# Towards Model-and-Code Consistency Checking

Markus Riedl-Ehrenleitner
Johannes Kepler University
Linz, Austria
EMail: Markus.Riedl@jku.at

Andreas Demuth
Johannes Kepler University
Linz, Austria
EMail: Andreas.Demuth@jku.at

Alexander Egyed
Johannes Kepler University
Linz, Austria
EMail: Alexander.Egyed@jku.at

***Abstract*—In model-driven engineering, design models allow for efficient designing without considering implementation details. Still, it is crucial that design models and source code are in sync. Unfortunately, both artifacts do evolve frequently and concurrently which 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 outline a novel approach for checking consistency between design models and source code. Our approach aims at detecting inconsistencies instantly and informing developers about a project's consistency status live during development.**

***Keywords*—Model-and-Code Consistency Checking, Model-Driven Enginering, Incremental Consistency Checking.**

## I. Introduction

*Model-Driven Engineering (MDE)* [1] promotes the use of 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 an important development artifact as it embodies the executable system. Thus, it is of crucial importance that both, design models and source code, are consistent. Otherwise, the executable system deployed to a customer may differ in functionality and quality from the design model – a severe problem if the design model was validated against the customer's needs or documents the system.

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 yet model-to-code transformations typically apply a fixed set of transformation rules. Thus, an automatically produced solution may be correct but not necessarily the one intended by developers (e.g., a translation of an unbounded multiplicity defined in *UML* [4] may always use a `List` although the semantics might call for a `Set` instead).

Therefore, manual adaptation of source code is inevitable. Since for those adaptations usually no further guidance is provided, they might introduce inconsistencies between design models and source code. The issue of inconsistencies becomes even worse when considering that, especially with the increasing popularity of iterative development processes (e.g., *Spiral Model* [5], *Extreme Programming (XP)* [6] or *Scrum* [7]), design models and source code are evolved concurrently [8]. This means that both development artifacts evolve frequently and independently, a situation in which even sophisticated, automatic, consistency-preserving model-to-code transformation technologies (e.g., [3], [9], [10], [11], [12], [13]) do encounter serious issues that are hard to solve [3]. For instance, manual changes in the code may inadvertently 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 address the issue sufficiently – especially with frequently evolving artifacts: they typically require additional, manual adaptations for which only little support is provided and which may still introduce inconsistencies. Existing technologies are thus mostly helpful for generating an initial version of code, if none existed. However, to support the continuous evolution of model and code, consistency detection mechanisms are required.

In this short paper, we outline a novel approach to detect inconsistencies between design models and source code, called *Model-and-Code Consistency Checking (MCCC)*. MCCC is an incremental and highly scalable approach and it aims to provide instant, detailed feedback about a project's consistency status and thus helps developers to not let design models and source code drift apart. In order to detect inconsistencies between artifacts in development projects, consistency rules written in traditional constraint languages (such as OCL [14]) are applied. A developer can change these rules at will so that they are applicable to different development projects. 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.

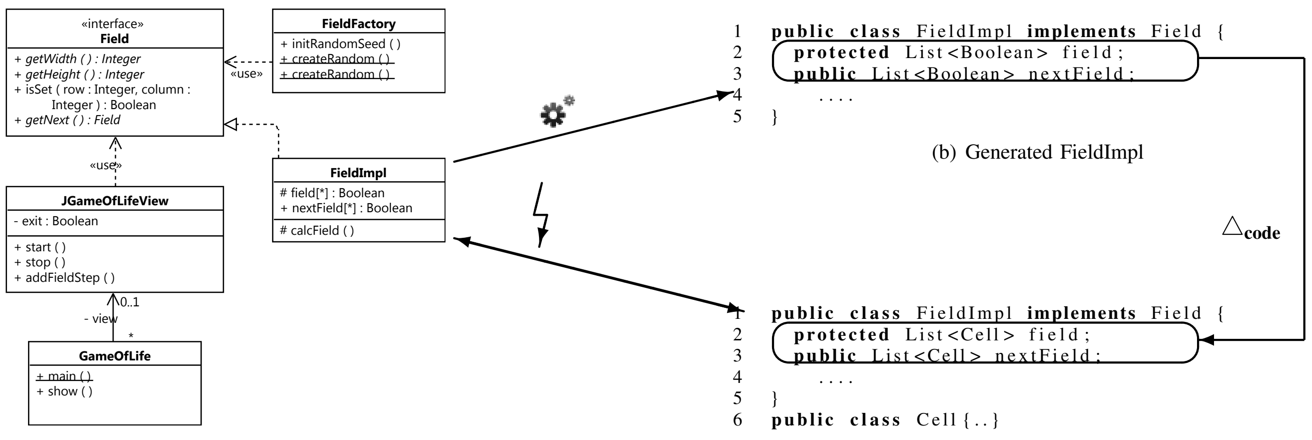
The proposed contributions of this work are thus:

1) A mechanism for live consistency checking among concurrent evolving development artifacts.
2) Adaptable consistency rules that allow for alternative interpretations and semantics of artifact (design model or source code) elements.

