JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF
Technisch-Naturwissenschaftliche
Fakultät

Model-and-Code Consistency Checking

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplomingenieur

im Masterstudium

Informatik

Eingereicht von:
Markus Riedl Ehrenleitner, BSc.

Angefertigt am:
Institut für System Engineering und Automation

Beurteilung:
Univ.-Prof. Dr. Alexander Egyed M.Sc.

Mitwirkung:
Dipl.-Ing. Andreas Demuth

Linz, September, 2013

Kurzfassung

In modellgetriebener Entwicklung, erhöhen Designmodelle die Abstraktionsstufe und erlauben es somit effizient, ohne auf Implementierungsdetails Rücksicht zu nehmen, zu modellieren. Bedauerlicherweise entwickeln sich beide Entwicklungsartefakte nicht nur häufig sondern auch gleichzeitig, mit der Gefahr, dass sich beide auseinander entwickeln. Obwohl Modell-zu-Quelltext Transformationen eingesetzt werden, um beide Artefakte zu synchronisieren, sind manuelle führungslose Anpassungen notwendig. Daher verhindern diese nicht effektive das Inkonsistenzen eingeführt werden. In dieser Arbeit wird daher ein Ansatz zu Model-und-Quelltext Konsistenzprüfung vorgestellt. Der Ansatz erkennt Inkonsistenzen und informiert Entwickler über den Konsistenzstatus eines Projekts. Ein automatisches Konsistenzprüfungsframework und 13 Konsistenzregeln für UML und Java wurden während der Arbeit entwickelt. Das Framework wurde in einer empirischen Studie, basierend auf 10 meist industriellen Projekten, evaluiert. Die Resultate der Evaluierung zeigen einerseits die technische Machbarkeit und andererseits die Skalierbarkeit des Ansatzes.

Abstract

In model-driven engineering, design models raise the level of abstraction and allow for efficient designing without considering implementation details. Still, it is crucial that design models and source code are in sync. Unfortunately, both development artifacts do evolve not only frequently, but also concurrently – which likely causes them to drift apart over time. Even though technologies such as model-to-code transformations are commonly employed to keep design models and source code synchronized, those technologies typically still require unguided, manual adaptations. Hence, they do not effectively prevent inconsistencies from being introduced. In this paper, we present a novel approach for checking consistency between design models and source code. The approach detects inconsistencies instantly and informs developers about a project's consistency status live during development. We developed an efficient and fully automated consistency checking framework and also provide a set of consistency rules for UML and Java projects. The framework was evaluated in an empirical study with 10 mostly industrial projects. The results showed that our approach is technically feasible and also highly scalable.

Contents

List of Figures

List of Tables

1 Introduction

Model-Driven Engineering (MDE) [1] promotes the use of design models as first-class development artifacts to address the inability of third-generation languages to alleviate the complexity of platforms and to express domain concepts effectively.

However, source code remains to be an important development artifact as it is the main deliverable of typical development projects and embodies the executable system. Thus, it is of crucial importance that both, design models and source code, are consistent and describe the same system. Otherwise, the executable system deployed to a customer may, for example, provide different functionality than what the customer was expecting based on design models. Commonly, model-to-code *transformations* [2] are employed to address this problem by generating source code automatically from design models. However, design models do not include all information necessary to generate fully functional implementations [3]. Moreover, design models typically allow for different implementations. Automatic approaches, however, do apply a fixed set of rules for translating design models to source code, thus generating one possible solution without being able of considering specific situations.

Thus, the produced solution may be correct but not necessarily intended by developers (e.g., a translation of an unbounded multiplicity defined in *UML* [4] may always use a `List`). Therefore, manual adaptation of generated code is often inevitable, meaning that inconsistencies between design models and code could be introduced as for those adaptations usually no further guidance is provided [5].

The issue of possible inconsistencies between design models and code becomes even worse when considering that – especially with the increasing popularity of iterative development processes – both design models and source code are developed concurrently [5]. This means that both development artifacts evolve frequently and independently, a situation in which even sophisticated, automatic consistency-preserving technologies (e.g., [3, 6, 7, 8, 9, 10]) do encounter additional serious issues that are hard to solve [3]. For instance, manual changes may be overwritten or – if the synchronization tries to avoid that – redundancies may be introduced.

Overall, existing approaches that are commonly employed for keeping design models and source code consistent do not fully address the issue – especially with frequently evolving artifacts: they typically require additional, manual adaptations for which only little support is provided and which may still introduce inconsistencies.

In this paper, we present a novel approach to address the issue of inconsistent design models and source code. *Model-and-Code Consistency Checking (MCCC)* is an incremental and highly scalable approach that detects inconsistencies between design models and source code at all times and live during development. It provides instant, detailed feedback about a project's consistency status and thus helps developers to not let design models and source code drift apart. Moreover, can be chained with other approaches (e.g., [11] that use detected inconsistencies to derive possible adaptations automatically – this work, however, focuses on inconsistency detection only. In contrast to other approaches that try to automate artifact synchronization, MCCC fully supports concurrent evolution of development artifacts and avoids otherwise common issues such as lost updates or incomplete information.

In order to detect inconsistencies in development projects that use UML and Java, we developed an initial set of 13 generic consistency rules between UML design models and Java source code. This set can easily be customized or extended for other modeling or programming languages and tailored to specific projects or domains. To validate our approach and show its feasibility, we developed a prototype and conducted a case study in which we applied the consistency rules to 10 industrial development projects based on UML design models and Java source code. The results indicate that the approach scales and detects inconsistencies live during development without interrupting developers.

To summarize the goals of this work:

1. Live consistency checking that handles concurrent evolution of development artifacts.

2. Adaptable consistency rules that allow for alternative interpretations and semantics of artifact (design models or source code) elements.

3. Initial set of generic rules for object-oriented modeling and programming, that is adaptable and expandable to fit domain and project specifics.

2 Model-Driven Engineering

France et al. [12] state that a challenge to face in developing complex software is the wide conceptual gap between the stated problem and its implementation. Therefore, one of the primarily concerns of MDE is to bridge the gap between the problem and its software implementation on a specific platform and to express domain specifics efficiently.

A key technology in MDE are transformations from problem-level abstractions(design models) to source code [1]. Design models describe the system at multiple levels of abstraction and a variety of perspectives and should shield developers from underlying implementation platforms.

Domain Specific Languages (DSLs) [13], are programming languages to model a problem-solution, -representation or -domain. Existing technologies such as the *Eclipse Modelling Framework (EMF)* [6] use domain-specific models written in a domain-specific language, to model the domain specifics of a software solution and then generate code. Several challenges have to be faced in order to achieve the MDE vision, which we will discuss in the following.

2.1 MDE Research

According to France et al. [12] the ever growing complexity of software systems will eventually not only overwhelm developers but as well available implementation abstractions, resulting in a new problem-implementation gap. Further rising the abstraction level of software development will become apparent. Such technologies then will enable even more complex systems, resulting in another cycle of research on problem-implementation gap, but on a higher level. To support this growing complexity, MDE research needs to develop technologies to generate domain-specific software development environments. Major challenges Model-Driven Engineering faces can be grouped in the following categories ([12]):

- Modeling language challenges. Provide support for creating and using problem-level abstraction in modeling languages, and for analyzing design models.
- Separation of concerns challenges. Provide support for multiple, overlapping viewpoints that utilize possibly heterogeneous languages.

- Model manipulation and management challenges. Provide support for i) defining, analyzing, and using design model transformations, ii) maintaining traceability links among design model elements for model evolution, iii) maintaining consistency among viewpoints, iv) tracking versions, and v) using runtime models.

MDE provides automated support for software engineering and thus can be considered as evolution of *computer-aided software engineering (CASE)* [14]. However, MDE broadens the role of design models to an extent that they become the primary development artifact. Further on, each of the challenges will be discussed briefly.

2.2 Modeling Languages

Creation of modeling languages faces two major challenges ([12]):

- Abstraction Creation and manipulation of problem-level abstractions as first-class modeling elements in a language.
- Formality Formalization of a modeling language's semantic to support formal manipulation.

Formalization tends to restrict modeling languages to provide analysis, transformation and generation techniques. Analyzing design models is important to secure the quality of those. Not only analysis of design models but as well guidance for developers creating such is required to produce quality design models.

2.3 Separation of Concern

Designing complex software need to balance multiple interdependent, possibly conflicting, concerns. Developers should be allowed to separate features addressing those concerns in different viewpoints and analyse their interaction to identify faulty interactions.

2.4 Model Manipulation and Management

To completely fulfill the vision of MDE, rigorous transformation-, modeling- and analysis-techniques are needed. As well a repository-based infrastructure that support a variety of design model manipulations in a team based environment.

Model Transformation Model transformations define relationships among two sets of models, according to France et al. [12]. One set of models is designated as source set and the other as target set. Transformation techniques then transform source models into target models. These transformations can also be employed to maintain the consistency among the set of models. A change in one model triggers changes in the other set. Other transformations that will become more used as MDE matures: i) model composition, the source model represents different views, the target model then integrates all those views, ii) model decomposition, a source model can be used to produce multiple target models, iii) model translation, a source model can be translated into different target models.

Model Management During the lifetime of a project, design models at varying levels of abstraction are created, they evolve, are analyzed and are transformed. MDE approaches according to France et al. [12] must have the capability to store models produced by a variety of tools, monitor and audit model manipulations and automatically extract information from audits to establish, updated or remove relationships among models.

Model management should as well audit the consistency among different sets of models. MCCC addresses the problem of inconsistent development artifacts.

3 Running Example

This section introduces the illustrative example used throughout the paper. For the two development artifacts design model and source code, the most important details (e.g., classes and methods) are presented.

3.1 Model and Code

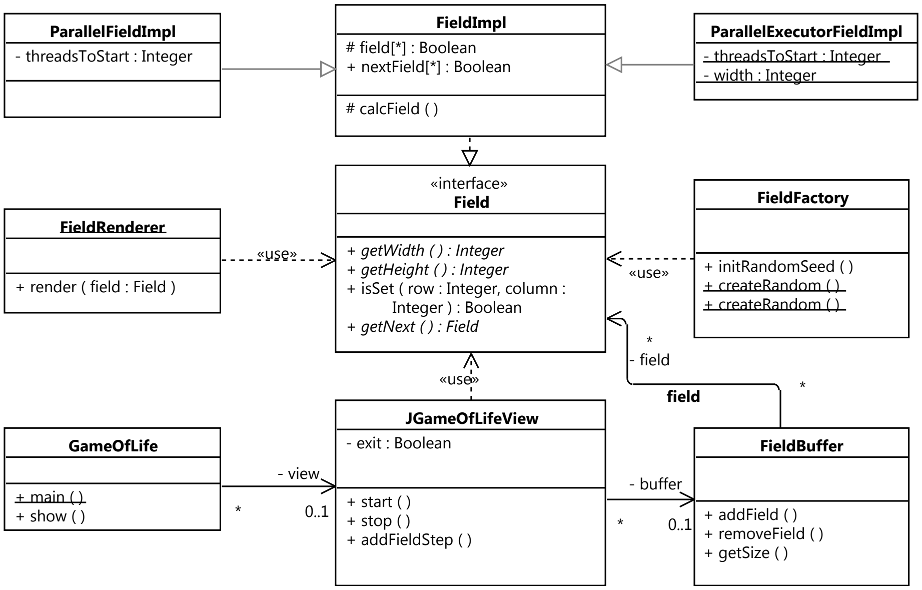


Figure 1: Class Diagram

To illustrate our work, we chose a small implementation of Conway's *Game of Life (GoL)* [15], a simulation of cellular development based on specific rules. It is a non-player game – based on a random generated population, further generations are computed one at a time. Despite its simplicity, the example is sufficient to illustrate typical issues regarding model-and-code consistency. Figure 1 depicts the class diagram overview of the system. The class `GameOfLife` contains the `main`-method in which data structures are initialized and an infinite loop, calculating at each step a new generation, is started. This is visualized in the sequence diagram shown in Fig. 2. Each generation of a GoL population is calculated from classes implementing the interface `Field`.

Single thread calculation is realized in `FieldImpl`. There are two multiple

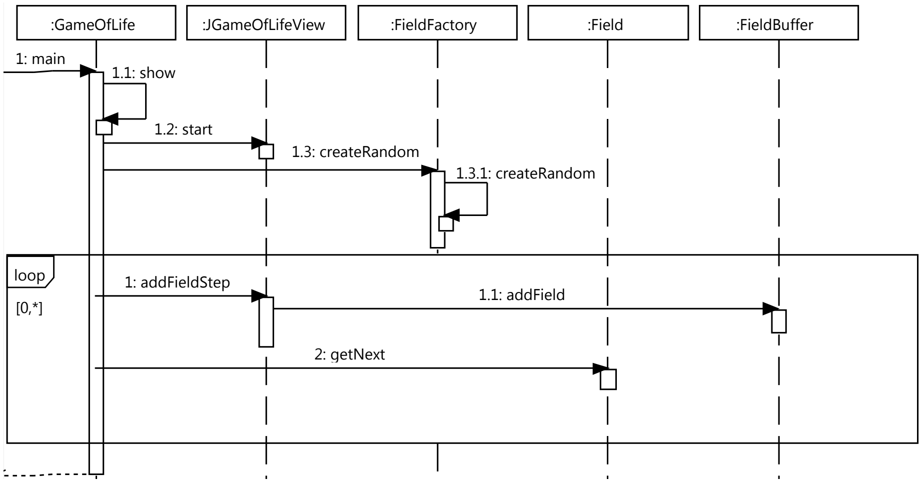


Figure 2: Initialization and Calculation Sequence

thread implementations: i) `ParallelFieldImpl` with common thread handling, and ii) `ParallelExecutorFieldImpl` with pooled thread handling. Synchronization between the visualisation `JGameOfLife` and the next generation calculations is done by `FieldBuffer`. Both `addField()` and `removeField()` are synchronized methods, forcing threads to wait, if the buffer is either full or empty, and waking up threads, when a `Field` is added or removed, respectively.

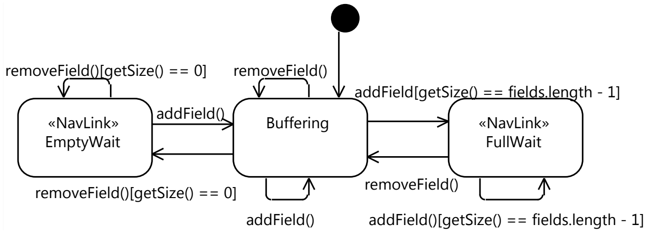


Figure 3: FieldBuffer Statespace

Figure 3 shows the simple state space of the `FieldBuffer`. `Buffering` characterizes the normal queueing of `Fields`. `FullWait` and `EmptyWait` are synchronizing states. One might argue that there are intermediate steps between `Buffering` and `FullWait`, or `EmptyWait`, respectively, because those states are only reached if the buffer is already full or empty. However, those states do not

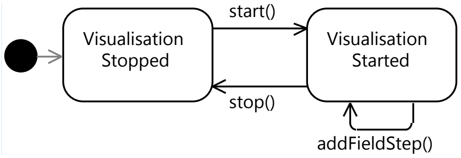


Figure 4: Visualization State Space

differ from `Buffering`, it still can be considered as buffering. Not until another element is added or removed, the thread is forced to wait. Listing 1 shows the synchronized methods, assuming that the buffer is full, a thread would notice this only if he tries to add another `Field`. There is also no code line handling this case. Therefore, neither can be statically verified, runtime information of the system is needed.

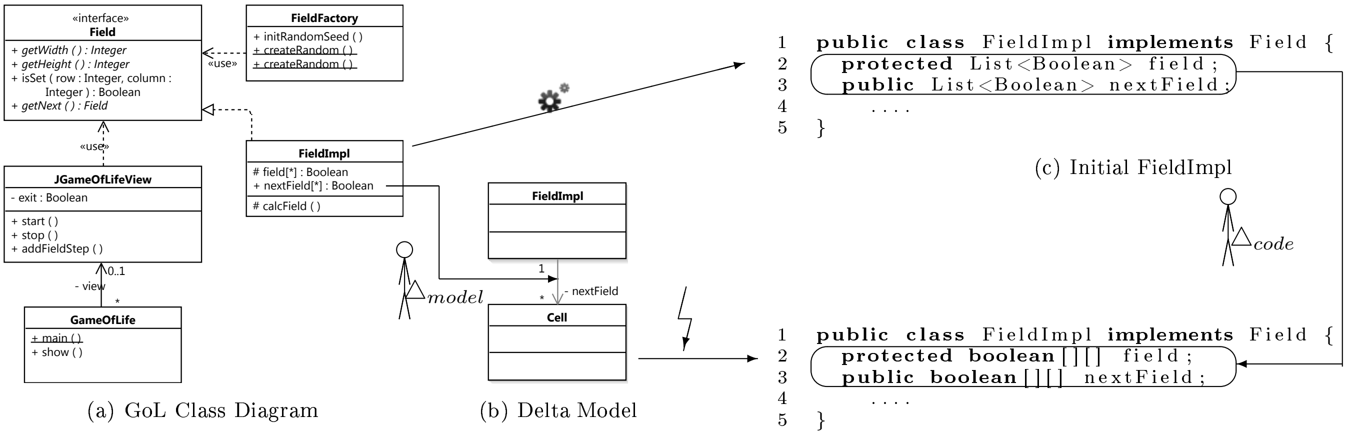
`JGameOfLifeView` is the visualization class of the system, which displays each generation, its state space is depicted in Fig. 4. To begin the visualization, `start()` has to be called. New generations to display are added with the `addFieldStep()` method. A helper-class `FieldRenderer` was implemented to generate a visual representation out of the two-dimensional boolean array. Finally `stop()` ends the visualization.

3.2 Evolution

This section discusses typical issues regarding model-and-code evolution in the context of the provided example. We show how these issues easily cause inconsistencies among those development artifacts. First, a source code and a design model evolution are discussed. Followed by a scenario in which multiple over-

```java
public synchronized void addField(final Field field)
  throws InterruptedException {
  while (getSize() == fields.length - 1) {
    wait();
  }
  notifyAll();
}
public synchronized Field removeField() throws InterruptedException {
  while (getSize() == 0) {
    wait();
  }
  notifyAll();
}
```

Listing 1: Code Snippet FieldBuffer



```
public class FieldImpl implements Field {
  protected List<Boolean> field;
  public List<Boolean> nextField;
    ....
}
```

```
public class FieldImpl implements Field {
  protected boolean[][] field;
  public boolean[][] nextField;
    ....
}
```

(a) GoL Class Diagram (b) Delta Model (c) Initial FieldImpl (d) Desired FieldImpl

Figure 5: Field Evolution

lapping models describe different possible runtime behavior of a single method in source code.

