Hierarchical Distribution of Consistency-relevant Changes in a Collaborative Engineering Environment

1st Michael Alexander Tröls  
Johannes Kepler University, Linz, Austria  
michael.troels@jku.at

2nd Atif Mashkoor  
Johannes Kepler University, Linz, Austria  
atif.mashkoor@jku.at

3rd Alexander Egyed  
Johannes Kepler University, Linz, Austria  
alexander.egyed@jku.at

Abstract—Engineering is a collaborative process, incorporating a multitude of heterogeneous artifacts. These artifacts share interdependent properties, which must be kept consistent during the engineering process (e.g., code must correspond to architectural design documents). To address this circumstance, existing works propose the unification of heterogeneous artifacts in a single collaborative engineering environment where artifacts are analysed for consistency on the basis of their interdependent properties. The work presented in this paper expands the collaboration possibilities of such environments by adopting hierarchically organized work areas storing artifact changes. We discuss different implications such hierarchies have on the computation of consistency information and propose a mechanism of instantly distributing consistency-relevant change information throughout the collaborative engineering environment. This way, engineers are provided with a unique perspective on their engineering artifacts, which is immediately re-evaluated whenever new changes are made within the hierarchy. To evaluate this mechanism, we provide an experiment – motivating the instant distribution of change information – as well as a scenario simulation – testing our mechanism against an exhaustive set of possible inputs.

Index Terms—collaboration, engineering artifacts, consistency checking

I. INTRODUCTION

In modern engineering projects, many experts from various fields contribute a multitude of different engineering artifacts. These engineering artifacts, such as requirements, specifications or code, are highly interdependent. If such engineering artifacts change, all dependent artifacts must change with them. We refer to maintaining the consistency between heterogeneous, interdependent engineering artifacts as global consistency checking. This form of consistency checking can become very complex since many artifacts, which may not be intuitively related, are often implicitly dependent on each other. For example, specifications may have an impact on designs, which, in turn, may have an impact on code.

The research reported in this paper has been partly funded by the Austrian Science Fund (FWF) grants (P31989-N31 & I4744-N), the Austrian COMET K1-Centre Pro2Future and the Austrian COMET K2-Centre LCM funded by the Austrian Research Promotion Agency (FFG) in cooperation with the Austrian ministries BMVIT and BMDW and the province of Upper Austria, and the LIT Secure and Correct Systems Lab supported by the province of Upper Austria.

Various works have proposed ways of unifying engineering artifacts in collaborative engineering environments (CEE) (e.g., [20]–[23]). In such a CEE, engineering artifacts are synchronized from their respective tools and stored in a syntactically uniform representation. Contrary to currently established collaboration systems (e.g., versioning software like Git\(^1\) or SVN\(^2\)) this allows the engineers to natively store heterogeneous engineering artifacts. On this foundation, the artifact’s properties can be analysed by a consistency checking mechanism – making global consistency checking feasible.

However, a problem arises with regards to the iterative workflows common in modern engineering projects. A CEE with a single global artifact storage may be fit for integrating large milestones of a project and checking them in intervals of several weeks or months, but global consistency checking must meet the requirements of continuous integration. This requires the continuous analysis of the engineers’ artifact changes throughout the engineering process.

The work presented in this paper expands the concept of global consistency checking within a CEE by further differentiating the artifact storage and elaborating on the mechanisms necessary to keep consistency therein. We distinguish between public artifacts, which are generally available to engineers in a common repository, and private artifact changes that are adopted within separate, private storages. We define a change as a creation, update or removal of an artifact’s properties with regards to its public state. We organize private storages in a hierarchical structure, where the child’s changes always overrule the parent’s changes (respectively the public state of an artifact, which acts as a hierarchy root).

Both public artifacts and hierarchically ordered private changes in combination result in a specific perspective on the artifact storage. Every level in this hierarchy reveals a certain state of artifacts corresponding to a certain set of changes. Every set of changes has its own implications on the consistency of artifacts. We compute the consistency information concerning these changes before they are committed through

\(^1\)Git: https://git-scm.com
\(^2\)SVN: https://subversion.apache.org
the hierarchy. This requires us to continuously distribute all consistency-relevant change information (i.e., all changes that may affect the consistency state of a perspective) down the hierarchy for potential consistency re-evaluations, as some perspectives may become out-of-date. This distribution mechanism is the core novelty of this work. It contributes the possibility to provide consistency information concerning the perspectives on the artifact storage. An instant distribution is required, so engineers can continuously check their changes in relation to a collaborative state of engineering artifacts (stored in a higher level such as the public area) before they start merging their work.

The rest of this paper is structured as follows: In Section II, we describe a typical consistency checking scenario. In Section III, we outline the issues arising within the illustrative example. In Section IV, we present the data model of the CEE as the technical foundation for our mechanism of instantly distributing change information. The proposed mechanism is then discussed in Section V. We evaluate our work with an experiment and an exhaustive scenario simulation, which are discussed in Section VI. The related work is discussed in Section VII. The paper is concluded in Section VIII.