Next, we focus on these contribution.

## II. Motivational Example

This section introduces an illustrative example and discusses typical issues regarding model-and-code evolution in the context of the provided example. We chose a small implementation of Conway's *Game of Life (GoL)* [15], a simulation



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

(b) Generated FieldImpl

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

(a) GoL Class Diagram

(c) Desired FieldImpl

Fig. 1. Field Evolution

of cellular development based on specific rules. It is a non-player game; based on a random generated population, cell generations are computed one at a time. Figure 1(a) depicts the class diagram of the system. The class `GameOfLife` contains the `main`-method in which data structures are initialized and an infinite loop, calculating at each iteration a new generation, is started. This is visualized in the sequence diagram shown in Fig. 2(a). A second initialization sequence, in which a failure case is modeled, is shown in Fig. 2(b). Each generation of a GoL population is calculated from classes implementing the interface `Field` (e.g., `FieldImpl`). Notice that the first implementation (Fig. 1(b)) is generated from the given design model (Fig. 1(a)), as indicated by the gears in Fig. 1.

## A. Field

As an example of a change, consider that a developer decides to use a dedicated type `Cell` to represent individual cells in GoL, instead of using boolean values (as depicted by $\triangle_{code}$ in Fig. 1, the updated state of the source code is shown in Fig. 1(c)). With this change, the developer created an inconsistency between the design model and the source code as the type `Boolean` does not match the type `Cell`. Note that after this code evolution, an automated synchronization may either override the previous code evolution, or generate a new declaration (i.e., `List<boolean> nextField`) besides the existing, manually updated declaration `List<Cell> nextField`, which is incorrect. Either way, trying to automatically handle the manually introduced inconsistency may lead to unintended — and potentially still inconsistent — results.

## B. Sequence

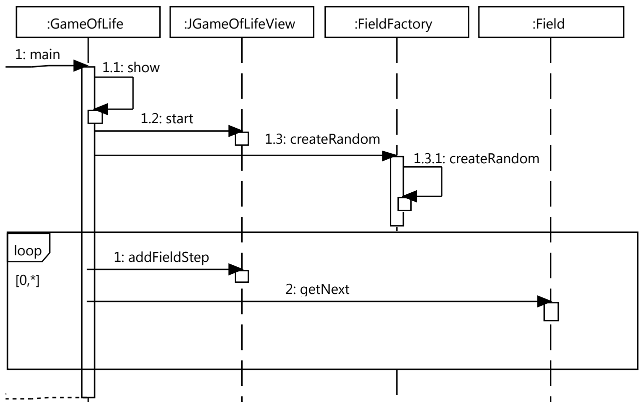
While the example above demonstrated a problem that some might consider a nuisance, there are much more severe and less trivial examples. Consider the following example of sequence diagrams. It is understood that a sequence of messages defined in a sequence diagram should be found in code as well [16] – if a message is not reflected by a corresponding method call in source code, this should be considered as inconsistency.

```
1  public static void main(String[] args) {
2    final GameOfLife gol = new GameOfLife();
3    gol.show();
4    gol.view.start();
5    Field field = FieldFactory
6    .createRandom(800,600,0.2f);
7    for (;;) {
8      try {
9        gol.view.addFieldStep(field);
10       field = field.getNext();
11     } catch (InterruptedException e) {
12       gol.view.stop();
13     }
14   }
15
16 }
```

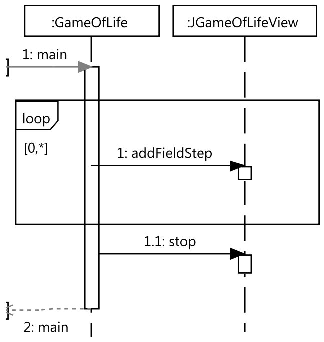
Listing 1. Java Initialization Sequence

To implement the behavior specified in the sequence diagrams, a developer produced the implementation shown in Listing. 1. Notice that the code actually contains an inconsistency. The sequence diagram, modeling the failure case (Fig. 2(b)), specifies that if a failure occurs, the simulation must be correctly ended (i.e., the endless loop must be stopped and the visualization must be ended with a `stop()`-call). However, the try-catch block (lines 8- 13 in Listing 1) is inside the endless loop, effectively stopping the visualization but not the endless loop. While the first sequence diagram (Fig. 2(a)) is reflected in the source code, the specified behavior in the failure case is not correctly reflected. Therefore, the source code does not conform to all its specifying views.

