Live and Global Consistency Checking in a Collaborative Engineering Environment[*]

Michael Alexander Tröls
Johannes Kepler University
Linz, Austria
michael.troels@jku.at

Atif Mashkoor
Johannes Kepler University
Linz, Austria
atif.mashkoor@jku.at

Alexander Egyed
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

ABSTRACT

During software and systems engineering, engineers have to rely on different engineering tools in order to capture different kinds of artifacts, such as requirement specifications, design models or code. Even though the artifacts that engineers capture with these tools are interdependent, the tools have limited abilities to detect inconsistencies among them. Today no approach exists that is able to provide live inconsistency feedback of engineering artifacts – captured and maintained in different engineering tools – without disrupting the engineers' workflow. The work presented in this paper introduces a novel approach for live, multi-tool, consistency checking where engineers continue to use their respective tools and receive inconsistency feedback across their tools' artifacts in a live manner. The approach uses a cloud-based engineering platform to replicate the tool's artifacts and to detect inconsistencies there. Within the cloud, engineers may link these artifacts and define cross-tool consistency rules. The approach was validated through an empirical study and two industrial case studies to demonstrate usefulness, correctness and scalability.

ACM Reference Format:
Michael Alexander Tröls, Atif Mashkoor, and Alexander Egyed. 2019. Live and Global Consistency Checking in a Collaborative Engineering Environment. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3297280.3297454

1 INTRODUCTION

Contemporary software and systems engineering is a process that involves a wide range of tools, bringing together knowledge of many engineers. In doing so, engineers create a wide variety of engineering artifacts, ranging from requirements to design model specifications and code. With increasing number of tools and engineers involved, the number of interdependent engineering artifacts grows non-linearly. Maintaining consistency among artifacts of these separate tools is a challenging task [9]. We speak of global consistency checking.

Most approaches towards consistency checking focus on individual tools or documents. Unless engineering artifacts are merged and combined into a single tool or document, the existing work is not capable of detecting inconsistencies among artifacts of different tools, e.g., between UML models and source code or hardware designs. Unfortunately, merging artifacts is time consuming and disruptive. Thus engineering artifacts across tools are rarely checked for global consistency and get out-of-sync. This means that documentation does not correspond to implementations, the design is no longer in line with the requirements specifications, or code does not match its model specifications.

Global consistency checking, which takes into account the sum of all artifacts captured in all tools used in a project, has key differences compared to state-of-the-art tool-centric or document-centric consistency checking technologies. It navigates across artifacts of different tools and handles conflicting artifact changes due to the concurrent use of different tools by engineers on the basis of globally defined consistency rules.

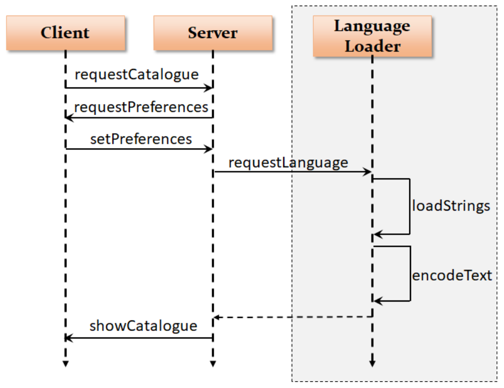
This paper thus presents an approach for live, global consistency checking. It utilizes an engineering cloud which mirrors relevant parts of the engineering artifacts within engineering tools. It centralizes consistency checking and reduces the overall memory and computation time. Tools merely need to synchronize engineering artifacts to the cloud as engineers change them and receive inconsistency feedback live. The approach also lets engineers define links among artifacts in the cloud so that the consistency checker may navigate them. Freely definable link types let engineers connect artifacts of different tools. Consistency rules, consistency checking results and links are stored in the cloud using the same uniform representation that artifacts are stored in. A consistency checker can thus look for inconsistencies beyond the traditional boundaries of a single engineering tool. As a result consistency checking becomes global and is no longer bound to a single tool or limited to specific kinds of artifacts. Since all consistency checking is done in the cloud, the engineers can continue working with their tools without interruption.

Our approach was implemented as a cloud service utilizing the DesignSpace [5] infrastructure. In the cloud, merely one consistency checker implementation is needed rather than many implementations for each and every tool. Given that the cloud computes consistency feedback once for all artifacts, it is in fact more efficient than tool-centric solutions where each tool would have to compute the consistency feedback separately (to the extent they are capable of doing that). Empirical evidence demonstrates scalability

[*]This work is partially supported by the Austrian Science Fund (FWF): FWF P 25513-N15 as well as P 31989-N31, the JKU Linz Institute of Technology (LIT), the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy and the Province of Upper Austria in the frame of the COMET center SCCH.

SAC '19, April 8–12, 2019, Limassol, Cyprus
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297454

Figure 1: Sequence diagram of the language selection process



and usability across a wide set of tools. Two industrial case studies provide further evidence of usefulness.

The rest of this paper is organized as following: Section 2 presents a running example to illustrate the problems that are tackled. The main part of the paper – Section 3 and 4 – provides an overview of the platform used to realize our approach, as well as a presentation of the approach's architecture and functionality. We validate our approach with multiple case studies in Section 5. Section 6 discusses related work. The paper is finally concluded in section 7, giving an outlook on future work.

2 PROBLEM ILLUSTRATION

To outline the problems arising with regular consistency checking and to illustrate our approach, this section discusses a running example. Consider the following situation: A company would like to implement the graphical user interface (GUI) of a video streaming web service. When a user logs into the movie streaming service, the server displays a catalog of videos to choose from. For our example, three different engineers are working on three different workstations, creating three different types of engineering artifacts.

2.1 Initial State

In the spirit of an iterative development, a first implementation has already been produced and presented – based on a selected, initial set of requirements. UML models exist that describe basic structure and scenarios. These artifacts were created by different engineers over the course of a longer time frame. For simplicity, let us assume one engineer for each engineering tool:

- John is specifying functional requirements,
- Alex is creating UML models in the form of class and sequence diagrams, and
- Alice is implementing the code based on UML models in a separate Java programming IDE.

The goal is to have all requirements realized in the UML design and consistently implemented in the code. As all engineers work concurrently, parts of the requirements, model, and code are stable, while other parts are still subject to change.

