Model Transformation for Asset Administration Shells

Torben Miny, Michael Thies and Ulrich Epple
Chair of Process Control
RWTH Aachen University
Aachen, Germany
{t.miny, m.thies, epple}@plt.rwth-aachen.de

Christian Diedrich
Institute for Automation Engineering (IFAT)
Otto-von-Guericke Universitat Magdeburg
Magdeburg, Germany
christian.diedrich@ovgu.de

***Abstract*—In the scope of Industry 4.0 (I40), one goal is the standardized access to asset information and asset services using standardized submodels (submodel templates) in the Asset Administration Shell. Since submodel templates are modeled by different groups of people, the same asset information will be contained in several submodel templates. For automatic generation of new submodels based on existing information from other submodels, model transformation can be a solution. Therefore, in this contribution, we present a guideline on how to develop a new model transformation language for a given use case and apply this guideline to the concrete use case (model transformation for Asset Administration Shells). As a result, we define of the abstract syntax of a customized model transformation language called AASMTL.**

***Index Terms*—Modell transformation, Industry 4.0, Asset Administration Shell**

I. Introduction

In the scope of Industry 4.0 (I40), one goal is the standardized access to asset information and asset services for all stakeholders. Therefore, the Asset Administration Shell (AAS) as a digital representation of an asset was introduced [1]. It consist of different information models, so-called submodels, which represent different aspects of the asset, e.g. identification information, construction information or asset service information like drilling. A submodel should be constructed for interoperability and should therefore be formulated in a machine-readable form. This implies that the semantics of the provided information must be defined explicitly. In line with properties that are standardized in property libraries like eCl@ss[1] or IEC61360-CDD[2], submodels should be standardized with the help of so-called submodel templates.

Since submodel templates are modeled by different groups of people, the same asset information will be contained in several submodel templates. However, due to the flexibility in the metamodel of the AAS, it may be modeled differently. The modeling of minimum and maximum values shall serve as an example here: In one template these can be modeled with separate "Property" elements and in another template with a "Range" element. However, they both contain the same information.

[1]https://www.eclass.eu/
[2]https://cdd.iec.ch/

At runtime, submodel instances of these templates have to be created and the information has to be filled in. Therefore, tools have to be developed to support users with these tasks. In [2] we propose the use of model transformation to ensure an easier creation and filling process. We gave a short classification and showed what kind of transformations in the area of submodels are necessary. However, we did not define the syntactic details of the model transformation language in detail.

For this reason, the contributions of this paper are a guideline on how to develop a new model transformation language for a given use case and the application of this guideline on the transformation of Asset Administration Shell submodels, which leads to the definition of an abstract syntax of a customized model transformation language.

The rest of the paper is structured as follow: A summary of the state of the art in Sec. II, that introduces the concepts of the AAS and model transformation as well as the object constraint language. In Sec. III, a guideline for the development of MTLs is given. This guideline is applied to a MTL for AAS submodels in Sec. IV. The developed DSTL, called AASMTL (Asset Administration Shell Model Transformation Language), is described in Sec. V. Finally, Sec. VI summarizes the contribution of this paper and discusses some still open problems and steps for future research activities.

II. State of the Art

A. Asset Administration Shell

The Asset Administration Shell (AAS) has been introduced as a digital representation of assets in the context of Industry 4.0 systems. It is meant to act as a single uniform interface to information and services of the asset [1]. An AAS may be provided as a file on some exchangeable memory, as part of a network interface using HTTP or OPC UA protocols, or as an interactive service communicating via the "industry 4.0 language" protocol. The description of its implementation concept, use cases, data model, and infrastructure is the subject of several working groups. Currently, the concepts are transferred to international standardization.