3.2.1 Field

Consider the multiplicity of the `UML::Property` `nextField` in the `UML::Class` `FieldImpl`, as shown in Fig. 5(a), and the corresponding `Java::Field` as shown in Fig. 5(c). Obviously, this translation of the multiplicity is valid and there is no inconsistency.

However, the developer, instead of using the created list of `Boolean`s, wants to represent the two-dimensional grid of cells in a two-dimensional array of `boolean`, as shown in Fig. 5(d). This manual evolution is indicated in Fig. 5 by $\triangle_{code}$.

Even though this adaptation does not introduce inconsistencies as the updated source code still reflects the specified multiplicity in the class diagram, such manual adaptations could as well introduce inconsistencies (e.g., if the code was changed to `boolean nextField;`). Note that model-and-code synchronization mechanisms would not detect such an inconsistency, thus propagating the inconsistency back to the model automatically.

Naturally, changes must not be limited to the code only. In order to better follow the paradigm of object-oriented modeling, a developer decides to use a dedicated type `Cell` to represent individual cells in GoL instead of using boolean values (Fig. 5(b)). This change is depicted as $\triangle_{model}$ in Fig. 5. Therefore, the developer created an inconsistency among design model and source code as the primitive type `boolean` of Java does not match the complex type `Cell`.

Note that due to the previous code evolution, an automated synchronization may easily either override the previous code evolution, or it may generate a new declaration (i.e., `List<Cell> nextField`) besides the existing, manually updated declaration `boolean [][] nextField`, which is incorrect. Either way, trying to automatically handle the manually introduced inconsistency may lead to unintended and potentially still inconsistent results. Thus, another technology that checks consistency between design models and source code at all times is necessary.

3.2.2 Sequence

A common understanding is that a sequence of messages defined in a sequence diagram should be found in code as well [16] – if a message is not reflected by a corresponding method call in code, the artifact elements are considered to be inconsistent.

The initialization sequences of our running example is modeled in two separate sequence diagrams (Fig. 2 and Fig. 6) that are partially overlapping (i.e., they are equal except for the last message).

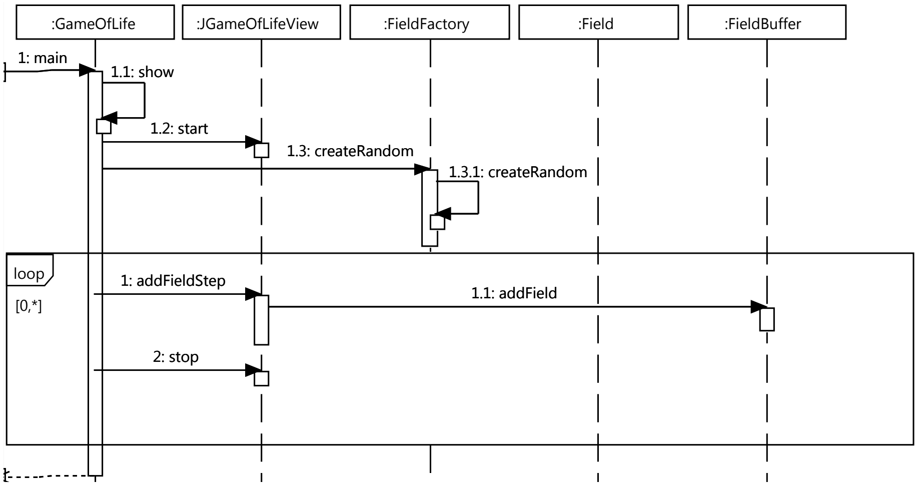


Figure 6: Alternative Initialization Sequence

However, model-to-code transformation approaches may not be able to successfully generate a single coherent method in source code that reflects both possible scenarios. Clearly, the missing implementation is an inconsistency that developers are required to fix by implementing the method in the source code manually. Unfortunately, transformation approaches do not inform developers about such situations and require them to explore and complete the inconsistent code without further assistance.

As a consequence, a developer manually produced the implementation shown in Listing. 2. Notice that the code actually contains an inconsistency. Both sequence diagrams define that the call `start()` to `JGameOfLiveView` is in between `show()` and `createRandom()`. However, in Listing. 2 (line 5) this call is incorrectly placed after `createRandom()` and therefore neither of the sequence diagrams is reflected in the source code.

This is another situation where the need for model-and-code consistency checking becomes apparent.

3.3 Contained Inconsistencies

We now want to discuss corresponding artifact elements and inconsistencies among those. For discovered inconsistencies, we also elaborate strategies how to resolve those. Each inconsistency discussed is briefly summarized in a table, that is organized as follows: the top line shows the name of the intended consistency rule (left) and the context (i.e., the specific element) for that it is evaluated (right). In the middle, the design model and source code elements (or parts thereof) that cause the inconsistency are shown. The bottom of the table summarizes the inconsistency and gives a strategy to resolve it.

3.3.1 Field

Consider the visibility modifier of the `UML::Property` `field` in the `UML::Class` `FieldImpl` and the corresponding `Java::Field` (both shown in Table 1). Obviously the modifiers of the corresponding artifact elements does not match, the modeled property is private, however the implemented field is public.

ClassFieldConform	 ClassFieldConform[nextField]
Property `field` in `UML::Class FieldImpl` in Fig. 1	`private boolean[][] field;`
Source of Inconsistency Although the `UML::Property` is private, the `Java::Field` instead is public.	**How to fix it?** Change the modifier of the `UML::Property` to public or the `Java::Field` to private.

Table 1: Field Inconsistency

3.3.2 Method

Methods provide the public interface of a type (e.g., `UML::Class`, `Java::Class`, `UML::Interface`, ...) and should be found in a correct translation in source code. For instance, compared to the other `UML::Operations` of `Field`, `isSet` is not set to `abstract` as one can see in the middle line in Table 2 on the left side. The matching `Java::Method` is as well set to `abstract`, Table 2 on the right side. The developer most likely forgot to set the `UML::Operation` abstract. Arguably interface-methods are already abstract, but since the methods were deliberately set abstract, the `UML::Operation isSet` should also exhibit the modifier. We discussed this inconsistency in the context of operations being abstract or not, it could also result from any other modifier.

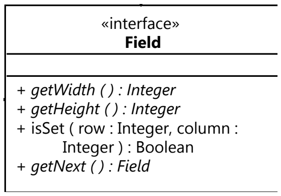
<table>
<tr><th>ClassMethodConform</th><th>✖ ClassMethodConform[isSet]</th></tr>
<tr><td>
</td><td><pre>public abstract
boolean isSet(...);</pre></td></tr>
<tr><td>Source of Inconsistency
Although the Java::Method isSet is abstract the UML::Operation does not exhibit the abstract modifier.</td><td>How to fix it?
Set the UML::Operation abstract.</td></tr>
</table>

Table 2: Operation Abstract Inconsistency

```java
public static void main(String[] args) {
  final GameOfLife gol = new GameOfLife();
  gol.show();
  Field field = FieldFactory.createRandom(800,600,0.2f);
  gol.view.start();
  for (;;) {
    try {
      gol.view.addFieldStep(field);
    } catch (InterruptedException e) {
      gol.view.stop();
    }
    field = field.getNext();
  }
}
```

Listing 2: Initialization Sequence

3.3.3 Sequence

This section summarizes the previously mentioned example (Section 3.2.2) in Table 3. Figure 2 shows the initialization sequence of our running example, whereas Listing 2 shows the corresponding implementation. In Fig. 2, a call to start the visualization `start()` follows after a `show()`-call to `GameOfLife`. However, in Listing 2 the call `start()` is performed after a random field is created, as it would be required by Fig. 2 (line 5 of Listing 2). Therefore, there is a inconsistency between design model and source code.

InteractConform	 ✖ InteractConform[Initialization]
Message `start` in Fig. 2.	Line 5 in Listing 2.
Source of Inconsistency The call to `start` in the visualization is misplaced.	**How to fix it?** Move the call between `show` and `createRandom`.

Table 3: Interaction Inconsistency

3.3.4 Statechart

Transitions in a state machine diagram can be messages or calls to an `Object`. A sequence of possible transitions from state to state then defines also allowed sequences of calls to its corresponding implementation. If after a sequence of calls a final or the beginning state, respectively, is not reached it has to be considered as inconsistent. An example of such an inconsistency is summarized in Table 4. The state space of the visualization is depicted in Fig. 4. Considering the state space of the system, the state `VisualizationStarted` is reached, even though line 5 is wrongly placed. However, the `stop()`-call, which would lead to the beginning state, is in a `catch`-block, therefore it is only reached in a failure case. If now a developer adds another `stop()`-call after line 13, a non-failure control flow would be consistent with the state chart in Fig. 4, as the beginning state is reached. Nevertheless, the call is directly after an infinite loop and will not be reached.

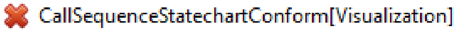
CallSequenceStatechartConform	 ✖ CallSequenceStatechartConform[Visualization]
Fig. 4.	Listing 2.
Source of Inconsistency The call to end the visualization after the loop ending in line 13 in Listing 2 is missing.	**How to fix it?** The call needs to be added after line 13 in Listing 2. However, in order to be reached, the loop has to be changed.

Table 4: Visualization Inconsistency

3.3.5 Statechart Uncertainty

Calls to an `Object` alter its state, however, these messages can be arbitrarily nested in sub-calls, loops or if-expressions. These possibilities introduce a source of possible uncertainty if the modeled behavior is reflected correctly in the implemented runtime behavior. Such an example is summarized in Table 5. Listing 3 shows an alteration of the already known initialization sequence for GoL. The call to start the visualization (line 6 is now intended in an if-expression. Without runtime information of the system, one can not be absolutely sure that this call is actually executed. However, the call exists and might be executed, this can be attributed as uncertainty in reasoning about a sequence of calls.

```
1  public static void main(String[] args) throws InterruptedException {
2    final GameOfLife gol = new GameOfLife();
3    gol.show();
4    JGameOfLifeView view = gol.view;
5    if (view != null) {
6      view.start();
7    }
8    Field field = FieldFactory.createRandom(800, 600, 0.2f);
9    for (int i = 0; i < 10; i++) {
10     view.addFieldStep(field);
11     field = field.getNext();
12   }
13   view.stop();
14 }
```

Listing 3: Alternative Initialization Sequence

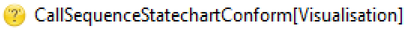
CallSequence-StatechartConform	 CallSequenceStatechartConform[Visualisation]
Figure 4.	Listing 3
Source of Uncertainty The call to start (Listing 3 line 6) the visualisation is in an if-expression.	**How to fix it?** ...

Table 5: Visualization Uncertainty

4 Principles of Model-and-Code-Consistency Checking

Model-and-Code Consistency Checking (MCCC) addresses the issue of inconsistent development artifacts by continuously evaluating a project's consistency status. It thus guides developers through the software development cycle with constant and immediate feedback on inconsistencies between artifacts. In doing so, it effectively helps developers to take measures to not let design models and source code drift apart.

4.1 Explicit Consistency Rule

Let us illustrate how developers may want to intuitively express the conditions that are required to hold for artifacts in order to be consistent. Listing 4 shows a stylized informal example of an explicit consistency rule between `UML::Property`s and `Java::Field`s:

```
1  UML::Property x
2   x.name = x.javaField.name &&
3   x.visibility = x.javaField.visibility &&
4   if x.cardinality=* then
5     x.javaField is a collection or an array
```

Listing 4: Stylized Consistency Rule

First, both the `UML::Property` and its corresponding `Java::Field` must exhibit the same name (line 2). In order to compare those, at first a developer has to navigate to the corresponding `Java::Field`, using the `javaField`-property. Second, the visibility of both need to be compared (line 3). Finally, the multiplicity of the `UML::Property` can map to different manifestations of a multiplicity in Java (lines 4-5).

4.1.1 Consistency Rules Prevent Drift

As we have discussed above, development artifacts do evolve frequently for various reasons. Consistency rules evaluate continuously whether specific aspects of those artifacts are consistent (i.e., one artifact does not violate or contradict information that is specified in the other artifact). Therefore, they detect immediately if an evolution of either artifact causes (or also removes) such a violation or contradiction.

4.1.2 Consistency Rules Promote Convergence

In practice, design models often precede the production of source code and developers rely on model-to-code transformations. However, as we have discussed in Section 3.2.2, the result may be a skeleton implementation which could be mostly inconsistent due to missing method bodies. In this situation, it is crucial that developers are informed about all inconsistencies and that they are provided with guidance that helps them converging the artifacts – eventually reaching a consistent state.

Recall the discussion of Section 3.2.2 in which two different initialization sequences (Fig. 2 and Fig. 6) are to be combined into a single piece of source code that could not be generated. Consistency rules can evaluate for each sequence diagram individually whether it is reflected in the source code. For example, a consistency rule can validate for each incoming call to a `UML::Lifeline` whether the following outgoing calls are performed in the right order in the corresponding `Java::Method`, ignoring method calls that are not modeled in the specific lifeline (or sequence diagram). For both Fig. 2 and Fig. 6, for instance, this rule would search for different sequences of method calls performed in the method `main` of `GameOfLife` – it would check whether the respective call-sequences in the sequence diagrams are sub-sequences of the statements in `main`. With an empty skeleton implementation, the consistency information provided by the rule would assist developers to stepwise add missing statements and remove inconsistencies. During this process, consistency information from other rules also ensures that those adaptations do not accidentally introduce new inconsistencies.

4.1.3 Challenges

In order to write consistency rules like the one shown in Listing 4, different challenges have to be faced. These are namely: i) design models and source code are not integrated in a single cohesive language – which means a single consistency rule cannot access elements of both, ii) design models and source code do not allow inter-model navigation – which requires tedious navigation, iii) design models and source code editing generates different $\triangle$ for each artifact – which requires artifact specific change handling.

4.2 Framework

The MCCC approach integrates artifacts of different sources and provides navigation links that allow for simple writing of explicitly stated consistency rules and their efficient evaluation – also after artifact evolution – by an incremental consistency checker. An overview of the approach is given in Fig. 7. Before we discuss a set of consistency rules that check consistency of design models and object-oriented code, and their application, in Section 5, we first present the domain-independent aspects of the approach in this section. However, we will begin with formalizing consistency rules. An overview of the approach is given in Fig. 7, several entities are shown: i) the `Source` artifact is any unit of code in plaintext (e.g., Java-class, C#-file), ii) the `Code`, an internal abstract representation of the `Source`, iii) the `Model`, this is as well an internal abstract representation of a model (e.g., UML), iv) the `Integration Layer`, unifying access to both `Model` and `Code`, and v) the `Consistency Checker` evaluates the explicit `Consistency Rules` after every model change. Before we discuss a set of consistency rules that check consistency of design models and object-oriented code, and their application, in Section 5, we first present the domain-independent aspects of the approach in this section. However, we will begin with formalizing consistency rules.

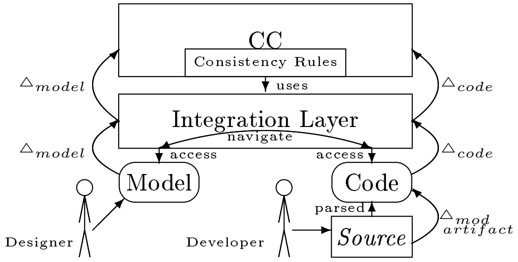


Figure 7: Overview

4.3 Rule Elements

To express consistency among development artifacts, our approach relies on explicit consistency rules. These rules are adaptable at all times and thus allows for any semantics required – imposed by project, company or domain specifics. Consistency rules, as defined in [17] and shown in Eq. 1, are boolean conditions, or invariants, that a developer wants to hold among development artifacts Such

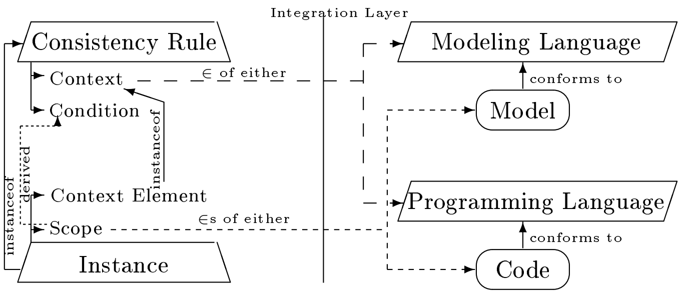


Figure 8: Consistency Rules

rules are typically formulated for specific types of artifact elements.

$$\begin{aligned}
&\text{Consistency Rule} = \\
&<\text{Condition, ContextElement}> \rightarrow \text{Bool} \\
&\text{where ContextElement} \in \text{MetaModelElements}
\end{aligned} \tag{1}$$

Figure 8 intends to give an overview of the concept of an explicit consistency rule.

A consistency rule consists of two parts: *Context* and *Condition*. The *Context* is an artifact element specified in either a *Modeling Language* or *Programming Language* and determines on which artifact elements the rule should be evaluated. Each individual evaluation is then an instantiation of the rule. The *Condition* is written from the viewpoint of the *Context* and specifies the invariant that should hold.

On the bottom of Fig. 8 an *Instance* – a formal definition was given in [17] and it is shown in Eq. 3 – of a consistency rule is shown, it as well consists of two parts: *Context-Element* and *Scope*. The *Context-Element* is an instance of its corresponding *Context* for which the rule is actually evaluated. The *Scope* [18], formally defined in [17] and shown in Eq. 2, is a set of elements that are accessed when evaluating the *Condition* on the specific *Context-Element*.

$$\begin{aligned}
&\text{Scope(Instance)} = \\
&\text{Set of } <\text{ModelElement, Field}> pairs \\
&\text{accessed during Evaluation of CRI.}
\end{aligned} \tag{2}$$

$$\begin{aligned}&\text{Instance} = <\text{ConsistencyRule, Model Element}>, \textit{where}\\&\text{ModelElement instance of ContextElement}\\&\text{ConsistencyRule.}\end{aligned} \tag{3}$$

Both `Consistency Rule` and `Instance` are oblivious to the fact that different development artifacts exists, this is visualized by the straight vertical line in Fig. 8.

Noticing that statecharts as design elements had a specific challenge to consistency rules, the definition was slightly altered, to allow expressing uncertainty as shown in Eq. 4.

$$\begin{aligned}&\text{Consistency Rule} =\\&<\text{Condition, ContextElement}> \rightarrow \text{Bool, Float}\\&\text{where ContextElement} \in \text{MetaModelElements}\end{aligned} \tag{4}$$

In static analysis one can not be absolutely sure that certain runtime specifics are actually executed the way they are expressed in the model. Consider a message that leads to the next state in a statechart. However, its corresponding call in the implementation is contained in an if-expression. Without runtime information it is impossible to be certain, that this call is actually executed. This property is attributed as an uncertainty factor, in Eq. 4 consistency rules therefore can also return values of type float(=uncertainty). A value of 0.0 is considered as completely inconsistent, 1.0 express complete consistency, any value in between expresses an uncertain result.