II. ILLUSTRATIVE EXAMPLE

To demonstrate the functionality and benefit of hierarchical work areas with instant distribution of consistency information, this section outlines an illustrative example. Assume the following scenario: A company is developing and deploying robot arms for the maintenance lines of various factories. Within this setup, three engineers: Alice, Bob and Carol, are working on separately developed but interdependent engineering artifacts. Alice is creating UML activity diagrams that guide the implementation of robot logic while Bob is setting up and maintaining hardware specifications for the entire engineering department. In his line of work, Bob is also responsible for maintaining various specification documents some of which are relevant to Alice’s work. Carol is implementing and maintaining the code based on the results of both Alice’s and Bob’s work. All engineers are using a CEE, which records individual artifact changes. The hierarchical order of work areas sees both Alice’s and Bob’s work joined in a private work area, directly subordinate to the public area. Carol’s private work area acts as an equal sibling to this joined private work area. This way, Alice’s and Bob’s committed changes are first considered privately before they become visible (public) to Carol. This setup is illustrated in Figure 1.

Imagine now the following adaptions: Bob receives a new hardware specification for the latest version of a robot arm. He does his necessary calculations and updates the corresponding specifications. He synchronizes his changes with the CEE as depicted in step 1 of Figure 1. In the meantime, Alice is working on an activity diagram which references hardware-related constants, as specified by Bob.

Bob commits his work from his private work area to the joined work area (step 2). At this point, his changes are only visible to him and Alice. They are not yet visible to Carol. Since Alice’s diagram still contains reference to the old operational distance of the robotic arm, she must now realize that her work is inconsistent. The corresponding consistency check is triggered instantly as the consistency-relevant change information (Bob’s adaptation of the specified operational distance of the robotic arm) becomes visible from Alice’s perspective on the artifact storage (step 3). Alice, now aware of Bob’s changes, can adapt her work (step 4), before committing it (step 5). The sooner she is aware of the changes, the less effort it will take to integrate them. Otherwise, at the time of the commit the inconsistency may have propagated into other parts of her work and may require substantial refactoring.

With Alice’s and Bob’s consistent changes committed to the joint work area, the said work area can be committed to the public area (step 6). After this commit, Carol will be aware of the new specification and the changed activity diagram (step 7). She can now adapt her implementation accordingly (step 8) and commit her work to the public area (step 9).

Alternatively, we could set this scenario up without the hierarchical structure of private work areas and immediately commit all changes into the public area; however, this may cause problems. Had either Alice or Bob committed an inconsistent state of their work, the issue would have been propagated into Carol’s work.

III. PROBLEM STATEMENT

Due to the delayed propagation of consistency information, various problems may arise.

- **Merging delays**: In many collaboration systems, conflict identification is tackled when the work of different engineers is brought together (e.g., while merging branches on Git). If a conflict is detected, the merging process is unnecessarily delayed. This may lead to organizational problems, as well as hasty resolutions or workarounds.

- **Extensive refactoring**: To tackle problems during merging, engineers must often carefully refactor portions of their work. This is a particular problem if conflicts are deeply rooted in the work that is about to be merged.

- **Tolerated inconsistencies**: During the engineering process, some temporary inconsistencies may be tolerated (e.g., between designs and implementation to guarantee a flexible workflow) [18]. However, in many cases the documentation of these inconsistencies is lacking. This potentially leads to flaws that are – if at all – recognized much later, which makes them harder to correct [19].

These problems could be avoided through the instant propagation of consistency information, gathered from consistency checks between heterogeneous artifacts. Delays during the merging phase can be countered through a simplified preparation of engineering artifacts, while they are still worked on. Counter-checked against the work of other engineers, engineering artifacts can be adapted to match before the merging process is started. Furthermore, they can be adapted in an ongoing manner, immediately when an inconsistency with the work of another engineer arises. This saves the engineers work with regards to refactoring. Tolerated inconsistencies can
be kept in check as consistency checking results are stored as a part of the propagated consistency information.

IV. DATA MODEL

In this work, we use a CEE to store engineering artifacts from different engineering tools. These tools are equipped with custom tool adapters, which synchronize the artifacts with the environment. Once synchronized, artifacts are stored in a unified representation. This representation can be analyzed, read or modified by engineers or supporting services such as a consistency checker (an exemplary synchronization can be seen in the supplementary material respectively online3).

Throughout the process of storing and analyzing engineering artifacts, the CEE provides a syntactic common ground. Its data model helps us implementing the instant distribution of consistency-relevant change information. In the following, we will re-iterate the data model’s elements, which can be seen in Figure 2. We distinguish between Artifact Storage Elements – data model elements concerned with the storage of engineering artifact information, Artifact Declaration Elements – data model elements concerned with the declaration of an artifacts structure, and Consistency Checking Elements – data model elements with both declarative and information storage purpose.

A. Artifact Storage Elements

Artifact Storage Elements are responsible for the storage of engineering artifact information. They represent concrete work results of the engineering process, respectively changes performed on fine-grained aspects thereof.

