[KMR24] M. Konersmann, J. Michael, B. Rumpe:
Towards Reference Models with Conformance Relations for Structure.
In: Informing Possible Future Worlds. Essays in Honour of Ulrich Frank,
pp. 247-269, Logos Verlag Berlin, Mar. 2024.

Chapter 14

Towards Reference Models with Conformance Relations for Structure

Marco Konersmann, Judith Michael and Bernhard Rumpe

The term ‘reference model’ is broadly used in publications and discussions. To our understanding, the role of reference models and their use in modeling is not standardized and understood mostly informally in many communities. We posit that a more concise and formal understanding of the concept of a reference model is needed, to help make tool-assisted use of reference models and their relation to concrete models. In this contribution, we present our understanding of reference models and conformance relations, which specify which models are valid concretizations of a reference model. We argue why no general conformance relation can exist for all modeling languages. On the example of data models using UML class diagrams, we discuss potential recurring requirements of conformance relations, representations for mappings between concrete models and reference models, and the challenge of given code implementations to be attached to reference models and how to transfer this code to a concretization.

14.1 Introduction

In existing publications, there seems to be no agreement on what the term ‘reference model’ explicitly means. Authors define ‘technical reference models’ (Joshi and Michel 2008), ‘business reference models’ (Eom and Fountain 2013; Frank and Strecker 2007; Frank 2007), ‘business process reference models’ (Scheer 1994; Fettke et al. 2006; Reinhartz-Berger et al. 2010), the ‘ISO/OSI 7 layer communication reference model’ (American National Standards Institute 1981), ‘enterprise architecture reference models’ (Zimmermann et al. 2015), ‘performance reference models’ (Forme et al. 2007), ‘reference models for deep learning frameworks’ (Atouani et al. 2021), ‘reference models for digital shadows’ (Michael et al. 2023), and various others. Occasionally meta-models (Bucaioni et al. 2022; Michael et al. 2023; Mayr et al. 2018), meta-models and OCL (van der Aalst and Kumar 2001), or component-connector architecture models (Dalibor et al. 2022) are used and treated as reference models.

Gray and Rumpe raise the question of whether it is possible to leverage the notion of a reference model. In particular: ‘How can the notion of a reference model, explicitly denoted in a given modeling language, be formalized together with a precisely defined and tool-assisted notion of a conformance relation that would enable concrete realizations of the reference model?’ (Gray and Rumpe 2021). In this contribution, we present a formalization of reference models for structure data and their conformance relations along with operations

upon them for tool-assisted conformance checking and for detecting mappings between concrete models and their reference models.

We consider a reference model to be a model, according to our understanding of the general definition of 'model' (which is based on Stachowiak 1973): (1) there should be an original, real system/entity that is being modeled, (2) the model should be an abstraction of the original, and (3) the model has something to do with the original system. Based on this definition, we draw the following consequences for reference models, which will be formally defined in Definition 1:

- A model can be used as a substitution for the real system: The *Principle of Substitution* then applies to reference models as well. It states that a useful subset of questions about the original can be answered by querying the model. This seems to fit the definition of a reference model because such a model should be a reference for many (similar) systems, even though there is typically a concretization development step between the definition of the reference model and the finalization of the system.
- A model is usually defined *explicitly in an appropriate modeling language.* For example, if we design a reference model for some data standard, we can use class diagrams as a language for its expression.
- It is often feasible to *use a similar or the same language for both reference and concrete models*, even though sometimes specific language constructs may only be used in the reference model, respectively, the concrete model. This may be a restriction for the general notion of the reference model because, in principle, it might be possible to actually use different modeling languages for reference models and their concrete realizations, but we ignore this possibility for now.
- It is often *not easy to distinguish a reference model from the concrete realizations* that are possible. It may be that the same model may serve as a concrete model or as a reference model in different contexts.
- Given a concrete model that is used for realization within a concrete system, it must be clearly decidable whether and how far the *concrete model actually conforms to the reference model.* A somewhat unclear conformance relation is not necessarily a problem when the reference model serves education, communication, or an otherwise informal purpose. However, when there is tool assistance available or needed, the question of whether a concrete model conforms to its reference becomes more critical. When the reference model is defined from a legal or standard context, the conformance question also becomes relevant for certification.

While the word 'reference model' is used quite frequently, to our understanding, many domains and their communities have different and mostly informal understandings of the significance of a reference model and, consequently, different approaches to using reference models in a particular domain (see also Arora et al. 2022). We posit that a more concise and formal understanding of the concept of a reference model is needed that improves the tool-assisted relationship between a general 'reference model' and the set of concrete realizations that conform to that reference model.

The remainder of this contribution is structured as follows: The next section introduces a running example of structure reference models. We present and discuss our understanding of structure reference models and their conformance relations in Section 14.3 and their consequences on operations upon models and reference models, including conformance

checking in Section 14.4. Alternatives on representing mappings between reference models and concrete models are shown in Section 14.5. In Section 14.6, we discuss the challenge of given code implementations to be attached to reference models and how to transfer this code to a concretization. We show related work in Section 14.7 before we conclude in Section 14.8.

14.2 Example Reference Model for Structure Data Models

We illustrate a possible conformance relation for class diagrams as known from the UML (Rumpe 2017) using an example. A class diagram includes concepts for classes, interfaces, enumerations, attributes, methods, constants, and binary associations and compositions of various forms. In the following listing, we show a class diagram in textual form, i. e., a model of MontiCore's CD4A language[1] (Rumpe 2017), which we regard as a reference model:

```
classdiagram AccessRights {
        class User;
        class Role;
        class Right;
        class Create extends Right;
        class Update extends Right;
        association userroles User -> Role [*];
        association Role -> Right [*];
}
```

The next listing shows a concrete model example where we want to make statements about conformance.

```
classdiagram Flensburg {
        class Person;
        class Admin;
        class CarOwner;
        association Person -> Admin [0..1];
        association myCars   Person -> CarOwner [*];
        class Permission;
        class CreateCar extends Permission;
        class UpdateCar extends Permission;
        association allowed [1] CarOwner <-> Permission [1..3];
}
```

The concrete model can be considered conforming with the reference model if we consider a mapping of concrete model elements to reference model elements as informally shown in Listing 14.1. We call this mapping an *incarnation mapping*, because it describes which concrete model elements (on the left side) *incarnate* which reference model element on the right side. Here, we use the term 'incarnation' in contrast to instantiation, to highlight a specific difference in its characteristics: incarnation relates models elements of the same kind, here classes with classes and associations with associations, while instantiation normally relates different kinds, e. g., objects with classes.

1 https://github.com/MontiCore/cd4analysis

Listing 14.1: A mapping that renders the concrete model conformant to the reference model (informal)

```
// Mapping of classes
Person                          ===>  User
Admin, CarOwner                 ===>  Role
Permission                      ===>  Right
CreateCar                       ===>  Create
UpdateCar                       ===>  Update
// Mapping of associations
(Person -> Admin), myCars       ===>  userroles
allowed                         ===>  (Role -> Right)
```

To decide whether the concrete model is a valid concretization of the reference model, we need rules that describe a valid *conformance relation.* In our example, these conformance rules state that each reference model element must be incarnated at least once, and the cardinality of reference associations may be restricted in the concrete model, but not relaxed (we will discuss further rules in Section 14.3.4). The information provided by the reference model, the concrete model, the incarnation mapping, and the conformance relation can now be used to validate whether the concrete model is a valid concretization of the reference model. In our example, the mapping can be considered valid with respect to the conformance rules, denoting it a valid concretization.