The interweaving of multiple models into one consistent source code still poses significant challenges. Currently there exists no technology effectively and correctly handling such scenarios. We thus propose *Model-and-Code Consistency Checking (MCCC)* to at least detect these problems as inconsistencies.



(a) Initialization Sequence



(b) Alternative Initialization Sequence

Fig. 2. Sequence

## III. 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 guides developers through the software development cycle with immediate feedback on inconsistencies. In doing so, it effectively helps developers to take measures to not let design models and source code drift apart. While consistency checking between design models and source code is far from trivial, relatively simple tasks are necessary for our approach to be applicable to development projects. The main task is writing a set of consistency rules that meet the needs of the project or domain or adapt the current set provided.

### A. Consistency Rules

MCCC relies on consistency rules provided by developers. These rules define invariants that the design models and source code are required to fulfill in order to be consistent. Let us illustrate how a developer could state such a consistency rule. Listing 2 shows a stylized informal example of an explicit consistency rule between `UML::Propertys` and `Java::Fields`:

```
1  UML::Property p
2  p.name==javaField(p).name &&
3  javaClass(p.owner)==javaField(p).owner implies
4  if p.cardinality=* then
5    javaField(p) is a collection or an array
```

Listing 2. Stylized Consistency Rule

The rule defines that if both the `UML::Property` and its corresponding `Java::Field` (given by the function `javaField`) exhibit the same name and both are owned by the same class (the corresponding Java class of a UML type is obtained by the function `javaClass`[1]), the multiplicity condition in lines 4-5 applies. The condition in the if-expression defines that if the given multiplicity (i.e., in UML a cardinality line 4) equals "*" the equivalent Java field must either be a collection or an array.

Before we can write consistency rules such as the one shown in Listing 2, we have to address the following problems: i) design models and source code are not integrated in a single cohesive language (how can a consistency rule between model and code be written without them being integrated?), ii) design models and source code do not allow inter-model navigation (how can one navigate conveniently among artifacts?), iii) design model and source code do generate different change sets $\triangle$ for each artifact (how can artifact changes be handled uniformly?).

### B. Framework

To address these problems, we developed a model-and-code consistency checking framework that integrates artifacts of different sources and provides navigation links that allow for simple writing of explicitly stated consistency rules and their efficient evaluation – even after artifact evolution – by an incremental consistency checker. An overview of the framework is given in Fig. 3.

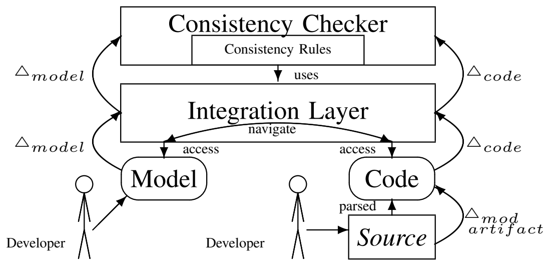


Fig. 3. Overview

### C. Artifact Integration

Both, design models and source code, are typical development artifacts in MDE. For detecting inconsistencies between those artifacts, they must be integrated in a manner that a consistency checker can access them in a unified way. Therefore, the metamodels of both artifacts must be integrated and inter-model navigation enabled.

*1) Metamodel:* Unfortunately, model and code artifacts are typically instances of different metamodels and are edited in different development tools. For example, a UML model element is an instance of an UML metamodel element and edited in a UML modeling tool. In turn, Java code needs to conform to the Java language specification and is typically edited in a Java programming tool. Moreover, retrieving information from models differs significantly from accessing source code. We provide a uniform, in-memory access to both, using the

[1]Functions `javaField` and `javaClass` will be discussed in Section III-C2

already in-memory design model and generating an in-memory representation for source code. Following, we refer to `Code` and `Model` as the in-memory representation of source code and design model, respectively – see bottom of Fig. 3. Both are accessible via the `Integration Layer`.

Although both, design model and source code, are now commonly accessible through a common infrastructure, each artifact still conforms to its respective metamodel, which is a problem for consistency checkers that typically work with a single metamodel only (e.g., [17], [18], [19]). This requires additional integration which is handled by the `Integration Layer`. It provides a single coherent metamodel for both `Model` and `Code` to the consistency checker (i.e., the consistency checker works with a single emulated metamodel that contains the metaclasses of both artifacts) in Fig. 3. Additionally, the unified metamodel 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.