2.2 Subsequent Changes Made by Engineers

The following describes a simple change scenario where the aforementioned engineers adapt their respective artifacts using their corresponding tools. The first implementation reveals shortcomings in the connection process to the server. When a user logs in for the first time, the server should request a language for displaying the catalog of available movies. The server then sends all textual descriptions accordingly. John records this request as an additional requirement. The adjustment of the requirement specification has direct impact on the UML design of the server. John's change makes it necessary for Alex to adapt his UML design. The new process is illustrated in Figure 1. The framed parts of the sequence diagram are an addition to the existing design; however, the sequence diagram is but one scenario that describes the client/server communication protocoll. As their company is progressive and aware of benefits of maintaining traceability between requirements and design, Alex needs to ensure that the changed/added parts of the design are mapped back to the requirements that motivated them.

Naturally, these design changes imply further changes to the implementation. Alice needs to adapt her implementation to make it match John's new requirement. At the same time, she must be mindful of Alex's changes in the sequence diagram. In doing so, the implementation must ensure simple goals, such as the naming of Java methods according to UML messages in the sequence diagram but also more complex goals, such as implementing the client/server communication protocols, which is described over a number of sequence diagrams of which Figure 1 is merely a part.

It is easy to see that changes to either requirements, design, or code can get out-of-sync since each engineer has his/her own limited views of these artifacts. Obviously, Alice is able to see Alex's UML design but how can she be certain that she implemented his design correctly and completely – while both design and code are subject to continuing change. Neither the programming tool nor the design tool will be of much use to her in ensuring this consistency.

2.3 Problem Statement

If changes are not carried through correctly, requirements, design and code become out-of-sync. In short, they may become inconsistent. These inconsistencies are particularly hard to spot if they relate to artifacts within different engineering tools – whether or not they are created by the same engineers. Consistency checking should help engineers in detecting and keeping track of these inconsistencies. Doing so should not be disruptive to the engineers' work. Consistency checking should let Alice know that her client's *setPreference* method is out-of-sync because it is missing a *requestLanguage* call or message which does not yet exist.

Due to the limited perspectives of engineering tools, current state-of-the-art tool-centric consistency checking mechanisms are not able to detect these inconsistencies. That is, the UML modeling tool representing the sequence diagram would be aware of Alex's addition of the messages and could quickly identify inconsistencies with other parts of the UML model. However, it could not depict how Alex's changes relate to the source code. A document-centric consistency checker would be able to provide more comprehensive consistency feedback but its use would be disruptive and certainly not live and incremental in providing up-to-date inconsistency feedback as engineers make changes available. Inconsistencies across tools may thus go undetected during software and systems engineering. This has several negative implications: 1) the end product may not correctly implement the new requirement, 2) the code may be harder to maintain because the UML model no longer documents the code correctly, and 3) there might be higher costs of fixing resulting problems [8].

The goal of this work is thus to enable live and global consistency checking across all artifacts of all tools, to prevent the problems mentioned above. We intend to achieve this in a way that ensures a global, uniform view on artifacts, in which said artifacts can be associated with each other, independent of their origin. Subsequently, the consistency rules must be writable in a manner that is oblivious to the syntactic and semantic differences among the engineering artifacts (i.e., textual source code versus graphical model elements). Furthermore, the engineer's tools should be able to integrate inconsistency feedback seamlessly.

3 INFRASTRUCTURE

This section discusses the infrastructure of our global consistency checking approach. There are three major issues to solve in order to enable global consistency checking. The first issue is the physical separation of engineering artifacts, which are usually created by engineers using different tools and located on different machines. This issue is discussed in Section 3.2. The second issue is about the incompatible representation languages that different tools use – while code adheres to a certain programming language, UML may be represented in the XML format. This issue is discussed in Section 3.2.1. The third issue is about the logical separation that remains even if the artifacts are represented in the same language. This issue is discussed in Section 3.3.

3.1 Overview

To overcome these issue, our approach uses the DesignSpace [5] infrastructure, which provides a central, versioned storage space for engineering artifacts. It provides a fine-grained, uniform, model / metamodel-style artifact representation. The DesignSpace infrastructure not only acts as a storage space for engineering artifacts but it also holds the links that connect engineering artifacts from different tools. By linking artifacts through a cloud environment both the problem of physical and logical separation of engineering information can be overcome. A pre-requisite for linking artifacts is a syntactic common-ground onto which links can build. This is achieved through a common uniform artifact representation into which artifacts are translated. As a result, it is possible to, e.g., link Java classes with the UML model elements that implement them. Once links are known, engineers may formulate consistency rules that define correctness conditions for artifacts from different tools. These rules could be written by the same engineers using the tools; though, it is more likely that they are predefined by domain engineers. Once linked in their common representation, engineering artifacts can furthermore be analyzed and manipulated by services such as a consistency checker.

An illustration for our global consistency checking mechanism interlinked with the DesignSpace's core concepts can be seen in

Figure 2: Basic work principle of Global Consistency Checking within the cloud environment

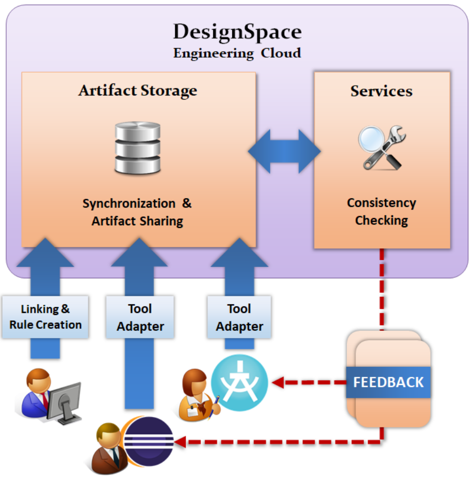


Figure 2. Engineers are working with their respective tools: Eclipse for Java and IBM's Rational Software Architect (RSA[1]) for UML models. There are adapters for these tools in order to automate the mapping of engineering artifacts to the uniform data representation. Furthermore, adapters automate the synchronization with the cloud environment. Links and consistency rules can be created and modified with separate tools. In our case the linking tool directly modifies the artifacts synchronized with the cloud. With the links set up our global consistency checking approach then validates the defined consistency rules. Inconsistencies are shared with the engineers in the form of network messages from the cloud, which can be directly fed into the engineers tool through the tool adapter.