14.3 Reference Model, Conformance, and Incarnation

We introduce a formal definition for structure reference models, their conformance relations, and their constructive realization, namely the incarnation mapping. For that purpose, we also introduce a set of conformance rules.

14.3.1 Reference Model

Definition 1 (Reference Model). *A reference model in systems, enterprise, and software engineering is*

- *a model in a given language,*
- *used to describe a set of domain concepts and their domain-specific relations, and*
- *there exists a conformance relation, that precisely identifies the set of conforming models within the language the reference model belongs to.*

As a consequence of Definition 1, each reference model is also regarded as normal (concrete) model. The main difference between a reference model and a concrete model is, therefore, not a syntactical issue but based on the purpose of the model—or to be more precise, on the multiple purposes in the various activities of a development process. We identify these main purposes for reference models, where not all purposes must be addressed with each model:

P1 The introduction of *domain concepts* (which itself could be regarded as a fuzzy term). We assume that a domain concept has a name and potentially a number of properties that, together, can somehow be represented in our modeling language. These domain concepts are (and this is again somewhat informal) of general interest and, thus, typically can also be regarded as elements of a glossary or an ontology of a certain domain (Mayr and Thalheim 2020).

P2 *Domain-specific relations* connect these domain concepts. These relations are also described in the reference model.

P3 The *reference model itself* usually also has a name and, thus, allows us to talk about the overall system structure by the means of the reference model.

P4 There may be *additional information* or *methodical development advice* available that fits the domain concepts and, thus, can be manually reused in a development process.

P5 There may be a predefined *reference implementation* available that fits the domain concepts and, thus, can be automatically reused directly or mapped to the concrete model and its implementation via smart tooling.

P1 and P2 are usually the case in data models describing the application data of a domain, such as `Person` or `Account`, and their relation, like `owns`. P3 is, e. g., the case in the ISO/OSI 7 layer communication model or in various design patterns. Another example is a traffic light model with two states, `green` and `red`, and the corresponding transitions between those states as a reference model. The latter serves all three purposes (P1–3) if refinable, e. g., to traffic lights with the third state `yellow`.

Using reference models, however, becomes really interesting if P4 or even P5 can be leveraged. In adequate tooling, for example, the concept `User` of our example in the first listing may be equipped with possibilities for a user to administrate her own setting, acquire roles, or receive rights to read/write data. It depends on the conformance relation and the tooling that (a) checks conformance and (b) has smart concepts on how to transfer predefined functionality from the reference model to the concrete model, respectively, its implementation.

The purposes may arise independently from the model itself. Therefore, any model can in principle be regarded as a reference model when deemed necessary or helpful. With this respect, we define the set of reference models as a subset of the set of models (see Definition 1). The sets of reference models and concrete models of a modeling language are not disjoint but may have a large overlap. However, they don't necessarily have to be identical. There may be models that only can be used as reference models, e. g., because certain abstract concepts in the models such as abstract classes do not have concretizations yet.

14.3.2 Conformance Relation

Definition 2 (Conformance Relation). *Let* $\mathbb{M}$ *be the set of all concrete models, and* $\mathbb{R} \subseteq \mathbb{M}$ *the set of all reference models. The conformance relation*

$$C \subseteq \mathbb{M} \times \mathbb{R}$$

is a binary, reflexive, and transitive relation that describes whether the concrete model $M \in \mathbb{M}$ *is a concretization of the reference model* $R \in \mathbb{R}$.

The conformance relation (see Definition 2) is defined as a binary relation between models. It includes the reference model – describing relevant concepts and relations – and its implementation, respectively, the concrete model. This definition has some consequences for conformance relations: First of all, the conformance relation generally relates more abstract models with more concrete models. Because the reference model and concrete model belong to the same language (see the consequences in Section 14.1), we may accept

that a reference model can also be a concrete implementation, which means conformance is *reflexive*. The conformance relation can also be *transitively* applied, which can become interesting when concrete models become reference models themselves or when reference models for other reference models are created. It is, however, *not symmetric*, but it relates the more abstract to the more concrete models. It may be that it is not fully asymmetric either, but we currently see no application scenario for this. In general, different conformance relations can exist for different reference models. And finally: multiple conformance relations can exist for a reference model.

In a development project that uses a reference model, the conformance relation is potentially a development artifact on its own. Please note that we do not enforce that a reference model describes a structure, such as a data structure or a communication infrastructure. We can perfectly understand that people use Statecharts, BPMN models, Petri Nets, or activity diagrams as reference models. Furthermore, it may well be that a domain-specific language created for a specific purpose also comes along with the construct of reference models within the language.

However, we made the following observations with respect to the languages that we analyzed: The construction of reference models, and, in particular, the conformance relation needed, is highly driven by the semantics of the particular language. Thus, a general, purely syntax-based conformance relation will not be sufficient. As a consequence, conformance relations need to be defined for modeling languages individually.

Because the conformance relation is an abstract relation between models, it is necessary to refine this relation into a relation between individual elements of the concrete and the reference model. We call such a detailed relation a *incarnation mapping*.

14.3.3 Incarnation Mapping

Incarnation mappings describe which concrete model elements map to which reference model elements, as shown in the example in Listing 14.1. Therefore, an incarnation mapping looks into the concrete syntax of the models to be related and by definition is highly dependent on the modeling language. Without precisely clarifying the concept of 'model element' in this contribution, we assume that for a given model M, we can denote the set of relevant, syntactic model elements by M_e. We define incarnation mappings as follows:

Definition 3 (Incarnation Mapping). *For a given concrete model M and reference model R an incarnation mapping is a binary relation between their relevant, model elements*

$$I(M, R) \subseteq M_e \times R_e$$

such that the existence of an incarnation mapping ensures the conformance of the models, i. e., $C(M, R) \iff \exists i \in I : i(M, R)$.

The definition uses the term "relevant model elements", without precisely defining what is relevant. This is deferred to concrete applications. In our class diagram example from Section 14.2, we can identify classes and associations as relevant (even though other options would be possible).

A well-formed incarnation mapping identifies a conformance relation. To develop a concrete model that conforms to a reference model, the task is thus to establish a well-formed incarnation mapping.

The relation between concrete model elements and reference model elements is closely related to instantiation. We use the term 'incarnation' as in Definition 3 to highlight specific differences in its characteristics: (1) Incarnation relates models elements of the same kind, i. e., in our class diagram examples it relates classes with classes and associations with associations, while instantiation normally relates different kinds, e. g.,, objects with classes. (2) Incarnation is transitive and reflexive on the model elements (similar to the conformance relation on models itself), while instantiation is not. In general, multiple incarnation mappings can be possible for any non-trivial concrete and reference model. This means that the same reference structure is being applied several times in a concrete model in either completely disjoint or potentially overlapping substructures.

