Software and Systems Modeling (2022) 21:2489–2515
https://doi.org/10.1007/s10270-022-00984-4

REGULAR PAPER

Instant and global consistency checking during collaborative engineering

Michael Alexander Tröls[1] · Luciano Marchezan[1] · Atif Mashkoor[1] · Alexander Egyed[1]

Received: 18 December 2020 / Revised: 29 October 2021 / Accepted: 20 January 2022 / Published online: 8 April 2022

Abstract
Engineering projects involve a variety of artifacts such as requirements, design, or source code. These artifacts, many of which tend to be interdependent, are often manipulated concurrently. To keep artifacts consistent, engineers must continuously consider their work in relation to the work of multiple other engineers. Traditional consistency checking approaches reason efficiently over artifact changes and their consistency implications. However, they do so solely within the boundaries of specific tools and their specific artifacts (e.g., consistency checking between different UML models). This makes it difficult to examine the consistency between different types of artifacts (e.g., consistency checking between UML models and the source code). Global consistency checking can help addressing this problem. However, it usually requires a disruptive and time-consuming merging process for artifacts. This article presents a novel, cloud-based approach to global consistency checking in a multi-developer/-tool engineering environment. It allows for global consistency checking across all artifacts that engineers work on concurrently. Moreover, it reasons over artifact changes immediately after the change happened, while keeping the (memory/CPU) cost of consistency checking minimal. The feasibility and scalability of our approach were demonstrated by a prototype implementation and through an empirical validation.

Keywords Consistency checking · Multi-developer environment · Model-driven engineering

1 Introduction

Software engineering is an inherently collaborative discipline with engineers working concurrently on a wide range of *engineering artifacts*—requirements, use cases, design, code, and more. To modify these engineering artifacts, an equally diverse tool landscape exists—each tool usually specializing in specific types of engineering artifacts (e.g., Intellij[1] specializes on source code or IBM Rational Software Architect[2] on UML models). Each tool thus provides a partial view on a software system, merely considering a subset of the involved engineering artifacts [1]. After all, engineers often do not necessarily need access to all engineering artifacts [2–5].

The software engineering landscape is characterized by the distributed and concurrent modification of engineering artifacts by multiple engineers. This follows a familiar pattern: Typically, engineers download artifacts from a shared repository to their workstations and modify them independently there before finally uploading their changes back to the repository for others to see. In doing so, engineers create a local environment where they typically only have access to a subset of artifacts—those that can be modified by the tool(s)

Communicated by Richard Freeman Paige.

The research reported in this paper has been partly funded by the Austrian Science Fund (FWF) (Grant # P31989-N31 as well as Grant # I 4744-N), the LIT Secure and Correct System Lab sponsored by the province of Upper Austria, and by the Austrian COMET K1-Centre Pro2Future of the Austrian Research Promotion Agency (FFG) with funding from the Austrian ministries BMVIT and BMDW.

✉ Atif Mashkoor
atif.mashkoor@jku.at

Michael Alexander Tröls
michael.troels@jku.at

Luciano Marchezan
luciano.marchezan_de_paula@jku.at

Alexander Egyed
alexander.egyed@jku.at

[1] Institute of Software Systems Engineering, Johannes Kepler University, Linz, Austria

[1] Intellij: https://www.jetbrains.com/idea/.

[2] IBM RSA: https://www.ibm.com/developerworks/rational/products/rsa.

they use. For example, the designer will use a modeling tool to modify modeling artifacts while the programmer will use a programming tool to modify code artifacts.

Inconsistencies arise if artifacts contradict one another. It is particularly hard to spot inconsistencies in situations where different engineers modify heterogeneous, interdependent engineering artifacts within different tools. For example, the modification of a model may result in inconsistencies with the code that represents it. Such inconsistencies might go unnoticed and might exist within the project for a long time, potentially resulting in severe and costly rework. The detection of inconsistencies in a timely manner is still an ongoing research topic [5]. While many consistency checking approaches exist today (e.g., [1,4,6–12]), they do not focus on heterogeneous artifacts and their interdependencies.

Existing approaches normally check engineering artifacts in a local environment, e.g., within an engineering tool. They rarely operate on the merged sum of all engineering artifacts, i.e., a global environment like a shared repository. The modifications within a local environment cannot be considered with regard to the global environment unless the modifications are merged first. However, this is contradictory to the typical circumstances of an engineer's workflow. Engineers typically store their artifact modifications locally first. Yet, these modifications may have implications on the consistency of heterogeneous, interdependent artifacts in the global environment. We refer to the analysis of this consistency as global consistency checking.

This article introduces a novel approach for global consistency checking in a multi-developer, multi-tool engineering environment. Our approach relies on a cloud infrastructure to maintain i) a central, public space that contains the entirety of engineering artifacts shared by all engineers (representing the global environment of artifacts from different tools), and ii) individual work spaces for each tool used by each engineer to reflect their individual, not yet merged modifications (representing the local environments). Modifications that engineers perform in their tools are instantly propagated (e.g., following a tool-internal change event) to their respective individual work spaces in the cloud. There, these modifications are checked for their consistency with the public space, thus enabling global consistency checking while considering each engineer's local modifications. This happens before merging any content of the individual work space with the public space, meaning that global consistency information can be provided without performing a time-consuming merging process. We designed our approach to systematically reuse public and instant consistency checking knowledge to increase our consistency checker's performance.

In this study, we extend our previous work [13–18]. This extension includes a discussion of the functionality of layering our platform considering both public and individual work spaces. This layering mechanism presents a technical novelty as it deals with additional immediate perspectives on the consistency data (see Sect. 4). This mechanism expands our previous work [15], which only dealt with the concept of public work spaces. Furthermore, we expand on the discussion and use of trace links among artifacts (see Sect. 4.8). This linking process abstracts and expands concepts introduced in our previous work [17]. Lastly, we describe the role of change notifications including the technically related discussion about how to handle different commit scenarios (Sect. 5). These two discussions are essential to the current solution and were not detailed in previous works.

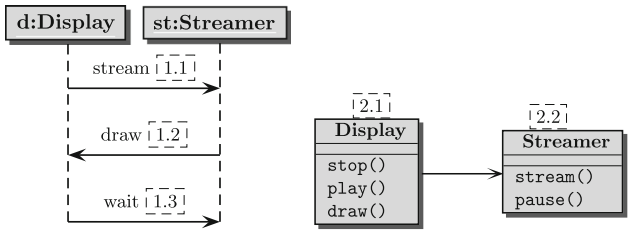
The rest of this article is organized as follows: In Sect. 2, we present a simple example to illustrate inconsistency related problems arising during the collaborative development. In Sect. 3, we outline the problem addressed by the goals of this work. Section 4 discusses the architecture of our approach, respectively, the cloud environment utilized in it. Section 5 describes how we realize global consistency checking. Section 6 discusses the applicability of the proposed approach: we analyze its computational complexity and memory consumption, and show its feasibility by developing a prototype. We also discuss limitations regarding our approach and the future work planned for addressing them in Sect. 7. The article is concluded after discussing the related work in Sect. 8.

2 Illustrative example

In order to demonstrate the approach presented in this paper, we use the model and code fragments of a video on demand (VOD) system [19] as an illustrative example. Let us assume that both model and code are concurrently modified by two engineers: Alice, who writes the Java code, and Bob, who provides the equivalent UML models. Note that when referring to elements of the example, we will use the syntax `[UML::<type>]<name>`, where `<type>` is a placeholder for a specific UML type and `<name>` for its name. Likewise, `[Java::<type>]<name>` will refer to the same on the Java side.

2.1 Initial state

Figure 1 shows the UML model of the VOD system that both Bob and Alice are familiar with. The UML model consists of two diagrams: (a) a sequence diagram outlining the interaction among those two classes, and (b) a class diagram showing the entities `Display` and `Streamer`. Likewise, Listings 1 and 2 show an excerpt of the current Java classes.



(a) Sequence Diagram **(b)** Class Diagram

Fig. 1 Current public UML diagrams

```
1  Public class Display
2  {
3      Streamer streamer;
4      Void stop()
5      {
6          streamer.stop();
7      }
8
9      Void play()
10     {
11         streamer.stream();
12     }
13
14     Void draw (){...}
15 }
```

Listing 1 Excerpt of Java class "Display"

```
1  Public class Streamer
2  {
3      Display display;
4      Void stream()
5      {
6          display.draw();
7      }
8
9      Void wait(){...}
10 }
```

Listing 2 Excerpt of Java class "Streamer"

In the interest of a consistent final product, both the UML diagrams and code must meet the following rules:

- Rule 1: The messages used in the sequence diagram must be present as an operation in their respective receiver's class (e.g., `[UML::Message] stream` must be present in `[UML::Class] Streamer`).
- Rule 2: All operations of a UML class must be present in the respective java implementation (e.g., `[UML::Operation] pause` must be present as a method in `[Java::Class] Streamer`).

The former rule expresses a circumstance mostly covered by traditional, tool-centric consistency checking. Consistency checking between different artifacts from the same engineering discipline can often be performed by the tool in which the artifacts were originally designed, e.g., a UML tool can guarantee that only valid messages are added to a sequence diagram. However, the latter rule requires a more sophisticated approach toward consistency checking. It considers the consistency between equivalent concepts of different types of artifacts in unison (UML models and code). Traditional, tool-centric consistency checking approaches are normally not able to cover this. Consistency checking between different types of engineering artifacts requires a tool or a formalization that can define semantic equivalences between them.

One way to formalize consistency rules is to express them in Object Constraint Language (OCL) [20]. We will refer to such expressions (and their respective representations as engineering artifacts) as Consistency Rule Definitions (CRD)—see `CRD1` and `CRD2` in Listings 3 and 4 .

```
1  UML::Message m
2  m.receiveEvent.covered->forAll(Lifeline l |
3      l.represents.type.ownedOperation->exists(Operation o |
4      o.name = m.name))
```

Listing 3 (CRD1) Message must be defined as an operation in the receiver's UML class

```
1  UML :: Class c
2  self.javaLink -> notEmpty() implies
3  self.operations -> forAll(Operation o |
4      self.javaLink.methods -> exists(Method m |
5          o.name = m.name))
```

Listing 4 (CRD2) All operations in the UML class diagram must be present in their linked Java implementation

As is typical in OCL, each consistency rule is written for a specific *context*. The context describes the type of artifact for which the rule must be evaluated. Since we focus on UML and Java in this example, the context is either a specific UML or Java artifact. CRD1 with the context `UML::Message` checks whether a given `UML::Message` is defined as an operation in the respective UML class. For CRD1 in the sequence diagram in Fig. 1, we find three `UML::Message` instances (`[UML::Message]stream`, `[UML::Message]draw` and `[UML::Message]wait`), and thus three evaluations of CRD1 are necessary—once for each message. We will refer to such an evaluation (and its representation as a concrete engineering artifact) as a consistency rule evaluation (CRE) throughout the rest of the article. Similarly, CRD2 checks whether operations in class diagrams correspond to methods in the java classes. As there are two such classes, we must evaluate CRD2 five times—once for each operation.

This way of formalization lets us consider both UML models and Java, provided that both types of artifacts are explicitly linked through one of their defined properties. If we assume that the consistency checker has access to both UML models and the source code then complete consistency checking would be possible. In that case, a consistency checking service would first identify all needed CREs by searching for all UML model instances matching the context. Indeed, for the VOD example, there are five CREs:

– three instantiations of CRD1 corresponding to three messages in the sequence diagram: `CRE.1.1` for `[UML::Message]stream`, `CRE.1.2` for `[UML::Message]draw` and `CRE.1.3` for `[UML::Message]wait`; (recall Fig. 1a), and
– two instantiations of CRD2 corresponding to the two classes in the class diagram (recall Fig. 1b): `CRE.2.1` for the class `[UML::Class]Display` and `CRE.2.2` for `[UML::Class]Streamer`.

The only unusual part not normally found in consistency rules are trace links between model and code—as needed in CRD2. A trace link is a property added to an artifact, where the value of this property is a reference to another artifact, thus creating a trace between both artifacts. For example, a trace link may reveal which Java class implements which UML class. Although this example uses the same textual names for both, naming conventions can be deceiving, as similarly named entities can refer to different concepts in different engineering fields. Explicit links are needed for a consistency checker to navigate among artifacts from different tools. The OCL element javaLink thus links the UML class Streamer to the same-named Java class. This is a domain specific extension that our approach supports and, as expected, can be customized to many different kinds of links.

Except for `CRE.1.3`, all instances are consistent as they adhere to the defined rules. `CRE.1.3` is inconsistent because there exists no operation called "wait" in the class diagrams.

2.2 Multi-developer consistency checking

Consider now that Alice and Bob modify the UML model and the Java code independently. Each engineer checks out the respective engineering artifacts to his or her local workstation and uses the corresponding tool. Let us first assume that each tool has a separate consistency checker. In our illustration, there must be at least two tools involved: Alice needs to use a programming tool whereas Bob needs to use a modeling tool. Thus, say, Alice has partial knowledge on the full state of the project's engineering artifacts. She merely sees the code with no (immediate) knowledge of its UML representation.

Note that in the following, we append the starting letter of the engineer's name to the consistency rule evaluations to distinguish them (e.g., `CRE.1.3.B` is Bob's consistency rule evaluation of `CRE.1.3`). We think of this as an evaluation in Bob individual work space, as Bob may make changes unknown to Alice. Thus, the consistency checking at this point is *instant*, as it only considers the instant changes performed by Bob. Inconsistencies between Bob and Alice's artifacts are only meaningful once Alice becomes aware of Bob's changes not earlier. Hence, once Bob's changes are global (global consistency checking). The instantiations of individual CREs are illustrated in Fig. 2.

2.2.1 Adaptations by Bob

Bob modifies his individual working copy of the UML model by adding a feature: in order to stream movies a user must first connect to the `[UML::Class]Streamer`. Bob thus makes the following changes consecutively:

– an operation called "connect" is added to the `[UML::Class]Streamer` in the class diagram,
– a connect message is added to the sequence diagram, and
– the operation `[UML::Operation]pause` in the class diagram is renamed to "wait."

The new state of Bob's working copy is depicted in Fig. 3. These changes alter the state of the UML model and thus have implications on the CREs. Incremental consistency checkers are able to react to these changes in a fine-grained manner without having to re-evaluate the entirety of the model. For this we utilize the concept of *scope* [8]. A scope is a set of model elements that is attached to a CRE. If one of these elements is changed, it triggers the re-evaluation of the scope's corresponding CRE. The scope is created automatically at the first evaluation of the CRE and is a list of model elements accessed during the evaluation of the CRE on the UML model. Only changes to these accessed elements can cause the CRE state to change from consistent to inconsistent or vice versa. A CRE is thus re-evaluated if an element in its scope changes. Please see the work of Egyed [8] for further details in this regard and also note that most incremental consistency checkers have similar concepts (e.g., *critical node* [21], or *impact matrix* [22]).