3.2 Cloud Environment

Global consistency checking among artifacts of different tools requires live, continuous access to these artifacts. Synchronizing engineering artifacts with a cloud environment enables such access. In this work, the DesignSpace was chosen as the underlying platform. The artifact storage in the DesignSpace provides a shared space that has the following characteristics:

- **Uniform Data Representation**: To provide a common format for distinct kinds of engineering artifacts, such as source code (which is usually textual), or models (which is usually graphical), the DesignSpace stores data in a uniform representation.
- **Typing**: To provide a well-defined language (metamodel) and language constructs for data, the DesignSpace offers custom artifact types that can be instantiated dynamically.
- **Incremental Live Synchronization**: Tools may continuously synchronize changes made by the engineers, ensuring that the global consistency checker is always up-to-date.
- **Conflict Handling**: Mutually changed artifacts may stand in conflict with each other. The DesignSpace provides conflict handling strategies for such situations.

[1]IBM RSA: https://www.ibm.com/developerworks/downloads/r/architect/index.html (last accessed: 06.12.2018)

The following sections will discuss these characteristics and their importance to global consistency checking in more detail.

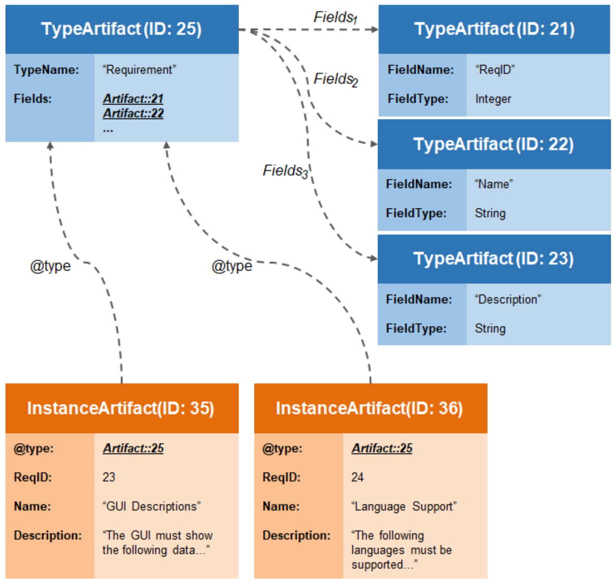
3.2.1 Uniform Data Representation. In the DesignSpace engineering artifacts (respectively parts thereof) are translated into a uniform data representation. The data structure itself is a generic mapping between keys and values, representing arbitrary properties of engineering artifacts. The keys are simple strings, while the values are primitive data types or references to other artifacts. For example, if we wish to translate a Java class into the uniform data representation we could use field names as keys and their default values as values. Something important to note is that such a mapping can as well represent links between artifacts, since artifacts in the uniform data representation carry a unique ID to be identifiable. This ID is then simply referenced in the property value. Any structured data that can be represented in such a manner can, furthermore, be uploaded to the DesignSpace. It should be noted that in no way our uniform data representation is solely laid out for software engineering artifacts. It can as well be used for, e.g., electrical engineering artifacts. Hence, we do not use notation or terminology from a specific field. To simplify conversion into the artifact format the DesignSpace's API provides appropriate methods. All created artifacts, respectively their mappings, must correspond to a defined type. This means that artifacts have a fixed base structure.

3.2.2 Typing. In engineering tools, artifacts are typically defined through a metamodel. This metamodel defines the structure of an engineering artifact within its respective engineering tool and often semantics as well. For example, a Java code contains Java classes and each class may have an arbitrary number of Java methods. This conformity to a metamodel is also required by our consistency checker, which uses the metamodel to custom-tailor consistency rules for specific kinds of artifacts. Consequently, artifacts synchronized with the DesignSpace are typed. Each type may have a set of properties, which can again refer to other types or primitive data types - respectively collections thereof. For example, in Java there exists a type called *Java Class* with properties such as name (of type String) or methods (of collection references to type *Java Method*). The type definitions can be created through a seperate tool or during the first initialization of a corresponding tool adapter (e.g., an adapter for a programming IDE would first initialize types for the programming languages it supports. The DesignSpace API used in tool adapters provides methods to create types in the cloud environment).

3.2.3 Incremental Live Synchronization. Every tool synchronizing its engineering artifacts is connected to the cloud via a tool adapter. The tool adapter – which is ideally implemented as a plugin in the respective tool – observes the tool's internal data structure. When a change happens, it is incrementally synchronized with the DesignSpace.

For our running example John's tool adapter creates and instantiates types corresponding to the requirements he wishes to synchronize with the DesignSpace. The artifact type for requirements defines the necessary properties, e.g., requirement name, requirement ID and description. These types are then instantiated by the requirements tool adapter, whenever a new requirement is added. A part of this example is illustrated in Figure 3, where two instantiations of a requirement artifact type are created.

Figure 3: Exemplary types and instantiations for requirements



Types must be uploaded only once. Instances of these types are uploaded as often as they are found in their respective tools. As such, there is one requirement type but as many requirement instances as engineers create. Once artifacts from the tool have been uploaded to the DesignSpace, changes made on these artifacts are uploaded incrementally. An incremental change of an artifact triggers our global consistency mechanism, in order to keep consistency information of the respective artifact up-to-date. The instant nature of the synchronization process between the tool and cloud environment, furthermore, enables live checking of artifacts.

3.2.4 Conflict Handling. Engineers mostly do not modify the same artifacts concurrently. In case mutually overlapping changes exist, the DesignSpace applies user preferences specifically set for such situations. Users can either overwrite or ignore changes from other users. These options can be set for each user individually. Users could, for example, overwrite their own changes with information from one user, yet ignore conflicting artifacts from another one, keeping their own changes intact. In the latter case, the user's data would only be complemented by changes that are not in conflict.