4.4 Artifact Integration

Both, design models and source code are typical development artifacts in MDE. For detecting inconsistencies between the design model and source code, those artifacts must be integrated in a manner that a consistency checker can access them in a unified way.

4.4.1 Metamodel

Unfortunately, artifacts are typically instances of different metamodels and of different development tools. For example, an UML model element is an instance of an UML metamodel element and typically edited in an UML modeling tool.

In turn, Java code needs to conform to the Java language specification and typically edited in a Java programming tool.

Moreover, retrieving information from design models differs significantly from accessing source code. Although both artifacts are typically stored in plain text files, modeling tools do use a in-memory representation (e.g., [19, 20]), whereas programming tools often work with plain text files only (e.g., [21, 22]). Text files are as well not necessarily saved continuously during editing and since the consistency checker requires continuous access to both design model and source code (as well instant access to changes). We opted to provide an uniform, in-memory access to both. Following, we refer to `Code` and `Model` as the in-memory representation of source code and design model, respectively – see bottom of Fig. 7. Both are accessible via the `Integration Layer`.

However, although both design model and source code are now commonly accessible through a standard infrastructure, each artifact still conforms to its respective metamodel. Therefore, it is still not possible without additional integration to link `Model` elements to `Code` elements or support explicit consistency rules that govern both. This integration is handled by the `Integration Layer`, that provides a single coherent metamodel for both `Model` and `Code` to the consistency checker (i.e., the consistency checker works with a single metamodel that contains the metaclasses of both artifacts) in Fig. 7. This enable the use of standard consistency checkers that typically work with a single metamodel (e.g., [23, 24, 25]). Additionally, the unified metamodels provides the possibility of using `Navigation Links` between artifact elements (e.g., to allow a specific `UML::Class` to be linked to a specific `Java::Class`). Next, we discuss the benefits of such direct navigation between artifact elements and how it is handled by the integration layer.

4.4.2 Navigation

Even with a unified view on `Model` and `Code`, both remain independent artifacts and elements of both are typically not linked explicitly. As such, it is hard to know which `Model` and `Code` elements to match. Revisiting the consistency rule discussed in Section 4.1, note how the functions `javaClass` and `javaField` are used to navigate between `Model` and `Code` elements. Indeed, this is convenient for writing explicit consistency rules, but how can those methods return `Code` elements for given `Model` elements? The solution are explicit navigation links that establish bidirectional relations between `Model` and `Code` elements. Note that although a developer may have clear relations in mind (e.g., a `UML::Class` and a `Java::Class` with equal names), such explicit links must be present in order to enable convenient navigation. The alternative would be tedious navigation from an artificial root element of a `Model`- or `Source`-element to the "corresponding" element having the correct name. Note that such a search-based navigation would require consistency rules to not only be more complex to write, but also to be more complex to evaluate by a consistency checker. By using shortcuts in navigation through `Navigation Links`, scope sizes are reduced as well.

For example, consider that a developer wants to compare all `UML::Property`s (`#n`) against all `Java::Field`s (`#m`) of a `Model` and `Code` class pair. Then, `n*m` evaluations will be done at each change, leading to higher execution times as the scope size is large. The consistency rules we implemented are written on a specific context to compare, for example, one instance of `UML::Property` against all `Java::Field`s of a corresponding `UML::Class` and `Java::Class`-pair. This favours execution times, instead of `n*m` evaluations only `m` evaluations have to be done. However, for all `UML::Property`s in a `UML::Class` still `n*m` evaluations have to be done, at least at the beginning. If now a specific `UML::Property` is changed, this requires only requires `1` re-evaluation. If a `Java::Field` is changed, it requires the re-evaluation of all instances, which scope contains the field. If, however, a `Java::Field` is added, it still requires `n` re-evaluations and `n*m` checks in total.

The scope size is of great importance as artifacts evolve frequently and each evolution requires consistency rule instances to be re-evaluated. Having to traverse complex data structures for each re-evaluation could impose a significant performance threat. Therefore, the `Integration Layer` augments the respective metamodel of the artifact with additional functionality for direct and bidirectonal navigation between model and code.

Those `Navigation Links` can either be `implicit` based on an inherent property of an `Model` or `Code` element as mentioned above, or `explicitly` stated in corresponding elements. Given explicit consistency rules, artifact integration, and navigation among `Model` and `Code`, we now discuss how evolving artifacts can be checked for consistency efficiently.

4.5 Incremental Consistency Checking

Both, `Model` and `Code`, do evolve over time for various reasons (e.g., because of iterative development processes). MCCC therefore promotes the use of an incremental consistency checker that handles changes efficiently and provides immediate feedback.

Recall Fig. 7, it as well shows the process of incremental consistency checking: The left hand side depicts the process of changes in the `Model` $\triangle_{model}$, those are instantly sent to the checker. The situation is different for `Code`, incrementally update of the internal representation had to be emulated. If a developer edits a `Source`-element, this new version is compared with the in-memory `Code` version, resulting in a $\triangle_{mod.artifact}$. With this information the in-memory element is updated. This subsequently leads to the generation of $\triangle_{code}$, initiating the re-evaluation of the scope. Note that the use of atomic changes and incremental re-evaluation of consistency rule instances reduces the effort for keeping consistency information up-to-date significantly when development artifacts are manipulated frequently. Moreover, the order in which change information is passed to the consistency checker does not affect consistency results. Thus, simultaneous updates of both artifacts do not lead to race conditions. Next, we demonstrate the application of our approach to the motivational example from Section 3.

5 UML/Java Application

Let us now present in detail how MCCC is applied to our running example. In this section, we first show how the metamodels of UML and Java are integrated and how navigation is realized. Then, we present a set of consistency rules that is suitable for object-oriented modeling and programming and discuss possible alternative semantics. Finally, we illustrate how our approach helps developers to identify and correct inconsistencies between development artifact elements.

5.1 Metamodels and Navigation

Two possibilities exist to integrate the development artifacts metamodels: i) merge `Model`- and `Code`-metamodel into one comprehensive model and add to this metamodel the inter-model navigation properties, ii) there exists a third metamodel which imports `Model`- and `Code`-metamodel and in which `Navigation Links` are defined. The second option has the advantage that tools for modeling and coding can still use their own metamodels, whereas with the first option either tool would have to use the new metamodel, or the elements need to be translated. Therefore, we decided to use the second approach. `Navigation Links` are then defined in the third metamodel and each `Navigation Link` object contains a reference to a UML- and an Java-element. Figure 9 depicts the solution.

The `Navigation Link`-Rectangle is the `Object` storing the references to UML and Java, depicted as solid lines between `Navigation Link` and `UML` or `Java`, respectively. The dashed line, on the other hand, is the ideal world view the tool emulates, opposites of `Model` or `Code` elements are directly accessed, with no visible intermediate link.

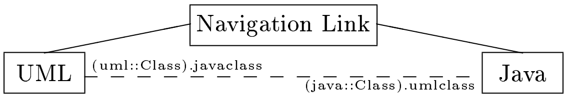


Figure 9: Navigation Overview

One might argue that `Objects` storing those `Navigation Links` are not necessary. This is true if links are achieved by a simple heuristics (e.g., if a `UML::CLass` and a `Java:Class` have the same name, then they are considered opposites of each other). However, each access of a `Navigation Link`-reference would result in a search in the complete `Model` or `Code` for an element fulfilling the heuristics. Considering that by evaluating a consistency rule a link – possibly the same – needs to be evaluated multiple times, which would lead to unnecessary computational overhead.

Navigation among metamodels, and therefore between `Model` and `Code` elements, should follow standard navigation patterns. Thus, we chose to provide additional references that are added to artifact elements. If no corresponding element is specified, the reference returns `null`.

As outlined above, there exists a metamodel specifying possible links. From this metamodel possible navigation references for each metamodel are computed. Take the example of Fig. 9, suppose the UML-reference (in the `Navigation Link-Object`) is of type `UML::Class`, with name `umlclass` and the Java of `Java::Class`, with the name `javaclass`. Then, for `UML::Class` a reference called `javaclass` will be provided and accepted by the consistency checker. This reference does not really exist in the respective metamodel definition of `UML::Class`, but since the `Integration Layer` abstracts `Model` and `Code` accesses from the consistency checker it will be nevertheless accepted. At interpretation time of a rule instance, navigation references are instead handled by a `NavigationHandler`, which manages the `Navigation Links`.

5.1.1 Implicit Navigation Links

Simple heuristics can be used to introduce `Navigation Links`, but instead of applying those at every access, MCCC provides a one time computation of all possible match-ups, which are then stored. As previously mentioned, a simple, yet often practical heuristic could consider a `UML::Class` and a `Java:Class` to be opposites if they have the same name. This simple name heuristics (Listing 5) will often be sufficient for typical UML/Java projects, of course any heuristics can be used, to meet project specific needs.

```
for UML::Classifier u : Design Model
  for Java::Classifier j : Source Code
    if u.type matches j.type //Interface, Enumeration, Class
     && u.name == j.name then
      generate Navigation Link
```

Listing 5: Simple Name Heuristics

```
package shapes;
/* @NavLink(id="shapes.Rectangle") */
public class Rectangle {
  //@NavLink(id="shapes.Rectangle.getXCoord")
  public double getXCoord(){
    //...
  }
}//Rectangle
```

Listing 6: Navigation Link Examples

5.1.2 Explicit Navigation Links

Storing the `Navigation Links` also opens up the possibility to set explicit links in `Model` and `Code`. This possibility was introduced for situations were a developer does not want to evaluate the whole project or introduce connections that are not obvious to an algorithm. A match of two entities is achieved based on a `String-id`. If two `id`'s of an `Model` and `Code` element are the same one can navigate among those. For Java we experimented with the possibility to let the developer introduce an annotation type, called `NavLink`. This had obvious disadvantages, since this type had to be accessed from every point in the code, and might violate architectural rules. Moreover, Java annotations can not be applied to statements, which would limit the scope of `Navigation Links`. Therefore, links were put into comments, but the syntax was kept, to distinguish it from normal ones.

Listing 6 illustrates possible examples of explicit `Navigation Links`, those are fully qualified to prevent ambiguities. However, it can be any string and any type of comment (as one can see in the Listing).

For explicit links in UML a profile is generated, containing a set of stereotypes used to set explicit `Navigation Links`. Profiles in UML allow providing stereotypes for specific types, allowing that`Navigation Links` are only provided for types for which in the third metamodel a possible `Navigation Link` to another model is defined. Such a generated stereotype also has an `id`-field, generating the possible match with its (Java-)code opposite.

5.2 Explicit Consistency Rules

An initial set of consistency rules was defined to explore capabilities of MCCC, an overview of those can be seen in Table 6. We developed consistency rules for the following three types of UML diagrams: i) class diagrams, ii) sequence diagrams, and iii) state machine diagrams.

ID	Name	Context	Description
CR1	AssociationConform	uml::Association	Each memberEnd of an association has a corresponding field in code.
CR2	AssociationFieldConform	uml::Association	Variation of AssociationConform. Association has direct navigation link to code.
CR3	ClassFieldConform	uml::Property	A field with the same name, type and modifiers as in UML exists.
CR4	ClassMethodConform	uml::Operation	Every UML::Operation has a corresponding Java::Method.
CR5	EnumConstantConform	uml::EnumerationLiteral	An Java::EnumConstant exists that has the same name and at least one value equal to the ones specified in UML.
CR6	UsageConform	uml::Usage	In the using element, there exists a variable, parameter or return type of the used element.
CR7	GeneralizationConform	uml::Class	A Java::Class inherits the same classes as its corresponding UML::Class.
CR8	InterfaceConform	uml::Class	The same interfaces are implemented for both Model and Code
CR9	InteractConform	uml::Interaction	A specified sequence of messages in a sequence diagram must be found in Code.
CR10	LineConform	uml::Lifeline	Incoming messages must exist and have the same outgoing calls.
CR11	StatePattern	uml::State	A verification of the state pattern from Gamme et al. [26].
CR12	SyncStateDiagram	uml::StateMachine	Correct implementation of a synchronized data structure.
CR13	CallSequenceStatechartConform	uml::StateMachine	Is a sequence of calls to a class allowed from its state machine.

Table 6: Consistency Rules

Class diagrams describe the static structure of a software system. *AssociationConform* evaluates if an association among two `Model` elements is also to be found in `Code`. A variation is *AssociationFieldConform*, instead of using `Navigation Links` on class level, this rule uses links from an `UML::Association` directly to the corresponding `Java::Field`(s). All `UML::Property`s of an `UML::Class` should have a corresponding implementation element (with the same name, type, multiplicity), *ClassFieldConform* tries to detect inconsistencies. *ClassMethodConform* does the same but for `UML::Operations`.

An enumeration defined in UML should have a corresponding `Java::Enum` with the same enumerators. Since `UML::EnumerationLiteral`s are only capable of representing one value, the enumerator should be found in the list of possible values of a corresponding `Java::EnumConstant`. An `UML::Usage` can be interpreted as follows, in the using type should exist a field, parameter, return type or a local variable of the used type. *UsageConform* tries to find one of those examples given. *InterfaceConform* and *GeneralizationConform* evaluate if a pair of `Model` and `Code` elements implement (or inherit) the same types.

Sequence diagrams describe the communication among certain design model elements. *InteractConform* evaluates if a sequence of messages and control structures can be found in a `Code` element, context is a `UML::Interaction`. *LineConform* does similar but only in the context of a `UML::Lifeline`, incoming calls must be found as methods in a `Java::Class` and in this method all outgoing calls must be contained in the correct order. For both rules holds, in between UML calls can be arbitrarily many Java statements. However, this semantics can easily be tightened, to not allow in between Java statements so that it becomes "exactly this sequence and nothing less or more".

State machine diagrams describe runtime behavior of a system or certain artifact elements. *StatePattern* evaluates if a state pattern [26] is correctly implemented. *SyncStateDiagram* is in the context of synchronized data buffers and evaluates correct thread handling. State machines can be used to model the correct usage of a certain `Object` (e.g., File: `open(){read()}close()`). *CallSequenceStateChartConform* evaluates if a sequence of calls to e.g., a `File-Object` comply to a state machine. Other UML diagrams, such as component-, object-, deployment-diagrams provide a view of the modeled system that is not accessible from the source code artifact alone. However, although this set of rules evaluates important concepts of consistency among `Model` and `Code` it still is an initial set that can be extended and customized to match the needs of a developer or a specific project. To define those consistency rules, we use the *Object Constraint Language (OCL)* [27], which is one of the most common constraint languages and typically used to write invariants that must hold within a single UML model. To discuss changes to the metamodels of the respective artifacts and to the constraint language an actual constraint, *ClassFieldConform* (as shown in Listing 7), will be discussed.

Note from the consistency rule in Listing 7 that the constraint language remains unmodified, only references supporting the navigation had been added to either metamodel. For instance, consider the `javaclass` - reference in line 3 of Listing 7, in this case the specified (java-)opposite of the `UML::Class` is returned. The prosa definition of the given constraint is as follows: Select all `Java::Fields` from the specified opposite `Java::Class`. In this set (if not empty) should

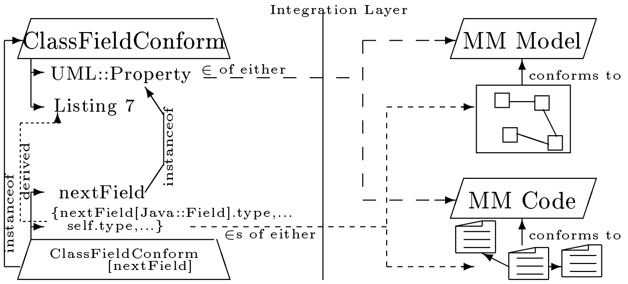


Figure 10: ClassFieldConform[nextField]

exist a `javaField` with the same name and correct translation of multiplicity . Subsequent aspects of consistency, such as the same visibility modifiers (`public`, `private`,..), type conformity and finally `static` and `abstract` are not depicted for space reasons.

5.3 Incremental Evaluation

Following, we show how MCCC is aware of changes of either `Model` or `Code`, in order to re-evaluate consistency rule instances, holding the changed elements. Following, we discuss how such changes are handled efficiently in our approach, starting with a design model adaptation, followed by a source code adaptation.

5.3.1 Model Evaluation

Recall the example in Section 3.2.1 the type of the `Model` element `nextField`[`UML::Property`] was changed to `Cell`. The consistency checker should, after performing the change, evaluate the rule instance and show its

```
[Context: uml::Property]
let fields : Set(members::Field) =
 self.class.javaclass.getFields() in
if not(fields->isEmpty()) then
fields->select(javaField |
 javaField.name = self.name
 and
   self.upperValue = -1 implies
    (field.arrayDimension > 0 or
    javaField.type
      .oclAsType(Java::TypeReference).target
      .oclIsTypeOf(java.lang.Collection))
   or ...
 static, visibility, abstract ...
)else false endif
```

Listing 7: ClassFieldConform

inconsistency. In Fig. 7 the developer directly edits a `Model` element, in this case the type of `nextField` is set to `Cell`. Figure 10 shows the current instantiation of this particular consistency rule. The $\triangle_{model}$ for the `Model` is specified by `nextField` of type `UML::Property` and its property `type`, which is set from `Boolean` to `Cell`. Given to the consistency checker, if the changed property is in the scope of `ClassFieldConform[nextField]`, a re-evaluation of this consistency rule instance is triggered. Figure 11 now also states the inconsistency of the pair, this correctness feedback is already output from the MCCC tool.


✖ ClassFieldConform[nextField]

Figure 11: ClassFieldConform[nextField]

5.3.2 Code Evaluation

The process for `Code` is similar, but has one additional stage. A developer does not directly change the code model in-memory; at each change of an edited `Source`-element the corresponding model representation has to be updated. Recall the discussion in Section 3.2.2 about sequence diagrams. The Fig. 2 and the Listing 2 actually are inconsistent. The call to `JGameOfLifeView` starting the visualization was misplaced (line 5). It should instead be placed (according to the sequence diagram Fig.2) between `show()` and `createRandom()`. If a developer now re-positions this call, parsing the edited element and subsequently comparison with the in-memory version is triggered, resulting in ($\triangle_{mod.artifact}$). Based on the compare information, the method is updated and the call is copied at the corresponding place in the method. This update changes the statements of the method `main`, $\triangle_{code}$ is then specified with type `Java::ClassMethod`, and property `statements`. This change is now forwarded to the consistency checker, which a re-evaluates all consistency rule instances which scope contains `statements` of the given `Java::ClassMethod`. Now Figure 12 states the consistency of both elements.