As an example, the next listing shows a reference model for user ids. The corresponding concrete model shown below describes a data structure where a `Person` has exactly one personal id and one tax id, which allows for two different incarnation mappings using two different ids from the concrete model.

```
classdiagram UserID {
        class User;
        class ID;
        association id User -> ID [1];
}
classdiagram TaxablePersonell {
        class Person;
        class PersonalID;
        class TaxID;
        association pId Person -> PersonalID [1];
        association tId Person -> TaxID [*];
}
```

In this example, the `Person` has multiple IDs. There are two mapping relations as shown in the first two mappings given in Listing 14.2. The first and second sets of mappings denote that the `Person` is mapped to the class `User` once in the personal ID mapping and once in the tax ID mapping. This allows for checking these sets individually. Following the conformance rules stated in the example of Section 14.2, the personal ID mapping is valid. The tax ID mapping is, however, to be regarded as invalid, because the association `tId` is relaxed in the concrete model, which contradicts a (below discussed and rather useful) rule that associations must retain or restrict their cardinalities, but not relax them.

Listing 14.2: Multiple incarnation mappings (informal)

```
// First Incarnation Mapping (Personal ID Mapping) - valid
Person      ===> User
PersonalID  ===> ID
pId         ===> id

// Second Incarnation Mapping (Tax ID Mapping) - invalid
Person      ===> User
TaxID       ===> ID
tId         ===> id // invalid
```

Incarnation mappings may be of complex forms, and it could well be that there is no generalized form for all purposes. However, we expect at least the following property

to hold: the correctness of an incarnation mapping and thus of a conformance relation between concrete and reference model can be effectively identified and checked by an algorithm. However, there are two fundamentally different relevant cases, (a) the mapping is already and explicitly given as some kind of parameter of the conformance relation, vs. (b) the mapping must be constructed. Case (b) is definitely more complex and potentially ambiguous, but would be of strong interest for developers.

14.3.4 Conformance Rules on the Class Diagram Case

We can argue that the model `Flensburg` in our running example is a conforming implementation of the reference model `AccessRights` with respect to the incarnation mapping and the two rules stated in Section 14.2. The rules stated in the example are simplified for illustrative purposes. In this section, we discuss rules that specify conformance relations for structure reference models. For clarification, we talk about *concrete classes C* and *concrete associations A* when referring to the *concrete model* and *reference classes R*, and *reference association S* when referring to the *reference model*. We found the following set of rules helpful in the case of UML class diagrams that we typically use:

R1 A concrete class C incarnates (at most) one reference class R (e. g., `Person` incarnates `User`). Thus, it is possible that additional classes exist in the concrete model.

R2 The same reference class R may be incarnated by many concrete classes (e. g., `Role` by `Admin` and `CarOwner`), but at least one must be given.

R3 A concrete association A incarnates (at most) one reference association S (e. g., `allowed` incarnates `(Role -> Rights)`). Again it is possible that additional associations exist in the concrete model.

R4 In analogy to classes, the same reference association S may be incarnated by many concrete associations (e. g., `userroles` has two independent incarnations, `(Person -> Admin) and myCars`), but at least one must be given.

R5 Subclassing relations are preserved, i. e., the concrete class $C2$ has to extend $C1$ if their reference classes $R2$ extends $R1$. Class $C2$ does not necessarily need to be a direct subclass but can be found in the subclassing hierarchy of $C1$.

R6 Reference associations are preserved. This means that given concrete classes $C1$ and $C2$, their reference classes $R1$ and $R2$, and a reference association $S : R1 \rightarrow R2$, then there must also be a concrete association $A : C1 \rightarrow C2$, such that A incarnates S.

These rules are definitely incomplete, because all practical and complete conformance relations also need to take typing and cardinalities of associations into account. Furthermore, real class diagrams also contain attributes, methods, abstract classes and interfaces, as well as a sophisticated type system, including enumerations.

In our example, all concrete classes have respective concrete associations attached except the `Admin`, which violates condition R6. We could argue that the admin doesn't need explicit permissions because, in our application, she has all permissions anyway, but it does violate the conformance rule.

As said earlier, in different mappings, it is allowed that one concrete class incarnates multiple reference classes, therefore, playing different 'roles' with respect to the reference model. As an example, consider the observer pattern as reference model (Listing 14.3) and a class diagram where two classes are mutual observers (Listing 14.4), with the incarnation mapping shown in Listing 14.5.

Listing 14.3: Simplified reference model for the observer pattern (as CD4A Model)

```
classdiagram ObserverPattern {
        class Observer;
        class Observed;
        association observes Observer -> Observes [*];
}
```

Listing 14.4: Concrete model of mutually observing classes (as CD4A Model)

```
classdiagram MutualObservers {
        class Attacker;
        class Defender;
        association watches Attacker <-> Defender [*];
}
```

Listing 14.5: Mapping mutual observers to the observer pattern (informal)

```
// Incarnation Mapping 1
Attacker ===> Observer
Defender ===> Observed
watches  ===> observes

// Incarnation Mapping 2
Defender ===> Observer
Attacker ===> Observed
watches  ===> observes
```

In specific projects or situations, the rules for conformance relations and their incarnation mappings might be more complex than stated above. Several properties can potentially be relaxed or restricted. Let us give several examples:

- The reference model uses an attribute `long ident` as an internal identification mechanism. It may be allowed to incarnate that attribute by `String ident`, because strings also have a linear order or by `int id` because we know that the reduced range is sufficient. In case a framework implementation, however, already relies on the data type `long`, such a change would be illegal.
- The reference model uses a certain cardinality for association $S : C1 \rightarrow C2$ `[card]`. Cardinality can, in principle, be restricted because we know from the domain that certain restrictions are okay, or extensions, e. g., from `[1]` to `[*]`, because that is needed in the domain. It may be that both changes are okay, or neither of them.
- The reference model has a direct subclass relationship between classes R_1 and subclass R_2. In the concrete model, there may be many incarnations of R_2 arranged in a deep hierarchy.
- Furthermore, it may be that adding additional attributes or associations is restricted because there are predefined implementations (or predefined generative techniques) of certain reference classes.

Potentially, there are many more details to be handled, such as freely chooseable independent names or attribute types, how to deal with enumerations, what are valid or necessary incarnations of interfaces and abstract classes, etc. A specifically interesting question is,

for example, whether a reference attribute or a reference association can be incarnated in the concrete model by a derived attribute or a composed association. In both cases, the realization is hidden in the combination of data structure and algorithmic functionality, such as getters with calculation bodies for derived attributes or an OCL-like logic formula describing how to derive a composed association from basic ones.

To summarize, we argue that there is no such thing as one single conformance relation even for class diagrams, but there may be a core, as shown in the requirements R1 to R6, plus a number of additional or optional relaxations. These rules define the conformance relation and therefore need to be processable by tools.

When looking at other kinds of languages, for example, state charts, the very same observations apply. There we have to map states, transitions, nesting of states, preconditions, entry and exit actions, etc. Preconditions may, for example, be relaxed or strengthened, or alternative transitions in nondeterministic state charts may be removed. A simulation relation for state charts or, more precisely, a refinement calculus like in Rumpe (1996) or Paech and Rumpe (1994) could be applied here. Especially challenging is the question of whether a reference transition may be realized by a sequence of concrete transitions and what happens if the sequence can be interrupted in the middle.

14.4 Operations upon Models and Reference Models

Having a concise and formal definition of the concept of a reference model as shown in Section 14.3 allows us to define functions for automating the handling of the relation between models and reference models. We can:

1. Check the conformance of models with respect to a reference model and
2. Constructively find valid mappings of a model to a reference model.