Bob's first change requires a re-evaluation of CRE.2.2 because it affects its context [UML::Class]Streamer. A local (tool-centric) consistency checking mechanism interpreting CRD2 would be unable to assess whether the additional method "connect" is also present in [Java::Class]Streamer, as it has no knowledge of the respective Java class.

The second change Bob makes also impacts consistency because it adds a new instance with regard to CRD1; therefore, a new CRE is required. CRE.1.4.B evaluates [UML::Message]connect. This CRE is consistent because Bob added the corresponding operation to the Streamer class with the first change.

Finally, with the third change, Bob intends to resolve the previous inconsistency between the class and the sequence diagram. By renaming [UML::Operation]wait to [UML::Operation]pause CRE.1.3 is now consistent.

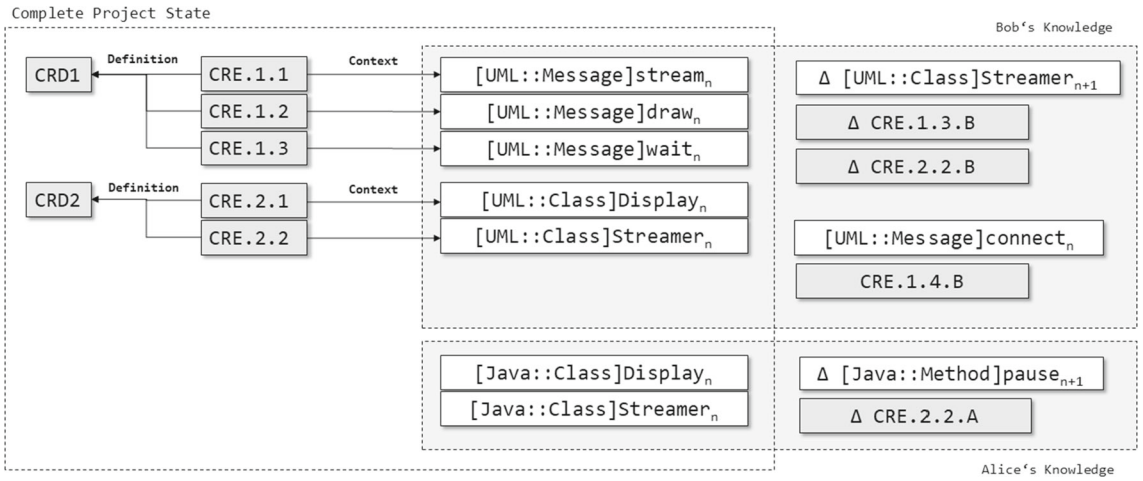


Fig. 2 Consistency rules define CREs for specific engineering artifacts. Alice and Bob produce individual deltas on the state of these CREs by adopting both Java and UML engineering artifacts according to their wishes

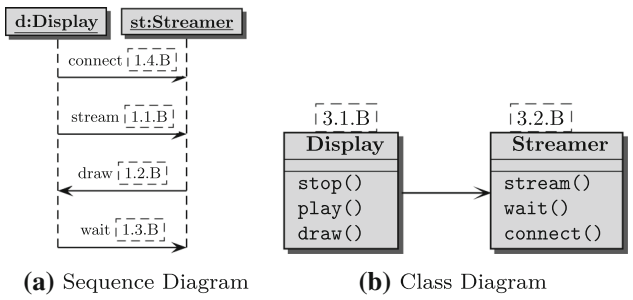


Fig. 3 UML diagram version in Bob's individual work space

days, or even weeks after Alice and Bob made these changes.

```
1  Public class Streamer
2  {
3      Display display;
4      Void stream()
5      {
6          display.draw();
7      }
8
9      Void pause(){...}
10 }
```

Listing 5 Excerpt of Java class "Streamer" in Alice's individual work space

2.2.2 Adaptations by Alice

Let us assume that Alice is aware of the inconsistency with regard to CRE.2.2 between the Java class and the UML class diagram. To resolve it (and unaware of Bob's changes), she decides to rename the [Java::Method] "stop" to "pause"—hence ensuring that her code conforms to the UML model. The state of her working copy is depicted in Listing 5. This change resolves the inconsistency from her perspective and would require the re-evaluation of CRE.2.2. However, she would have to check this inconsistency manually because Alice's programming tool does not have access to the UML model. Even if she were to check out the model, the two tools would not be capable of checking inconsistencies between them. Much like Bob's CRE.2.2.B, Alice's CRE.2.2.A remains non-computable. Alice also remains unaware of the changes Bob made, most significantly of the fact that her change conflicts with Bob's change. If both Alice and Bob committed their changes to the project's main repository, they would find that CRE.2.2 remains inconsistent even though both had the impression they resolved the inconsistency before the commit. This realization may come hours,

3 Problem statement

In this section, we describe the main challenges that motivate our research, as well as the overall aim of this study. As the traditional consistency checking tools like Model/Analyzer [23] or CLIME [1] have limited views on engineering artifacts, this has negative consequences in a collaborative engineering project. We relate these problems to the example discussed in Sect. 2:

– *Incomplete information* Traditional approaches are capable of checking the individual working copies (artifacts within tools) efficiently and incrementally. However, they require all engineering artifacts to be available locally, which is not often the case in multi-developer, multi-tool development scenarios. No tool is capable of representing all engineering artifacts; nor does every engineer need access to all artifacts. In relation to our illustrative example: Alice's renaming of her java method is inconsistent because of her incomplete information regarding Bob's original refactoring of his operation.

– *No support for individual working copies* Traditional approaches that allow for incomplete local knowledge (e.g., [4,6,7,24,25]) do not support the idea of individual adaptations (i.e., every change an engineer performs must immediately be considered public). This limits the applicability of those approaches because not every engineer would like changes to be publicly visible immediately (i.e., trial and error). It is inherently understood that changes are individual until engineers explicitly make them public. In relation to our illustrative example: Both engineers work independently from each other (i.e., individually). An (automated) exchange of consistency information could therefore only happen if the consistency checker could access both Alice's and Bob's (individual) engineering artifacts.
– *Late recognition* Since engineers work separately on their artifacts, their adaptions must be committed and integrated into a single shared, public repository. At this point, consistency checking could detect inconsistencies encompassing the engineering artifacts from different engineering disciplines. However, depending on the frequency of commits this can be late and irregular. Moreover, not all engineers are expected or required to commit at the same time; hence, continuously resulting in a partially incomplete public repository. Merging conflicts and refactoring phases may introduce further errors. In relation to our illustrative example: The inconsistencies between Bob's and Alice's work would at best be recognized during the merge process of their changes. This would likely happen much later than the introduction of the inconsistencies when the error may have spread into other parts of their work.

To address these problems, our goal is to develop an environment that is capable of instantly checking the global impact of artifact changes even if these changes are performed by engineers working with local, individual copies (i.e., tools). The environment's computational effort and memory consumption should be scalable and its performance should allow for quick, incremental consistency feedback, even including in situations as discussed earlier.

4 Cloud environment

Our approach to scalable consistency checking in a multi-developer, multi-tool environment combines the advantages of version control systems, individual work spaces, instant notification, and incremental consistency checking. Our solution systematically reuses computed consistency checking knowledge and provides complete consistency checking for all engineers at all times that is customized to their respective individual spaces. Central to our approach is the utilization of the engineering cloud environment [13]. This environment is crucial as it allows our approach to work as a generic collaborative engine [26]. Thus, allowing the communication and collaboration between multiple tools, users, use of heterogeneous artifacts, and deploying and consuming customizable services, such as the consistency checker. We describe the functionality and architecture of this environment in the following. While doing so, we expand on certain concepts that are critical to our way of consistency checking.

4.1 Overview

Figure 4 depicts an overview of our approach's architecture, respectively, its workflow. With the help of tool adapters (Sect. 4.2), engineering tools transform their internal engineering artifact structure into a fine-grained, uniform representation (Sect. 4.3) and synchronize it with the cloud-based artifact storage. There, the artifacts are stored in a public space (Sect. 4.4). The version history is based on atomic changes, similar to the version control of Resource Description Framework (RDF) [27] or operation-based version control of the Eclipse Modeling Framework (EMF) [28]. Changes on public artifacts—as synchronized by the tool adapters—are stored in a separate individual work space for each individual tool, respectively, tool adapter. A retrieval of data, from the perspective of a tool, always considers the individual changes first and layers them on top of the public artifact. This way engineering artifacts are stored in different tool-specific views, which represent the current work states of engineers. This layering also avoids duplication. Merely artifact changes are stored in individual work spaces and (as we will see later) merely affected consistency rule evaluations are stored there. When a change is stored in an individual work space, a corresponding change notification (Sect. 4.6) is fired. This notification triggers various services (Sect. 4.7) such as the consistency checker. The consistency checker then reacts depending on the nature of the change and reasons over the linked artifact structure (Sect. 4.8) to compute a new consistency state. Contrary to other version control systems, our work maintains consistency checking artifacts (Sect. 4.9) alongside regular engineering artifacts. Consistency checking in our approach is always performed from an individual perspective on the artifact storage, giving us the possibility to provide individualized feedback (Sect. 4.10) to the engineers.

In the following, we discuss the outlined concepts in further detail.

4.2 Tool adapters

The tool adapters of Fig. 4 complement development tools used by engineers. They are custom implementations using the cloud environment's API for both the transformation of

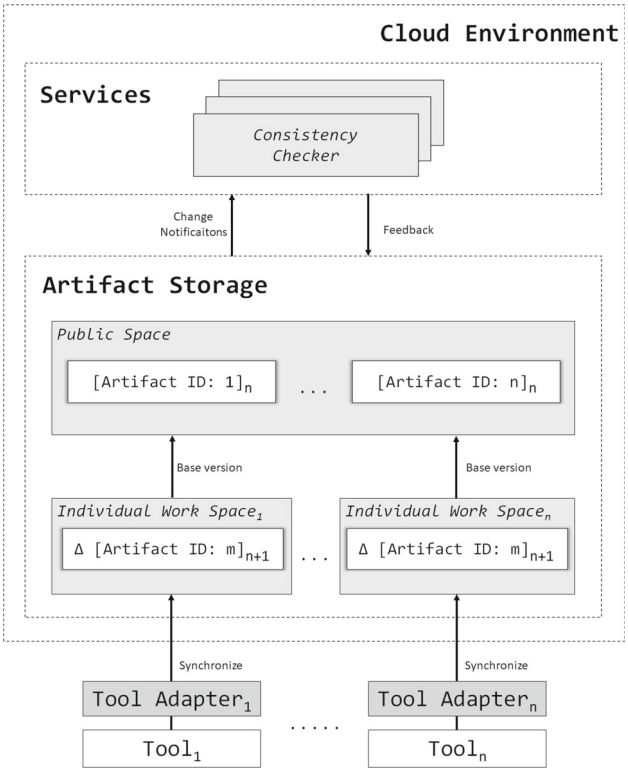


Fig. 4 Architecture of our approach

artifacts into the uniform representation and their synchronization with the cloud environment. Their technical details depend on the respective tool they are built for.

There are, however, a few requirements that these tool adapters must support: i) the adapter must have access to the artifacts created in the tool; ii) the adapter must deal with change notifications happening within the tool; iii) the adapter must connect to the server to provide communication between work spaces; and, iv) the adapter must listen to changes from the server to make the required updates in the local tool. For example, if we consider Eclipse Papyrus'[3] tool adapter, it accesses the UML models created in the tool and listen from change notifications in these models. These notifications are triggered by Papyrus whenever a model is changed. The tool adapter forwards these notifications to the server as changes. In parallel to this process, the tool adapter is also listening to notifications from the server. If notifications are identified, the tool adapter will send them to Papyrus to update the models.

In addition to these tool-adapter requirements, the tools must also satisfy some requirements [26]: i) the tool must allow the tool adapter to identify elements individually (e.g., using identifiers), ii) the tool must have a Software Development Kit (SDK) that communicates with the tool adapter, and iii) the tool's SKD should allow multi-threading for tool adapter to work asynchronously.

As a proof of concept, our engineering environment already supports a wide range of tools such as Papyrus, Eclipse[4], IntelliJ, Microsoft Excel[5], Microsoft Visio[6], Eplan[7], Creo Elements Pro[8], and others.

4.3 Unified representation

Before we discuss our approach further, we need to discuss how engineering artifacts are stored in the cloud and how this allows us to integrate development tools with our infrastructure. As previously mentioned, engineering artifacts (e.g., models and the source code) are translated to a uniform representation. A simplified illustration for this uniform representation is depicted in Fig. 5.

An artifact has a unique `id`. Furthermore, the Boolean flag `alive` indicates whether the artifact is alive or not (i.e., whether it has been deleted or not). Moreover, artifacts may have a number of `properties`, which are key-value pairs. A property has a unique name and is linked to a value, which is stored in the form of a list of deltas, recording every change that has been made on a property. Retrieving a property always retrieves its latest value (the last entry on the list of deltas). The contents of the delta list are either basic data types such as `Boolean`, `Integer`, `Float` and `String`, or references to other engineering artifacts. Various lists are also supported but omitted for simplicity. Both an artifact's properties and their respective data types are defined in the artifact's referenced `type`. For example, Fig. 7 (discussed later in detail) shows a UML Class artifact type, a Java Class artifact type and a Link artifact type. Every time a new engineering artifact is synchronized with the cloud, a corresponding artifact type is instantiated.

Any tool that stores its engineering artifact in a graph-like structure of nodes and edges can easily be mapped to our uniform representation (e.g., ECore [29] structures or abstract syntax trees). The semantics of the original representations do not change; however, sometimes it makes sense to only synchronize the essential parts of an engineering artifact with the cloud. The respective translation of the original representation into the cloud's uniform representation is done by the respective tool adapters with the help of the cloud's API. However, the mapping between the original and the uniform representation must be individually implemented for each type of engineering artifact (e.g., the implementer of a UML

[3] Eclipse Papyrus: https://www.eclipse.org/papyrus/.

[4] https://www.eclipse.org/.

[5] https://www.microsoft.com/excel.

[6] https://www.microsoft.com/visio.

[7] https://www.eplanusa.com/solutions/eplan-platform.

[8] https://www.ptc.com/de/products/cad/creo.