- **Artifacts:** An artifact represents a certain work result achieved during the engineering process. Such a work result could, for example, be code or requirements. To guarantee a certain structure, artifacts adhere to a certain type (e.g., Java Class Artifact or Requirements Artifact). Depending on the type, artifacts consist of specific properties.
- **Property:** Properties represent fine-grained aspects of a work result. Such fine-grained aspects may, for example, be methods within code or names of requirements.

3Exemplary synchronization: https://tinyurl.com/y7pny6ao
A property’s current value is determined by the latest change on the respective aspect. Earlier changes are part of a property’s change history. As such, the properties of an artifact describe an incrementally built sum of all changes on an engineer’s work results. Properties must adhere to a field declared within an artifact’s type.

- **Change**: A change represents a certain value of a named property at a certain time. It contains a timestamp as well as a change type – a property can either be created, updated or deleted. A change is always denoted as the product of exactly one work area (i.e., its origin). Every incremental change triggers a change event that is propagated through the engineering environment and read by potential listeners, e.g., a consistency checking mechanism.

- **Work area**: Work areas contain changes, which refer to properties, which, in turn, form artifacts. The latest change value represents the current value of a property. Work areas are organized in a hierarchy. The lowest work areas in this hierarchy only store the changes performed by a single, respective engineer. Only the engineer that created the work area can see its contents; therefore, we refer to it as a “Private Work Area” (PWA). When an engineer decides to make his or her changes visible to other engineers, these changes are committed into a higher work area. Every lower work area can access the contents of its parent. Subsequently, the contents of the highest work area are publicly available to every engineer. We refer to this work area as the “Public Area” (PA). Contrary to PWAs, which only store an engineer’s individual changes on properties, the PA contains the entire incremental change history of a property. If a PWA commits a property change, the change is added to the existing changes in its parent and eventually the PA. Every PWA must have exactly one parent. The advantage of this approach lies within the fact that every engineer, creating a PWA, can retrieve a certain individual perspective on the artifact storage. This perspective is dependent on the changes stored within the PWA hierarchy and the full artifact representation in the PA. It should be noted that changes in a PWA always overrule current changes in the PWA’s parents, respectively the PA. If, for example, we regard the simple work area hierarchy within Figure 3, we can see that for work area “PWA1” the artifact with ID 78 would retrieve “0.75m” for its property “length” whereas work area “PWA2” would retrieve “1.00m”. From the perspective of work area “PWA2” both “name” and “variant” have been changed. For “PWA1” these properties still appear as “Robot Arm” and “1.74_A”. The full representation of the artifact with ID 78 is different from the perspective of both PWA’s, respectively the tool adapters that utilize them. Further, work areas have a certain timestamp. The timestamp denotes the time of the work areas creation.

---

**Fig. 3.** An overview of an artifact as represented in the context of a work area hierarchy.

---

**B. Artifact Declaration Elements**

Artifact Declaration Elements are concerned with the definition of an artifact’s structure. They are defined by users through the tool adapters and can be instantiated as concrete artifacts.

- **Type**: A type defines the structure of an artifact in the form of fields. Types are automatically instantiated as artifacts when the tool adapters synchronize engineering artifacts to the CEE.

- **Field**: A field is part of a type and describes a property of an artifact. It defines the property’s primitive data type as well as its cardinality.

**C. Consistency Checking Elements**

Basic consistency checking in the cloud environment is realized as a service running in parallel to engineers modifying the artifact storage. It analyzes changes on engineering artifacts and notifies the respective engineer who issued the change. Consistency checking is always performed from the perspective of a certain work area. The checks themselves happen according to certain user-defined consistency rules. The consistency checker requires two kinds of information, which are stored as artifacts as well:

- **Consistency Rule Definition artifacts (CRD)**: These artifacts contain a defined rule for a certain artifact type (the “context” of a rule). For example, a rule definition for Java Class artifacts might define a naming convention on the classes’ methods. These methods may be stored as separate, linked artifacts. The defined rule, a simple string, then formalizes a way to navigate through the artifact storage from the class artifact to the method artifacts and their name properties. Consistency Rule Definitions are stored as artifacts so they can easily be retrieved from the perspective of a Consistency Rule Evaluation artifact.

- **Consistency Rule Evaluation artifacts (CRE)**: These artifacts realize a Consistency Rule Definition for a certain artifact instance (the “Context Element” (CE)). They contain a scope, which is a collection of all properties navigated through during a rule evaluation.
For the Java Class example, the scope would contain every property linking between the class artifact instance and the method artifact instances as well as the name properties of the method artifact instances. Should any of these scoped properties change, a re-evaluation of the respective Consistency Rule Evaluation artifact is triggered. Re-evaluation results are treated like any other artifact property. They can change on a work area basis. Keeping this information in the artifact storage, rather than in the inner logic of the consistency checker, has the advantage that it can easily be read by other services and re-used for further analysis.

