JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

[KRS+24] M. Konersmann, B. Rumpe, M. Stachon, S. Stüber, V. Voufo:
Towards a Semantical Useful Definition of Conformance with a Reference Model.
In: Journal of Object Technology (JOT), Volume 23(3),
AITO - Association Internationale pour les Technologies Objets, pp. 1-14, Jul. 2024.

Towards a Semantically Useful Definition of Conformance with a Reference Model

Marco Konersmann, Bernhard Rumpe, Max Stachon, Sebastian Stüber, and Valdes Voufo
Software Engineering, RWTH Aachen University, Germany
https://se-rwth.de

ABSTRACT

In software engineering reference models are used as a guidance for implementing potential solutions to recurring problems, e.g., as reference architecture models or reference data models. Despite the broad use of reference models in practice, there is no clear definition for what it means that a concrete model conforms to a reference model. This prevents us from developing automated tools for conformance checking. In this paper, we provide a semantically useful definition of conformance of concrete models to reference models. We then present concepts and tools for concrete automated conformance checks for (1) class diagrams, (2) feature diagrams, and (3) state charts, which we developed based on that definition. Finally, we discuss (1) the commonalities and differences of the presented automated conformance checks and (2) general design considerations for developing reference model conformance checkers in the context of model-driven engineering. Key findings are that reference models should use the same language as their concrete models, conformance checks require conformance mappings between reference and concrete model elements, and conformance rules must be based on formally defined language semantics.

KEYWORDS Reference Models, Conformance, Model-Driven Engineering, Semantic Refinement, UML.

1. Introduction

Reference models (Fettke & Loos 2006) are broadly used in practice, *e.g.,* for creating standards (ITU-T 1994), employing best-practices (Gamma et al. 1997), education, and communication. A typical way of employing reference models in software engineering is to develop and communicate a reference model informally with a diagram and accompanying text (e.g., Gamma et al. 1997), sometimes accompanied by an implementation (e.g., Lu et al. 2020), and to expect that developers build models, components, or applications, which resemble the reference model. In the existing literature we find reference models, *e.g.,* for cloud[1] or IoT architectures (Bauer et al. 2013), digital shadows for smart manufacturing (Michael et al. 2023), business processes[2] (Kirchmer & Franz 2020) or the famous ISO OSI Layer reference model (ITU-T 1994). However, interpretations of reference models are quite diverse (Arora et al. 2022).

In Model Driven Engineering (MDE) we utilize models as first-class citizens to formally specify systems, using automated methods that take these models as input, *e.g.,* for analysis or code generation (Hölldobler et al. 2021). In contrast, reference models are mostly used informally (Gray & Rumpe 2021).

Reference models specify properties that must hold for concrete models, *e.g.,* which model elements may or must exist and how they interrelate. However, what existing reference models specify for their concrete models and how they do that varies considerably in practice. While a lot of research has already been put into reference modeling (e.g., Fettke & Loos 2006; Frank 2006; Becker & Delfmann 2007), and especially reference architectures (e.g., Garcés et al. 2021; Bucaioni et al. 2022; Cloutier et al. 2009), the use of reference models in MDE has not been received the same attention.

JOT reference format:
Marco Konersmann, Bernhard Rumpe, Max Stachon, Sebastian Stüber, and Valdes Voufo. *Towards a Semantically Useful Definition of Conformance with a Reference Model.* Journal of Object Technology. Vol. 23, No. 3, 2024.
http://dx.doi.org/10.5381/jot.2024.23.3.a5

[1] AWS reference architectures https://aws.amazon.com/architecture/, accessed 2023-11-24

[2] e.g., https://orwiki.org

A major benefit of using reference models is that concrete models can be compared to them. However, currently no generic conceptualization of *conformance* to a reference model exists. At the moment, conformance is simply an informal notion that has to be checked manually. This leads to ambiguous understanding and manual effort. A formal definition of reference model unifies the understanding and enables automatic tooling. This paper's objective is to lay the foundations for automated conformance checks of reference models in the context of MDE. We investigate the research question *"How can automated conformance checks be implemented for reference models in model-driven engineering?"*.

As a motivating example, Figure 1 shows a simple reference model and a concrete model alongside two so-called *incarnation mappings* which map concrete model elements or *incarnations* to their corresponding reference model elements. The reference model shows a Class Diagram (CD) of the adapter pattern (Gamma et al. 1997). The concrete model shows a CD of a graphics library, which implements the pattern twice. The two incarnation mappings are visualized by edges from the concrete to the reference models that are labeled with an ID according to the mappings they belong to Figure 1. Concerning the incarnations: First, the `GraphicalEditor` takes the role of the `Client` and the `GraphicalObject` interface takes that of the `Target` interface. The `EdgeAdapter` takes the role of the `Adapter` to adapt the `Edge`. Analogously, the concrete model implements the adapter pattern by adapting the `Node` via the `NodeAdapter`.

To enable an automated conformance check, the meaning of conformance needs to be well-defined. While the specifics might differ from language to language, any conformance checker must be able to decide whether a concrete model captures the essence or meaning of a reference model and thus requires some formal notion of model semantics (Harel & Rumpe 2004).

To make automated conformance checks available for MDE, we require a formal understanding of reference models, their notation, their semantics, and the mapping of concrete models to their reference model. The contributions of this paper are as follows:

1. We formally define reference models and conformance relations (section 3),
2. We present the concept and tools of three concrete automated conformance checks, for CDs (subsection 4.1), Feature Diagrams (FDs) (subsection 4.2) and statecharts (subsection 4.3),
3. We discuss general aspects of conformance as well as language-specific considerations, and examine commonalities and differences of our approaches (section 5).

In further sections we comment on related work (section 2) and conclude in section 6.

2. Related Work

Conformance checking—sometimes also called *compliance checking*—of concrete models and reference models can be found for reference architecture models. E.g., Herold et al. (2013) present a rule-based architecture conformance checking approach with an industrial case study. They use an explicit meta-model for reference architectures and a set of rules to automatically evaluate whether an architecture model conforms to the meta-model. Similarly, Bucaioni et al. (2022) use meta-models for specifying reference architectures. Weinreich & Buchgeher (2014) extract the architecture from source code and check the conformance to rules, that specify the reference architecture. Caracciolo et al. (2015) employ a DSL for reference architecture conformance rules. In our approach we model reference models in the same language as the concrete model. This allows us to specify reference models for each modeling language without developing a new language for rules, although we still need to develop concrete rules for each language.

Kim & Shen (2007) evaluate the structural conformance of CDs to design patterns. Their design patterns are specified using extended UML CDs. They decompose the model for efficient conformance checks and variants. Our work is not limited to CDs or predefined design patterns.

Conformance checking techniques are also used between models and their meta-models. E.g., Egea & Rusu (2009) validate the conformance of models to meta-models by including context conditions. These approaches use OCL or programming languages to define context conditions. In contrast, we check the conformance relation of two models (reference model and conformance model) in the same language.

Other approaches to conformance checking include the compliance of models to policies. E.g., Tran et al. (2012) check whether a model complies to regulatory policies, as stated, e.g., by legislation. They describe a model-based approach, which is tailored for non-developers, using an OCL-like, extensible DSL. The DSL core can be extended for different domains. We follow a different approach as we have a different target group: Our users are modelers and language developers. Therefore, reference models in our approach are models in the same language as the concrete models.

We have previously discussed notions of a conformance relation between concrete and reference models as well as incarnation mappings, focusing primarily on CDs (Konersmann et al. 2024). Now we expand upon and revise some of those notions to craft a more formal definition better suited for enabling automatic conformance checking and further tool-support.

Our work on semantic differencing of class diagrams (Maoz et al. 2011b; Nachmann et al. 2022; Ringert et al. 2023), feature diagrams (Drave, Kautz, et al. 2019), component-and-connector architectures (Butting et al. 2017), activity diagrams (Maoz et al. 2011a; Kautz & Rumpe 2018), and statecharts (Drave, Eikermann, et al. 2019) can be seen as preliminary work for this paper. In our approach, we compare two models regarding their semantics and if possible, compute diff witnesses, which are legal instances of the first model that are not permitted by the second. If no such witnesses exist, we can conclude that the first model semantically refines the second.

