Instant Distribution of Consistency-relevant Change Information in a Hierarchical Multi- Developer Engineering Environment

Doctoral Thesis
to obtain the academic degree of
Doktor der technischen Wissenschaften
in the Doctoral Program
Technische Wissenschaften

Submitted by
Dipl.-Ing. Michael
Alexander Tröls, BSc

Submitted at
Institute for Software
Systems Engineering

Supervisor and
First Evaluator
Univ.-Prof. Dr. Alexan-
der Egyed

Second Evaluator
Univ.-Prof. Dr. Ruth
Breu

Co-Supervisor
Dr. Atif Mashkoor

February 2021

Abstract

Nowadays engineering solutions are integrated with almost every aspect of society. Digitisation is opening up possibilities for affordable workflow optimizations in many different professional industries. Engineering solutions play an increasingly important role in medical care, agriculture, manufacturing, logistics, public services, banking and other critical fields.

As a side effect of this development, engineering projects can no longer be finished by a team of technical experts alone. Modern engineering solutions require the input of multiple experts from multiple disciplines. The input of these experts is encapsulated in the results of their respective work. We refer to these results as *engineering artifacts*. Such artifacts may cover a design document contributed by a requirements engineer, UML class diagrams by a software engineer, mechanical specification documents by a mechanical engineer, code written by programmers, etc.

Naturally, this results in a multitude of heterogeneous artifacts that have to be maintained during the development process. Consequentially, engineering is becoming a multidisciplinary field, respectively an interface between traditional industries and the possibilities of modern digital technology.

This makes it necessary to consider engineering artifacts in the context of their dependencies with each other. While heterogeneous artifacts are often syntactically different, their semantic overlaps imply interdependencies that cannot be ignored during the engineering process. If an engineering artifact changes, all dependent artifacts must change with it. For example, if a UML diagram changes, the corresponding implementation must change as well. Failing to propagate these changes, causes inconsistencies between engineering artifacts. With the increasing scale of engineering projects, the identification of such inconsistencies becomes highly complex. The longer inconsistencies prevail within a project, the further they propagate throughout other related artifacts. The later an inconsistency is identified the more expensive it becomes to fix.

This work proposes the immediate propagation of consistency-relevant change information concerning the interdependent properties of heterogeneous engineering artifacts. To achieve this, we conceptualize and develop an artifact storage system that can overcome the heterogeneity of engineering artifacts and relate them on the basis of their semantic overlaps. For this, we develop a dedicated data representation and utilize a consistency checking mechanism that is able to analyse the said representation. The computed consistency information is subsequently propagated throughout the artifact storage and distributed to all affected users.

Kurzfassung

Moderne Ingenieursansätze durchdringen in der heutigen Zeit nahezu jeden Aspekt unserer Gesellschaft. Die Digitalisierung eröffnet Gewerben und Industrie zahlreiche Möglichkeiten zur Verbesserung ihrer Arbeitsprozesse. Medizin, Landwirtschaft, Produktion, Logistik, öffentliche Betriebe, Banken und andere bedeutende Institutionen unseres täglichen Lebens greifen immer häufiger auf integrierte Ingenieursansätze zurück.

Als Nebeneffekt dieser Entwicklung können moderne Ingenieursprojekte nicht mehr ausschließlich von technischen Experten fertiggestellt werden. Moderne Ingenieursansätze erfordern den Beitrag verschiedenster Experten aus einer Vielzahl an Disziplinen. Der Beitrag dieser Experten ergibt sich aus den konkreten, produktiven Resultaten ihres jeweiligen Arbeitsfeldes. Wir sprechen im Bezug auf diese Resultate von sogenannten *Ingenieursartefakten*. Diese Artefakte umfassen beispielsweise die Design Dokumente eines Requirements-Ingenieurs, die UML Klassendiagramme eines Software-Ingenieurs, die Spezifikationen eines Mechatronikers, den Programmcode eines Programmierers, usw.

Dies führt natürlicherweise zu einer Vielzahl an heterogenen Artefakten, die im Entwicklungsprozess eines Ingenieursprojekts Beachtung finden müssen. Als Konsequenz dieser Tatsache erweist sich das Ingenieurwesen immer mehr als multi-disziplinäres Feld beziehungsweise, als eine Schnittstelle zwischen klassischen Industrien und den Möglichkeiten der modernen Digitaltechnologie.

Dieser Umstand macht es notwendig, Ingenieursartefakte im Bezug auf ihre Abhängigkeiten zueinander zu betrachten. Während heterogene Artefakte oft syntaktisch unterschiedlich sind, implizieren ihre semantischen Überschneidungen eine Reihe von Wechselwirkungen, welche während des Ingenieursprozess nicht außer Acht gelassen werden dürfen. Ändert sich ein Ingenieursartefakt, so müssen sich alle in Wechselwirkung stehendend Artefakte ebenso ändern. Wird beispielsweise ein UML-Diagramm angepasst, so muss die darin beschriebene Implementierung genauso angepasst werden.

Werden entsprechende Änderungen nicht korrekt propagiert, können Inkonsistenzen zwischen Ingenieursartefakten auftreten. Durch den wachsenden Umfang moderner Ingenieursprojekte wird die Identifikation solcher Inkonsistenzen zunehmend schwieriger. Umso länger eine Inkonsistenz in einem Projekt verbleibt, umso tiefer propagieren sich ihre Auswirkungen durch verwandte Artefakte. Umso später eine Inkonsistenz erkannt wird, umso teurer wird ihre Reparatur.

Um diesen Problemen vorzubeugen, präsentiert diese Dissertation eine Methode zur unmittelbaren Propagierung von Konsistenz-relevanten Informationen. Diese Informationen beschreiben die Veränderung von in Beziehung stehenden, heterogenen Ingenieursartefakten. Zur Verwirklichung dieses Ansatzes konzipieren und entwickeln wir ein System zur Speicherung von Artefakten, welches deren Heterogenität handhaben und sie auf Basis ihrer semantischen Überschneidungen in Relation setzen kann. Wir entwickeln dazu ein Modell der Datenrepräsentation für heterogene Ingenieursartefakte. Diese Form der Datenrepräsentation wird von einem Konsistenzprüfungsmechanismus erfasst, welcher infolgedessen die entsprechende Konsistenz-relevante Information erzeugt. Die erzeugte Information wird daraufhin an alle betroffenen Benutzer des Speichersystems weitergeleitet.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

A handwritten signature in blue ink, consisting of a stylized 'M' followed by a horizontal line and a small upward stroke.

Michael Alexander Tröls, 1. Februar 2021

Contents

I	Prologue	1
1	Introduction	3
1.1	Outline	4
2	Problem Definition	5
2.1	Engineering Artifacts	5
2.2	Inconsistencies	6
2.3	Consistency Management	6
3	Approach	9
3.1	Overview	9
3.2	Running Example	10
4	Paper A: An Empirical Study on the Impact of Inconsistency Feedback during Model and Code Co-changing	15
4.1	Discussion	15
4.2	Results	16
4.3	Contributions & Extension	16
5	Paper B: Collaboratively Enhanced Consistency Checking in a Cloud-based Engineering Environment	17
5.1	Discussion	17
5.2	Results	19
5.3	Contributions & Extension	19
6	Paper C: Live and Global Consistency Checking in a Collaborative Engineering Environment	21
6.1	Discussion	21
6.2	Results	22
6.3	Contributions & Extension	23
7	Paper D: Multifaceted Consistency Checking of Collaborative Engineering Artifacts	25
7.1	Discussion	25
7.2	Results	26
7.3	Contributions & Extension	27
8	Paper E: Timestamp-based Consistency Checking of Collaboratively Developed Engineering Artifacts	29
8.1	Discussion	29
8.2	Results	30
8.3	Contributions & Extension	31

9	Paper F: Instant Distribution of Consistency-relevant Change Information in a Hierarchical Multi-Developer Engineering Environment	33
9.1	Discussion	33
9.2	Results	35
9.3	Contributions & Extension	35
10	Paper G: Ensuring Safe and Consistent Co-Engineering of Cyber-Physical Production Systems: A Case Study	37
10.1	Discussion	37
10.2	Results	38
10.3	Contributions & Extension	39
11	Conclusion & Outlook	41
II	Papers	43
A	An Empirical Study on the Impact of Inconsistency Feedback during Model and Code Co-changing	45
1	Introduction	46
2	Related Work	47
3	Method	48
4	Subjects	49
4.1	Independent Variables	49
4.2	Dependent Variables	51
4.3	Experiment Variations	51
4.4	Experiment Tool	52
4.5	Procedure and Material	52
5	Analysis and Results	52
5.1	Research Questions	52
5.2	Overall Analysis	53
5.3	Research Question 1	54
5.4	Research Question 2	55
5.5	Research Question 3	57
6	Threats to Validity	58
6.1	External Validity	58
6.2	Internal Validity	60
6.3	Construct Validity	60
6.4	Conclusion Validity	61
7	Conclusion	61
8	Acknowledgement	62
B	Collaboratively Enhanced Consistency Checking in a Cloud-based Engineering Environment	63
1	Introduction	64
2	The DesignSpace Engineering Cloud	64
2.1	Uniform Data Representation	65
2.2	Live Feedback	65
2.3	Artifact Typing	66
2.4	Linking	66
2.5	Private Work Areas	66
2.6	Collaboration Services	66
3	Consistency Checking	66

3.1	Basic Architecture & Runtime Functionality	67
3.2	Collaborative Enhancement	68
4	Evaluation	70
4.1	Prototype Implementations	70
4.2	Experiments & Case Studies	71
5	Conclusion & Future Work	72
6	Acknowledgement	72
C	Live and Global Consistency Checking in a Collaborative Engineering Environment	73
1	Introduction	74
2	Problem Illustration	75
2.1	Initial State	75
2.2	Subsequent Changes Made by Engineers	75
2.3	Problem Statement	76
3	Infrastructure	77
3.1	Overview	77
3.2	Cloud Environment	77
3.3	Linking Engineering Artifacts	80
4	Global Consistency Checking	81
4.1	Data Structure	81
4.2	Evaluation Process	82
4.3	Inconsistency Feedback	84
4.4	Example	84
5	Validation	87
5.1	Existing Tool Adapters	87
5.2	Links	87
5.3	Empirical Study	87
5.4	Case Studies	88
5.5	Threats to Validity	89
6	Related Work	89
7	Conclusion	90
8	Acknowledgement	91
D	Multifaceted Consistency Checking of Collaborative Engineering Artifacts	93
1	Introduction	94
2	Problem Statement	94
3	Multifaceted Consistency Checking	97
3.1	Cloud Deployment	97
3.2	Global Constraints	97
3.3	Live Capabilities	98
3.4	Context Views	98
3.5	Group Orientation	99
4	Realization	99
4.1	Uniform Data Representation	99
4.2	Linking	100
4.3	Artifact Storage	100
4.4	Live Services	101
4.5	Consistency Checking	102
5	Discussion	106
6	Related Work	107

7	Conclusion & Future Work	109
8	Acknowledgements	110
E Timestamp-based Consistency Checking of Collaboratively Developed Engineering Artifacts		111
1	Introduction	112
2	Illustrative Example	113
3	Problem Statement	115
4	Technical Prerequisites	115
4.1	Uniform Data Representation	115
4.2	Private Work Areas and Public Area	116
4.3	Consistency Checking in a Cloud-based CEE	116
5	Timestamp-based Consistency Checking	117
5.1	Timestamp Configurations	118
5.2	Consistency Checking Mechanism	120
6	Evaluation	122
6.1	Model Data	123
6.2	Procedure	123
6.3	Discussion	124
6.4	Threats to Validity	125
7	Related Work	126
8	Conclusion	127
F Instant Distribution of Consistency-relevant Change Information in a Hierarchical Multi-developer Engineering Environment		129
1	Introduction	130
2	Illustrative Example	131
3	Problem Statement	132
4	Data Model	133
4.1	Artifact Storage Elements	133
4.2	Artifact Declaration Elements	134
4.3	Consistency Checking Elements	135
5	Distribution of Change Information	137
5.1	Setup	137
5.2	Committing Changes	137
6	Evaluation	141
6.1	Empirical Case Study	141
6.2	Exhaustive Scenario Simulation	142
6.3	Counter-checking and Results	143
6.4	Threat to Validity	144
7	Related Work	144
8	Conclusion	145
G Ensuring Safe and Consistent Co-Engineering of Cyber-Physical Production Systems: A Case Study		147
1	Introduction	148
2	Background	149
3	Problem Statement	150
4	Goals and Requirements	151
5	Approach	152
5.1	Artifact Integration	153
5.2	Work Areas	158
5.3	Traceability	158

5.4	Consistency Checking	161
6	Application to Challenge Problem	164
6.1	After Step 3	164
6.2	After Step 5	165
6.3	After Step 6	165
7	Discussion	166
7.1	Goals	166
7.2	Requirements	166
7.3	Challenges and Lessons Learned	166
8	Related Work	167
9	Conclusions and Future Work	169
Bibliography		171

Part I

Prologue

Chapter 1

Introduction

In modern day engineering large-scale projects often face a multitude of technical and organizational problems [11, 14, 59, 78]. One major issue arises with the strong interdisciplinary aspect of modern engineering solutions. The development of these solutions covers a great number of highly varied, complex challenges (e.g., in logistics [52], medical care [71], agriculture [62] and many others). Engineering problems are no longer solved within the boundaries of a single engineering discipline, but require the collaboration of a wide variety of experts coming from multiple domains. These experts must tightly integrate their knowledge with that of other stakeholders to meet the requirements of a modern project.

Software engineering, as an interface between many different engineering disciplines, plays a crucial role in this regard. Not only does it include various sub-disciplines in itself – such as requirements engineering or architectural design – but it incorporates the results of many other engineering fields as well (e.g., the designs of an architecture must correspond to hardware specifications, constants within code must match with the calculations from a mechanical engineer, etc). As a result, engineering – in particular software engineering – must deal with a wide variety of heterogeneous work results. We refer to these results as *engineering artifacts*.

While these artifacts are often syntactically independent from each other, they share semantically overlapping concepts and therefore, strong interdependencies. To name a simple illustrative example, while software code and UML are syntactically different, they may semantically describe the same behaviour. If UML is then used to define a software architecture, the code is strongly dependent on the related UML designs. While these interdependencies exist, the corresponding artifacts are often developed by entirely different engineers. They are, further, created in different tools and stored in different locations.

The interdependence between heterogeneous engineering artifacts requires feasible co-evolution strategies. If an engineer changes an artifact, it must be ensured, that the corresponding information about the change is delivered to all engineers that work on related artifacts. These engineers must then adapt their work accordingly. Naturally, the bigger an engineering team is, the more difficult this kind of information propagation becomes.

Observing the issue realistically, it is reasonable to say, that inconsistencies between interdependent, heterogeneous artifacts are bound to arise. Even if there was perfect documentation of dependencies, the co-evolution of artifacts would sometimes be offset due to technical reasons or simple delays.

It is therefore crucial for every modern engineering project, to keep a detailed record on change information, in order to identify and document potential inconsistencies and support their eventual correction. Failing to do so may result in deeply rooted flaws, which are highly expensive to fix [8, 95] or even harmful to human well-being [98].

This work examines a solution for this challenge, in the form of incremental, global consistency checking, storing change information and giving corresponding, immediate

feedback to engineers.