Each AAS is composed of individual parts, designated as "submodels", to hold information pertaining to different aspects or functional domains of the asset. Information within the submodels is structured in objects according to a metamodel, defined in [3]. Different object classes allow to provide single-valued “Properties”, ranges value “Ranges”, or to full binary files as “Blob”. Additionally, object types are defined for referencing external resources, expressing relationships between other objects, and representing callable operations. Each object is identified by a string identifier (“idShort”) within the submodel. They may be structured hierarchically using “SubmodelElementCollections’‘.

The contained information may be semantically annotated by referencing an external or custom concept description, to allow an automatic interpretation of the information, even if it does not follow a fixed schema. Human-readable descriptions in multiple languages may be provided for each object, as well.

There are efforts to standardize the elements of submodels for common aspects. For this purpose, submodel templates are created. These are special submodels which are not associated with an AAS. Still, they contain the objects required for the designated purpose to define the structure, object identifiers, object types and semantics of corresponding submodel instances. Instances of a submodel template are created by copying the template including its contained objects and filling them with specific values. By including a reference to the template’s global identifier, the semantics of the submodel instance can be defined precisely.

B. Model Transformation

Model transformation is the process which converts source models into target models. For this purpose, abstract rules describe how elements of the source models should be transformed into the target models. A so-called transformation system can execute them on a specific set of source models.

In [4]–[6] different model transformations were analyzed and the distinguishing features were determined. These features can be categorized into general features, features for source and target (meta)models, rule features and features of rule usage. The different features are mentioned below with their possible characteristics in brackets. For more details, we recommend the mentioned publications.

In the first category, a model transformation can be distinguished by

- direction of transformation (unidirectional or bidirectional) and
- incrementality (target incrementality, source incrementality or preservation of user edits in the target).

In the second category, a model transformation can be distinguished by

- type of the source and target model (model-to-model or model-to-text),
- number of source and target models (1:1, 1:M, M:1 or M:N),
- abstraction level of the source and target models (horizontal or vertical),
- relationship between the source and target models (in-place or out-place) and
- type of the meta models (endogenous or exogenous).

In the third category, features of rules can be distinguished according to some of the mentioned features above, like direction of transformation or type of the source and target model. Additionally, rules can be distinguished by

- used language paradigms (declarative, imperative or hybrid),
- used structures (variables, patterns, logic, untyped, syntactically typed, semantical typed),
- syntactical separation (e.g. LHS and RHS),
- execution conditions (e.g. when-conditions) and
- parameterization (e.g. with flags or data types).

In the last category, model transformations can be distinguished by

- scheduling of rule execution (explicit scheduling, internal explicit scheduling and internal scheduling),
- rule selection (explicit conditions, non deterministic selection, conflict solution mechanism, interactive),
- rule iteration (looping, recursion or fixed point iteration),
- phasing (e.g. creation phase and setting phase),
- modularization (supported or not supported),
- reuse of rules (e.g. inheritance, composition) and
- structuring (source oriented, target oriented or independent).

Additionally, it can be distinguished if the MTL is for general purposes or domain specific. General purpose model transformation languages (GPTLs) are usable for different kinds of source and target metamodels (mostly UML-based metamodels). To achieve this, the language elements are kept on a general level and rely on model semantics only to a small extent.

In contrast, domain specific model transformation languages (DSTLs) are based on a domain specific language (DSL) of the target domain. Therefore, a DSTL is easier to learn for a domain expert because most of the language elements are already known. However, a DSTL is tied to a concrete domain-specific modeling language and all software tooling has to be implemented for each DSTL.

C. Object Constraint Language

The Object Constraint Language (OCL) [7], defined by the Object Management Group, is an expression language to describe constraints on object-oriented models in conjunction with the Unified Modeling Language (UML). With OCL expressions it is possible to specify queries, variants or conditions for operations. The expressions are evaluated instantaneously and without any side effects.

While it was originally only used with UML, OCL is now a key component of model-driven engineering [8]. Many model transformation languages are using OCL expressions for queries.

The specification [7] defines the abstract syntax, the semantics and a concrete textual syntax of the expression language. In Fig. 1 an abstract view of the OCL metamodel is given.