14.4.1 Conformance Checking

Definition 4 (Conformance Checking). *A conformance check is defined as a function*

$$checkConformance(M, I, R, C)$$

which defines whether the concrete model M conforms to the reference model R with respect to the given incarnation mapping I and the conformance rules C.

Conformance checking (see Definition 4) means to validate whether a model conforms to a reference model. Here we have to take the incarnation mapping into account and check it against the conformance rules of the conformance relation. The result of the checking in general is the statement whether the given model is a valid concretization of the reference model, taking as input the concrete model, the reference model, and the incarnation mapping. With the definition of incarnation mappings and conformance relations we can identify which mapping pair of model elements in M_e and R_e violates which rule of the conformance relation and regard this as additional, explaining output.
The implementation of this function depends on the rules defined for the conformance relation. Only few rules can be checked by taking only the incarnation mapping into account, but many rules also have to take the model and reference model into account. For example, rules R5 and R6 require knowledge about the inheritance relationships and associations

in both models. We consider graph matching algorithms (Livi and Rizzi 2012) a suitable implementation strategy for implementing conformance checking. This allows for a formal specification of rules in terms of graph pattern matching. When the conformance check requires information about the semantics, this must be encoded in the graph matching rules. Efficient implementations of graph matching exist for various modeling frameworks, e. g., QVT (Kurtev 2007), ATL (Jouault et al. 2006), or Henshin (Strüber et al. 2017) for the Eclipse Modeling Framework or MontiTrans (Hölldobler 2018) for MontiCore (Hölldobler et al. 2021).

Semantic differencing (Nachmann et al. 2022; Drave et al. 2019; Maoz et al. 2011) is an advanced analysis technique to understand the differences between two class diagrams on the basis of the actual object structures that are allowed in one but not the other model. This kind of semantic difference uses identical class, attribute, and association names to identify comparable classes. Semantic differencing, however, can be extended to reference models and their incarnations to understand the differences between the concrete realization and the original reference model when incorporating the conformance relation into the semantic difference calculation.

14.4.2 Incarnation Mapping Detection

Definition 5 (Incarnation Mapping Detection). *Detecting one or many incarnation mappings of a model M and a reference model R is defined as a function*

$$findMappings(M, R, C) = \{I_1, I_2, ..., I_n\}$$

that detects a set of valid incarnation mappings for a model M to a reference model R with respect to a conformance relation C.

Incarnation mapping detection (see Definition 5) means to automatically find valid incarnation mappings between a model and a reference model, given a conformance relation. There is not always exactly one valid incarnation mapping as a result, but the resulting set can contain zero, one, or many mappings. However, the search space as well as the result is finite, because the number of model elements M_e and R_e is finite.

Finding valid incarnation mappings can be computationally expensive, e. g., if all potential mappings between all model and reference model elements are checked against the conformance rules. We see some approaches for decreasing this computational complexity. (1) We can use additional knowledge to reduce the search space. If a model uses a known naming convention related to the reference model, this can be exploited. For example, in Listing 14.1, we can see that the incarnations of `Create` and `Update` all start with the respective name from the reference model elements. (2) We can produce a model resembling the target model and the corresponding mapping by applying the conformance rules constructively, as it is known from the generation of trace graphs in triple graph grammars (Schürr 1995). For example, consider the reference model in Section 14.2 given by the reference model `AccessRights`, the model `Flensburg`, and the conformance rules from Section 14.3.4.
We can produce a mapping by first applying rule R1, and let the concrete class `Person` incarnate the reference class `User` (correctly), and the concrete class `CarOwner` incarnate the reference class `Role` (correctly). Now we can apply the rules R6 together with R1 and R3, for the association `myCars` from `Person` to `CarOwner` to correctly incarnate `userroles` from `User` to `Role`. If we instead tried to produce a mapping where we let the concrete class `Person`

incarnate `Role` (instead of `User`), and the concrete class `CarOwner` incarnate the reference class `Role` (correctly), we also have to find an association that incarnates `Role -> Right [*]` according to rule R6. But there is none in the concrete model. The only remaining association to build from the `Person` is defined as `Person -> CarOwner`, and `CarOwner` does not incarnate `Right`. This incarnation mapping is not possible with respect to the rule R6. Further approaches to efficiently implement an incarnation mapping identification may exist and are subject to research.

It should be noted, that also a partial incarnation mapping can be helpful. A result of an incarnation mapping identification might also be a partial mapping, stating that the model does not conform to the reference model, but there are some incarnation mappings that come close. As an example, consider a scenario where a novel software architecture reference model for a specific domain has been developed based on existing software architecture models in practice. The developers of the reference model want to know to which degree existing architecture models conform to the novel reference architecture model. Partial mappings can now show for each individual architecture model how close it is to a valid concretization of the novel reference model, and consequently whether and where the reference model or the concrete model should be adapted.

Such partial mappings can also be used as a basis for repair techniques, like defined in Kautz and Rumpe (2018), which is the identification of necessary modifications to the concrete model to establish a complete realization of the reference model.

14.5 Representations of the Incarnation Mapping

An interesting question is what an explicit representation of an incarnation mapping looks like, if needed. Above, we have given an informal list to describe this mapping. The list contains explicitly given names for classes and associations but also maps unnamed associations by using the `(C1 -> C2)` notation. We actually see several alternatives for representing such mappings:

A1 An explicit mapping language similar to the artifact shown in Listing 14.1 can be used.

A2 The mapping can be extracted from the two models by using naming conventions.

A3 The concrete model explicitly refers to the reference model by using the domain concepts introduced in the reference model as stereotypes, labels, or flags.

A4 The concrete model is actually an extension of the reference model, which means that all model elements (here, classes and associations) defined in the reference model are explicitly included and therefore used in the concrete model.

Alternative A3 can be seen quite commonly when generators are used. For example, consider the example in Listing 14.6. Here a lot of stereotypes pollute the otherwise clean concrete model. Using a number of conventions, it may be that certain labels have not been defined explicitly but can be reconstructed. This is regularly the case for a concrete association when both its concrete classes have been stereotyped.
Another possibility is to combine the explicit labeling with the implicit use of naming conventions, as described in Alternative A2 such as (1) same name or (2) a shared prefix (see `UpdateCar`) or analogously a shared postfix (like `StopState`). This, of course, shares the problem of unwanted, coincidental matches but largely reduces the amount of labeling necessary. This is a common approach when applying design patterns, which we also regard

as a special form of reference models denoted as class diagrams, e. g., in Gamma et al. (1995), where often the roles of the individually participating classes are used as parts of the actual class names.

Listing 14.6: Concrete model using stereotypes (as CD4A Model)

```
classdiagram <<labels:"AccessRights">> Flensburg {
        class <<User>> Person;
        class <<Role>> Admin;
        class <<Role>> CarOwner;
        association <<userroles>> Person -> Admin [0..1];
        association <<userroles>> myCars   Person -> CarOwner [*];
        class <<Right>> Permission;
        class <<Create>> CreateCar extends Permission;
        class <<Update>> UpdateCar extends Permission;
        association allowed [1] CarOwner <-> Permission [1..3];
}
```

To prevent potential pollution with stereotypes from various technological spaces, we have explored the possibility of using an extra tagging language that allows labeling model elements without invasively changing the original model stored in extra artifacts in Greifenberg et al. (2015). Tagging models can be applied in various domains (Dalibor et al. 2019; Maoz et al. 2016).