✓ InteractConform[Initialization]

Figure 12: InteractConform[Initialization]

6 Validation

To demonstrate both feasibility and scalability of MCCC, we developed a prototype implementation that was used to conduct a case study on 10 mostly industrial projects. Moreover, in this section we also discuss the general applicability and possible limitations of our approach. We start with a description of the prototype.

6.1 Tool

Our MCCC approach was implemented as a set of plug-ins for the *IBM Rational Software Architect (IBM RSA)* [19]. Therefore, our prototype provides a single solution to modelling, implementation, and feedback about consistency – it is available at [28]. EMF Core (*Ecore*) has been chosen as common metamodel in that `Model` and `Code` are represented. This allows that elements of either artifact can be treated equally and the possibility to import metamodels enables data integration.

Incremental Consistency Checker. As incremental consistency checker, the *Model/Analyzer* [23] is employed. Specifically, an EMF-based version that allows for consistency checking of arbitrary EMF models was used.

Metamodels and Navigation. EMF is basically a standard to implement DSL's [29, 30], because of that reason it was used to express the model representation of the design model and source code. To generate an Ecore representation of Java source code, the *Java Model Parser and Printer (JaMoPP)* [31] was used.

Programming Language Support. MCCC technically supports any programming language for that an `Ecore` representation can be obtained. Practically, this means that a full metamodel (/language specification) of the programming language needs to be specified in ECore. EMF then uses the *ANTLR* language recognition tool to generate a text parser. Given this, the ECore-file, containing the currently specified `Navigation Links`, needs to be updated to introduce new explicit links. Furthermore, a handler class must be implemented to search for explicit links. Next, we discuss the case study we performed using the prototype implementation.

6.2 Case Study: UML and Java

Since MCCC is expected to be applied in large-scale software development environments with industrial-size models and large code bases, both computational scalability (i.e., the time it takes to (re-)evaluate consistency after changes) and memory consumption (i.e., the space in memory required by `Navigation Links`, `Ecore` code model, and the consistency checking mechanism itself) are critical factors for its practical value. To asses our approach in those factors, we performed a case study on projects of different sizes and domains. UML was chosen as modelling language as it is basically an industry standard for object oriented modelling. As programming language Java, which is mostly ranked in the top 5 of TIOBE's community ranking [32].

6.2.1 Projects

For the case study, we used 10 mostly industrial projects, an overview is given in Table 7.[1] `#Instance` specifies the number of design rule instantiations, `#Model Elements` (`UML::Class`, `UML::Association`, etc.) and `#Code Elements` (`Java::Class`, `UML::Field`, etc.) the size of `Model` and `Code`, respectively. All 13 rules from Table 6 were applied to each project. For projects with missing model, the design was reverse engineered from the source code. The projects span a wide range of sizes, from small single-developer to large multi-developer projects. `GameOfLife` is the project presented in this paper and the smallest in the test set with 286 `Model` elements and 261 `Code` elements. The biggest project in the test set is the `ArgoUML` tool [20], with 6,039 `Model` elements and 19,557 `Code` elements.

ArgoUML ArgoUML is an UML diagram editor written in Java, it is released under the *open source Eclipse Public License* [33]. It does not yet support the complete UML specification. The tool started as Ph.D. thesis by Jason E. Robbins at UC Irvine but then became an open source project.

The extracted diagram is a class diagram, showing all the used entities and their relations.

ATMExample ATMExample is a simulation of an automated teller machine. Hardware of the ATM is simulated, GUI elements and transactions to and from

---

[1] A detailed description of each project can be found at [28].

	Project Name	Class	Sequence	Statechart	#Model Elements	#Code Elements	#Instances
PR01	GameOfLife	x	x	x	286	261	83
PR02	TestService	x	x	x	397	302	157
PR03	Checkers	x	x		425	342	95
PR04	ATMExample	x	x		1,341	1,550	601
PR05	SMTSolver	x	x		1,254	2,021	400
PR06*	TaxiSystem	x	x		1,930	3,488	608
PR07	ObstacleRace	x	x		1,992	3,555	886
PR08	VOD3	x	x		2,538	3,613	857
PR09	biter	x			2,648	4,015	1,244
PR10*	ArgoUML	x			6,039	19,557	2,005

* Design model was reverse engineered using [19]

Table 7: Case Study Projects

the bank are emulated. A simulated bank handles the incoming transactions – is withdrawal over limit? –, manages accounts and security.

The class diagram of this project depicts an overview of the system. Sequence diagrams show, session management, startup, shutdown and transaction handling.

biter This project implements and describes a robot to participate in a robocup. The robot maintains a world model, containing itself, other players, opponents and the ball. Strategical calculations down to calculations for positioning or how to best kick the ball are handled.

Several class diagrams describe not only the overview of the system but as well more detailed aspects of the robot.

Checkers Checkers, or draughts, is a strategy board game. On a chess board (8x8) a player has 12 pieces, which can move and capture only diagonally. Straight movements are only allowed after a piece has been crowned. This project implements this game and as well provides a GUI.

The class diagram shows a detailed overview of the game. Several sequence diagrams show algorithms of the system (i.e., if a move is legal).

GameOfLife This project was used throughout the thesis to discuss concepts of inconsistent development artifacts. It is an implementation of Conway's GoL, a cellular automaton devised in 1970. The game is a non-player game, the progress is completely determined by its initial state and rules inherit to the game. The initial state is a two-dimensional orthogonal grid of square cells, which is either dead or alive. Each cell, regardless of dead or alive, interacts with its neighbour cells (at maximum 8 cells, which are directly adjacent) and determines from the the following rules its next state:

- A living cell with fewer than two neighbours dies of under-population.
- A living cell with two or three neighbours survives into the next generation.
- A living cell with more than three neighbours dies of over-population.
- A dead cell with exactly three living neighbour cells becomes a living cell in the next generation (reproduction).

GoL is a simplification of a problem stated by John von Neumann in 1940, who attempted to build a hypothetical machine reproducing itself.

The diagrams used for this project where mostly discussed throughout the paper.

SMTSolver An *Satisfiable Modulo Theories (SMT)* [?] solver framework for Java. SMT is a decision problem for logical formulas, in which different background theories are combined and expressed in first-order logic. Examples of such theories are, theory of real numbers, theory of integers, theory of data structures (e.g., list, array or bit vectors).

For this project a class diagram were given and sequence diagrams specifying initialization of data structures and algorithms used.

TaxiSystem An implementation of a taxi system, drivers have to log on/off to a central report cab rides and as well are observed by a GPS.

The class diagram shows the system overview, sequence diagrams show detailed interactions like logging on, logging of or rejects.

TestService The project we used to test our approach during implementation.

It contains a loosely coupled set of class-, sequence- and state machine-diagrams. Different semantics were collected from code generators and as well from examples employed from books.

VOD3 An implementation of a video on demand system. Contained is a MPEG decoder, server, client and communication among those. This part of the system contains no GUI.

Class diagrams show the entities of the system. Sequence diagrams depict in detail receiving and decoding headers, receiving and processing single frames of a movie. The process of selecting and playing/stopping is as well detailed modelled in detail.

6.2.2 Usability

The OCL definition was introduced in UML in release version 1.3 1996, followed by an important revision 2003 labeled as 2.0. OCL, as previously stated, is well known constraint language. Thus, as OCL usually constrains UML models, the Java metamodel is the only unknown part to developers. This circumstance allows the assumption that the initial learning phase is quite short for someone who already knows OCL and UML. Part of the definition of usability (defined from Jakob Nielsen [34]) is learnability since our tool requires the user to learn only one additional metamodel, it is save to assume it fulfills this requirement.

As well for the rest of the definition: efficiency, see further on; memorability, the tool extends the IBM RSA-GUI and follows the style guidelines; errors, e.g. syntax failure, feedback is provided to recover from failure and finally satisfaction is given by the satisfying the previous points.

6.2.3 Computational Scalability

To validate the computational scalability of our approach, we systematically changed elements in each of our 10 projects and captured the time required for processing the change (i.e., the time required for re-evaluation all consistency rule instances whose scope contained the changed element). Specifically, we ensured that each element that was present in a consistency rule instance's scope was changed. Furthermore, each element change triggered a full re-evaluation of all affected consistency rules instances. Raw data of the tests can be accessed at [28].

Mean and median times were observed and used for our analysis.[2]

For the validation we used an Intel(R) Core i7-3610QM machine with 8GB of memory running Windows 7 Professional. Figure 13(a) displays the mean

---

[2] Note that we did not use any optimization mechanisms (i.e., [35]) for the Model/Analyzer in order to produce valid and comparable results for incremental consistency checkers.

processing times per affected consistency rule instance for a change depending on the project. Additional Fig. 13(b) shows the mean affected instances for a change. Although total processing times do depend on the project, note that observed processing times per affected instance stay in the interval of minimum 30 ms and 85 maximum ms for all projects. This indicates that the total processing time does depend primarily on project-specific model characteristics but not on model sizes or the total number of consistency rule instances. Moreover, the total processing times also remain below 50 ms on average, which is still an acceptable time for tool users [34].

The time required to evaluate a consistency rule instance is determined primarily by the size of its scope, which depends on how the consistency rule is written, how it uses `Navigation Links`, and also the specific characteristics of the evaluated elements (e.g., the number of elements in an accessed collection).

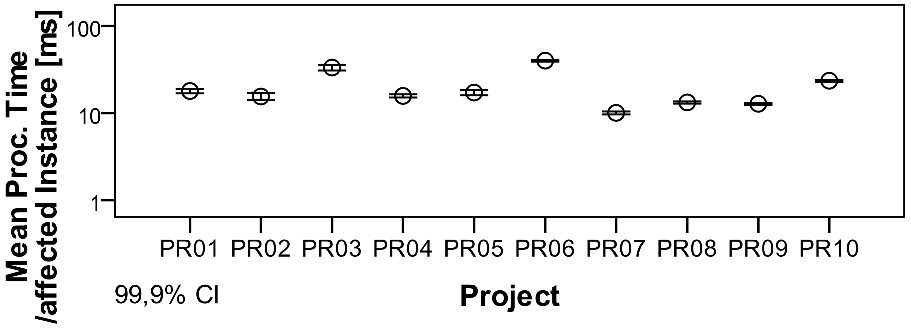
In Fig. 13(c), the mean number of scope elements per rule instance is shown for each project. The results are similar to those observed in previous validations of the Model/Analyzer [17] where similar models but only consistency rules for checking consistency within design models where used. Note that scope sizes remain relatively constant except for PR10.

This is due to the characteristics of the project itself, most consistency rule instances of PR10 have a scope size similar to the other projects, whereas few instances consist of a scope size beyond 200 elements. However, this does not affect the mean processing time per rule instance significantly, as shown in Fig. 13(a).

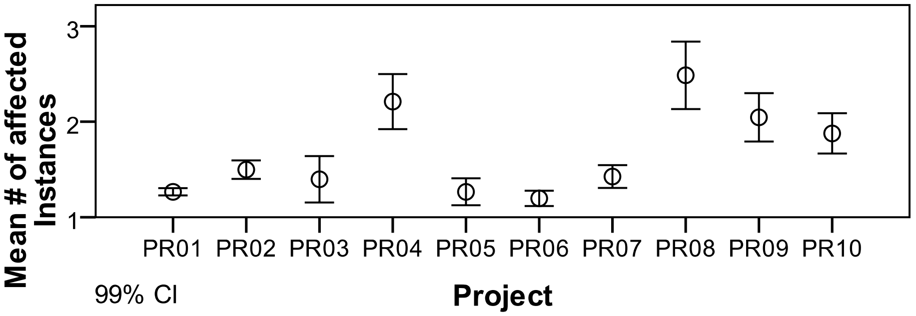
Overall, the obtained results indicate that the evaluations times of approach and consistency rules scale and that MCCC provides immediate feedback to developers.

6.2.4 Memory Consumption

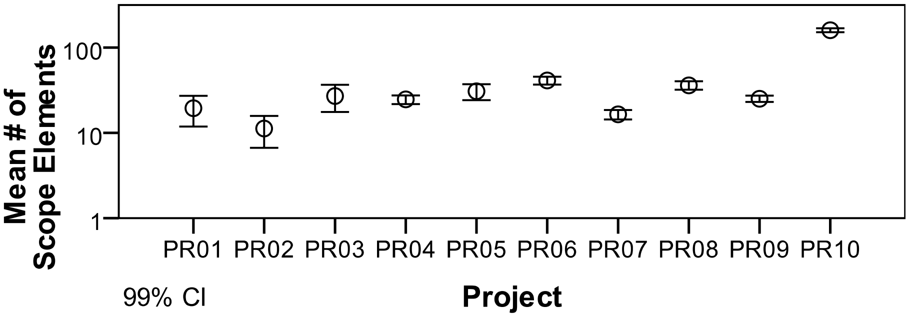
Given that the evaluation mechanism uses a scope increases the evaluation performance at the cost of increased memory consumption. Therefore, we have to investigate the memory consumption increase with increasing project size. Three factors influence the memory consumption of MCCC: i) storage of `Navigation Links`, ii) memory consumption of the Ecore code representation, and iii) memory consumption of the evaluation mechanism. For point i), we observed that memory consumption was negligible as navigation links are implemented in a simple map of a `String` to an `Object`-reference. For point



(a) Comparison of Total Processing Time



(b) Comparison of affected Instances



(c) Comparison of Scope Size

Figure 13: Validation Results

ii), our case studies showed that for all projects the in-memory representation required less space than the on-disk plain text version. Finally, for point iii), Fig. 13(c) depicts that there is little correlation between mean scope sizes and project sizes, indicating that memory consumption per rule instance is generally independent of the project size. The total memory consumption is expected to grow linearly with the number of consistency rule instances, which typically grow linearly with project size [17]. Therefore, the conducted case study showed that our approach scales in both terms, processing times and memory consumption. MCCC is thus suitable for being used in large-scale development projects.

6.3 Applicability and Limitations

Applicability. As the conducted case study has shown, our approach is applicable to UML as modeling language and Java as programming language and shows efficiently inconsistencies among those. However, our consistency rules assume that the `Model` describes elements that can also be found in the `Code` (therefore the `Model` should be on *nearly* implementation level). Consider now that the `Model` could also only show the domain entities. Thus, on a higher level of abstraction than the implementation. Reasoning about model-and-code consistency then can not produce valid assumptions. On the other hand, our approach needs at least in principle a difference in either level of abstraction and/or semantics between design model and source code to check consistency. If, for example, the code representation is on the same abstraction level and semantically equal to its model, the implementation is the design model itself. Another benefit of using an `Integration Layer` and providing a single metamodel view to the consistency checker is the possibility of integrating multiple development artifacts of the same kind (e.g., source code in different programming languages or multiple design models that conform to different metamodels). However, this increases slightly the complexity regarding the heuristics-based calculation of links. Despite that, it would not affect consistency rules – except for more references available for navigation.

Development Process. Development projects often start with only one artifact available, relying on code generators or reverse engineering techniques to produce the second artifact automatically. However, our approach needs (initial versions of) both artifacts to provide results. Thus, MCCC does not aim to replace artifact-generating technologies but it provides a complementary technology that addresses existing issues. In practice, such technologies can be used to get an initial skeleton of the implementation or a reverse engineered design model automatically. Our approach can then detect inconsistencies between initial artifact versions that are based on unintended semantics used by the code generator or reverse engineering method, respectively. During further evolution of artifacts during development, MCCC detects inconsistencies introduced by either developer.

Feedback. Each detected inconsistency provides meaningful feedback to developers. In addition to simply informing them about the existence of an inconsistency, it also contains information about the location and the kind of the inconsistency. Moreover, it can be utilized to provide to developers a list of possible repairs automatically [11]. Although those repairs are often abstract and require active user input, they also reduce significantly the effort of finding suitable solutions for inconsistencies.

Technical Limitations. Regarding the prototype implementation, there are only few prerequisites a project must adhere for our tool to be applied, as discussed in Section 6.1. Specifically, the tool requires `Ecore` model representations from design model and source code. However, technologies for generating and managing such representations from source code are available (e.g., [31, 6]) and common modelling tools are based on EMF [19, 20]. Thus, the restriction that a model-based representation is needed does not limit the general applicability of our approach in practice.

Overall, we believe that our approach is generally applicable and is an enhancement to MDE-based software processes.

7 OCL Consistency Rules

Table 6 gives an overview of the currently implemented consistency rules and a short description, but neither was discussed in detail. In Section 10 all constraints and queries are listed, partially shortened for better readability. In the following sections each consistency rule and the queries used by those will be discussed. For each rule holds, feedback will be provided for specific `Model` or `Code` elements, to enable developers to deduce the location of the inconsistency. For instance, instead of validating a `UML::Class` on its whole, with all its properties and operations, we decided to evaluate a `UML::Property` or `UML::Operation` in specific.

7.1 Class Diagram

Class diagrams describe the domain elements and their relations used in a software system. Chosen elements for consistency rules were, `UML::Association`, `UML::Usage`, `UML::Property`, `UML::Operation`, `UML::EnumerationLiteral`, interface-implementation and inheritance.

7.1.1 AssociationConform

`UML::Associations` describe relations among design model elements that are expressed as properties of a `UML:Class`, `UML::Interface` or `UML::Enumeration`. These should then be found in source code as well (e.g., for a directed association: the (Java-)implementation of the using UML type should contain a `Java::Field` of the (Java-)implementation of the used UML type). For undirected associations both ends should contain such a `Java::Field`. `UML::Associations` also specify the multiplicity of both ends, so a correct translation should be ensured.

Context of `AssociationConform` is `UML::Association`, the rule differentiates between directed associations (first branch, beginning at line 3) and undirected associations (else branch, beginning at line 16). In both branches `Java::Field`(s) of the corresponding types are searched.

7.1.2 AssociationFieldConform

This rule is an alteration of `AssociationConform`, which uses `Navigation Links` on `UML::Class` level. This rule, however, uses links of `UML::Associations` directly to its implementing `Java::Field`(s). Intent of this rule was to show consequences of different `Navigation Links`. A possible consequence of this link could have been a shorter overall rule. However, it is not, after evaluating the type conformance, additionally one has to prove that the fields are contained in corresponding types (line 5- 12). For undirected associations, two `Java::Fields` have to be considered, stored in `fieldmtrace` line 17. Evaluating that a field with a correct type was linked, is done by following the `typeReference` (line 3) of the `Java::Field`.