It defines eight types of expressions:

- *CallExp:* An expression that represents the evaluation of an object feature (operation, property) including predefined iterators for collections.
- *LiteralExp:* An expression to represent a literal value.
- *IfExp:* An expression which resolves to one of two alternative expressions depending on the evaluated value of a condition.
- *VariableExp:* An expression that references a bound variable.
- *TypeExp:* An expression used to refer to an existing type.
- *MessageExp:* An expression that evaluates a collection of OclMessage values.
- *StateExp:* An expression used to refer to a state of a class within an expression.
- *LetExp:* An expression that defines a new variable with an initial value.

The specification also defines "BasicOCL" as a subset of the complete OCL. BasicOCL includes all elements of the complete OCL, except for MessageExp and StateExp and some associated minor features.

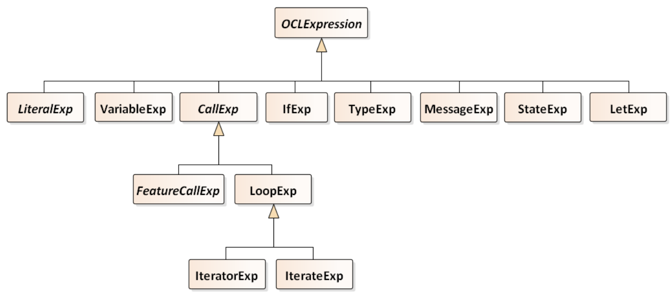

Fig. 1. OCL metamodel based on [7]

III. Developing a model transformation language for a given use case

This section describes the steps towards developing a MTL in a generic way. It may serve as a guideline for a structured approach of finding a MTL for a given transformation task in practice.

A. Classifying the model transformation to be performed & the source/target metamodels

To select or develop a suitable language for a given model transformation problem, the model transformation's properties need to be known. Thus, the problem should be classified according to the distinguishing features of model transformations presented above. In contrast to rule features and features of rule usage which depend on the chosen MTL, the general features and source and target (meta)model features can already be determined from the model transformation problem. Additionally, the meta-meta-model of the source and target models needs to be considered, since the model transformation language must provide support for the types of model elements used in the source and target metamodels. Most existing model transformation languages deal with UML-based metamodels.

B. Assembling requirements for the model transformation language metamodel

In addition to the required features for the given model transformation problem, requirements for the formulation of transformation rules may be given. These requirements need to be fulfilled by the abstract syntax (i.e. the set of available syntax elements) of the MTL. Depending on the expected structure of transformation rules, different styles of rule syntax and rule execution semantics may be preferred: syntactically separated vs. combined rule components, explicit rule execution conditions vs. automatic selection, usage of rule inheritance or helper functions, etc.

C. Designing a customized model transformation language

As a first step, an evaluation of existing MTLs should be carried out with regard to their suitability or adaptability to the given requirements. If there is no fitting language, designing a new MTL will be necessary. However, tools for semi-automatically generating them are available and should be considered to reduce the effort. Only in case this does not lead to a good result, a completely new design should be chosen.

1) Evaluate suitability of existing model transformation languages or possibility of adaption: Instead of implementing a new MTL from scratch, it might be sensible to use an existing model transformation framework and adapt the language metamodel and tools for to the given requirements. Particularly, it might be possible to use a GPTL and tailor it to the relevant domain. In [9] such an adaption of a GPTL is shown. Therefore, existing MTLs should be evaluated in terms of their eligibility for the requirements as defined above. On a very basic level, the supported model transformation characteristics, metamodels and language paradigms should match in order that a framework can be deemed suitable. In contrast, minor modifications to the syntax of the language may be possible; especially by introducing additional language elements and restricting the available syntax elements to a smaller set of required functionality.