Alternative A4 is quite common, especially if the reference model has a framework implementation, where the classes of the reference model are already and explicitly implemented under the very same names. Using different names is then not a possibility. The concrete model would then have to be specified as shown in Listing 14.7.

Listing 14.7: Concrete model as extension (as CD4A model)

```
classdiagram Flensburg extends AccessRights {
        class Person extends User;
        class Admin extends Role;
        class CarOwner extends Role;
        class Permission extends Right;
        class CreateCar extends Permission, Create;
        class UpdateCar extends Permission, Update;
}
```

This model is relatively compact because associations need not be mentioned explicitly again but are inherited from `AccessRights`. However, this approach does neither allow for adapting cardinalities of associations nor building multiple incarnations of associations. It also has limitations with respect to multiple inheritance: In the example in Listing 14.7, the classes `CreateCar` and `UpdateCar` are modeled to extend the class `Permission`. In addition they now also extend their respective reference model classes. While this is possible in UML class diagrams, multiple inheritance is not permitted in many programming languages. It is, thus, ambiguous how to implement the classes. A similar approach could be to develop the concrete model independently at first and then apply a generalized merge algorithm, similar to the one provided by MontiCore's CD4A language, and merge the framework classes, associations, and, thus, infrastructure into the concrete model.

14.6 Mapping a Reference Implementation to a Concrete Implementation

The reuse of models and the use of reference models becomes particularly interesting, when not only the reference model and its particular design can be reused, but also a reference implementation exists, that already embodies a lot of technical details and which can also be reused. We therefore assume, that as described in Section 14.3, reference models may be accompanied by a reference implementation.

The really interesting part is, therefore, the transfer of an existing reference implementation, e. g., for `AccessRights` into the concrete implementation. Ideally, this is done as automatically as possible. If the reference implementation exists in the form of a methodological approach this must be done manually, but still is of great help. If it is predefined in a framework there are chances that the concrete implementation reuses that framework either through subclassing or through delegation. Even better would be a setting, where a generator, which would map the model to code anyway, directly adds all the technical details from the reference implementation, such as rights management, database storage, etc., into such a concrete implementation.

Smart tooling is required to transfer predefined code for the reference model to the concrete model, respectively, its implementation. This transfer technologically depends on the representation of the mapping (see Section 14.5). If an explicit mapping language is used (A1), code can be generated based on a reference implementation and an incarnation mapping model. Here it will be necessary to exchange the names of code elements during the generation, as the mapping can effectively be seen as a refactoring. The same applies when the mapping can be extracted by using naming conventions (A2) or stereotypes, labels, or flags (A3). If a concrete model extends the reference model (A4), there is also the opportunity to define a reference implementation in the extended classes (e. g., `AccessRights`, `User`, `Role`, and `Right` in Section 14.2). Figure 14.1 shows the renaming approach on the example of `AccessRights` to Listing 14.1. In this alternative, the code given by the reference model element is copied and refactored. The reference implementation itself is not part of the code for the concrete model.

The left side of the example shows excerpts of code for reference model elements, while the right side shows the refactored code for the concrete model elements. The class `Right` is

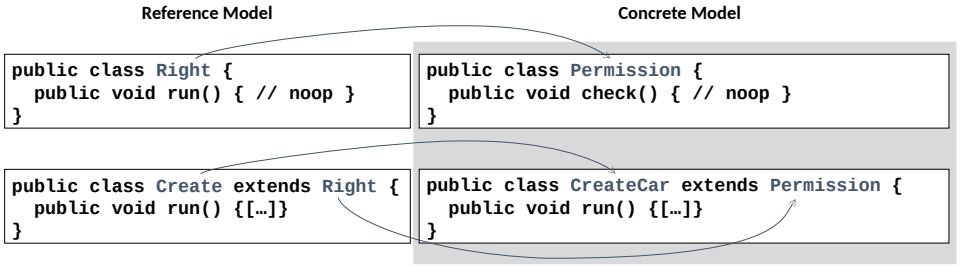

Figure 14.1: Example of the concretization of given Java code for reference models via renaming

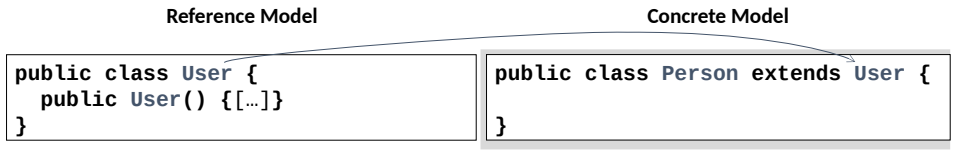

Figure 14.2: Example of the concretization of given Java code for reference models via extension

code that implements the default functionality of the respective reference model element (upper left part of Figure 14.1). The default functionality is that it provides a public method `run`, that does nothing but can be overridden by subclasses. The code for the concrete model element is created by applying a renaming refactoring on a copy of that code, that replaces the reference model element name with the concrete model element name. The reference model element `Create` on the lower left extends the class `Right`. It overrides the method `run` to implement a default functionality for the right to create something. The lower right side shows the refactored code for the concrete model element `CreateCar`. Here we renamed the class name and the name of the extended class according to the example's incarnation mapping.
Figure 14.2 shows the extension approach for the reference model element `User` and its concretization `Person`. In contrast to the former alternative, here the code for the reference model element is part of the code for the concrete model. The reference implementation on the left side of Figure 14.2 provides a constructor. The code for the concrete model element extends that reference model class. Due to the inheritance the constructor of the class `Person` is resolved to the constructor of the reference model element class `User`, thus effectively providing a default implementation to the concrete model element class `Person`.

The extension mechanism has drawbacks due to the limitation to multiple inheritance in many programming languages. For example, the extension mechanism could not be used unchanged in this example for the specific rights `Create` and `Update`. One could expect that code for these reference model elements is implemented as Java classes that extend the class `Right`. With the extension mechanism for implementation, we thus define the two classes `CreateCar` and `UpdateCar`, which extend both classes `Create` or `Update` respectively due to the extension mechanism and `Permission` due to the inheritance relation in the concrete model. This multiple inheritance cannot be implemented in languages such as Java due to them not allowing multiple inheritance.

A workaround to this issue in Java could be to use Java interfaces with their default implementation functionality, because Java classes can implement an arbitrary number of interfaces. Figure 14.3 shows this workaround for the example above. Instead of classes, in this example we define Java interfaces for the reference model elements `Right` and `Create`. They provide their reference implementation via default methods in Java interfaces. To showcase the consequence, these default implementations print their class names. The concrete model elements `Permission` and `CreateCar` are defined as classes. Following the extension mechanism, `Permission` now implements the interface `Right`. `CreateCar` implements the interface `Create`, following the extension mechanism, and extends the class `Permission` as modeled in the concrete model. It is now not unambiguously resolvable, which method will be executed upon calling `CreateCar.run`. The Java compiler marks this

Reference Model

```
public interface Right {
  public int getGranted();
  public void increaseGranted();
  public default void run() {
    System.out.println("Right");
  }
}
```

```
public interface Create
                    extends Right {
  public default void run() {
    System.out.println("Create");
  }
}
```

Concrete Model

```
public class Permission implements Right {
  int granted = 0;
  public int getGranted(){[…]}
  public void increaseGranted(){[…]};
}
```

```
public class CreateCar extends Permission
                          implements Create {
  public void run() {
    Create.super.run(); // "Create"
    super.run();        // "Right"
  }
}
```

Figure 14.3: Example of the concretization of given Java code for reference models via interfaces with default implementations

as error. Therefore, we override the method `run` and explicitly call the respective method. Conceptually, the class represents the functionality of the model class `CreateCar`, which incarnates the reference model element `Create`. Therefore, we expect that the functionality of `Create` should be called. In the example, we show how to execute either method for reference.