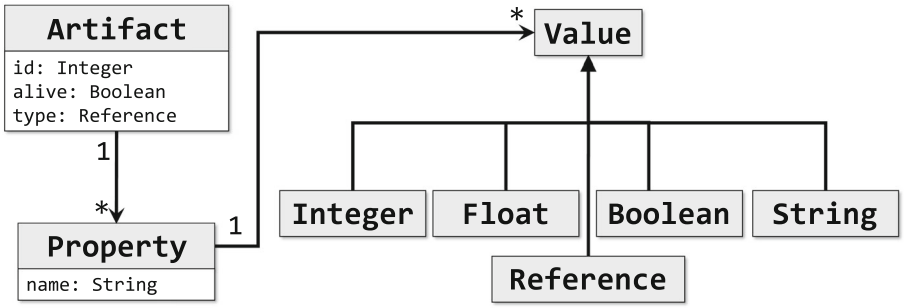


Fig. 5 Structure of engineering artifacts in the uniform representation

tool adapter must decide how specific models and their elements are mapped to the uniform representation, i.e., they must decide on the structure of the respective artifact type).

While designing the cloud environment, we considered the application of EMF [29], 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 that is not tied to the Ecore/Eclipse framework. One of the main reasons for not applying Ecore is to simplify the model representation, including only elements that would be needed for the communication between tools, tool adapters, and the cloud environment. If we had decided to use Ecore, our representation would contain properties and elements that are not necessarily useful for our environment. Another reason for creating our own representation is the possibility to extend it based on the requirements related to the services that communicate with the cloud environment.

Furthermore, we need to address the main benefits of using a unified representation for the artifacts. As reported by Torres *et al.* [30], the majority of consistency checker tools do not provide support for identifying inconsistencies in models from different domains. The main reason for that is that for checking the consistency of such models, their properties and values must be represented using an equivalent structure. Otherwise, we would be unable to compare their properties and values. For instance, if we apply consistency checking considering UML and Excel artifacts, as each tool has a completely different representation, the way of accessing properties and values from their artifacts is completely different. Thus, model transformation techniques may be required. Such techniques, however, may cause the loss of important data [30] that can affect the result of the consistency checker.

The use of an intermediate representation allows our approach to transform all models from different tools into a single type of model that is equivalent independently of the artifact being represented. Furthermore, as the tool adapters are responsible for performing the transformation from the original model into our unified representation, we can prevent loss of important data as each tool adapter is developed considering their respective tool. In this case, data loss may happen with data that is not relevant for the collaborative environment, neither the consistency checker. Furthermore, not all data provided by the tools need to be carried out when conducting model transformation [31].

The main drawback for applying this strategy is the need of developing tool adapters for each tool that will be included in our collaborative engineering environment. As discussed in the previous section, there are requirements that must be fulfilled by the tools to allow a tool adapter to be constructed. Currently, we have already included a variety of tools from different domains that use different models evidencing the potential for the extensibility of our cloud environment.

4.4 Public and individual work spaces

A key aspect of our approach is the organization of the cloud's artifact storage into a Public Space (`PS`) and several individual work spaces (`IWS`). We define these terms as follows:

- *Public Space (PS)* a PS represents an environment in the artifact storage that is shared among all engineers through their individual work spaces (Fig. 4). In the PS, all artifacts only have one version/state. Hence, when changes come from the individual work spaces, these changes must be merged, and possible conflicts must be resolved.
- *Individual Work Space (IWS)* IWSs are owned by the engineers. As the name suggests, in this "space," engineers work individually in their respective tool(s). Hence, each IWS is only accessible to a single engineer, and the artifact's version in this IWS represents this engineer current work. As illustrated in Fig. 4, IWSs are connected with each tool adapter. Each engineer may own multiple IWSs, depending on how many tools the engineer is working on.

This configuration allows the approach to capture the differences between the engineering artifacts in the tools and the shared, publicly accessible engineering artifacts in the cloud. It is important to note that we do not expect development tools to run within the cloud as engineers typically use them on their local workstations. However, IWSs do reside in

the cloud together with the public space. Each IWS reflects the changes (deltas) between the artifacts in the given engineer's tool and the PS. This is achieved by the means of tool adapters, that observe artifact changes in tools and forward them to the corresponding IWS [13].

Tool adapters support a workflow that is typical for development repositories such as SVN [32] (i.e., check-out, modify, commit). Most importantly, they immediately forward artifact changes to their respective IWSs. This ensure that the IWSs are always up-to-date. Hence, the cloud's IWSs reflect tool artifact states. The cloud is thus fully aware of all changes made by all engineers at all times (even though these changes are treated individually unless engineers tell it otherwise). A tool is typically only concerned with the engineering artifacts that it can edit (e.g., a modeling tool edits UML models but not the source code). Yet, to perform a complete consistency check, a consistency checker needs access to all engineering artifacts. By placing IWSs in the cloud, a service has access to each engineer's changes and the PS. Therefore, by extension a consistency checker operating on the IWSs can produce consistency feedback about each engineer's IWS with regard to the public PS—in short, complete public knowledge superimposed with the engineers' reflective individual changes.

During consistency checking, artifacts are retrieved from their respective IWSs. In this process, the changes of the IWS are layered on top of the respective public artifacts in the PS. This is illustrated in Fig. 6. If an IWS does not hold an artifact referenced by a consistency rule, then the consistency checker automatically falls back onto the PS. This way, all publicly available artifacts can be integrated into the consistency checking mechanism, despite the fact, that only parts of them may be relevant to the engineer's work. This organization of engineering artifacts allows for global consistency checking, involving an engineer's individual modifications. Since tool adapters immediately propagate artifact changes from tools to IWSs and a change immediately triggers a consistency check, our approach avoids the problem of late error recognition. Finally, engineers may continue working "individually" while still receiving individualized consistency feedback.

If we consider our illustrative example (Sect. 2), the cloud environment contains all public UML models and the respective source code (See Fig. 1a and Listings 1 and 2). There would be two IWSs (similar to Fig. 4)—one for Bob's UML designer tool and one for Alice's source code tool; both initially empty. As Bob and Alice modify artifacts through their respective tools, these changes are forwarded to their respective IWSs. Bob's IWS eventually contains the changes Bob made as described in Sect. 2.2.1 and Alice's IWS contains the changes she made as described in Sect. 2.2.2 (illustrated as deltas (Δ) in Fig. 4). However, simply executing n copies of a consistency checker—e.g., Model/Analyzer [23]—in the

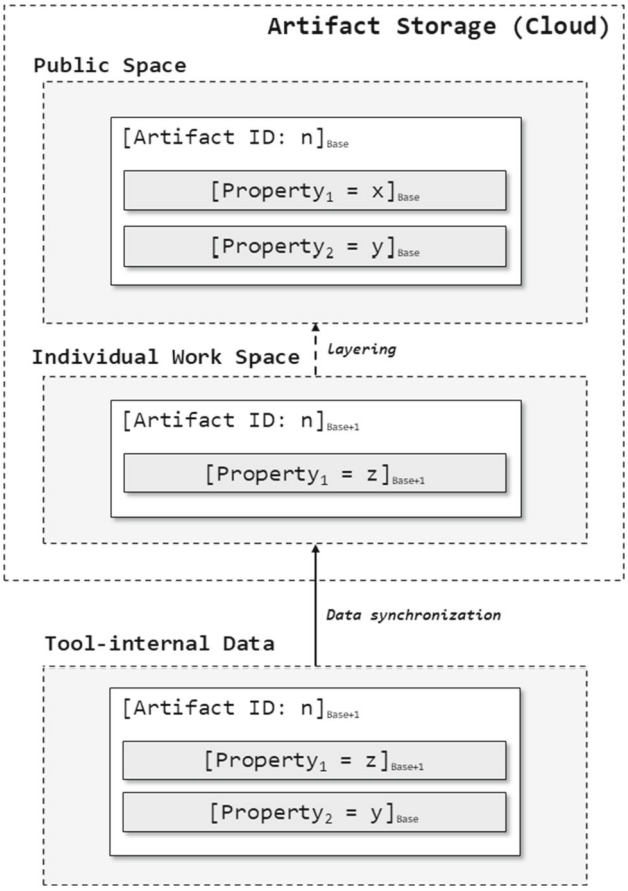


Fig. 6 Retrieval of an artifact by a tool, where the connected individual work space holds a change. The change is layered onto the public artifact

cloud would be inefficient. This would lead to unnecessary CREs and memory consumption because each IWS would replicate similar consistency information. Considering the illustration again, if both IWSs of Alice and Bob have available separate consistency checkers, then both would maintain CREs that, in part are identical.

To address this issue, each IWS not only maintains the delta of the engineering artifacts with respect to the PS (i.e., `IWS.`Δ`artifacts`) but also its corresponding consistency delta (i.e., the added/modified/removed CREs stored in `IWS.`Δ`CRE`). Whenever an engineer modifies an engineering artifact, the consistency checker re-evaluates the affected CREs. When CREs change then they are stored as CRE deltas in the IWS, which constitutes the difference between the tool's consistency state and public consistency state. The version controlling of consistency data also avoids expensive, initial batch consistency checking as the CRE can be checked-out together with the artifacts. A single consistency checker can take advantage of this by re-evaluating consistency information from the perspective of an individual IWS.

4.5 Exemplary artifact retrieval

To follow the illustrative example, Alice at first needs to obtain (i.e., check-out) various Java classes. Alice thus uses the check-out function in her programming IDE—provided by the tool adapter—where she specifies the code artifacts she wants to check out. Listing 6 describes the check-out algorithm.

```
1 checkOut(artifact)
2     IWS = create IndividualWorkingSpace()
3     return IWS, getElements(artifact)
```

Listing 6 The checkOut function

The algorithm first creates an IWS for her tool. The IWS's Δ of the code is empty upon check-out because Alice has not made any changes yet the algorithm returns a tuple of the IWS and the checked-out code artifacts to the tool adapter. The tool adapter then translates the model artifacts to her tool's internal language—which is tool-specific and therefore not depicted.

4.6 Changes and change notifications

Whenever an engineer adapts an engineering artifact within his or her tool, the tool adapter translates this adaptation into one or many atomic changes on the uniform artifact representation of the said engineering artifact. These changes are then forwarded to the cloud where they are stored in the engineer's respective IWS. A change can be one of three types:

- *Create* A "Create" change is caused by the creation of an artifact and its properties.
- *Modify* Whenever a property is modified with a new value it counts as an "Modification" change.
- *Delete* A "Delete" change happens whenever an artifact and its properties are deleted.

Each change type is complemented by a value. The value can be of one of the types illustrated in Fig. 5. How changes are stored effectively depends on the concrete type of the respective work space. In an IWS, a change is only stored as a delta with regard to the same engineering artifact in the PS. When the IWS is eventually committed to the PS, the change is removed from the IWS. In the PS, the change is then added as the latest entry to a list of changes, i.e., a property in the PS references multiple changes. This list of changes acts as a change history.

Whenever a change is stored the respective work space fires a change notification. The notification is sent to listeners of the work space, such as the consistency checker service. In our case, the consistency checker is notified about changes from any IWS and checks them for inconsistencies.

Take, for example, Bob's addition on the `[UML::Class] Streamer` (connect). This change will cause a re-evaluation of `CRE.2.2.B`. The corresponding change notification will be fired from Bob's IWS. On re-evaluating the consistency rule, the consistency checker can write feedback on the change into Bob's IWS, immediately making Bob aware of the new consistency state. Bob can then continue and commit his changes to the PS. On Alice's side, the consistency checker can then re-evaluate her own version of the consistency rule, making sure, that her local changes are not in conflict with the latest changes of her co-worker. This reduces the problem of the conflicting, concurrent changes made by Alice and Bob (discussed earlier). She may decide to immediately attune her work to Bob's change. As a result, her later commit to the public work space will not cause an inconsistency to the public `CRE.2.2`.

4.7 Services

In our approach, consistency checking is realized as a service. A service is an automated mechanism that can analyze and alter the contents of the cloud environment's artifact storage. To do so, a service reacts toward the changes within a specific work space. The service's mechanism is then executed on the respective space. The consistency checker adds and alters artifacts solely in terms of evaluated CREs, which are kept alongside regular artifacts.

4.8 Linking

A major aspect for every modern engineering project is the awareness and documentation of relationships between its engineering artifacts. Documenting these relationships provides useful insights for several engineering disciplines, e.g., in the form of traceability links, widely used in requirements engineering. It also provides us with information about artifacts that need to be kept consistent, regardless of whether there is an automated mechanism to check consistency for us. The relationships between artifacts can either be of an implicit nature (e.g., established via naming conventions) or be explicitly documented. The documentation of relationships is often done in separate tools (e.g., traceability tools), independently of the concrete engineering artifacts. This limits the possibilities for analysis and requires additional effort.

In our approach, the relationships, respectively, links, between artifacts are captured side-by-side with the concrete engineering artifacts. There are two ways to establishing links in the cloud environment. They can be simple references of one artifact to another by adding a respective property field to the linked engineering artifact. Or they can be separate artifacts pointing at both the source and the target of the link as property fields (see Fig. 7). The way how to represent links is mostly up to the concrete problem—conceptually there is

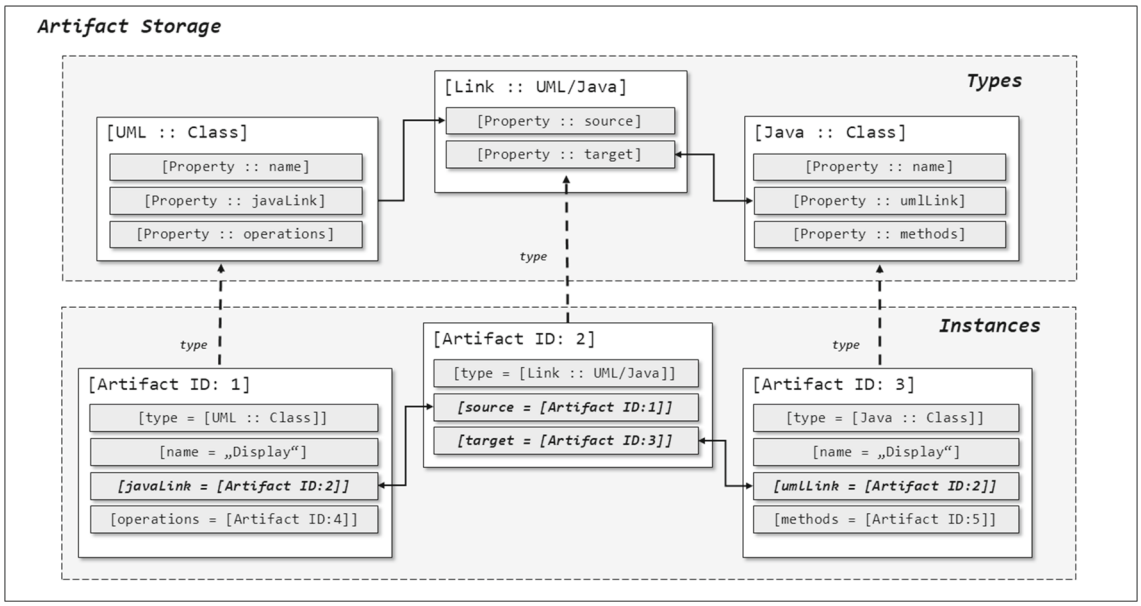


Fig. 7 Two differently typed artifacts instantiations linked through a dedicated link artifact type, respectively, its instantiation

no deciding advantage in one way or the other. In the latter case however, links are organized as actual engineering artifacts themselves. This provides us with further possibilities to reason over them in our consistency checking mechanism (e.g., by the addition of meta-information on the concrete link artifact, such as a specific link type—a UML-specific "implements" association—could be captured this way). In this work, it was sufficient to represent links as regular references.