As mentioned above, consistency rules are written as simple strings defining a navigation path through the artifact structure. In fact, they are a set of expressions connected through operations, which are executed on whatever values stand at the end of the navigation path. An operation could, for example, be a simple “equals”. Furthermore, these operations can be connected in conjunctions. Formalized, a rule can be expressed as follows:

\[
\text{Conjunction(Operator(\text{Expression}_A, \text{Expression}_B)^{1...})}
\]

A single expression is written in an object-access like manner, denoting properties of the artifacts that are navigated. For example, the expression

\[
\text{self.type.name}
\]

would navigate from a certain artifact instance (“self”) through a link to it’s type and retrieve the name of the said type artifact. In its basic implementation, the core functionality of the consistency checker is activated for the PA of the cloud engineering environment. It listens to artifact change events, which trigger the following normal course:

1) **Change analysis:** If the change concerns the creation of an artifact for whose type there is a CRD, the consistency checker automatically instantiates a corresponding CRE and continues with the “Data Gathering” process. If the change concerns a property that is part of a scope, the respective consistency rules must be re-evaluated. This is done by retrieving the associated CRE, which also initiates the “Data Gathering” process. All changes leading to this process are considered consistency-relevant.

2) **Data gathering:** During the data gathering process, the consistency checker first retrieves the concrete CRD through the CRE (see data model in Figure 2). From the CRD, the consistency rule is retrieved. The consistency checker then navigates through the artifact structure as it is described within the rule that is re-evaluated. Encountered values are stored for the evaluation.

3) **Rule evaluation:** During the evaluation, the expressions of the consistency rule are substituted with the stored values. The encapsulating operations are executed and the final logical conjunction is constructed.

4) **Storing results:** The final result is stored as a change on the CRE’s result property.

5) **Providing consistency feedback:** The consistency checking mechanism notifies the initiator of the change about the results. Writing the result into the CRE is another change event in the artifact storage, which can be handled by further mechanisms for additional analysis.

The computation of consistency information can be expanded to changes stored within a PWA. In practice, this means that certain changes in a PWA potentially trigger a consistency rule re-evaluation. Within this re-evaluation, the change is considered on top of whatever information is available in the PA. A new consistency checking result is computed and stored as a change in the PWA that triggered the re-evaluation. This effectively means that engineers can check their changes’ consistency with publicly available artifacts from the isolated perspective of a PWA. This has the major advantage that consistency information, necessary for the merging process of engineering artifacts, is available to engineers before the merging process is started. Furthermore, the consistency information is re-evaluated on every change and therefore always immediately available. An exemplary demonstration of consistency checking in the CEE can be seen in the supplementary material^4.

V. DISTRIBUTION OF CHANGE INFORMATION

Expanding global consistency checking to hierarchies of PWAs requires additional coordination in case of a commit. When an artifact is committed to a parent work area (PWA or PA), the consistency information in multiple child PWAs potentially becomes outdated. Our mechanism secures that consistency information is kept up-to-date at all times and that committed changes lead to according re-evaluations of affected PWAs. This requires special consideration of the commit procedure, respectively the distribution of change information, upon which certain re-evaluations must be triggered. As the core aspect of our mechanism, we discuss this commit procedure in the following.

A. Setup

Consistency information is stored next to regular artifact changes (in the form of CREs and CRDs). Therefore, a commit also transfers the PWA’s consistency information into the parent work area and overwrites the respective artifact values that may already reside there. This way, consistency information can be read, stored and expanded independently from the actual artifacts it refers to. However, this comes with the requirement that the consistency information of a PWA must be completely up-to-date at commits. Otherwise incorrect results may be published and the parent’s consistency information is no longer in sync with the artifacts it refers to. In our mechanism, this requirement is secured continuously after every change registered on the parent work area. Committing does not require a preliminary consistency check between the parent and the committing PWA.

^4Consistency Checking in the CEE: https://tinyurl.com/y4pz23s
B. Committing Changes

The process of committing changes can be split into different stages:

- Issuing a commit through the tool adapter
- Collecting and removing all changes from the PWA
- Transferring the collected changes to the respective properties in the parent work area
- Distributing change information to the rest of the engineering environment
- Consistency checker and other services reacting towards change information

When engineers wish to publish their work in the CEE, they must transfer their changes from their PWA to its parents, respectively the PA. From the perspective of the individual engineer, this means that they must issue a "commit" command on their PWA. This can be done through the CEE’s API, respectively the tool adapters using the said API. Generally commits can be done in two ways:

- Continuously: Changes are immediately committed to the parent work area once they are synchronized with the PWA. When changes are continuously checked out as well, this allows for live collaboration between tools that normally do not support such a workflow.
- Periodically: Changes are published in intervals. This way of committing the PWA is similar to traditional version control systems. While continuous commits are an automated process, periodic commits can happen manually.

How commits are issued within a tool is dependent on the implementation of the respective tool adapter.

When the tool adapter receives a "commit" command, it forwards the command to the CEE. The latest changes stored in the connected PWA must now be transferred to the parent work area. There, the cloud automatically appends every change to its corresponding property. In case of the PA, these changes eventually form a fine-grained change history for artifacts, respectively their properties.