In such a setup, engineers would perform certain atomic changes on their respective engineering artifacts, e.g., they change a parameter in their code, or an attribute in their UML diagram. Such changes would be analyzed in relation to interdependent engineering artifacts on the basis of certain global consistency rules. These interdependent engineering artifacts need not originate from the same engineering discipline, i.e., code artifacts may be checked against UML artifacts, given the necessary consistency rule. If a change then violates one of these rules, the violation is recorded and affected engineers are immediately notified.

For software development, the importance of such immediate, global consistency checking is further underlined by common practices during frequently applied team organization techniques. For example, in agile development daily meetings are complemented by very detailed time planning for individual task (e.g., in burndown charts tasks are planned on an hourly basis). This means, that a holistic (global) coordination between team members only happens once a day, while exact and timely execution of tasks is constantly required. If an inconsistency affecting several developers is identified at all, it will probably be identified during the coordination meetings. However, between these sessions of coordination several tasks may have already been finished by the engineers. Errors may have propagated and fixing these errors will very likely disrupt the time planning on which agile development depends. Agile development, as well as other techniques for organizing and planning software development, could greatly benefit from an immediate and global consistency checking mechanism, that observes and checks the interdependent properties of the engineers' individual artifacts. Instead of daily coordination, it would see for continuous live coordination.

The high complexity of this issue is ideally approached in an automated manner. This thesis proposes the utilization of a collaborative engineering environment, for the storage and analysis of engineering artifacts. It suggests the incremental synchronization of artifact changes with the environment, and further proposes the immediate identification of potential inconsistencies. Inconsistency identification is realized on the basis of a user-defined set of (global) consistency rules. The recorded results of these consistency checks are instantly propagated throughout the environment and automatically distributed to all affected engineers.

1.1 Outline

This thesis is structured as a cumulative dissertation. We present a series of papers on the introduced subject matter and discuss their respective contributions in-depth. The thesis is organized in two parts:

In Part I of the thesis we give insights into the problem, discuss our approach with an extensive running example and summarize each paper. We re-iterate on how our incremental solution was formed and reflect on possible alternatives regarding individual decisions. To conclude, we give an outlook on potential future work.

In Part II we present the seven core papers of our work (Papers A – G). In six out of these (Papers B – G) the author of this thesis was also the main author of the paper. In one work (Paper A) he assisted the design, execution and evaluation of the described experiment, as well as the writing process of the paper's initial version.

Besides minor adjustments of the layout, the papers are presented in the dissertation as they were published. It should be noted however, that two of our works (Papers E and F) were still under submission, at the time of writing this thesis. Future, camera-ready versions of these works may slightly differ from the state presented here.

Chapter 2

Problem Definition

In this work, we examine how engineering artifacts are changed and how the respective change information must be analysed and processed to uphold a consistent artifact state. For this, we must first clarify some fundamental concepts and their respective issues.

2.1 Engineering Artifacts

Engineering artifacts, as the core subject of this thesis, are defined as the “results” of an engineer’s work. Such results are, for example, a set of requirements, a fraction of code but also a single constant within a hardware specification or similar. In this thesis, we mainly deal with two kinds of artifacts (see Figure 2.1), which can be distinguished by their involvement with the concrete implementation of an engineering product:

- **Documentation Artifacts:** Any artifact that specifies the functionality of the final product, without being part of the concrete realization of the said functionality. Examples would be design documents, hardware specifications or requirements.
- **Implementation Artifacts:** Any artifact that realizes the functionality of the final product. Examples would be machine instructions, software or hardware parts.

We treat these kinds of artifacts equally, since the meticulous maintenance of both is required for a sound end product. In order to create this end product, we must coordinate engineering artifacts and compile them into a functional unit. Depending on the number of stakeholders in a project, this coordination process is more or less prone to error. There are both organizational and technical issues to consider. From the organizational perspective, troubled coordination may be caused by delays, failure in communication, weak process

Figure 2.1: The different artifact kinds considered in this thesis.

management, key engineers leaving, inaccurate planning and other problems. From the technical perspective the coordination of engineering artifacts is affected by three major factors:

- **Distribution:** Engineering artifacts are usually distributed over several virtual or physical locations. A physical distribution is given if artifacts are stored on different machines. A virtual distribution is given if these artifacts are technically available at the same physical location, but they are stored in separate repositories or edited in separate tools. The latter is usually the case due to a difference in syntax.
- **Syntax:** Different engineering artifacts usually follow different syntactic formats. We speak of heterogeneous engineering artifacts. Not every team member may be able to interpret the syntax of these artifacts. This poses a problem in coordination because semantically overlapping aspects of engineering artifacts must be understood by all affected engineers.
- **Semantics:** The main effort of artifact coordination lies within the coordination of the artifact's semantically overlapping properties. These properties express similar concepts among different artifacts and commonly, but not exclusively, act as the junction point between documentation and implementation (e.g., the class names in a UML class diagram and in its concrete implementation should match). Semantically overlapping properties are indicative of interdependencies between artifacts.

The coordination of engineering artifacts is mostly concerned with maintaining their interdependent properties. This thesis mainly deals with the technical perspective of artifact coordination, respectively with the failure to coordinate correctly. Such failure expresses itself in the form of inconsistencies.

2.2 Inconsistencies

When the interdependent properties of engineering artifacts are no longer in a coherent state we speak of inconsistencies. Such a state is often reached when engineers adopt changes that lack organizational coordination. A lack in technical coordination sometimes furthers this circumstance. For example, an engineer misunderstands the hardware specifications of a colleague and therefore implements a software module under the wrong assumptions. In this case there exists a failure in communication and the syntactic differences between hardware specification and code may further the engineer's misconceptions. This example illustrates that even with complete technical coordination, there still exists the potential for inconsistencies due to lacking organizational coordination.

Inconsistencies are often caused by a single change, however, sometimes a set of changes may eventually lead to an inconsistent project state. The complexity of the problems this may cause, motivates a feasible strategy for prevention.

2.3 Consistency Management

To counter inconsistencies projects often apply consistency management. This practice is made up of three parts, which largely build on each other, as illustrated in Figure 2.2.

- **Consistency Planning:** The prevention of inconsistencies is largely of an organizational nature. Detailed planning and a well aligned team of stakeholders can keep a project's engineering artifacts mostly consistent. However, temporary inconsistencies are sometimes necessary to keep the development of a project flexible. Consistency management mainly dependent on planned prevention (e.g., via best practice documents) requires highly disciplined and experienced engineers.

Figure 2.2: The different aspects of consistency management building on each other. Repair requires checking, checking requires planning respectively rules.

- **Consistency Checking:** Consistency checking relies on either the manual or automatic validation of consistency rules. If such a rule is broken, an inconsistency is identified.
- **Consistency Repair:** When an inconsistency is identified, engineers must devise a way to correct it. This process can be supported by automatic inconsistency repair mechanisms.

This work mainly focuses on automated consistency checking, in order to support later inconsistency repairs. Consistency checking applies a rule set that expresses a coherent state for the interdependent properties of engineering artifacts. Such a rule set can be formalized in different ways:

- **Natural Language:** The engineers agree on certain rules, e.g., naming conventions for the documentation and implementation of an interface. Such procedure is usually only fit for manual consistency checking. This normally happens during meetings.
- **Declarative Language:** Declarative languages define certain constraints on the concrete properties of artifacts. An example would be the Object Constraint Language (OCL). Such languages are fit for automated consistency checking, which may be applied continuously and during the engineers' work.

In this thesis we rely on explicit consistency rules, defined in a declarative language. Consistency checking itself, can be realized in various ways. The most important distinction in this regard is local versus global consistency checking.

Local (or traditional) consistency checking is defined as the consistency checking of a single kind of engineering artifact (e.g., requirements, code, UML models, etc) originating from a single tool (e.g., a text editing tool, a programming IDE, a UML design tool, etc). The consistency checker is often integrated with the said tool and checks the well-formedness of the engineering artifact with regards to its respective meta-model or a similar definition (e.g., a parser within a programming IDE checks the written code against the language's grammar).

Global consistency checking, on the other hand, is defined as the consistency checking of multiple kinds of engineering artifacts, originating from different tools. These artifacts are checked against each other on the basis of their interdependent properties. This kind

of consistency checking could not happen within a single tool, unless such a tool could reason over all kinds of engineering artifacts. Typically this is not the case. To realize global consistency checking the corresponding mechanism must overcome technical coordination problems. It must be able to validate various consistency rules on heterogeneous engineering artifacts and provide a way to distribute the results to affected engineers.

Chapter 3

Approach

In this chapter, we discuss how our approach addresses the problems presented in Section 2. We describe its functionality and compare its workflow to a traditional engineering setup with an illustrative example.

3.1 Overview

In this work we apply a multi-developer, multi-tool engineering environment, which stores and analyzes engineering artifacts and informs engineers about potential conflicts. The computation of such information requires technical coordination as discussed in Section 2.1. To be analyzed on a uniform foundation, engineering artifacts must overcome the hurdles of distribution, syntax and semantics.

Our approach - which is illustrated in Figure 3.1 - achieves this in the following way: Collaborating engineers equip their respective tools with tool adapters. The tool adapters transform the artifacts from the tools' internal data structure into a uniform representation. In this representation, artifacts are abstracted to a point, at which they can be compared. For most engineering artifacts, this can be achieved in a graph-like structure, that represents their individual attributes and their relationship with each other. With this, the syntactical differences between engineering artifacts can be overcome.

The tool adapters then incrementally synchronize the changes on this uniform representation to a centralized, remote artifact storage. All changes in their entirety represent the current state of an engineering artifact. This way, we overcome the physical and virtual distribution.

With all data in one place, a coordination of semantically overlapping properties becomes a matter of relating the synchronized artifacts. Relationships can be established by enhancing the artifact's individual graph structures with additional links between them. Such links - in the form of separate artifacts - serve as navigational paths for the analysis of the artifact structure. Among other things, such analysis may include examinations on traceability, risk assessment, correlation or - as in our case - consistency.

Consistency checking is ideally employed as a preemptive measure, meaning that it tells us, what the consistency state of our project were, would we go through with certain changes that we have in mind. This preemptive concept is realized within our approach via a differentiation in the artifact storage. After the tool adapter synchronizes an artifact's changes, these changes are stored within a private, user-specific work area. A private work area builds the lowest part in a hierarchy of work areas, where the changes in the lower levels overrule the changes in the higher levels. As a result, every private work area has a specific perspective on the artifact storage and its contents. This perspective is individualized by the private work area's contained changes, and the changes contained in the hierarchy above it. The root of the hierarchy is visible from every private work area and therefore referred to as the public area.

Figure 3.1: An overview of the approach and its functionality.

Changes can be transferred from a work area to its parent by “committing”. This may affect the consistency state of other underlying work areas and therefore requires the distribution of consistency-relevant change information and subsequently the re-evaluation of the private work area’s consistency state.

3.2 Running Example

To illustrate our approach and its effect on the traditional workflow of engineers, we will discuss a running example in the context of both a traditional engineering setup and the setup used in our approach. Consider the following situation: A team of engineers wishes to realize an engineering project, encompassing the installation of a factory robot arm next to a conveyor belt. Cooperating among these engineers are Alice, Bob and Carol. Alice is designing the necessary functionality with the help of UML diagrams. Bob is maintaining documentation on the required hardware. Carol is implementing the functionality of the respective software controller. The artifacts produced by these cooperating engineers are highly heterogeneous, yet interdependent. A change in hardware specifications (e.g., firmware updates, interface changes, enabled features, deprecated frameworks, etc) may expand or limit the possibilities in design and have an impact on the functionality of the concrete software implementation.

Assume such a change comes to pass in a traditional engineering setup, where every engineer is working with their own tool and largely independent from others - meaning, that their heterogeneous artifacts aren’t considered together in terms of automated analysis (i.e., consistency between heterogeneous artifacts is at best checked manually). Bob is informed of a new hardware specification, in which the robot arm’s operational distance is slightly shorter than in the original specification (see Figure 3.2). In effect this means, that the original specification is out of date, as is Alice’s diagram, which refers to the operational distance by a named property called “range” (see Figure 3.3). Likewise, the

Figure 3.2: An update in the hardware specification of the robot arm causes a reduction of its operational distance (Old: Variation A, New: Variation B)

implementation of Carol is out of date, as it references the values of the UML diagram in a constant of her code (see variable “opDist” in Figure 3.4). Note that all three engineers use different names to refer to the same concept. This is a typical semantic overlap in a traditional engineering setup, without pre-declared naming conventions and a lack of consistency planning. But even with declared naming conventions manual consistency management may run into major problems.

As Bob is the one receiving the original change, he updates his respective documents and must then - at the very least - contact Alice, who will in turn contact Carol. This form of coordination may include delays, misunderstandings or simply incomplete information. This is especially true if cooperating engineers are working in different departments, locations or are simply unaware of the effects their changes may have on the artifacts of others. Negative consequences may unfold in a multitude of ways. For example, a delay in the propagation of change information may go hand in hand with propagating errors on the side of the uninformed. The original length of the robot arm may be used in further diagrams by Alice or defined in different parts of Carol’s code. The longer the delay lasts, the more likely such errors propagate through the engineers’ work. The more time-critical a project’s planning is, the more harmful such delays may become, since important milestones might not be met if extensive reworks are required later on. This calls for a more secure, automated approach, that distributes change and consistency information as artifacts are modified.

We will now discuss how the scenario laid out above, would play out in our approach. We assume that all engineers incrementally synchronize their artifacts with the remote artifact storage and that consistency planning has provided the necessary rules to keep their interdependent properties (i.e., Bob’s specified robot arm length, Alice’s “range” value and Carol’s “opDist” constant) consistent. We further assume that artifacts have so far not been shared among the engineers (i.e., they have not yet been committed to any work area higher in the hierarchy). To illustrate this aspect of our solution, Alice and Bob share a joint work area, the contents of which are invisible to Carol - this may be the case because of organizational coordination. Figure 3.1 again serves as an illustration:

Bob first enters the new specification in his document. The adapter for his tool recog-

Figure 3.3: Alice's UML diagram on the robot arm's functionality.

nizes an incremental change on the respective part of the document. If for example, Bob works with a spreadsheet, a change within a single cell can be recognized. The entered value is transformed into the uniform representation and adopted on the corresponding artifact. This change is then synchronized with the artifact storage and stored in Bob's private work area (see Figure 3.1, step [1]).

Together with the joint and the public work area, Bob's private work area forms a perspective on the artifact storage that is individualized by its contained changes. These changes overrule any overlapping changes in parent work areas. From this perspective, the consistency checker can now react with an evaluation of the new value. Since related artefacts (respectively, interdependent properties according to provided consistency rules) can neither be found in the joint work area, nor in the public area, the consistency check would not be concluded and therefore not be relevant.

Bob would then continue to commit his work to the joint work area (see step [2]). This triggers the distribution of consistency-relevant change information: As Bob's changes reach the parent work area, the change information is propagated to the other child work areas (in this case Alice's - see step [3]). With this step, Bob's data enters Alice's perspective. Her work is now potentially inconsistent. As a result the consistency checker must re-evaluate relevant artifacts according to the distributed change information. Whether and where such re-evaluations happen depends on the internal consistency state of the private work areas and is a later focus of this thesis.

The consistency checker will detect Alice's "range" value to be inconsistent with Bob's newly changed specification. This requires these fields to be linked during consistency planning (respectively, while introducing the artifact types in the environment) and to be covered by the corresponding consistency rule. Aware of the inconsistency, Alice can now change her work and synchronize it with her private work area (see step [4]). This causes further re-evaluations, now showing the corrected consistency state. In this state Alice can then commit her work to the joint work area as well (see step [5]). Given that no further changes have to be made, both Alice and Bob can commit their work to the public area (see step [6]).