Another important consideration to make is the automation of the linking process in the cloud. Potential solutions for this issue range from fully automated (e.g., name-matching) to semi-automated (e.g., heuristic recommender-systems) to manual approaches (i.e., regular artifact linking via specific tools). Each of these solutions has its own advantages and disadvantages with regard to flexibility, correctness, and completeness of the resulting link structure. However, the necessary automation of the linking process is also up to the circumstance and scale of the engineering project. Bigger projects may require an automated approach, which can be realized in the form of an automated service operating on the artifact storage of the cloud environment. While such approaches for the automatic establishment of links exist (e.g., as presented by Ghabi et al. [33]), manual approaches are still the prevalent way to tackle the problem. Our work does not prescribe a mechanism. How to link is up to the engineers.

We establish links between related artifacts to provide navigation structures for the consistency checking mechanism. As such, consistency rules may include linking property fields to refer from one artifact to another. Consider again consistency rule CRD2 (see Listing 4), referring to the artifact structure illustrated in Fig. 7. In this rule, the consistency checker would first retrieve the name field of an operation before comparing it to a linked java-specific method name field. The "javaLink" property serves as a navigation link between the code and the UML artifact. Vice versa, the artifact representing the java class is referencing its respective UML representation, via a property called "umlLink." Every link in the cloud environment is established bidirectionally. If a user links an artifact to another, the reverse link is automatically created. The property field for the reverse link is defined in the type information of the linked artifacts.

4.9 Consistency checking artifacts

Earlier we discussed the concepts of CRDs and CREs (see Sect. 2). In the following, we discuss how these concepts can be realized. In our approach, engineers may add CRD at any time. For this the consistency checker uses the uniform representation of artifacts in order to store both rule definitions (CRDs) and results (CREs). The consistency checker therefore creates instances of two different engineering artifact types:

- *Consistency Rule Definition (CRD) Artifact* These artifacts hold information on the type of artifact a rule is applied to (the context), as well as the concrete rule itself—written in OCL. These artifacts are the representation of a CRD.
- *Consistency Rule Evaluation (CRE) Artifact* These artifacts realize CRDs for a certain artifact instantiation (the

context element). They hold their current consistency result as well as a reference to the respective CRD. Further, they keep track of the so called "scope." Equivalent to the concept described in Sect. 2.2.1, a scope is a list of artifact/property pairs that are involved in the evaluation of a certain rule, regarding a specific context element.

When an engineer creates a type, this type can be used as the context of a CRD. With the creation of a CRD (done by an engineer via a specialized tool), a CRE is automatically created for each artifact of the context type. These CREs are stored in the work spaces the respective artifacts reside in.

For example, if Alice creates an artifact type for Java packages, our tool would automatically recognize the said type and make it possible to create a rule definition for it. If she goes on to create a CRD for this type, the consistency checker will create the respective CREs in Alice's IWS. These CREs are immediately evaluated on the basis of the newly created rule.

During the initial evaluation, the scope of a CRE is built. The scope is a set of all artifacts, respectively, their properties, which are traversed during the rule evaluation. On side of the referred property, a reference to the CRE is added (as a delta within the IWS from whose perspective a rule is checked). Should any of these properties change during runtime, the reference is followed back to the CRE and a re-evaluation is initiated.

4.10 Feedback

Since services can write their results into IWSs in the form of artifacts, tool adapters can synchronize and interpret such results. Every tool adapter can transform service results into individual feedback, corresponding to the changes adapted on their tool-internal engineering artifacts. In case of the consistency checker, the service writes a result as a delta on a CRE. Such a delta can be recognized by the tool adapter and lead to individualized consistency feedback within the tool, e.g., a screen warning or the different coloring of inconsistent artifacts in a UML tool.

5 Consistency checking

An overview of the consistency checking approach is illustrated in Fig. 8, showing the main workflow related to the consistency checking performed. In *Step 1*, namely, *engineers perform changes*, Alice and Bob perform the changes in their corresponding artifacts as discussed in Sect. 2. These changes are synchronized with their respective IWSs. Then, the instant consistency checking mechanism starts (*Step 2*), where the new changes performed in their IWSs are analyzed to find possible inconsistencies. The next step, *Step 3*, is when the global consistency checking happens. This is triggered by commits from the individual work spaces into the public space. During this step, the changes committed from an IWS are analyzed to find possible inconsistencies originating from them in the public space. Hence, our approach only needs to analyze the artifacts modified by these new changes, rather than the whole artifact storage in the public space. Furthermore, it is possible that these new changes (e.g., Alice's changes), now included in the public space, create inconsistencies in the IWS of other engineers (e.g., Bob). In this case, the approach performs consistency notifications (*Step 4*), notifying Bob which artifacts in his IWS are inconsistent concerning the public space. In this context, the approach does not need to re-analyze Bob's artifacts as these changes and their originated inconsistencies were already checked in the public space.

In this work, we do not focus on the consistency rule mechanism as it is handled by a separated, exchangeable mechanism, the discussion of which is omitted to uphold the readability of this work (for more information on rule evaluation please refer to Reder et al. [23]). The aspect of handling the results from the changes is mostly concerned with writing information on the correct IWS, which is implicitly decided during the analysis of these changes. Feedback regarding the consistency notification is dependent on the implementation of the respective tool adapter, which is not the focus of our contribution.

The following section discusses our approach toward the analysis of the changes performed by engineers. We also discuss how the cloud environment from Sect. 4 can be utilized during data gathering. Depending on the nature of the change, the consistency checking mechanism must react accordingly. We discuss each change type in detail and describe the respective reaction from the consistency checker's side. Furthermore, we discuss the consistency checker's tasks whenever users decide to commit their IWSs into the public space.

5.1 Data gathering by change type

Engineers may modify artifacts within their development tools. Changes made by an engineer are observed by the tool adapter and propagated to the engineer/tool's corresponding IWS. These changes are represented artifact properties/values that were added, modified or deleted. There the changes are stored as deltas with regard to the public space. These deltas represent the differences between the artifact in the public space and the changes applied in an IWS. Less straightforward is computing the consistency implications of these changes. If an artifact is part of a CRE, it means that its state have to evaluated for consistency checking. Thus, affected CREs have to be re-evaluated and the new state also needs to be stored in the IWS. However, this process varies

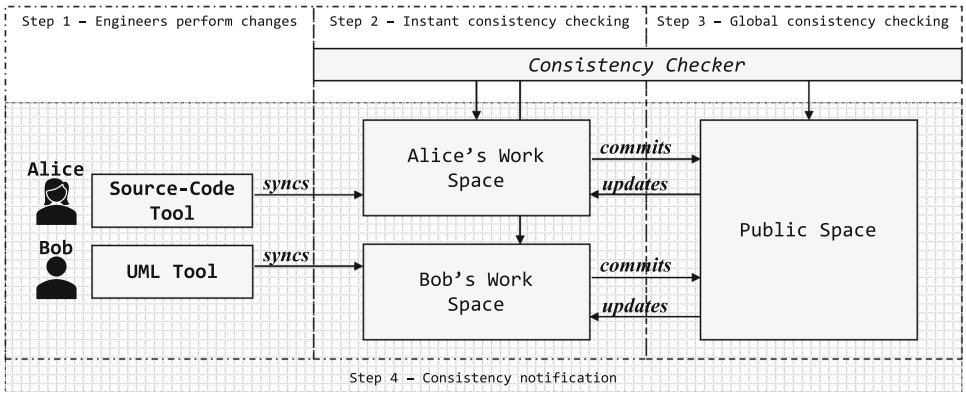


Fig. 8 The instant and global consistency checking approach

with regard to the nature of changes: `modification`, `addition`, and `deletion`. Each of the aforementioned changes triggers a notification to connected IWSs, where relevant CREs have to be evaluated as well. In the next sections, we give more details of how these types of changes are analyzed.

5.1.1 Modification

Modifications describe changes that update the value of a property in a given artifact. For instance, recall that Alice renames her Java method from "wait" to "pause" (see Fig. 2). She does this in her tool and the tool adapter will update Alice's IWS with the modified name ("pause"). Listing 7 describes the algorithm to handle modifications of artifact's properties in the IWS. The property from the artifact that was modified is added in line 2 as an artifact delta.

The syntax of this line is to be understood as follows: `IWS.`Δ`artifact` contains the delta (Δ) for all artifacts. The delta for an individual artifact with identifier i can be accessed through `IWS.`Δ`artifact[i]`. Recall that each artifact is identified by a unique id, which is utilized for this access. Furthermore, the Δ of individual properties of artifacts can be accessed by `IWS.`Δ`artifact[i].[p]`, p again being the identifier for the property. In this case, the name of a property is the unique identifier for the access of property deltas. This is possible as property names are unique in the context of an artifact.

```
1  modify(IWS, i, property, oldValue, newValue)
2      IWS.Δartifact[i].[property] = newValue
3      CREs = {CRE ∈ PS.CRE |
4                  CRE.alive == true and CRE ∉ IWS.ΔCRE}
5      CREs ∪= IWS.ΔCRE
6      for CRE ∈ CREs
7          if (i, property) ∈ CRE.scope
8              IWS.ΔCRE[CRE] = evaluate(CRE)
```

Listing 7 The modify function

An example of such an access could be `IWS.`Δ`[[Java::Class]pause].[name]`. This artifact/property does not exist in the IWS before the modification and is set to the value "pause"—thus overruling the value "wait" from the same artifact/property found in the public space. At this point, the change is only applied to the IWS, but the consistency checker is still aware of both the currently values of the IWS as well as the value from the public space. Each artifact/property may have at most one entry in `IWS.`Δ`artifact`. Should Alice later overwrite this name again (e.g., change "pause" to "stop"), then the new name would overwrite the previous change in the IWS (e.g., the current delta would hold the value "stop" instead of "pause").

For each modification, all CREs affected by the modification must be re-evaluated and the result must be persisted in `IWS.`Δ`CRE`. Similar to artifacts for tool adapters, a consistency checker has an internal language for CREs. Thus, CREs are similarly translated into the uniform representation and also uniquely identified and accessible as artifacts. Since these CREs are as well subject to version control, we must distinguish two cases: CREs that already are in `IWS.`Δ`CRE` and CREs that are in the PS. The variable `CREs` is the union of the individual CREs and the public CREs. Computing the individual CREs is straightforward because it is `IWS.`Δ`CRE` itself.

In our approach, the search for public CREs relevant to a change is simplified by the existence of backward links. When a CRE is first evaluated, it builds a scope. This scope is a set of all properties from artifacts related to that CRE. When a property is added to the scope, the CRE saves a reference to the said property. This reference can be traversed bi-directionally. So anytime there is a change, the consistency checking service checks whether the changed property has a backward link to a CRE. If so, the CRE is retrieved.

Once the CREs are retrieved, we need to re-evaluate those CREs where the scope contains the changed properties. The new evaluation result is then stored in `IWS.`Δ`CRE` (line 8), which either overwrites a previous individual result, or—if there was none—creates a new entry.

This explains the need to filter the overwritten public CREs as discussed earlier. For Alice, `IWS.`Δ`CRE` was initially empty. Of the five CREs that existed in the PS during check-out (Fig. 1), only `CRE.2.2` had a scope that included

the Java class changed by Alice. Therefore, CRE.2.2 needs to be re-evaluated, resulting in the individual CRE.2.2.A stored in her IWS. This individual CRE exists for Alice only and is superordinate to the public CRE.2.2. The public CRE.2.2 cannot be replaced physically because it corresponds to the public artifact. For example, Bob's IWS still needs to see the public CRE.2.2 as he does not see Alice's changes at this point and he has not made his changes either. The evaluation mechanism evaluate(CRE) is not discussed at this point for brevity. In this regard, interested readers may consult the work of Reder et al. [34] for more details.

5.1.2 Creation

Creations describe additions to the model. These additions can be the creation of a new artifact or the creation of properties in an existing artifacts. In either case, the new artifact and new properties need to be inserted in IWS.Δartifact (line 2 in Listing 8).

The creation of an artifact cannot cause a re-evaluation because a new artifact cannot yet be part of the scope of any CRE.

However, if a CRD exists whose context matches the type of the newly added artifact, then a CRE must be created and evaluated (lines 5–8). For example, recall that Bob adds a new [UML::Message]connect in his sequence diagram (Fig. 3). After Bob's tool adapter sends this newly created artifact, the add algorithm adds the new message to IWS.Δmodel and adds a newly instantiated CRE.1.4.B to IWS.ΔCRE—recall that a CRE is instantiated for every instance of a matching context element.

```
1  add(IWS, i)
2      for property of artifact
3          IWS.Δartifact[i].[property] =
4              artifact.[property]
5      for CR ∈ CRs
6          if cr.context == artifact.type then
7              CRE = create instance
8          IWS.ΔCRE[CRE] = evaluate(CRE)
```

Listing 8 The add function

5.1.3 Deletion

Deletion refers to the exclusion of artifacts or their properties. The deletion of an artifact (Listing 9) requires the removal of all CREs whose context elements match the deleted artifact (as opposed to creations that cause the creation of CREs). A CRE residing in the IWS can be deleted simply by removing it from IWS.ΔCRE. However, CREs residing in the public space cannot be deleted. These CREs must be flagged as "not alive." This is done by adding/modifying a CRE to IWS.ΔCRE with the alive flag being false. Regular artifacts are handled likewise. For example, should Bob delete [UML::Message]wait, then the model element and its CRE must be flagged deleted and added to the IWS.Δartifact and IWS.ΔCRE.

```
1  deletion(IWS, artifact, i)
2      CREs = ... //as defined in change(..)
3      for CRE ∈ CREs
4          if artifact ∈ CRE.scope then
5              if CRE ∉ PS.CRE then
6                  remove CRE from IWS.Δ.CRE
7              else
8                  CRE.alive = false
9      if artifact ∉ PS.artifact then
10         for property of artifact
11             remove IWS.Δartifact[i].[property]
12     else
13         IWS.Δartifact[i].alive = false
```

Listing 9 The deletion function

5.2 Committing consistency artifacts

Once users have finalized their changes, they can transfer these changes to the PS and make their work visible to other users of the cloud environment. In the cloud environment, this can be done by committing the deltas stored in an IWS into the PS, where they are appended to the current state of the corresponding engineering artifacts, respectively, their properties. an IWS generally stores all deltas of artifacts with regard to the PS, i.e., all changes an engineer performs on a publicly available artifact. Additionally, the consistency checker stores deltas on its organizational artifacts (CREs and CRDs) within an IWS, effectively documenting the current consistency state of the IWS's contents. This means, if an artifact is adapted and the change results in an altered consistency state, this alteration is stored as a delta on the respective CRE artifact result properties, within the IWS that stored the original change. This process is illustrated in Fig. 9.