2) Evaluating possibility of generation of a domain specific model transformation language: A different approach to customizing a MTL is to use a framework for generating DSTLs for specific use cases. For some classes of transformation problems, it may be possible to generate a MTL using such a framework. Typically, this approach requires the source and target metamodels to be given as a formal language definition, or the features of the transformation language itself must be specified in a formal way. The semi-automatic generation of a DSTL from an abstract definition of its features is described in [10]. In [11], a framework for deriving DSTLs for formally defined textual domain-specific languages is presented.

3) Designing a custom-built MTL: However, even frameworks for generating custom DSTLs make assumptions about the model transformations and the abstract syntax of the transformation rules, which might not fit the given use case. Moreover, the provided software tools may not fit the existing tools and frameworks in the given application scenario, so developing a new tool set (MTL parser, transformation engine, etc.) is required anyways. This might be significantly easier for a custom-built MTL, containing only the required syntax elements.

In this case, it is a reasonable choice to design and implement a custom-built MTL. In [12] and [13] example general procedures for the creation of MTLs are given. Both approaches start the development from scratch. Summarized and compared with other approaches, the process includes three major steps:

- designing an abstract syntax and static semantics (their meaning for the transformation and invariants especially type invariants)
- designing a concrete syntax
- implementing a parser, checker and interpreter/transformation engine for the language.

Despite building the language from scratch, one might reuse parts of existing transformation, programming or expression languages. Especially when designing the abstract syntax, it will be useful to embed comprehensive specifications of an expression and—depending on the intended paradigms—statement syntax, to avoid uncovered corner cases. [14] gives an overview over typical elements of MTLs.

The step of implementing the required checking and execution tools can be facilitated by generators. Such a generator for creating code templates from a formally defined MTL, which are later used to build executable model-transformers from transformation definitions, is presented in [15].

IV. Developing a model transformation language for asset administration shells

In this section, the abstract procedure, presented in Sec. III is applied to the model transformation of submodels of the AAS.

A. Classification

According to the distinguishing characteristics of model transformations from Sec. II-B, the required model transformation for generating and filling submodels from existing submodels can be generally classified as follows:

- unidirectional transformation, and
- non-incremental transformation[3].

In terms of source and target models, the transformation can be classified as

- Model-to-Model transformation (no textual representation involved),
- M:1 transformation (any number of source submodels are transformed into a single target model),
- horizontal transformation (same abstraction level of source and target models),
- out-place transformation (no modification of source model),
- endogenous transformation (same metamodel of source and target model).

[3]incremental transformation will be required for corner-cases like updating existing submodels and online transformation

B. Requirements

In addition to the functional classification of the model transformation to be performed, we can formulate some requirements for the design of the language syntax based on how it will be used in practice (cf. sec. III-B).

The general use case of the transformation language will be the description of automatic generation of AAS submodels based on the existing information in other submodels [2]. The transformation definitions will primarily be created by data modelers in companies to support their individual workflows. Thus, the transformation language syntax should be simple to use and close to the metamodel of source and target models, i.e. the metamodel of the AAS. This allows to refine the transformation language into a true DSTL, which is build upon a concrete syntax of the AAS metamodel, as soon as such a syntax is specified.

Transformation definitions will be created to generate a single well-formed submodel from collected information from existing submodels. In contrast to most transformation problems, the major structure of the target model is defined by the transformation definition instead of being derived from the source models by applying rules to each of their objects. "For-each" structures iterating the source models' objects will only be required occasionally.

Therefore, the MTL should be "target-oriented": It should allow the specification of the target model's structure in a comprehensible, preferably declarative template-like style [16]. Conversely, no explicit definition of single rules is required. To put it in the terminology of existing model transformation frameworks: Each transformation definition requires only a single rule defining the full structure of the target model. Thus, there is also no need for explicit execution conditions, rule iteration, rule selection, scheduling or phasing. A syntactical separation is also not required.

Still, a powerful expression syntax is required to allow manipulation and combination of the source models' information. Additionally, there should be means of modularization and reuse of expressions with parameterization.

