Fulfilling Industrial Needs for Consistency Among Engineering Artifacts

Luciano Marchezan
Inst. of Software Systems Engineering
Johannes Kepler University Linz
Linz, Austria

Wesley K. G. Assunção
Inst. of Software Systems Engineering
Johannes Kepler University Linz
Linz, Austria

Edvin Herac
Inst. of Software Systems Engineering
Johannes Kepler University Linz
Linz, Austria

Felix Keplinger
Inst. of Software Systems Engineering
Johannes Kepler University Linz
Linz, Austria

Alexander Egyed
Inst. of Software Systems Engineering
Johannes Kepler University Linz
Linz, Austria

Christophe Lauwerys
Corelab MotionS
Flanders Make
Belgium

***Abstract*—Maintaining the consistency of engineering artifacts is a challenge faced by several engineering companies. This is more evident when the engineering artifacts are created using different tools and have different formats. This is the context of a company that builds agricultural machines, where components are developed using a decentralized iterative process. In this study, we present an approach developed in collaboration with an industry partner to address the issues and requirements of a real engineering scenario. These issues include the manual execution of consistency checking, without guidelines that formalize the activity. Furthermore, the industry partner aims at a flexible solution that can be applied without disrupting the current development process significantly. The proposed approach applies consistency rules (CR) defined to automatically detect and provide inconsistency feedback to engineers in real-time. The approach presented in this work also allows the customization of the CRs, giving flexibility to how the consistency checking is applied. The feasibility of our approach is demonstrated in such an industrial scenario, with a discussion about how the issues were addressed and the limitations of the current solution. We also perform a scalability evaluation showing that the approach can be applied in large systems (up to 21,061 elements) in a reasonable amount of time, taking less than 0.25 milliseconds to apply a CR, in the worst cases.**

***Index Terms*—Model-Driven Engineering, Consistency checking, Trace generation, Consistency flexibility**

I. Introduction

Consistency checking is widely used in companies to find inconsistencies in engineering artifacts [1]. Benefits of consistency checking include improving the quality of artifacts, reducing the chance of requirements not being met by design or implementation, as well as ensuring the safety of the system [2]. To achieve these goals, consistency checking approaches apply different strategies, commonly for checking the consistency of UML models [3]–[8]. Due to the importance and benefits of the consistency checking activity, research in the field has expanded beyond models, dealing with different artifacts [2]. These approaches allow companies to check the consistency of artifacts created with different tools [9]–[11] which is important since most companies rely on different types of engineering artifacts [12].

This is the case for a company that builds agricultural machines, a partner in this study, where software and hardware components must be developed and maintained until the machine development process is finished. Flanders Make, a research center that acts in Belgium's manufacturing industry, collaborates with this company to address their issues regarding consistency checking (Section II). These issues include the consistency checking being performed manually, possibly leading to errors. Also, the company relies only on the knowledge of engineers for checking the consistency among the artifacts. Since the engineers responsible for the consistency checking may change, e.g., leaving the company, this process can lead to different results. Lastly, the company requires flexibility regarding the consistency checking process due to its decentralized development process, where engineers have different preferences and work from different locations.

In this work, we present an approach (Section III) designed to solve these issues, as well as to conform to specific technological requirements such as being executed in the tools used by this company. Our approach supports the definition of consistency rules (CR) that can be used to automatically detect inconsistencies between two or more engineering artifacts. Relationships between artifacts from different tools are created by generating trace links that connect two properties of different artifact types, e.g., UML models and source code. The CRs provided with our approach automate the way that the consistency checking is applied, being always the same independently of who is applying the approach. Furthermore, the approach is flexible, allowing the modification, creation, and deletion of CRs at runtime.

We demonstrate the approach's feasibility in the industrial scenario, discussing how we address the scenario's issues (*Contribution 1*) and how challenges and limitations impacted the solution (*Contribution 2*). We also present an empirical scalability evaluation demonstrating how the approach handles the consistency checking between Java source code and UML models (*Contribution 3*). This evaluation (Section IV) is performed using 28 CRs that check for inconsistencies between both artifact types in six real systems. Results show that the approach can be applied in a variety of systems ranging in their size from 7,613 to 21,061 elements. The application of CRs is performed within a reasonable amount of time as each CR is executed in less than 0.25 milliseconds, in the worst cases. Related work is presented in Section V and conclusions in Section VI.

II. Background

In this study, we collaborate with Flanders Make,[1] a research center acting in the Belgium manufacturing industry. Flanders Make contributes to the technological development of vehicles, machines, and factories, providing services to multinational companies. The problem investigated in this study originates from one of Flanders Makes collaborations with a company that builds agricultural machines composed of sensors, actuators, engines, and other hardware components. The machines also contain software components and interfaces describing their communication. The systems of these machines can have between 10 and 15 components (depending on the configuration desired). This company has several teams with engineers responsible for multiple components in their domains, including hydraulics, mechanics, and software. Each team has from 10 to 50 engineers, depending on the number of components that they are responsible for.

The communication between the system components is performed through the Robot Operating System (ROS) protocol [13]. ROS provides a set of software libraries and tools for building applications with an anonymous publish/subscribe service that allows message exchange between different ROS processes. ROS uses the Interface Description Language (IDL) [14] for message definition and serialization. In this study, Flanders Make collaborates with our institute to combine their experience and provide a solution for the agriculture company problem. The results aim at being beneficial to the industry partner, as well as serve as a case study for the application of research technologies from academia in an industrial context. Flanders makes provided an industrial scenario that illustrates the development process of agricultural machines and their components.

A. Industrial Scenario

The incremental development process of the agricultural machines is illustrated in Figure 1. The company follows a decentralized process where different teams of engineers work on various components for the same machine (Figure 1 depicts the process from the perspective of one team). The process presented in this study is a simplified version of their real process. Furthermore, our collaboration with them is focused on addressing issues that happen during the `Design` and the `Development` steps. During the `Design` step, one or more engineers from the team are responsible for modeling the components using UML models. These models express the structure and the communication of the software and hardware components.

[1]Flanders Make website: https://www.flandersmake.be

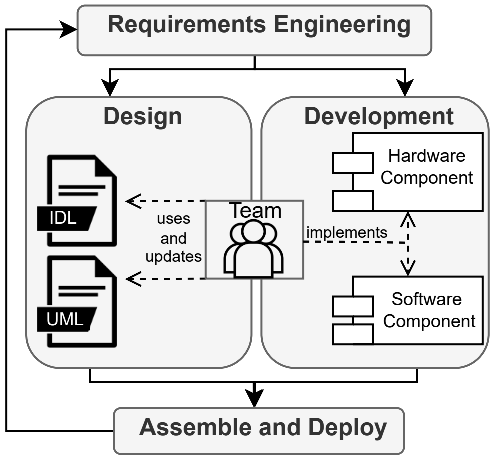
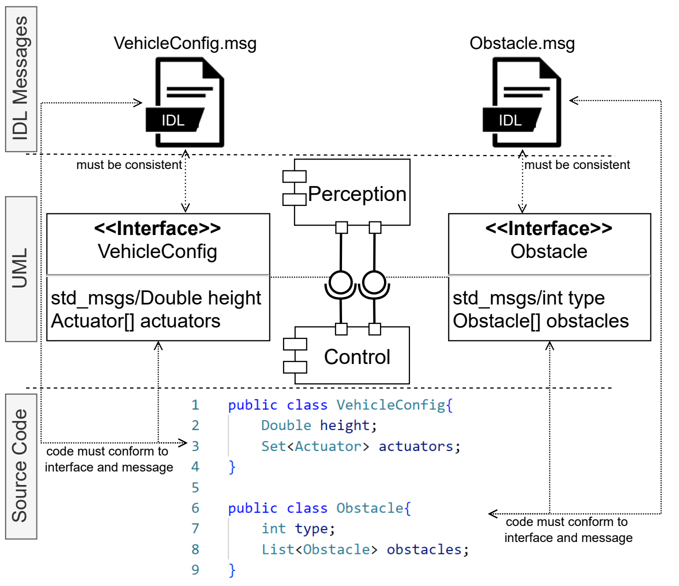


Fig. 1: Development process of agricultural machines.