When there are CRE deltas stored in an IWS, this means that IWS holds changes that would also alter the consistency state of the PS after committing. Since consistency-related artifacts are treated as regular artifacts, CRE deltas would simply be appended to the artifacts in the PS. A retrieval of CRE results from the PS would then reflect the consistency state taken over from the IWS. In a basic scenario, with only one IWS this would not pose a problem. The consistency state in the PS is correctly updated whenever changes are committed. In other words, both committed changes and consistency states are always in sync. However, if another IWS regards the same PS, committing artifacts into the PS may, in certain scenarios, require a re-evaluation of the other IWS's consistency state. Failing to do so, may result in incorrect consistency information being held by the IWSs. In case such incorrect information was committed to the PS, this would corrupt the consistency state of all IWSs. The scenarios which require re-evaluation can be categorized by the type of their committed changes (creation, modification, deletion). Their correct handling can be counter-checked by applying a set of

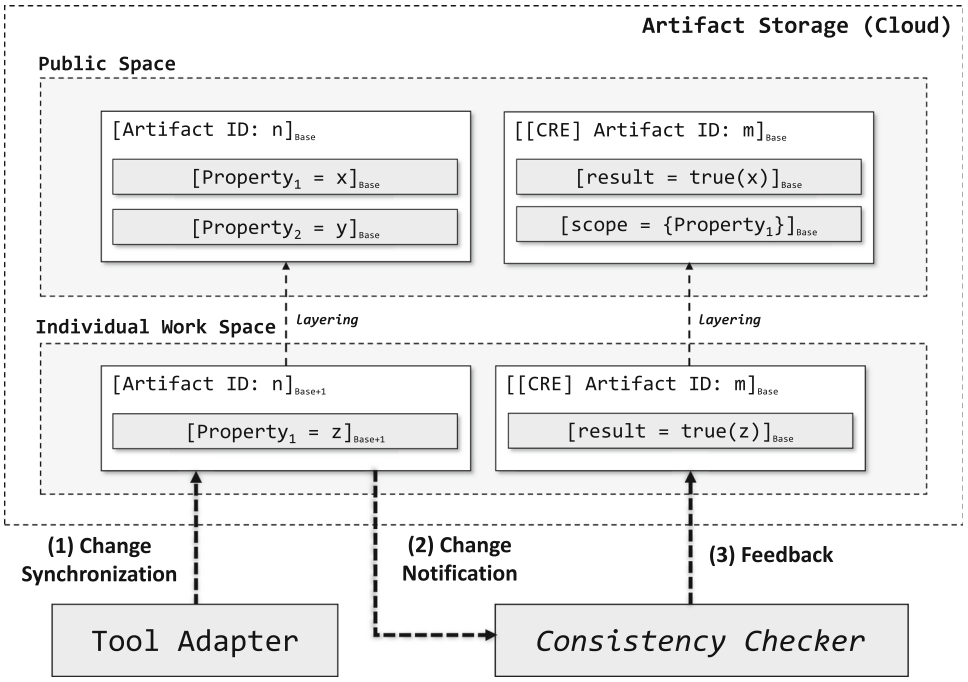


Fig. 9 Overview of CRE result deltas being stored in an IWS. This way the consistency state of the IWS's content is documented

post-conditions to the IWSs. We discuss these aspects in the following.

5.2.1 Committed creations

Scenario C1 (Context element creation with individual CRD) Assume the PS receives an artifact of a certain type. This type is defined as a context in a CRD that is only existing within an IWS. The consistency checker must now automatically instantiate and re-evaluate an according CRE within the IWS. The CRE is then committed together with the CRD (i.e., as soon as the CRD exists in the PS, all corresponding CREs appear along with it).

Scenario C2 (Scope element creation) Assume a commit contains new scope elements for a CRE. If an IWS contains its own result delta on the CRE, this result would now be outdated, since the evaluation did not consider the newly added scope element. Therefore, the CRE must be re-evaluated from the IWS's perspective.

Scenario C3 (CRD creation) Assume the PS receives a new CRD. an IWS contains context elements for this CRD. The consistency checker must automatically re-evaluate the IWS and create the corresponding CREs.

5.2.2 Committed modifications

Scenario M1 (Context element or scope element modification) Assume a commit contains deltas on existing scope or context elements of a CRE. If an IWS contains deltas on non-overlapping scope elements of the same CRE (i.e., any scope element of the CRE that was not modified in the commit), the combination of individual and newly committed deltas may lead to a different consistency result. Therefore, the CRE must be re-evaluated from the IWS's perspective. If, on the other hand, an IWS contains changes on the same context or scope elements, it must already have its own delta on the CRE result. This result may be different from the newly committed result in the PS; however, it is still corresponding to the IWSs individual view on the PS—as the individual CRE result is always layered over its public version (see Fig. 9). Therefore, this specific case would not require a re-evaluation.

Scenario M2 (CRD modification) Assume the PS modifies a CRD and the corresponding CREs. an IWS contains scope or context elements referenced by the CREs. If the rule changes, all CREs referencing the modified CRD must be re-evaluated. If the context changes, this is equal to the deletion of an old and the creation of a new CRD. The consistency checker must clean and modify the CREs accordingly. However, if an IWS contains an overlapping delta on the same CRD, this means the rule definition must be interpreted differently from the perspective of the IWS. There is no need for a re-evaluation.

5.2.3 Committed deletions

Scenario D1 (Context element or scope element deletion) Assume the PS deletes a context or scope element and the corresponding CREs. an IWS contains changes on the same context or scope elements. Subsequently the IWS must already have its own delta on the CRE result. This result, respectively, the changes on the artifacts, must overrule the deletions within the PS, since, from the perspective of the

IWS the corresponding context or scope element artifacts are still alive (see Fig. 9). A commit of these artifacts must naturally restore all artifacts within the PS. A re-evaluation is not necessary. However, if the PS only deletes non-overlapping scope elements, the modified scope in combination with the IWS's individual changes on other scope elements, may lead to a different consistency state. Therefore, this case would require a re-evaluation.

Scenario D2 (CRD deletion without overlap) Assume the PS deletes a CRD and all its corresponding CREs. an IWS contains scope or context elements. The correspondingly stored CREs must be re-evaluated, respectively, removed from the IWS in the re-evaluation process. However, if the IWS contains an overlapping modify on the CRD and corresponding CREs, the commit of the IWS restores the CREs as well as the CRD.

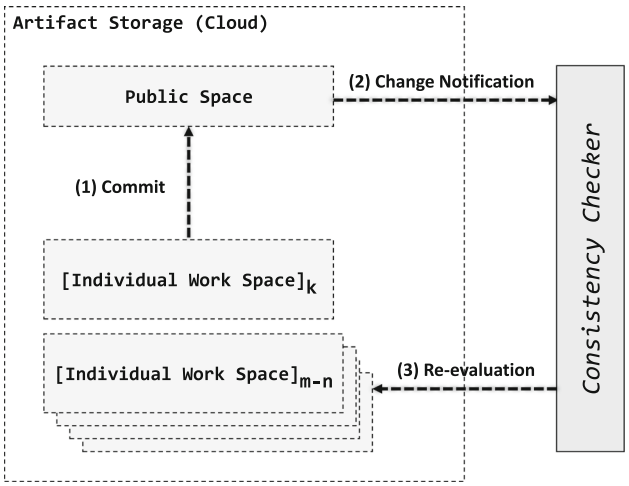


Fig. 10 Overview of an IWS (k) committing its contents to the PS, resulting in a re-evaluation of other IWSs (m–n)

5.2.4 Post-conditions

To trigger the appropriate consistency checker actions, the commit of an IWS into the PS is forwarded to the rest of the cloud environment. The consistency checker then, depending on the situation, re-evaluates the corresponding IWSs. Handling commits this way keeps consistency checking information up-to-date in all IWSs at all times. The process is illustrated in Fig. 10.

We secure this way of propagating consistency information by defining a universal post-condition in Listing 10. This post-condition must hold true for every work space after every commit.

$$
\begin{aligned}
&P = P_1 \wedge P_2 \wedge P_3 \\
&\\
&P_1 = \{\forall CRD \mid CRD \in IWS \rightarrow \ \{\forall CRE_{CRD} \mid CRE_{CRD} \in IWS\}\} \\
&\\
&P_2 = \{\forall CE \mid CE \in IWS \rightarrow \ \{\forall CRE_{CE} \mid CRE_{CE} \in IWS\}\} \\
&\\
&P_3 = \{\forall CRE \mid (CE_{CRE} \in IWS \vee CE_{CRE} \in IWS_{Parents}) \\
&\quad \wedge (CRD_{CRE} \in IWS \vee CRD_{CRE} \in IWS_{Parents})\}
\end{aligned}
$$

Listing 10 Commit post-condition

The post-condition P is a logical conjunction of three sub-conditions P_1, P_2 and P_3. These three sub-conditions can be read as follows:

- P_1: Every CRD must have all its corresponding CREs in the same work space at all times (CRD perspective)
- P_2: Every context element (CE) must have all its corresponding CREs in the same work space at all times (context element perspective)
- P_3: Every CRE must have its CRD, respectively, its context element in the same stack of layered work spaces (i.e., the CRD or context element are either in the same or a parent work space)

The CRD perspective guarantees that every newly created or modified CRD automatically creates or re-evaluates the corresponding CREs within the same work space. By keeping the CREs as tightly bound to CRDs as possible, the layered stacks of IWSs retain an individual perspective on stored engineering knowledge, while still keeping their consistency rules up to date. The context element perspective guarantees the same for the creation and modification of consistency relevant engineering artifacts. By keeping CREs tightly bound to context elements, we guarantee that every work space that is contributing to the consistency state of an engineering project, contains its corresponding consistency information. This gives the engineer using the respective IWS insight into the consistency of their created or modified artifacts, without having to commit their work into the PS. Post-condition P_3 is a weaker post-condition than the previous ones, because it is inherently given by how CREs are functionally bound to both CRDs and context elements.

Securing these post-conditions helps us to avoid late re-evaluation of engineering artifacts and to keep consistency information up-to-date in every work space at all times. In a tightly interwoven engineering project, this is a relevant task—in case of a late consistency check, ongoing work might already be in conflict with inconsistent changes and the arising problems may have carried over to other areas of the project (in our illustrative example, think, e.g., of an inconsistent method name being used in many different classes, or being overloaded several times). Such situations can easily occur with version control systems, that do not instantly propagate knowledge about inconsistent engineering artifacts. What follows are lengthy merging situations and cleaning up artifacts locally before committing them into the repository.

Reconsider the collaboration setup of Alice and Bob. Assume that both are uploading an artifact type relevant to their work. Bob is uploading a type for UML operations, whereas Alice is uploading a type for Java methods. Bob then creates a CRD for his UML operation artifacts, stating that every UML operation must be realized in an equally named Java method.

Depending on the order of commits at different stages of work, different scenarios may now unfold. Assume that Bob commits both his CRD and changes first. In this specific case, Alice's IWS would remain unaffected, as her (newly added and still individual) artifacts are not linked up to UML artifacts yet should she decide to link them up herself, this would cause a change of the UML context element, triggering a regular re-evaluation. Similarly, should Alice commit first and Bob links the artifacts, the same would happen on his side. In neither case, any of the IWSs contained any relevant artifacts, requiring an automatic re-evaluation after a commit. However, in a later stage of work, when Alice and Bob are no longer creating new artifacts, but rather adapting their already existing publicly available versions, commits must be given special consideration.

Assuming that Bob's UML artifacts are already linked up with artifacts currently being changed by Alice, a modification of his CRD would cause the scenario "M2." Should Alice commit her new changes before Bob, this would cause the scenario "M1." Should Bob only modify his UML context elements and commit, the scenario "M1" would occur from his side, and so on.

Not handling these commits in a specific way would inevitably result in conflicting consistency states. Consider a simple change like the refactoring of an operation name in UML, as outlined in our illustrative example. Bob commits the changed UML artifact (and his re-evaluated version of the related CRE) to the PS. Naturally, if Alice's linked method is no longer matching in name, the consistency state of her related CRE[9] must change. However, without automatic re-evaluation after Bob's commit, the consistency state of Alice's IWS would remain invalid, until she causes another re-evaluation of the related CREs. It is not certain that this would ever happen. If she were to commit her work immediately and without re-evaluation, she would commit both an inconsistent Java method and deprecated consistency information. This conflict situation is illustrated in Fig. 11.

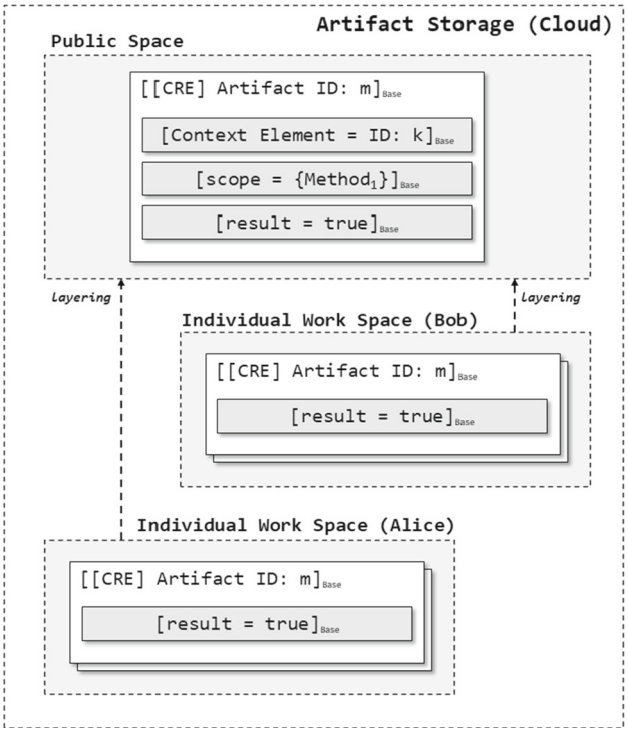


Fig. 11 Conflicting CREs in case of absent commit notifications between IWSs. If Bob commits his CRE and Alice's version is not re-evaluated, her result may become inconsistent with the publicly available versions of modified scope elements

5.2.5 Consistency checking before committing

A characteristic of our approach, which warrants being highlighted, is the possibility to check consistency of an individually changed artifact with regard to publicly available artifacts. This is mainly due to the layering of IWSs on the PS and the way how data gathering is performed by the consistency checker. Every IWS can be regarded as a specific perspective on the artifact storage. In itself an IWS only represents certain changes on artifacts. Layered on the PS it represents the full artifact storage in an altered state. When our consistency checking mechanism retrieves artifacts for analysis, it does so, for a specific IWS, i.e., from a specific perspective on the artifact storage, containing the changes of an engineer. Since the PS can constantly change, the perspective of an IWS is always up-to-date with the latest public state of artifacts. This is different from most traditional version control systems, where artifacts are checked out in a base version that remains static for the time of its editing. The consistency information results are written back into the IWS, to equip its perspective with corresponding consistency information. If the IWS is now committed, its perspective becomes part of every other IWS's perspective, requiring the re-evaluations discussed beforehand.