The AAS is meant to provide interoperability between many different platforms and frameworks in the IT and automation technology. To facilitate implementing the required software tools in these different "ecosystems", the MTL should be kept as simple as possible overall. It should also be possible to build online transformations for calculating the target submodel on the fly.

C. Designing a customized model transformation language

As described in Sec. III-C, we first evaluated existing model transformation languages with the objective to directly use or to adapt these. Because the metamodel of the AAS is quite new (first version released in November 2018), only one model transformation for this purpose is published [17]. Unfortunately, this approach cannot be used, because it only enables a simple mapping of elements, which does not fit the requirements defined in the previous sections. Therefore, typical GPTLs like the Query/Views/Transformation Language (QVT), the Atlas

Transformation Language (ATL) or the Epsilon Transformation Language (ETL) were evaluated. However, they all do not fit the requirements: mostly because they have to many not needed features and constraints like the definition of *rules* that have to be implemented into the existing tools for online transformation.

Therefore, in a second step, frameworks for generating a DSTL were analyzed. Baar and Whittle developed in [18] a procedure to generate the DSTL metamodel from the metamodel of the domain specific modeling language. However, [18] only describes how to create the syntax and not how the rules can be executed or what an implementation should look like. A finished implementation is not provided. Additionally, the frameworks introduced in Sec. III-C2 [10], [11] have been checked for their suitability. It turned out not to be given, primarily due to the lack of a formal modeling language specification and the implementation overhead for not required syntax elements when realizing the MTL in different programming eco systems.

In summary, all frameworks for generating custom DSTLs are not usable. Either the frameworks are not fully developed, cover only a few steps, make assumptions, which do not fit the given use case—e.g. need a concrete syntax of the modeling language—, or do not fit into the existing tools in the domain of AAS. As a consequence, a new custom-built MTL has to be developed for which tool sets (MTL parser, transformation engine, etc.) can be implemented for each AAS software framework, so that online transformations are possible. The new-defined custom-built MTL, called AASMTL, is described in Sec. V.

V. Model Transformation Language AASMTL

As described in the previous section, a new custom-built MTL has to be developed. Therefore, an abstract syntax including semantics and a concrete syntax must be designed and a tool set must be implemented. In this paper we introduce the designed abstract syntax and language semantics.

Based on the classification and requirements, the needed rule functionalities for a model transformation (based on [2]) were defined:

- searching model elements in different source submodels, mostly based on the *idShort* parameter given in the corresponding submodel templates,
- set attributes of model elements in the target submodel and
- create model elements in the target submodel which are optional in the corresponding submodel template.

These functionalities require expression language elements for the definition of literals (e.g. string, boolean or collection), the access of attributes and macros, case differentiation, type checking, loops, as well as definition and usage of variables and macros. All of these language elements already exist in the expression language OCL which is part of the well established Unified Modeling Language (UML). Since the meta model of AAS is also based on UML and OCL is a well-known language, we decided to use it as basis of our new custom-build MTL. For the given requirements, it is sufficient to use the BasicOCL instead of the full OCL language (see sec. II-C).

A. Additional Syntax Elements

All of the expressions from BasicOCL can be reused for our use case. Additionally, the following new expression classes are added: *TransformationDefinition*, *ObjectLiteralExp* with *AttributeBinding*, *Macro* and *MacroCallExp*. The new language meta model, containing these elements, is shown in Fig. 2, with the new elements highlighted.

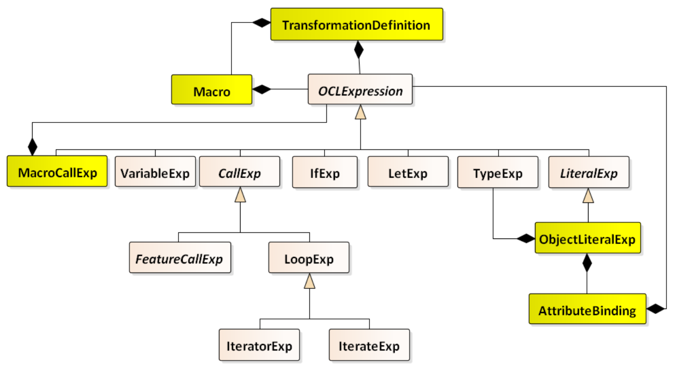

Fig. 2. Extension to OCL abstract syntax

A more detailed view of the additional elements, including their associations, is given in Fig. 3.

