Projective Views* (Unpublished master's thesis). Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany.

About the authors

Lars König is a PhD researcher at Karlsruhe Institute of Technology (Germany). You can contact the author at lars.koenig@kit.edu.

Tobias Stickling has a master's degree from Karlsruhe Institute of Technology (Germany). You can contact the author at tobias.stickling@kit.edu.

Alexander Kocher is a PhD researcher at Karlsruhe Institute of Technology (Germany). You can contact the author at alexander.kocher@kit.edu.

Hüseyin Kemâl Çakmak is Head of Research Group at Karlsruhe Institute of Technology (Germany). His research interests include sector-coupled energy system modeling and analysis coupled with high-performance and distributed computing. You can contact the author at hueseyin.cakmak@kit.edu.

Erik Burger is a post-doctoral researcher at Karlsruhe Institute of Technology (Germany). His research areas include model-driven and view-based development. You can contact the author at burger@kit.edu.

Veit Hagenmeyer is a full professor at Karlsruhe Institute of Technology (Germany). His research interests include modeling, optimization, and control of sector-integrated energy systems, machine-learning-based forecasting of uncertain demand and production in energy systems mainly driven by renewables, and integrated cybersecurity of such systems. You can contact the author at veit.hagenmeyer@kit.edu.

Anne Koziolek is a full professor at Karlsruhe Institute of Technology (Germany). Her research in software engineering is concerned with the early phases and activities in the development of software, or more general software-intensive systems. You can contact the author at koziolek@kit.edu.

Ralf Reussner is a full professor at Karlsruhe Institute of Technology (Germany). He is spokesperson of the Collaborative Research Centre *Consistency in the View-based Development of Cyber-physical Systems*. His research are the interplay of software architecture and predictable software quality as well as on view-based design methods for software-intensive technical systems. You can contact the author at reussner@kit.edu.