5.3 Advanced consistency checking

An advanced application of our approach can be implemented in the form of group-oriented consistency checking.

[9] Given that Alice is updating artifacts, which are already linked to UML artifacts, her changes must trigger the re-evaluation of related CREs via the scope, and, therefore, new results are stored in her IWS.

When a change happens within an IWS, there are, depending on the setup of work spaces, two ways our consistency checking approach can react. In the default setup, the IWS's consistency state is checked with regard to the PS. However, if a work space is in a group with other work spaces, the consistency of the change can be counter checked with regard to all grouped work spaces. For this, consistency checking relevant properties, such as the elements of a scope, are first retrieved from the IWS, then from the grouped work spaces and then from the PS. The only extended functionality of this consistency checking mechanism is the data retrieval order and how changes are layered upon each other from the perspective of the consistency checker.

Connections between work spaces can be established manually via a tool provided by the cloud environment. The grouping information can be stored alongside other artifacts within the artifact storage as well. Change notifications received by the consistency checker hold information about the work space from whose perspective consistency needs to be checked. From the said work space, grouping information can be retrieved if it exists. The consistency checking service can now modify consistency information in the following environments:

- Within the work space that performed the change
- Within all (individual) work spaces that are affected by the change

The former consistency check is similar to a simple individual consistency check, while the latter is a re-evaluation of the consistency rule, with regard to all work spaces holding a delta on related scope elements. The former consistency check is individual and only modifies CREs in one work space, while the latter is global and modifies the CREs in all grouped work spaces. As of now a group-oriented consistency checker implementation is hypothetical; however, its conceptual formulation already illustrates the wide applicability of our approach in different work situations.

6 Validation of applicability

In Sect. 5, we discussed how our approach integrates IWSs with a PS in a cloud-based environment to enable comprehensive, complete consistency checking in a multi-developer environment. While in theory this approach can solve the problems described in Sect. 2, its practical application requires further validation. This is a primary concern, as the scale of many regular industrial projects can overwhelm an experimental approach to a point where it is unusable. An approach, which is potentially handling thousands of artifacts, must scale from an algorithmic as well as a hardware-related viewpoint. Therefore, we validate the applicability of our approach, by (1) analyzing its computational complexity, (2) analyzing the memory consumption of storing the CREs, and (3) discussing its advantages in terms of memory consumption for each IWS. To put our results into perspective, we compare them to evaluation results gained from the Model/Analyzer [8,34]. The Model/Analyzer is an established consistency checker, which acts as a predecessor to our approach. However, it was never integrated in a cloud or used a cloud infrastructure to its advantage.

6.1 Prototype implementation

Our approach was implemented as a prototype[10] applying the principles discussed in this work. We built this prototype on a Java-based client/server architecture, using gRPC[11] for network communication. The inventory of tool adapters supporting DesignSpace includes Eclipse IDE, Eclipse Papyrus, Jetbrains IntelliJ, Microsoft Excel, Microsoft Visio, Eplan, Creo Elements Pro, among others, such as a simple tool for documenting various software requirements, as well as a tool for work spaces and artifact management (e.g., work space creation, linking, and definition of consistency rules). While the distribution of change information and the storage of consistency results were handled as described in this work, the concrete rule evaluation was performed by an implementation of the Model/Analyzer [23], which was integrated in the engineering environment. It should be noted that the rule evaluation part of our approach is modular and can easily be replaced by any integrated rule evaluation mechanism. An exemplary artifact synchronization, as well as the consistency checking of artifacts can be seen in the supplementary material.[12,13]

6.2 Computational complexity

The applicability of our approach is dependent on its scalability. Scalability evaluation's results can be achieved by analyzing different attributes of a system, such as memory usage, network usage, CPU usage, among others. In this work, however, we focus our scalability evaluation on the algorithm/computational complexity of our approach.

The computational complexity of consistency checking is mostly a factor of the number of necessary re-evaluations. The changes that cause the re-evaluations are `create` and `modification`—the `deletion` of model elements does not cause re-evaluations. According to Egyed [8], previous

[10] Our prototype is the DesignSpace project. Its wiki page is available at https://isse.jku.at/designspace.

[11] gRPC: https://grpc.io/.

[12] Artifact Synchronization: https://tinyurl.com/yxfwpfoy.

[13] Consistency Checking: https://tinyurl.com/y4pzx23s.

validations of the Model/Analyzer suggest that the average number of evaluations per change is between 3 and 11 (depending on the model size). Following, we denote this value as a constant c. We define the set of performed changes (which trigger re-evaluations, i.e., create and modification) by an individual engineer as $change.am$. Equation 1 defines the total number of CRE evaluations $CRE.E_{approach}$ as the sum of evaluations required for each individual engineer.

$$CRE.E_{approach} = \sum_{i=1}^{|engineer|} (c * |change.am_i|) \quad (1)$$

This equation shows that the computational complexity grows linearly with the average number of performed changes per engineer and the total number of engineers. Thus, our approach scales.

To empirically assess the overhead imposed by our approach, we re-enacted an evaluation of the Model/Analyzer using our approach. During the evaluation, we used the same UML models as described by Reder and Egyed [34]. Table 1 presents these 22 UML models, the number of model elements from each model, as well as their source (Academia or Industry). We also performed random changes in these models by simulating the changes for all model elements in each of the used models. More specifically, we captured the time required for re-evaluating all affected CREs and to persist the new result.

Each change was performed several times and the raw data for each change was recorded. The experiment was performed on a Windows 7 Professional PC with an Intel(R) Core(TM) i7-3770 CPU @ 3,4GHz and 16GB of RAM. Figure 12 depicts the evaluation times per affected CRE on average depending on the project size. We furthermore performed an ordinary least squares regression analysis[14] on the total processing time (i.e., the time it takes to find affected CREs and to re-evaluate them). The obtained results are summarized in Table 2, which shows that the model size has a significant, yet quite small effect on the evaluation time. Increasing the model size by 1000 elements leads to a total processing time increase of 0.4 ms on average. The results furthermore indicate that the number of affected CREs and evaluation times per CRE individually do not affect the total processing time. However, in combination those factors are significant determinants of the total processing time. On average, an increase in the number of affected instances by one increases the total processing time by 23ms. An increase in the evaluation time per affected instance by 1ms increases the total processing time by 1.5ms on average. Note that more than 99.9% of sample variations are explained by the analyzed factors. Comparing to previous evaluations of the Model/Analyzer [34], we obtained similar results and our framework did not slow down the evaluation times.

[14] Ordinary least squares regression is a method to estimate the relationship between one or multiple independent and a dependent variable. This method uses a coefficient of determination (R^2) to address how well the data points fit the regression line.

Table 1 UML models used in the evaluation

Name	#mes	Source
VOD paper	103	Academia
ATM	219	Industry
Microwave oven	289	Industry
ModelViewController	417	Industry
eBullition	511	Industry
Curriculum planner	762	Academia
Teleoperated robot	1113	Industry
Dice3	1272	Industry
ANTS visualizer	1280	Industry
Inventory and sales system LCA	1295	Industry
Course registration system	1404	Academia
UML model IOC F05a T12 V5.0	1451	Industry
Vacation and sick leave system	1657	Industry
Home appliance control system	1706	Industry
HDCP defect seeding system LCA	1783	Industry
DESI2.3	1972	Industry
iTalks	2211	Industry
Hotel management system	2581	Academia
Biter robocup client	2630	Industry
Word pad	8076	Academia
dSpace3.2	8759	Industry
oodt07	9826	Industry

#mes—Number of model elements

6.3 Memory consumption

The memory consumption of our approach correlates with the number of CREs that need to be maintained overall and for each engineer individually. Equation 2 defines the total number of CREs that have to be persisted in our approach ($CRE_{approach}$). The first factor describes the set of CREs needed for the PS, which is the number of CREs for a given model ($CRE(model)$). Furthermore, since an IWS stores the delta (Δ) with respect to the PS, we must consider the effect of modifications, creations, and deletions. Each of these operations adds a new entry to `IWS.ΔCRE`. Recall the average number of CREs affected by a single change c: In the worst case, each performed change adds a corresponding amount of CRE entries. Note that the number of CREs needed for an entire model grows linearly with the model size (factor $CRE(model)$). The CREs' memory consumption in the PS is thus linear with the model size. Furthermore, the CRE's memory consumption in the IWSs is again linearly depen-

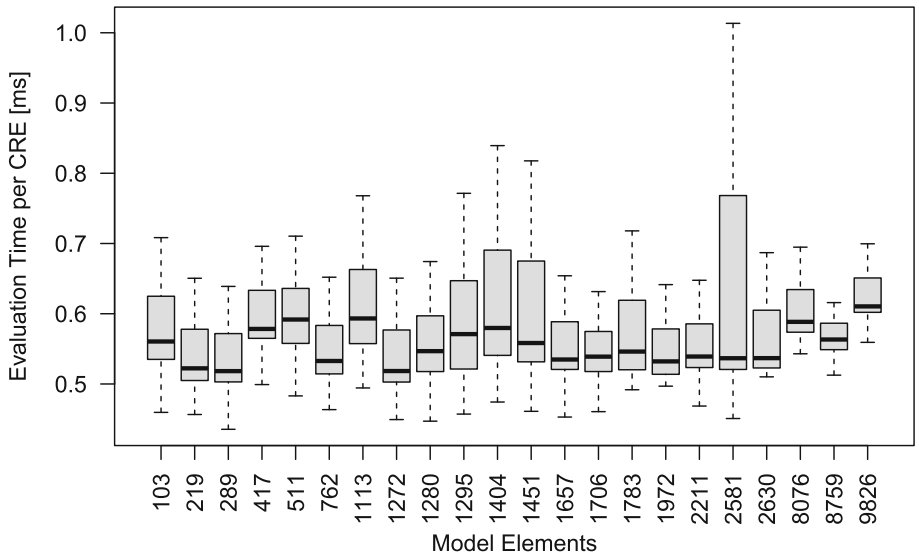


Fig. 12 Evaluation time per affected instance

dent on the number of changes per engineer and the total number of engineers.

$$CRE_{approach} = CRE(model) + \sum_{i=1}^{|engineer|} (c * |change_i|) \quad (2)$$

This equation again shows the scalability of our approach as we can see the linear growth of its computational complexity.

6.4 Version control mechanism

In this subsection, we discuss the advantages of our approach in terms of memory consumption for each individual IWS. As previously stated, an IWS stores the Δmodel and furthermore ΔCRE. We provide a behavioral analysis of how an IWS's memory footprint within the cloud environment depends on model changes compared to using the Model/-Analyzer as a plugin.

In this discussion, we use the function $M(x)$ to denote the memory consumption of a specific element x. In our approach, for both Δmodel and ΔCRE there exists an upper bound in terms of memory consumption: (i) $M(model)$, the memory it takes to store the *complete* model after the adaptations are implemented by an engineer (Eq. 3), and (ii) $M(CRE)$, the memory the consistency checker needs to store the corresponding *complete* consistency information (Eq. 4).

$$M(\Delta\texttt{model}) \leq M(model) \quad (3)$$

$$M(\Delta\texttt{CRE}) \leq M(CRE) \quad (4)$$

Note that the plugin Model/Analyzer's footprint always equals M_{SU} as defined in Eq. 5. M_{SU} describes the memory consumption of a complete consistency check where all engineering artifacts are locally available.

$$M_{SU} = M(model) + M(CRE) \quad (5)$$

Intuitively, in the worst case (i.e., if the complete model was changed) our approach is similar to the plugin Model/Analyzer as in this case our approach stores the whole model again in its entirety in the IWS and re-evaluates all CREs. Subsequently, as long as this is not the case our approach only stores a fraction of the whole model. Thus, our approach provides the advantage of $(M(model) - M(\Delta_{model})) + (M(CRE) - M(\Delta_{CRE}))$. Therefore, the amount of saved memory depends on the size of Δmodel and ΔCRE.

Equations 6 and 7 state our assumptions about both Δmodel and ΔCRE. The size of Δmodel is expressible through a function of the model size (Eq. 6) (e.g., an engineer may always change a fixed number of elements per commit). Furthermore, the size of ΔCRE is completely deter-

Table 2 Regression results for total processing time

	Total processing time (ms)
Model elements	0.0004581***
	(0.0000068)
Affected instances (AI)	−0.0017938
	(0.0074605)
Evaluation time per instance (ET)	−0.0001086
	(0.0003659)
AI*ET	1.0003651***
	(0.0002575)
Observations	48708
R^2	0.9996

Standard errors in parentheses
$^*p < 0.05$; $^{**}p < 0.01$; $^{***}p < 0.001$

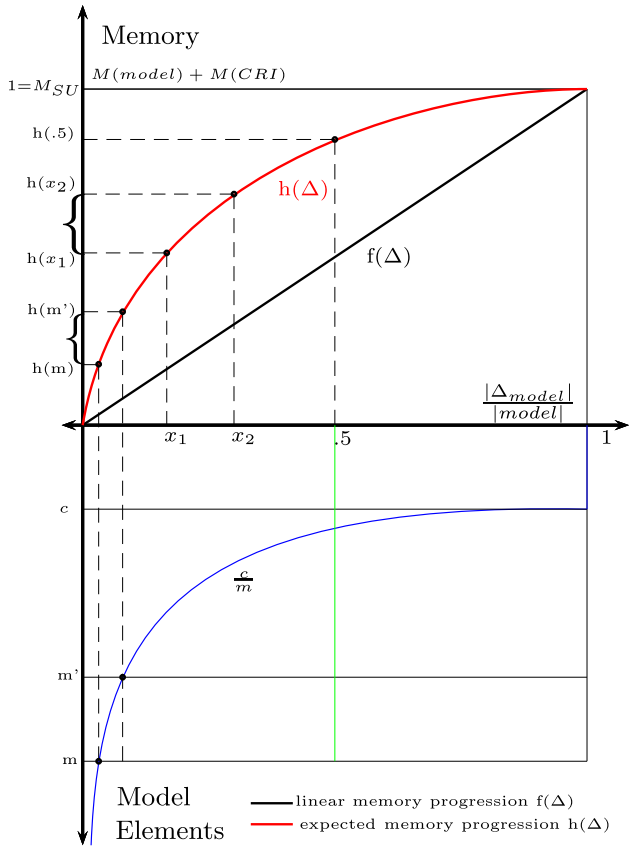


Fig. 13 Results of the memory consumption evaluation

mined by the function CRE (Eq. 7), which takes as argument $\Delta\texttt{model}$.

$$|\Delta\texttt{model}| = DeltaModel(|model|) \tag{6}$$

$$|\Delta\texttt{CRE}| = CRE(\Delta\texttt{model}) \tag{7}$$

Memory consumption of IWSs is mostly influenced by $\Delta\texttt{model}$. In Fig. 13, a behavioral analysis for possible cases of $DeltaModel$ is presented, as described next.