Conformance checking is a form of comparative model analysis. In software engineering, we usually distinguish between two types of analyses: static and dynamic. For computer programs, the former analyzes the program's code to reason over

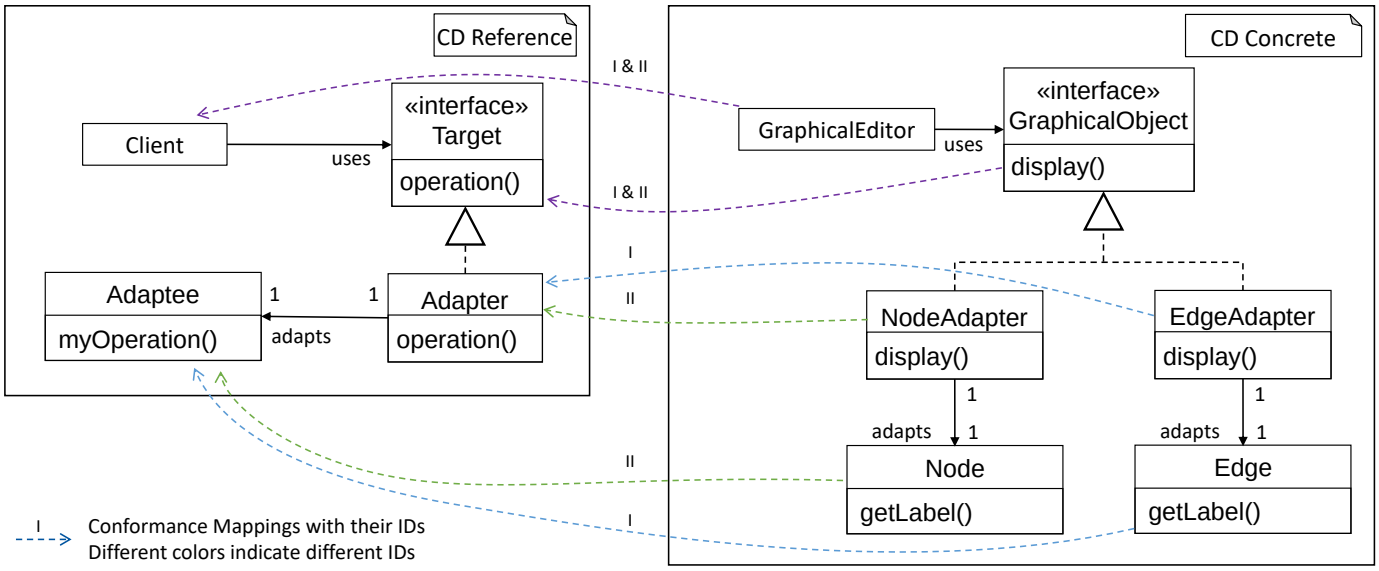

Figure 1 The adapter patterns as reference Class Diagram (CD) model (left) and a concrete model (right) with two incarnation mappings (arrows). The first and second incarnation mappings mark the `NodeAdapter`, respectively the `EdgeAdapter` and adjacent elements as incarnation of the reference model. Both share mapped elements (`GraphicalEditor` and `GraphicalObject`).

all possible behaviors at runtime, while the later executes the program and observes the executions. Analogously, models can be analyzed by considering the semantic implications of their abstract syntax or by executing/instantiating them. Ernst (2003) argues that while these two types of analyses are traditionally viewed as separate domains with different use-cases, they are in fact complementary and can be used in tandem. Parsons & Murphy (2004) combine dynamic and static analysis to in a framework that automatically detects performance antipatterns in a component-based systems. Richner & Ducasse (1999) present an approach for reverse engineering object-oriented applications by utilizing a logic programming language to query both static and dynamic information, and then using that information to create high-level views. Similarly, (Systa 1999) discusses reverse engineering of Java-based software and focuses in particular on the use of static and dynamic analysis to create both static and dynamic views of the software-system and how these views can be connected. In the context of automatic conformance checking, the extraction of views from concrete software-system is useful when no concrete model of a system exists or if it is outdated. The extracted view could then be checked against a given reference model to ensure compliance to software requirements and specifications.

3. Concept and Definition

The concept of a reference model is contingent on its relation to other more concrete models. By itself a reference model is simply a model within a given modeling language that is used to describe domain concepts and their domain-specific relations in an exemplary manner. A reference model may be a solution to a concrete problem that is now used in a different context as a reference for a set of similar, related problems. Alternatively, a reference model may be specifically constructed as a design pattern. Regardless of its origin, its contextual purpose defines a reference model. (Konersmann et al. 2024)

A reference model must in some manner be more abstract than the concrete models derived from it, which we refer to as concretizations: While the latter each describe a specific solution to a problem within some application domain, the former, at least in this context, represents a solution approach for a class of similar problems. Conversely, we might say that a concretization is in some manner a refinement of the reference model, as its domain concepts and domain-specific relations should translate to the derived concrete model. Therefore, in order to develop tools that can automatically check for conformance of a concrete model to a reference model, we must consider some formal notion of model-semantics. Moreover, to ensure that a semantic comparison is always possible, we require that the concrete model is of the same language, as the reference model.

As a general requirement for conformance, we state that the essence or meaning of the reference model must be preserved in the concrete model, or more formally the concrete model must semantically refine the reference model when we consider translated instances of *incarnations* within the semantic domain. Here, the term *incarnation* refers to an element of the concrete model that represents, or we say *incarnates*, a corresponding element of the reference model. In order to translate instances of incarnations correctly, they need to be mapped to their respective reference elements. Accordingly, such mappings are referred to as *incarnation mappings*.

Note the difference between *instance* and *incarnation*: An incarnation is usually the same kind of element as its corresponding reference element, an *instance*, however, is an element within the semantics and therefore not of the same kind, *e.g.,* the incarnation of a class is usually also a class, but the instance

of a class is an object (see Figure 2 for reference).

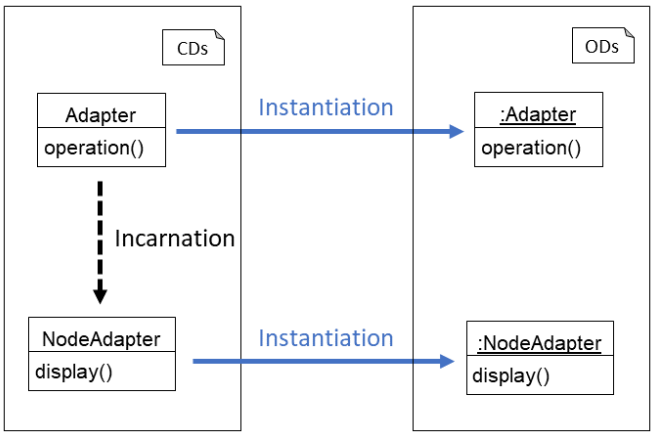

Figure 2 The class `NodeAdapter` incarnates the class `Adapter`. Both are instantiated by corresponding objects.

The general semantic requirement for conformance is visualized in Figure 3. Accordingly, we assume that there is some semantic mapping *sem* that relates models of a given modeling language to corresponding instances within a semantic domain as described by Harel & Rumpe (2004). Since the elements of the concrete model might have different names than the elements of the reference model, a direct comparison of semantics is often not helpful. Instead, we require an *incarnation* mapping M of concrete elements to reference elements, which then allows us to translate the instances of *incarnations*. As this translation depends on both the semantic mapping *sem* and the incarnation mapping M, we denote it as tr_M^{sem}. Analogous to semantic refinement which requires that $sem(A)$ includes $sem(B)$, we require for conformance that $sem(A)$ includes $tr_M^{sem}(B)$. This semantic refinement check is possible using semantic-difference analysis, as discussed in section 2.

3.1. Incarnation Mapping

An incarnation mapping is used to map incarnations in a concrete model to corresponding elements in a reference model. The *kinds* of elements that can be mapped must be determined when constructing the conformance relation and depend both on the modeling language and the use-case. The granularity of this classification into *element-kinds* should at least be fine enough to correspond to the different *kinds* of element-instances within the semantic domain of the modeling language. It is usually advisable to only permit mapping elements of the same kind. However, for certain languages it might be sensible to allow for composition and decomposition of elements when deriving a concretization from a reference models. Here, multiple elements of different kinds might be mapped to one element, and vice versa. This occurs, *e.g.,* when incarnating a class from a reference CD via multiple classes and connecting associations.

To allow for (de-)composition, we define an incarnation mapping as a partial function that maps sets of elements from a concrete model to sets of elements from a reference model. As such, let $ELEM(\mathcal{L})$ be a universe of elements for the language $\mathcal{L}$ and let $elem_{\mathcal{L}} : \mathcal{L} \to 2^{ELEM(\mathcal{L})}$ be a function that returns the set of elements from a given model in the language. We define an incarnation mapping as follows:

Definition 3.1 (Incarnation Mapping: M)**.** Given a modeling language $\mathcal{L}$ and two models $A, B \in \mathcal{L}$, an incarnation mapping from B to A is a partial function $M : 2^{elem_{\mathcal{L}}(B)} \rightharpoonup 2^{elem_{\mathcal{L}}(A)}$ such that for all sets of concrete elements $I \subseteq elem_{\mathcal{L}}(B)$ we have $I \neq \emptyset \implies M(I) \neq \emptyset$ and $M(\emptyset) = \bot$.

3.2. Conformance Relation

A conformance relation relates concretizations to their corresponding reference models in a reflexive and transitive manner. Any conformance relation must adhere to the general semantic requirements for conformance, but is otherwise specific to a modeling language and use-case / application domain. It also requires an incarnation mapping as an additional parameter. While we could simply consider the existence of some mapping that induces conformance, such a mapping might not always be sensible. For tooling purposes, it is therefore also generally preferable to have a mapping encoded by a domain expert to ensure practical relevance.