3.3 Linking Engineering Artifacts

With data synchronized in a cloud environment, the next issue to overcome is the logical separation of engineering artifacts. Even after engineering artifacts are uploaded to the DesignSpace, they remain semantically separate. The artifacts co-exist but they are not integrated. Indeed, no such integration is attempted or even needed by the DesignSpace. Instead, our approach utilizes links between different artifacts in order to navigate beyond the boundaries of a single engineering tool. Such links can be captured in various forms. The current state-of-the-art relies on trace matrices, respectively defined navigable links between engineering artifacts. Since the data format is uniform and all artifacts can be linked with each other, links can be set up between arbitrary artifacts of different engineering fields, e.g., a code artifact can store links pointing to a UML diagram representing that code, which in turn can then point towards a requirements specification, describing the feature captured in the UML diagram. It should be noted that links can also be established between the properties of artifacts. As a result the paramenters of an operation in, e.g., a UML class diagram, can point to their counterparts in the respective code artifact. In Figure 4, the link "Affects" respectively "AffectedBy" is defined within the type of artifacts and established at instantiation, between requirement, UML sequence and code artifact. Links can be defined on the artifacts themselves or - with a stricter typing discipline in mind - as separate artifacts with their own types. Either way, link artifacts have to adhere to a type definition that can be added onto existing artifact types. The DesignSpace offers a tool for the creation of link types as well as their instantiation. The link references two type artifacts, one as a source, the other as a target of the link. Once instantiated these link properties must correspond to instances of the referenced types. Engineers can use this linking tool to create arbitrary links. Principally, engineers define links manually through the linking tool. We refer to this as capturing explicit links.

Concerning our running example it may be interesting for John, which of his requirements affect which parts of Alex's class diagrams. Likewise, Alex may be interested in the Java classes which are affected by his UML classes. In the type definition of their engineering artifacts they may even introduce a specific property to save such references, as given in Figure 4. A responsible engineer could then fill these references manually. Naturally, one could create a DesignSpace service software analyzing and manipulating engineering artifacts to automatically establish links between them [13]. However, this is beyond the scope of the approach presented in this paper.

4 GLOBAL CONSISTENCY CHECKING

With artifacts stored and linked in the cloud environment, we can proceed with checking their consistency. When engineers change artifacts through their tools, the consistency checker is notified. It then automatically identifies and validates the changes according to corresponding consistency rules. These consistency rules are provided by the users, e.g., a domain expert. The results of the consistency checker are then returned to the engineers respectively their tools. The tools may then provide specialized feedback to the engineers who benefit from the live consistency checking on their changes. Consistency checking is done in the cloud and engineers do not have to interrupt their work. Our implementation is similar to the incremental consistency checker discussed in [6]. The key differences are its integration with the unified data structure, its ability to change arbitrary artifacts, its ability to handle new types and consistency rules during runtime, and its ability to work seamlessly with tool artifacts and links.

4.1 Data Structure

Our consistency checker has access to all the information stored in the DesignSpace infrastructure. All of its internal data is stored in the cloud as well. Two artifact types build the core of its data structure:

4.1.1 Rule Definition Artifacts. Rule definition artifacts define consistency rules. A rule consists of a context type and a condition.

- **Context Type**: The context type of a rule definition is the type of an artifact to which the rule applies. The rule is then evaluated for every instance of this type. If, for example, a user wishes to write a consistency rule for Java classes, the context type of the rule definition would point towards the type Java class artifacts.
- **Condition**: A condition is an OCL-like string, defining a set of requirements that have to be met by the type instance - typically a tool artifact. During a rule evaluation the condition is read and checked. It should be noted, that a condition can incorporate links between types. This way a rule evaluation can navigate to and compare values from different artifact types. The context type then acts as the starting point for the navigation.

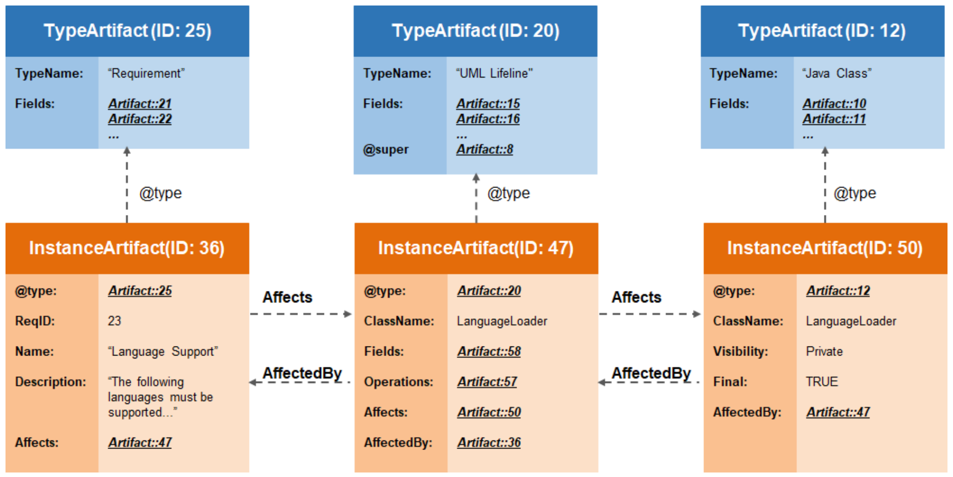
4.1.2 Rule Evaluation Artifacts. A rule evaluation is the evaluation of a consistency rule definition on an artifact instance. Recall that the consistency rule definition defines a context type and the rule must consequently hold for every instance of that context. A single rule evaluation is one such check of the rule definition on an instance. Thus, there are as many rule evaluation artifacts as there are instances of the rule definition context types. The rule evaluation artifact also contains a scope, which is a list of all artifacts that affect its consistency. If an artifact changes then all rule evaluations that contain this artifact in its scope must re-evaluate. The rule evaluation thus contains these fields:

- **Context Instance**: A context instance is the starting point in a rule evaluation. Any artifact can be the instance of a rule evaluation, but the instance type must correspond to the type defined in the context of the rule definition. If, for example, the context type of a rule definition refers to Java class then the context instances of the corresponding rule evaluations must refer to instances of *Java Class*.
- **Scope**: Rule evaluations store a change impact scope. The scope of a rule evaluation stores all artifact instances that affect it. This list is saved for re-evaluations.
- **Result**: A consistency check either indicates an inconsistency (condition evaluates to false) or consistency. This result is stored.