6.4.1 Memory consumption per distinct model elements changed

First, we discuss memory consumption per distinct changed model elements, which is plotted on the positive y-axis in Fig. 13. Values on the positive y-axis are normalized as follows:

$$(M(\Delta_{model}) + M(\Delta_{CRE}))/M_{SU}$$

This describes the ratio between actual memory consumption and possible memory consumption. The x-axis shows that the ratio of distinct model element changes to the model size (1 implies the entire model changed). Two possible assumptions for distinct elements changed are: (1) memory consumption may rise linearly with the percentage of changed elements (linear function $f(\Delta)$) or more pessimistically and (2) memory consumption may rise faster in the beginning than toward the end $h(\Delta)$. The function h represents a pessimistic case for our framework, as few changes would already lead to high memory consumption. Consider now x_1 and its corresponding memory consumption of $h(x_1)$. If an engineer changes more of the model (e.g., x_2) then the memory consumption rises to $h(x_2)$. Hence, memory is saved with regard to M_{SU} (even under a pessimistic assumptions).

6.4.2 Memory consumption depending on model size

The negative y-axis in Fig. 13 indicates model sizes in total. One can assume different functions of how much of a model is changed by an engineer. Consider now that before each commit regardless of model size a roughly constant number of elements is changed (function $\frac{c}{m}$), resulting in the memory consumption of $h(m)$. As models increase in size, the memory consumption relatively decreases. For example, if the model size increases from m' to m then memory consumption changes from $h(m')$ to $h(m)$. If we assume that the commit sizes increase with larger models then our approach is still beneficial. For example, assume that each commit changes about fifty percent of the model. In this case regardless of model size, the memory savings of our approach are—based on the assumed memory consumption—constant (e.g., 1-$h(.5)$).

6.5 Correctness

The correctness of our approach was evaluated by applying the prototype implementation into an exhaustive scenario simulation. This simulation created 2994 input scenarios manipulating the artifact storage of the cloud environment. Furthermore, these scenarios were executed in sequence and the resulting end state of the artifact storage was counter-checked against a series of post-conditions. These post-conditions defined the required number and location of CREs applied to artifacts within the IWSs after a manipulation of the artifact storage. The end state of all created input scenarios satisfied the defined post-conditions. Since the input scenarios cover a large number of potential user inputs, it is reasonable to argue that our scenario simulation evaluates the correctness of our approach in a representative way. For further information on the exhaustive scenario simulation, we invite readers to consult this paper [35].

6.6 Threats to validity

In terms of computational complexity, we demonstrated that for each engineer only a small set of CREs needs to be

stored/evaluated, i.e., the CREs that are affected by changes made by the engineers. Furthermore, the model of the version control mechanism showed that it is optimal in terms of memory consumption. The equations and models defined in the validation are based on the presented algorithms and observed behavior of the Model/Analyzer. Thus, their correctness can be inferred from the algorithms themselves. Finally, the empirical validation confirmed that our approach scales well (based on a diverse and large set of artifacts authored by different engineers). As far as the validity of the empirical study, we performed is concerned, we believe that the used models were representative. They were diverse in size and origin, i.e., they were created by different (groups of) engineers and companies. This paper does not discuss performance threats the communication overhead might pose, as this is mostly an implementation detail. Further, incrementally propagating changes from tools to the IWS leads to more communication than the occasional batch processing of changes. We ignored this in our models but we believe that this does not pose a threat to validity as a variety of cloud-based services already employ this pattern (e.g., Google Docs).

7 Limitations and future work

In the following, we discuss the limitations that our current work presents as well as possibilities for future work to address these limitations.

When dealing with artifacts from different domains, consequently, we have to deal with artifacts from different tools. Although we provide a unified representation (Fig. 5) for structuring these artifacts, some limitations regarding this approach must be considered. In this sense, the tool's internal data object can be transformed into our cloud environment internal engineering artifact as long as it can be mapped to the unified representation through a tool adapter. The usage scenarios are only limited by the applicability of such tool adapters, respectively, the applicability of the unified representation. As a result, the question of limitations is one that concerns these two forms of applicability. Technically, the applicability of tool adapters could be limited by closed APIs. If the internal data objects are unavailable, for whatever reason, the tool adapter cannot read or synchronize them into the engineering environment. One could still transform the file outputs of a tool, however, this would not allow for live synchronization. This may restrict any advanced form of collaboration between different IWSs. Also, it is likely that the file formats of a tool with a closed API are similarly proprietary and unreadable. Hence, our approach is equally limited by proprietary data formats, respectively, data structures, e.g., when a tool API is given but does not provide the required insight into the data structures to do a live synchronization. In case that all tool internal data is available and readable the approach can still be limited by the fact that the read data is unmappable to the unified representation for whatever reason. Generally speaking, this is the case when data cannot be transformed into a key/value mapping. This could, for example, be the case with extensive signal data. This, however, would be more of an inconvenience rather than a limitation, because even such data can be summarized into blocks/windows that can be mapped as values.

For investigating this problem in more detail, future work is being carried out, where we investigate how to represent models from different domains using different types of operations commonly found in version control systems [36]. These operations are used for creating these models by applying the unified representation structure.

Furthermore, the issue regarding dealing with artifacts heterogeneity becomes a matter of granularity and a question of how detailed the data must be represented on an artifact's property level. This is a matter that must be decided by the engineers. The combination of certain tools or mechanisms with the artifact structure is again a question of whether these mechanisms can be complemented with a tool adapter. The storage of artifacts is independent of the actual mechanism that tools provide. Thus, a tool, such as a code generator, could retrieve model artifacts from the engineering environment (i.e., the tool adapter would transform them from the unified representation into the tool internal data structure), apply its mechanism to the transformed engineering artifacts (which is a purely tool-internal process), and synchronize the generated code back to the engineering environment (i.e., the tool adapter transforms tool internal data structures into the unified representation—in this case, code artifacts).

The tough question in this regard is how to link the model with its corresponding code, respectively, how to handle changes in the model and correctly propagate them to the code artifacts. This is an issue concerning the propagation of model transformations, which unfortunately goes far beyond the scope of this work. For the time being, generated code artifacts would have to be replaced in full if they were regenerated from changed model artifacts. Alternatively, the new code artifacts could be synchronized as a different version of the previously synchronized code artifacts. Currently, we have ongoing work with regard to traceability generation and refinement. Our goal is to provide recommendations for engineers when creating trace links between artifacts from different domains.

Furthermore, we consider exploring new aspects of collaboration environments that may be used for improving the global consistency checking. For instance, we carried out studies regarding the hierarchical distribution [35,37] and timestamp constraints [38] of work spaces and how they can affect the consistency checking. These two aspects of the cloud environment may bring advantages to our approach.

The exploratory results showed that by considering these aspects when performing the consistency checking, we can improve the results considering both scalability and usability. We plan to conduct additional empirical evaluations considering these contributions to collect evidence about their applicability in real engineering environments.

8 Related work

Consistency checking, collaborative modeling and version control are active fields of research related to our own work. As reported in literature, however, there is a lack of approaches handling all these fields together [30]. Hence, in the following, we discuss related works classifying them by their main active field.

Global Consistency Checking These approaches address the general problem of consistency checking across possibly distributed engineering artifacts of a system. Finkelstein et al. [6] introduced consistency checking in multi perspective specifications. Each engineer owns a perspective (i.e., a *viewpoint* [3]) of the system according to his or her knowledge, responsibilities or commitments. Multiple viewpoints can describe the same design fragment, leading to overlap and the possibility of inconsistencies. The issues involved in inconsistency handling of multi perspective specifications are outlined in Finkelstein et al. [6]. An important insight for handling consistency is to allow models to be temporarily inconsistent, rather than enforcing full consistency at all times. Despite this advantage, no implementation is provided. Further, the approach does not differentiate between, a public state that is fixed (i.e., the contents of the repository) and individual modifications.

Nentwich et al. with *xlinkit* [10] present a framework for consistency checking distributed software engineering documents encoded in XML. Sabetzadeh et al. [39] presented global consistency checking by model merging. The approach focuses on handling inconsistencies between multiple models expressed in a single language. Although both approaches consider distributed models, at the time of the consistency check there is no distinction between individual adaptations and public knowledge.
Our approach largely builds on both Demuth et al. [14] as well as Troels et al. [15,16,40], which considers global consistency checking among artifacts from different engineering disciplines. Our approach extends this by considering individual work spaces, isolating the artifacts of different engineers and thus splitting up consistency information, which has to be modified on a global basis.

Consistency Checking in General Numerous approaches exist for consistency checking, specializing on specific artifact types or across artifacts [41]. Many of these approaches can also be used to check consistency across several engineering artifacts.

Finally, two approaches need to be highlighted, as these could have been replacements for the Model/Analyzer. Blanc et al. [42] look at the sequence of operations used to produce the model rather than looking at the model itself. Thus, they can not only verify structural consistency of model but also methodological consistency. Reiss presented an approach (CLIME) to incremental maintenance of software artifact [1]. This approach covers a multitude of engineering artifacts (presented are source code, models and test cases). As uniform representation information is extracted from the engineering artifact (e.g., symbol table for source code, the class diagram itself) and stored in a SQL Database. CLIME, however, does not consider any multi-developer aspects, it is required that all engineering artifacts are locally available.

Collaborative Modeling Koshima et al. [43] presented DiCoMEF, an approach for conflict detection, reconciliation and merging while collaboratively editing EMF models. Debreceni et al. [44] presents the MONDO collaboration framework, discussing how adoption of secure views work in collaborative modeling. Such views use rule-based access control on models. The authors address the protection of intellectual property across heterogeneous teams as one of their main concerns. In our work, access to engineering artifacts is controlled by work spaces. Artifacts are inherently individual until an engineer decides to commit them to the PS. Yet, individual engineering artifact changes can always be analyzed in relation to the PS. Such analysis can happen without specific security policies—as data is not exposed to team members. However, analysis provides insights that would otherwise only be available by fully integrating (i.e., exposing) an engineer's work. As a result, our approach allows heterogeneous teams to automatically analyze their individual work with regard to public knowledge, both protecting their intellectual property and enriching their work with additional meta-information (e.g. consistency information).

With Kitalpha [45], engineers are given the possibility to focus on system architecture and create workbenches for Model-Based Engineering. This involves the consideration of heterogeneous model artifacts. Similar to our cloud environment, bridges between workbenches enable the bidirectional exchange of information. However, Kitalpha puts little focus on incremental consistency checking.

Obeo,[15] including the Sirius project[16] and its commercial extensions, offer a strong focus on collaboration [46]. In these solutions, model artifacts from both native and custom tools can be integrated. Collaborative manipulation as well as conflict management are a focus of the Sirius project. However, it is largely built on an Eclipse basis. While in our

[15] Obeo: https://www.obeo.fr/en.

[16] Sirius: https://www.eclipse.org/sirius.

own approach, although we started with a similar solution, we soon concluded that such a foundation adds an additional layer of complexity. This complicates the integration of custom tools and limits the experimental applicability of our approach. Therefore, we made our approach largely independent from third-party frameworks (with the exception of network communication and back-end storage).

Version Control Version control for text-based engineering artifacts permeate software engineering and academia. Further, extensive research was conducted on version control of models, a survey is presented by Altmanninger et al. [47]. Research in the area of version control systems analyze their version controlled engineering artifacts to find inconsistencies. An example of such an approach was presented by Taentzer et al. [48]. The approach considers the abstract syntax of models as graphs. Revisions are graph modifications. Based on this, they identify two kinds of conflicts, operation-based and state-based as a result of merged graph modifications. State-based conflicts are concerned with the well-formedness, operation-based conflicts are then concerned on the parallel dependence of graph transformations and the extraction of critical pairs. Cicchetti et al. [49] proposed a meta-model for representing conflicts which can be used for specifying both syntactic as well as semantic conflicts. Finally, two popular version control systems need to be mentioned *GIT* [50] and *Apache Subversion (SVN)* [32]. Both inherently provide capabilities to create a continuous integration, during which source code checks are executed. Applying our approach to the continuous integration phase, would allow that for a commit or push to a feature branch the consistency checker would verify the impacts of the performed changes with respect to all engineering artifacts. However, it should be noted that an equivalent to the layering of IWSs would be a much more complex process in traditional version control systems. Merging conflicts between the feature branch and the remote repository are bound to arrive. Naturally any further analysis of artifacts suffers from the same complexity. The authors are not aware of any tools to extend traditional version control systems to also check consistency across multiple artifacts.

With GIT, each obtained working directory is a full-fledged repository—a `clone` copies the entire history of the repository to the working directory. Therefore, an engineer has the complete standard workflow available without being dependent on network access or a central server. To publish changes a `push` to a remote server is necessary. However, since different implementations of GIT (and extensions to the typical workflow) exists, a push may at first be pushed to a staging area, where it needs to be reviewed, before being approved and becoming part of the main trunk (e.g., Gerrit[17]).

[17] Gerrit Code Review: https://code.google.com/p/gerrit.

Characteristics Comparison Table 3 summarizes the comparison of our approach with related studies in the field. This comparison is performed based on the main characteristics of version control systems that provide consistency and collaboration mechanisms [30,51]. These characteristics are:

(i) *users*, if the approach support single-/multiple-users;
(ii) *checking type*, if the consistency checking mechanism evaluates the whole artifact at once, or does the evaluation incrementally;
(iii) *rule definition*, type of rules that are supported the consistency checking;
(iv) *artifact types*, if the approach support multiple types of artifacts or not;
(v) *extensibility*, if it is possible to extend the approach, adding new features;
(vi) *requirements*, which are the requirements to execute/apply the approach;
(vii) *working spaces*, if the approach supports public and individual work spaces;
(viii) *activeness*, whether the consistency checking and collaboration mechanisms are triggered automatically (proactive) or need user interaction (reactive);
(ix) *comparison type*, which type of comparison is performed for checking the artifacts across multiple users/domains.