In the same way as the consistency-relevant change information spread from the joint work area to Alice's work area, the public area's newly committed change information is now propagated to Carol's work area (see step [7]).

Carol can react and synchronize her changes (see step [8]) to finally commit her work

```

1  public class RobotArmController
2  {
3
4      private final static float opDist = 1.0;
5
6      public RobotArmController()
7      {
8          super();
9      }
10
11     private void grab(Target target)
12     {
13         if(target.distance <= opDist)
14         {
15             //do something
16         }
17     }
18
19 }
20

```

Figure 3.4: Carol's code within her IDE.

to the public area as well (see step [9]).

All change information is immediately propagated as it becomes available in the respective parent work areas. Likewise, all consistency information is immediately updated to keep the consistency information in the individual work areas correct. As we will later see in this thesis, this is an absolute necessity when consistency information is kept in artifact form. The workflow allows for no delays in consistency checking and immediately makes change information and its consequences clear to engineers. The backbone of this workflow is the instant distribution of consistency information and the handling reaction of the consistency checker towards it. This makes sure that all artifacts are correctly re-evaluated and the individual perspectives of engineers hold the latest consistency information. The corresponding algorithm can shortly be summarized as described in Listing 3.2. This algorithm is followed by a propagation of change information to all children of the private work area for which the consistency checker is handling the change.

In the following chapters we give a short overview of all published papers and summarize how the presented approach was developed.

```

1  switch(change.changeType)
2      case CREATE:
3          //depending on artifact type:
4          //create or retrieve consistency information
5          //for re-evaluation in private work area
6      case MODIFY:
7          //depending on artifact type:
8          //retrieve consistency information
9          //for re-evaluation in private work area
10     case DELETE:
11         //depending on artifact type:
12         //clean-up or re-evaluate consistency information
13         //within private work area

```

Listing 3.1: Algorithm on the basis of which the consistency checking mechanism handles change information from the perspective of a private work area

Chapter 4

Paper A: An Empirical Study on the Impact of Inconsistency Feedback during Model and Code Co-changing

In this paper we conduct a controlled experiment. We study the effect of instantly available consistency information on the performance of engineers, during the propagation of consistency-relevant changes. We examine, whether consistency information, respectively the lack thereof, has an impact on the correctness of co-changing models and code. The experiment serves as a motivation for the distribution of consistency information and presents potential advantages of an automated solution.

4.1 Discussion

The controlled experiment described in this paper requires subjects to solve certain engineering tasks. These tasks revolve around model and code co-changing. The subjects are provided with a code basis and UML diagrams. Between these engineering artifacts certain change requests have to be completed (either on the side of the code or the model), while specified consistency rules have to be upheld. Some tasks provide information about preceding changes while others do not. Each subject is provided with tasks of different complexity. The complexity of a task is determined by the amount of involved model diagrams. The experiment is conducted with 36 subjects, all of which have at least fundamental knowledge about programming and UML. In the analysis of our results we particularly examine two different characteristics:

- **Correctness:** To solve a task, the subjects have to adapt multiple changes. For each task, these changes are categorized in three groups (e.g., method renaming, class relation restructuring, class renaming etc). If none of the groups can be adapted in their entirety the task is graded as *“incorrect”*. One or two entirely adapted groups of changes are graded as *“partially correct”*. If all changes are adapted properly the task is graded as *“correct”*.
- **Time effort:** Each subject has to record its time required to work on a task. The entire allocated time for the experiment is 180 minutes, plus 20 minutes of briefing. The subjects ideally require 18 minutes for a single task.

For both characteristic we wish to analyse whether the availability of consistency information has a positive impact.

4.2 Results

Our controlled experiment came to the results that, regardless of model or code base, consistency information improved results dramatically.

With regards to correctness, the availability of consistency information resulted in the subjects producing almost three times more correct solutions and roughly half the amount of incorrect solutions. Among the tasks with consistency information, 59 were solved correctly while 33 were incorrect. Among the tasks without consistency information only 22 were solved correctly, while 68 were incorrect. The consistency information had no significant impact on partially correct solutions. Subjects failed especially in tasks with higher complexity, though the availability of consistency information produced better results.

Our experiment further showed, that tasks with consistency information would on average require slightly more time. However, this was mostly due to low complexity tasks, which were correctly solved faster without consistency information. The mean time for complex tasks was lower when consistency information was available.

4.3 Contributions & Extension

In this work, we examined whether the immediate availability of consistency information has a beneficial impact on the correctness of an engineer's work. We conducted a case study with 36 participants and provided a discussion of the results. These results illustrated a positive effect, which gave us further motivation to develop an automated solution for the computation and distribution of consistency information. Our next steps would involve the conceptualization of a corresponding system, that allows such distribution, as well as the utilization of a consistency checking mechanism for the computation of the distributed information.

Chapter 5

Paper B: Collaboratively Enhanced Consistency Checking in a Cloud-based Engineering Environment

In this paper we outline the functionality of a cloud-based engineering environment. We tackle important aspects of such an environment, like the uniform data representation, live feedback, typing of artifacts, linking of artifacts and how artifacts can be stored in private work areas. These private work areas act as isolated containers for the engineer's work. We specify the benefits an artifact analysis mechanism, respectively a collaborative service could gain from such an environment. Such benefits include full activity awareness, multi-tool consistency checking as well as team driven error handling. We define individual steps a consistency checking mechanism must fulfill in order to provide correct consistency information within a cloud-based engineering environment.

5.1 Discussion

In this paper we discuss the cloud-based engineering environment and how a specific set of features within such an environment may contribute to the collaborative enhancement of a given artifact analysis mechanism (e.g., a consistency checker). This set of features includes:

- **Uniform Data Representation:** A generalized data representation that is achieved by mapping the attributes of engineering artifacts to so called properties. A property has a name and a value. Several properties form an artifact. An artifact is uniquely identifiable.
- **Live Feedback:** Changes have to be transferred to the engineering environment through incremental updates. By doing so we want to provide a complete timely analysis of changes which results in live feedback for whatever the engineers are doing. This feature naturally requires fast analysis mechanisms, integrated with the engineering environment.
- **Artifact Typing:** To further organize the uniform data representation, artifacts must correspond to a certain type. The type pre-determines an artifact's properties as well as their cardinality.
- **Linking:** References between different artifacts are crucial to consistency checking. Therefore, the engineering environment must provide a simple and efficient way to

link from one artifact to another. In practice this is done by filling a property with another artifact's ID.

- **Private Work Areas:** Private work areas store an engineer's changes on whatever artifacts are currently held in the engineering environment. It is only visible to the engineer that is using it and complemented with existing artifacts of the engineering environment it forms an individual perspective on existing engineering artifacts. Later papers will focus stronger on the aspect of private work areas, since their existence requires us to think about how consistency information must be distributed, in order to give users a correct perspective on whatever is stored in the engineering environment.
- **Collaborative Services:** Since analysis mechanisms should be integrated directly into the engineering environment, we considered a generalized interface for integration. This consideration was mostly an engineering challenge and was improved throughout various prototype implementations.

We can conceptualize a consistency checking mechanism, that builds upon such a feature set and which is entirely integrated into the cloud environment. Roughly outlining the mechanism, such consistency checking must be done in a five step procedure:

- **Trigger:** The consistency checking mechanism must be triggered in a certain way. Either manually or automatically. Later in development we mostly focus on the automatic triggering of the mechanism.
- **Rule Retrieval:** When the mechanism is started, the corresponding consistency rules must be retrieved. In earlier prototypes of our approach, rules were stored separately from the cloud environment. For this approach we would settle with rules being integrated into the environment as well.
- **Data Gathering:** Traversing the artifact structure according to the rule is necessary for the retrieval of all consistency relevant knowledge. On the basis of these results the rule evaluation happens. In later papers the rule evaluation step is separated from the data gathering. On the one hand, we needed to go into further detail with data gathering, on the other hand a separated rule evaluation gives us the possibility of switching out the rule evaluation mechanism.
- **Writing Back Results:** After results have been computed, they have to be recorded. We settle for recording the results within the artifact structure. The rational behind this is the fact that every piece of development related information forms an engineering artifact in itself and therefore should be shared on the cloud. The distribution of these elements is further explored in later papers.
- **Providing Feedback:** Once all results are recorded, feedback must be given to users. In our approach we wish for this feedback to be delivered instantly.

The integration of a consistency checking mechanism into a cloud environment collaboratively enhances the said mechanism and the user's experience with consistency checking in general. The users can benefit in many different ways. To us, the most important ways are identified in the paper as follows:

- **Activity Awareness:** Users immediately know which artifacts exist within an engineering project. Any collaborative services, such as a consistency checker, can benefit from this circumstance as well, since changes can be immediately identified and analysed. This causes greater change awareness for users, from which they can benefit in two ways. Either engineers immediately know what the consequences

of their own changes are in an isolated area (private change evaluation), or which consequences newly published changes from other engineers have (public change evaluation).

- **Multi-Tool Consistency Checking:** As artifacts from different engineering tools are stored within the environment (on a uniform basis) a consistency checking mechanism can easily reason over their interdependencies. For the users this means that related concepts from different engineering artifacts, which had to be compared in meetings before, can now be compared automatically (e.g., naming conventions between UML designs and implementations to ensure traceability).
- **Team-Driven Error Handling:** Since the engineering environment's artifact storage is concurrently modified and analysed, the integration of a consistency checker can enhance team-driven error handling. By identifying the origin of certain inconsistencies incrementally and timely, engineers can immediately collaborate with others to find a solution instead of trying to fix issues in isolation.

5.2 Results

This paper acted as a road map for future development. Its main contributions were the different outlines for both an engineering environment and the integrated consistency checking mechanism. As a proof of concept we discussed different case studies and experiments that have been conducted with a former prototype implementation of a similar, but less evolved, approach. We discussed how some benefits of our approach could already be achieved during the consistency checking of EPLAN Electric P8 drawings (FMTC Case Study). We further discussed which problems could be addressed with regards to the linking of UML models and CAD drawings (ACCM Case Study). An industrial study, which addresses multi-tool consistency checking was shortly outlined as well.

5.3 Contributions & Extension

In this paper we outlined the basics of consistency checking within a cloud-based engineering environment. We discussed the functionality of such a consistency checker and how it could benefit from the cloud environment. The core talking points of these discussions, defined our goals for future implementations. With these goals defined, we set out to realize our approach step by step. The work after this paper would include details on the realization of individual aspects of the environment and how they would impact the consistency checking mechanism. We would particularly inspect the role of private work areas and how they affect the distribution of consistency information within the cloud environment.

Chapter 6

Paper C: Live and Global Consistency Checking in a Collaborative Engineering Environment

This paper discusses a basic realization of consistency checking within a collaborative engineering environment. It expands on previous work by further defining the internal concepts of the environment. With these definitions, the paper goes into detail on the different elements involved in consistency checking and how consistency information must be represented within the environment. The paper further discusses how incremental changes, which are synchronized with the environment, must be handled prior, respectively during the evaluation process in order to provide instant consistency feedback. Engineering artifacts are integrated into the collaborative engineering environment with the help of tool adapters and a uniform data representation. The artifacts are stored in a single work area, which acts as a holistic global knowledge repository. This means, every engineer can see all artifacts and changes that are currently synchronized with the environment.

6.1 Discussion

This paper discusses basic conceptual details on how to implement a collaborative engineering environment. It further describes which features must be given to provide comprehensive consistency checking within such an environment. In this work, we limit ourselves to a single work area, holding all incrementally synchronized artifacts. This means every engineer can see and manipulate every artifact. At this point of development there is no implementation for storing isolated changes within the environment. One should note however, that implicitly there is always a certain notion of isolation given with tools and tool adapters. Tool adapters synchronize the engineer's changes with the engineering environment. If synchronization can be explicitly triggered by the user, the artifacts of an engineer remain isolated from other engineers until they are part of the artifact storage's contents. Of course such changes cannot be considered during consistency checks and their synchronization may cause inconsistencies. Such inconsistencies must be identified with every incrementally synchronized change in the artifact storage. To continuously incorporate the engineers changes with the consistency checking mechanism we will later utilize a private work area, which stores changes before they are integrated with the rest of the artifact storage.

In terms of the consistency checking mechanism, this paper discusses a set of required concepts. Most importantly we define the difference between a context type and a con-

text element. The earlier represents an artifact type for which a consistency rule must hold. The latter represents a concrete artifact type instantiation for which a result can be computed. Accordingly, we introduce artifact types that encapsulate this distinction. The (Consistency) Rule Definition Artifact (later referred to as “CRD”) defines a rule for a context type. The (Consistency) Rule Evaluation Artifact (later referred to as “CRE”) stores the result of such a rule definition for a specific context element. Further, it holds a “scope”, which records all artifacts and the properties involved in a rule evaluation. We refer to these as “scope elements”. The differentiation between these two artifact types is important for the discussion of the evaluation process.

The consistency rule evaluation process discussed in this paper roughly corresponds to the first three steps of the five step procedure defined in Chapter 5. We focus especially on the triggering of the mechanism, by narrowing down how the consistency checking mechanism would react in three different scenarios:

- **Addition:** Whenever a rule (concerning a certain context type) or a context element is added to the artifact storage, the consistency checker must create corresponding artifacts, holding consistency information.
- **Modification:** Whenever an artifact is modified, the corresponding consistency information must be updated.
- **Deletion:** Whenever a rule or an artifact is deleted, the corresponding consistency information must be cleaned up as well.

It must be noted at this point that within our implementation of a collaborative engineering environment, no artifact is truly ever deleted. Artifacts carry a flag describing their “alive” status. If the flag is set to false, the artifact is ignored in all regular artifact retrieval processes. In the paper we also shortly discuss how feedback about inconsistencies can be shared with the developers. We propose to propagate all computed consistency information to tool adapters by sending them a corresponding notification. This notification holds information such as the consistency results and involved artifacts. The interpretation of the notification is up to the tool adapters, respectively the tool. This aspect will be carried over into later papers. During the development of the consistency checking mechanism a continuous point of discussion was the question, how users should be warned about inconsistencies. It was further discussed whether such warnings could be standardized to a certain degree. However, we eventually settled with the decentralized approach defined in this paper (i.e., tools decide their warnings). Some of the tool adapter implementations later showed, that the different visualization methods of artifacts (e.g., text-based for code, graph based for UML diagrams, etc.) all require special consideration when it comes to visual cues concerning inconsistencies. A screen warning would be possible for all tools, this however would be rather distracting. Inconsistency feedback is handled in a better way if it is directly integrated in the visualization method of the respective tool (e.g., underlining inconsistent code sections, coloring inconsistent UML model elements in red, etc.). The realization of such feedback must therefore be up to the implementer of the tool adapter.

6.2 Results

The main result of this paper was a basic implementation of the collaborative engineering environment and an integrated consistency checking mechanism. We provided different tool adapters as a proof of concept and discussed how requirements could be linked to UML elements and in turn to Java code. This was done to underline the capabilities of the uniform data representation with regards to heterogeneous engineering artifacts. With

this basic implementation we could provide instant consistency feedback on heterogeneous artifacts in a single work area.

During the implementation, another concern of ours was the "live" applicability of our approach. We therefore discussed an examination of our mechanism's scalability. Ten projects were analyzed on the basis of 14 consistency rules. The mean observed processing time was 10.7 ms, while the median was 8.8 ms. On average the processing time would remain below 50 ms, which is acceptable for the "live" application of our approach.