One example of UML model is illustrated in the middle part of Figure 2. Here, two components (`Perception` and `Control`) are being designed. These components exchange messages using two interfaces, represented as the interfaces `VehicleConfig` and `Obstacle`, describing how the communication of these components should be implemented. The company uses Microsoft Visio to create UML models. The interfaces are also defined as IDL messages (top part of Figure 2) which are a textual representation of the UML interfaces and are used by ROS to realize the communication of software and hardware components. For instance, they allow the mapping of the components designed to different object-oriented programming languages, such as C++ and Python.

As illustrated in Figure 1, the UML models and the IDL messages from the `Design` step are used and updated by the engineers when implementing components in the `Development` step. Since engineers have different preferences when implementing the components, they can rely either on the IDL messages or on the UML models. Thus, the UML models and the IDL messages must be consistent with each other. In the current context of the company, however, *the consistency of these artifacts is performed manually (Issue 1).* This issue is represented by the `must be consistent` connection between the IDL messages and the UML models at the top part of Figure 2.

Considering the manual process of fixing inconsistencies, if an engineer identifies an inconsistency, e.g., an interface property defined in the UML model which does not exist in the IDL definition, this inconsistency must be fixed. This leads to a problem since *the identification of the inconsistency at this point relies completely on the engineer's knowledge, being an error-prone activity (Issue 2).* Such inconsistencies, if not fixed, lead to rework, as errors will emerge later, e.g., during the integration of components of different teams. Furthermore, as illustrated by Figure 1, the `Design` and `Development` steps are concurrently performed. This means that changes in the system components can happen either in the UML models, the IDL messages, or the source code.



```java
public class VehicleConfig{
    Double height;
    Set<Actuator> actuators;
}

public class Obstacle{
    int type;
    List<Obstacle> obstacles;
}
```

Fig. 2: Engineering artifacts and their intended relationships.

Note that the source code of the software components has no direct connection to the UML models or the IDL messages, e.g., traceability. Hence, if the structure of a component changes in the source code, the UML model has to be manually updated. At this moment, if an engineer decides to change the design of a component, then the source code becomes inconsistent and must be updated (*Issues 1* and *2*). These issues are illustrated by the relation `code must conform to interface and message`, between IDL messages, UML models, and the source code. Once again, this process is currently performed manually, being error-prone.

Moreover, there are cases where the implementation of source code based on the UML model (and vice-versa) may be ambiguous. This is more evident when components modeled in UML can be implemented in different programming languages, which is the case of agricultural machines. Since each language can have its data structures and primitive types, the way of checking the consistency between the source code and the models may change. For instance, the collection `actuators` defined in the UML interface `VehicleConfig` is implemented as a *Set* in the source code of class `VehicleConfig` (bottom part of Figure 2). Another example is the collection `obstacles` that describes the possible obstacles in a route of an agriculture machine. This collection is implemented in the source code as a *List* instead of a *Set*. In each case, the consistency checking should be applied differently, allowing engineers to customize the checking mechanism. Hence, *engineers must have flexibility concerning how consistency checking can be applied*(Issue 3).

The three issues led us to define the goals of this study, which are related to the requirements of the industry partner regarding a possible solution.

B. Goals and Requirements

To address the aforementioned issues, we define the goals of a possible solution as: i) automatically detecting inconsistencies based on consistency rules (Issue 1); ii) increasing awareness about consistency between engineering artifacts (Issue 2); iii) allowing the definition of customizable consistency mechanisms between engineering artifacts (Issue 3).

Our industry partner also defined requirements that the solution must address. Firstly, they require that consistency checking can be performed in real-time on Microsoft Visio. The second requirement is allowing the use of GitHub repositories to connect the IDL message definitions and the source code implemented to the UML models from Visio. Considering the source code, for demonstration purposes we agree on providing support for Java, although the company uses mostly C++ and Python. The main reason is that we have already some results regarding how Java source code can be checked for consistency [15], [16]. Additionally, the current workflow of the development process must not be disrupted significantly, i.e., the current engineering tools and artifact types should not be replaced. Moreover, engineers may choose to design the models and implement the source code from different perspectives, e.g., looking first at the IDL messages, or first creating the source code and then updating the models. Hence, the solution must not be intrusive. Thus, we avoid relying on code generation since it could impact the development process workflow, which is not desired by the industry partner. In the following section, we describe an approach applied as the solution to address these goals and requirements.

III. Approach's Realization

In this section, we describe the main definitions of our approach and its overall workflow.

A. Approach's Unique Representation

To check for the consistency of different types of artifacts, we need to compare their properties and values. To achieve this, we define a unique representation that is illustrated by the UML class diagram presented in Figure 3. In this unique representation, a `model` consists of `model elements`, which contain `properties`. A property has one `value` or multiple values (collections). Model, model elements, and properties have types based on the metamodel of the artifacts being represented. This relationship is illustrated at the metamodel level, at the top part of Figure 3. Using this definition, the interface `Obstacle` from Figure 2 is a model element of the type *interface* with properties such as a name (of the type *String*) with a value equals to "Obstacle" and the collection called `obstacles`. The source code can also be represented using this definition.

The class `VehicleConfig` (bottom part of Figure 2) has properties such as the fields `height` and `actuators` of the types *Double* and *Set*, respectively. Model elements of different model types can be connected using `trace links`.

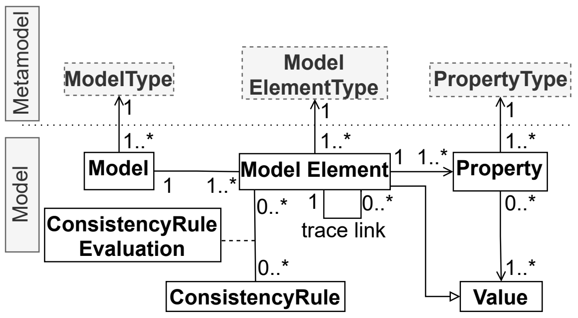


Fig. 3: UML class diagram of the unique representation used by our approach.

CR 1 All properties of a UML interface must be implemented as fields in the source code.

```
context UML interface inv:
self.linkedSourceCode.traces -> forAll(sourceCodeClass : <
      SourceCodeClass> | self.ownedProperties -> forAll(
      umlProperty : <UMLProperty> | sourceCodeClass.fields->
       exists(field : <Field> | field.name = umlProperty.
      name)))
```

Trace links are represented by the relation *one-to-many* that model elements have with themselves (center of Figure 3). Traces links are important for our approach as they allow us to perform consistency checking considering artifacts from different model types, i.e., originating from different tools such as Visio and Github.