4.2 Evaluation Process

Whenever an artifact is created, modified or deleted, the tool adapters synchronize the change with the DesignSpace. The DesignSpace notifies the consistency checker, which then processes this change. In most cases, a change causes a (re-)evaluation of a rule definition artifact's condition. In the said evaluation the consistency checker follows the artifact structure described in the condition, navigating from a context instance (saved in a rule evaluation artifact) to one or many literals. These literals are compared according to the condition, resulting in a boolean value, which indicates the consistency state of the respective rule evaluation artifact. This has to be done for all artifacts affected by a change. The types of such changes

Figure 4: Three instantiations of three types are linked together via the artifact properties "Affects" respectively "AffectedBy", alternatively links can be instantiated in the form of typed instances



(Created, Modified or Deleted), will be discussed individually in the following.

4.2.1 *Artifact Creation (see Algorithm 1).*

When an artifact is created then the notification to the consistency checker contains the new artifact (its ID). Two scenarios have to be considered:

- A new rule definition was created.
- A tool artifact was created whose type corresponds to existing rule definition's context types.

Whenever a new rule definition is created, the consistency checker has to retrieve all artifact instances that correspond to the defined context type. For each of these artifact instances, the consistency checker creates a rule evaluation, with the artifact instance as its context element. These rule evaluations are then immediately evaluated and feedback is sent to the engineers. During the first evaluation, the OCL statement is fully parsed and the scope is built, i.e., a reference to each artifact respectively property that is affected during the evaluation is stored in the rule evaluation artifact. Later, when an artifact is changed, the consistency checker only re-evaluates the rules, whose scope contain the said artifact.

If an artifact instance is created whose type corresponds to an existing rule definition's context type then a rule evaluation must be created and evaluated for this artifact instance.

4.2.2 *Artifact Modification (see Algorithm 2).*

If an artifact is modified, the cloud sends a notification to the consistency checking service. As a modification is always about a change to an artifact property, this notification contains the changed artifact instance (its ID) and property name. If the artifact respectively property is listed in the scopes of rule evaluations then these rule evaluations are re-evaluated.

4.2.3 *Artifact Deletion (see Algorithm 3).*

If an artifact is deleted then the notification to the consistency checker contains the deleted artifact. Two scenarios have to be considered here:

- An existing rule definition was deleted.
- A tool artifact was deleted which was a context element for a rule evaluation

Algorithm 1: Artifact Creation

```
Data:
A = Added artifact
AST = Set of all Artifacts
RD = Set of all Rule Definitions
RDtype = ID of Rule Definition Type
begin
    if $A.type == RDtype$ then
        $C \longleftarrow A.contextType$
        for $a \in AST$ do
            if $a.type == C$ then
                re = createRuleEvaluation
                 re.setContextInstance(a)
                 re.setRuleDefinition(A)
                 evaluate(re)
            end
        end
    else
        for $rd \in RD$ do
            $CT \longleftarrow RD.contextType$
            if $A.type == CT$ then
                re = createRuleEvaluation
                 re.setContextInstance(A)
                 re.setRuleDefinition(RD)
                 evaluate(re)
            end
        end
    end
end
```

If a rule definition was deleted, all corresponding rule evaluations have to be deleted as well. If the context instance of a rule evaluation was deleted, all rule evaluations carrying this context instance have to be deleted. If a property of an artifact is deleted the consistency checking service has to scan the corresponding scopes and set rule evaluation's result to invalid. In this case no rule evaluation is deleted, because the property might be replaced later.

Algorithm 2: Artifact Modification

```
Data:
ID = ID of modified artifact
RE = Set of Rule Evaluation Artifacts
begin
    for $re \in RE$ do
        $S \longleftarrow re.scope$
        if $ID \in S$ then
            evaluate(re)
        end
    end
end
```

Algorithm 3: Artifact Deletion

```
Data:
A = Deleted Artifact
RDtype = ID of Rule Definition Type
RE = Set of Rule Evaluation Artifacts
begin
    if $A.type == RDtype$ then
        for $re \in RE$ do
            $DEF \longleftarrow re.definition$
            if $DEF == A$ then
                delete(re)
            end
        end
    else
        for $re \in RE$ do
            $CI \longleftarrow re.contextInstance$
            if $A == CI$ then
                delete(re)
            end
        end
    end
end
```

4.3 Inconsistency Feedback

Once a consistency rule has been evaluated, the corresponding result is stored. The result is also sent to tool adapters in the form of a notification. The notification also contains the related rule evaluation artifact, from which further inconsistency information can be retrieved. The interpretation of inconsistency feedback is fully up to the user respectively the implementation of the tool adapter. Depending on the integration of the adapter into the tool, e.g., in the form of a plugin, the inconsistency feedback could be visualized through warning messages within the tool. For example, UML model elements can be marked in a tool's graphical modeling editor.

4.4 Example

Coming back to our example from Section 2, let us define the following two consistency rules:

- CR1: Operations in sequence diagrams must be defined as methods in the corresponding code implementation of a sequence.
- CR2: Functional requirements must be represented within the UML design.

In our approach these consistency rules would be formalized through a Rule Definition Artifact. To create such an artifact, the users are presented with a GUI, in which they can enter an OCL expression for a certain artifact type. Similar to our Rule Definition Artifact an OCL rule consists of a context and a condition. The context refers to the object (in our case artifact) type for which the condition of the rule must hold.

Consider the artifact structure depicted in Figure 4: To describe consistency rule CR1, we define the UML lifeline type as the context (TypeArtifact(ID: 20)). The corresponding condition then compares two sets, more precisely, the names of outgoing messages from the lifeline to the names of methods in the affected Java class. Note that in Section 3.2.2, we mentioned that artifact properties could also hold collections of values. In this case both message and method names could be held in a string collection. For all message names there must be a matching method name. The fully realized consistency rule can be seen in Listing 1.