Default implementations for interfaces do not allow saving states between method calls. That is, interfaces cannot have attributes. Therefore, using interfaces with default implementations limits the code for reference model elements to operations without side effect. To showcase how to save state as a functionality with this method, we added a getter and setter method to the interface `Right` as functionality for the respective reference model element. This requires the concrete implementation `Permission` to handle these methods and provide means to save the state. Alternatively, a runtime could inject a context element as method parameter. Another issue with this alternative arises when there are multiple interfaces that implement the functionality of mapped reference classes, which declare the same methods. For example, consider that the class `CreateCar` incarnates multiple reference model classes, which all implement a method `run`. In that case the implementing class has to decide which method to execute, as shown in the example in the class `CreateCar`. If multiple of them are executed, it also has to decide on an order. Our example shows the drawback of the extension mechanism. Nevertheless, such libraries and frameworks are widespread in practice. This alternative has the advantage that the respective libraries can be efficiently distributed by the usual means (e. g., providing download links or providing them using repositories for dependency resolvers like Maven[2] for Java or pip[3] for Python) and dynamically linked. With the renaming alternative, there is no library to distribute, but the complete code needs to be generated and included in the resulting program code.

2 https://maven.apache.org/

3 https://pypi.org/project/pip/

14.7 Related Work

In a literature survey, Arora et al. (2022) identified that there is no universally accepted understanding or definition of reference models. They extracted nine qualities to characterize reference models: *reusable, flexible, reliable, designed systematically, generally valid, required, user-centered, comprehensive*, and *educative*. They define a reference model as 'a holistic collection of methodologies systematically structured in an architecture in which every element (e. g., guidelines, methods, procedures, and entities, ...) is made transparent and outlined as a generic solution based on both best practices and innovative approaches.' We take a similar, but more formal perspective on reference models. It is more focused on the formalization of the concepts of reference models and the relation between concrete models and reference models.

Reference models are subject to research in the domain of software architecture frequently. The ArCh approach by Herold et al. (2013) describes reference architectures and their constraints with an ontology and first-order logic formulas. Concrete models are extracted from models or source code and translated to first-order logic formulas. The correspondence check is then implemented by checking the reference architecture and constraint formulas against the concrete model formulas. Our approach uses the same language for reference models as it is used for concrete models, which is closer to the day-to-day development of software engineers.

Weinreich and Buchgeher (2014) describe reference architectures for service-oriented architectures (SOA) and automated conformance checking. In this approach, reference models are described using roles, relationships between roles, and logic-based constraints for those elements. Roles and their relations are defined using rules, which include rules for mapping concrete elements to reference model elements by properties. For example, a role can be mapped to all links between concrete services with the property 'local'. Conformance is checked by validating that the rules hold for the given SOA. In contrast, we explicitly distinguish the mapping from the reference model elements in the conceptualization and describe multiple alternatives to represent incarnation mappings.

Bucaioni et al. (2022) describe MORE, an MDE approach for modeling reference architecture models, checking the conformance of concrete architectures to reference architectures, and to ensure the compliance of concrete models by providing guidelines and rules. They describe a reference architecture meta model that can be used to describe reference architecture models. These models are then lifted to domain-specific meta models of which concrete architecture models can be constructed. A meta model can be used to define explicit mappings from concrete architectures to reference architectures. A conformance check can then validate whether the concrete model conforms to the reference model based on the relationship between meta models and their models. In this approach architecture models instantiate the domain specific meta model. Our definition allows for arbitrary model-based languages.

14.8 Conclusion

The term 'reference model' is broadly used publicly with a variety of meanings, as many different domains and their communities have different and mostly informal understandings of reference models. A more concise and formal understanding of the concept of reference

models is needed to improve the tool-assisted relationship between a general reference model and the set of concrete realizations that conform to that reference model. In this contribution, we discussed our understanding of reference models and conformance relations, which specify which concrete models are valid with respect to the reference model. We demonstrated our understanding of conformance relations on structure reference models for data, represented as class diagrams from the UML. We argue that conformance relations must be defined for modeling languages individually because they are driven by the semantics of the particular language. Purely syntax-based conformance relations will not suffice.

To be able to validate whether concrete models conform to reference models it is required to specify a mapping between concrete model elements and reference model elements. We describe different ways to represent these mappings with different characteristics and argue that there is no generalized form that is fit for all purposes. We discuss the challenges of implementing conformance checking and the detection of mappings between reference model elements and concrete model elements. We also identify the attachment of an implementation or other additional information to reference models as a challenge and provide alternative solutions for automating the translation of such a reference implementation to a concrete implementation for concrete models. Further research is necessary towards the automation of conformance checking and for automatically detecting (partial) mappings, as well as smart tooling, e. g., to automatically develop valid concrete models from reference models as development support, and to transfer reference implementations to concrete implementations.

Acknowledgement

We would like to thank our colleagues Max Stachon and Sebastian Stüber for the valuable discussions on this topic.

In Honour of Ulrich Frank

We contribute to this anthology in honour of Ulrich Frank as we have hold him in high esteem for his professional expertise, his willingness to engage in discussion and the lively exchange of ideas for many years. Some of his research highlights which we appreciate very much are his work towards open reference models (Frank and Strecker 2007), the evaluation of reference models (Frank 2007), and his contribution to globalizing domain-specific languages (Cheng et al. 2015) together with the lively discussions in the related Dagstuhl seminar. His research on a method for designing domain-specific modeling languages (Frank 2010), inspired us to restructure our initial approach for a domain-specific modeling method to create languages for the ambient assistance domain (Michael and Mayr 2015). More recently, he worked on characteristics of low-code platforms (Bock and Frank 2021b; Bock and Frank 2021a), providing open research ideas and opportunities for future developments. This work inspired us to discuss adaptation mechanisms for data and process models within our work on low-code development platforms for digital twins (Dalibor et al. 2022).

As an active member of the conceptual modeling (ER conference) and model-driven software engineering (MODELS conference) community, he is a link between advocates of data modeling and those who have a stronger focus on software engineering and model-driven methods. Due to his interest in industry collaborations and keen sense of business

challenges, we were not surprised by his research on a conceptual framework and modeling languages for multi-perspective enterprise modeling (MEMO) (Frank 2002) integrating different perspectives. For his current focus on multi-level modeling (Frank 2022; Frank 2014b, which, to cite him, 'has not yet made it to the research mainstream' (Frank 2022), the research community is showing an increased awareness of the problems it promises to solve.

Ulrich Frank is an active member of the German Informatics Society and especially the QFAM[4], a forum for modeling enthusiasts from different areas in Computer Science. He is an active participant in our regular modeling discussion rounds with Bernhard Thalheim and Heinrich C. Mayr introducing new perspectives. His broad expertise and interests make him an excellent organizer of PhD symposium on conferences, as he provides helpful feedback and inspires PhDs to rethink their research ideas. Back in 2012 in a lecture room full of new PhD candidates (Sinz and Schürr 2012), it was his statement and questions which influenced one of the co-authors of this article to rethink the evaluation (Mayr and Michael 2012; Michael and Mayr 2017) of the modeling language created in her PhD (Michael and Mayr 2013). His suggestions on increasing the value of research by promoting value to the researcher are timelessly current: He suggests getting outraged, relaxing and not taking the factual manifestations of academia or ourselves as researchers too seriously, enjoying research, being free and ambitious, to surprise your peers, and help to create a better world (Frank 2014a).