To check the consistency of the model elements, the approaches apply `consistency rules` (CR) which are conditions that model elements must fulfill. A CR is written for a context, which is a model element type (see Consistency Rule in Figure 3). An example of CR is given in CR1 where the rule checks if all properties of a UML interface are implemented in the source code as fields. Note that the definition is based on the language used for creating the CRs. In our case, we use the Abstract Rule Language (ARL).[2] ARL is based on OCL [17] with minor differences, e.g., instead of having "OCLAsType" the ARL language has "asType" expression. Continuing on the definition of CR1, the rule iterates over all trace links from the UML interface to the source code. The CR uses a universal quantifier (`forAll`) to check if all traces of source code conform to the CR. It checks if, for all UML properties, there is at least one (`exists` quantifier) field implemented with the same name as the given property. When a CR is applied to a model element, a `consistency rule evaluation` (CRE) is created. Hence, the associative class `ConsistencyRuleEvaluation` connecting `ModelElement` and `ConsistencyRule` in Figure 3. A CRE evaluates to a Boolean value, *true* (consistent) or *false* (inconsistent). Details about the approach are described in the following section.

[2]Language documentation is available at https://isse.jku.at/designspace/index.php/Abstract_Rule_Language

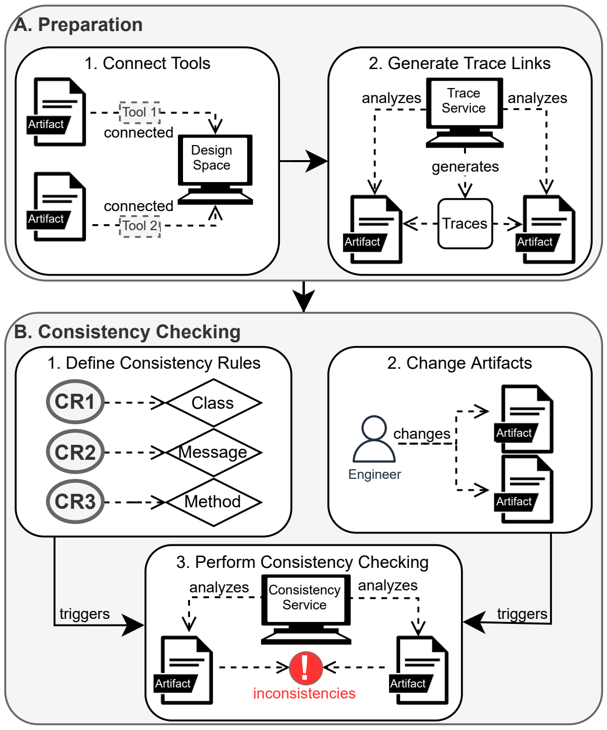


Fig. 4: Overview of the approach's workflow.

B. Approach's Workflow

Our approach's workflow, illustrated in Figure 4, is divided into two phases, namely `Preparation` and `Consistency Checking`.

1) Preparation: During the `A.Preparation` phase (Figure 4) the tools from which artifacts are created must be accessible to the consistency checking service, also trace links between artifacts are generated using the trace service.

Connect Tools: to support multiple types of artifacts, these need to be accessible to the trace and consistency checking services. This is achieved by connecting the engineering tools (see `Step A1` in Figure 4), e.g., Microsoft Visio, to the DesignSpace server [18] where the services are running.[3] We decided to use the DesignSpace server for establishing the connection of these artifacts as it provides an easily extensible infrastructure. For instance, we implement and use the trace and consistency checking services that are running in the DesignSpace server. Figure 5 illustrates a simplified version of the DesignSpace architecture [11], [18]. Services such as the trace (`Step A2`) and consistency (`Steps B1, B2, and B3`) are running on the server. Tools can be connected to the server through the use of tool adapters, that send and receive artifacts' information used by the services to achieve different goals, such as to check for consistency. To address the requirements of the industry partner, we developed tool adapters as plugins for connecting Visio, GitHub, and IntelliJ to the DesignSpace server.

[3]Video demos demonstrating this connection are available at our online repository [19]

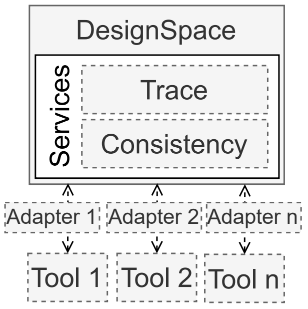


Fig. 5: The architecture of the DesignSpace server.

Tool adapters are responsible for performing the communication between the tools and the DesignSpace server (Figure 5). For instance, the Visio tool adapter reads the shapes designed in Visio using UML elements and transforms them using the unique representation (Figure 3). This is achieved by the use of a specific parser that identifies the Visio shapes corresponding to UML elements and instantiates them into DesignSpace as model elements with properties and values from the unique representation. Similarly, the tool adapter for GitHub reads the textual files present in the repository to which DesignSpace is connected. This repository is set by the engineers in a configuration file and once connected, DesignSpace fetches the files. If a file has an *.msg* extension, which means it is an IDL file written for ROS, a parser is used to transform this *.msg* using the unique representation.

The IntelliJ adapter also has a parser that transforms the elements from the *.java* files into instances of the unique representation. This adapter uses a Java-specific metamodel that defines classes, methods, and fields, among other properties of the Java programming language that can be analyzed during the consistency checking by applying CRs. These steps can be replicated in other tools, such as by parsing *.java* files directly from GitHub or other IDEs to create Java-like elements within the DesignSpace server.

Generate Trace Links: the generation of trace links (`Step A2`, Figure 4) is semi-automated and it is based on two decisions from the engineer. The first decision is related to which model element types can have traceability. For example, the engineer may decide that UML interfaces should be traced to IDL message definitions. The second decision is related to which model elements, of the model element type selected, should be traced. Our approach aids this decision by providing a trace service with an automatic tracing generator. This tracing generator can be customized for different contexts. In this work, we use the generator based on artifact names. Since the agriculture company also relies on naming for comparing IDL message definitions to UML interfaces, this strategy fits their context. Thus, if two model elements have the corresponding types that need to be traced, e.g., UML interface and IDL message, and the same name, the approach automatically creates a trace between them. We provide, however, a way of extending the trace service to other contexts as engineers can modify the auto-generated traces, create, or delete traces.

CR 2 All properties of a UML interface must be represented as fields in the IDL messages.

```
context UML interface inv:
self.linkedIDLMessages.traces-> forAll(idlMessage : <
    IDLMessage> | idlMessage.fields ->forAll(idlField : <
    IDLField> | self.properties -> exists(umlProperty : <
    UMLProperty> | idlField.identifier = umlProperty.
    identifier and idlField.type = umlProperty.type)))
```

CR 3 A property defined as a collection in a UML interface must be implemented as a Set in the source code.

```
context UML interface inv:
self.linkedSourceCode.traces -> forAll(sourceCodeClass : <
    SourceCodeClass> | self.ownedProperties -> forAll(
    umlProperty : <UMLProperty> |  sourceCodeClass.fields->
     exists(field : <Field> | umlProperty.type = '
    Collection' and umlProperty.name = field.name implies
    field.type = 'Set')))
```

The `Preparation` only needs to be performed once per tool, being less intrusive as possible.

2) Consistency Checking: The consistency checking applied by our approach is incremental, thus not necessarily all model elements are checked every time the consistency checking is triggered. More specifically, the consistency checking (`Step B3` in Figure 4) can be triggered by two actions: i) when CRs are created, deleted, or modified (`Step B1`); or ii) when the artifacts are changed (`Step B2`).