6.3 Contributions & Extension

In this paper we provided a basic realization of our approach. In this iteration of the implementation, we were able to synchronize heterogeneous engineering artifacts with a collaborative engineering environment and check the consistency between them in a single, global artifact storage. Our incremental approach allowed for immediate consistency feedback towards synchronized changes. The notion of isolated changes, which are stored in private work areas was not part of this work. As a result there was only a single perspective on the engineering artifacts which could be utilized for consistency checks. In future work we would incorporate private work areas in the engineering environment. These would allow us to store individual artifact changes in isolation from the rest of the artifact storage. The changes, together with the existing artifact storage used in this paper, would allow us to define individualized perspectives on the engineering environment's contents. In future work we would adapt the consistency checking mechanism to these individualized perspectives.

Chapter 7

Paper D: Multifaceted Consistency Checking of Collaborative Engineering Artifacts

This paper discusses a more involved realization of consistency checking in a collaborative engineering environment. Various critical issues that arise during regular consistency checking are explained. The paper discusses how these problems can be tackled with the application of our consistency checking mechanism. It expands on previous work with the distinction between private and public knowledge. Public knowledge denotes all artifacts stored in the engineering environment that are publicly available to engineers. These are generally visible and stored in a single (public) work area. Private knowledge, on the other hand, denotes individual changes on the public knowledge, which are stored in several (private) work areas. These changes are synchronized by individual engineers and represent the changes they make on artifacts within their tools. The synchronized changes are layered on the existing contents of the artifact storage to form an individualized perspective on it. We further discuss how such private knowledge must be considered during consistency checking and what this means for the distribution of consistency information.

7.1 Discussion

We start this work with a detailed discussion of the problems commonly found in current collaborative engineering projects. We identify a set of relevant talking points and discuss how our work could contribute to solving the described problems. These problems include the following:

- **Syntactic Differences:** Syntactic differences between artifacts complicate the consistency checking on their interdependent aspects. With the collaborative engineering environment we eliminate these differences and check consistency on a uniform basis.
- **Difficult merging process:** With existing inconsistencies between artifacts, changes on the said artifacts may become difficult to merge. Uncoordinated solutions between engineers may nullify each other unless both are fully aware of the latest state of artifacts. Being able to continuously check consistency on the basis of private changes can help with this issue, since engineers are always fully aware of the state of dependent artifacts.

- **Late or no recognition of inconsistencies:** Often inconsistencies are detected long after they were originally caused. This may lead to serious follow up errors. Our approach cuts down the time between introducing and detecting an inconsistency by immediately providing consistency feedback for every incremental change on artifacts.

These problems are especially relevant to our work, as instant consistency checking on a heterogeneous basis can largely prevent them. A requirement for this however, is the computation of consistency information on the basis of privately stored changes.

We therefore continue with an extension of previously discussed concepts. In previous work we described how artifacts are synchronized with the engineering environment and stored in a single work area for consistency checking. The problem with this approach is the fact that individual changes of engineers can not be integrated into consistency checks. Engineers would change artifacts within their tools where the said changes are unavailable to the consistency checking mechanism. Changes could only be checked whenever they were integrated with the artifact storage's content. We decided that this was rather late. Further it would allow for the synchronization of inconsistent changes. We determined that it would be better if engineers continuously knew about the consistency implications of their changes before they would integrate them into the artifact storage. Therefore, this work focuses on the adoption of private work areas, which act as containers for individual artifact changes in the collaborative engineering environment. When retrieving an artifact from the collaborative engineering environment, these containers' contents are layered on the (public) artifact storage. Therefore, the engineers can retrieve and store their individual changes in the environment, without interfering with the work of others (as it would be the case with a single work area). In the private work areas the engineer's changes are available to the consistency checking mechanism. As a result we can provide consistency information about an engineers changes, before they are integrated with other artifacts. This is in contrast to contemporary collaboration software, where holistic consistency checks usually happen on a full artifact repository with integrated individual changes (e.g., in the form of a merged branch in Git).

With regards to consistency checking, the adoption of private work areas requires us to consider one of the central research questions of this thesis: *How must consistency information be organized and distributed in a collaborative engineering environment, so the engineers' individual perspectives on the artifact storage remain correct and up-to-date.* Therefore, this paper reiterates on the outlined rule evaluation process and discusses every step in further detail. The work especially focuses on the "Change Analysis", which is the process of reacting towards the engineers' synchronizing changes with the collaborative engineering environment. The goal here is to find a definition, that can be generalized for any artifact storage element, both the private as well as the public work area. With the consistency checking mechanism computing consistency information in a generalized way the remaining challenge is the distribution of this information.

7.2 Results

The implementation results of this work covered an engineering environment, which allowed us instant consistency feedback on heterogeneous artifacts in subordinate, private work areas.

We further implemented a tool for the linking of stored engineering artifacts. Links could be realized in the form of typed artifacts which were stored alongside regular artifacts, synchronized by engineers. The realization of linking is critical for consistency checking in so far, as consistency rules often span several domains and artifacts. It is necessary to link these artifacts through bi-directional reference, so the consistency checking mechanism

can navigate the artifact structure. Naturally, such navigation must seamlessly work between private and public work area, i.e., a privately stored artifact (respectively a set of equivalent, incremental changes) could be linked to a publicly stored artifact - however, not the other way around.

7.3 Contributions & Extension

With this work we extended the concept of the artifact storage, by the addition of private work areas. These allowed for the isolated storage of artifact changes synchronized by the engineers. These changes could be layered upon the global, respectively public, artifact storage. The engineers received an individualized perspective on the public artifacts, complemented by their own changes. We would later expand on this by adopting a work area hierarchy in which work areas could be connected in arbitrary parent/child relationships. The public work area would then act as a root of a tree of work areas. The structure of the tree would determine the individual perspectives engineers would have on the artifact storage's contents. Additionally we would expand on the perspectives by narrowing them down via timestamps. A perspective could be showing the latest state of the artifact storage (in combination with individual changes) or the state up to a certain point in time. Both the hierarchies and the timestamp-based limitations would have their own implications on the consistency checking mechanism and the distribution of change information. This would especially prove to be true for the commit mechanism, which appends the changes of a child work area to the changes of its parents.

Chapter 8

Paper E: Timestamp-based Consistency Checking of Collaboratively Developed Engineering Artifacts

This paper discusses global consistency checking of engineering artifacts at different timestamps. The considered timestamp can be adjusted. On this basis we gain different timestamp-based perspectives. These perspectives provide corresponding consistency information concerning a work area's contents.

8.1 Discussion

Whether live updates are desirable during an engineer's work procedure depends heavily on the respective work situation. Consider, for example, the work on a software architecture, where one team works on a largely independent module, another team works on the core and its interfaces and yet another team works on core extensions. Assume there are changes in a core interface. An engineer within the module team may not wish to be immediately disturbed by such changes. Eventually the engineer will have to adopt changes but - given that the module is largely self-contained and only uses core methods selectively - the necessary adaptations will be easily manageable. For such an engineer a live update concerning core changes may not be desirable. Engineers within the core extension team on the other hand must immediately know about changes in the core. The teams' work is deeper involved with the core and outdated interfaces may propagate through the engineers' work. For such engineers a live update of consistency information between their and the core team's work may be very desirable.

In the end however, both the core extension and the module team will have to adopt all changes before committing their work to a super-ordinate, respectively public, work area. Ideally, the extension team would have kept its work consistent throughout all live updates. The module team on the other hand may now encounter a few inconsistencies, as it updates its work with the latest core changes. For the correction of such inconsistencies it is important to know the exact changes that introduced them, ideally by a record of consistency states. Such a record documents the consistency state of a work areas content with regards to all possible versions of another work area. To create such a record the collaborative engineering environment requires a timestamp on each change.

This paper focuses on how to utilize a set of timestamps to generate different timestamp-based perspectives, providing a complete record of a work area's consistency states.

We discuss an extensive illustrative example to show the issues that can be tackled

with timestamps. The example covers individual steps in a typical engineering procedure, as well as ways how timestamps can be used in a beneficial manner therein. Potential problems within the procedure are described and respective research questions are defined. More precisely, these questions involve:

- Potential applications of timestamps in terms of different perspectives on the artifact storage (timestamp configurations).
- Potential re-iterations on the consistency checking mechanism within the context of such an application.
- The practical benefit of timestamp-related consistency checking in private work areas.

With regards to these questions the work re-iterates on various elements of the collaborative engineering environment's architecture. The paper expands on work areas, which are now equipped with a timestamp equivalent to their creation time. It further illustrates consistency-relevant artifacts and their structure. Eventually the work discusses timestamp-based consistency checking and introduces three different timestamp configurations:

- PWA Timestamp: Under this configuration the engineer can only see artifacts and changes from the public work area if they were created before the private work area.
- PA Timestamp: Under this configuration the engineer can see the latest state of the public work area, including all artifacts and changes.
- Intermediate Timestamp: Under this configuration the engineer can see artifacts and changes from the public work area up to a certain, defined timestamp.

For all these configurations the paper re-iterates on the consistency checking mechanism. In particular it discusses the data gathering step, where artifacts, respectively their properties, must be collected from the artifact storage, depending on the selected timestamp configuration. From these collected properties, respectively their timestamp-relevant changes, the consistency checker can then compute consistency information. The storage of this information is discussed as well, since multiple possible configurations mean multiple consistency results. A single result field in the CRE - as it was the case in earlier papers - is insufficient at this point. Instead, multiple result fields for the respective configurations are provided (with the exception of intermediate results, which are only relevant for intermediate review and therefore omitted).

To evaluate the practical benefit of our approach and timestamp-based consistency checking, this paper provides a case study on ten different real-world UML models. These models are incrementally loaded into the collaborative engineering environment, where consistency checking is performed after every change. We examine, how often a consistency check requires access to the public work area, as required properties can only be found there. This illustrates, how often a PA timestamp configuration is more beneficial than a PWA timestamp configuration in a simple setup, where an engineer's work is highly involved with ongoing changes (i.e., we simulate the core extension team's work from above).

8.2 Results

In this paper we provided details on timestamp-based consistency checking. Such a mechanism allowed us instant consistency feedback on heterogeneous artifacts in private work areas, under the consideration of differently timestamped changes.

Our evaluation showed, that on average a fifth of all consistency checks must be completed with data from the public area. This means that work areas containing changes that are highly involved with ongoing public changes, should definitely take advantage of the PA timestamp configuration. If a work areas content isn't relevant to ongoing changes the engineers could fall back on the PWA timestamp configuration.

We further discussed our approach with regards to run-time complexity, correctness and applicability. Complexity settles at a manageable $O(n)$. For correctness we tested our implementation against 2994 generated user scenarios. In terms of applicability we referenced an earlier study, which involved multiple types of engineering artifacts.

8.3 Contributions & Extension

In this paper we expanded the concept of work areas with a set of timestamp configurations. With these configurations, the individualized perspective on public artifacts could be further modified, by limiting it to a certain range of changes. These ranges could either involve the latest state, the private state or any intermediate state of artifacts. After discussing the inner workings of the collaborative engineering environment's artifact storage, we could move on to representing more complex hierarchical structures within this storage. These hierarchical structures would be more representative of modern engineering projects and their team organization. Naturally, a more complex structure of work areas would require equivalent re-considerations in terms of consistency checking. This would especially be true for the distribution of consistency-relevant change information throughout the work area hierarchy.

Chapter 9

Paper F: Instant Distribution of Consistency-relevant Change Information in a Hierarchical Multi-Developer Engineering Environment

This paper discusses the distribution of consistency information in a hierarchy of different private work areas. It describes how committed information must be propagated throughout the hierarchy to keep private work areas - respectively their consistency information - correct and up-to-date.

9.1 Discussion

In previous work we discussed the concrete realization of consistency checking information being stored in a collaborative engineering environment. Such information was stored in work areas that could be connected in a simple (single level) parent/child relationship. However, in practical application such simple relationships cannot represent modern engineering projects. Projects are often divided into different teams (e.g., mechanical branch, electronics, software, etc) which are again organized in different sub-teams (calculations, circuit design, front-end implementation, etc). Coordination between these sub-teams usually happens on a more abstract level (higher in the hierarchy), whereas coordination between team members must naturally happen on a more concrete level (lower in the hierarchy). To represent this type of team organization we expand the capabilities of work areas, in terms of hierarchies. In this paper we abstract the implementation for the simple child/parent relationship between the private and public work area into a generic relationship between arbitrary work areas. This way we can connect any set of work areas in any arbitrary hierarchy and properly represent modern engineering team organization. Of course this also expands the engineering environment's capabilities in terms of collaboration. In this paper different teams can organize their work in joint work areas before publishing it to a higher work area, such as the hierarchy root (i.e., the public work area). Such additional capabilities require equivalent adjustments on the side of the consistency checking mechanism. It is necessary to propagate the consistency information through the hierarchy in a generic way, so that the differently organized sub-teams (respectively, sub-hierarchies) always stay up-to-date.

In this paper, all work areas are technically treated the same, however, conceptually, we distinguish two different types:

- **Single work areas:** These work areas are actively used by the engineers to synchronize their work with the collaborative engineering environment.
- **Joint work areas:** These work areas are organizational elements that coordinate and collect the changes of different subordinate work areas or work area hierarchies (which are again organized in single and joint work areas). These work areas are usually only storage for different commits by the single work areas. They are controlled by team leaders or other responsible engineers.

With joint work areas essentially acting as a “passive” storage for commits, we re-iterate on the work areas’ reaction towards incoming change events in this paper. We provide two different ways to (either manually or automatically) trigger the stream of such events:

- **Commit:** All changes stored in a work area are transferred to the super-ordinate work area. There the changes are appended to the respective artifacts. This action is equivalent to the regular child/parent commit scenarios from previous work.
- **Update:** All changes from a super-ordinate work area are “made visible” for the respective updating work area. This updates the perspective on the artifact storage according to the super-ordinate perspective. The possibility to do this manually is given, so teams can first test and coordinate their work internally before coordinating it with other teams. Previous work generally assumed automatic updates of work areas.

With this way of structuring work areas into hierarchically updating and committing storage elements, the collaborative engineering environment can provide a sophisticated branching mechanism. Changes can be isolated in sub-hierarchies and can be actively checked against public knowledge, before committing a sub-team’s work to the rest of the engineering team. This happens by committing their joint work area into the root or public work area.

In this context the paper discusses the final data model for our approach. We re-iterate on different artifact storage elements, with special focus on the work area. We further re-iterate on artifact declaration elements and consistency checking elements. For the distribution of change information we discuss how layering of information is done within a hierarchy. We especially re-iterate on the algorithm for change analysis and how the consistency checking mechanism handles change information from the perspective of a single work area.

We eventually discuss various commit scenarios that may arise during the handling of information. Due to the layering of artifacts, each work area and its super-ordinate hierarchy constructs a perspective for the engineers that is individualized by their personal changes. Consistency information must be in accordance with these perspectives. This means, that in different work areas consistency information may vary. The scenarios discussed in this paper are focused on how the consistency checking mechanism must re-evaluate its results and distribute them throughout the collaborative engineering environment. Committing a work area plays a special role in this regard, since it potentially alters another engineer’s perspective. A commit may change the information in another work area’s super-ordinate hierarchy. Most importantly the following kinds of commit scenarios – regarding CRDs, context elements and scope elements – do so:

- **Committed creation:** If a CRD is created in a super-ordinate work area, the consistency rule set for a subordinate work area changes – CRE creations become necessary. If a context element is created a CRE might have to be created alongside (if a corresponding rule is existent in a work area). If a scope element is created a re-evaluation is necessary.