Since both the PA and the hierarchy of PWAs form an engineer’s perspective on the artifact storage, new changes in a PWA’s parents require a re-evaluation within the PWA to keep this perspective’s consistency information correct.

The goal of our mechanism is to keep all consistency information in the CEE up-to-date at all times. If a commit happens, the committed consistency information must naturally be up-to-date as well. For this, we continuously re-evaluate a PWA’s consistency information, whenever a) something in the PWA changes (e.g., a tool synchronizes a change), or b) something in the PWA’s parent changes. A change in the parent can only happen whenever a commit is issued by a child PWA or a service modifies the contents of the parent. The consistency information of other child PWAs can then become out-dated. This can happen in the following scenarios SC1-7 (supporting illustrations for the scenarios can be found in the supplementary material available online):

- SC1 – Committed CRD creation: If a commit adds a new CRD to a parent work area, all child PWAs carrying changes on potential CEs are now outdated. Any artifact that "turns" into a CE is consistency-relevant and must always be accompanied by a corresponding CRE result. Therefore, when a new CRD is committed into the parent, such a CRE result must be added in the affected children. If the CE already existed in the parent before the commit, the corresponding CRE was created and committed together with the CRD. If the CE only exists within the child PWA, the corresponding CRE must be created in the said PWA.
- SC2 – Committed CE creation: If a regular artifact is committed to a parent work area, this artifact is not consistency-relevant. However, if a child PWA contains an uncommitted CRD referring to the type of this artifact, the consistency information of the respective child PWA is now incomplete. From the perspective of the child PWA, there is an artifact that requires a CRE. The corresponding CRE must be created in the child PWA to be committed with the CRD later (leading to SC 1).
- SC3 – Committed Scope Element creation: When a child PWA contains a CRE referencing a set of scope elements and the said set is manipulated in the parent, naturally the consistency result of the CRE is outdated. Subsequently, every newly committed scope element must trigger a re-evaluation of the corresponding CREs.
- SC4 – Committed CRD update: A CRD can be updated in two ways. Either the rule is updated or the context. In the earlier case, all corresponding CREs in child PWAs must be re-evaluated. If the context is updated, our mechanism treats it like the deletion of an old and the creation of a new CRD (i.e., first SC 6, then SC 1).
- SC5 – Committed CE or Scope Element update: When a context or a scope element is updated in a parent work area during a commit, a child PWA's consistency information can become outdated. This is up to two requirements: Firstly, the PWA must contain a corresponding CRE. Secondly, the PWA must not contain changes on the same context or scope element. If the latter is the case, these changes would have already produced their own consistency results, which overrule any of the parent’s consistency information.
- SC6 – Committed CRD deletion: When a CRD is deleted within a parent work area, all corresponding CREs, residing within the parent or child PWA’s, must be deleted as well. The only exception are child PWAs containing an update on the CRD. In this case, the updated CRD overrules the deleted CRD. Within such a PWA, correspondingly updated CREs are already present. None of these CREs are deleted.
- SC7 – Committed CE or Scope Element deletion: If a CE or a scope element is deleted in a parent work area during a commit, a child PWA’s consistency information is potentially outdated. For this to happen, the PWA must contain a CRE referring to either the scope or the CE.

5Supporting illustrations: https://tinyurl.com/y93lo8bk
this is the case, the corresponding CRE must be deleted (after CE deletion) or re-evaluated (after scope element deletion). The child PWA must not contain a change on the deleted scope or CE. Otherwise, its changes overrule the parent’s deletion.

To consider these scenarios, our mechanism makes use of a commit event fired after changes are appended in a work area. This commit event contains all transferred change information and is distributed to all child PWAs, except the one from which the commit originated. From the perspective of these PWAs the consistency checking mechanism can now interpret the distributed change information and act accordingly. From the scenarios above, we derive a set of post-conditions (P1-3), which must be fulfilled by the consistency checker:

- **P1**: At all times a CRD must be accompanied by all its corresponding CREs (i.e., all CREs must always exist in the same work area as the CRD). If a CRD is newly created in a PWA, this work area must also contain all CREs of CEs in its parents. This must be the case because these CEs are part of the PWA’s perspective on the artifact storage. Without the CREs the consistency information of the PWA would be incomplete. Naturally, when a CRD is committed, its CREs are committed alongside.

- **P2**: A CE must always be accompanied by its corresponding CREs. The exception are CEs that are not yet CEs from the perspective of the work area they are in. When a CE is part of a parent work area, but the CRD is in a child PWA, the corresponding CRE would reside in the said PWA. From the perspective of this PWA the potential CE is already treated as such. The CE only “publicly” becomes a CE once the CRD is committed, after which it is immediately accompanied by the corresponding CRE.

- **P3**: A CRD, as well as a CE, must always be retrievable from the work area in which a corresponding CRE exists. If there is a CRE in a child PWA (e.g., because of a changed scope element), the corresponding CE and CRD, must either reside in the same PWA, or in its hierarchy of parents.