Define Consistency Rules: our approach applies CRs in the artifacts connected to the DesignSpace server where the consistency service is running. These rules can be created and customized using the ARL language. In this approach, we already provide some rules that aim at addressing the Issues discussed in Section II. One example of a rule is CR1, which checks all UML interfaces of a UML model. Another example of CR provided with the approach is CR2. This rule checks for inconsistencies considering the fields of IDL messages and UML interfaces. This addresses the *must be consistent* relationship between these two artifacts, illustrated in Figure 2. The definition of additional CRs is possible by using our ARL language documentation. Engineers define the CRs by choosing a context (model element type) and the definition in ARL as a *String*. This information is provided in the tools used, such as Visio, and added to the DesignSpace server. Once the CRs are added to the server, the consistency checking mechanism will use them whenever `Step B3` is triggered.

Another example of CR applied to the artifacts is described in CR3. This CR checks if, for a given property of a UML interface of the type *Collection*, there is a corresponding field in the source code with the same name and of the type *Set*. Here, the CR is checking the consistency between the source code and UML. A similar rule can also check the same type of consistency between the source code and the IDL messages. The only difference is the context of the rule and minor modifications to address the IDL message structure. The approach provides flexibility to the consistency checking by allowing engineers to create, delete, modify, enable and disable CRs at runtime.

Change Artifacts: the consistency checking service listens for changes in the artifacts analyzing if the model element changed is part of any CRs' context. For instance, CR1, CR2, and CR3 have the context of UML interfaces. Thus, if a UML interface is modified, this change triggers the consistency checking service to start evaluating CR1, CR2, and CR3 for this interface. Note that only the rules that have UML interface as context are evaluated (or re-evaluated). However, changing a UML property also changes the UML interface, depending on the operation performed. For example, if the property is deleted, then the UML interface will have fewer properties, triggering the consistency checking service.

Perform Consistency Checking: once the consistency checking is triggered, either by `Steps B1` or `B2`, the model elements that are in the context of the CRs are evaluated. To describe how consistency checking is applied to CRs, we can consider CR2. If the service was triggered by `Step B1`, CR2 is applied to all model elements of its context, UML interface (Context in CR2), creating CREs. If the service was triggered by `Step B2`, however, only model elements that were changed and are in the context of the rule are evaluated. Continuing on the definition of CR2, the consistency checking service accesses the property "linkedIDLMessages.traces" from each UML interface. This property contains a collection with all trace links created between UML classes and IDL messages (`Step A2`). The CRs' definition in ARL is parsed using a compiler that calls functions for each one of the ARL expression types present in the CR. For instance, in CR2 the `forAll` iterator is used to check if, for all fields of an IDL message in that trace link, there is at least one UML property (`exists` iterator) corresponding to the IDL field. This is performed by two equals (=) expressions "idlField.identifier = umlProperty.identifier" and "idlField.type = umlProperty.type" connected by an `and` conjunction. Thus, the functions called by the parser correspond to the `forAll`, `exists`, and to the `and` conjunction with the two equals expressions. The approach structures the evaluation of a CR as a tree where, in this case, the `forAll` expression would be the parent of the `exists` expression which is the parent of the `and` conjunction. The calculation of the CRE also considers this tree structure, i.e., the results of the leaf nodes and the parent nodes are combined. For example, a `forAll` expression needs all the child nodes to be consistent. If the CRE of all child nodes results in *true*, there are no inconsistencies in the given UML interface. If the result of any child node is *false*, the parent node is also *false* due to the `forAll` logic.

Once the consistency checking is finished, feedback is given to the engineer (`Step B3` in Figure 4). Figure 6 illustrates an example of the feedback given in Visio. The consistency feedback in the tool is given as a comment added to the inconsistent element, in this case, the `Obstacle` interface. Furthermore, the tool changes the color of the inconsistent element (by default the inconsistent element is changed to red, but the engineer can customize that).

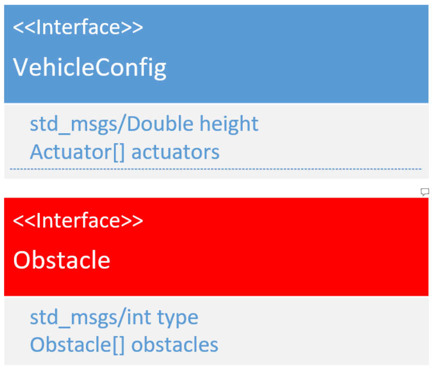
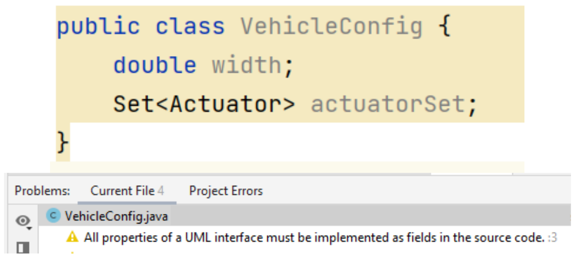


Fig. 6: Excerpt of Visio showing the consistency feedback as a comment.



```
public class VehicleConfig {
    double width;
    Set<Actuator> actuatorSet;
}
```

Problems: Current File 4 Project Errors

VehicleConfig.java

All properties of a UML interface must be implemented as fields in the source code. :3

Fig. 7: Excerpt of IntelliJ showing the consistency feedback as a warning.

Figure 7 shows an example of the feedback given as a warning in IntelliJ IDEA. The engineer can also customize how the feedback is given, e.g., underlining, an error message, or a warning.[4] The approach proposed can be applied as the solution for the industrial scenario presented in Section II. Our approach, however, can be extended to be applied in different scenarios where different tools are used. We discuss and demonstrate this in the next section.

IV. Evaluation of the Approach

In this section, we describe how the proposed approach is applied in the industrial scenario. We also present and discuss a scalability evaluation.

A. Application to Industrial Scenario

Addressing the Issues: rules such as CR1 and CR2 are designed and provided to the industry partner to address *Issues 1* and *2*. The application of these rules by our approach can be used to automatically detect inconsistencies in their context, increasing the awareness of engineers about the consistency of the artifacts. Furthermore, additional CRs are applied to check if the IDL messages have trace links to the UML interfaces since all IDL messages must be represented as UML interfaces, and vice-versa (CRs 7 and 8).[5] These rules are not detailed in this paper, for the sake of simplicity. In

[4]Video demos for the tools are available at our online repository [19].

[5]All CRs applied to the industrial scenario are available in our repository [19].

addition, CR3 is defined to address *Issue 3* as we aim at giving a customizable mechanism for engineers. Hence, the tool adapters that we developed for the tools allow engineers to edit the CRs at runtime. CR3, for instance, could be customized to check if the field is implemented as *List*. Furthermore, the tool adapters also allow engineers to have two or more CRs with similar scopes, e.g., one CR for checking if the collection is implemented as a *Set* and another CR for checking if the collection is implemented as a *List*. The tool adapters also allow engineers to enable and disable CRs at runtime enhancing the customization of the consistency checking service.

Addressing the requirements: transforming artifacts of different types into a unique representation allows the approach to use the same checking service independently of the domain. The only requirement is to apply a transformation from the original format, e.g., Java, into the unique representation. To do this, we implement tool adapters as plugins in the tools that create these artifacts. By applying this strategy, we can easily apply our approach to tools that are already used by the industry partner, addressing their requirement for a non-intrusive approach. Furthermore, the unique representation allows us to extend the applicability of the approach to any tool that supports a property/value relationship between model elements. It is important, however, to check the scalability of the approach as it may be required to apply it in projects larger than the one from our industrial partner.

B. Scalability of the approach

We aim at investigating the scalability of the trace and consistency checking services by applying our approach to large systems. Since the system from the industry partner cannot be presented in detail due to a confidentiality agreement, we do not use it in this evaluation.

Evaluation design: we evaluate our approach's scalability by considering three Research Questions (RQ):

RQ1. To what extent is our approach scalable, considering the number of CREs and inconsistencies that can be identified?