- **Committed modification:** Modifications of artifacts in a super-ordinate work area generally require re-evaluations of CREs in subordinate work areas. The exception are work areas with an overlapping modification on the same artifact. Due to layering these modifications overrule any modifications in super-ordinate work areas. Subsequently, consistency information does not have to be re-evaluated.
- **Committed deletion:** Deletions of CRDs or context elements usually require a cleanup of CREs (unless there is an overlap, similar to the modification scenarios). A deletion of scope elements requires a re-evaluation.

The paper discusses these scenarios in more detail and defines a set of post-conditions that has to hold after every commit:

1. A CRD must always be accompanied by all corresponding CREs.
2. A CE must always be accompanied by its individual corresponding CRE.
3. A CRD as well as a CE must always be retrievable from the perspective in which a corresponding CRE resides.

Being “accompanied” by a CRE means, that the CRE, respectively a change on its result property, resides in the same work area as the accompanied artifact.

We evaluate the final iteration of our consistency checking mechanism and the distribution of consistency-relevant change information through an exhaustive scenario simulation.

9.2 Results

In this paper we re-iterated on various aspects of our consistency checking mechanism to achieve the immediate distribution of consistency information on heterogeneous artifact changes in a hierarchy of work areas. This consistency information is computed from the perspective of a certain work area, under consideration of changes in its respective super-ordinate work areas. We evaluated our mechanism in an empirical case study. We conducted this case study on the basis of an exhaustive scenario simulation. For this simulation, we constructed a directed graph illustrating all possible user inputs that could be executed on the collaborative engineering environment (based on the environment’s interface). We then combined the different sequences of inputs into an exhaustive set of all possible input scenarios. These scenarios were automatically fed to the collaborative engineering environment. After every input we automatically checked the resulting state of the work areas against the post-conditions defined in this paper. In total our evaluation executed 2994 scenarios. In all of these scenarios the post-conditions held.

9.3 Contributions & Extension

In this paper we examined how work areas could be generalized for a hierarchical organization of the artifact storage. We went into detail how consistency information must be distributed within such a hierarchy and discussed a set of critical scenarios. In these scenarios engineers published their work to super-ordinate work areas and our mechanism re-evaluated affected work areas correspondingly.

For future work, single work areas in a hierarchy could be grouped. In these groups consistency checking could be executed on an entirely collaborative basis, i.e., inconsistencies among group members are immediately pointed out, irregardless of any hierarchy. This would enable a more direct way of live collaboration within teams and sub-teams. Naturally, it would require a further re-iteration of the consistency checking mechanism, particularly with regards to the data gathering process during the re-evaluation.

Chapter 10

Paper G: Ensuring Safe and Consistent Co-Engineering of Cyber-Physical Production Systems: A Case Study

This paper discusses an application of our global consistency checking approach during a case study conducted with industry partners. We utilize a collaborative engineering environment to synchronize different kinds of real world engineering artifacts. We manually build links between these artifacts and check certain consistency rules on them. This paper is a journal extension of earlier work [21], which was updated according to the latest standards of our implementation.

10.1 Discussion

This paper discusses a practical application of our approach. It describes a case study, in which the consistency between a software controller and its corresponding electrical models is upheld. The utilized collaborative engineering environment is equipped with a respective consistency checking mechanism. The data was provided by a leading company in the area of production automation. The paper discusses different engineers involved in the creation of a conveyor belt. It goes into detail on the activities of these engineers and how their engineering artifacts are related. The discussion of the engineer's work procedure describes how inconsistencies can arise and why support for their detection is crucial. The goals of the paper can be summarized as an enhanced documentation and utilization of relationships between different engineering artifacts. We establish traceability between artifacts and identify their consistency. Crucial core technologies that had to be covered for this were Microsoft Excel, EPlan Electric P8 as well as the Eclipse IDE. For the synchronization of the engineering artifacts with the collaborative engineering environment we could employ one of two principles:

- **Tool Adapters:** As described in previous papers, these adapters incrementally synchronize changes directly from a tool to the collaborative engineering environment. The synchronization happens at tool run-time.
- **File Adapters:** These adapters synchronize engineering artifacts on the basis of saved files. Whenever an engineering tool stores its engineering artifacts on the hard-drive the delta between the previous and the current file is detected and synchronized with the collaborative engineering environment as a change. This synchronization

approach is comparatively complex, however, the application of regular tool adapters is always dependent on the availability and extent of a tool's API.

For the case study, the available tool API's are sufficient. Therefore, we employ regular tool adapters. The paper describes each of these tool adapters and discusses various implementation details. For the synchronization of Excel data we rely on annotated cells, which are observed by the adapters and whose contents (respectively changes thereof) are transferred to the artifact storage. This is done to avoid unnecessary synchronization of irrelevant data. In the Eclipse IDE a similar concept was applied. Annotations are added through comments and synchronization happens after the compiler is executed. For EPlan Electric P8 we rely on custom properties that mark the respective model elements for synchronization. Relationships between marked elements are automatically synchronized. Type information, as it is required for synchronization with the collaborative engineering environment, is drawn from the respective meta-models of each kind of engineering artifact. Links between different engineering artifacts are established as explicit artifacts (respectively artifact types) themselves. We rely on three types of links for traceability:

- Excel to Code
- Excel to EPlan
- EPlan to Code

These links are established manually with the help of a standalone linking tool. Building upon these links, the paper describes consistency rules that can further support traceability among certain connected engineering artifacts. For this, the consistency checking mechanism observes whether the engineering artifacts correctly refer to linking artifacts. More precisely, when an EPlan element links to another EPlan element, the corresponding code elements must be linked as well. Otherwise an inconsistency is recorded. Additionally, links among the same types of engineering artifacts are checked against self-references. The paper then continues to discuss this approach with regards to a challenge problem given by the cooperating company.

10.2 Results

This paper showed how our approach could be used to a) provide traceability for engineering artifacts and b) utilize this traceability information in the context of consistency checking. In terms of traceability the approach provided the possibility to represent both pre-defined links (e.g., links between EPlan elements) as well as custom links between different types of engineering artifacts (e.g., Excel to EPlan links).

In terms of consistency checking, the traceability information could be used to navigate between different engineering artifacts and realize consistency rules concerning the relationships between them. Such (global) consistency checking was feasible as long as the corresponding mechanism operated on a syntactically uniform basis. The collaborative engineering environment could provide this basis as engineering artifacts were incrementally synchronized with it.

Using the information won in traceability and consistency checking, we provided comprehensive change impact analysis. By incrementally analysing the consistency of their linked artifacts, the engineers could quickly identify related components affected by their changes. Managers at the cooperating company were confident about the approach and made it clear that more involved support for engineers was appreciated.

10.3 Contributions & Extension

In this work we provided a case study describing a practical application of global consistency checking in a real world scenario. This practical application was part of a cooperation with industry partners. In this cooperation, consistency was upheld between different heterogeneous engineering artifacts and immediate feedback was provided.

The case study in this paper was mainly concerned with consistency information being distributed between individual engineers. These engineers were treated in a flat organization, i.e., there was no hierarchy involved in the propagation of information. While for a case study on global consistency checking this was reasonable, hierarchical team structures would have been more representative of modern day engineering projects. For future work, such structures should, if possible, be considered during a case study. Such a study could provide us with further information about the flow of consistency information between teams and sub-teams. This information could prove useful for further improvements of our approach by giving us insight into how teams publish their work and how they coordinate it with others in practice.

Chapter 11

Conclusion & Outlook

The world of engineering is becoming more and more complex. With increasing scale traditional models of project planning are facing severe organizational issues. The amount of interdependent properties in various engineering artifacts makes it highly difficult to account for project consistency. The complete distribution of consistency information to all affected engineers is an equally challenging problem.

Complementary to existing ways of collaboration, this work proposed the adaption of a collaborative engineering environment, as well as the computation and distribution of consistency information therein. The thesis at hand, showed the benefits of such a way of collaborating and illustrated how it could be motivated, conceptualized, developed and utilized for the distribution of consistency information. We provided a study that shows the beneficial effect of such information on the engineer's work. We further outlined the functionality of a collaborative engineering environment. We went into structuring an algorithm that could work on a data representation which incorporates heterogeneous engineering artifacts. Thereafter, we constructed a way of layering different changes on these artifacts in a way that would produce a unique perspective on an artifact storage. These layers of changes - respectively work areas - could eventually be connected in arbitrary hierarchies. From the perspectives constructed by these hierarchies we would compute consistency information and distribute it to engineers. To show a concrete application of our concepts we presented a case study conducted with industry partners. This study also shows that with relatively little effort an appreciated benefit could be drawn from our consistency checking mechanism, respectively from the distribution of its results.

The outlook for this work is twofold, as both the consistency checking mechanism as well as the collaborative engineering environment can be expanded in various ways. With regards to consistency checking, an extension of work areas concerning a grouping mechanism may prove useful. Consistency information would then primarily be determined on the basis of changes within a group of work areas, while secondarily it could rely on any work area hierarchy super-ordinate to the group. Further research that could be conducted on the collaborative engineering environment may include automated traceability through heuristics. For this, semantically overlapping concepts of engineering artifacts could be analysed. Artifacts could then automatically be linked and used by other analysis mechanisms. Process management could be accomplished within the engineering environment through the adaption of corresponding tools. For optimization, processes could then be analysed on the basis of the involved engineering artifacts. For such and many other interesting topics, research may benefit from a collaborative engineering environment.

Part II

Papers

Paper A

An Empirical Study on the Impact of Inconsistency Feedback during Model and Code Co-changing

Published Journal of Object Technology; Volume 18; Number 2; Pages 10:1–21;

Authors Georgios Kanakis, Djamel Eddine Khelladi, Stefan Fischer, Michael Alexander Tröls, Alexander Egyed

Abstract Model and code co-changing is about the coordinated modification of models and code during evolution. Intermittent inconsistencies are a common occurrence during co-changing. A partial co-change is the period in which the developer changed, say, the model but has not yet propagated the change to the code. Inconsistency feedback can be provided to developers for helping them to complete partial co-changes. However, there is no evidence whether such inconsistency feedback is useful to developers. To investigate this problem, we conducted a controlled experiment with 36 subjects who were required to complete ten partially completed change tasks between models and code of two non-trivial systems. The tasks were of different levels of complexity depending on how many model diagrams they affected. All subjects had to work on all change tasks but sometimes with and sometimes without inconsistency feedback. We then measured differences between task effort and correctness. We found that when subjects were given inconsistency feedback during tasks, they were 268% more likely to complete the co-change correctly compared to when they were not given inconsistency feedback. We also found that when subjects were not given inconsistency feedback, they nearly always failed in completing co-change tasks with high complexity where the partially completed changes were spread across different diagrams in the model. These findings suggest that inconsistency feedback (i.e. detection and repair) should form an integral part of co-changing, regardless of whether the code or the model changes first. Furthermore, these findings suggest that merely having access to changes (as with the given partially completed changes) is insufficient for effective co-changing.

1 Introduction

Model-Driven Engineering (MDE) has shown to be effective in the development and maintenance of large scale and embedded systems [4, 45]. Models are used in all development stages, from specifying the customers' requirements, design, all the way to source code. The benefits range from increased productivity to reduced time to market [27]. These benefits, however, hinge on the assumption that model and code remain consistent over time which is often not the case. Changes in models and code cause inconsistencies and subsequent errors if these inconsistencies are not recognized in a timely manner [29]. Any conclusions or automation based on them cannot be trusted for as long as these inconsistencies remain. Inconsistencies are better detected with today's consistency checking mechanisms [56]. However, literature tends to treat inconsistencies as errors - typically as errors within models (and their representations). Indeed, there is very little discussion about inconsistencies between model and code due to evolutionary changes. That is, model and code inconsistencies are often the result of evolutionary changes: changes where the model might have been evolved in response to changing requirements or defects, but the code has not been updated to reflect those changes. Here, inconsistencies imply that either model, code or both are out of date. Consequently, inconsistencies between model and code are cues about the incomplete propagation of changes from model to code or code to model. These inconsistencies occur intermittently - after, say, the engineer changed the model but before the engineer propagated the change to the code. The questions we are interested in here are: do engineers benefit from inconsistency feedback during model and code evolution? Or more concretely: does this inconsistency feedback benefit effort (i.e., save time) or does it improve quality (i.e., result in more complete change propagations)? Or both? The goal of this paper is to explore these questions. To examine the usefulness of inconsistency feedback during model and code co-change, we performed a controlled experiment with 36 subjects - all bachelor level students at the Johannes Kepler University but with average professional work experience of almost 2 years (many students do work part-time). The subjects were confronted with model or source code changes and they were then asked to complete those changes (i.e., to repair the inconsistencies that had been caused by those changes). The key distinguishing factor was that subjects were sometimes given inconsistency feedback and other times not. This way we could measure whether the availability of inconsistency feedback was beneficial to the subjects while they repaired the inconsistencies. Specifically, we measured their effort and their correct/complete propagation.

To ensure wider applicability, we asked each subject to work on ten change request tasks covering two, non-trivial software systems. Five tasks were about changes to an open source monopoly game called "Matador" and the other five tasks were about a proprietary calendar application called "Calendarium". Each task could come in two possible change directions: with the model-to-code change direction, subjects were presented with model changes and were asked to update the code (i.e., propagate changes from model to code). With the code-to-model change direction, subjects were asked to update the model in response to code changes (i.e., propagate changes from code to model). The change direction, task order, and availability of inconsistency feedback was randomized for each subject. Hence, every subject was asked to perform some change propagations with and some without inconsistency feedback; some from model to code and some from code to model; and all tasks in a different order. Out of the possible 360 tasks (if all 36 subjects had managed to complete all ten change requests), we obtained 281 completed solutions - 7.9/10 completed tasks per subject on average. Our findings showed that subjects working with inconsistency feedback solved 268% more correct tasks than those without inconsistency feedback (59 correct tasks with consistency feedback as opposed to 22 tasks without feedback). We found that the correctness improved regardless of the type of project or

the direction of the change. Hence, this benefit is solely the result of the availability of inconsistency feedback. Inconsistency feedback thus helps engineers to correctly propagate changes between model and code.

Surprisingly, the findings seemed to show little difference in effort for subjects with and without inconsistency feedback. A deeper analysis revealed that, when deprived of inconsistency feedback, subjects mostly failed to solve certain tasks entirely. We found that this applied particularly to tasks with higher complexity where the changes affected multiple diagrams of a model (e.g., engineers had to understand a class and a sequence diagram change together to understand how to co-change the code). Our results imply that inconsistency feedback is an essential complement for change propagation between model and code. Merely understanding model changes is not sufficient to understand how to co-change the code and vice versa. Given the strong correctness benefit, we argue that the use of inconsistency feedback should be common practice not just for error detection but also be change propagations/co-evolution given that there is almost no cost involved in using consistency checking mechanisms (e.g., Nentwich et al [68], Egyed [29], Musuvathi et al [65]) and given that many commonly usable consistency rules are available [86]. We believe that there are likely subsequent benefits that stem from the quicker availability of inconsistency feedback for more correct models or code which we did not measure since it was out of the scope of this paper. Given that we chose two reasonably complex projects, ten diversely complex change request tasks, two task directions, and different kinds of UML diagrams, we believe that our findings have significant value to the software engineering community. The rest of the paper is structured as follows. Section 2 discusses related work in the area of consistency checking and similar experiments. Section 3 presents the experimental set-up in-depth and the variables that were taken into consideration. Section 4 reports the results and the findings of the experiment. In Section 5, we discuss the threats to validity and how threats were mitigated or avoided. Finally, Section 6 concludes this paper and discusses possible areas of future research.