This consistency rule seems quite simple in principle but it would not work without links that connect Java method names and UML messages. In this example, we added explicit links. This is illustrated in Figure 4 where a requirement artifact is linked with a UML sequence artifact, which in turn is linked to an artifact representing a piece of code implementing the said sequence diagram (via "Affects"). The artifacts are also linked backwards (via "AffectedBy"). Consistency rule CR1 is thus able to utilize some of these links. It is important to observe that we utilize the standard OCL language for defining these consistency rules, because within the DesignSpace all artifacts including links are properly typed.

```
context: UML Lifeline
self.MessagesOut->
forall(m | self.Affects.Fields->exists(f |
f.name = m.name))
```

Listing 1: Operations in sequence diagrams must be defined as methods in the corresponding code implementation of a sequence (CR1).

```
context: Requirement
self.Affects->notEmpty() implies
self.@type.Fields->exists(f |
f.FieldName = 'Affects' and
f.FieldType = self.Affects.@type.@super )
```

Listing 2: Requirements must be represented within the UML design (CR2).

Consistency Rule CR2 is handled in a similar fashion. To describe consistency rule CR2, we define the Requirement Type as the context (TypeArtifact(ID: 25)). The corresponding condition utilizes the "Affects" link, making sure that it points towards an artifact. The condition could also be extended to ensure that the target of the "Affects" link corresponds to the right type - in this case a UML lifeline (ID: 20). In that regard, consistency rules can also cover basic well-formedness checks. One thing to consider here is, that requirements might affect several UML elements. So, if specified, we may want to check the super type (UML diagram) of UML Lifeline instead. On the requirements side we now need a value to compare this type to. Mind that the type of a field can be specified in the field definitions of an artifact (see Figure 3; e.g., the field "Name" is of type string, while the requirement ID is an integer). This can be utilized as a comparative value in our rule's condition, if the aforementioned super type (UML diagram) is specified as the type of the "Affects" field (remember that types can be linked to other types and, therefore, reference them as type for their fields). The fully realized consistency rule can be seen in Listing 2.

With the consistency rules defined and set up as Rule Definition Artifacts within the cloud, the adaptions made by engineers (see Section 2.2) can take effect:

The first adjustment is John recording an additional requirement in his requirements tool. Like all tools in this example, John's tool is incrementally synchronized with the cloud environment (see Section 3.2.3). This means, whenever he adds a requirement in the GUI of his tool, an artifact of the type "requirement" (see Figure 3 respectively Section 3.2.2) is automatically instantiated in the cloud. This action triggers the consistency checker's "Artifact Creation" algorithm (see Algorithm 1) through a notification, since the "Requirement" type is a context in a rule definition (see Listing 2). Subsequently the consistency checker will create a new Rule Evaluation artifact, with the newly instantiated requirement artifact as context element. This Rule Evaluation artifact will be evaluated immediately. Since the requirement is completely new and not yet represented within the UML design, the result of the consistency check will indicate an inconsistency and consistency rule CR2 as broken. This is reported to John through the GUI of his synchronized tool, e.g., by marking the corresponding requirement in an overview.

On Alex's side, the new requirement makes it necessary to adapt his UML Design. In his synchronized UML tool he adds the new lifeline "Language Loader" to an existing sequence diagram (see Figure 1). This triggers the instantiation of a new artifact of type "lifeline" (see Figure 4). A new Rule Evaluation artifact is created, with the said "lifeline" artifact as the context element. The corresponding Rule Definition artifact represents consistency rule CR1. The rule is evaluated immediately and the result indicates that consistency rule CR1 is broken. Through the cloud's linking tool (see Section 3.3) Alex, or a dedicated domain engineer, can now link the newly instantiated lifeline artifact to the requirement. This results in an update to both the lifeline and the requirement artifact, which triggers the consistency checkers "Artifact Modification" algorithm (see Algorithm 2) and a re-evaluation of the respective rule evaluation artifacts. This time consistency rule CR2 will hold, since Alex linked the requirement artifact's "Affects" link, respectively the "AffectedBy" link on the lifeline artifact (see Figure 4). This information will also be propagated to John's tool, whose GUI now marks the requirement as consistent. However, the evaluation of consistency rule CR1 will still fail and the according GUI elements (the lifeline, respectively its operations) in Alex's UML tool will be marked as inconsistent.

The final changes need to be made by Alice. She implements the new requirement in the code, respecting the design laid out by Alex. To ensure this she links the newly instantiated class artifacts (see again Figure 4) with Alex's lifeline artifact. This update triggers the consistency checker's "Artifact Modification" algorithm and causes the re-evaluation of the lifeline artifact, making sure that Alice's methods are the same as the operations in the sequence diagram. Since the lifeline artifact is now correctly linked, consistency rule CR1 will hold as well. This information is also propagated to John's tool, removing the inconsistency warnings from its GUI elements. With this, all rules hold and the state of the project is consistent.

5 VALIDATION

To demonstrate the feasibility of our approach, this section covers a listing of various tool adapters, validating its applicability within a diverse range of engineering projects. Furthermore, this section validates both the scalability and the usability of our global consistency checking approach, on the basis of an empirical study and two case studies.

5.1 Existing Tool Adapters

Currently, the DesignSpace offers multiple tool adapters. These tool adapters demonstrate the wide range of engineering artifacts the DesignSpace can handle, as well as the adaptability of its API to several different programming languages and engineering domains.

Tool adapters have been written for software engineering tools (e.g., Eclipse, RSA and Papyrus), electrical engineering tools (e.g., EPlan Electric P8 [3][7] and PTC Creo [7]), as well as for common purpose tools (e.g., Microsoft Excel and Visio). All of these tools produce very different engineering artifacts, including Java code, UML models, electrical models, CAD drawings, data tables and regular diagrams. Engineering artifacts synchronized with the DesignSpace adhere to different metamodels. Among others these include various tool-internal metamodels (EPlan and Creo), the Ecore metamodel and extensions thereof (Papyrus and RSA) as well as the Java metamodel. These metamodels, or parts thereof, are synchronized with the DesignSpace as well. Tool Adapters have been written in Java, C# and C++. Thanks to the adaptability of the DesignSpace API the range of potential tool adapter languages can easily be extended.