Let us now present a more formal definition for a conformance relation of a given modeling language $\mathcal{L}$. We start by defining a simple notion of conformance that takes as input only a single incarnation mapping and then extend the definition to a non-empty set of incarnation mappings.

Definition 3.2 (Conformance Relation: *cf*)**.** Let $\mathcal{L}$ be a modeling language with a corresponding semantic domain D and a semantic mapping $sem : \mathcal{L} \to D$. Furthermore, for any incarnation mapping M, let $tr_M^{sem} : D \to D$ be a transformation that renames instances within the semantic domain according to M.

Given a universe of incarnation mappings $MAP(\mathcal{L})$ over the language $\mathcal{L}$, we say that $cf \subseteq \mathcal{L} \times MAP(\mathcal{L}) \times \mathcal{L}$ is a *conformance relation* if and only if for all models $A, B, C \in \mathcal{L}$ with corresponding incarnation mappings M_1 from B to A and M_2 from C to B the following properties hold:

1. $cf(B, M_1, A) \implies tr_{M_1}^{sem}(sem(B)) \subseteq sem(A)$ (semantic refinement),
2. $cf(A, id_A, A)$ for the identity mapping id_A on A (reflexivity),
3. $cf(C, M_2, B) \wedge cf(B, M_1, A) \iff cf(C, M_2 \circ M_1, A)$ (transitivity).

If the conformance relation is clear from context, we may also write $B \longmapsto_M A$ instead of $cf(B, M, A)$. Moreover, we can extend the relation to sets of incarnation mappings such that for any $\mathcal{M} \subseteq MAP(L)$ we define:

$$B \longmapsto_{\mathcal{M}} A :\iff \forall M \in \mathcal{M} : B \longmapsto_M A$$

Note that the first property in this definition corresponds to the general semantic requirements as depicted in Figure 3, while the second and third property in the definition ensure reflexivity and transitivity, respectively. Choosing equivalence for the first property, would have rendered the second and third redundant.

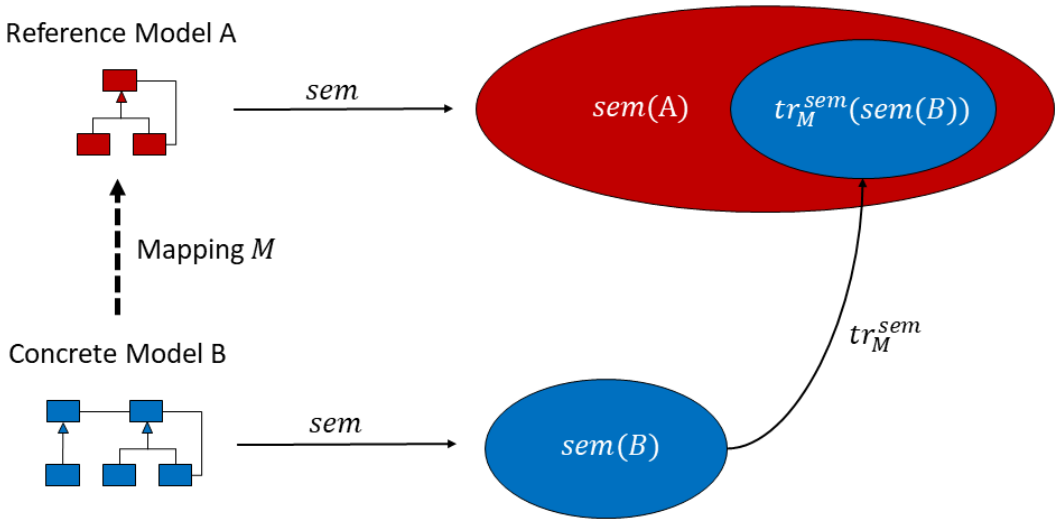

Figure 3 Given a reference model A, a concrete model B, and an incarnation mapping M, as well as a corresponding translation of semantic elements tr_M^{sem}, the semantic requirement for conformance is that: $tr_M^{sem}(sem(B)) \subseteq sem(A)$.

However, we chose an implication for the first property, to allow for custom conformance relations that fit additional requirements regarding the language and use-case. Alternatively, we could have required a more use-case specific definition of formal semantics, but from our experience (Lindt et al. 2023), this has several disadvantages:

1. We sacrifice a more general notion of semantics for a given language, reducing comparability across use-cases.
2. It is rather difficult to construct a precise semantics that encompasses all relevant aspects of a given use-case.
3. The resulting semantics would usually be fairly complex and cumbersome to deal with.
4. It would also put an unnecessarily heavy burden of proof on the developers of conformance checking tools.

For the sake of completeness, we also provide a formal definition of a reference model and a concretization:

Definition 3.3 (Reference Model)**.** Given models $A, B \in \mathcal{L}$ with an incarnation mapping M from B to A such that $B \longmapsto_M A$, the model A is called a *reference model*.

Definition 3.4 (Concretization)**.** Given models $A, B \in \mathcal{L}$ with an incarnation mapping M from B to A such that $B \longmapsto_M A$, the model B is called a *concretization* of A.

4. Application of Conformance Checking

In this section we apply the general concept of conformance from section 3 to three modeling languages and demonstrate the developed tools. As modeling languages we have chosen CDs, FDs, and statecharts. For one, these modeling languages are widely known and used in practice. Furthermore, they describe different concepts and are used to demonstrate different cases of conformance checking. (1) CDs describe data-structures. Mapping and conformance check are directly implemented in Java. (2) FDs model product-lines and configuration options. The conformance checker utilizes a translation to logic formulas and employs a custom mapping language that allows for (de-)composition of features during incarnation. (3) Statecharts model the behavior of systems. The approach uses an encoding for incarnations that permits the mapping of complex expressions such as string-comparison. An SMT-Solver is then used to check conformance for statecharts.

4.1. Class Diagram (CD)

CDs are the most widely employed Unified Modeling Language (UML) models in industry and research (Dobing & Parsons 2006; Langer et al. 2014; Hutchinson et al. 2011). They are used to model data-structures and object-oriented software-systems. A CD contains type-declarations and defines associations between types. For our purposes, we consider the UML/P variant of CDs (Rumpe 2016). As such, we differentiate between three kinds of types: classes, interfaces, and enumerations. A type always has a name. Classes may contain members, *i.e.,* attributes and method-signatures, and each enumeration defines a set of constants. An association always connects two types. It can have a name and a direction. Each side of an association has a role-name, and may also have a cardinality. Compositions exist as a special form of associations. For our purposes, we consider the formal semantics of a CD to be the set of its valid instances, *i.e.,* the object structures it permits (Maoz et al. 2011b).

Well-known examples for reference CDs are the design patterns introduced in the eponymous book by Gamma et al. (1997). Concretizations of these patterns are usually integrated into larger systems. We require that each concretization conforms independently to its corresponding reference CD.

Example Consider the example CDs in Figure 1. The adapter pattern modeled in the reference CD on the left is incarnated twice within the CD on the right. For this purpose we use two injective incarnation mappings. Note that the class `GraphicalObject` incarnates the reference class

`Target` in the context of both mappings. On the other hand, the class `Adapter` is incarnated by both `NodeAdapter` and `EdgeAdapter`, each having a directed association with the role-name `adapts` to an incarnation of the reference class `Adaptee`, named `Node` and `Edge` respectively. Note that `NodeAdapter` and `Node` are mapped to their respective reference classes within the same incarnation mapping. Thus, incarnation mapping I by itself produces a valid concretization of the reference diagram. The same goes for incarnation mapping II.

Approach As opposed to a closed-world assumption on semantics which forbids instances of elements that are not explicitly modeled in the CD, our approach necessitates an open-world assumption that considers the reference CD to be under-specified (Nachmann et al. 2022; Ringert et al. 2023). This is because the concrete model should be permitted to contain additional elements that are not incarnations of reference elements within the context of an incarnation mapping.

This assumption complicates the semantic comparison somewhat, as we now have to consider instances of elements not modeled in the CD as part of its semantics. However, to ensure semantic refinement after translation, it is sufficient to check for each mapping if the concrete CD is an expansion (or syntactic refinement) of the reference CD when translated (Fahrenberg et al. 2014; Lindt et al. 2023).

In this manner, conformance checking for CDs can be reduced to a structural analysis: We first check for each element of the reference CD if there is a corresponding element of the same kind in the concrete CD that claims to be its incarnation. Then we check conformance by ensuring that its properties are preserved or "syntactically refined", and related elements in the reference CD have corresponding incarnations in the concrete CD. A class in the reference CD must be incarnated by a class in the concrete CD, such that its incarnation contains incarnation of all of its members, extends/implements incarnations of all of its super-types, and connects to incarnations of all of its associations. By default inherited incarnations of attributes and associations are considered, as well. Interfaces and enumerations are checked analogously. The incarnation of an association must connect incarnations of its types, but also preserve navigability and defined cardinalities. For each attribute in the reference CD, we require that its incarnation must be contained in or inherited by the incarnation of its class. Furthermore, the type of the incarnation has to be compatible with the type of the reference attribute.