The results of the comparison (Table 3) show that version control systems, such as GIT and SVN, present no rule definition for consistency checking, relying only on conflict detection. Furthermore, consistency checking specific approaches, such as xlinkit, CLIME, and the Model/Analyzer, only support one user working in the artifacts at a time, preventing collaborative work. If we consider Demuth et al.'s approach, our approach expands on it by providing the possibility for users to work in individual working spaces before making changes publicly available.

In summary, version control approaches do not address consistency checking problems, while consistency checking approaches were not designed to be used in a collaborative environment. In this sense, our proposal tackles both limitations, while allowing multiple users to collaborate during the consistency checking process.

9 Conclusion

This paper presents a novel approach to multi-tool/multi-developer consistency checking. It discusses how the highly heterogeneous nature of engineering artifacts can become problematic in terms of artifact consistency. It further discusses, how to overcome these problems by integrating the

Table 3 Comparison with related work

Approach	Users	Checking type	Rule definition	Artifact types	Ext.	Req.	Working spaces	Act.	Comparison type
xlinkit [10]	Single	Non-incremental	Own language	Multi	–	XML representation	Ind.	Rea.	Textual
Model/analyzer [23]	Single	Incremental	OCL	Single	✓	EMF	Ind.	Rea.	Model
Git [50]	Multi	Non-incremental	None	Multi	✓	None	Ind. & Pub.	Rea.	File
SVN [32]	Multi	Non-incremental	None	Multi	✓	None	Ind. & Pub.	Rea.	File
CLIME [1]	Single	Incremental	Own language	Multi	✓	SQL query	Priv.	Rea. & Pro.	Model
DiCoMEF [43]	Multi	Non-incremental	None	Single	–	EMF	Ind. & Pub.	Pro.	Model
Kitalpha [45]	Single	Non-incremental	Own language	Multi	✓	EMF	Ind.	Pro.	Textual
MONDO [44]	Multi	None	Own language	Multi	–	SVN	Ind. & Pub.	Pro.	Model
Sirius [46]	Multi	Non-incremental	None	Multi	✓	EMF	Pub.	Rea.	Model
Sabetzadeh et al. [7]	Single	Non-incremental	RML	Multi	–	Model merging	Ind.	Rea.	Model
Demuth et al. [13,14]	Multi	Incremental	OCL	Multi	✓	Tool-Adapters	Pub.	Pro.	Model
Ours	Multi	Incremental	OCL	Multi	✓	Tool-Adapters	Ind. & Pub.	Rea. & Pro.	Model

Ext.—Extensibility; Req.—Requirements; Act.—Activeness; Ind.—Individual; Pub.—Public; Rea.—Reactive; Pro.—Proactive

engineering artifacts in a cloud environment and checking consistency there.

Our approach elaborates on how to handle consistency information within the cloud environment. We explain how public and individual work spaces in the cloud can be used to store changes as well as consistency checking results. We further outline scenarios how to propagate these results and define a post-condition that must hold whenever individual work spaces commit their changes to the public space. Furthermore, the approach eliminates consistency checking redundancies to reduce the CPU usage and memory footprint to a relative constant per engineer. We demonstrate that our approach is agnostic to the size of a model, hence easily scalable.

As a future work, we plan to integrate more development tools in our cloud-based engineering environment. We also intend to investigate delta operations regarding the creation and union of models. Additionally, we plan to investigate how to improve the performance of the cloud infrastructure by using indexing to access model elements faster. Lastly, the planning and conduction of a case study in an industrial environment is part of our future work as well.

Funding Open access funding provided by Johannes Kepler University Linz.

References

1. Reiss, S.P.: Incremental maintenance of software artifacts. IEEE Trans. Softw. Eng. **32**(9), 682–697 (2006)
2. Kruchten, P.B.: The 4+1 view model of architecture. IEEE Softw. **12**(6), 42–50 (1995)
3. Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M.: Viewpoints: a framework for integrating multiple perspectives in system development. Int. J. Softw. Eng. Knowl. Eng. **2**(01), 31–57 (1992)
4. Fradet, P., Le Métayer, D., Périn, M.: Consistency checking for multiple view software architectures. In: Software Engineering—ESEC/FSE99, pp. 410–428. Springer (1999)
5. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: Future of Software Engineering, pp. 37–54 (2007). https://ieeexplore.ieee.org/document/4221611

6. Finkelstein, A.C., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency handling in multiperspective specifications. IEEE Trans. Softw. Eng. **20**(8), 569–578 (1994)
7. Sabetzadeh, M., Nejati, S., Easterbrook, S., Chechik, M.: Global consistency checking of distributed models with TReMer+. In: Proceedings of the 30th International Conference on Software Engineering, pp. 815–818. ACM (2008)
8. Egyed, A.: Automatically detecting and tracking inconsistencies in software design models. IEEE Trans. Softw. Eng. **37**(2), 188–204 (2011)
9. Vierhauser, M., Grünbacher, P., Egyed, A., Rabiser, R., Heider, W.: Flexible and scalable consistency checking on product line variability models. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 63–72, ACM (2010)
10. Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A.: xlinkit: a consistency checking and smart link generation service. ACM Trans. Internet Technol. (TOIT) **2**(2), 151–185 (2002)
11. Riedl-Ehrenleitner, M., Demuth, A., Egyed, A.: Towards model-and-code consistency checking. In: 2014 IEEE 38th Annual Computer Software and Applications Conference (COMPSAC), pp. 85–90. IEEE (2014)
12. Grundy, J.C., Hosking, J., Li, K.N., Ali, N.M., Huh, J., Li, R.L.: Generating domain-specific visual language tools from abstract visual specifications. IEEE Trans. Softw. Eng. **39**(4), 487–515 (2013)
13. Demuth, A., Riedl-Ehrenleitner, M., Nöhrer, A., Hehenberger, P., Zeman, K., Egyed, A.: DesignSpace: An infrastructure for multi-user/multi-tool engineering. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, pp. 1486–1491. ACM (2015)
14. Demuth, A., Riedl-Ehrenleitner, M., Egyed, A.: Efficient detection of inconsistencies in a multi-developer engineering environment. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 590–601. IEEE (2016)
15. Tröls, M.A., Mashkoor, A., Egyed, A.: Live and global consistency checking in a collaborative engineering environment. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, pp. 1776–1785 (2019)
16. Tröls, M.A., Mashkoor, A., Egyed, A.: Collaboratively enhanced consistency checking in a cloud-based engineering environment. In: Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS '19, (New York, NY, USA). Association for Computing Machinery (2019)
17. Tröls, M., Mashkoor, A., Demuth, A., Egyed, A.: Ensuring safe and consistent coengineering of cyber-physical production systems: a case study. J. Softw. Evol. Process **33**(9), e2308 (2021). (**e2308 smr.2308**)
18. Tröls, M.A., Mashkoor, A., Egyed, A.: Instant distribution of consistency-relevant change information in a hierarchical multi-developer engineering environment. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing, SAC '21, (New York, NY, USA), pp. 1572–1575. Association for Computing Machinery (2021)
19. Egyed, A.: Instant consistency checking for the UML. In: Proceedings of the 28th International Conference on Software Engineering, pp. 381–390. ACM (2006)
20. Richters, M., Gogolla, M.: OCL: Syntax, Semantics, and Tools, pp. 42–68. Springer, Berlin (2002)
21. Xu, C., Cheung, S.-C., Chan, W.-K.: Incremental consistency checking for pervasive context. In: Proceedings of the 28th International Conference on Software Engineering, pp. 292–301. ACM (2006)
22. Blanc, X., Mougenot, A., Mounier, I., Mens, T.: Incremental detection of model inconsistencies based on model operations. In: International Conference on Advanced Information Systems Engineering, pp. 32–46. Springer (2009)
23. Reder, A., Egyed, A.: Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 347–348. ACM (2010)
24. Nentwich, C., Emmerich, W., Finkelstein, A., Ellmer, E.: Flexible consistency checking. ACM Trans. Softw. Eng. Methodol. (TOSEM) **12**(1), 28–63 (2003)
25. Beyer, D.: Relational programming with CrocoPat. In: Proceedings of the 28th International Conference on Software Engineering, pp. 807–810. ACM (2006)
26. Sun, C., Xia, S., Sun, D., Chen, D., Shen, H., Cai, W.: Transparent adaptation of single-user applications for multi-user real-time collaboration. ACM Trans. Comput. Hum. Interact. **13**, 531–582 (2006)
27. Beckett, D., McBride, B.: RDF/XML syntax specification (revised). W3C recommendation, vol. 10, no. 2.3, (2004)
28. Koegel, M., Helming, J.: EMFStore: a model repository for EMF models. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Vol. 2, pp. 307–308. ACM (2010)
29. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: eclipse modeling framework. Pearson Education (2008)
30. Torres, W., Van den Brand, M.G., Serebrenik, A.: A systematic literature review of cross-domain model consistency checking by model management tools. Softw. Syst. Model. **20**(3), 897–916 (2021)
31. Ball, A., Ding, L., Patel, M.: An approach to accessing product data across system and software revisions. Adv. Eng. Inform. **22**(2), 222–235 (2008)
32. Pilato, C.M., Collins-Sussman, B., Fitzpatrick, B.W.: Version Control with Subversion: Next Generation Open Source Version Control. O'Reilly Media, Inc. (2008)
33. Ghabi, A., Egyed, A.: Exploiting traceability uncertainty among artifacts and code. J. Syst. Softw. **108**, 178–192 (2015)
34. Reder, A., Egyed, A.: Incremental consistency checking for complex design rules and larger model changes. In: International Conference on Model Driven Engineering Languages and Systems, pp. 202–218. Springer (2012)
35. Tröls, M.A., Mashkoor, A., Egyed, A.: Hierarchical distribution of consistency-relevant changes in a collaborative engineering environment. In: 2021 IEEE/ACM Joint 15th International Conference on Software and System Processes (ICSSP) and 16th ACM/IEEE International Conference on Global Software Engineering (ICGSE), pp. 83–93 (2021)
36. Alanen, M., Porres, I.: Difference and union of models. In: Stevens, P., Whittle, J., Booch, G. (eds.)$\ll UML \gg$ 2003—The Unified Modeling Language. Modeling Languages and Applications, pp. 2–17. Springer, Berlin (2003)
37. Tröls, M.A., Mashkoor, A., Egyed, A.: Team-oriented consistency checking of heterogeneous engineering artifacts. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 250–251 (2021)
38. Tröls, M.A., Mashkoor, A., Egyed, A.: Timestamp-based consistency checking of collaboratively developed engineering artifacts. In: ICSSP'21: Proceedings of the International Conference on Software and System Processes (2021)
39. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S., Chechik, M.: Consistency checking of conceptual models via model merging. In: Requirements Engineering Conference, 2007. RE'07. 15th IEEE International, pp. 221–230. IEEE (2007)
40. Tröls, M.A., Mashkoor, A., Egyed, A.: Multifaceted consistency checking of collaborative engineering artifacts. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 278–287. IEEE (2019)

41. Lucas, F.J., Molina, F., Toval, A.: A systematic review of UML model consistency management. Inf. Softw. Technol. **51**(12), 1631–1645 (2009)
42. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: ACM/IEEE 30th International Conference on Software Engineering, 2008. ICSE'08, pp. 511–520. IEEE (2008)
43. Koshima, A.A., Englebert, V.: Collaborative editing of EMF/Ecore meta-models and models: conflict detection, reconciliation, and merging in DiCoMEF. Sci. Comput. Program. **113**, 3–28 (2015)
44. Debreceni, C., Bergmann, G., Ráth, I., Varró, D.: Secure views for collaborative modeling. IEEE Softw. **35**(6), 32–38 (2018)
45. Langlois, B., Exertier, D., Zendagui, B.: Development of modelling frameworks and viewpoints with Kitalpha. In: Proceedings of the 14th Workshop on Domain-Specific Modeling, pp. 19–22 (2014)
46. Viyović, V., Maksimović, M., Perisić, B.: Sirius: a rapid development of DSM graphical editor. In: IEEE 18th International Conference on Intelligent Engineering Systems INES 2014, pp. 233–238 (2014)
47. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. Int. J. Web Inf. Syst. **5**(3), 271–304 (2009)
48. Taentzer, G., Ermel, C., Langer, P., Wimmer, M.: Conflict detection for model versioning based on graph modifications. In: International Conference on Graph Transformation, pp. 171–186. Springer (2010)
49. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing model conflicts in distributed development. In: International Conference on Model Driven Engineering Languages and Systems, pp. 311–325. Springer (2008)
50. Loeliger, J.: Version control with Git: powerful techniques for centralized and distributed project management. O'Reilly (2009)
51. Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G.: Towards a taxonomy of software change. J. Softw. Maint. Evol. Res. Pract. **17**(5), 309–332 (2005)

Michael Alexander Tröls received his PhD in Computer Science at the Johannes Kepler University in Linz, Austria. From 2016 to 2021 he worked at the Institute of Software Systems Engineering (ISSE). The focus of his research is on consistency checking within collaborative engineering environments. He is currently self-employed as a Software Developer.

Luciano Marchezan is a PhD student at the Institute of Software Systems Engineering (ISSE) at the Johannes Kepler University Austria, supervised by Prof. Dr. Alexander Egyed. He received his master degree in Software Engineering from the Federal University of Pampa (Unipampa-Brazil). His research interests include Model-Driven Software Engineering, Automated Software Engineering, Software Reuse and Empirical Software Engineering.

Atif Mashkoor is a senior research scientist at the Johannes Kepler University, Austria. He is also the founding managing director of Sino-Pak Center for AI @ Pak-Austria Fachhochschule: Institute of Applied Sciences & Technology, Pakistan. He holds a doctoral degree from the University of Lorraine, France, and a master's degree from the Umeå University, Sweden, both in computer science. Additionally, he has studied computational linguistics at the Rovira i Virgili University, Spain. He can be reached at atif.mashkoor@jku.atlspcai.paf-iast.edu.pk.

Alexander Egyed is a professor for software-intensive systems and heads the Institute of Software Systems Engineering (ISSE) at the Johannes Kepler University, Austria. He received his doctorate from the University of Southern California, USA. He previously worked many years in the industry before joining academia. Dr. Egyed was recognized among the Top 10 scholars in software engineering, and his work has received numerous awards. He can be reached at alexander.egyed@jku.at.