7.1.3 ClassFieldConform

Starting from a `UML::Property` the rule first searches the properties parent (line 2- 9)(`UML::Class`, `UML::Enumeration` or `UML::Interface`). Given this parent, the corresponding Java-type is accessed and all its fields are stored in a set (line 5). From this set, the rule filters all `Java::Fields` with different names (line 11). Doing this prevents that more costly queries are executed on all `Java::Fields` of the type. The `Java::Field` should then have the same type (line 14), the same visibility modifiers (line 15, 16), a correct translation of multiplicities and exhibit the same abstract and static-modifiers.

7.1.4 ClassMethodConform

`ClassMethodConform` works similar as `ClassFieldConform`, defined on the context of a `UML::Operation`, it as well selects all `Java::Methods` of the corresponding Java type of its parent (line 2- 9). Those methods are pre-filtered based on their name (line 11, 12). The method should then exhibit the same return type (line 14), all of the parameters should be type and multiplicity conform (line 16- 19). If the `UML::Method` is abstract, the implementing `Java::Method` should as well be abstract (line 20, 21), this should as well hold for the static modifier (line 22, 23). The parameter list size for a `Model` and `Code` operation pair should as well be equal. It is not guaranteed after one has proven that for each UML parameter type exists a corresponding Java parameter type, that they also have the same amount of parameters – two UML types could map to the same Java type. `UML::Operations` consider their return type as a normal parameter, `Java::Methods` not, this has to be considered, resulting in the lines 24- 33.

7.1.5 EnumConstantConform

This consistency rule, like `ClassFieldConform` and `ClassMethodConform`, intends to find for a given `UML::EnumerationLiteral` a corresponding `Java::EnumConstant`. As well a pre-selection based on the name (line 4) is done. UML allows only one value for each literal, based on its type (if-expression line 12- 27) a corresponding Java value is searched in the list of possible constants (line 5) of the Java enumeration. An example are integer values (starting from line 12), if the `specification` of the `UML::EnumerationLiteral` is an integer, it is compared against a possible integer value of `enumConstant`. Complex types are especially considered, starting from line 27, those are in UML `InstanceValue`-types. If the `UML::EnumerationLiteral` is of a complex type, there should exist a (line 28) `UML::InstanceValue` in the list of `Java::EnumConstants` with a `Navigation Link` to the UML - type (line 30, 31)

7.1.6 GeneralizationConform

From the viewpoint of a `UML::Class GeneralizationConform` evaluates if a super class exists (line 10), which has a `Navigation Link` to the parent of the corresponding `Java::Class`. There should only exist one, since Java does not support multiple inheritance.

7.1.7 InterfaceConform

All implemented interfaces of an `UML::Class` should have a `Navigation Link` (line 6) leading to one of the implemented interfaces of a `Java::Class`.

7.1.8 UsageConform

`UML::Usages` can have four different manifestations: i) In the using type exists a field of the used type (line 2- 17). ii) There exists a method that returns the used type (line 32). iii) A method exists with a parameter of the used type (line 33, 34). iv) It is used as a local variable type (line 35- 39).

7.1.9 Queries

The following queries are used among all the previous discussed consistency rules, they are used to compare visibility, multiplicity and prove type conformance.

visibilityConform In the context of a `Java::Modifier` – that can be of type, `Public`, `Private` or `Protected` – this query compares the `UML::VisibilityKind` with the instance of the Java visibility modifier.

multiplicityConform Currently the following translations for multiplicities are allowed, if the `upperValue` of the UML multiplicity is either -1 or $>$ 1 , then its allowed implementations can either be an array or a subtype of `java.util.Collection` (line 5- 21). Otherwise, if the value is 0 or 1, then it should neither be an array nor a reference to a collection (line 24- 40).

typeConform This query provides mappings of `UML::PrimitiveTypes` to Java primitive types (line 7- 24). Since `String` is a complex type in Java and in UML a primitive type it needs to be especially considered (line 48- 61). Mappings of complex types, that are used in the project, are done by following the `Navigation Link` and comparing it with the given to match type (e.g., line 31, 32).

containsType Context of this query is a `UML::Type` that should be found in a set of `Java::Statements`. `Java::Statements` can be arbitrarily nested (e.g., if-expression, while-loop, try-catch block). For each of those sub-blocks of statements, the query is called again.

7.2 Sequence Diagram

Sequence diagrams model the interaction among certain design model elements. Sequence diagrams itself have different interpretations: i) An implementation should contain only the specified order of calls and nothing else. ii) Different calls can be intertwined and specified calls can also be in subcalls. Assignments, calls to collections or calculations in general are examples of intertwined statements that will not be modelled by a developer and as well are not in the scope of sequence diagrams. The semantics applied for sequence diagrams is that several statements can be intertwined in the specified sequence of calls.

7.2.1 InteractConform

A `UML::Interaction` defines an interaction among multiple UML elements. From the viewpoint of a `UML::Interaction`, all `UML::BehaviorExecutionSpecifications` are selected and its corresponding outgoing calls and control flow structures are searched in the correct order in the associated implementation.

7.2.2 LineConform

`LineConform` employs a more narrow view than `InteractConform`. This rule selects incoming calls to a `UML::Lifeline`. For each of those incoming calls, its outgoing calls and control flow structures are selected. The `UML::Lifeline` has a type associated, the corresponding Java type should then own a method – representing the incoming call –, containing the sequence of outgoing calls and control flow structures.

7.2.3 Queries

Both consistency rules consist only of a call to a query, consistency rules itself can not be recursive and allow no loops – and are therefore not Turing complete. However, to evaluate such complex properties, a construct like loops is necessary. Therefore, the previous consistency rules were implemented in queries (presented further on) as they can be recursive.

validateSubInteractionFragments/validateInteractionFragments
Both queries are very similar, they differ only in details, therefore only `validateSubInteractionFragments` will be discussed in detail. This query first selects all `UML::BehaviorExecutionSpecifictions`, for each of those, all contained `UML::InteractionFragments` (all outgoing calls, `UML::BehaviorExecutionSpecifiction`, and control flow structures, `UML::CombinedFragments`= are selected (line 15- 40). For incoming calls, the implementing method is selected in the `Java::Class`. Given this method, it should contain the previously selected `UML::InteractionFragments` in the order they are specified (`containsFragmentsInOrder` line 57). All `UML::CombinedFragments` that are on the same level as incoming call must be considered separately (line 63- 67).

containsFragmentsInOrder To this query two parameters are passed, a `Sequence(UML::InteractionFragment)` and a `Sequence(Java::Statement)`. The query considers those not as sequences but rather as queues. Based on the first element in the `fragments` queue it searches for a corresponding element in the `statements` queue. For example, if the first element in `fragments` is a `UML::BehaviorExecutionSpecification` it tries to find a corresponding call in the queue of `statements` (line 10- 25). If a statement does not contain the call, the first element of `statement` is discarded and `containsFragmentsInOrder` is called again (line 19, 20). If a statement contains the call, the first elements of `statements` and `fragments` are removed and `containsFragmentsInOrder` is called again, now searching for the next `UML::InteractionFragment` (line 22, 23).

containsCall This query is used to find the searched `UML::InteractionFragment` in any form of Java-element (e.g., statement, condition, ...). The callee-type from the `UML::MessageOccurrenceSpecification` is stored in a temporary variable (line 11- 10). Then all `Java::MethodCalls` are selected from the incoming `Java::Commentable` (lines 11- 17). If in this set of calls exists a call with the corresponding name (as given by the `UML::MessageOccurrenceSpecification`) and the correct callee-type `true` is returned.

existsCombinedFragment UML sequence diagrams uses different fragments, that model control flow, such as `loop`, `alt`, `opt`, `par` and many more. Nevertheless, for this work we only consider `loop`, `alt` and `opt`. This query provides possible match ups for those fragments. If a `loop` turns up, corresponding implementing statements can be `ForLoop`, `WhileLoop`, `ForeachLoop` and `DoWhileLoop`.

7.3 State Machine Diagram

The runtime behavior of objects or even whole systems is modelled in state machine diagrams. Exactly this property makes it hard to devise a general semantics, since any two classes can exhibit totally different runtime behavior. Therefore, we searched for situations, which can occur in different amendments, but nevertheless are the same. Such situations were: an implementation of the StatePattern [26], synchronizing buffers and state based resources (e.g., file stream, network connection).

7.3.1 CallSequenceStateChartConform

This consistency rule assumes that calls to an `Object` alter the state of it (e.g., a file that is opened). These states and possible transitions to and from a state are modelled in a state machine diagram. Thus, this rule evaluates if a sequence of calls to an `Object` conform to its constricting state machine. At first the initial state is selected (line 3, 4), then the state to which the initial state points is stored in a temporary variable (line 5). In the lines 11- 13 a query is called to calculate which state is reached after the sequence of calls. If this state is the initial state or is of type `UML::FinalState`, a query calculates the reached score, otherwise (if no final or the initial state is reached) it is considered as inconsistent. Recall Table 5, it summarizes a pair of corresponding state machine and implementation with uncertainty.

calculateScore This query calculates the score reached by a sequence of calls. It treats all incoming statements as queue and tries to find a match to a sequence of transitions from a given state chart in arbitrary nested expressions. If a statement is a container (line 8- 35), except a try-catch block, the result of this subsequence is weighted with a constant factor to express uncertainty (line 28). For other statements, at first the state reached up to this point is calculated (line 36). If the first statement is a call to another method (`isMethodCall` line 40), then all its statements are as well considered (line 41- 51). At last if the first statement in the queue is a transition to the next state (line 56) its target is stored in a temporary variable – possible next states are selected from the outgoing transitions of the current state (line 53- 54). Following, the first statement can be discarded from the queue and the score of the rest of sequence can be computed (line 58, 61).

calculateStateToStatement `calculateStateToStatement` is similar to `calculateScore` but instead of calculating a score it calculates the reached `Model`-state by a sequence of Java statements.

isMethodCall If a `Java::Statement` contains a method call, the callee type is returned.

isTransitionToNextState A transition in this semantics is considered as a call to an `Object`, modifying the state of it. If the callee type matches the target of the transition and the operation name matches the call event name, respectively, the selected `Java::IdentifierReference` leads to the next state of the `Object` (line 23- 25).

7.3.2 StatePattern

A general semantics for implementing state machines is devised in [26]. This pattern is commonly known and widely used, this consistency rule tries to evaluate the correct implementation of the modeled state machine. The consistency rule is defined in the context of a `State`, in this semantics each state is implemented in one `Java::Class`. Therefore each state has an explicit link to its corresponding implementation. In Fig. 14 one example of such a pattern is shown. State transitions are calls to the context variable, as in Listing 8 line 6. A transition consists of two parts, the method containing the call to the context variable and its corresponding condition. The condition should be found in source code exactly the way its specified in the state machine. This is done from line 8 to 30 in the consistency rule.

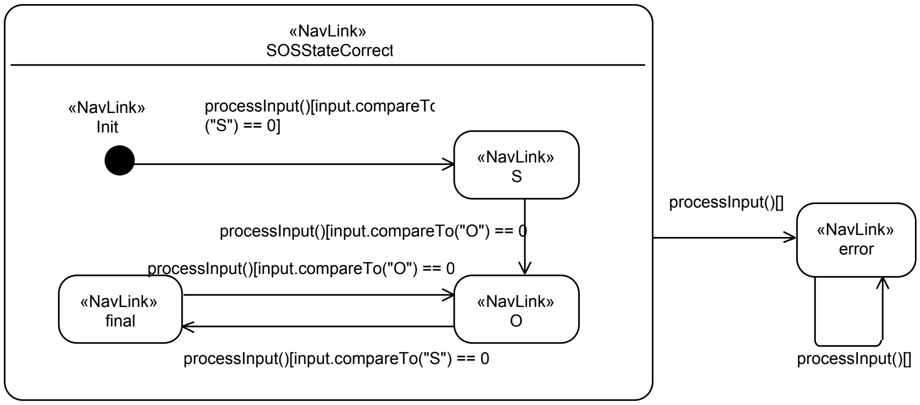


Figure 14: SOS State Pattern

To be more specific, first the method has to found in the Java source code (line 9- 13). All statements are traversed to search for the call, which sets the context-variable to the next state (line 15). If a state has subregions (e.g., `SOSCorrect`), in all its sub-states a call to the superstate (line 25) should be found, to handle the rest of the input (line 17- 28).

Listing 8 shows an implementation of a state, shown is the `processInput` method, if the input is `"S"` the context variable is set to the final state. Also the super call in line 8 can be seen, handling the rest of the input.

```
/* @NavLink(id="SOSStateO") */
public class SOSStateO extends SOSStateCorrect {
  @Override
  public void processInput(SOSContext context, String input) {
    if (input.compareTo("S") == 0)
      context.setState(new SOSStateFinal());
    else
      super.processInput(context, input);
  }
}
```

Listing 8: Code Snippet SOSStateO

traverseStatements The query traverses all `Java::Statements` passed as arguments in order to find an implementation of the `UML::Transition`. At first the possibility has to be considered that this statement is actually a container, then all of its sub-statements have to be observed. At last, if it is no container, a possible transition to the next state is searched in the `Java::Statement`. If the `UML::Transition` is not found in the `Java::Statement`, the statement is excluded and the query is called again.

containsTransition Transitions to a next state can be of two forms, either with guard condition (line 5- 20) or without (line 21- 25). If a corresponding pair of guard (implemented as condition) and call event (implemented as call) can be found (`isTransitionToNextStateClass`) it is considered as transition to the next state.

isTransitionToNextStateClass State transitions in this semantics are calls to the `context`-variable, setting the new state variable. The query at first gets a possible `Java::IdentifierReference` in a `Java::Statement`, from this reference the target is extracted and compared against the Java-opposite of the (UML) context variable. Then, in the method call the next state should either be found as a constructor call, or, in case of a self loop, as reference to itself.

searchForSuperCall This query traverses all as arguments passed statements in search of a super call.

isSuperCall This query searches for a super call in a statement. First a possible `Java::SelfReference` (these encode also super-references) is stored in a temporary variable. If the super type reference and the corresponding method call name match the transitions source and the transitions call event name, respectively, it is the searched super call.

7.3.3 SyncStateDiagram

Synchronizing buffers may in general exhibit some characteristics that are common to them (e.g., thread handling). At first this rule selects the state characterizing the "metastate" (in general normal buffering, e.g., Fig. 3 `Buffering`) by selecting the target the initial state points to. Then all transitions of the state machine must be contained in the `Java::Class` (line 12- 13). Special states like `FullWait` or `EmptyWait` of Fig. 3 have a direct `Navigation Link` to their corresponding statements.

isContainedWithinJavaClass Two possibilities can exist for transitions, either they lead to the "metastate", then its corresponding implementation should contain a `notifyAll`-call or they lead to a special state, identified by a `Navigation Link`.

methodLeadsToState This query searches in the incoming statements for a condition that is coherent with the state machine. If a corresponding condition is found, the if-expression body should contain the searched `targetStatement`.

8 Related Work

MDE is a wide and active field of study, different challenges such as modelling languages, separation of concern, and model manipulation and management have to be faced. Following, research related to our approach will be discussed. The *Eclipse Modeling Framework (EMF)* [6] is a modelling framework with a code generation mechanism. A new annotation to mark source elements as generated is introduced, presence or absence determines if the associated code element is overwritten at code regeneration. In contrast to MCCC, EMF generates code and tries to handle the update problem via a `@generated` annotation. Nevertheless, adaptations of the code generator to adapt to specific needs are limited to certain details and does not allow full control.

Zheng et al. [3] introduced an approach called *1.x-way architecture-implementation mapping*, implemented in ArchStudio 4, an Eclipse-based architecture development environment. Important principles are i) the deep separation of generated (architecture-prescribed) and non-generated (user-defined) code, ii) an architecture change model, iii) architecture based code regeneration, and iv) architecture change notification. In their work, changes in the architecture affect only architecture-prescribe code. This approach mitigates the update problem via deep separation principle but as well tries to enforce model and code consistency by code generation.

Heidenreich et al. [31] presented the *Java Model Parser and Printer (JaMoPP)*, which treats Java code like any other model, by defining a full metamodel and text syntax specification. They allow generating Java code from model and vice versa. A Java program can be specified as a whole in the model and then generated, consistency is only achieved by construction and as well do not consider incremental update of either. Cicozzi et al. [36] go a step further, based on the *CHESS Modelling Language* [37] and the *Action Language for Foundational UML (ALF)* [38] they produce a fully functional embedded systems, with explicit traceability links from source to model for further monitoring and adjustment to requirements. Opposite to our approach code can not optimized by hand, iterations only occur after validating feedback from the executed system.

DiaSpec [7] uses a specific *Architectural Description Language (ADL)* [39] that integrates a new concept called interaction contract, which is part of the architecture description and describes allowed interactions between components. Its implementation is generated into a for the programmer unmodifiable framework and therefore does not support co-evolution of either model and code. They also rely only on the Java compiler to detect inconsistencies. *ArchJava* [40] unifies a Java program with its architecture. It is a mapping approach, the language itself is extended to provide mappings in code.

A similar approach in terms of architecture description as part of the implementation is *Archface* [9]. New interfaces mechanisms are used as ADL in the design phase, in implementation phases programming interfaces. To specify the collaboration among components, *Aspect-Oriented Programming (AOP)* [41] concepts such as `pointcut` and `advice` are utilized. Model and code in both ArchJava and Archface are not two separate entities, both have to evolve as soon as one changes.

Murphy et al. [42] use a batch like approach, which tries to exploit the drift between architecture and implementation instead of preventing it. A high-level structural model that is "good-enough" for reasoning is produced. An engineer first defines a model of interest. Then, a model of the source code (depicting certain actions: call graph, event interactions) is extracted. Finally, mappings have to be defined between the models. Given the high-level structural model, then the source model and the mappings, a `software reflexion model` is computed to determine inconsistencies. This approach is closest to MCCC, however consistency checking is not done incrementally. Diskin et al. [43] uses consistency checking among heterogeneous models but does not consider implementation and also considers a different approach [44] by using formal semantics. Inconsistencies in models can not only be detected, they also can be resolved in a semi-automatic way. Reder et al. [11] try to detect inconsistencies only in models but also compute based on detected inconsistencies repair actions. To the user a filtered list of actions that transform the model is presented, from which he can choose the "correct" one to resolve a inconsistency.