Implementation We have implemented this approach as part of the `CD4Analysis` project which is publicly available on GitHub[3]. In order to map reference elements to their corresponding incarnations, we use the stereotype feature (Hölldobler et al. 2021; Gogolla & Henderson-Sellers 2002) which is included in the `CD4Analysis` grammar. Additionally, we also allow mapping by name as a fall-back. More specifically, when searching for incarnations of a specific kind, we use a combined matching strategy that is composed of individual matching strategies which all implement the same interface. The interface is displayed in Listing 1.

[3] https://github.com/MontiCore/cd4analysis

```
public interface MatchingStrategy<T> {
  List<T> getMatchedElements(T srcElem);
  boolean isMatched(T srcElem, T tgtElem);
}
```

Listing 1 Matching strategy interface

The combined strategy for classes and attributes first searches for a concrete element with a corresponding stereotype and afterward considers its name. The combined strategy for associations additionally considers the source type and target role-name of directed associations, when no concrete association with a corresponding stereotype or name was found. In summary the implementation works as follows:

1. Check if each reference element is incarnated:
   (a) Check if a concrete element has a corresponding stereotype,
   (b) else, check if a concrete element has the same name
   (c) else, check for associations if a concrete element has a matching source type and target role-name in a navigable direction
2. Check each incarnation for conformance:
   (a) Check if the properties of the reference element are preserved or refined.
   (b) Check if related elements include necessary incarnations.

After building the project, the CD conformance check can be performed by executing the `MCCD.jar` as seen in Listing 2.

```
java -jar cdtool/target/libs/MCCD.jar              \
              -i "doc/GraphAdapter.cd"             \
              --reference "doc/Adapter.cd"         \
              --map "m1" "m2"
```

Listing 2 Executing the CD conformance checker

Evaluation To evaluate our implementation, we translated the example from Figure 1 into the textual format of `CD4Analysis` using stereotypes to encode the two mappings. Here, we only need to map the types and methods explicitly, as the tool is able to match the associations of the reference CD to their incarnations based on the source type and target role-name. In a second test, we mapped `Node` to `Adaptee` in both mappings:

```
<<m1="Adaptee", m2="Adaptee">> class Node {
   <<m1="myOperation", m2="myOperation">>
   void getLabel();
}
```

Listing 3 Modified mapping: `Node` is now mapped to `Adaptee` in both incarnation mappings.

```
===== Check if GraphAdapterF conforms to Adapter
    with respect to m1 =====
===== CONFORM =====
===== Check if GraphAdapterF conforms to Adapter
    with respect to m2 =====
Node is not a valid incarnation of Adaptee
Incarnations of associations are missing or
    incorrect!
association [1]EdgeAdapter->(adapts)Edge [1];
is not a valid incarnation of
association [1]Adapter->(adapts)Adaptee [1];
===== NOT CONFORM =====
```

Listing 4 Tool output for non-conformance

Shown in Listing 4 is the tool's output on the console informing us that `Node` is not a valid incarnation concerning the second mapping, as the incoming association is missing or incorrect. This happens because the source-class `NodeAdapter` is now no longer mapped to `Adapter` by the second mapping.

Note that concerning this example it is possible to use a single non-injective incarnation mapping as seen in Listing 5.

```
classdiagram Concrete {
  <<m="Client">> class GraphicalEditor;

  <<m="Target">> interface GraphicalObject {
    <<m="operation">> void display();
  }
  association GraphicalEditor
                      -> (uses) GraphicalObject;

  <<m="Adapter">> class NodeAdapter
                    implements GraphicalObject;

  <<m="Adaptee">> class Node{
    <<m="myOperation">> void getLabel();
  }
  association [1] NodeAdapter -> (adapts) Node [1];

  <<m="Adapter">> class EdgeAdapter
                    implements GraphicalObject;

  <<m="Adaptee">> class Edge{
    <<m="myOperation">> void getLabel();
  }
  association [1] EdgeAdapter -> (adapts) Edge [1];
}
```

Listing 5 Concrete CD with one incarnation mapping

While this is more convenient, it also somewhat reduces the strictness of the conformance check, as we only require that each incarnation of a reference element must also contain or connect to at least one incarnations of each related element in the reference CD. Still, we found it necessary to include this option for the sake of practicality, as the number of needed injective mappings might potentially increase exponentially with size of the concrete CD.

Outlook It is currently not possible for a concrete element to be an incarnation of multiple reference elements within the same mapping. However, we might consider composition and decomposition of reference elements during incarnation, in the future. For instance, an attribute in the reference CD might be split into multiple concrete attributes or multiple reference classes might be incarnated by one concrete class. Based on the concept of 150% models (Grönniger et al. 2008), we might also consider conformance to 150% reference CDs from which only a subset of elements need to be instantiated. Note that we would still operate under an open-world assumption with respect to the semantics of the reference model. However, we would need to consider the semantics of the concrete model under a closed-world assumption. This is because the optional elements of the reference CD must not be permitted to be instantiated within the semantics of the concrete CD unless they were incarnated. A similar approach for a mixed assumption on semantics was introduced in Ringert et al. (2023) for the purpose of semantic differencing.

4.2. Feature Diagram (FD)

FDs are used to model product-lines and configuration options (Grönniger et al. 2008). An FD is a directed tree that describes the dependency between individual features. A child feature always depends on the parent feature. The potential sets of child-features depends on the kind of edge used. An edge can be *mandatory* or *optional*, and edges can be bundled into an *inclusive* or *exclusive* selection. Furthermore, features can *require* or *exclude* other features. The semantics of an FD is the set of possible feature configurations (Schobbens et al. 2006).

Example An FD might be used to, *e.g.,* model the product-line for a car. When designing a new product-line for a different model of car, the original FD can be used as a reference. To conform to the reference FD, we have to ensure that for each valid feature configuration of the new model, the set of incarnated reference features corresponds to a valid configuration in the old model, *i.e.,* the inter-dependency of the reference features must be preserved by their incarnations. Consider, *e.g.,* the root feature `CoolCar2` of the concrete FD on the right side of Figure 4. It is mapped to the root feature `CoolCar1` of the reference FD on the left. The reference features `Engine` and `Gasoline` are mapped to incarnations of the same name. Moreover, we can see that the features `NavigationsSystem` and `Radio` compose an incarnation of the optional reference feature `InfotainmentSystem`. Note that the concrete FD contains additional features that are not incarnations of any features in the reference FD. This includes the added features `ComfortFunctionalities` and `AirConditioning`, and the `Engine` of `CoolCar2`, that can also be `Electric` or `Hybrid` as opposed to the `Engine` of `CoolCar1`, which is only available in the variant `Gasoline`.

Approach FDs can be translated into propositional formulas with each feature corresponding to a Boolean variable (Durán Toro et al. 2017). A child feature always implies its parent feature. A parent feature also implies mandatory child features. Inclusive and exclusive selections of child features are translated using corresponding disjunctions. Similarly, a *requires*-relations is translated into an implication, and an *excludes*-relations implies the negation of the targeted feature.

Now conformance checking can be reduced to a SAT-problem in the following manner: First, translate the concrete FD into a logical formula, then do the same for the reference FD and negate the resulting formula. Next, translate the incarnation mapping into a set of implications, where each incarnation implies its reference feature. If we conjugate all the resulting formulas, any valid variable assignment corresponds to a feature configuration in the semantics of the concrete FD with an invalid set of incarnations, *i.e.,* a diff-witness (Drave, Kautz, et al. 2019) that demonstrates non-conformance. On the other hand, if no valid assignment exists, then the concrete model conforms.

Implementation For our implementation, we have utilized the Java-library of the SMT-solver Z3[4]. We use the textual `feature-diagram` language of the `MontiCore` language-family, as well as a custom mapping language, also developed