A *TransformationDefinition* element is the entry point for each transformation definition. It includes metadata, such as *sourceTemplates* and *targetTemplate*, which define the submodel templates to which the source submodels must conform and the template of the target submodel. The transformation itself is given by the *value* property, which associates a single OCLExpression, which evaluates to the target submodel. In addition, a TransformationDefinition can contain any number of *Macro* elements to define reusable, parameterizable macro expressions.

ObjectLiteralExp syntax elements allow to specify new instances of a model class and their attributes' values in the transformation definition. They comprise a TypeExp via the *objectType* association to define the model class of the object to be created and any number of *AttributeBinding* elements, which allow defining one attribute value each. The ObjectLiteralExp evaluates to the new object instance and can be used in any place where a class-typed expression is expected.

Each *AttributeBinding* element is part of an ObjectLiteralExp and allows to define an attribute of the instantiated object. It includes a reference to the class property specifying the attribute (not shown in Fig. 3) and can contain an expression as *initExpression* to define the value of the attribute. The initExpression's type must match the property's type.

A *Macro* can be included with a TransformationDefinition or be part of a library for general use (not shown in the Fig. 3). It defines an arbitrary OCLExpression as *body* and may define any number of variables as *parameter*, which are assigned a value when the macro is called.

For calling a macro, the *MacroCallExp* is introduced. It is an OCLExpression subclass that resolves to the body expression of the *referredMacro*. It may contain multiple of OCLExpressions as *parameterValue* to be assigned to the Macro's parameter variables when evaluating the expression.

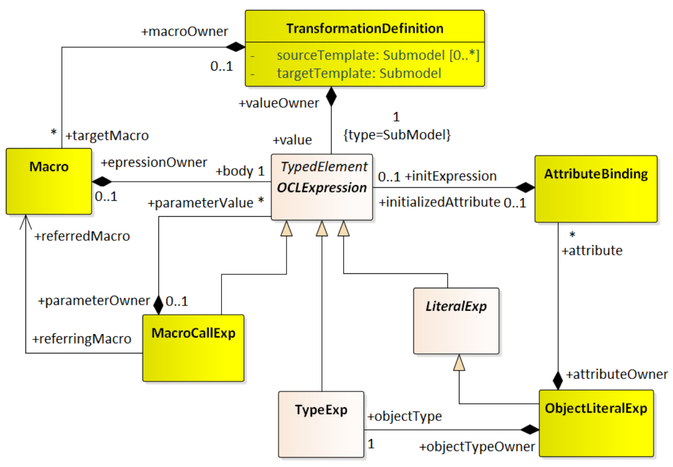

Fig. 3. Extension to OCL abstract syntax

B. Well-formedness Rules

For a full language specification of the MTL which allows checking the validity and type-consistency of transformation definitions, rules for the well-formedness of syntax elements need to be specified in addition to the class model given above. These can be defined in OCL itself.

We use the well-formedness rules defined by the OCL specification [7] as a basis and extend it with rules about our additional elements. As an example, the type of an *ObjectLiteralExp* element can be defined as an invariant:

```
context ObjectLiteralExp
inv: objectType.referredType.oclIsKindOf(Class)
inv: type = objectType.referredType
```

Due to its length, we omit the full set of rules here.

VI. Conclusion and Outlook

As information modeling for submodel template is done by expert groups of different domains, the same asset information will be contained in several submodel templates but maybe modeled differently. To transform information between the submodels respectively, the concept of model transformation can be applied. Therefore, in this paper, we first described a guideline for developing a model transformation language for a given use case including problem classification, requirements definition and design of the language. After that, we performed these steps to develop a model transformation language for Asset Administration Shells.