RQ2. What is the time required for applying the consistency checking and trace services?

RQ3. How does the approach react to changes performed in the artifacts connected?

To answer the RQs, we selected six real systems implemented in Java. These systems are shown in Table I and have been used for evaluating related work [20], [21]. Three of these systems (S1, S2, and S3) already possessed UML models. This is required for evaluating the trace service considering different model types. For the other three systems (S4, S5, and S6) we reverse-engineered the source code to generate the UML models. The reverse engineering was performed using Modelio.[6] We applied the consistency checking and trace services using a bidirectional strategy.

[6]Modelio (https://www.modelio.org/).

TABLE I: Systems used in the scalability evaluation.

Id	Name	# elements			# CREs	# incons.
		Java	UML	Total		
S1	VOD3	15386	1706	17092	13061	8304
S2	BiterRobocupC.	15588	2628	18216	15033	8334
S3	ObstacleRace	18051	3216	21267	16034	10824
S4	Gantt	11281	9780	21061	10175	2208
S5	BankApp.	4185	3428	7613	3609	812
S6	TaxiSystem	3969	4231	8200	3943	1283

We applied 28 CRs in the systems, where 14 CRs checked if Java source code was consistent with the UML models and 14 CRs checked if the UML models were consistent with the source code. These CRs were defined based on standard UML and Java constraint-based mechanisms [?], [4]. Collecting the data about the number of CREs and inconsistencies identified in these systems is used to answer RQ1. We also measure the runtime required for performing these tasks, including the creation of traces (RQ2).

Furthermore, to collect data to answer RQ3, we simulated engineer's changes based on common refactoring changes applied in real systems [22]–[24]. More specifically, we applied nine different types of changes for Java source code: *Extract Method*, *Move Method*, *Rename Method*, *Delete Method*, *Rename Field*, *Move Field*, *Delete Field*, *Rename Class*, and *Delete Class*. We also applied nine types of changes for the UML models: *Extract Operation*, *Move Operation*, *Rename Operation*, *Delete Operation*, *Rename Property*, *Move Property*, *Delete Property*, *Rename UML Class*, and *Delete UML Class*. For each change type, we applied 100 changes in 100 random model elements (one change per model element) for each system. This distribution resulted in 1,800 changes performed for each system. After the execution of each change, we collected the data related to the CREs evaluated, inconsistencies created and runtime required.

The specifications for the execution environment are an Intel Core i7-7700 CPU @3.6GHz with 16GB RAM (8GB available for the DesignSpace server) and Windows 10 x64-based. The evaluation data, Java source code, UML models, as well as the CRs used, are available at our online repository [19].

Answering the RQs: Table I shows that the size of the systems (# elements) ranges from 7,613 (S5) to 21,267 (S3) in total, being from 3,969 to 18,051 considering Java elements and from 3,428 to 9,780 considering UML elements. Considering the results about CREs and inconsistencies found per system (see Table I), the approach created from 3,609 (S5) to 16,034 CREs (S3). This led to the identification of several inconsistencies ranging from 812 (S5) to 10,824 (S3). We also consider the results of CREs and inconsistencies generated per each CR applied, illustrated in Figure 8. Most CRs generated less than 2,600 CREs for all systems, with two exceptions being CR3B and CR9B [19]. Firstly, CR3B checks if all methods in a Java class have trace links to the corresponding UML operations. Secondly, CR9B checks if the return type of the Java method matches the return type of the corresponding UML operation.

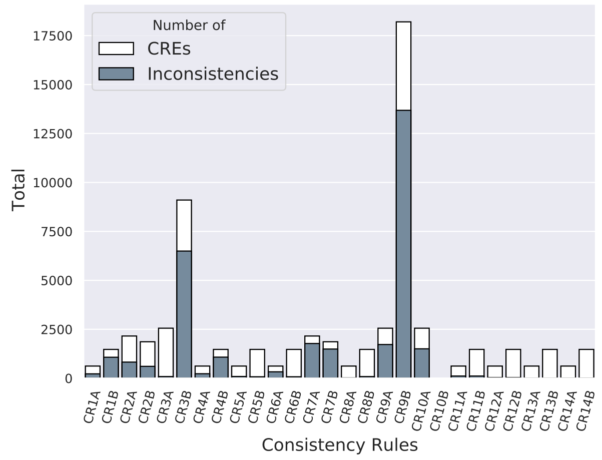


Fig. 8: CREs and inconsistencies per CR.

Since the number of methods in the systems was always larger than the number of classes or fields (the other two types of elements evaluated by CRs), it was expected that the number of CREs for CRs checking methods would be higher. Considering these results, we argue that *the consistency checking service is scalable, as it can be applied in large systems to create CREs and identify several inconsistencies (RQ1)*.

Furthermore, the result regarding the number of inconsistencies of some CRs (Figure 8) is directly impacted by the number of traces generated or missing. Considering CR3B, for instance, if a trace between a Java method and a UML operation is not created by the trace service, that method is considered inconsistent. Table II shows the results regarding the number of traces generated between Java classes, methods, and fields to UML classes, operations, and properties, respectively. Column *# elements* shows the number of elements that were considered for the trace service, namely java classes, methods, and fields. We only considered these elements, as the CRs applied were also designed for the context of these elements. However, other elements from the systems were also analyzed by the consistency service by using the traces. For instance, rules CR10A/CR10B (see [19]) check if the parameters of a method are the same in both source code and models, respectively. Since we have the traces of the methods, we can access their parameters, avoiding the need to create traces between parameters.

Table II also shows the number and percentage of traces generated from the elements considered (Column *# traces*). For the three systems that were not reverse-engineered (S1, S2, and S3), the trace service generated between 11% and 28% traces between elements. By applying the CRs that check for traces, however, we can identify the elements for which the traces were not generated. These traces can be created manually by an engineer. These tasks are related to the `A.Preparation` (Figure 4) and only need to be performed once for each tool. Furthermore, the traces created for S4, S5, and S6 did not cover 100% of the elements used.

TABLE II: Summary of results of applying the trace service.

System	# elements	# traces (%)	Time (ms)	Avg. time (ms)
S1	2989	617 (20)	519.97	0.17
S2	3230	909 (28)	484.54	0.15
S3	3708	411 (11)	634.48	0.17
S4	1589	1447 (91)	481.34	0.3
S5	527	490 (92)	109.03	0.21
S6	392	387 (98)	98.04	0.25

# elements: java classes, methods, and fields

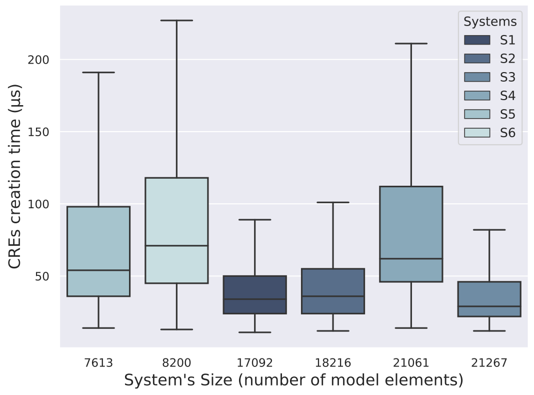


Fig. 9: Time required for creating CREs per system.

The results considering the number of traces for S4, S5, and S6 is the main reason why these systems also presented several inconsistencies (Table I), although their UML models were created by reverse engineering the source code. Considering these results, we argue that *the trace service is scalable (RQ1) as it can be applied in large systems to generate traces*. There is, however, a limitation with the strategy of generating traces based on naming, requiring additional effort from engineers.