We would like to thank Ulrich Frank for all his very valuable contributions and his stimulating and discussion-promoting remarks and questions. Our experience has shown that it is worthwhile to listen actively to his remarks and to reflect on them in one's own research context. Thus, we are looking forward to all next conferences, workshops, and discussion sessions with a versatile modeling enthusiast with practical relevance.

References

Atouani, A., Kirchhof, J. C., Kusmenko, E. and Rumpe, B. (Oct. 2021). 'Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems'. In: *20th ACM SIGPLAN Int. Conf. on Generative Programming: Concepts and Experiences (GPCE 21)*. Ed. by E. Tilevich and C. De Roover. ACM SIGPLAN, pp. 55–68.

American National Standards Institute (Apr. 1981). 'Data Processing-Open Systems Interconnection - Basic Reference Model'. In: *SIGCOMM Comput. Commun. Rev.* Vol. 11. 2. Association for Computing Machinery (ACM), pp. 15–65. DOI: 10.1145/1015586.1015589.

Arora, S., Ceccolini, C. and Rabe, M. (2022). 'Approach to Reference Models for Building Performance Simulation'. In: *10th Int. Conf. on Model-Driven Engineering and Software Development, MODELSWARD 2022*. Ed. by L. F. Pires, S. Hammoudi and E. Seidewitz. SCITEPRESS, pp. 271–278. DOI: 10.5220/0010888800003119.

Bock, A. C. and Frank, U. (2021a). 'In Search of the Essence of Low-Code: An Exploratory Study of Seven Development Platforms'. In: *2021 ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 57–66. DOI: 10.1109/MODELS-C53483.2021.00016.

4 Querschnittsfachaussuss Modellierung der Gesellschaft für Informatik e.V. https://qfam.gi.de/

Bock, A. C. and Frank, U. (2021b). 'Low-Code Platform'. In: *Business & Information Systems Engineering* 63.6, pp. 733–740. DOI: 10.1007/s12599-021-00726-8.

Bucaioni, A., Di Salle, A., Iovino, L., Malavolta, I. and Pelliccione, P. (Aug. 2022). 'Reference architectures modelling and compliance checking'. In: *Journal Software and Systems Modeling (SoSyM)*. DOI: 10.1007/s10270-022-01022-z.

Cheng, B. H. C., Degueule, T., Atkinson, C., Clarke, S., Frank, U., Mosterman, P. J. and Sztipanovits, J. (2015). 'Motivating Use Cases for the Globalization of DSLs'. In: *Globalizing Domain-Specific Languages: International Dagstuhl Seminar, Dagstuhl Castle, Germany, October 5-10, 2014, Revised Papers*. Springer International Publishing, pp. 21–42. DOI: 10.1007/978-3-319-26172-0_3.

Dalibor, M., Heithoff, M., Michael, J., Netz, L., Pfeiffer, J., Rumpe, B., Varga, S. and Wortmann, A. (2022). 'Generating Customized Low-Code Development Platforms for Digital Twins'. In: *Journal of Computer Languages (COLA)* 70. DOI: 10.1016/j.cola.2022.101117.

Dalibor, M., Jansen, N., Kirchhof, J. C., Rumpe, B., Schmalzing, D. and Wortmann, A. (2019). 'Tagging Model Properties for Flexible Communication'. In: *Proc. of MODELS 2019. Workshop MDE4IoT*. Ed. by N. Ferry, A. Cicchetti, F. Ciccozzi, A. Solberg, M. Wimmer and A. Wortmann. CEUR Workshop Proceedings, pp. 39–46.

Drave, I., Kautz, O., Michael, J. and Rumpe, B. (2019). 'Semantic Evolution Analysis of Feature Models'. In: *Int. Systems and Software Product Line Conference (SPLC'19)*. Ed. by T. Berger, P. Collet, L. Duchien, T. Fogdal, P. Heymans, T. Kehrer, J. Martinez, R. Mazo, L. Montalvillo, C. Salinesi, X. Tërnava, T. Thüm and T. Ziadi. ACM, pp. 245–255.

Eom, S.-J. and Fountain, J. E. (2013). *Enhancing information services through public-private partnerships: Information technology knowledge transfer underlying structures to develop shared services in the U.S. and Korea*, pp. 15–40. DOI: 10.4018/978-1-4666-4173-0.ch002.

Fettke, P., Loos, P. and Zwicker, J. (2006). 'Business Process Reference Models: Survey and Classification'. In: *Business Process Management Workshops*. Ed. by C. J. Bussler and A. Haller. Springer Berlin Heidelberg, pp. 469–483.

Forme, F. G. L., Botta-Genoulaz, V. and Campagne, J. (2007). 'A framework to analyse collaborative performance'. In: *Comput. Ind.* 58.7, pp. 687–697. DOI: 10.1016/j.compind.2007.05.007.

Frank, U. (2002). 'Multi-perspective enterprise modeling (MEMO) conceptual framework and modeling languages'. In: *35th Annual Hawaii International Conference on System Sciences*, pp. 1258–1267. DOI: 10.1109/HICSS.2002.993989.

Frank, U. (2007). 'Evaluation of Reference Models'. In: *Reference Modeling for Business Systems Analysis*. Ed. by P. Fettke and P. Loos. IGI Global, pp. 118–140. ISBN: 9781599040547. DOI: 10.4018/978-1-59904-054-7.ch006.

Frank, U. (2010). *Outline of a method for designing domain-specific modelling languages*. ICB-Research Report 42.

Frank, U. (2014a). 'Higher Value of Research by Promoting Value for Researchers'. In: *Communications of the Association for Information Systems* 34. DOI: 10.17705/1CAIS.03443.

Frank, U. (2014b). 'Multilevel Modeling: Toward a New Paradigm of Conceptual Modeling and Information Systems Design'. In: *Business & Information Systems Engineering* 6.6, pp. 319–337. DOI: 10.1007/s12599-014-0350-4.

Frank, U. (2022). 'Multi-level modeling: cornerstones of a rationale'. In: *Software and Systems Modeling* 21.2, pp. 451–480. DOI: 10.1007/s10270-021-00955-1.

Frank, U. and Strecker, S. (2007). 'Open Reference Models - Community-driven Collaboration to Promote Development and Dissemination of Reference Models'. In: *Enterprise*

Modelling and Information Systems Architectures – International Journal of Conceptual Modeling (EMISAJ) 2.2, pp. 32–41. DOI: 10.18417/emisa.2.2.4.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Greifenberg, T., Look, M., Roidl, S. and Rumpe, B. (2015). 'Engineering Tagging Languages for DSLs'. In: *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*. ACM/IEEE, pp. 34–43.

Gray, J. and Rumpe, B. (2021). 'Reference models: how can we leverage them?' In: *Journal Software and Systems Modeling (SoSyM)* 20.6, pp. 1775–1776.