[4] https://github.com/Z3Prover/z3

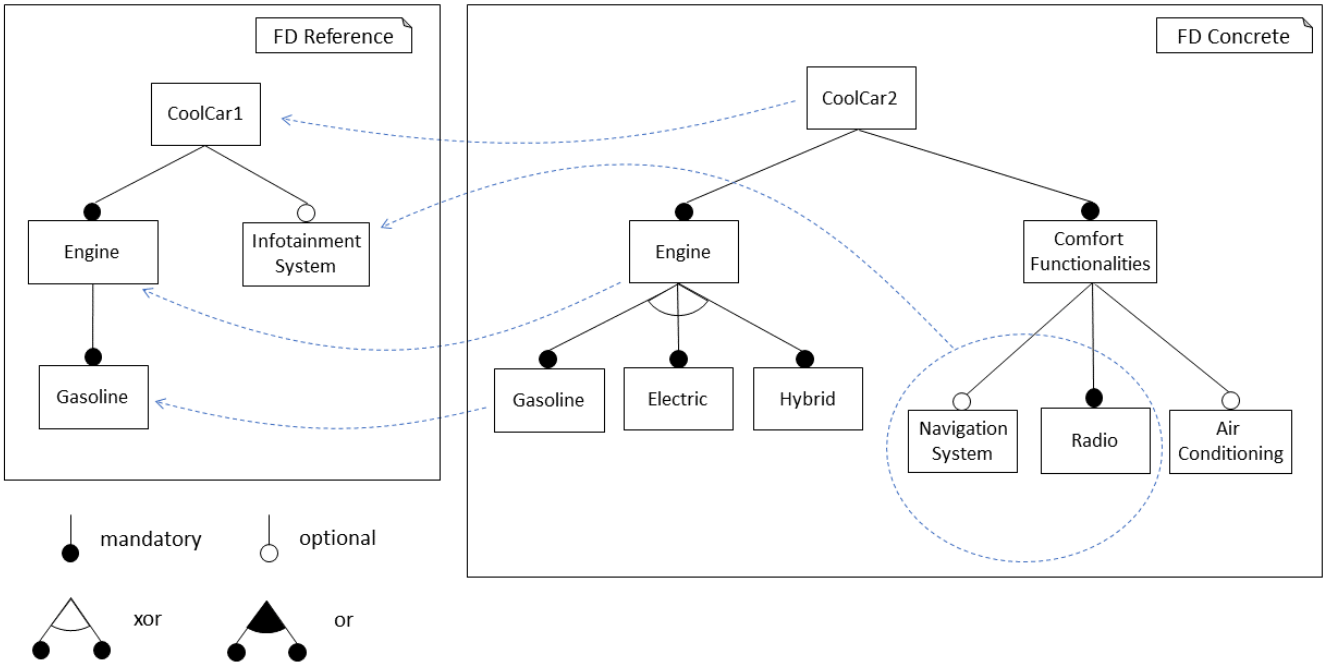

Figure 4 Feature Diagram conformance example

using `MontiCore`, that allows for composing and decomposing incarnations of reference features. Similar to our approach for CDs, we match features by name as a fall-back option. The tool has been integrated into the `feature-diagram` project which is publicly available on GitHub[5]. After building the project, the FD conformance check can be performed by executing the `FDConf.jar` as seen in Listing 6.

```
java -jar fd-conformance/target/libs/FDConf.jar     \
              --reference "doc/Reference.fd"        \
              --concrete "doc/Concrete.fd"          \
              --map "doc/Mapping.map"
```

Listing 6 Executing the FD conformance checker

Evaluation We evaluated the tool using the example from Figure 4. The FDs were translated to the textual syntax of `MontiCore`'s `feature-diagram` language and the mapping encoded in our custom mapping language:

```
1 mapping Mapping {
2   CoolCar2 ==> CoolCar1;
3   NavigationSystem ++ Radio ==> InfotainmentSystem;
4 }
```

Listing 7 Textual version of the mapping in Figure 4

When checking for conformance, the tool informs us that the concrete FD does in fact not conform to the reference FD concerning the provided incarnation mapping, as can seen from the console output displayed in Listing 8.

To fix this issue it is necessary to also map the features `Hybrid` and `Electric` to the reference feature `Gasoline`.

[5] https://github.com/MontiCore/feature-diagram

This is sensible as the reference FD models the previous product-line with the reference feature `Gasoline` being the only available engine type. Our new product-line, however, has access to alternative engine types: `Hybrid` and `Electric`. Since they also represent engine types, we can map them as incarnations of the engine type `Gasoline` from the reference FD. Note that we now also have to explicitly map the concrete feature `Gasoline`. The updated mapping is displayed in Listing 9.

```
1 ===== Check if Concrete conforms to Reference with
      respect to MapV1 =====
2 ===== NOT CONFORM =====
3 Concrete Configuration: [ComfortFunctions, Hybrid,
      CoolCar2, Radio, Engine] is valid.
4 Reference Configuration: [CoolCar1, Engine] is NOT
      allowed!
```

Listing 8 Tool output for non-conformance

```
1 mapping MappingV2 {
2   CoolCar2 ==> CoolCar1;
3   NavigationSystem ++ Radio ==> InfotainmentSystem;
4   Gasoline ==> Gasoline;
5   Electric ==> Gasoline;
6   Hybrid   ==> Gasoline;
7 }
```

Listing 9 Updated mapping for the FDs in Figure 4

Outlook One feature of the `feature-diagram` language we do not yet support is cardinalities for inclusive disjunction. This is because they are not a standard feature of FDs and the concept of allowing a specific range of multiplicities is not easily translated to SMT. Additionally, our current notion of FD-conformance is also agnostic to the structure of the FDs and

only concerned with the formal semantics, *i.e.,* the set of permitted feature configuration. However, a structural check is worth considering for the future as it could prohibit nonsensically-structured concretization of well-structured reference FDs

4.3. Statechart

As our final application example, we present a conformance checker for statecharts (Harel 1987; OMG 2017). Statecharts are widely used to describe behavior in an accessible form. They are applied in various domains, from rocket-science (Harris & Patterson-Hine 2013; Alves et al. 2011) to interactions of the stock market (Borshchev 2014; Dumas et al. 2002).

Statechart differ from the languages in the previous two sections as they describe behavior. For the textual notation of a statechart we use MontiArcAutomaton (Haber 2016; Ringert et al. 2013) and for its semantics we refer to I/O automata (Drave, Eikermann, et al. 2019; Rumpe 1996).

Example As a running example to demonstrate the statechart conformance checker, we use a login system. The reference model is shown on the left side of Figure 5. This abstract model shows an exemplary system with a simple login handling. Only when a user is `LoggedIn` the input `Actions` lead to a `Response`. When a user is `LoggedOut`, the input `Action` leads to an `Error`. This statechart contains enough information to prove properties over the behavior. Mainly that the system only responds after login.

Using this statechart as a reference, we now develop a concrete system with more information, as seen on the right side of Figure 5. In the concrete system the `Login` is performed with a `password` of type `String`. The internal state contains a number named `counter`. This `counter` can be increased with the input-action `incCounter`. With `getCounter` the current value of the counter is returned as response.

Refinement The transition function of a statechart X can be mathematically described as a predicate

$$trans_X : state \rightarrow input \rightarrow state \rightarrow output \rightarrow \mathbb{B}$$

The first parameter is the state from which the transition starts, the next parameter is the input message. The third parameter is the target-state of the transition and the fourth parameter is the output produced. The predicate returns true iff the transition is possible in the statechart. We omit partiality considerations in this paper and instead refer to Rumpe (1996).

For refinement of statecharts, it is sufficient to show that (Rumpe 1996):

$$\begin{aligned}&\forall state\ input\ nextState\ output\ .\\ &\quad trans_{Refined}(state, input, nextState, output)\\ &\quad \implies trans_{Original}(state, input, nextState, output)\end{aligned}$$

Every transition possible in the refined statechart is also allowed in the original statechart. Semantic refinement of statecharts allows proving properties over the more abstract statechart which directly hold over the refined statechart. This general concept is described in more detail in Broy & Stølen (2001).

Mapping To check conformance we additionally define mappings between concrete and reference statechart. Since there are the three data types `input`, `output` and `state`, we use three mappings instead of one.

1. $mapInput$: $concInput \rightarrow refInput$ Map from concrete input to reference input.
2. $mapOutput$: $concOutput \rightarrow refOutput$ Map from concrete output to reference output.
3. $mapState$: $concState \rightarrow refState$ Map from concrete state to reference state.

These three mapping can also be combined into one mapping using a sum-type $\oplus$ (also called tagged union). For example with $unionMap$: $(concInput \oplus concOutput \oplus concState) \rightarrow (refInput \oplus refOutput \oplus refState)$. However, for clarity we use the three separate definitions instead of the unified definition.

Textually, the mapping is defined using a similar mapping language as used for FDs. However, the expression language is more powerful, *e.g.,* supporting string comparison such as `password == "correct"`. Other operators like `&&`, `!`, `+`, `-`, `%` are also allowed. For the example from above, we define the mapping as shown in Listing 10.

```
1  stateMap {    // Definition of State Mapping
2   Concrete.state == Known
3           ==> Reference.state == LoggedIn;
4   Concrete.state == Anon
5           ==> Reference.state == LoggedOut;
6  }
7  inputMap {    // Definition of Input Mapping
8   Concrete.input == Input.GET_VALUE
9           ==> Reference.input == Input.ACTION;
10  Concrete.password == "correct"
11          ==> Reference.input == Input.LOGIN;
12 [...]
```

Listing 10 Textual mapping for the statechart in Figure 5

Conformance Combining the mapping with semantic refinement leads to the following formula:

$$\begin{aligned}&\forall state\ input\ output\ nextState.\\ &\quad trans_{Concrete}(state, input, nextState, output)\\ &\implies trans_{Reference}(mapState\ state, mapInput\ input,\\ &\qquad mapState\ nextState, mapOutput\ output)\end{aligned}$$

When this formula holds, the mappings are correct and the two statecharts are in a conformance relation. To check this property we use the Java API of the SMT-Solver Z3. To get more detailed output and increase performance the formula from above is split up into multiple SMT-checks. For each transition in the reference model, *e.g.,* the transition from `Logged-In` to `Logged-In` with input `Action` and output `Response` (see Figure 5), a separate SMT problem is created. This way each SMT problem is smaller and thus more easily solvable, and we get more information which can be used in the output.