With these post-conditions defined, our consistency checking mechanism can check all changes from the perspective of PWAs. Doing so, the mechanism filters consistency-relevant change information on a PWA basis, i.e., only if a change is actually relevant to the consistency information within the work area, the respective CREs are created or re-evaluated. This selective approach is done to avoid a full re-evaluation of all work areas after every commit. A depiction of the resulting algorithm can be seen in Listing 1.

```
switch(change, ChangeType)
  case CREATE:
    if (CRD is created)
      foreach (CE in perspective)
        if (corresponding CRE is NOT element of perspective)
          create CRE for CE
        else
          retrieve CRE for CE
    elseif (CE is created)
      foreach (context—corresponding CRD in perspective)
        if (corresponding CRE is NOT element of
```

Listing 1. Algorithm on the basis of which the consistency checking mechanism handles change information from the perspective of a PWA.

Since this algorithm is generally applicable to changes being observed from a certain PWA’s perspective, it is also valid for a regular change synchronization from a tool adapter (PWA changes). In other words, this algorithm can entirely replace the legacy event handling that was originally used in the CEE.

VI. EVALUATION

The evaluation of our mechanism is split into an experimental and a practical part. For the experimental part, we conducted an empirical study to illustrate the importance of the presented problems and to highlight the necessity for an automated solution. For the practical part, we checked an implementation of our mechanism against an exhaustive set of simulated input scenarios. These scenarios automatically test possible situations in which our mechanism must react. The results are checked against a pre-defined post-condition.

A. Empirical Case Study

In order to demonstrate the applicability of our mechanism and to underline the relevance of the addressed issue, we conducted an empirical study on the impact of instant inconsistency feedback during the engineering process [13]. The study was conducted with 36 bachelor-level computer science students at an mean age of 23.7 years. The average programming experience was two years. All subjects were familiar with Java and UML (one or more years of experience). The results from these subjects could be generalized for junior engineers. The subjects were given ten tasks each. The tasks were focused on the co-changing of code and UML models within two non-trivial systems. The first system was...
an open source game named “Matador”, featuring roughly 6000 lines of code across 37 Java classes. The code was documented by six UML diagrams with 661 model elements. The second system was a proprietary calendar application named “Calendarium”, featuring roughly 21000 lines of code across 150 Java classes. These were documented in 12 UML diagrams with 2843 model elements. The systems were chosen due to their size and the availability of both models and code. The goal was to reach a consistent state between UML models and code after certain changes were performed on one or the other. The results were graded as “correct” (all required changes were performed), “partially correct” (at least half of the required changes were performed) or “incorrect” (less than half of the required changes were performed). One half of the given task descriptions provided consistency-relevant change information, while the other one did not. We found that, given the change information, subjects were much more likely to solve their individual tasks correctly. The improvement of correctness in tasks with consistency information over those without was at a factor of 2.68 (see Table I - note that the discrepancy between the total amounts of tasks was the result of an odd number of tasks finished by the participants). This means that the success rate of solving tasks was more than twice as high than without consistency information. Furthermore, the amount of incorrect solutions was 51.5 percent lower when consistency information was present. There was no significant difference in partially correct solutions. This study gives us confidence that the instant distribution of consistency-relevant change information can drastically improve the engineering process and lower the risk of costly errors. It is therefore reasonable to assume that engineers can benefit greatly from our proposed mechanism.

**B. Exhaustive Scenario Simulation**

To validate the correctness of our mechanism – respectively the subsequent propagation of consistency information – we exhaustively simulate possible input scenarios that manipulate the artifact storage of the CEE. The results of these scenarios are counter-checked against a post-condition (as defined in Section V), which secures the correctness of the outcome. The evaluation is conducted on a CEE implementation with the features presented in Section IV.

1) **Procedure:** For this evaluation, we define an input scenario as a combination of valid input sequences a user can perform. These sequences are limited in the following ways:

- The user interface determines which inputs a user can enter. The possible inputs are a commit, the creation of an artifact, the modification of a property and the removal of an artifact (where removal is just a further modification of the “alive” property of an artifact).
- Certain inputs require specific preceding inputs. For example, an artifact can only be updated after creation. This leaves us with a set of rules that is determinant of valid input sequences.
- The consistency checking mechanism only reacts towards changes in consistency checking relevant artifacts – inputs irrelevant to consistency checking can be ignored.
- The propagation of consistency information is always handled in pairs of work areas, that is one child and one parent work area. If a parent has multiple children, propagation is handled for each child individually (i.e., again on the basis of one child and one parent).

The resulting input sequences are overlapping (e.g., sequence A creates an artifact while sequence B creates an artifact and then modifies it) and can be summarized in a graph structure. Such a graph can be traversed exhaustively. Each step (in combination with previous steps) of the traversal then represents a valid sequence of user inputs. These input sequences can be combined, exhausting all possible input sequences, respectively input scenarios. The limitations for input sequences, the generation of the graph as well as its traversal are discussed as following.