Herold, S., Mair, M., Rausch, A. and Schindler, I. (2013). 'Checking Conformance with Reference Architectures: A Case Study'. In: *17th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2013*. Ed. by D. Gasevic, M. Hatala, H. R. M. Nezhad and M. Reichert. IEEE Computer Society, pp. 71–80. DOI: 10.1109/EDOC.2013.17.

Hölldobler, K., Kautz, O. and Rumpe, B. (2021). *MontiCore Language Workbench and Library Handbook: Edition 2021*. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag.

Hölldobler, K. (Dec. 2018). *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 36. Shaker Verlag.

Joshi, H. and Michel, H. (2008). 'Integrated Technical Reference Model and Sensor Network Architecture'. In: *Int. Conf. on Wireless Networks*. Ed. by H. R. Arabnia and V. A. Clincy. CSREA Press, pp. 570–576.

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I. and Valduriez, P. (2006). 'ATL: a QVT-like transformation language'. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pp. 719–720.

Kautz, O. and Rumpe, B. (2018). 'On Computing Instructions to Repair Failed Model Refinements'. In: *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*. ACM, pp. 289–299.

Kurtev, I. (2007). 'State of the art of QVT: A model transformation language standard'. In: *International Symposium on Applications of Graph Transformations with Industrial Relevance*. Springer, pp. 377–393.

Livi, L. and Rizzi, A. (Aug. 2012). 'The graph matching problem'. In: *Pattern Analysis and Applications* 16.3, pp. 253–283. DOI: 10.1007/s10044-012-0284-8.

Mayr, H. C. and Michael, J. (2012). 'Control pattern based analysis of HCM-L, a language for cognitive modeling'. In: *Int. Conference on Advances in ICT for Emerging Regions (ICTer2012)*. IEEE, pp. 169–175. DOI: 10.1109/ICTer.2012.6421414.

Mayr, H. C. and Thalheim, B. (2020). 'The triptych of conceptual modeling'. In: *Software and Systems Modeling*. DOI: 10.1007/s10270-020-00836-z.

Michael, J. and Mayr, H. C. (2017). 'Intuitive understanding of a modeling language'. In: *Asia-Pacific Conference on Conceptual Modeling at the Australasian Computer Science Week Multiconference (ACSW'17)*. ACM.

Michael, J., Koren, I., Dimitriadis, I., Fulterer, J., Gannouni, A., Heithoff, M., Hermann, A., Hornberg, K., Kröger, M., Sapel, P., Schäfer, N., Theissen-Lipp, J., Decker, S., Hopmann, C., Jarke, M., Rumpe, B., Schmitt, R. H. and Schuh, G. (2023). 'A Digital Shadow Reference Model for Worldwide Production Labs'. In: *Internet of Production: Fundamentals, Applications and Proceedings*. Ed. by C. Brecher, G. Schuh, W. van der Aalst, M. Jarke, F. Piller and M. Padberg. Springer, pp. 1–28. DOI: 10.1007/978-3-030-98062-7_3-2.

Michael, J. and Mayr, H. C. (2013). 'Conceptual Modeling for Ambient Assistance'. In: *Conceptual Modeling - ER 2013*. Vol. 8217. LNCS. Springer, pp. 403–413.

Michael, J. and Mayr, H. C. (2015). 'Creating a Domain Specific Modelling Method for Ambient Assistance'. In: *International Conference on Advances in ICT for Emerging Regions (ICTer2015)*. IEEE, pp. 119–124.

Mayr, H. C., Michael, J., Shekhovtsov, V. A., Ranasinghe, S. and Steinberger, C. (2018). 'A Model Centered Perspective on Software-Intensive Systems'. In: *Enterprise Modeling and Information Systems Architectures (EMISA'18)*. Ed. by M. Fellmann and K. Sandkuhl. Vol. 2097. CEUR Workshop Proceedings. CEUR-WS.org, pp. 58–64.

Maoz, S., Ringert, J. O. and Rumpe, B. (2011). 'CDDiff: Semantic Differencing for Class Diagrams'. In: *ECOOP 2011 - Object-Oriented Programming*. Ed. by M. Mezini. Springer Berlin Heidelberg, pp. 230–254.

Maoz, S., Ringert, J. O., Rumpe, B. and Wenckstern, M. v. (2016). 'Consistent Extra-Functional Properties Tagging for Component and Connector Models'. In: *WS on Model-Driven Engineering for Component-Based Software Systems (ModComp'16)*. Vol. 1723. CEUR Workshop Proceedings, pp. 19–24.

Nachmann, I., Rumpe, B., Stachon, M. and Sebastian, S. (June 2022). 'Open-World Loose Semantics of Class Diagrams as Basis for Semantic Differences'. In: *Modellierung 2022*. Gesellschaft für Informatik, pp. 111–127.

Paech, B. and Rumpe, B. (1994). 'A new Concept of Refinement used for Behaviour Modelling with Automata'. In: *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*. LNCS 873. Springer, pp. 154–174.

Reinhartz-Berger, I., Soffer, P. and Sturm, A. (2010). 'Extending the Adaptability of Reference Models'. In: *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 40.5, pp. 1045–1056. DOI: 10.1109/TSMCA.2010.2044408.

Rumpe, B. (2017). *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International.

Rumpe, B. (1996). *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Dissertation, Fakultät für Informatik, Technische Universität München. München: Herbert Utz Verlag Wissenschaft.

Scheer, A.-W. (1994). *Business Process Engineering: Reference Models for Industrial Enterprises*. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-79142-0.

Schürr, A. (1995). 'Specification of graph translators with triple graph grammars'. In: *Graph-Theoretic Concepts in Computer Science*. Springer Berlin Heidelberg, pp. 151–163. DOI: 10.1007/3-540-59071-4_45.

Sinz, E. J. and Schürr, A., eds. (2012). *Modellierung 2012*. Lecture Notes in Informatics P-201. Springer.

Stachowiak, H. (1973). *Allgemeine Modelltheorie*. Wien, New York: Springer Verlag.

Strüber, D., Born, K., Gill, K. D., Groner, R., Kehrer, T., Ohrndorf, M. and Tichy, M. (2017). 'Henshin: A usability-focused framework for emf model transformation development'. In: *International Conference on Graph Transformation*. Springer, pp. 196–208.

van der Aalst, W. and Kumar, A. (2001). 'A reference model for team-enabled workflow management systems'. In: *Data & Knowledge Engineering* 38.3, pp. 335–363. ISSN: 0169-023X. DOI: https://doi.org/10.1016/S0169-023X(01)00034-9.

Weinreich, R. and Buchgeher, G. (2014). 'Automatic Reference Architecture Conformance Checking for SOA-Based Software Systems'. In: *IEEE/IFIP Conference on Software Architecture, WICSA 2014*. IEEE, pp. 95–104. DOI: 10.1109/WICSA.2014.22.

Zimmermann, A., Schmidt, R., Sandkuhl, K., Wißotzki, M., Jugel, D. and Möhring, M. (2015). 'Digital Enterprise Architecture - Transformation for the Internet of Things'. In: *19th IEEE International Enterprise Distributed Object Computing Workshop, EDOC Workshops 2015, Adelaide, Australia, September 21-25, 2015*. Ed. by J. Kolb, B. Weber, S. Hallé, W. Mayer, A. K. Ghose and G. Grossmann. IEEE Computer Society, pp. 130–138. DOI: 10.1109/EDOCW.2015.16.