The tool is part of the MontiArc project which is publicly available on Github[6]. After building the project, the check can be performed by executing the `SCConformance.jar` as seen in Listing 11.

[6] https://github.com/MontiCore/montiarc

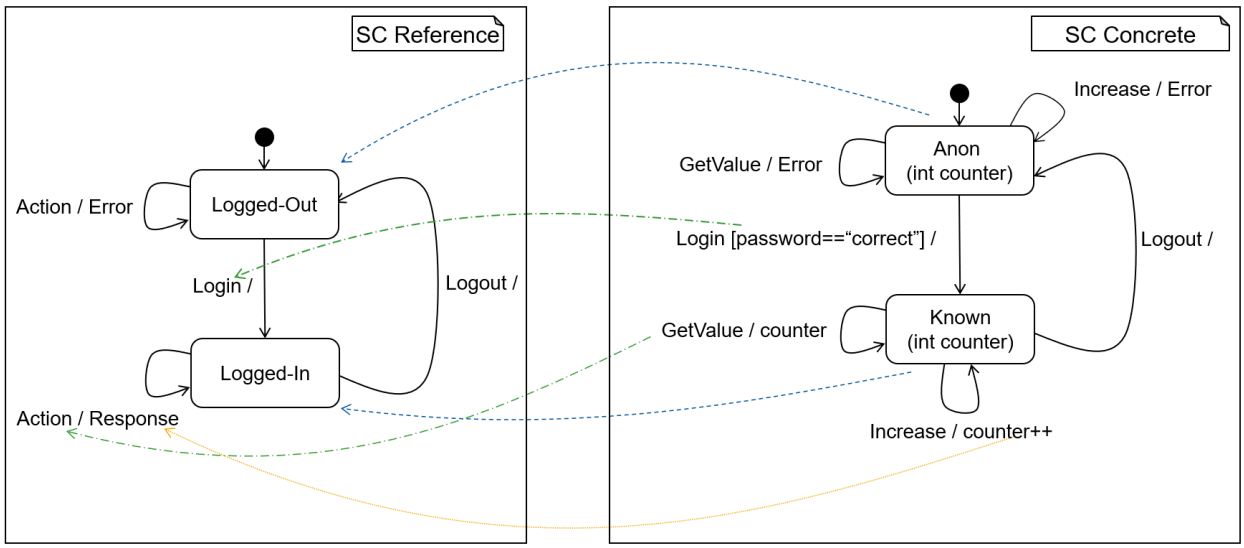

Figure 5 Statechart conformance example

```
java -jar SCConformance.jar                                    \
     --reference "Reference.arc" "Reference.cd"    \
     --concrete "Concrete.arc" "Concrete.cd"       \
     --map "Mapping.map"
```

Listing 11 Executing the statechart conformance checker

Output When the models conform, the output of the tool is a single line confirming this. However, in case of non-conformance, the tool provides sufficient information to the developer on why they do not conform. As the problem is split into multiple SMT-calls, we know which transition in the reference model is responsible. An example for input, state, and output which lead to non-conformance are displayed.

To demonstrate this, the concrete model from Figure 5 is modified. We add a transition from the state `Anon` to the state `Known` with the action `GetValue`. This behavior is forbidden in the reference statechart, since every action in the `Logged-Out` state leads again to the `Logged-Out` state. Applying the conformance checker now leads to the following output:

```
[WARN]  Transition is *NOT* conform!
Concrete.arc:<28,2>: Anon->Known[input==Input.
    GET_VALUE]/{};

Possible Transition in Concrete Model(Concrete):
    From State:   Anon{counter=2}
    With Input:   input=GET_VALUE
    To State:     Known{counter=2}
    Output:       output=[ ] value=[ ]

Impossible Transition in Reference Model(Reference):
    From State:   NotLoggedIn
    With Input:   input=ACTION
    To State:     LoggedIn
    Output:       output=[ ]
```

Listing 12 Tool output for non-conformance

In the first lines the file (`Concrete.arc`) and position (`<28,2>`) of the non-conform concrete transition are shown. Furthermore, the transition definition is printed so that the developer does not need to open the file. After that, example input for a possible transition in the concrete model are printed. Such a transition is not possible in the reference model.

This verbose output helps the developer to quickly understand and fix the source of non-conformance. However, there are certain assumptions. Firstly, the error is assumed to be in the concrete model and not the reference model. This is compatible with the normal process, where the reference model is developed before the concrete model. However, there are cases where non-conformance would be fixed by changing the reference model. For example, when new corner-cases are discovered while using the reference model on concrete use-cases. This case is the reason why the “Impossible Transition in Reference Model” is part of the output. The fix might be to add such a transition to the reference model. Secondly, the mapping may be wrong. This can be especially hard to find out, since the mapping is only visible in the combination of all printed values.

Outlook The tool can currently only check conformance between two statecharts. Additionally, MontiArc supports the composition of components. The behavior of this composition is defined using FOCUS (Broy & Stølen 2001). Checking conformance of a network of components is left as future work.

5. Discussion

In the following we discuss both the general aspects of conformance as well as the language-specific considerations for CDs, FDs, and statecharts. We examine commonalities and differences found in our conformance checking approaches as well as further utilization of reference models and conformance checking in MDE.

5.1. General Aspects of Conformance

Our definition of reference models and conformance might be more restrictive than similar notions found in existing literature. However, we thereby guarantee semantic refinement of the

reference model in its concretization and thus preservation of desired properties. Furthermore, this approach enables better tool-support and comparability across different languages and domains. While conformance across different languages could be permitted as long as their semantics are comparable, a simple translation of the reference model should then also be possible.

Nonetheless, we still allow for a great degree of flexibility: Our notion of an incarnation mapping maps sets of reference elements to sets of concrete elements and thus permits (de-)composition during incarnation. Of course conformance might be trivialized if a nonsensical mapping is chosen, but we do not consider this to be a major issue, as these mappings should always be constructed by someone with appropriate domain expertise. The conformance relation itself is also customizable as long as it still adheres to the basic requirement of semantic refinement. This not only permits adjusting a conformance relation to fit domain- or language-specific requirements not easily captured by a formal semantics definition but often allows these stricter conformance checks to scale better than standard semantic difference analysis. As our application case for CDs has demonstrated, it might be both sufficient and prudent to only check for specific syntactic properties which then imply a formal semantic refinement. We believe that this aspect is especially useful when considering more complex and semantically richer modeling languages, as we might be able to reduce the complexity of conformance checking by relying on sufficient and easy-to-check criteria for semantic refinement.

5.2. Language-Specific Considerations

We now compare the three conformance checking approaches and consider their commonalities and differences. Let us first review the most distinctive characteristics of each approach:

Conformance checking for CDs reduces the problem to a syntactic analysis that compares the elements in the concrete CD with its mapped counterparts in the reference CD, determining whether they are valid incarnations based on their properties and related elements. Incarnation mappings for CDs are encoded using stereotypes. This is sufficient, as we currently do not consider (de-)composition during incarnation. Matching by name is used as a fall-back. Our approach operates under an open-world assumption regarding CD semantics and exploits a notion of expansion or syntactic refinement in order to guarantee semantic refinement. As it enables refinement checking, albeit with some degree of incompleteness, it is complementary to the semantic differencing operator by Ringert et al. (2023): the former checking a sufficient condition for refinement, the latter detecting diff-witnesses within a given scope that serve as a conclusive counter-example.

Conformance checking for FDs is translated to a SAT-problem and can be considered an extension of the open-world semantic differencing operator introduced by Drave, Kautz, et al. (2019). Incarnation mappings for FDs use a custom mapping language that allows for (de-)composition of features during incarnation. Matching by name is used as fall-back.

Conformance checking for statecharts utilizes an SMT-solver to translate expressions that map states, input, and output from the concrete to the reference model. Incarnation mappings for statecharts use a custom mapping language similar to that used for FDs, although more powerful, as it can map a variety of expressions that are then interpreted by an SMT-solver. However, (de-)composition and a name-based matching as a fall-back are currently not supported. Our notion of conformance is considerably stricter than the semantic requirement outlined in Definition 3.2, as we not only consider the automaton's behavior as a whole (black box view), but also each individual transition (glass box view). This both simplifies the check and at the same time allows for handling a more expressive subset of statecharts than previous semantic differencing operators (Butting et al. 2017; Drave, Eikermann, et al. 2019). For example, infinite state and message spaces are allowed. This was demonstrated in the example use case (see Figure 5) with the natural number `counter` and the string `password`.