Figure 9 shows the results related to the runtime time (in microseconds μs) that our approach takes to create a CRE per system. The time stayed between 1 and less than 250 μs (0.25 milliseconds) per system. The median, for all systems, stayed below 100 μs. Considering the results from Table I, S3 was the system with the most CREs created (16,034). The worst time for evaluating a CRE in S3 was 76.06 μs. This means, that the runtime for evaluating the whole S3 system, in the worst case, would be around 1,219,546.04 μs or 1.21 seconds. Thus, we argue that the runtime results are satisfactory as, even for evaluating the whole system, the required runtime is acceptable. Furthermore, the runtime for generating all traces for S3 was 634.48 ms (0.63 seconds), averaging 0.3 ms per trace (Table II). *Hence, our approach can be applied within a reasonable amount of time (RQ2)*.

Figure 10 shows the results related to the average number of CREs and inconsistencies created by each change type applied per system (RQ3). Two change types (*Delete UML Class* and *Extract Operation*) are not displayed in the figure, as they led to zero CREs being created. Furthermore, the CREs created per change, on average, present little impact on the performance of the approach. This is supported by the average runtime for creating each CRE per change (Figure 11).

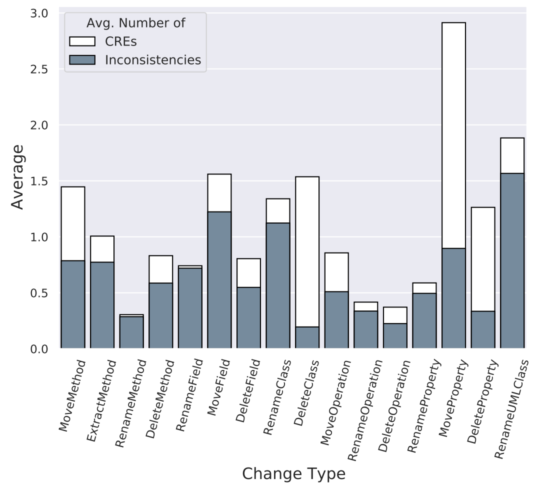


Fig. 10: CREs and inconsistencies per change type.

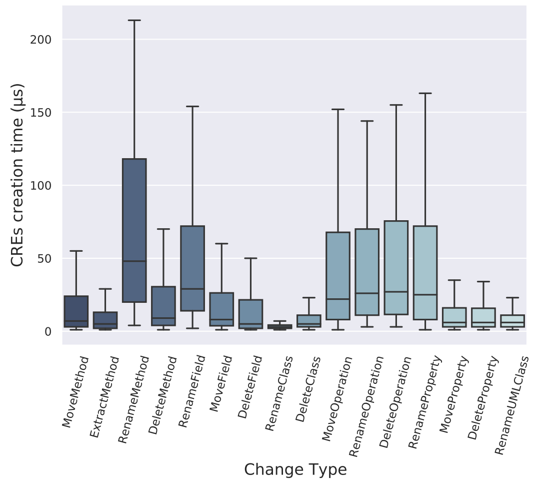


Fig. 11: Time required for creating CREs per change type.

Figure 11 shows that the runtime required to create each CRE per change stayed between 1 and 250 μs with the median staying below 50 μs for all change types. This implies that, in the worst case, considering the number of CREs created, the *Move Property* change creates 3 CREs on average and takes at most 250 μs per CRE, totalizing 750 μs, i.e, 0.75 ms. Hence, *our approach can react to changes performed in the artifacts, re-evaluating the CRs affected by the changes within a reasonable amount of time (RQ3).*

The results of the scalability evaluation demonstrate that our approach may be applied in other contexts, besides the industrial scenario from our partner. Furthermore, we could also identify limitations in the approach that must be addressed in the future.

Threats to the validity of the scalability evaluation: the selection of the systems used in the evaluation may be an internal threat. For mitigating this threat, we selected real systems from open-source projects. This resulted in a total of 61,855 of CREs created, aggregating to a total amount of 31.765 inconsistencies. These numbers show that the systems used are varied enough for supporting our findings. Another threat is the CRs used, as inconsistencies are found based on these rules. To mitigate this threat, we defined a varied set of rules with different sizes and expression types that evaluate different contexts [19]. We created these rules based on standard UML and Java consistency mechanisms [**?**], [4]. The results regarding the number of evaluations per system support our claim that this threat was mitigated as around 50% of the model elements were evaluated per system. This shows the variety of CRs, as they were not only checking the consistency of a small set of model elements per system.

An external threat is related to the generalization of our results to other domains. In this evaluation, we used Java source code and UML models, so we only have evidence to support the scalability of our approach with these types of artifacts. Our approach, however, was also applied in the scenario provided by the industry partner. In this scenario, a different set of CRs, artifacts, and traces are created. Furthermore, the trace and consistency checking services used for the Java/UML consistency checking are the same as applied to the Visio, IDL, and source code checking in the industrial scenario.

A conclusion threat is related to the changes considered for RQ3 data collection. To simulate changes that may be applied by real engineers, we define the type of changes based on the most used refactoring actions applied in source code [22]–[24]. Since these are changes created synthetically, they may still not reflect real changes created by engineers. However, the changes were created to observe how our approach reacts to them. So, although synthetic, the changes provided us with the data required to answer RQ3 (number of CREs created and runtime).

C. *Limitations and Future Work*

Challenges of extending the approach: there are limitations in connecting artifacts from different domains to a single server such as DesignSpace. Since different tools use different metamodels, e.g, UML classes have "operations" while Java classes have "methods", applying the same consistency checking service in different artifacts is challenging. Thus, once the connection is performed, the artifacts from the engineering tools have to be transformed using the unique representation depicted in Figure 3. This transformation is performed to guarantee that artifacts of different types, e.g., Java and UML, can be checked using the same consistency checking service. This transformation, however, may lead to important data from the original artifact being lost due to the limitation of the unique representation. Data loss is a common problem of cross-domain solutions that may lead to consistency checking not being performed correctly [2].

When designing the tool adapters, we have to carefully analyze the data required for the trace and consistency services to work properly. Furthermore, adapters of tools that allow the creation of different types of artifacts may not work for all artifacts created by these tools. The Visio tool adapter, for instance, was designed to send information about all shapes from Visio. However, the parser used to transform these shapes into a unique representation considers only UML models and their structure. This means that if another kind of shape is used, e.g., a BPMN shape, the parser will not be able to transform it into the unique representation used by DesignSpace. Other tools, such as Papyrus UML, rely on the use of the well-defined Eclipse Modeling Framework (EMF) [25] that is used for all artifacts created using that tool. In this case, we can parse any artifact of that tool to our unique representation, as long as we follow the EMF structure. Hence, for some tools, tool adapters have to be developed in a domain-specific manner not being applied in all artifacts of that tool. This is an open challenge in our research that will be further explored in future work. We do, however, have applied similar approaches for consistency checking in a variety of different artifacts, showing the potential extensibility of the unique representation [15], [26], [27].

Another challenge is related to the effort and costs of deploying the approach in different industrial scenarios. The varied number of artifacts, engineers, and tools being used in projects can have a direct impact on the cost/effort of applying the approach. Moreover, the larger the project, the more resources need to be available for the DesignSpace server to run the services. A possible strategy to address this challenge is to implement a decentralized solution as a microservice architecture. Evaluating the impact of having this type of architecture, however, remains a future work.