9 Conclusion and Future Work

In this paper, we presented MCCC, a novel approach for model-and-code consistency checking that addresses issues regarding inconsistency between design models and source code. In particular, our approach provides the following advantages: i) fast incremental consistency checking for arbitrary object-oriented-modelling and programming languages, ii) adaptable consistency rules that support the evaluation of domain specific semantics, and iii) scalability that allows instant user feedback even in big development projects with frequent artifact evolution. The validation results show that the approach fulfills the initial goals defined in Section 1 and that it is suitable for being applied live during modelling to actively guide developers without interrupting and even enhancing their workflow. However, this research allows for further development. For future work, we plan to integrate approaches for the fixing of inconsistencies in our prototype implementation. A further evolution of MCCC could also be in integration of arbitrary numbers of development artifacts-metamodels. Moreover, we want to increase support for distributed development by providing a consistency checking environment that integrates data from different development tools.

10 Appendix

10.1 Class Diagram

10.1.1 AssociationConform

```
[Context: UML::Association]
if not(self.ownedEnd->isEmpty()) then
  let fields : Set(members::Field) =
  if self.ownedEnd->at(0).type.oclIsTypeOf(uml::Class) then
    if self.ownedEnd->at(0).type.oclAsType(uml::Class).javatrace
      <> null then
      self.ownedEnd->at(0).type.oclAsType(uml::Class).javatrace
      .members->select(oclIsTypeOf(members::Field))
    else Set{} endif
  else
    -- Enumeration || Interface
  endif in
  fields->exists(field |
    field.typeReference.typeConform(self.memberEnd->at(0).type) and
    self.memberEnd->at(0).multiplicityConform(field))
else
  self.memberEnd->forAll(property |
  let fieldsA : Set(members::Field) =
  if property.class.oclIsTypeOf(uml::Class) then
    if property.class.javatrace <> null then
      property.class.javatrace.getFields()
    else Set{} endif
  else
    -- Enumeration || Interface
  endif in
  if not(fieldsA->isEmpty()) then
    fieldsA->exists(field |
      field.typeReference.typeConform(property.type) and
      property.multiplicityConform(field))
  else false endif
  )
endif
```

Listing 9: AssociationConform

10.1.2 AssociationFieldConform

```
[Context: UML::Association]
if self.fieldtrace <> null and not(self.ownedEnd->isEmpty()) then
  self.fieldtrace.typeReference.typeConform(self.memberEnd->at(0).type) and
  self.memberEnd->at(0).multiplicityConform(self.fieldtrace) and
    let classifier : classifiers::ConcreteClassifier =
     self.fieldtrace.getContainingConcreteClassifier() in
      if classifier.oclIsTypeOf(classifiers::Class) then
        classifier.oclAsType(classifiers::Class).umltrace =
         self.ownedEnd->at(0).type
      else
```

```
        -- Enumeration || Interface
      endif
else
  if self.fieldmtrace <> null and self.ownedEnd->isEmpty() then
    ((self.fieldmtrace->size() = 2) and
    self.fieldmtrace->forAll(fieldtrace |
      self.memberEnd->exists(property |
        fieldtrace.typeReference.typeConform(property.type))))
  else
    false
  endif
 endif
```

Listing 10: AssociationFieldConform

10.1.3 ClassFieldConform

```
[Context: UML::Property]
let fields : Set(members::Field) =
  if self.class.oclIsTypeOf(uml::Class) then
    if self.class.javatrace <> null then
      self.class.javatrace.members->select(oclIsTypeOf(members::Field))
    else Set{} endif
  else
    -- Enumeration || Interface
  endif in
  if not(fields->isEmpty()) then
  let nameConformFields : Set(members::Field) = fields->select(jField |
    self.name = jField.name ) in
      nameConformFields->exists(javaField |
        javaField.typeReference.typeConform(self.type) and
        javaField.getModifiers()->exists(modifier |
          modifier.visibilityConform(self.visibility)) and
        self.multiplicityConform(javaField) and
        self.isStatic = javaField.isStatic()
      )
  else false endif
```

Listing 11: ClassFieldConform

10.1.4 ClassMethodConform

```
[Context: UML::Operation]
let methods : Set(members::Method) =
  if self.class.oclIsTypeOf(uml::Class) then
    if self.class.javatrace <> null then
      self.class.javatrace.members->select(oclIsTypeOf(members::Method))
    else Set{} endif
  else
    -- Enumeration || Interface
  endif in
  if not(methods->isEmpty()) then
  let nameConformMethods : Set(members::Method) = methods->select(jMethod |
  self.name = jMethod.name) in
```

```
      nameConformMethods->exists(javamethod |
        javamethod.typeReference.typeConform(self.type) and
        self.ownedParameter->forAll(uparam |
        (uparam.direction <> uml::ParameterDirectionKind::return) implies
              (javamethod.parameters->exists(jparam |
               jparam.typeReference.typeConform(uparam.type) and
               uparam.oclAsType(uml::Property).multiplicityConform(jparam)))) and
        self.isAbstract = javamethod.getModifiers()->exists(modifier |
         modifier.oclIsTypeOf(java::Abstract)) and
        self.isStatic = javamethod.getModifiers()->exists(modifier |
         modifier.oclIsTypeOf(java::Static)) and
        (
          (
            (self.type = null) implies
              (self.ownedParameter->size() = javamethod.parameters->size())
          ) or
          (
            (self.type <> null) implies
            ((self.ownedParameter->size() - 1) = javamethod.parameters->size())
          )
        )
      )
    else false endif
```

Listing 12: ClassMethodConform

10.1.5 EnumConstantConform

```
[Context: UML::EnumerationLiteral]
let javaEnum : classifiers::Enumeration = self.enumeration.javaenumeration in
let nameConformEnums : Set(members::EnumConstant) =
javaEnum.constants->select(enumConstant | enumConstant.name = self.name) in
  nameConformEnums->exists(enumConstant |
    (
      (self.specification = null  and enumConstant.arguments = null) or
      (self.specification.oclIsTypeOf(uml::LiteralNull) and
       enumConstant.arguments = null) or
      if self.specification <> null and
       not(self.specification.oclIsTypeOf(uml::LiteralNull)) then
        (if self.specification.oclIsTypeOf(uml::LiteralInteger) and
        enumConstant.getFirstChildByType((literals::DecimalIntegerLiteral)
        .oclClass()) <> null
         then
         enumConstant.getFirstChildByType((literals::DecimalIntegerLiteral)
         .oclClass()).oclAsType(literals::DecimalIntegerLiteral).decimalValue.round()
         = self.specification.integerValue()
          else false endif )
        or
        -- String Value
        or
        -- DecimalInteger
        or
        -- Boolean
        or
```

```
        ( if self.specification.oclIsTypeOf(uml::InstanceValue) then
         enumConstant.getArgumentTypes()->exists(javaType |
          if javaType.oclIsTypeOf(classifiers::Class) then
            javaType.oclAsType(classifiers::Class).umltrace =
            self.specification.oclAsType(uml::InstanceValue).type
          else
            -- Enumeration || Interface
          endif
            )
        else false endif
        s)
      else
        false
      endif

    )
  )
```

Listing 13: EnumConstantConform

10.1.6 GeneralizationConform

```
[Context: UML::Class]
let javaClass : classifiers::Class = self.javatrace in
  if javaClass <> null then
    if self.superClass->isEmpty() and javaClass.extends = null then
      true
    else
      if javaClass.extends = null and not(self.superClass->isEmpty()) then
        false
      else
       self.superClass->exists(superClass |
         superClass.javatrace = javaClass.extends.getTarget())
      endif
    endif
  else false endif
```

Listing 14: GeneralizationConform

10.1.7 InterfaceConform

```
[Context: UML::Interface]
let javaClass : classifiers::Class = self.javatrace in
  if javaClass <> null then
    self.getImplementedInterfaces()->forAll(uinterface |
      javaClass.implements->exists(jinterface |
        uinterface.javainterface = jinterface))
  else false endif
```

Listing 15: InterfaceConform

10.1.8 UsageConform

```
[Context: UML::Usage]
self.target->forAll(targetElement |
  targetElement.oclIsKindOf(uml::Classifier) implies
  let target : uml::Classifier = targetElement.oclAsType(uml::Classifier) in
  self.source->forAll(sourceElement |
    sourceElement.oclIsKindOf(uml::Classifier) implies
      let fields : Set(members::Field) =
        if sourceElement.oclIsTypeOf(uml::Class) then
          if sourceElement.oclAsType(uml::Class).javatrace <> null then
            sourceElement.oclAsType(uml::Class).javatrace
            .members->select(oclIsTypeOf(members::Field))
          else Set{} endif
        else
          -- Enumeration || Interface
        endif in
        fields->exists(javaField | javaField.typeReference.typeConform(target))
  )
) or
self.target->forAll(targetElement |
  targetElement.oclIsKindOf(uml::Classifier) implies
  let target : uml::Classifier = targetElement.oclAsType(uml::Classifier) in
  self.source->forAll(sourceElement |
    sourceElement.oclIsKindOf(uml::Classifier) implies
      let methods : Set(members::Method) =
        if sourceElement.oclIsTypeOf(uml::Class) then
          if sourceElement.oclAsType(uml::Class).javatrace <> null then
            sourceElement.oclAsType(uml::Class).javatrace
            .members->select(oclIsTypeOf(members::Method))
          else Set{} endif
        else
          -- Enumeration || Interface
        endif in
        methods->exists(method |
          method.typeReference.typeConform(target) or
          method.parameters->exists(parameter |
            parameter.typeReference.typeConform(target)) or
          if method.oclIsTypeOf(members::ClassMethod) then
            target.oclAsType(uml::Type)
            .containsType(method.oclAsType(members::ClassMethod)
            .statements->asSequence())
          else false endif
         )
  )
)
```

Listing 16: UsageConform

10.1.9 Queries

```
[Context: Java::Modifier]
[Arguments: UML::VisibilityKind]
(self.oclIsTypeOf(modifiers::Public)
  and umlvisibility=uml::VisibilityKind::public) or
(self.oclIsTypeOf(modifiers::Private)
```

```
  and umlvisibility=uml::VisibilityKind::private) or
(self.oclIsTypeOf(modifiers::Protected)
  and umlvisibility=uml::VisibilityKind::protected)
```

Listing 17: visibilityConform

```
[Context: UML::Property]
[Arguments: Java::Variable]
let field : java::Variable = javaField in
(
  (self.upperValue.oclAsType(uml::LiteralUnlimitedNatural).value = -1 or
  self.upperValue.oclAsType(uml::LiteralUnlimitedNatural).value > 1) and
    (
      field.getArrayDimension().round() > 0 or
      if field.typeReference.oclIsTypeOf(types::NamespaceClassifierReference) then
        field.typeReference.oclAsType(types::NamespaceClassifierReference)
        .classifierReferences->exists(classifierRef |
          if classifierRef.target <> null
          and classifierRef.target.oclIsTypeOf(classifiers::Class) then
              classifierRef.target.oclAsType(classifiers::Class)
              .getAllSuperClassifiers()->exists(reference |
                  reference.getContainingCompilationUnit().namespaces->at(0)
                    = 'java' and
                  reference.getContainingCompilationUnit().namespaces->at(1)
                     = 'util' and
                  reference.name = 'Collection')
          else false endif)
      else false endif
    )
) or
(
   (self.upperValue.oclAsType(uml::LiteralUnlimitedNatural).value = 1 or
   self.upperValue.oclAsType(uml::LiteralUnlimitedNatural).value = 0) and
    (
      field.getArrayDimension().round() = 0
      and
      if field.typeReference.oclIsTypeOf(types::NamespaceClassifierReference) then
        field.typeReference.oclAsType(types::NamespaceClassifierReference)
        .classifierReferences->exists(classifierRef |
          if classifierRef.target <> null
          and classifierRef.target.oclIsTypeOf(classifiers::Class) then
              not(classifierRef.target.oclAsType(classifiers::Class)
              .getAllSuperClassifiers()->exists(reference |
                  reference.getContainingCompilationUnit().namespaces->at(0)
                     = 'java' and
                  reference.getContainingCompilationUnit().namespaces->at(1)
                     = 'util' and
                  reference.name = 'Collection'))
          else false endif)
      else false endif
    )
)
```

Listing 18: multiplicityConform

```
[Context: Java::TypeReference]
[Arguments: UML::Type]
let type : uml::Type = umltype in
(self.oclIsTypeOf(types::Void) and umltype = null) or
(
  if type <> null then
      (
        self.oclIsTypeOf(types::Boolean) and
        type.oclIsTypeOf(uml::PrimitiveType) and type.name ='Boolean'
      ) or
      (
        (self.oclIsTypeOf(types::Int) or
        self.oclIsTypeOf(types::Short) or
        self.oclIsTypeOf(types::Long)) and
        type.oclIsTypeOf(uml::PrimitiveType)
         and type.name ='Integer'
      ) or
      (
        (self.oclIsTypeOf(types::Double) or
        self.oclIsTypeOf(types::Float)) and
        ((type.oclIsTypeOf(uml::PrimitiveType) and
        type.name ='Real') or  (type.oclIsTypeOf(uml::PrimitiveType)
        and type.name ='EFloat'))
      ) or
        (
          if self.oclIsTypeOf(types::NamespaceClassifierReference) then
            self.oclAsType(types::NamespaceClassifierReference)
            .classifierReferences->exists(classifierRef |
              if classifierRef.getTarget()
              .oclIsTypeOf(classifiers::Class) then
                classifierRef.getTarget()
                .oclAsType(classifiers::Class).umltrace = type
              else
                -- Enumeration || Interface
              endif
              or (
                classifierRef.typeArguments
                ->exists(typeArgument |
                  if typeArgument.oclIsTypeOf(generics::QualifiedTypeArgument) then
                    let typeRef : types::TypeReference = typeArgument
                    .oclAsType(generics::QualifiedTypeArgument).typeReference in
                    typeRef.typeConform(type)
                  else false endif)
                )
              )
            else false endif
        ) or (
          if type.oclIsTypeOf(uml::PrimitiveType)
          and type.name ='String'
          and self.oclIsTypeOf(types::NamespaceClassifierReference) then
            self.oclAsType(types::NamespaceClassifierReference).classifierReferences
            ->exists(reference |
                reference.target.getContainingCompilationUnit()
                .namespaces->at(0) = 'java' and
                reference.target.getContainingCompilationUnit()
```

```
                .namespaces->at(1) = 'lang' and
                reference.target.name = 'String'
              )
          else false endif
        )
    else false endif
)
```

Listing 19: typeConform

```
[Context: UML::Type]
[Arguments: Sequence{Java::Statement}]
let statements : Sequence(statements::Statement) = incStatements in
statements->exists(statement |
  if statement.oclIsTypeOf(statements::LocalVariableStatement) then
    statement.oclAsType(statements::LocalVariableStatement)
    .variable.typeReference.typeConform(self)
  else
    if statement.oclIsTypeOf(statements::StatementListContainer) then
      self.containsType(self.oclAsType(statements::StatementListContainer)
      .statements->asSequence())
    else
      if statement.oclIsTypeOf(statements::StatementContainer) then
        self.containsType(Sequence{self.oclAsType(statements::StatementContainer)
        .statement}->asSequence())
      else false endif
    endif
  endif
)
```

Listing 20: containsType

10.2 Sequence Diagram

10.2.1 InteractConform

```
[Context: UML::Interaction]
self.validateSubInteractionFragments(self.fragment->asSequence())
```

Listing 21: InteractConform

```
[Context: UML::Interaction]
[Arguments: Sequence{UML::InteractionFragment}]
let incomingFragments : Sequence(uml::InteractionFragment) = iFragments in
let fragments : Sequence(uml::InteractionFragment) =
incomingFragments->select(oclIsTypeOf(uml::BehaviorExecutionSpecification) or
oclIsTypeOf(uml::CombinedFragment))->collect(oclAsType(uml::InteractionFragment)) in
let execSpecifications : Sequence(uml::BehaviorExecutionSpecification) =
incomingFragments->select(oclIsTypeOf(uml::BehaviorExecutionSpecification))
  ->collect(oclAsType(uml::BehaviorExecutionSpecification)) in
execSpecifications->forAll(msg |
  let type : uml::Type = msg.start.oclAsType(uml::MessageOccurrenceSpecification)
    .message.receiveEvent.oclAsType(uml::BehaviorExecutionSpecification)
    .covered->asSequence()->first().represents.type in
```

```
  let subFragments : Sequence(uml::InteractionFragment) =
  fragments->select(fragment |
    incomingFragments->asSequence()->indexOf(msg) <
    incomingFragments->asSequence()->indexOf(fragment) and
    incomingFragments->asSequence()->indexOf(fragment) <
    incomingFragments->asSequence()->indexOf(msg.finish)
    and
    (if fragment.oclIsTypeOf(uml::CombinedFragment) then
      fragment.oclAsType(uml::CombinedFragment).covered->exists(lifeline |
      lifeline.represents.type = type)
    else
      if fragment.oclIsTypeOf(uml::BehaviorExecutionSpecification) then
        fragment.oclAsType(uml::BehaviorExecutionSpecification)
        .start.oclAsType(uml::MessageOccurrenceSpecification)
        .message.sendEvent.oclAsType(uml::MessageOccurrenceSpecification)
        .covered->asSequence()->first().represents.type = type
      else false endif
    endif)
    ) in
      let sFragments : Sequence(uml::InteractionFragment) = subFragments
      ->select(fragment | not(subFragments->exists(enclosing |
        enclosing.oclIsTypeOf(uml::BehaviorExecutionSpecification) and
    incomingFragments->indexOf(enclosing) <
    incomingFragments->indexOf(fragment) and
    incomingFragments->indexOf(fragment) <
    incomingFragments->indexOf(enclosing
    .oclAsType(uml::BehaviorExecutionSpecification).finish)))) in
    if type.oclIsTypeOf(uml::Interface) then true else
      let methods : Set(members::Method) =
        if type.oclIsTypeOf(uml::Class) then
          if type.oclAsType(uml::Class).javatrace <> null then
            type.oclAsType(uml::Class).javatrace.getMethods()
          else Set{} endif
        else
          -- Enumeration
        endif in
      let corrMethod : Sequence(members::ClassMethod) =
      methods->select(method | method.oclIsTypeOf(members::ClassMethod)
        and method.name =
         msg.start.oclAsType(uml::MessageOccurrenceSpecification).message.name)
         ->collect(oclAsType(members::ClassMethod))->asSequence() in
        if not(sFragments->isEmpty()) and not(corrMethod->isEmpty()) then
          corrMethod->exists(method |
            msg.containsFragmentsInOrder(sFragments, method.statements->asSequence()))
        else
          sFragments->isEmpty() and not(corrMethod->isEmpty())
        endif
      endif
  ) and
let containingCombinedFragments : Sequence(uml::CombinedFragment) =
fragments->select(oclIsTypeOf(uml::CombinedFragment))
  ->collect(oclAsType(uml::CombinedFragment)) in
containingCombinedFragments->forAll(cFragment | cFragment.operand->forAll(op |
```

```
    self.validateSubInteractionFragments(op.fragment->asSequence()))))
```