We observe that currently only the conformance checker for FDs permits (de-)composition of incarnations, owing to the simplicity of the language. Moreover, the conformance relation is equivalent to the semantic requirement, thus allowing for complete refinement checking. However, as mentioned before, this permits the modeling of structurally-nonsensical concretizations. To mitigate this issue, it might be reasonable to add additional structural requirements to conformance even if they do not affect the formal semantics, *e.g.,* requiring a partial ordering of incarnations within the tree structure of the concrete FD that corresponds to the tree-structure of the reference FD.

Next in terms of complexity is the conformance checker for CDs. By making the conformance relation slightly stricter than the formal semantics, we are able to avoid bounded model checking and additionally consider aspects of CDs not captured by the semantics that are relevant for code-generation and dynamic aspects of programs (Lindt et al. 2023). (De-)composition is more difficult to handle here, as (1) a CD element can have multiple instances within an individual object-structure and (2) different kinds of elements would need to be combined to or derived from a single element. For example, if a class is split into two classes during incarnation these classes need to be connected via association or an inheritance relation. Finally, statecharts are arguably the most complex and expressive out of the three languages. Accordingly, we use the most complex form of mapping and consider a comparatively strict glass-box view for the purpose of conformance. Since we can map complex expressions, name-based matching as a fall-back is not straightforward to implement and would require dedicated effort.

A general commonality found among our approaches for conformance checking is a notion of under-specification or open-world assumption regarding the semantics of the reference model. The examples that we chose all provide reference models that constitute the *minimal* requirements for a concrete system, which the concrete model may extend/expand. While it is also possible for reference models to describe more than the strictly necessary aspects of a system, concrete models might still contain yet more elements describing additional functionalities or structures. The observed necessity of an open-world assumption fits well with our understanding of semantic refinement of models in the early design phases of development (Ringert et al. 2023), as the concretization of reference models is ultimately

a variant of such refinement. As such, we argue that an open-world approach would also be appropriate for other kinds of reference models not discussed in this paper, such as reference architecture model: *e.g.,* a three-layered reference architecture model for business information systems requires at least three layers (UI, business logic, persistence) in its concretizations. However, other layers may also exist.

In general, we found that despite the commonalities regarding semantics, the conformance relations and conformance checking approach need to be defined and implemented in a language-specific manner. This language and use-case specific customization of the conformance relation often simplifies the operation when compared to traditional refinement checks and semantic differencing, and thus allows handling more expressive variants of modeling languages without sacrificing the generality of the underlying semantics definition. We therefore believe that designing conformance checkers in a modular and extensible manner is vital, as it allows for better configuration of the conformance relation to fit a specific use-case. This can be done by, *e.g.,*, utilizing the strategy pattern (Gamma et al. 1997).

5.3. Automatically Deriving Concrete Models and Artefacts

In this paper, we have focused mostly on the design and compliance aspects of concrete systems. However, the aspect of implementation is also of interest to us. If a reference model additionally provides a handwritten reference implementation, we believe that it should be possible, at least in some cases, to automatically adapt this code for a concrete system given a corresponding concrete model with appropriate incarnation mappings that then conforms to the reference model. Of particular interest are CDs, as they are well-suited for code generation (Rumpe 2017) and extension by handwritten code artifacts . Given a concrete model, a reference model and corresponding incarnation mappings, a conformance checker should then be able to decide whether the handwritten reference code can be adapted for the concrete system or not. To adapt the reference code, any tool would then need to compute the inverse of the incarnation mappings in order to translate the reference code to a concrete context. If the inverse mappings are known and the translation rules well-defined, existing code refactoring tools or transformation languages can then be used to execute that code adaption. This necessitates that the conformance relation is designed with additional implementation-specific aspects in mind, as static semantics—such as the one we employ for CDs—cannot capture dynamic aspects of software systems. But even in the case of modeling languages with behavioral semantics such as MontiArc Statecharts, certain aspects of the target programming language must be taken into consideration to define a use-case appropriate conformance relation. Again, this is why we recommend designing modular and extensible conformance checkers to allow for use-case specific configuration. Note that this adaptation need not be limited to code, but could also be used for other types of related artifacts or attached information, *e.g.,* OCL-constraints. Moreover, adaptation and *merging* (Lindt et al. 2023) can be applied to the reference model itself in order to complete a concrete model.

6. Conclusion

While reference models are broadly used in practice, they are mostly used informally. This informal usage may lead to misunderstandings when trying to determine the meaning of a reference model and its implication on concrete systems, as well as manual errors when checking conformance. To avoid these issues and enable better tool support, we propose a formal concept of a conformance relation that is based on semantic refinement but otherwise language and use-case specific. In this paper, we presented concepts and tools for automated conformance checking of concrete models against reference models for CDs, FDs, and statecharts. Moreover, we discussed the commonalities and differences of the implemented conformance checks and their implications on conformance checking in general. We argue that our approach lays the foundation for automated, tool-based checking of conformance relations and efficient reuse of information attached to reference models.

For the future, we plan to develop conformance checking approaches for process and architectural models, and explore the assisted derivation of concrete models from reference models combined with reuse and automated adaptation of reference implementations as well as other types of related artifacts. Additionally, we will apply the general conformance-notion to anti-patterns, which forbid certain concrete elements.

Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 250902306

References

Alves, M. C. B., Drusinsky, D., Michael, J. B., & Shing, M.-t. (2011). Formal validation and verification of space flight software using statechart-assertions and runtime execution monitoring. In *2011 6th International Conference on System of Systems Engineering* (pp. 155–160).

Arora, S.-J., Ceccolini, C., & Rabe, M. (2022). Approach to Reference Models for Building Performance Simulation. In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development*. SCITEPRESS - Science and Technology Publications. doi: 10.5220/0010888800003119

Bauer, M., Bui, N., De Loof, J., Magerkurth, C., Nettsträter, A., Stefa, J., & Walewski, J. W. (2013). IoT Reference Model. In A. Bassi et al. (Eds.), *Enabling Things to Talk: Designing IoT solutions with the IoT Architectural Reference Model* (pp. 113–162). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-40403-0_7

Becker, J., & Delfmann, P. (Eds.). (2007). *Reference Modeling*. Physica-Verlag HD. doi: 10.1007/978-3-7908-1966-3

Borshchev, A. (2014). Multi-method modelling: AnyLogic. *Discrete-Event Simulation and System Dynamics for Management Decision Making*, 248–279.

Broy, M., & Stølen, K. (2001). *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg.

Bucaioni, A., Di Salle, A., Iovino, L., Malavolta, I., & Pelliccione, P. (2022, August). Reference architectures modelling and compliance checking. *Software and Systems Modeling*, *22*(3), 891–917. doi: 10.1007/s10270-022-01022-z

Butting, A., Kautz, O., Rumpe, B., & Wortmann, A. (2017, April). Semantic Differencing for Message-Driven Component & Connector Architectures. In *International conference on software architecture (icsa'17)* (p. 145-154). IEEE.

Caracciolo, A., Lungu, M. F., & Nierstrasz, O. (2015, May). A Unified Approach to Architecture Conformance Checking. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*. IEEE. doi: 10.1109/wicsa.2015.11

Cloutier, R., Muller, G., Verma, D., Nilchiani, R., Hole, E., & Bone, M. (2009, January). The Concept of Reference Architectures. *Systems Engineering*, *13*(1), 14–27. doi: 10 .1002/sys.20129

Dobing, B., & Parsons, J. (2006). How UML is used. *Communications of the ACM*, *49*(5), 109–113.

Drave, I., Eikermann, R., Kautz, O., & Rumpe, B. (2019, February). Semantic Differencing of Statecharts for Object-oriented Systems. In S. Hammoudi, L. F. Pires, & B. Selić (Eds.), *Proceedings of the 7th international conference on model-driven engineering and software development (modelsward'19)* (p. 274-282). SciTePress.

Drave, I., Kautz, O., Michael, J., & Rumpe, B. (2019, September). Semantic Evolution Analysis of Feature Models. In T. Berger et al. (Eds.), *International Systems and Software Product Line Conference (SPLC'19)* (p. 245-255). ACM.

Dumas, M., Governatori, G., ter Hofstede, A. H., & Oaks, P. (2002). A formal approach to negotiating agents development. *Electronic commerce research and applications*, *1*(2), 193–207.

Durán Toro, A., Benavides, D., Segura, S., Trinidad, P., & Ruiz-Cortés, A. (2017, 10). FLAME: a formal framework for the automated analysis of software product lines validated by automated specification testing. *Software and Systems Modeling*, *16*. doi: 10.1007/s10270-015-0503-z

Egea, M., & Rusu, V. (2009, December). Formal executable semantics for conformance in the MDE framework. *Innovations in Systems and Software Engineering*, *6*(1–2), 73–81. doi: 10.1007/s11334-009-0108-1

Ernst, M. D. (2003). Static and dynamic analysis: Synergy and duality. In *Woda 2003: Icse workshop on dynamic analysis* (pp. 24–27).

Fahrenberg, U., Acher, M., Legay, A., & Wąsowski, A. (2014). Sound Merging and Differencing for Class Diagrams. In *Fundamental Approaches to Software Engineering*.