2 Related Work

To the best of our knowledge, no empirical study ever investigated the benefits of inconsistency feedback for engineers between code and models. However, there is significant related work on consistency checking mechanisms and consistency rules. Krishnan and Kellner [56] reported the relation between software defects and the lack of software process consistency in an empirical study. The study was based on 45 products from leading software vendors which adopted various practices of the Capability Maturity Model (CMM). Their focus was on the number of produced defects in a product when the vendors do not consistently follow the processes described in the CMM. The authors found that consistent adoption of CMM practices and personal capability of the team reduce the number of product defects in a developed system. Mäder and Egyed [60,61] conducted a similar experiment to measure if traceability benefits software developers when they evolve and maintain a software system. [61] extended the work presented in [60] and presented the effect on performance that the subjects had with the use of traces during their maintenance and development tasks. They found that effort and quality improved if subjects had available traceability. While traceability links are not the same as consistency rules, they do both imply dependencies - albeit in different ways. Hence, it can be argued that any manner which helps developers understanding relationships say between model and code, is useful.

Spanoudakis and Zisman [82] discovered that different stakeholders produced inconsistencies in overlapping models in their survey. They presented their findings as parts of a process for identifying and resolving inconsistencies. They also presented processes for identifying overlapping models and, how to diagnose inconsistencies on these overlapping

models. Finally, they described the handling, tracking and management of inconsistencies for these overlapping models. Usman et al. [92] presented consistency checking techniques for UML models in their survey. They found that nearly all consistency checking techniques are based on rules and monitor these rules on the UML models. They analyzed each tool with their defined parameters e.g. Nature of consistency checking, UML diagrams, UML version etc. and they classified the techniques according to their intermediate representation.

These studies have shown the potential of consistency checking and fixing mechanisms but they did not show the effect for engineers when correcting inconsistencies between model and code. This is the objective of our empirical study to explore any benefit of such mechanisms. Thus, the nature of our study differs from the previous work described in this section.

3 Method

This section details the method used in our controlled experiment with student subjects and allows its reproducibility. The subjects had to manually complete change requests between model and code based on consistency rules provided to them. For some of these change requests, we provided inconsistency feedback. For others we did not. In total, subjects had to complete ten such change requests with different degrees of complexity. The task complexity was based on the number of diagram types a change request affected - simple tasks affected one type of diagram, complex tasks affected two or more types of diagrams. The premise was always the same: the subjects were given an already changed model or code together with a change description; and they were asked to propagate the changes to the unchanged code or model. This is where the consistency information came into play. Inconsistency feedback becomes relevant after the model or code changes. The experiment thus explored if the availability of inconsistency feedback made a difference in propagating the initial changes. We define inconsistencies as situations where engineering artifacts are contradictory. Since, this experiment focused on model and code, inconsistencies imply that model and code are out of synchronization. Our working assumption was that the initial change to the model or code was correct and the inconsistencies was the result of not yet having propagated the change to the code or model respectively. Inconsistencies are defined through consistency rules - which describe a condition that must hold [92]. Since our experiment focused on models and code (specifically UML class, state and sequence diagrams as well as Java source code), we considered consistency rules between model and code applied in this experiment, Table 1 shows these six rules. For example, for a sequence diagram the order of calls among messages must correspond to the sequence of calls in the code. A change request came in two variations: either the code was changed by us and the subjects were asked to resolve inconsistencies with the model; or the model was changed by us and hence subjects had to resolve inconsistencies with the code. Regardless of task, project or change direction, the consistency rules were always the same.

To control the experiment, a laboratory was used. The allocated time slot for the experiment was three hours. Since the laboratory could not accommodate all subjects at once, we performed the experiment three times in a row with roughly 12 subjects each. The data were analyzed as one set because each setup was identical and this experiment should not be considered a family of experiments [4] but rather as one experiment.

The three groups' experiment parameters were kept constant. A procedure was set up that followed a predefined schedule. Instructions were given to subjects orally. A demo change request was done together with the subjects to familiarize them to the tasks. An initial session of 20 minutes was reserved for this. Subjects could ask questions. Since, we considered each subject's individual effort and correctness as part of the experimental

Table 1: Consistency rules used in the experiment from Riedl et al. [76]

Diagram Type	Consistency Rule
Class Diagram Consistency Rules	CR1: The structural features of the class diagram to be represented in the code (vice versa)
	CR2: The code class references and inheritance must be present in the class diagram (vice versa)
Sequence Diagram Consistency Rules	CR3: Order of calls in the Sequence Diagram must correspond to the code (vice versa)
	CR4: Methods' calls from a code class must be represented to the lifeline of that class (vice versa)
State Chart Diagram Consistency Rules	CR5: The transition between states are methods with condition and must be represented in the code (vice versa)
	CR6: The code execution state space must follow the state chart diagram's state sequence (vice versa)

analysis, subjects were not allowed to interact with one another during the experiment.

4 Subjects

The subjects performed the experiment as a supplementary part of a software engineering course. We asked them to volunteer for this experiment and their participation was not a mandatory part of their course. We did not exclude any volunteering student from the experiment. We had 36 subjects, all bachelor-level computer science students at our university: 31 were male and five were female. Their mean age was 23.7 years with the youngest subject having been 20 years old and the oldest subject having been 38 years old. Their professional programming experience ranged from 0 years (no experience) to 17 years. The average professional experience for the subjects was two years. As computer science students, all had extensive academic programming experiences; all were familiar with Java and UML. The subjects had an average Java experience of three years and about one year of UML modeling experience. The results from these subjects can be generalized for the target group of junior engineers.

4.1 Independent Variables

The experiment aimed to measure the effort and correctness of subjects while they resolved the inconsistencies stemming from the change requests. Every subject received the same change request tasks - half the tasks with inconsistency feedback and the other half without inconsistency feedback. Every subject thus had tasks with and without feedback. Furthermore, every subject had to work on both change directions - half the tasks focusing on model-to-code changes and the other half focusing on code-to-model changes. This setup avoided accidental groups of "good" subjects or "bad" subjects.

Table 2: Summary of Projects in Model and Code Size

Project	Lines of Code	Number of Classes	Different Diagrams	Diagrams	Model Elements
Matador	5.674	37	Class, Sequence, Statechart	6	661
Calendarium	21.016	150	Class, Sequence, Statechart	12	2.843

We prepared ten tasks for two different projects to ensure that our findings were not limited to a specific project or a specific task. Each project had five tasks. Each task involved types of UML diagrams. Moreover, the tasks were of different complexity. Two tasks affected one UML diagram type; two tasks affected two UML diagram types and one task affected all three UML diagram types. Each change request task has a model-to-code and code-to-model direction with or without inconsistency feedback. Hence, the experiment’s independent variables were: *the project, the change direction, the (availability of) inconsistency feedback and the complexity of the tasks*. The independent variables are described in detail next.

Projects We used two different projects to ensure that the results are applicable beyond a specific project. The first project was an open source monopoly game called “Matador” and the second project was a proprietary calendar application called “Calendarium”. Both projects were implemented in Java and they were selected because of the availability of source code and different UML diagrams types. The game project *Matador* is a Monopoly-like game with some alterations from the original Monopoly game. It is quite small with <6KLOC across 37 Java classes. Altogether with six diagrams and 661 model elements were available. The calendar application *Calendarium* is larger with 21KLOC across 150 Java classes. It has 12 UML diagrams and 2843 model elements. Nevertheless, the size of these project was ideal for a three hours experiment while being diverse and complex enough. Note that the sizes of the projects, as seen in Table 2, vary among tasks because of the changes we made to trigger the experiments. The data set is archived in the FigShare platform¹ to be used for reproducibility purposes.

Change Direction In model driven engineering, programmers work side-by-side with modelers. A modeler could thus initiate a task by changing the model which the programmer then needs to implement in the code. Or a programmer could perform the requested changes first which the modeler will then document in the model. Consequently, inconsistencies may arise from either model changes or code changes in both directions. Regardless, these inconsistencies need to be resolved. Thus, for each of the ten tasks, we created two change directions: a task could either be initiated by model changes that need to be propagated to code; or by code changes that need to be propagated to the model. Each subject was randomly assigned a change direction for each task. Every subject thus had five model to code and five code to model change requests.

Inconsistency Feedback The experiment examined the impact of having knowledge about inconsistencies between model and code. Thus, another independent variable was the availability of such knowledge. We considered consistency rules that cut between model and code. Consistency rules imposed by the specific languages for their artifacts are not violated by the created tasks and they are respected by the participants. Table 1 presented the rules that must hold between the model and code. Table 3 presents example messages for each consistency rule that subjects had to consider as part of the inconsistency feedback.

Complexity of Task We choose ten change request tasks with increasing complexity. We define the task complexity to be related to the number of diagrams a task is affecting. Multi diagram tasks are more complex than one diagram tasks since multiple elements must be consistent not only between model and code but also across the affected diagrams. On the code side, multi diagram tasks usually affect multiple classes and methods in them in order to achieve consistency between the model and the code. Four change requests were designed to affect only one kind of diagram, e.g., class diagram only or statechart diagram only. Four more change requests were designed to affect two types of diagrams, e.g., class and sequence diagrams or sequence and statechart diagrams. The last two change request tasks were designed to affect all types of diagrams, e.g., class, sequence and statechart diagrams. The tasks covered actual change requests that can occur in a

¹<https://figshare.com/s/cb1a69751b68de5cc6a7>

Table 3: Example of inconsistency feedback per rule provided to the subjects

Inconsistency	Example Message
Violation of CR1	Methods getTransaction() and getConnections are missing in the class ConnectionManager in "dblayer class diagram"
Violation of CR2	Missing relationship to Interface Controller from classes Game Controller, Board Controller, Player Controller ...
Violation of CR3	Messages lock(), commit() and processSql() in "DBLayer sequence diagram" are called in the wrong order as it is the code in the method deleteEntityObject()
Violation of CR4	Messages lock(), commit() and processSql() in "DBLayer sequence diagram" are called from the wrong lifeline
Violation of CR5	Transitions in the "PersistentBroker State Machine diagram" are not declared as methods in the class PersistenceBroker
Violation of CR6	State generated by the payDept() not present in the statechart diagram "Matador_Statecdiagram"

project development phase where change can affect multiple type of model diagrams and code places. Every task is designed to be totally independent from the others task of the project.

4.2 Dependent Variables

The experiment measured the performance of subjects resolving inconsistencies when inconsistency feedback was provided in contrast to when it was not provided. To measure the performance, we observe two variables: 1) *the time (effort)* subjects required to perform each change request task and 2) *the correctness* of each provided solution. The time variable describes the duration a subject required to complete a task. The time was measured in minutes and it was calculated by subjects recording the start and end times for every task they worked on. The correctness variable describes the correctness of the given solution with respect to resolved existing inconsistencies. Correctness is a categorical variable with three discrete values, *incorrect*, *partially correct* and *correct*. Each solution required multiple changes to be performed for the solution to be complete. We split the changes into three groups of related changes (e.g, method renaming was one group, class relationships was another group, etc.). Subsequently, we assessed how many of these groups the subjects had completed. Tasks for which subjects did not complete any group were considered incorrect. Tasks for which one or two groups were completed were considered partially correct. Finally, tasks for which all three groups of changes were completed were considered as correct. Correctness was assessed by three people, two authors and an external person.

4.3 Experiment Variations

The experiment was designed with two projects in mind that every subject would have worked on, each with five change request tasks. Each task has a direction (model change to code or code change to model). A change direction was either supplemented with inconsistency feedback or it was not. Hence, there were $2 \times 2 \times 5 = 20$ possible settings how the experiment could unfold (following the factorial design [45]). While the tasks were randomly assigned to subjects, we ensured that all possible task variations were included. However, every setting had the tasks of each project being grouped together. Thus, a subject could start with Matador or Calendarium project and had to complete its tasks before the subject could move to the next project.

4.4 Experiment Tool

The experiment was conducted using the Eclipse Integrated Development Environment (IDE)² in the Java development edition. We installed the UML Designer plug-in³ for the UML model of the projects. We did not use any other tools for the experiment since the Eclipse IDE with the modeling plug-in could serve both developer roles (Modeler and Programmer). The use of a standard IDE was preferred over the development of a specialized tool due to the support for both types of work artifacts for the experiment and the familiarity this tool had among all subjects.

4.5 Procedure and Material

Before the start of the experiment, we provided informational material to subjects. The material contained a questionnaire about the demographic information of the subjects. We also provided a demo task to illustrate an example of the problem and the solution (another task which was not used in the experiment but served for training purposes only). Finally, the subjects received the list of ten tasks they had to complete. The experiment did not require the subjects to be aware of the functionality of the projects. We spent the initial 20 minutes of the experiment for a briefing. The briefing included the subjects filling in the required demographic information and then proceeding to explaining the purpose and scope of the experiment. This briefing also included, a presentation about the Eclipse IDE and its modeling perspective for the UML model of the task. During the presentation of Eclipse IDE, we also discussed the demo task to provide a concrete example.

Main Experiment Part Following the briefing, the subjects started the main phase of the experiment. In this phase, each subject had to solve the ten tasks in the assigned order. Initial test runs (conducted with other students beforehand) showed that the tasks were solvable in roughly three hours. Therefore, we considered three hours working time to be sufficient for the experiment. For each task, subjects had to record the start and end times on their own task sheets. The changes had to be implemented using the Eclipse IDE. Both the task sheets and the workspaces of the Eclipse IDEs (this is where the model and code is stored) were collected at the end of the experiment to measure time and correctness.

After the experiment The final step was to check the answers provided by the subjects for correctness and to record the data in an excel sheet for further statistical analysis. In this step, we filled the column with the correctness value after checking one by one the answers of the subjects. Correctness was checked by three people independently. Differences were discussed and resolved to derive a single answer for each task: Correct, partially correct, or incorrect. The recorded time was also stored in the same excel sheet for the analysis phase. Unfinished tasks were rejected from the analysis process since they could not be measured in respect to time and correctness.

5 Analysis and Results

This section analyses the results we gathered from the performed experiment. The research questions provide the motivation for the following analysis of the results.

5.1 Research Questions

In this experiment we are investigating the effect of inconsistency feedback provided to engineers resolving inconsistencies. We will answer the following research questions.

²Eclipse IDE website: <https://eclipse.org/>

³Plug-ins website <https://marketplace.eclipse.org/content/uml-designer>

Table 4: ANOVA analysis of the effort depended variable Time

Variables	Df	Sum Sq	Mean Sq	F value	Pr (>F)	Signif.
Feedback	1	161.6	161.62	4.8151	0.029199	*
Direction	1	1	0.8	0.024	0.87679	
Project	1	174.1	174.09	5.1864	0.023671	*
Complexity	9	169	18.7	0.581	81181	
Feedback:Direction	1	329.6	329.61	9.8198	0.001948	**
Feedback:Project	1	293	293.1	9.093	0.0029	**
Residuals	195	6127.3	31.422			

1. What is the effect of inconsistency feedback on correctness when engineers are presented with inconsistent models and code changes?
2. What is the effect of inconsistency feedback on time (effort)?
3. Does the complexity of the tasks affect time and correctness?