2) **Limitations:** To narrow down the amount of potential input sequences, this evaluation only considers user-manipulated artifact types relevant to the consistency checking mechanism. Such types are CRDs, CEs and Scope Elements. These artifacts can either be created, modified or committed, thus narrowing down the amount of potential input sequences further. It is important to note here that our evaluation considers the modification and the removal of an artifact equivalently.

Further, sequences of inputs underlie certain rules. They define which inputs must precede other inputs, so the sequence is valid. Two rules are critical in this regard. Firstly, the modification of an artifact is always preceded by the creation of the said artifact. Secondly, an artifact must be committed to be modified by a parent work area. Subsequently, the creation of a CRD must always precede its modification. The same is true for the CEs and Scope Elements. A valid input sequence could therefore look like the following:

\{Create: CE (child); Commit (child); Modify: CE (parent);\}

Given that a CRD exists, the mechanism must react with the creation of a CRE once the CE is created in the child work area. After committing, both CE and CRE are transferred to the parent work area. The modification of the CE must then again produce a CRE re-evaluation from the perspective of the parent work area. The result is then stored as a change on the previously committed CRE.

3) **Graph Building and Traversal:** With the limitations and rules for inputs defined, we can build a graph of valid input sequences. The input sequence from above could originate

<table>
<thead>
<tr>
<th>Correctness</th>
<th>Information</th>
<th>No Information</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>59 (41.84%)</td>
<td>22 (15.71%)</td>
<td>268%</td>
</tr>
<tr>
<td>Partially Correct</td>
<td>49 (34.75%)</td>
<td>50 (35.71%)</td>
<td>0%</td>
</tr>
<tr>
<td>Incorrect</td>
<td>33 (23.4%)</td>
<td>68 (48.57%)</td>
<td>-51.5%</td>
</tr>
<tr>
<td>Tasks</td>
<td>141 (100%)</td>
<td>140 (100%)</td>
<td></td>
</tr>
</tbody>
</table>
from a graph structure as depicted in Figure 4. The nodes and edges in orange represent the exemplary input sequence. Note that every consecutive step of this sequence can also represent its own valid input sequence. Further, note that this graph structure is only a small example. A full graph considers all possible children for every individual node, leading into their own valid sequences. This means, “Create:CE” could be followed up by “Modify:CE” and then “Commit”. Such a sequence would be represented in its own sub-graph, branching off “Create:CE”. This is done to avoid a cyclic graph structure. Repetitions of the same node are not necessary since only the position of a node in the input sequence makes a difference for the consistency checking mechanism. Variations of these positions are already exhaustively covered by the combined input sequences.

4) Input Scenario Extraction and Combination: With a full graph structure built, we traverse the graph and extract every consecutive step in the traversal as its own valid input sequence. All combinations of different input sequences give us all input scenarios under the given limitations.

C. Counter-checking and Results

The input scenarios can now be executed automatically by sequentially forwarding every input of each scenario to the CEE. After every input we counter-check the state of the artifact storage against a post-condition. This tells us whether the consistency information, created through various inputs, is propagated correctly throughout the CEE. In total, we produce 2994 input scenarios. The end state of all these input scenarios satisfies the defined post-condition.

D. Threat to Validity

A scenario simulation may not pose a real-world utilization of our mechanism. However, in an exhaustive scenario simulation real-world input scenarios merely represent a subset of all scenarios. An exhaustive simulation poses as a holistic evaluation of the functionality of our mechanism and includes all input scenarios a regular user could perform. This is due to the fact that the set of possible inputs is determined by a limited interface. The combination of the inputs in a graph and its full traversal therefore results in a full coverage of potential user actions. We can, therefore, state that an exhaustive scenario simulation evaluates the correctness of our work in a representative way.

VII. RELATED WORK

Collaborative engineering approaches: Various works have been conducted in the field of collaborative engineering (e.g., [4], [8]–[11], [15], [17], [25]). The literature review of Bruneliet et al. [1] focuses on approaches that concern themselves with the unification, presentation and manipulation of different models within a singular environment.

Similar to the one presented in this paper, other works discuss the type structure of their underlying artifact basis (in the sense, whether meta-models can be unified within a single view), the management of consistency at run-time, and the possibilities to manipulate the artifact basis. Most notably EMF Views [2] Kitalpha [16], Model Join [3], and Viatra [6] share similarity with our work, since they all consider multiple meta-models (or engineering artifact types) within a single view, while giving the possibility to augment the model- respectively artifact-basis with enhancing information. Out of these approaches, only Viatra considers an incremental re-computation of consistency as our work does. For this, the Viatra framework performs model queries and the respective re-evaluations on a change basis. However, the framework does not natively implement additional infrastructure for a cloud execution of its features. As a result, the multi-user aspect of our work is not given within Viatra. An instant distribution of change information would therefore have to be handled in a different way, for example a separate infrastructure. Similar disparities can be identified with the Vitruvius [14] approach and ModelJoin, which use a combination of meta-models to establish flexible views on engineering artifacts. The latter of these approaches allows the presentation of heterogeneous artifacts in a non-intrusive way, by combining legacy meta-models into a single underlying model. In our work this operation is equivalently taken over by the tool adapters and can happen during runtime.