Fettke, P., & Loos, P. (2006, 01). Perspectives on Reference Modeling. In *Reference Modeling for Business Systems Analysis* (pp. 1–21). IGI Global. doi: 10.4018/9781599040547.ch001

Frank, U. (2006). Evaluation of Reference Models. In *Reference Modeling for Business Systems Analysis* (pp. 118–140). IGI Global. doi: 10.4018/978-1-59904-054-7.ch006

Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. (1997). *Design Patterns: Elements of Reusable Object-Oriented Software*. Prentice Hall.

Garcés, L., Martínez-Fernández, S., Oliveira, L., Valle, P., Ayala, C., Franch, X., & Nakagawa, E. Y. (2021, September). Three decades of software reference architectures: A systematic mapping study. *Journal of Systems and Software*, *179*, 111004. doi: 10.1016/j.jss.2021.111004

Gogolla, M., & Henderson-Sellers, B. (2002). Analysis of UML Stereotypes within the UML Metamodel. In *Lecture Notes in Computer Science* (pp. 84–99). Springer Berlin Heidelberg. doi: 10.1007/3-540-45800-x_8

Gray, J., & Rumpe, B. (2021). Reference models: how can we leverage them? *Journal Software and Systems Modeling (SoSyM)*, *20*(6), 1775-1776.

Grönniger, H., Krahn, H., Pinkernell, C., & Rumpe, B. (2008). Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen* (p. 76-89). TU Braunschweig.

Haber, A. (2016). *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Shaker Verlag.

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, *8*(3), 231-274. doi: https://doi.org/10.1016/0167-6423(87)90035-9

Harel, D., & Rumpe, B. (2004, October). Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer Journal*, *37*(10), 64-72.

Harris, J. A., & Patterson-Hine, A. (2013). *State machine modeling of the space launch system solid rocket boosters* (Tech. Rep.).

Herold, S., Mair, M., Rausch, A., & Schindler, I. (2013, September). Checking Conformance with Reference Architectures: A Case Study. In *2013 17th IEEE International Enterprise Distributed Object Computing Conference*. IEEE. doi: 10.1109/edoc.2013.17

Hölldobler, K., Kautz, O., & Rumpe, B. (2021). *MontiCore Language Workbench and Library Handbook: Edition 2021*. Shaker Verlag.

Hutchinson, J., Whittle, J., Rouncefield, M., & Kristoffersen, S. (2011). Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering* (pp. 471–480).

ITU-T. (1994, July). *Information technology – Open Systems Interconnection – Basic Reference Model: The basic model* (ITU-T No. X.200). International Tecommunication Union. ITU-T Recommendations.

Kautz, O., & Rumpe, B. (2018, October). Semantic Differencing of Activity Diagrams by a Translation into Finite Automata. In *Proceedings of models 2018. workshop me.*

Kim, D.-K., & Shen, W. (2007, March). An approach to evaluating structural pattern conformance of UML models. In *Proceedings of the 2007 ACM symposium on Applied computing*. ACM. doi: 10.1145/1244002.1244305

Kirchmer, M., & Franz, P. (2020). Process Reference Models: Accelerator for Digital Transformation. In *Business Modeling and Software Design* (pp. 20–37). Springer International Publishing. doi: 10.1007/978-3-030-52306-0_2

Konersmann, M., Michael, J., & Rumpe, B. (2024, March). Towards Reference Models with Conformance Relations for Structure. In *Informing possi-*

ble future worlds. essays in honour of ulrich frank (p. 247-269). Logos Verlag Berlin. Retrieved from http://www.se-rwth.de/publications/Towards-Reference-Models-with-Conformance-Relations-for-Structure.pdf

Langer, P., Mayerhofer, T., Wimmer, M., & Kappel, G. (2014). On the usage of UML: Initial results of analyzing open UML models. *Modellierung 2014*.

Lindt, A., Rumpe, B., Stachon, M., & Stüber, S. (2023, July). Cdmerge: Semantically sound merging of class diagrams for software component integration. *Journal of Object Technology*, *22*(2), 2:1-14. doi: 10.5381/jot.2023.22.2.a1

Lu, Y., Liu, C., Wang, K. I.-K., Huang, H., & Xu, X. (2020, feb). Digital Twin-Driven Smart Manufacturing: Connotation, Reference Model, Applications and Research Issues. *Robot. Comput.-Integr. Manuf.*, *61*(C). doi: 10.1016/j.rcim.2019.101837

Maoz, S., Ringert, J. O., & Rumpe, B. (2011a). ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on foundations of software engineering (esec/fse '11)* (p. 179-189). ACM.

Maoz, S., Ringert, J. O., & Rumpe, B. (2011b). CDDiff: Semantic Differencing for Class Diagrams. In M. Mezini (Ed.), *Ecoop 2011 - object-oriented programming* (p. 230-254). Springer Berlin Heidelberg.

Michael, J., Koren, I., Dimitriadis, I., Fulterer, J., Gannouni, A., Heithoff, M., ... Schuh, G. (2023, June). A Digital Shadow Reference Model for Worldwide Production Labs. In C. Brecher, G. Schuh, W. van der Aalst, M. Jarke, F. Piller, & M. Padberg (Eds.), *Internet of production: Fundamentals, applications and proceedings* (pp. 1–28). Springer. doi: 10.1007/978-3-030-98062-7_3-2

Nachmann, I., Rumpe, B., Stachon, M., & Stüber, S. (2022, June). Open-World Loose Semantics of Class Diagrams as Basis for Semantic Differences. In *Modellierung 2022* (p. 111-127). Gesellschaft für Informatik.

OMG. (2017). *Unified Modeling Language, Version 2.5.1.*

Parsons, T., & Murphy, J. (2004). Data mining for performance antipatterns in component based systems using run-time and static analysis. *Trans. on Automatic Control and Computer Science*, *49*(63), 49–63.

Richner, T., & Ducasse, S. (1999). Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings ieee international conference on software maintenance - 1999 (icsm'99). 'software maintenance for business change' (cat. no.99cb36360)* (p. 13-22). doi: 10.1109/ICSM.1999.792487

Ringert, J. O., Rumpe, B., & Stachon, M. (2023, July). On implementing open world semantic differencing for class diagrams. *Journal of Object Technology*, *22*(2), 2:1-14. doi: 10.5381/jot.2023.22.2.a11

Ringert, J. O., Rumpe, B., & Wortmann, A. (2013). MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)* (p. 10-12). IEEE.

Rumpe, B. (1996). *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. München, Deutschland: Herbert Utz Verlag Wissenschaft.

Rumpe, B. (2016). *Modeling with UML: Language, Concepts, Methods*. Springer International.

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

Schobbens, P.-Y., Heymans, P., & Trigaux, J.-C. (2006). Feature Diagrams: A Survey and a Formal Semantics. In *14th IEEE International Requirements Engineering Conference (RE'06)* (p. 139-148). doi: 10.1109/RE.2006.23

Systa, T. (1999). On the relationships between static and dynamic models in reverse engineering java software. In *Sixth working conference on reverse engineering (cat. no.pr00303)* (p. 304-313). doi: 10.1109/WCRE.1999.806969

Tran, H., Zdun, U., Holmes, T., Oberortner, E., Mulo, E., & Dustdar, S. (2012, June). Compliance in service-oriented architectures: A model-driven and view-based approach. *Information and Software Technology*, *54*(6), 531–552. doi: 10.1016/j.infsof.2012.01.001

Weinreich, R., & Buchgeher, G. (2014, April). Automatic Reference Architecture Conformance Checking for SOA-Based Software Systems. In *2014 IEEE/IFIP Conference on Software Architecture*. IEEE. doi: 10.1109/wicsa.2014.22

About the authors

Marco Konersmann is a postdoctoral researcher at the chair of software engineering of the RWTH Aachen University in Germany. His recent research focuses on software architecture, software modeling, and continuous software engineering, in the application domains of information systems, cyber-physical software systems and automotive. You can contact the author at konersmann(at)se-rwth.de or visit www.se-rwth.de.

Bernhard Rumpe is Professor at RWTH Aachen University, Germany and head of the Chair of Software Engineering. His main interests are rigorous and practical software and system development methods based on adequate modeling techniques. This includes agile development methods as well as model engineering based on UML/SysML-like notations and domain-specific languages. You can contact the author at rumpe(at)se-rwth.de or visit www.se-rwth.de.

Max Stachon is a research assistant and Ph.D. candidate at the Chair of Software Engineering at RWTH Aachen University. His research focuses on model semantics, refinement and difference analysis. You can contact the author at stachon(at)se-rwth.de or visit www.se-rwth.de.

Sebastian Stüber is a research assistant and Ph.D. candidate at the Chair of Software Engineering at RWTH Aachen University. His research focuses on model semantics, refinement, formal verification and compositional analysis of systems. You can contact the author at stueber(at)se-rwth.de or visit www.se-rwth.de.

Valdes Voufo is a student assistant at the Chair of Software Engineering at RWTH Aachen. You can contact the author at valdes.voufo(at)rwth-aachen.de.