5.2 Overall Analysis

For the analysis of the collected data, we used the R statistical analysis environment [85]. We choose to analyze the dependent variable time with the ANOVA [27] method for the analysis of variance. ANOVA is based on the assumptions of normal distribution and homogeneity of variance. To test this, we performed the ShapiroWilk's [80] test for data normality in the continuous depended variable time ($W=0.99408$, $p\text{-value}=0.4749$, $=0.05$). The test provides the null hypothesis that the data are normal and the p-value confirmed it.

Since, the data was normally distributed, we performed the Bartlett's homogeneity test [81]($K\text{-squared} = 5.3777$, $p\text{-value} = 0.1461$) which provided no evidence for lack of the variance homogeneity. We have considered the following formula for the ANOVA analysis for the time depended variable

$$Time \sim Feedback + Change Direction + Project + Task Complexity \quad (1)$$

For the ANOVA analysis, we observe six different columns, the degree of freedom (Df) which is the number of freedom of a variable, the sum of squares (Sum Sq) is the sum of squares of the deviations of values from the expected value, the mean squares (Mean Sq) is the division of Sum Sq by the Df, the F value which is the ration between two mean squares that forms the basis for a hypothesis test, the p-value the probability to obtain the statistic F value if the null hypothesis is true and the Significance(Signif) where it shows the effect of a independent variable to the outcome of a dependent variable.

In Table 4, we see that the inconsistency feedback independent variable has an effect on the time dependent variable in correlation with project and direction independent variables. This means that the time is affected by the provided feedback both when the project changes and when the direction is different. We need to examine in the detailed analysis to find out in what way time is affected. In this ANOVA analysis, we also see a variation in time between the subjects resolving tasks in the different projects. We expected that due to the difference in size and nature of the two projects. We also performed ANOVA on the second depended variable correctness. The formula used for the ANOVA is

$$Correctness \sim Feedback + Change Direction + Project + Task Complexity \quad (2)$$

In Table 5, we observe that the variance of correctness is mostly affected by the inconsistency feedback independent variable. This is interesting since multiple other variables are changing but they do not seem to have a large effect in the correctness depended variable. The only ones that have some effect into the dependent variables are the direction and complexity. Complexity was expected to have some effect since our subjects have different level of experience, providing an interest point towards the effect of direction to correctness.

After the analysis of the variance for the two dependent variables, we present the overall time and correctness results of the subjects participated in the experiment. In total, we had 360 tasks: 180 with inconsistency feedback and 180 without inconsistency feedback. The subjects completed 141 tasks with inconsistency feedback and 140 without feedback. In order to answer our first research question, we present the analysis in the correctness variable in subsection 4.3 where we examine overall correctness and correctness per independent variable.

5.3 Research Question 1

Revisiting our first question, we are interested in identifying the effect of inconsistency feedback for subjects correctly resolving change tasks. Table 6 presents the correctness of the subjects in fixing inconsistencies at the requested tasks. We observe that subjects with inconsistency feedback have solved almost three time more correct tasks than without inconsistency feedback. This overall 268% increase is seen by having 59 correct tasks for subjects with inconsistency feedback and only 22 correct tasks for subjects without inconsistency feedback. The partially correct tasks remained at the same levels for subjects with and without inconsistency feedback.

Table 6 showed overall correctness. In the following, we will further examine the correctness with respect to the different projects and the change request direction. We expect from the variance analysis above that we would not have any significant difference from the overall analysis for the independent variables Projects and Direction seen in Table 5.

Table 7 verifies the findings from the ANOVA analysis about the correctness improvement due to inconsistency feedback. We see that in both projects, inconsistency feedback improved task correctness more than 200% with partially correct task remaining in the same levels for both groups. It is interesting to observe that the correctness with inconsistency feedback is better for the larger project “Calendarium” than compared to the smaller project “Matador”. This suggests that the importance of inconsistency feedback improves with project size but more data is needed to confirm this.

Table 5: ANOVA analysis of the Correctness dependent variable

	Df	Sum Sq	Mean Sq	F value	Pr (>F)	Signif.
Feedback (Feedb.)	1	13.69	13.695	25.137	1.19E-06	***
Direction (Dir.)	1	3.59	3.587	6.584	0.0110	*
Project (Proj.)	1	0.89	0.892	1.638	0.2021	
Complexity (Compl.)	9	11.97	1.330	2.441	0.0118	*
Feedb.:Dir.	1	0.01	0.010	0.019	0.8912	
Feedb.:Proj.	1	0.00	0.003	0.006	0.9380	
Dir.:Proj.	1	0.25	0.246	0.452	0.5024	
Feedb.:Compl.	8	4.23	0.528	0.970	0.4610	
Dir.:Compl.	8	8.72	1.089	2.000	0.0483	*
Feedb.:Dir.:Proj.	1	0.64	641.000	1.176	0.2795	
Feedb.:Dir.:Compl.	8	1.39	0.174	1.176	0.9579	
Residuals	197	107.33	0.545			

Finally, as can be seen in Table 8, the different change directions do not affect the correctness of inconsistency feedback regardless of direction. The benefit is respectively 285% and 237% for both model to code inconsistencies and code to model inconsistencies. Thus, we conclude that both designers and programmers can benefit equally from inconsistency feedback in keeping model and code consistent.

5.4 Research Question 2

Next, we are discussing the effect of inconsistency feedback on the effort that engineers need in identifying and resolving inconsistencies. We measure the effort with the time subjects needed for completing their tasks.

In Table 9, we see that the mean overall time that subjects required to complete the tasks without inconsistency feedback is in fact smaller than those with inconsistency feedback. This is the case for all above scenarios for tasks (all task, correct tasks, correct and partially correct tasks). In the case of the correct task, we also observe that the medium time is almost identical. We will explore this phenomenon with the time variable further down in subsection 4.5 where we present a more detailed analysis on the effort required by the subjects to complete the tasks with and without inconsistency feedback.

Table 10 presents the mean time of the subjects completing task correct or partially correct between the two projects. We observe that in both projects the mean time of the subjects is slightly higher for those having inconsistency feedback than those without inconsistency feedback. We expected a variation between the subjects with and without inconsistency feedback based on the variance analysis of the time dependent variable when correlated to the inconsistency feedback. According to the ANOVA analysis, we expect a variation for the time measurements between the different directions of the tasks. In Table 11, we observe that the model to code change request tasks have benefited in time from the inconsistency feedback which was expected since a model change usually is required to be implemented with a specific way to the code. Interestingly, the code to model direction is far from being considered benefited by the inconsistency feedback since there is large time difference between subjects with inconsistency feedback and subjects without it. In order to understand what is happening with the time measurements, we are going to limit the comparison groups to only correct tasks to see if the same results are present.

We again observe in Table 12 that subjects without feedback took less time than the subjects with inconsistency feedback. This is interesting because we expected before the experiment that the subjects with the inconsistency feedback will require less effort to complete the tasks. Thus, we need to examine deeper into the correct tasks to reveal the causes of such diversion from our expectation. From Table 10 and Table 11, we see that the time was not favorably affected by inconsistency feedback. However, not all tasks have the same complexity. Therefore, we investigated more carefully how time and correctness are influenced by the tasks' complexity. The findings are presented into the subsection below where we present the effect of complexity to the effort of subjects completing tasks and what kind of tasks were able to complete with and without inconsistency feedback.

Table 6: Overall Correctness

Correctness	Inconsistency Feedback	No Inconsistency Feedback	Incr
Correct	59(41.84%)	22(15.71%)	268%
Partially Correct	49(34.75%)	50(35.71%)	0%
Incorrect	33(23.4%)	68(48.57%)	-51,5%

Table 7: Correctness per Project

Matador	Inconsistency Feedback	No Inconsistency Feedback	Incr
Correct	34(44.74%)	15(20%)	226,7%
Partially Correct	18(23.68%)	24(32%)	-25%
Incorrect	24(31.58%)	36(48%)	-33,4%
Calendarium	Inconsistency Feedback	No Inconsistency Feedback	Incr
Correct	24(36.92%)	7(10.77%)	342%
Partially Correct	26(40%)	26(40%)	0%
Incorrect	15(23.08%)	32(49.23%)	-46,9%

Table 8: Correctness per Direction

Model to Code	Inconsistency Feedback	No Inconsistency Feedback	Incr
Correct	40(47.06%)	14(23.33%)	285,7%
Partially Correct	30(35.29%)	23(38.33%)	130%
Incorrect	15(17.65%)	23(38.33%)	-65%
Code to Model	Inconsistency Feedback	No Inconsistency Feedback	Incr
Correct	19(33.93%)	8(10.00%)	237,5%
Partially Correct	19(33.93%)	27(33.75%)	-70,4%
Incorrect	18(32.14%)	45(56.25%)	-40%

Table 9: Overall Time Measurements

Mean Time (min)	Inconsistency Feedback	No Inconsistency Feedback
All Tasks	21.86	19.11
Correct Tasks	22.90	22.64
Correct & Partial Correct	22.94	20.46

Table 10: Mean Time per Project for Correct and Partial Correct results

Mean Time (min)	Inconsistency Feedback	No Inconsistency Feedback	Diff
Matador	24.34	20.87	3.47
Calendarium	21.90	19.97	1.93

Table 11: Mean Time per Direction for Correct and Partial Correct

Mean Time (min)	Inconsistency Feedback	No Inconsistency Feedback	Diff
Model to Code	19.88	24.43	4.55
Code to Model	28.5	16.26	12.24

Table 12: Mean Time per Project and Direction for only Correct Tasks

Mean Time per Project (min)	Inconsistency Feedback	No Inconsistency Feedback	Diff
Matador	22.06	23.80	1.74
Calendarium	24.30	20.14	4.16
Mean Time per Direction (min)	Inconsistency Feedback	No Inconsistency Feedback	Diff
Model to Code	20.08	14.21	5.87
Code to Model	28.68	17.75	10.93

5.5 Research Question 3

Initially, we examined the effort subjects used based on the complexity of the tasks in order to investigate their effect on the mean time for the subjects with inconsistency feedback. In our experiment, we included tasks that are of different complexity. For each project, we have two tasks that are simple and affect only one kind of diagrams, two that affect two kinds of diagrams and one task that affects all three kinds of diagrams in the model. We grouped these tasks into simple tasks (i.e. one diagram type affected) and complex tasks (i.e. two or three diagram types affected). We also observed in Table 9 that the time subjects took to complete tasks are relative better for those without feedback. However, this does not unveil the effect of the tasks' complexity on the overall mean time for correct tasks.

In Table 13, we present the correct tasks divided by their different complexity with and without inconsistency feedback. We see that the number of complex tasks being resolved correctly are higher (31) with consistency feedback than without inconsistency feedback (10). We also observe that when feedback was provided, the number of complex correct tasks is higher than the simple ones (31 vs 28). Whereas without inconsistency feedback the number of complex tasks is lower than the simple tasks (10 vs 12). Table 14 similarly shows that the mean time for tasks divided by their different complexity with and without inconsistency feedback. It shows that for simple tasks when inconsistency feedback is not provided, their mean time is significantly lower than when inconsistency feedback is provided. This explains the overall mean time being relatively better for the subjects without inconsistency feedback since the simple tasks were the most resolved tasks in that case. Herein, the overall mean time without inconsistency feedback is mainly affected by the time subjects required for simple tasks.

Furthermore, in Table 14 we see that the mean time for correct complex tasks is lower for subjects with inconsistency feedback. This suggests that subjects benefited by inconsistency feedback when resolving complex tasks. However, when inconsistency feedback was not provided, the number of correct complex tasks per affected types of diagrams is statistically insufficient (e.g., two tasks for class*sequence) to draw conclusions.

Figure 1 presents the number of correct complex tasks based on the different UML diagram types when providing inconsistency feedback or not. There are four types of tasks that affect multiple diagrams. The tasks for all three diagrams and for a class and sequence diagrams are present in both projects. The class*statechart task is present only in the "Calendarium" project while the sequence*statechart is present only in the "Matador" project. This explains their lower number in Figure 1 in contrast to the other two combinations. We choose to have the class*sequence task interleaving between the two projects because there are the most commonly used diagrams in UML modeling. The class*sequence*statechart task is also present in both projects since it is the most complex task. Figure 1 shows that all different complex tasks benefit from inconsistency feedback. In particular, the correct class*sequence tasks and class*sequence*statechart

Table 13: Correctness for the different Complexities of Tasks

Correct Task Analysis	Inconsistency Feedback	No Inconsistency Feedback
Class	9	3
Sequence	12	6
State	7	3
Total	28	12
Class*Sequence	10	2
Class*State	3	2
Sequence*State	5	2
Class*Sequence*State	13	4
Total	31	10

Table 14: Mean Time for the different Complexities of Tasks

Correct Tasks Mean Time	Inconsistency Feedback	No Inconsistency Feedback	Diff
Class	23.1	18	-5.1
Sequence	17.50	12.60	-4.9
State	25.10	28.33	3.23
Total Mean Time	23.71	13.17	
Class*Sequence	22.30	68.00	45.7
Class*State	23.00	23.50	0.5
Sequence*State	23.20	15.50	-7.7
Class*Sequence*State	21.30	20.00	-1.3
Total Mean Time	22.10	28.30	

were respectively five and three times higher when inconsistency feedback was provided.

We performed the same analysis for the simple task to see which UML diagram type is more affected by the inconsistency feedback. We have class diagram tasks present only in the “Calendarium” project while we have statechart diagram tasks present only in the “Matador” project. Sequence diagram tasks are present in both projects. Figure 2 shows the number of corrects simple tasks per diagram type. We have twice as many sequence diagram tasks compared to the other two diagram types, because we included this task type in both projects. Another interesting observation is that sequence and statechart diagrams are favored around two times when inconsistency feedback is applied while the class diagram type tasks are improved by three times. This subsequently leads to class diagrams being the most favored among simple tasks when inconsistency feedback is present. Figure 2 shows that all different simple tasks benefit as well from inconsistency feedback.

6 Threats to Validity

In this section, we discuss the threats to validity ([35], [47]) of this study and explain how we mitigated these dangers in this experiment.

6.1 External Validity

A possible external validity threat is the use of students as subjects in the experiment. Recent studies [79, 83] have shown that students can be valid subjects for experiments and that students are well representative when it comes to developing tasks. To reduce

Figure 1: Correct Complex Task Analysis

Figure 2: Correct Simple Task Analysis

the effect of this threat, we included students with a variety of professional and academic experience in programming and modeling. We had subjects with minimum knowledge of programming and modeling and subjects which are considered experienced programmers.

From the subjects used in the experiment, one may conclude that our generalization group is junior engineers. This limitation is present in the use of students as subjects but we expect that consistency feedback will be beneficial to senior engineers working on larger projects. Thus, we expect to have similar findings for also more experienced engineers but reproducing the experiment with practitioners is required to make conclusions.

Another possible threat is the chosen projects. We chose to have two projects which were both implemented in Java but we expect that this does not affect the results in any other programming languages since consistency checking does not focus on programming language specific aspects but rather on the existence of key modeling aspects that should also be found in the code.

Our study was performed on projects that were medium and large for the scope and the three hours time subjects were asked to participate. Both projects had models that consisted of three type of diagrams: class, sequence and statechart diagrams. The projects were applications in different nature, a game and a calendar, thus, we can avoid the possibility of the results to be applicable only to the specific nature of a project. The

choice to avoid larger more complex (> 1M LOC) projects was deliberately taken because in complex, real-world systems, engineers often have indepth knowledge of the system they are designing and maintaining. Thus, they are likely to know where to propagate changes when inconsistencies arise. By choosing smaller projects, we aimed at mitigating the subjects' unfamiliarity and difficulty to understand the projects.