Following the guideline, a research on existing MTL and generating tools was carried out with the result that a new domain specific model transformation language (DSTL) has to be developed. Thus, we designed a custom-build MTL for this use case, called AASMTL, which targets automatic generation of new submodel instances based on already available information of other submodel instances. In order to not develop it from scratch, we took the abstract syntax (meta model) and the static semantics of the well-known OCL language as a basis and slightly extended it with additionally required elements.

In future research we are planning to design two concrete syntax specifications, one in textual form and one in a graphical form. After that, we want to build up the tool set using PyI40AAS, our Python SDK for Asset Administration Shells. Additionally, we want to show how online transformations can be realized in our SDK based on the defined abstract syntax and give a guideline for other SDKs.

References

[1] DIN SPEC 91345: Reference Architecture Model Industrie 4.0 (RAMI4.0), Standard, DIN - German Institute for Standardization, Berlin, DE, 2016.
[2] T. Miny, M. Thies, U. Epple, and S. Wein, "Concept for the automated generation of asset administration shell submodels using domain-specific transformation language elements," *Tagungsband Automation*, 2020.
[3] Plattform Industrie 4.0., "Details of the Asset Administration Shell - Part 1 - The exchange of informationen between partners in the value chain of Industrie 4.0 (Version 2.0)," specification, 2019.
[4] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.
[5] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electronic notes in theoretical computer science*, vol. 152, pp. 125–142, 2006.
[6] A. Metzger, "A systematic look at model transformations," in *Model-driven Software Development*, pp. 19–33, Springer, 2005.
[7] Object Management Group, "Object Constraint Language V2.4," specification, 2014.
[8] J. Cabot and M. Gogolla, "Object constraint language (ocl): a definitive guide," in *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pp. 58–90, Springer, 2012.
[9] A. Petter, *Modell-zu-Modell-Transformation von Modellen von Benutzerschnittstellen*. PhD thesis, TU-Prints, 2012.
[10] J. S. Cuadrado, E. Guerra, and J. de Lara, "Towards the systematic construction of domain-specific transformation languages," in *European Conference on Modelling Foundations and Applications*, pp. 196–212, Springer, 2014.
[11] K. Hölldobler, *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen*. PhD thesis, Shaker, 2018.
[12] J. I. Irazábal, C. Pons, and C. Neil, "Model transformation as a mechanism for the implementation of domain specific transformation languages.," *Electronic Journal of SADIO (EJS)*, vol. 9, pp. 49–66, 2010.
[13] E. Kalnina, A. Kalnins, A. Sostaks, E. Celms, and J. Iraids, "Tree based domain-specific mapping languages," in *International Conference on Current Trends in Theory and Practice of Computer Science*, pp. 492–504, Springer, 2012.
[14] E. Syriani, J. Gray, and H. Vangheluwe, "Modeling a model transformation language," in *Domain Engineering*, pp. 211–237, Springer, 2013.
[15] T. Reiter, E. Kapsammer, W. Retschitzegger, W. Schwinger, and M. Stumptner, "A generator framework for domain-specific model transformation languages," in *ICEIS (3)*, pp. 27–35, 2006.
[16] J. S. Cuadrado, "Towards a family of model transformation languages," in *Theory and Practice of Model Transformations*, (Berlin, Heidelberg), pp. 176–191, Springer, 2012.
[17] M. Platenius-Mohr, S. Malakuti, S. Grüner, and T. Goldschmidt, "Interoperable digital twins in iiot systems by transformation of information models: A case study with asset administration shell," in *Proceedings of the 9th International Conference on the Internet of Things*, pp. 1–8, 2019.
[18] T. Baar and J. Whittle, "On the usage of concrete syntax in model transformation rules," in *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pp. 84–97, Springer, 2006.