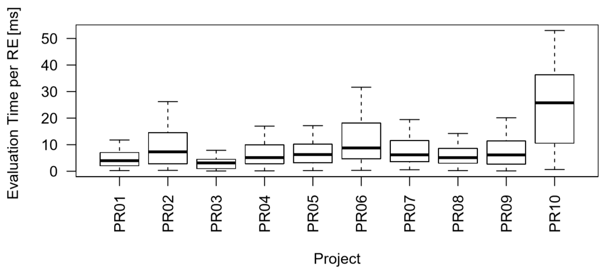
5.2 Links

In previous applications of the DesignSpace several different types of engineering artifacts have been linked together. For example, EPlan artifacts have been linked to Java code [3] and Creo CAD hardware components - respectively geometric shapes - have been linked to UML elements [7]. Note that these are very different engineering artifacts, coming from different engineering domains. Their linking acts as a proof of concept, that the semantic gap between engineering artifacts can be bridged with the help of the DesignSpace. Furthermore, this work demonstrated how requirements could be linked to UML elements and in turn to Java code.

5.3 Empirical Study

Our approach was applied to ten projects within a lab experiment to validate its computational scalability. An overview of these projects can be found in Table 1. The projects consisted of both UML models and code. UML Models were made up of class diagrams as well as sequence diagrams and state charts. While the column #Evaluations describes the number of Rule Evaluations, the columns #CodeElements and #ModelElements describe the size of model and code respectively. Projects ranged from small, like an implementation

Figure 5: Comparison of total processing time



	Project Name	Class	Sequence	Statechart	#Model Elements	#Code Elements	#Evaluations
PR01	GameOfLife	x	x	x	286	261	148
PR02	Checkers	x	x		425	342	171
PR03	TestService	x	x	x	397	302	268
PR04	SMTSolver	x	x		1,254	2,021	791
PR05	ATMExample	x	x		1,341	1,550	983
PR06	TaxiSystem	x	x		1,930	3,488	1,318
PR07	VOD3	x	x		2,538	3,613	1,534
PR08	ObstacleRace	x	x		1,992	3,555	1,600
PR09	biter	x			2,648	4,015	1,957
PR10	ArgoUML	x			6,039	19,557	3,827

Table 1: Case study projects

of the "Game of Life"[12], to big multi-developer projects like the ArgoUML tool[2]. The evaluation was based on 14 consistency rules, which can be found in [19].

We validated the computational scalability of our approach by systematically applying changes to all elements of our projects and by capturing the time required to re-evaluate all rule evaluation artifacts. We ensured that each element present in the scope of a rule evaluation artifact was changed. Mean and median times were observed and used for our analysis. Figure 5 shows the mean processing time per affected rule evaluation artifact. The mean observed processing time was 10.7 ms, while the median was 8.8 ms. The total processing time remain below 50 ms on average, which is an acceptable time for tool users [17].

5.4 Case Studies

Our approach was validated on the basis of two case studies. One was performed with the help of the Austrian Center of Competence in Mechatronics (ACCM[3]), another with the Flander's Mechatronics Technology Center (FMTC[4]). These case studies suggest the usability of our approach.

5.4.1 ACCM Robot Arm. This case study provides a live global consistency checking for the design, mechanical calculation and partial implementation of a robot arm [7]. Engineering artifacts involved were Excel sheets, CAD drawings, UML models based on IBM RSA and source code written in the Eclipse IDE. For explicit linking, the case study's implementation also provides the ability to define and check traceability links between different involved engineering artifacts.

5.4.2 FMTC. This case study provides live global consistency checking between electrical circuit diagrams as well as an implementation [3]. The artifacts involved were EPLAN Electric P8 drawings as well as source code. Both were provided by a third-party company.

5.5 Threats to Validity

Our empirical validation confirms that the approach presented in this paper is feasible within a real world environment. All projects were created by different groups of developers and companies. Both project sizes and domains were highly diverse.

One threat to validity is the effort required to implement and set up tool adapters. Both our case studies and the amount of exemplary tool adapters illustrate, that the integration of engineering artifacts can be completed with a reasonable amount of effort. Furthermore, the granularity of the data integration is fully up to the tool adapter implementation, meaning that even subsets of an engineering artifact's data can be synchronized with the DesignSpace.

Another threat to validity is the correctness and completeness of links connecting engineering artifacts. Since all links are defined by engineers - either explicitly in the form of artifacts or implicitly in the form of consistency rules - link completeness is dependent on the engineers' assessment of their respective projects. This is also true for the link correctness. Likewise the completeness of consistency rules depends on the necessity implied by the development project and its developers. To ensure syntactically correct consistency rules, our approach checks the syntax of a condition and the validity of the context type during the creation of a Rule Definition artifact.

6 RELATED WORK

Currently a multitude of consistency checking mechanisms exists (e.g., [10][11][18]). In practice, consistency checking mechanisms are either integrated into standalone tools (very common with UML modeling tools) or executed on documents (very commonly XML). While the consistency checking mechanisms for standalone tools tend to be agile and support live, often instant inconsistency feedback with changes, their consistency checking is limited to the kinds of artifacts available within the tool. Document-centric consistency checking approaches let engineers combine artifacts from multiple tools and hence support global consistenc checking; however, at the expense of agility. Document-centric approaches are not change driven and hence expensive and disruptive to use. In all cases, consistency checking mechanisms require a single, well-defined metamodel. For example, the consistency of a UML modeling tool is checked within the context of the UML metamodel or the consistency of a Java programming environment is checked within the context of the Java metamodel. Similiarly, the consistency of a merged document is checked within the context of that document's grammar (e.g., XMIs in case of XML documents). Nonetheless, the limitations of all mechanisms are always the same. Every approach today 1) either focuses on the limited knowledge

[2]ArgoUML: http://www.argouml.tigris.org/ (last accessed: 06.12.2018)
[3]ACCM/LCM: https://www.lcm.at/ (last accessed: 06.12.2018)
[4]FMTC: http://www.flandersmake.be/en (last accessed: 06.12.2018)

available within their respective engineering tools (even if the technology is applied to different engineering tools, each tool has a limited perspective) and/or 2) they are potentially able to merge artifacts from different tools but this requires explicit integration, merging effort and is far from instant (i.e., all engineers involved need to agree on some timing when to export and merge knowledge). There are only very few approaches that attempt live, global multi-tool consistency checking. These are discussed next.