Listing 22: validateSubInteractionFragments

10.2.2 LineConform

```
[Context: UML::Lifeline]
self.validateInteractionFragments(self.interaction.fragment->asSequence())
```

Listing 23: LineConform

```
[Context: UML::Lifeline]
[Arguments: Sequence{UML::InteractionFragment}]
let type : uml::Type = self.represents.type in
let incomingFragments : Sequence(uml::InteractionFragment) = iFragments in
let fragments : Sequence(uml::InteractionFragment) =
iFragments->select(fragment |
  if fragment.oclIsTypeOf(uml::CombinedFragment) then
    fragment.oclAsType(uml::CombinedFragment).covered->exists(lifeline |
      lifeline.represents.type = type)
  else false endif or
  if fragment.oclIsTypeOf(uml::BehaviorExecutionSpecification) then
    fragment.oclAsType(uml::BehaviorExecutionSpecification)
    .start.oclAsType(uml::MessageOccurrenceSpecification)
    .message.receiveEvent.oclAsType(uml::MessageOccurrenceSpecification)
    .covered->asSequence()->first().represents.type = type or
    fragment.oclAsType(uml::BehaviorExecutionSpecification)
    .start.oclAsType(uml::MessageOccurrenceSpecification)
    .message.sendEvent.oclAsType(uml::MessageOccurrenceSpecification)
    .covered->asSequence()->first().represents.type = type
  else false endif) in
let incomingMsgs : Sequence(uml::BehaviorExecutionSpecification) =
fragments->select(oclIsTypeOf(uml::BehaviorExecutionSpecification))
->collect(oclAsType(uml::BehaviorExecutionSpecification))
    ->select(inc |
      inc.start.oclAsType(uml::MessageOccurrenceSpecification)
      .message.receiveEvent.oclAsType(uml::MessageOccurrenceSpecification)
      .covered->asSequence()->first().represents.type = type) in
incomingMsgs->forAll(incMsg |
  let subFragments : Sequence(uml::InteractionFragment) =
  fragments->select(fragment |
    incomingFragments->asSequence()->indexOf(incMsg) <
    incomingFragments->asSequence()->indexOf(fragment) and
    incomingFragments->asSequence()->indexOf(fragment) <
    incomingFragments->asSequence()->indexOf(incMsg.finish) and
      (if fragment.oclIsTypeOf(uml::CombinedFragment) then
      fragment.oclAsType(uml::CombinedFragment).covered->exists(lifeline |
        lifeline.represents.type = type)
    else
      if fragment.oclIsTypeOf(uml::BehaviorExecutionSpecification) then
        fragment.oclAsType(uml::BehaviorExecutionSpecification)
        .start.oclAsType(uml::MessageOccurrenceSpecification)
        .message.sendEvent.oclAsType(uml::MessageOccurrenceSpecification)
        .covered->asSequence()->first().represents.type = type
```

```
      else false endif
    endif)) in
    let sFragments : Sequence(uml::InteractionFragment) =
    subFragments->select(fragment | not(subFragments->exists(enclosing |
      enclosing.oclIsTypeOf(uml::BehaviorExecutionSpecification) and
        incomingFragments->indexOf(enclosing) <
        incomingFragments->indexOf(fragment) and
        incomingFragments->indexOf(fragment) <
        incomingFragments->indexOf(enclosing
          .oclAsType(uml::BehaviorExecutionSpecification).finish)))) in
      if type.oclIsTypeOf(uml::Interface) then true else
        let methods : Set(members::Method) =
          if type.oclIsTypeOf(uml::Class) then
            if type.oclAsType(uml::Class).javatrace <> null then
              type.oclAsType(uml::Class).javatrace.getMethods()
            else Set{} endif
          else
            -- Enumeration
          endif in
        let corrMethod : Sequence(members::ClassMethod) =
          methods->select(method |
            method.oclIsTypeOf(members::ClassMethod) and
            method.name =
            incMsg.start.oclAsType(uml::MessageOccurrenceSpecification).message.name)
          ->collect(oclAsType(members::ClassMethod))->asSequence() in
        if not(sFragments->isEmpty()) and not(corrMethod->isEmpty()) then
          corrMethod->exists(method |
            incMsg.containsFragmentsInOrder(sFragments, method
            .statements->asSequence()))
        else
          sFragments->isEmpty() and not(corrMethod->isEmpty())
        endif
      endif
    ) and
let containingCombinedFragments : Sequence(uml::CombinedFragment) =
fragments->select(oclIsTypeOf(uml::CombinedFragment))
->collect(oclAsType(uml::CombinedFragment)) in
containingCombinedFragments->forAll(cFragment |
  cFragment.operand->forAll(op |
    self.validateInteractionFragments(op.fragment->asSequence())))
```

Listing 24: validateInteractionFragments

```
[Context: UML::BehaivorExecutionSpecification]
[Arguments: Sequence{UML::InteractionFragment}, Sequence{Java::Statement}]
let fragments : Sequence(uml::InteractionFragment) = iFragments in
let statements  : Sequence(statements::Statement) = incStatements in
let type : uml::Type = self.start.oclAsType(uml::MessageOccurrenceSpecification)
  .message.receiveEvent.oclAsType(uml::MessageOccurrenceSpecification)
  .covered->asSequence()->first().represents.type in
if fragments->isEmpty() then true else
if statements->isEmpty() then false else
if fragments->first().oclIsTypeOf(uml::BehaviorExecutionSpecification) then
  let toSearch : commons::Commentable =
    if statements->first().oclIsTypeOf(statements::Condition) then
```

```
      statements->first().oclAsType(statements::Condition).condition
    else
      -- other initializers or the first statement
    endif in
  if fragments->first().oclAsType(uml::BehaviorExecutionSpecification)
  .containsCall(toSearch) then
    self.containsFragmentsInOrder(fragments
    ->excluding(fragments->first()), statements)
  else
    self.containsFragmentsInOrder(fragments, statements
    ->excluding(statements->first()))
  endif
else
  if fragments->first().oclIsTypeOf(uml::CombinedFragment) then
    let cFragment : uml::CombinedFragment =
      fragments->first().oclAsType(uml::CombinedFragment) in
    let operand : uml::InteractionOperatorKind =
      cFragment.interactionOperator in
    if cFragment.existsCombinedFragment(statements->first()) then
      if operand = uml::InteractionOperatorKind::loop
      or operand = uml::InteractionOperatorKind::opt then
        let subStatements : Sequence(statements::Statement) =
          if statements->first().oclAsType(statements::StatementContainer)
          .statement.oclIsTypeOf(statements::Block) then
            statements->first().oclAsType(statements::StatementContainer)
            .statement.oclAsType(statements::Block).statements->asSequence()
          else
            Sequence{statements->first()
            .oclAsType(statements::StatementContainer).statement}
          endif in
        self.containsFragmentsInOrder(fragments->first()
          .oclAsType(uml::CombinedFragment).operand->asSequence()->first()
          .fragment->asSequence()->select(subFragment |
            if subFragment.oclIsTypeOf(uml::CombinedFragment) then
              subFragment.oclAsType(uml::CombinedFragment)
              .covered->exists(lifeline |
                lifeline.represents.type = type)
            else false endif or (
            if subFragment.oclIsTypeOf(uml::BehaviorExecutionSpecification) then
              subFragment.oclAsType(uml::BehaviorExecutionSpecification)
              .start.oclAsType(uml::MessageOccurrenceSpecification)
              .message.sendEvent.oclAsType(uml::MessageOccurrenceSpecification)
              .covered->asSequence()->first().represents.type = type
            else false endif))
          , subStatements)
      else
        if statements->first().oclAsType(statements::Condition)
        .elseStatement <> null then
          let firstOption : Sequence(statements::Statement) =
            if statements->first().oclAsType(statements::Condition)
            .statement.oclIsTypeOf(statements::Block) then
              statements->first().oclAsType(statements::Condition)
              .statement.oclAsType(statements::Block).statements->asSequence()
            else
              Sequence{statements->first().oclAsType(statements::Condition).statement}
```

```
          endif in
        let secondOption : Sequence(statements::Statement) =
          if statements->first().oclAsType(statements::Condition)
          .elseStatement.oclIsTypeOf(statements::Block) then
            statements->first().oclAsType(statements::Condition)
            .elseStatement.oclAsType(statements::Block)
            .statements->asSequence()
          else
            Sequence{statements->first().oclAsType(statements::Condition)
            .elseStatement}
          endif in
        self.containsFragmentsInOrder(
          fragments->first().oclAsType(uml::CombinedFragment).operand->asSequence()
          ->first().fragment->asSequence()->select(subFragment |
            if subFragment.oclIsTypeOf(uml::CombinedFragment) then
              subFragment.oclAsType(uml::CombinedFragment).covered->exists(lifeline |
                lifeline.represents.type = type)
            else false endif or (
              if subFragment.oclIsTypeOf(uml::BehaviorExecutionSpecification) then
                subFragment.oclAsType(uml::BehaviorExecutionSpecification)
                .start.oclAsType(uml::MessageOccurrenceSpecification).message
                .sendEvent.oclAsType(uml::MessageOccurrenceSpecification).covered
                ->asSequence()->first().represents.type = type
              else false endif))
         , firstOption) and
         self.containsFragmentsInOrder(
          fragments->first().oclAsType(uml::CombinedFragment).operand->asSequence()
          ->at(1).fragment->asSequence()->select(subFragment |
            if subFragment.oclIsTypeOf(uml::CombinedFragment) then
              subFragment.oclAsType(uml::CombinedFragment).covered->exists(lifeline |
              lifeline.represents.type = type)
            else false endif or (
              if subFragment.oclIsTypeOf(uml::BehaviorExecutionSpecification) then
                subFragment.oclAsType(uml::BehaviorExecutionSpecification)
                .start.oclAsType(uml::MessageOccurrenceSpecification).message
                .sendEvent.oclAsType(uml::MessageOccurrenceSpecification)
                .covered->asSequence()->first().represents.type = type
              else false endif))
          , secondOption)
        else
          self.containsFragmentsInOrder(fragments,
            statements->excluding(statements->first()))
        endif
    endif
    and
    self.containsFragmentsInOrder(fragments->excluding(fragments->first()),
      statements->excluding(statements->first()))
  else
    self.containsFragmentsInOrder(fragments,
    statements->excluding(statements->first()))
  endif
else
  false -- should not happen in any case
endif
```

```
endif endif endif
```

Listing 25: containsFragmentsInOrder

```
[Context: UML::BehaviorExecutionSpecification]
[Arguments: Java::Commentable]
let inc : commons::Commentable = incoming in
if inc <> null then
if  not(inc.getChildrenByType((references::MethodCall).oclClass())->isEmpty())
or inc.oclIsTypeOf(references::MethodCall) then
  let receiveType : uml::Type =
    self.start.oclAsType(uml::MessageOccurrenceSpecification).message
    .receiveEvent.oclAsType(uml::MessageOccurrenceSpecification)
     .covered->asSequence()->first().represents.type in
  let calls : Sequence(references::MethodCall) =
    if inc.oclIsTypeOf(references::MethodCall) then
      Sequence{inc.oclAsType(references::MethodCall)}
    else
      inc.getChildrenByType((references::MethodCall).oclClass())
      ->asSequence()->collect(oclAsType(references::MethodCall))
    endif in
  calls->exists(call |
    call.target.name =
    self.start.oclAsType(uml::MessageOccurrenceSpecification).message.name and
    let jType : types::Type =
      if call.target <> null then
        if call.target.oclAsType(members::Method)
        .getContainingConcreteClassifier() <> null then
          call.target.oclAsType(members::Method).getContainingConcreteClassifier()
        else
          null
        endif
      else null endif in

    if jType.oclIsTypeOf(classifiers::Class) then
      jType.oclAsType(classifiers::Class).umltrace = receiveType
    else
      -- Enumeration || Interface
    endif)
else false endif else false endif
```

Listing 26: containsCall

```
[Context: UML::CombinedFragment]
[Arguments: Java::Statement]
let operand : uml::InteractionOperatorKind = self.interactionOperator in
let statement : statements::Statement = iStatement in
(operand = uml::InteractionOperatorKind::loop and
  (statement.oclIsTypeOf(statements::ForLoop)
  or statement.oclIsTypeOf(statements::WhileLoop)
  or statement.oclIsTypeOf(statements::ForEachLoop)
  or statement.oclIsTypeOf(statements::DoWhileLoop)))
  or ((operand = uml::InteractionOperatorKind::alt
  or operand = uml::InteractionOperatorKind::opt)
```

```
and (statement.oclIsTypeOf(statements::Conditional)
  or statement.oclIsTypeOf(statements::Condition)))
```

Listing 27: existsCombinedFragment

10.3 State Machine Diagram

10.3.1 CallSequenceStateChartConform

```
[Context: UML::StateMachine]
if self.javamethod <> null then
  let init : uml::Pseudostate = self.region->asSequence()->first()
    .subvertex->select(oclIsTypeOf(uml::Pseudostate))
    ->collect(oclAsType(uml::Pseudostate))->asSequence()->first() in
  let startstate : uml::State = init.outgoing->asSequence()
    ->first().target.oclAsType(uml::State) in
  let statements : Sequence(statements::Statement) =
    self.javamethod.statements->asSequence() in
  let context : uml::Class = self.context.oclAsType(uml::Class) in
  let endstate : uml::State =
    startstate.calculateStateToStatement(startstate, context, null,
    statements).oclAsType(uml::State) in
  if endstate <> null and (endstate = startstate
  or endstate.oclIsTypeOf(uml::FinalState)) then
    startstate.calculateScore(statements, self.javamethod, context)
   else 0.0 endif
else 0.0 endif
```

Listing 28: CallSequenceStatechartConform

```
[Context: UML::State]
[Arguments: Sequence{Java::Statement}, Java::Method, UML::Class]
let statements : Sequence(statements::Statement) = incStatements in
let method : members::ClassMethod = cMethod in
let context : uml::Class = incContext in
if statements = null then 0.0 else
if statements->isEmpty() then 0.0 else
if statements->first().oclIsKindOf(statements::StatementContainer)
or statements->first().oclIsKindOf(statements::StatementListContainer) then
  let subStatements : Sequence(statements::Statement) =
    if statements->first().oclIsKindOf(statements::StatementContainer) then
      if statements->first().oclAsType(statements::StatementContainer
      ).statement.oclIsTypeOf(statements::Block) then
        statements->first().oclAsType(statements::StatementContainer)
        .statement.oclAsType(statements::Block).statements->asSequence()
      else
        Sequence{statements->first().oclAsType(statements::StatementContainer).statement}
      endif
    else
      statements->first().oclAsType(statements::StatementListContainer)
      .statements->asSequence()
    endif in
  let subScore : Real = self.calculateScore(subStatements, method, context) in
  if subScore > 0.0 then
```

```
   let weightedScore : Real =
    if statements->first().oclIsTypeOf(statements::TryBlock) then
      subScore
    else subScore*0.9 endif in
   let nextScore : Real =
   self.calculateScore(statements->excluding(statements->first()), method, context) in
   if nextScore > 0.0 then weightedScore*nextScore else weightedScore endif
  else
    1.0*self.calculateScore(statements->excluding(statements->first()), method, context)
  endif
else
  let currentState : uml::State =
    self.calculateStateToStatement(self, context, statements->first(),
      method.statements->asSequence()).oclAsType(uml::State) in
  let subMethodCall : references::MethodCall =
    statements->first().isMethodCall(context).oclAsType(references::MethodCall) in
  if subMethodCall <> null then
    let subScore : Real =
self.calculateScore(subMethodCall.target.oclAsType(members::ClassMethod)
.statements->asSequence(), method, context) in
    let nextScore : Real =
      self.calculateScore(statements->excluding(statements->first()), method, context) in
    if subScore > 0.0 then
      subScore * nextScore
    else
      1.0*nextScore
    endif
  else
    let followup : uml::State = currentState.outgoing->select(transition |
      transition.isTransitionToNextState(statements->first(), context))
     ->asSequence()->first().target.oclAsType(uml::State) in
    if followup <> null then
      if currentState = followup then -- self loop
        1.0*self.calculateScore(statements->excluding(statements->first()), method, context)
      else
        let followScore : Real =
          self.calculateScore(statements->excluding(statements->first()),
            method, context) in
        if followScore > 0.0 then 1.0 * followScore else 1.0 endif
      endif
    else
      1.0*self.calculateScore(statements->excluding(statements->first()), method, context)
    endif
  endif
endif endif endif
```

Listing 29: calculateScore

```
[Context: UML::State]
[Arguments: UML::State, UML::Class, Java::Statement, Sequence{Java::Statement}]
let currState : uml::State = cState in
let context : uml::Class = incContext in
let stopStatement : statements::Statement = sStatement in
let statements : Sequence(statements::Statement) = incStatements in
if statements->first() = stopStatement or statements->isEmpty() then
```

```
  currState
else
  if statements->first().oclIsKindOf(statements::StatementContainer)
  or statements->first().oclIsKindOf(statements::StatementListContainer) then
    let subStatements : Sequence(statements::Statement) =
      if statements->first().oclIsKindOf(statements::StatementContainer) then
        if statements->first().oclAsType(statements::StatementContainer)
        .statement.oclIsTypeOf(statements::Block) then
          statements->first().oclAsType(statements::StatementContainer)
          .statement.oclAsType(statements::Block).statements->asSequence()
        else
           Sequence{statements->first().oclAsType(statements::StatementContainer).statement}
        endif
      else
        statements->first().oclAsType(statements::StatementListContainer)
        .statements->asSequence()
      endif in
  let followup : uml::State =
    self.calculateStateToStatement(cState, context,
    stopStatement, subStatements).oclAsType(uml::State) in
  if followup <> null then
    self.calculateStateToStatement(followup, context, stopStatement,
    statements->excluding(statements->first()))
  else
    self.calculateStateToStatement(cState, context, stopStatement,
    statements->excluding(statements->first()))
  endif
else
  let subMethodCall : references::MethodCall =
  statements->first().isMethodCall(context).oclAsType(references::MethodCall) in
  let followup : uml::State =
    if subMethodCall <> null then
      self.calculateStateToStatement(cState, context, stopStatement,
      subMethodCall.target.oclAsType(members::ClassMethod)
      .statements->asSequence()).oclAsType(uml::State)
    else
      cState.outgoing->select(transition |
        transition.isTransitionToNextState(statements->first(), context))
      ->asSequence()->first().target.oclAsType(uml::State)
    endif in
  if followup <> null then
    self.calculateStateToStatement(followup, context, stopStatement,
      statements->excluding(statements->first()))
  else
    self.calculateStateToStatement(cState, context, stopStatement,
      statements->excluding(statements->first()))
  endif
endif endif
```