EMF Views takes a similar approach to the integration of different engineering artifacts. In this approach, meta-models are synchronized into a viewpoint, respectively a corresponding combination of several meta-models. The synchronized artifacts can then be complemented by links, which are themselves instances of a separate weaving model, i.e., a linking meta-model. Our work can equivalently produce links as engineering artifacts. These links are then described by corresponding artifact types. Such types can be customized at runtime. Beyond these similarities, incremental consistency checking has not been a major focus of EMF Views.

Kitalpha is a framework focused on system architecture and the creation of workbenches for Model-Based Engineering. Subsequently, the presentation of heterogeneous engineering artifacts plays a major role in this framework. Bridges between workbenches enable the bidirectional exchange of information. However, there is little focus on incremental consistency checks and the distribution of computed information towards stakeholders using workbenches.
An approach with stronger focus on collaboration is developed by Obeo\(^6\). This includes the Sirius project\(^7\) as well as commercial extensions built on top of it. These solutions offer the integration of engineering artifacts from both native and custom tools. Collaborative editing as well as conflict management between different engineering artifacts are all considered aspects of the Sirius project. However, it is largely built on an eclipse basis. This adds an additional layer of complexity with regards to the integration of custom tools in the collaboration environment.

Further, both Herzig et al. [12] and Vangheluwe [24] have focused on the unified representation of heterogeneous engineering artifacts. Both approaches rely on the application of directed graphs. These graphs represent artifacts through a triple consisting of an object, an attribute and a value. This is similar to our own work, where artifacts would be equivalent to objects, properties to attributes. Values would be expressed through references between artifacts. The direction of edges in a graph could be realized through link artifacts with corresponding properties. This also illustrates the versatility of a mapping based structure, as additional meta-information can be appended to regular engineering artifacts.

In their work Dávid et al. [5] explore the management of inconsistencies in virtual products by process enactment (execution and monitoring of processes) and tool interoperability. This work focuses less on the distinction between the private perspective of a stakeholder onto the end product and the public representation thereof. The distribution of consistency-relevant change information among stakeholders is only discussed very briefly.

Further research conducted by Egyed et al. [7] propose a unification of artifacts in a cloud environment for the purpose of analysis and manipulation. One such form of analysis is incremental consistency checking, which is studied in the context of an industrial experiment. However, these works do not further explore the intricacies of distributing change information and reacting accordingly from the perspective of individual work areas.

**Version control tools:** The concepts discussed in this work depend on the capabilities of the CEE. However, it should be noted, that any of these concepts can be transferred to other version-control tools, given that the respective APIs allow the implementation of the required plugins. Going by the example of Git, the branch of a repository contains the changes of one or many developers. This can be treated as an equivalent to the PWA and consistency checking can be performed on it globally, as long as the branch is a full representation of all artifacts in a project. However, this is not the case usually. This is due to the fact, that Git normally does not store heterogeneous engineering artifacts. It is mostly made for text-based artifacts such as code or XML files. This also has an effect on the applied mechanism when a branch is merged back into the master branch. Git applies a line-based merging mechanism, which sometimes requires manual corrections, while the CEE applies a property-based merging mechanism, where the last committed change is always the valid one. In principle these differences are not in conflict with the presented concepts on the distribution of consistency information. If one were to implement the presented concepts in Git, it would be possible and the described scenarios would still apply. When an artifact is added to the Git branch (e.g., in the form of a class file) it is treated as a CE, which requires the creation of a CRE within the branch (e.g., as an entry in a CRE file). If a CRD is added, CREs for all CEs are required, etc. The implementation would be different; however, the concepts stay the same. This makes our work not only applicable to a CEE, but also to various repository-based artifact storages.

**VIII. Conclusion**

In this work, we illustrate the problematic nature of collaborative engineering. We discuss issues arising with a lack of incremental consistency checking mechanisms and how delayed consistency checking information may lead to extensive refactoring phases. As a solution, we present how consistency checking can be realized within a CEE. We discuss a way of keeping consistency information up-to-date with the latest published state of engineering artifacts. The instant distribution of consistency information plays a critical role in this problem area. This is especially true when engineers commit changed engineering artifacts to a public repository, while concurrently modifying its contents in a hierarchy of private work areas. We illustrate different scenarios, which lead to out-of-date consistency information and present a way of circumventing this problem. In our proposed mechanism, a set of post-conditions must be fulfilled for every work area within the CEE after every commit. These post-conditions are realized within a presented algorithmic solution. We demonstrate the efficacy of our work with an experimental study and evaluate its correctness on the basis of exhaustively generated test scenarios. The paper is concluded after a discussion on related approaches. For future work, we would like to check consistency within groups of work areas, independently from any work area hierarchy. To achieve this, we plan to expand the data gathering mechanism of the CEE’s consistency checker.

**References**


---

\(^6\)Obeo: https://www.obeo.fr/en

\(^7\)Sirius: https://www.eclipse.org/sirius