One document based approach is ArchJava, developed by Aldrich et al. [1]. It couples an architecture description language with implementation code. For this it utilizes an extended version of Java, featuring a unique type system that guarantees communication integrity between the code and the architecture. Archface [20] acts as both a programming-level interface as well as an architectural description language. It relies on architectural constraints within the implementation to realize traceability. Likewise, DiaSpec [2] uses an architectural description language to describe allowed interactions between components. Inconsistencies are detected with the help of the Java compiler. The problem of ArchJava, Archface and DiaSpec are the limitation to a specific programming language, respectively its compiler. Our approach relies on a generic data format into which a multitude of different engineering artifacts can be translated. Another document based approach is discussed by Nentwich et al. [16]. It generates links between distributed XML-based web artifacts and checks their consistency. In contrast our approach is not based on distributed documents but centralized engineering artifacts. This removes the network communication delay from the consistency checker's processing time. Furthermore, our approach delivers live consistency feedback to engineers.

Koenig et al. [14] presented an approach that is closer to the ideals of global consistency checking. This approach allows heterogeneous multimodels to be analyzed as locally as possible by only comparing model elements relevant to a global constraint. Doing so, significantly reduces the effort required for model matching respectively merging. TReMer+ [15] is a tool for model merging and global consistency checking. It includes merging operations for requirements, behavior models, design and implementation. Both Koenig et al. [14] and TReMer+ [15] rely on partial merging respectively the comparison of metamodels. Our approach relies on links between engineering artifacts, which act as a lightweight alternative to model merging.

Finally, Egyed et al. [7] as well as Demuth et al. [3][4][5] covered certain aspects of live, global consistency checking but with key differences. The solution proposed by Demuth et al. [5] alike ArchJava or DiaSpec provided a solution where Eclipse was used to represent both model and code. Egyed et al. [7] already introduced a cloud to provide a central place of artifact storage with a uniform representation. However, this paper focused on reducing the memory and computational footprint of checking the same consistency rules separately for each tool. Our approach checks the rules once centrally and then makes the results available to all relevant tools. Finally, Demuth et al. [3] reports on an industrial application of their proposed approach in the form of an experience report. The experience report focused primariliy on the need for live, global consistency checking and how a cloud could support it. However, that approach required engineers to be aware of the cloud, whereas our approach is able to effectively hide the existence of the cloud, such that engineers remain mostly unaware of it.

7 CONCLUSION

This paper discusses a novel approach towards live global consistency checking, with the help of an engineering cloud environment. It presents the functionality and architecture of the consistency checking mechanism and demonstrates its feasibility with two industrial case studies. The approach uses the DesignSpace as a cloud environment. In future, the consistency checking mechanism will be adapted to consider private views, respectively different versions of engineering artifacts. We will investigate a way to compute consistency checking information for differently versioned artifacts, including those that might be in conflict with those of other private views. We will also analyse how the results of rule evaluations could be provided to users in a helpful manner (e.g., through mutable pop-up warnings integrated in tool adapters).

REFERENCES

[1] J. Aldrich, C. Chambers, and D. Notkin. 2002. Architectural reasoning Archjava. *ECOOP* (2002), 334–367.
[2] D. Cassou, E. Balland, C. Consel, and J. Lawall. 2011. Leveraging Software Architectures to Guide and Verify the Development of Sense/Compute/Control Applications. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 431–440.
[3] A. Demuth, R. Kretschmer, A. Egyed, and D. Maes. 2016. Introducing Traceability and Consistency Checking for Change Impact Analysis across Engineering Tools in an Automation Solution Company: An Experience Report. *32nd International Conference on Software Maintenance and Evolution* (2016), 529–538.
[4] A. Demuth, M. Riedl-Ehrenleitner, and A. Egyed. 2016. Efficient detection of inconsistencies in a multi-developer engineering environment. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 590–601.
[5] A. Demuth, M. Riedl-Ehrenleitner, A. Noehrer, P. Hehenberger, K. Zeman, and A. Egyed. 2015. DesignSpace - An Infrastructure for Multi-User/Multi-Tool Engineering. (2015), 1486–1491.
[6] A. Egyed. 2011. Automatically Detecting and Tracking Inconsistencies in Software Design Models. *IEEE Transactions on Software Engineering* 37, 2 (2011), 188–204.
[7] A. Egyed, K. Zeman, P. Hehenberger, and A. Demuth. 2018. Maintaining Consistency across Engineering Artifacts. *IEEE Computer* (2018), 28 – 35.
[8] M. Fagan. 1976. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal* (1976), 182 – 211.
[9] A. Finkelstein. 2000. A Foolish Consistency: Technical Challenges in Consistency Management. In *Database and Expert Systems Applications*, Mohamed Ibrahim, Josef Küng, and Norman Revell (Eds.). Springer Berlin Heidelberg, 1–5.
[10] A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. 1994. Inconsistency Handling in Multiperspective Specifications. *IEEE Trans. Softw. Eng.* 20, 8 (Aug. 1994), 569–578.
[11] P. Fradet, D. Le Métayer, and M. Périn. 1999. Consistency Checking for Multiple View Software Architectures. *SIGSOFT Softw. Eng. Notes* 24, 6 (1999), 410–428.
[12] M. Gardner. 1970. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American* 223 (1970), 120–123.
[13] A. Ghabi and A. Egyed. 2012. Exploiting Traceability Uncertainty between Architectural Models and Code. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. 171–180.
[14] H. Koenig and Z. Diskin. 2016. Advanced Local Checking of Global Consistency in Heterogeneous Multimodeling. *ECMFA* (2016), 19–35.
[15] Sabetzadeh M., Nejati S., and Easterbrook S. 2008. Global consistency checking of distributed models with TReMer+. *ICSE08* (2008), 815–818.
[16] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelsteiin. 2002. Xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Trans. Internet Technol.* 2, 2 (May 2002), 151–185.
[17] J. Nielsen. 1993. Usability Engineering. (1993).
[18] S. Reiss. 2006. Incremental Maintenance of Software Artifacts. *IEEE Transactions on Software Engineering* 32 (10 2006), 682–697.
[19] M. Riedl-Ehrenleitner. 2013. *Model-and-Code Consistency Checking*. Master's thesis. Johannes Kepler University.
[20] N. Ubayashi, J. Nomura, and T. Tamai. 2010. Archface: A contract place where architectural design and code meet together. *2010 ACM/IEEE 32nd International Conference on Software Engineering* 1 (01 2010), 75–84.