Listing 30: calculateStateToStatement

```
[Context: Java::Statement]
[Arguments: UML::Class]
let context : uml::Class = incContext in
let reference : references::Reference =
```

```
  if self.oclIsKindOf(references::Reference) then
    self.oclIsKindOf(references::Reference)
  else
    self.getFirstChildByType((references::Reference).oclClass())
  endif in
if reference = null then null else
  if reference.getReferencedType() <> context.javatrace
  and reference.next.oclIsTypeOf(references::MethodCall) then
    reference.next
  else null endif
endif
```

Listing 31: isMethodCall

```
[Context: UML::Transition]
[Arguments: Java::Statement, UML::Class]
let statement : statements::Statement = incStatement in
let context : uml::Class = incContext in
let reference : references::IdentifierReference =
  if statement.oclIsTypeOf(references::IdentifierReference) then
    statement.oclAsType(references::IdentifierReference)
  else
    statement.getFirstChildByType((references::IdentifierReference).oclClass())
  endif in
if reference <> null then
  let jType : types::Type =
    if reference.target.oclAsType(types::TypedElement)
    .typeReference.oclAsType(types::NamespaceClassifierReference)
    .classifierReferences->asSequence()->first() <> null then
      reference.target.oclAsType(types::TypedElement)
      .typeReference.oclAsType(types::NamespaceClassifierReference)
      .classifierReferences->asSequence()->first().getTarget()
      else null endif in
   jType = context.javatrace and
   if reference.next <> null
   and reference.next.oclIsTypeOf(references::MethodCall) then
    reference.next.oclAsType(references::MethodCall).target.name =
    self.trigger->asSequence()
    ->first().event.oclAsType(uml::CallEvent).operation.name
  else false endif
else false endif
```

Listing 32: isTransitionToNextState

10.3.2 StatePattern

```
[Context: UML::State]
let methods : Sequence(members::ClassMethod) =
  if self.javastate <> null then
    self.javastate.getMethods()->collect(oclAsType(members::ClassMethod))
  else Sequence{} endif in
let context : uml::Class =
  self.containingStateMachine().context.oclAsType(uml::Class) in
self.outgoing->forAll(transition |
```

```
  let impMethod : members::ClassMethod =
    methods->select(method |
      method.name = transition.trigger->asSequence()
      ->first().event.oclAsType(uml::CallEvent).operation.name)
    ->first() in
  if impMethod <> null then
  transition.traverseStatements(impMethod.statements->asSequence(), context) and
  self.region->size() <= 1 and
  self.region->forAll(region | region.subvertex->collect(oclAsType(uml::State))
    ->forAll(state |
      state.outgoing->exists(sTransition |
        let smethods : Sequence(members::ClassMethod) =
         state.javastate.getMethods()->collect(oclAsType(members::ClassMethod)) in
        let subMethod : members::ClassMethod = smethods->select(method |
         method.name = sTransition.trigger->asSequence()
         ->first().event.oclAsType(uml::CallEvent).operation.name)->first() in
        transition.searchForSuperCall(subMethod.statements->asSequence())
      )
    )
  )
  else false endif
)
```

Listing 33: StatePattern

```
[Context: UML::Transition]
[Arguments: Sequence{Java::Statement}, UML::Class]
let statements : Sequence(statements::Statement) = incStatements in
let context : uml::Class = incContext in
if statements->isEmpty() then false else
if statements->first().oclIsTypeOf(statements::StatementContainer) then
  let sStatements : Sequence(statements::Statement) =
    if statements->first().oclAsType(statements::StatementContainer)
    .statement.oclIsTypeOf(statements::Block) then
      statements->first().oclAsType(statements::StatementContainer)
      .statement.oclAsType(statements::Block).statements->asSequence()
    else
      Sequence{statements->first()
      .oclAsType(statements::StatementContainer).statement}
    endif in
  self.traverseStatements(sStatements, context)
else
  if statements->first().oclIsTypeOf(statements::StatementListContainer) then
    self.traverseStatements(statements
    ->first().oclAsType(statements::StatementListContainer)
    .statements->asSequence(),context)
  else
    if self.containsTransition(statements->first(),context) then
      true
    else
      self.traverseStatements(statements->excluding(statements->first()),context)
    endif
  endif
```

```
endif endif
```

Listing 34: traverseStatements

```
[Context: UML::Transition]
[Arguments: Java::Statement, UML::Class]
let statement : statements::Statement = incStatement in
let context : uml::Class = incContext in
if statement.oclIsTypeOf(statements::Condition) and self.guard <> null then
  let condition : statements::Condition = statement.oclAsType(statements::Condition) in
    if self.guard.specification.oclAsType(uml::OpaqueExpression).body->first() =
    condition.condition.oclToString() then
      if condition.statement.oclIsTypeOf(statements::Block) then
        condition.statement.oclAsType(statements::Block).statements
        ->exists(subStatement |
          self.isTransitionToNextStateClass(subStatement,context))
      else
        self.isTransitionToNextStateClass(condition.statement,context)
      endif
    else
      false
    endif or
    self.isTransitionToNextStateClass(condition.elseStatement, context)
else
  if self.guard = null
  and (statement.getFirstChildByType((references::IdentifierReference).oclClass()) <> null
  or statement.oclIsTypeOf(references::IdentifierReference)) then
    self.isTransitionToNextStateClass(statement,context)
  else false endif
endif
```

Listing 35: containsTransition

```
[Context: UML::Transition]
[Arguments: Java::Statement, UML::Class]
let statement : statements::Statement = incStatement in
let context : uml::Class = incContext in
let reference : references::IdentifierReference =
if statement.oclIsTypeOf(references::IdentifierReference) then
  statement.oclAsType(references::IdentifierReference)
else
  statement.getFirstChildByType((references::IdentifierReference).oclClass())
endif in
  let jType : types::Type =
    if reference.target.oclAsType(types::TypedElement)
    .typeReference.oclAsType(types::NamespaceClassifierReference)
    .classifierReferences->asSequence()->first() <> null then
      reference.target.oclAsType(types::TypedElement)
      .typeReference.oclAsType(types::NamespaceClassifierReference)
      .classifierReferences->asSequence()->first().getTarget() else null endif in
  jType = context.javatrace and
  if reference.next <> null
  and reference.next.oclIsTypeOf(references::MethodCall) then
    reference.next.oclAsType(references::MethodCall).arguments->exists(argument |
```

```
      if argument.oclIsTypeOf(instantiations::NewConstructorCall) then
        argument.oclAsType(instantiations::NewConstructorCall)
        .typeReference.oclAsType(types::NamespaceClassifierReference)
        .classifierReferences->asSequence()
        ->first().getTarget().oclAsType(classifiers::Class).umlstate =
        self.target
      else
        self.source = self.target and
        argument.oclIsTypeOf(references::SelfReference)
      endif)
  else false endif
```

Listing 36: isTransitionToNextStateClass

```
[Context: UML::Transition]
[Arguments: Sequence{Java::Statement}]
let statements : Sequence(statements::Statement) = incStatements in
if statements->isEmpty() then false else
if statements->first().oclIsTypeOf(statements::StatementContainer) then
  let sStatements : Sequence(statements::Statement) =
    if statements->first().oclAsType(statements::StatementContainer)
    .statement.oclIsTypeOf(statements::Block) then
      statements->first().oclAsType(statements::StatementContainer)
      .statement.oclAsType(statements::Block).statements->asSequence()
    else
      Sequence{statements->first().oclAsType(statements::StatementContainer).statement}
    endif in
  self.searchForSuperCall(sStatements)
else
  if statements->first().oclIsTypeOf(statements::StatementListContainer) then
    self.searchForSuperCall(statements
    ->first().oclAsType(statements::StatementListContainer)
    .statements->asSequence())
  else
    if self.isSuperCall(statements->first()) then
      true
    else
      self.searchForSuperCall(statements->excluding(statements->first()))
    endif
  endif
endif endif
```

Listing 37: searchForSuperCall

```
[Context: UML::Transition]
[Arguments: Java::Statement]
let statement : statements::Statement = incStatement in
let reference : references::SelfReference =
if statement.oclIsTypeOf(references::SelfReference) then
  statement.oclAsType(references::SelfReference)
else
  statement.getFirstChildByType((references::SelfReference).oclClass())
endif in
let jType : classifiers::Class =
```

```
  if reference.self <> null then
    statement.getContainingConcreteClassifier().oclAsType(classifiers::Class).getSuperClass()
  else null endif in
jType.umlstate = self.source and
if reference.next <> null
and reference.next.oclIsTypeOf(references::MethodCall) then
  reference.next.oclAsType(references::MethodCall).target.name =
  self.trigger->asSequence()->first().event.oclAsType(uml::CallEvent).operation.name
else false endif
```

Listing 38: isSuperCall

10.3.3 SyncStateDiagram

```
[Context: UML::StateMachine]
let states : Set(uml::Vertex) = self.region->asSequence()->first().subvertex in
let transitions : Set(uml::Transition) =
  self.region->asSequence()->first().transition
 ->select(transition | not(transition.source.oclIsTypeOf(uml::Pseudostate))) in
let metastate : uml::State =
  self.region->asSequence()->first().transition
 ->select(transition | transition.source.oclIsTypeOf(uml::Pseudostate))
 ->asSequence()->first().target.oclAsType(uml::State) in
let javaClass : classifiers::Class = self.context.oclAsType(uml::Class).javatrace in
if javaClass = null then false else
  transitions->forAll(transition |
    transition.isContainedWithinJavaClass(javaClass, metastate))
endif
```

Listing 39: SyncStateDiagram

```
[Context: UML::Transition]
[Arguments: Java::Class, UML::State]
let javaclass : classifiers::Class = javaClass in
let metastate : uml::State = mstate in
let trigger : uml::Trigger = self.trigger->asSequence()->first() in
let jMethod : members::ClassMethod = javaclass.getMethods()->select(method |
 method.name = trigger.event.oclAsType(uml::CallEvent).operation.name and
 method.oclIsTypeOf(members::ClassMethod))
 ->collect(oclAsType(members::ClassMethod))->asSequence()->first() in
if jMethod <> null then
(
 (
  self.target = metastate and self.guard = null and
  jMethod.statements->exists(statement |
   let methodcall : references::MethodCall =
    if statement.oclIsTypeOf(references::MethodCall) then
     statement.oclAsType(references::MethodCall)
    else
      statement.getFirstChildByType((references::MethodCall).oclClass())
    endif in
   if methodcall <> null then
    methodcall.target.name = 'notifyAll'
   else
```

```
    false
   endif)
 ) or (
  self.target <> metastate and
  not(self.guard.specification.oclAsType(uml::OpaqueExpression).body->isEmpty()) and
  self.methodLeadsToState(jMethod.statements->asSequence())
 )
) else false endif
```

Listing 40: isContainedWithinJavaClass

```
[Context: UML::Transition]
[Arguments: Sequence{Java::Statement}]
let statements : Sequence(statements::Statement) = incStatements in
let targetStatement : statements::Statement =
 self.target.oclAsType(uml::State).javastatement in
statements->exists(statement |
  statement.oclIsTypeOf(statements::Condition) implies
    statement.oclAsType(statements::Condition).condition.oclToString() =
    self.guard.specification.oclAsType(uml::OpaqueExpression).body->first() and
  if statement.oclAsType(statements::Condition)
  .statement.oclIsTypeOf(statements::Block) then
    statement.oclAsType(statements::Condition)
    .statement.oclAsType(statements::Block)
    .statements->exists(statement | statement = targetStatement)
  else
    statement.oclAsType(statements::Condition).statement = targetStatement
  endif
)
```

Listing 41: methodLeadsToState

References

[1] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[2] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE Software*, vol. 20, no. 5, pp. 42–45, 2003.

[3] Y. Zheng and R. N. Taylor, "Enhancing architecture-implementation conformance with change management and support for behavioral mapping," in *ICSE*, pp. 628–638, 2012.

[4] O. M. Group, *Unified Modeling Language UML Version 2.4.1 http://www.omg.org/spec/UML/2.4.1*. OMG, 2010.

[5] B. Hailpern and P. L. Tarr, "Model-driven development: The good, the bad, and the ugly," *IBM Systems Journal*, vol. 45, no. 3, pp. 451–462, 2006.

[6] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd ed., 2009.

[7] D. Cassou, E. Balland, C. Consel, and J. L. Lawall, "Leveraging software architectures to guide and verify the development of sense/compute/control applications," in *ICSE*, pp. 431–440, 2011.

[8] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation," in *ICSE*, pp. 187–197, 2002.

[9] N. Ubayashi, J. Nomura, and T. Tamai, "Archface: a contract place where architectural design and code meet together," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1, pp. 75–84, 2010.

[10] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *IEEE Trans. Software Eng.*, vol. 21, no. 4, pp. 314–335, 1995.

[11] A. Reder and A. Egyed, "Computing repair trees for resolving inconsistencies in design models," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pp. 220–229, 2012.

[12] R. B. France and B. Rumpe, “Model-driven development of complex software: A research roadmap,” in *FOSE*, pp. 37–54, 2007.

[13] A. van Deursen, P. Klint, and J. Visser, “Domain-specific languages: An annotated bibliography,” *SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[14] C. Gane, *Computer-aided software engineering - the methodologies, the products, and the future.* Prentice Hall, 1990.

[15] M. Gardner, “The fantastic combinations of John Conway’s new solitaire game “life”,” *Scientific American*, vol. 223, pp. 120–123, Oct. 1970.

[16] Z. Micskei and H. Waeselynck, “The many meanings of uml 2 sequence diagrams: a survey,” *Software and System Modeling*, vol. 10, no. 4, pp. 489–514, 2011.

[17] A. Egyed, “Automatically detecting and tracking inconsistencies in software design models,” *IEEE Trans. Software Eng.*, vol. 37, no. 2, pp. 188–204, 2011.

[18] I. Groher, A. Reder, and A. Egyed, “Incremental consistency checking of dynamic constraints,” in *FASE*, pp. 203–217, 2010.

[19] “IBM Rational Software Architect https://www.ibm.com/developerworks/rational/products/rsa/ ,2013..”

[20] “ArgoUML http://argouml.tigris.org/ ,2013..”

[21] “Eclipse http://www.eclipse.org/ ,2013..”

[22] “GCC, the GNU Compiler Collection http://gcc.gnu.org/ ,2013..”

[23] A. Reder and A. Egyed, “Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML,” in *ASE* (C. Pecheur, J. Andrews, and E. D. Nitto, eds.), pp. 347–348, ACM, 2010.

[24] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein, “xlinkit: a consistency checking and smart link generation service,” *ACM Trans. Internet Techn.*, vol. 2, no. 2, pp. 151–185, 2002.

[25] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer, “Flexible consistency checking,” *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 1, pp. 28–63, 2003.

[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[27] OMG, *ISO/IEC 19507 Information technology - Object Management Group Object Constraint Language (OCL)*. ISO, 2012.

[28] "Model and Code Consistency Checking: Prototype Implementation. http://www.sea.uni-linz.ac.at/ Tool-Section, 2013."

[29] "MDT/UML2 http://wiki.eclipse.org/mdt-uml2 ,2013.."

[30] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende, "Derivation and refinement of textual syntax for models," in *ECMDA-FA*, pp. 114–129, 2009.

[31] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, "Closing the gap between modelling and java," in *SLE*, pp. 374–383, 2009.

[32] "TIOBE Programming Community Index http://www.tiobe.com/content/paperinfo/tpci/index.html (August 2013)."

[33] "Eclipse Public License, Version 1.0 (EPL-1.0) http://opensource.org/licenses/epl-1.0 ,2013.."

[34] J. Nielsen, *Usability engineering*. Academic Press, 1993.

[35] A. Reder and A. Egyed, "Incremental consistency checking for complex design rules and larger model changes," in *MoDELS*, pp. 202–218, 2012.

[36] F. Ciccozzi, A. Cicchetti, and M. Sjödin, "Towards a round-trip support for model-driven engineering of embedded systems," in *EUROMICRO-SEAA*, pp. 200–208, 2011.

[37] C. Project, *D2.1 CHESS Modelling Language and Editor*. ARTEMIS JU Distribution, 2013.

[38] O. M. Group, *Action Language for Foundational UML (ALF) http://www.omg.org/spec/ALF/1.0.1/Beta3/PDF*. OMG, 2013.

[39] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Trans. Software Eng.*, vol. 26, no. 1, pp. 70–93, 2000.

[40] J. Aldrich, C. Chambers, and D. Notkin, “Architectural reasoning in archjava,” in *ECOOP*, pp. 334–367, 2002.

[41] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-oriented programming,” in *ECOOP*, pp. 220–242, 1997.

[42] G. C. Murphy, D. Notkin, and K. J. Sullivan, “Software reflexion models: Bridging the gap between design and implementation,” *IEEE Trans. Software Eng.*, vol. 27, no. 4, pp. 364–380, 2001.

[43] Z. Diskin, Y. Xiong, and K. Czarnecki, “Specifying overlaps of heterogeneous models for global consistency checking,” in *MoDELS Workshops*, pp. 165–179, 2010.

[44] Z. Diskin, “Towards generic formal semantics for consistency of heterogeneous multimodels,” Jan 2011.

MARKUS RIEDL EHRENLEITNER

+436763808548 ◇ marx_riedl@gmx.at
Lahn 1 ◇ Regau, Austria 4844

EDUCATION

Johannes Kepler University, Linz *October 2011 - Present*
MSc. in Computer Science
Minor in Software Engineering

Johannes Kepler University, Linz *October 2008 - July 2011*
BSc. in Computer Science

INTERNSHIPS

Abatec, Group AG July - August 2012
Software Analyst *Regau, Austria*

- Feasibility analysis for a parking car detection system.

Abatec, Group AG August - September 2011
Software Engineer *Regau, Austria*

- Implement a GUI in WPF for a in house product.

Abatec, Group AG August 2010
Software Engineer *Regau, Austria*

- Produce a testing environment for a cow feeding system based on an embedded linux plattform.

Lenzing Instruments July 2009
Software Engineer *Lenzing, Austria*

- Developer at the currently to be developed framework.

Lenzing Instruments May - September 2008
Software Engineer *Lenzing, Austria*

- Developing the YIS 2000, yarn inspection system.

TECHNICAL STRENGTHS

Computer Languages	Java, C#, Knowledge of C++, C, OCL, Haskell, Prolog
Protocols & APIs	WPF, XML, UML2, EMF
Databases	MySQL, Microsoft SQL, HSQLDB
Tools	GIT, SVN, Vim, Visual Studio, Eclipse, IBM RSA

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, am 17.September 2013 Markus Riedl Ehrenleitner