*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 III-A, 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. Therefore, the `Integration Layer` augments the respective metamodel of the artifact with additional functionality for direct and bidirectonal navigation between model and code.

Such `Navigation Links` can either be `implicit` based on an inherent property of an `Model` or `Code` element as mentioned above, or `explicitly` stated.

## IV. Discussion

In the following, we discuss the greater context of this work in terms of artifact integration, possible collaboration patterns and its implications on incremental consistency checking.

### *A. Artifact Integration*

A key principle of MCCC is the integration of design model and source code in a way that a consistency checker can access both in a unified manner. This integration must happen on the level of metamodels and furthermore for specific artifact elements to navigate among design model and source code in a consistency rule. Both metamodels – design model and source code metamodel – must provide the means to navigate among the artifacts. Furthermore, `Navigation Links` (i.e., links among artifact elements that allow for bidirectional navigation) among artifact elements must be introduced. So far, this summarizes the principles of the previous section.

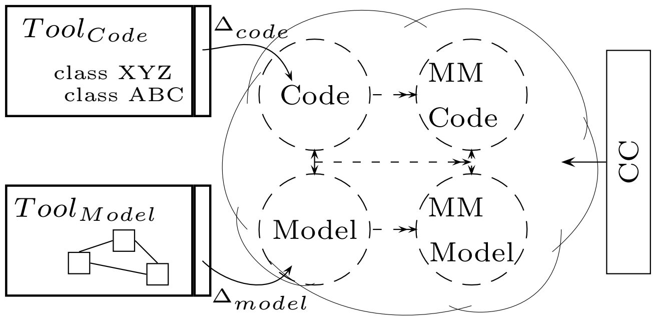


Fig. 4. Cloud

However, these principles may be realized quite differently. An intuitive solution is to alter both metamodels, so that they exhibit references to elements of another metamodel. Furthermore, `Navigation Links` can be constructed from information accessible in both design model and source code (e.g., if a `UML::Class` and `Java::Class` have the same name, they are considered opposites) or even expressed explicitly in corresponding artifact elements.

However, artificially appending navigation references to metamodels is to some degree depending on the used tools and their technologies (i.e., the *IBM Rational Software Architect* [20] uses the *Eclipse Modeling Framework (EMF)* [9] to represent both the UML metamodel and created models thereof, while *ArgoUML* [21] uses its own implementation of the UML standard, which can not easily be adapted). Therefore, a solution should be independent of actually used tools and their technology. Furthermore, `Navigation Links` among artifact elements may not easily be constructed from information in both artifacts or not easily explicitly stated.

Therefore, we propose to use an architecture, called *DesignSpace*. All development artifacts (and their metamodels) are stored in the DesignSpace using a unified representation. The consistency checker, integrated with the DesignSpace, is thus able to access all development artifacts in a unified manner. The unified representation is continuously kept in sync with development artifacts in their respective tools through adapters. This ensured that the unified representation of the artifact is always up-to-date, this is depicted in Fig. 4.

The DesignSpace also allows to append the unified representation of both metamodels with inter-model navigation references (depicted as unidirectional dashed arrow in Fig. 4 between `MM Code` and `MM Model`), without adapting the tools or their metamodels. Furthermore, `Navigation Links` can be established by adding navigation references to corresponding artifact elements, depicted as unidirectional arrow in Fig. 4 between `Model` and `Code`.

Please note that the concept of `instanceof` is represented in Fig. 4 as double headed dashed arrow. Both, source code and design models, are instances of their respective metamodels. A (potentially bidirectional) `Navigation Link` is an instance of a inter-model navigation reference.

### B. Collaboration

The integration of diverse development artifacts furthermore provides the possibility of supporting collaboration patterns among developers. Traditionally, a developer obtains a working copy of a design model or source code from a repository and adapts the artifact in private, before deciding to make the adaptations publicly available (e.g., by committing). This constitutes the collaboration philosophy promoted by SVN [22] or Git [23].

Typically, a developer work with either the model or the code but not necessarily both at a same time. However, changing one artifact may easily cause inconsistencies with the other artifact, as illustrated in Section II, and detecting such inconsistencies requires a consistency checker to access both kinds of artifacts involved: the adapted one (e.g., the source code in the private working copy), and the one remaining unchanged (e.g., design models in the repository). Unfortunately, the unchanged artifact is typically not available locally for a standard consistency checker.