Consistency checking is only the first step: once inconsistencies are found in the artifacts, they need to be repaired. The approach presented in this work does not directly provide repairs to engineers. It does, however, provide guidance on how to repair inconsistencies by providing consistency feedback. With this feedback, engineers can reason about the inconsistencies that should be fixed. The generation of repairs based on CRs has been explored in the literature [16], [28]–[31]. These approaches, however, focus on generating repairs for single artifact types, e.g., UML. We plan to extend our current approach to provide repair generation based on these repair approaches. The generation of repairs should be customizable, similar to how the consistency checking is, by providing rules to filter out repairs not desired by the engineers. For instance, filtering our repairs that affect the Visio models and keeping only repairs that affect the IDL messages.

V. Related Work

In this section, we present work related to our research.

Unique representation: while designing the unique representation, we considered using EMF [25], [32], more specifically Ecore, as a basis for collaborative modeling. We decided to abandon this concept in favor of a typed, more lightweight, uniform artifact representation not tied to the Ecore/Eclipse framework. The main reason for not applying Ecore is to simplify the model representation, including only properties that would be needed for the communication between engineering tools, tool adapters, and the DesignSpace server. Moreover, the use of Ecore would lead our representation to contain properties and elements that are not necessarily useful for our services. Another reason for creating our own representation is the possibility to extend it based on the requirements related to the services running on the DesignSpace server for different types of artifacts and tools.

Consistency checking considering different types of artifacts: consistency checking has been applied in artifacts of different domains, with a focus on models [2], [5], [16], [33]–[36]. Furthermore, consistency checking can be performed with different strategies such as applying patterns, relying on ontology, or using constraints in form of CRs [2]. In our approach, we rely on the definition and execution of CRs. In this sense, CRs can be defined using different languages. Studies focusing on UML consistency checking were able to identify how 94 primary sources define CRs [**?**], [1], [2]. The results show that plain English was used in 29% of the approaches and OCL in 21%, the second most adopted. They also identified the type of diagram most used for consistency checking, being class diagram (67%) the most used, and sequence diagram (45%) the second most used. Although in this paper, the CRs used for the scalability evaluation focused on class diagrams, our approach can be applied in any UML diagram based on the context of the CRs defined.

The review by Torres et al. [2] identified and analyzed 80 different consistency checking tools. Their findings, however, showed that only 32 tools (40%) provided consistency checking for more than one type of model. Another limitation is related to the strategies applied to keep artifacts of different domains consistent. They report that most approaches are not mature, may cause data loss, and only work for specific tools. To address these limitations, our work has used trace links to connect artifacts of different tools into a single central consistency checking service. By applying our unique representation, we can transform artifacts into equivalent models and perform consistency checking on them.

Other systematic studies have discussed the importance and applicability of consistency checking in design models, such as UML [4]. The authors also discussed the lack of vertical consistency checking in most approaches. Vertical consistency checking is related to how our approach may perform bidirectional consistency checking, for instance, checking for traces between both UML and IDL, and then checking for their properties and fields. Most approaches would only check for them individually, e.g., only looking at the UML. As discussed previously, this may reduce the flexibility of the consistency checking and may not conform to the engineer's preferences. Hence, our approach addresses this problem by allowing bidirectional consistency checking across multiple tools. This strategy is novel compared to the majority of consistency checking approaches [5], [33]–[36]. There are, however, several approaches describing the consistency checking between source code and models, as described next.

Consistency checking between source code and models: several approaches address this topic using different strategies [37]–[47]. Kaliappan et al. [36] present an approach that guides engineers to maintain design models consistent, by checking their differences with the source code implemented in C#. They apply their approach in three types of UML diagrams, namely, class, use case, and sequence. Their approach, however, is limited to the traditional way that CRs are applied, without customization possibilities. Khelladi et al. [10], [35] apply the use of change propagation, evolving source code based on changes found in metamodels related to the code. Although their approach does not apply CRs, their goal is to keep source code and models consistent by maintaining them synchronized. Their approach is implemented into a prototype that supports EMF models and Java source code. In comparison to our work, as their approach does not use CRs, it provides fewer customization options regarding how inconsistencies should be found.

The lack of customization options is a limitation in approaches that rely on the automatic generation of code from models, or models from code [48]–[50]. Automated code generation usually requires additional configuration and steps that may be too intrusive to the scenario from our industry partner. Furthermore, applying code generation would impact the results related to *Issue 3*, since the flexibility of the approach would be reduced. This could lead the solution to disrupt the current development life-cycle, going against the requirements of the industry partner. Another aspect where automatic generation may struggle is dealing with ambiguity when generating artifacts. This ambiguity problem (*Issue 3*) needs to be addressed as it may lead to wrong code/models being generated. Allowing the customization of the consistency checking mechanism considering different tools and CRs is a novel aspect of our study in comparison to the aforementioned approaches.

VI. Conclusion

In this study, we present an approach designed to solve issues and requirements from an industrial scenario of a company that develops agricultural machines. Our solution was developed in collaboration with Flanders Make and provides consistency checking and trace services that can be applied to maintain consistency across engineering artifacts from different tools. We demonstrate how the approach addresses the issues in the industrial scenario, also presenting results of an empirical evaluation considering the scalability of the approach. Future directions and limitations are also discussed, such as extending the approach to different domains and providing repairs for the inconsistencies identified.

VII. Data Availability

The evaluation's artifacts as well as demo videos of the approach are available in an online repository [19].

Acknowledgements

The research reported in this paper has been partly funded by the Austrian Science Fund (FWF) (grant #P31989-N31 as well as grant #I4744-N) and by the Austrian COMET K1-Centre Pro2Future of the Austrian Research Promotion Agency (FFG) with funding from the Austrian ministries BMVIT and BMDW. The research reported on in this paper has been tested using a use case related to privately funded research done by Flanders Make vzw, the Flemish strategic research center for the manufacturing industry.

References

[1] D. Torre, M. Genero, Y. Labiche, and M. Elaasar, "How consistency is handled in model driven software engineering and uml: a survey of experts in academia and industry," Carleton University, Tech. Rep., 2018.

[2] W. Torres, M. G. Van den Brand, and A. Serebrenik, "A systematic literature review of cross-domain model consistency checking by model management tools," *Software and Systems Modeling*, pp. 1–20, 2020.

[3] D. Torre, M. Genero, Y. Labiche, and M. Elaasar, "How consistency is handled in model-driven software engineering and uml: an expert opinion survey," *Software Quality Journal*, pp. 1–54, 2022.

[4] F. J. Lucas, F. Molina, and A. Toval, "A systematic review of uml model consistency management," *Information and Software Technology*, vol. 51, no. 12, pp. 1631–1645, 2009, quality of UML Models.

[5] 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.

[6] A. Egyed, "Instant Consistency Checking for the UML," in *28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 381–390.

[7] ——, "Automatically detecting and tracking inconsistencies in software design models," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 188–204, 2011.

[8] A. Reder and A. Egyed, "Incremental Consistency Checking for Complex Design Rules and Larger Model Changes." in *MoDELS*, ser. Lecture Notes in Computer Science, R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, Eds., vol. 7590. Springer, 2012, pp. 202–218.

[9] M. Riedl-Ehrenleitner, A. Demuth, and A. Egyed, "Towards model-and-code consistency checking," in *Annual Computer Software and Applications Conference*, 2014, pp. 85–90.

[10] D. E. Khelladi, B. Combemale, M. Acher, and O. Barais, "On the power of abstraction: A model-driven co-evolution approach of software code," in *International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 85–88.

[11] M. A. Tröls, L. Marchezan, A. Mashkoor, and A. Egyed, "Instant and global consistency checking during collaborative engineering," *Software and Systems Modeling*, pp. 1–27, 2022.

[12] R. Jongeling, "How to Live with Inconsistencies in Industrial Model-Based Development Practice," in *International Conference on Model Driven Engineering Languages and Systems Companion*, 2019, pp. 642–647.

[13] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[14] OMG, "IDL Specification," https://www.omg.org/spec/IDL/, 2018.

[15] L. Marchezan, W. K. G. Assunção, G. Michelon, E. Herac, and A. Egyed, "Code smell analysis in cloned java variants: The apo-games case study," in *International Systems and Software Product Line Conference - Volume A*, ser. SPLC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 250–254.

[16] L. Marchezan, R. Kretschmer, W. K. Assunção, A. Reder, and A. Egyed, "Generating repairs for inconsistent models," *Software and Systems Modeling*, pp. 1–33, 2022.

[17] OMG, "OCL Specification," http://www.omg.org/spec/OCL/, 2014.

[18] A. Demuth, M. Riedl-Ehrenleitner, A. Nöhrer, P. Hehenberger, K. Zeman, and A. Egyed, "Designspace: An infrastructure for multi-user/multi-tool engineering," in *Symposium on Applied Computing*, ser. SAC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1486–1491.

[19] L. Marchezan, W. K. G. Assunção, E. Herac, F. Keplinger, A. Egyed, and C. Lauwerys, "Fulfilling Industrial Needs for Consistency Among Engineering Artifacts - Evaluation Data," Oct. 2022. [Online]. Available: https://zenodo.org/record/7197600#.ZFuhw3ZByUk

[20] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[21] A. AbuHassan, M. Alshayeb, and L. Ghouti, "Software smell detection techniques: A systematic literature review," *Journal of Software: Evolution and Process*, vol. 33, no. 3, p. e2320, 2021.

[22] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[23] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? Confessions of GitHub contributors," in *International Symposium on Foundations of Software Engineering*, 2016, pp. 858–870.

[24] M. Paixão, A. Uchôa, A. C. Bibiano, D. Oliveira, A. Garcia, J. Krinke, and E. Arvonio, *Behind the Intents: An In-Depth Empirical Study on Software Refactoring in Modern Code Review*. ACM, 2020, p. 125–136.

[25] M. Koegel and J. Helming, "EMFStore: a model repository for EMF models," in *International Conference on Software Engineering-Volume 2*. ACM, 2010, pp. 307–308.

[26] C. Mayr-Dorn, R. Kretschmer, A. Egyed, R. Heradio, and D. Fernandez-Amoros, "Inconsistency-tolerating guidance for software engineering processes," in *International Conference on Software Engineering: New Ideas and Emerging Results*, 2021, pp. 6–10.

[27] A. Demuth, R. Kretschmer, A. Egyed, and D. Maes, "Introducing traceability and consistency checking for change impact analysis across engineering tools in an automation solution company: An experience report," in *International Conference on Software Maintenance and Evolution*, 2016, pp. 529–538.

[28] R. Kretschmer, D. E. Khelladi, and A. Egyed, "Transforming abstract to concrete repairs with a generative approach of repair values," *Journal of Systems and Software*, vol. 175, p. 110889, 2021.

[29] L. Marchezan, W. K. G. Assuncao, R. Kretschmer, and A. Egyed, "Change-oriented repair propagation," in *International Conference on Software and System Processes and International Conference on Global Software Engineering*, ser. ICSSP'22. New York, NY, USA: Association for Computing Machinery, 2022, p. 82–92.

[30] A. Barriga, A. Rutle, and R. Heldal, "Personalized and automatic model repairing using reinforcement learning," in *International Conference on Model Driven Engineering Languages and Systems Companion*, 2019, pp. 175–181.

[31] M. Ohrndorf, C. Pietsch, U. Kelter, and T. Kehrer, "ReVision: A Tool for History-Based Model Repair Recommendations," in *40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 105–108.

[32] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[33] M. A. Tröls, A. Mashkoor, and A. Egyed, "Timestamp-based consistency checking of collaboratively developed engineering artifacts," in *International Conference on Software and System Processes*, 2021.

[34] C. Nentwich, W. Emmerich, and A. Finkelsteiin, "Consistency management with repair actions," in *International Conference on Software Engineering*, ser. ICSE '03. IEEE, 2003, pp. 455–464.

[35] D. E. Khelladi, B. Combemale, M. Acher, O. Barais, and J.-M. Jézéquel, "Co-evolving code with evolving metamodels," in *International Conference on Software Engineering*. ACM, 2020, p. 1496–1508.

[36] V. Kaliappan and N. M. Ali, "Improving consistency of uml diagrams and its implementation using reverse engineering approach," *Bulletin of Electrical Engineering and Informatics*, vol. 7, no. 4, pp. 665–672, 2018.

[37] J. Adersberger and M. Philippsen, "Reflexml: Uml-based architecture-to-code traceability and consistency checking," in *European Conference on Software Architecture*. Springer, 2011, pp. 344–359.

[38] Z. Diskin, Y. Xiong, and K. Czarnecki, "Specifying overlaps of heterogeneous models for global consistency checking," in *International Workshop on Model-Driven Interoperability*, ser. MDI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 42–51.

[39] J. Lenhard, M. Blom, and S. Herold, "Exploring the suitability of source code metrics for indicating architectural inconsistencies," *Software Quality Journal*, vol. 27, no. 1, pp. 241–274, 2019.

[40] G. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: bridging the gap between design and implementation," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 364–380, 2001.

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

[42] D. Cassou, E. Balland, C. Consel, and J. Lawall, "Leveraging software architectures to guide and verify the development of sense/compute/control applications," in *International Conference on Software Engineering*, ser. ICSE '11. ACM, 2011, p. 431–440.

[43] H. M. Chavez, W. Shen, R. B. France, B. A. Mechling, and G. Li, "An approach to checking consistency between uml class model and its java implementation," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 322–344, 2016.

[44] Y. Zheng and R. N. Taylor, "Enhancing architecture-implementation conformance with change management and support for behavioral mapping," in *International Conference on Software Engineering*, 2012, pp. 628–638.

[45] R. Jongeling, J. Fredriksson, F. Ciccozzi, A. Cicchetti, and J. Carlson, "Towards consistency checking between a system model and its implementation," in *Systems Modelling and Management*, Ö. Babur, J. Denil, and B. Vogel-Heuser, Eds. Cham: Springer International Publishing, 2020, pp. 30–39.

[46] R. Jongeling, A. Cicchetti, F. Ciccozzi, and J. Carlson, *Towards Boosting the OpenMBEE Platform with Model-Code Consistency*. New York, NY, USA: Association for Computing Machinery, 2020.

[47] M. Zaheri, M. Famelis, and E. Syriani, "Towards checking consistency-breaking updates between models and generated artifacts," in *International Conference on Model Driven Engineering Languages and Systems Companion*, 2021, pp. 400–409.

[48] A. D. Durai, M. Ganesh, R. M. Mathew, and D. K. Anguraj, "A novel approach with an extensive case study and experiment for automatic code generation from the xmi schema of uml models," *The Journal of Supercomputing*, vol. 78, no. 6, pp. 7677–7699, 2022.

[49] W. Harrison, C. Barton, and M. Raghavachari, "Mapping uml designs to java," *SIGPLAN Not.*, vol. 35, no. 10, p. 178–187, oct 2000.

[50] U. Sabir, F. Azam, S. U. Haq, M. W. Anwar, W. H. Butt, and A. Amjad, "A model driven reverse engineering framework for generating high level uml models from java source code," *IEEE Access*, vol. 7, pp. 158 931–158 950, 2019.