By adding working copies to the DesignSpace, a consistency checker then has access to both the privately adapted artifact and the public, shared artifact. Thus, inconsistencies between an artifact in a working copy and another artifact in the repository can be detected efficiently (as both artifacts are stored in the DesignSpace) without requiring design/programming tool integration.

### C. Incremental Consistency Checking

Consistency checking between design models and source code can be performed at different times, e.g., before a developer decides to publish changes, at predefined intervals, or if the developer prompts the consistency checker to do so. After a developer is being notified about inconsistencies, he/she can either resolve the inconsistency, inform another developer, or even ignore the inconsistencies temporarily.

Incremental consistency checking, at the finest granularity possible (i.e., after every atomic change), means that consistency information about the current state of the system is available at all times. Deciding when to present this information to users is not of relevance for our approach but depends primarily on the kind of tool that is used, the kind of artifact that is edited, the kind of changes that are performed, and the user's individual preferences. For example, displaying updated consistency information might not be desired during a code refactoring but only after all changes have been executed. In contrast, a developer might prefer to be informed about the current consistency status even after atomic changes when working on a design model. However, doing consistency checking only at fixed intervals would mean that in the latter case there is no information available about the current consistency state of the system by the time it is desired.

Note that because of the efficiency of incremental consistency checking [24], frequently performing incremental revalidation after atomic changes does not impose an interruption in the workflow of developers. In addition, it is trivial to perform incremental consistency checking and delay only the presentation of consistency information while it might be hard to construct that information on demand when consistency checks are performed at fixed intervals.

### D. Repairs

After detecting inconsistencies, a subsequent action is to eventually resolve them [25]. Automatically derived repairs may be provided to a developer in order to resolve inconsistencies [26], [27]. Suppose that a developer may want to rename a `Java::Class`, and a consistency rule imposes that its corresponding `UML::Class` needs to exhibit the same name. A possible automatic refactoring could be to rename as well the `UML::Class`.

## V. Related Work

MDE is a wide and active field of study, different challenges such as modeling 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)* [9] is a modeling framework with a code generation mechanism. A new annotation to mark source code 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-prescribed code. This approach mitigates the update problem via deep separation principle but as well tries to enforce model and code consistency by generating code.

Heidenreich et al. [28] 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. [29] go a step further, based on the *CHESS Modelling Language* [30] and the *Action Language for Foundational UML (ALF)* [31] 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* [10] uses a specific *Architectural Description Language (ADL)* [32] that integrates a new concept called interaction contract, that 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* [33] 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* [12]. New interface mechanisms are used as ADL in the design phase, in implementation phases programming interfaces. To specify the collaboration among components, *Aspect-Oriented Programming (AOP)* [34] 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. [35] use a batch like approach, that 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.

## VI. Conclusion and Future Work

In this paper, we outlined the principles of MCCC, a novel approach for model-and-code consistency checking that detects inconsistencies between design models and source code. For future work, we will develop a proof-of-concept implementation and conduct case studies with industrial development projects. Moreover, we plan to integrate approaches for the fixing of inconsistencies in our implementation and to increase support for distributed development by providing a consistency checking environment that integrates data from different development tools.

## Acknowledgements

We gratefully acknowledge the Austrian Science Fund (FWF): P 25289-N15 and P 25513-N15

## 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. W. Boehm, "A spiral model of software development and enhancement," *IEEE Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[6] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[7] K. Schwaber and M. Beedle, *Agile Software Development with Scrum*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st ed., 2001.

[8] 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.

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

[10] 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.

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

[12] N. Ubayashi, J. Nomura, and T. Tamai, "Archface: a contract place where architectural design and code meet together," in *ICSE*, vol. 1, pp. 75–84, 2010.

[13] 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.

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

[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. 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.

[18] 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.

[19] 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.

[20] "IBM Rational Software Architect https://www.ibm.com/developerworks/ rational/products/rsa/ ,2013.."

[21] "ArgoUML http://argouml.tigris.org/ ,2013.."

[22] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, *Version control with subversion - next generation open source version control*. O'Reilly, 2004.

[23] J. Loeliger, *Version Control with Git - Powerful techniques for centralized and distributed project management*. O'Reilly, 2009.

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

[25] R. Balzer, "Tolerating Inconsistency," in *ICSE*, pp. 158–165, 1991.

[26] A. Reder and A. Egyed, "Computing repair trees for resolving inconsistencies in design models," in *ASE*, pp. 220–229, 2012.

[27] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency Management with Repair Actions," in *ICSE*, pp. 455–464, 2003.

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

[29] 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.

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

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

[32] 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.

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

[34] 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.

[35] 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.