However, the fact that we found strong benefit in correctly resolving inconsistencies in projects of this size, implies that more complex projects will be equally (or better) affected by the use of consistency feedback to resolve inconsistencies. We also believe that in more complex projects, we can gain effort as well since the complexity of the inconsistencies will be higher than in the projects used in our experiment. Finally, we choose to use a real life IDE for the experiment in order to imitate real working environments and condition for the experiment and the subjects.

6.2 Internal Validity

To avoid variability in knowledge between the subjects, we made sure not to provide any information about the experiment prior to the date of the experiment. All material was made available to the subjects during the beginning of the experiment and they were kept at a minimum. We asked the subjects after the briefings if they had any questions but apart from some organizational questions, they claimed to be ready for the experiment. The subjects had no prior knowledge of the projects. They could also not know about the tasks which were created by us for the purpose of this experiment.

To avoid the Hawthorne effect, we used subjects from a course that the authors were not tutor or lecturers so that the subjects will not feel examined from their teacher. We also during the briefing and invitation to participation that their performance in the experiment will not be assessed within their course. We confirm them that their performance was only relevant for metrics within our research group and for only research purposes.

The design of the study is crossover study, a variant of within-subject design in which all participants are exposed sequentially to all variances of the tasks. This design minimizes the number of participants required to identify statistically significant differences between the results [49]. This design has the learning effect as a threat. This threat was mitigated by assigning the tasks to the subject in different order. Thus, one subject could get a task as first whereas another subject would get it as last. All tasks were assigned in all different places.

The time that subjects could use to solve the tasks was limited to three hours in total without limitation per task. The time limit was chosen by preliminary test runs (with different subjects). Most of the 36 subjects have completed all of their task, which suggests that the assigned time was sufficient for the average subject. We also did not allow the students to communicate with each other and we provided different set up for each one of them.

We had created the initial changes and inconsistencies, allowing us to control the complexity of each task based on the affected UML diagram types. To reduce the threat of meaningless change requests we created real life inconsistencies from Table 1 between code and models. The goal was to investigate the effect of consistency feedback on the resolved tasks.

Finally, to avoid that completed tasks were incorrectly categorized, we performed the correctness check independently by three different people. In case of diverse opinion among the three, a discussion session was held for concluding the correctness of the specific task.

6.3 Construct Validity

The experiment aimed to evaluate the effect of consistency feedback in resolving inconsistencies between collaborating software modelers and software developers. The problem

is caused when the selected dependent variables do not relate to the target of the experiment. Inconsistencies usually require the developer's time to be resolved and not all of them are surely resolved correctly. Thus, we chose to assess the effect by measuring the effort and correctness of the subjects resolving the introduced inconsistencies. We measured the effort with the time subjects required to complete each task. Faster time in resolving inconsistencies implies less effort by the subject whereas correctness expects the subjects to resolve all inconsistencies between model and code based on the defined consistency rules.

6.4 Conclusion Validity

Possible threats to the statistical conclusion validity are low statistical power, low effect size and violation of the assumptions for the statistical procedure. To avoid these threats, we worked with 36 subjects, created equally sized groups with randomized task assignment. Before the data analysis, we verified the normality assumption using the Shapiro-Wilk test and we used the Bartlett's homogeneity test to verify homogeneity over the time data. Later, we performed the ANOVA analysis of time and correctness variables. However, for a finer grain analysis, such as for the effort (time) of subjects for complex tasks, we need to reproduce the experiment and gather more data points. This is future work.

7 Conclusion

In this paper, we investigated the usefulness of inconsistency feedback during the co-change of models and code. We conducted a controlled experiment with 36 subjects that were asked to resolve inconsistencies between model and code. We designed ten change requests tasks of different complexity for two diverse in nature projects. Each task has two directions of co-change one from model to code and one from code to model. Every direction has two variations with consistency feedback provided or not. For each project, we had a model with three types of UML diagrams (class, sequence, statechart diagrams) and its Java source code.

We found that consistency feedback provides a significant effect in correctly resolving inconsistencies. Subjects with consistency feedback resolved correctly 26 tasks than those without consistency feedback (59 with consistency feedback versus 22 without). This effect was similarly observed for any direction and any project. Effort was not significantly affected in the overall analysis, leading us to explore deeper the causes for it. Thus, we examined the cause of the time not being affected by consistency feedback focusing on the complexity of the tasks. We found that subject with consistency feedback have resolved correctly three time more complex tasks (31 over 10). Interestingly, with consistency feedback subjects resolved more complex tasks than simple tasks (31 vs 28). Whereas without consistency feedback subjects resolved more simple tasks than complex tasks (12 vs 10). The mean time for simple tasks was significantly lower when consistency feedback was not provided. This resulted in the overall mean time being relatively lower for subjects without consistency feedback. However, the mean time for complex tasks was lower when consistency feedback was provided.

Our findings provide evidence that the consistency feedback, when present, can benefit developers and designers in correctly resolving inconsistencies. In particular when correcting more complex tasks. As future work, we plan to reproduce our experiment to gain more evidence especially to further investigate the effect of consistency feedback on time. At the second experiment, we will add an additional post-study questionnaire to see the level of trust from subjects to the consistency feedback provided.

8 Acknowledgement

The research leading to these results was funded by the Land of Upper Austria - Joint International PhD Program in Informatics under the grant number Wi-2014-202477/6. Funded by the JKU Linz Institute of Technology (LIT) the state of Upper Austria, grant no. LIT-2016-2-SEE-019 and Pro2Future, a COMET K1-Center of the Austrian Research Promotion Agency (FFG), grant no. 854184. Received funding from Rennes Métropole under grant AIS no. 190270.

Paper B

Collaboratively Enhanced Consistency Checking in a Cloud-based Engineering Environment

Published EICS '19: Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems; June 2019; Pages 1–6;

Authors Michael Alexander Tröls, Atif Mashkooor, Alexander Egyed

Abstract Software systems engineering involves many engineers, often from different engineering disciplines. Efficient collaboration among these engineers is a vital necessity. Tool support for such collaboration is often lacking, especially with regards to consistency between different engineering artifacts (e.g., between model and code or requirements and specifications). Current collaboration tools, such as version control systems, are not able to address these cross-artifact consistency concerns. The consequence is unnecessarily complex consistency maintenance during engineering. This paper explores consistent handling of engineering artifacts during collaborative engineering. This work presumes that all engineers collaborate using a joint, cloud-based engineering environment and engineering artifacts are continuously synchronized with this environment. The artifacts can be read and modified by both engineers and analysis mechanisms such as a consistency checker. The paper enumerates different consistency checking scenarios that arise during such collaboration.

1 Introduction

Today’s engineering landscape is filled with a multitude of widely used tools, meeting the diverse needs of engineers. In software engineering alone, we have tools for implementation, architecture & design, requirements engineering, testing and more. Collaboration among engineers is essential, yet the growing number and complexity of engineering artifacts captured through their tools makes this increasingly difficult. A single engineer is unlikely to be aware of, let alone understand or interpret, all artifacts [21]. This is in part caused by today’s focus of collaborative tools, on single types of engineering artifacts. For example, version control systems such as Subversion¹ or Git², were originally developed for sharing code. It is technically possible to store different engineering artifacts in such systems as well, but the text-based nature of their mechanisms hinders a serious, fine-granular integration. This shortcoming results in a lack of information concerning artifact changes and their respective implications, meaning that the version control system may document changes, but not their intricate details (e.g., one could infer that a design document was altered but not which diagram or property within). This makes the analysis of changes rather cumbersome.

One major problem in this regard is, that inconsistencies - which are bound to arise, due to the heavily interdependent nature of engineering artifacts - become very hard to detect, before the integration of an engineer’s work with the work of others. There have been various attempts at tackling consistency checking in general (e.g., [37, 38, 66, 74, 76, 94]), but the issue described beforehand makes it clear to us, that a mechanism providing more immediate feedback is required. There is a need for consistency checking enhanced by a collaborative environment and while some approaches have gone into that direction (e.g., [51, 77]) they mostly consider only homogeneous engineering artifacts or require costly merging operations.

This paper presents a collaboratively enhanced consistency checking mechanism to better support the collaborative efforts of engineers. The approach is embedded in a cloud-based engineering environment, focusing on the integration of tools and synchronizing captured artifacts, to enable comprehensive reasoning, while engineers continue to use the tools they always have.

The rest of this paper is structured as follows: In Section 2, we present the DesignSpace engineering cloud [25] as the architectural foundation of our work. Realized on this platform, we present our approach of a collaboratively enhanced consistency checker in Section 3. We evaluate this work in Section 6 presenting the results of an experiment and two case studies. The paper is concluded with an outlook towards the future work in Section 9.

2 The DesignSpace Engineering Cloud

The DesignSpace is a cloud-based collaboration platform. It is organized into artifact storage and collaboration services. Artifact storage provides a central and uniform place where engineering artifacts are stored. These artifacts typically originate from the various tools that engineers use (e.g., programming tools, modeling tools, requirements tools). Through tool adapters, the artifacts within tools are continuously synchronized with the DesignSpace. Engineers continue to use the tools they already know.

Collaboration services augment and analyze the artifacts in the cloud to assist engineers during their collaboration. By immediately reacting towards changes in the artifact storage, services provide live feedback and corresponding guidance to engineers. Collaboratively enhanced consistency checking is one such service. It provides the means to

¹Apache Subversion: <https://subversion.apache.org/>

²Git: <https://git-scm.com>

reason over artifacts stored in the cloud and is able to provide customized consistency feedback on arbitrary types of artifacts. This effectively means that consistency rules can be formulated beyond the boundaries of a single tool and compare artifacts from different tools with each other (e.g., model and code). The basic architecture of the DesignSpace is illustrated in Figure 1. In the following we discuss the basic aspects of this cloud environment that build the architectural foundation of our presented consistency checking approach.

Figure 1: Illustration of multiple tools sharing artifacts on the DesignSpace and Services providing feedback

2.1 Uniform Data Representation

The artifact storage of the DesignSpace provides a uniform representation for arbitrary engineering artifacts. It ensures that engineers have access to all engineering artifacts and cloud services can parse them regardless of their origin. For this purpose, tool adapters translate the artifacts as they appear in the tools to a uniform data representation. This representation is a mapping between the properties of a data entity and their respective value. For example, a spreadsheet could be translated into the uniform data representation with properties for all relevant fields. The values mapped to these properties would represent the content of the respective fields. Likewise source code could be parsed such that classes, methods or fields become separate cloud artifacts where their properties reflect field or method names. The mapped values could be method bodies, return types, constant values or similar. The granularity of the artifact translation is defined in the tool adapters.

2.2 Live Feedback

As the engineering cloud is continuously updated with the incremental data, every change triggers an event that can be forwarded through the cloud. Subsequently, these events can be caught by cloud services, e.g., to trigger mechanisms, or sent to tool adapters, which can translate them into user feedback, e.g., screen warnings.

2.3 Artifact Typing

Artifacts within the DesignSpace must adhere to a defined structure. We refer to these structures as types. Types can be defined by tool adapters, specifying certain properties as well as the primitive data type and cardinality of their values. The users can then easily instantiate the types and fill their values via a method provided by the DesignSpace.

2.4 Linking

Since artifacts are uniquely identifiable within the cloud environment, they can also easily reference each other, by simply storing each other's ID. The consistency checking service can utilize these links to reason beyond the boundaries of a single engineering artifact's domain.

2.5 Private Work Areas

Before being accessible to all engineers in the public repository (PR), artifact changes are synchronized with a private work area (PWA). This is an isolated view on the engineering artifacts the engineers are currently working on in their tools - respectively the delta of these artifacts in relation to their publicly available version. Assume engineers wish to make changes to an artifact stored within the PR. They would first checkout the artifact to their tool through the tool adapter. There they would make changes to the artifact's properties. These changes would then be stored in the PWA. Once the engineers decide to publish the changes, they are incrementally added to the PR.

2.6 Collaboration Services

As artifacts become available in the cloud, they can be analyzed by services. These services may modify artifacts on their own (e.g., to repair an error) or provide structured feedback to engineers (e.g., to warn about conflicts).

3 Consistency Checking

Consistency checking in general, is a technique to validate whether engineering artifacts adhere to a certain standard. This standard is normally regarded in isolation, i.e., they only matter for a certain type of artifacts. For example, Java code adheres to certain rules defined through the Java metamodel, certain IDE's evaluate written code against internal standards and some UML environments make sure that models stay consistent with each other (e.g., by making sure lifelines in a sequence diagram correspond to a class in a class diagram). In broad terms, consistency checking differentiates between local and global consistency. The former describes the most common kind of consistency checking, as it only concerns itself with artifacts of a certain type adhering to internal standards (as described above). The latter, however, concerns itself with the consistency of several types of artifacts in relation to each other. Equivalently, one can also speak of intra- and inter-model consistency [89]. While systems for the former kind are rather frequent, the latter is rarely explored, mostly due to the lacking tool support. What is required for global consistency checking is a common ground on which engineering artifacts can be compared. Such a common ground is normally achieved through such activities as model merging. This is mostly connected with great efforts and thus rarely pursued. The DesignSpace, on the other hand, already provides a uniform data representation over which a consistency checker can reason. This is a major advantage since it allows us to countercheck changes on one typed artifact against values stored on a different type, e.g., one could synchronize

both code and design documents with the DesignSpace. The consistency checker can then compare concrete related values and warn engineers if there are discrepancies.

3.1 Basic Architecture & Runtime Functionality

Since we covered both the technical pre-requisites, as well as potential benefits of a consistency checker within a collaborative cloud environment, we now discuss the technical details of its implementation and execution.

Data Structure

The consistency checker uses the uniform data representation not only for reasoning, but also as a data structure for internal information. Technically, this gives us the possibility to evaluate the consistency checking data for inconsistencies, but more importantly it grants a tight integration with the rest of the collaboration environment. Therefore, changes to its data structure are adapted incrementally and the resulting events can be forwarded through the cloud, for example, to trigger mechanisms of other services. Likewise they can be forwarded to users, e.g., when a consistency rule is broken. The consistency checker requires two major data structures:

- *Consistency Rule Definition Artifacts*: These artifacts define rules as a simple OCL³-like strings. They, furthermore, define a context for the rule's evaluation. This context is an artifact type for which the rule has to hold. Within the rule the context is referred to as "self".
- *Consistency Rule Instance Artifacts*: These artifacts are instantiated for each artifact of a certain type referenced in a consistency rule definition. They store a context element (the concrete artifact instance of the context type) as well as a scope, which represents each artifact that is traversed by evaluating the consistency rule for the respective context element. Furthermore, the rule instances also hold the results of their individual rule evaluation.

As an illustrative example, consider the rule definition stated in Listing B.1. In it, the declared context is a `JavaClass`, respectively the artifact type corresponding to the name `JavaClass`. The rule can be read like a simple object access in object oriented programming languages, i.e., from the context (self) the rule evaluation navigates through a reference "UML" to the name of the referenced UML artifact and compares it to the name field found in the context artifact. The consistency checker will instantiate a Consistency Rule Instance Artifact for each Java class artifact instance and evaluate the said instance according to the defined rule.

Rule Evaluation

In the following we will discuss a typical rule evaluation for the proposed consistency checking mechanism. As aforementioned, the consistency checker will instantiate Consistency Rule Instance Artifacts according to a respective definition. Each such instance references a certain context element for which the defined rule is evaluated. Such an evaluation happens in various steps:

- *Triggering the consistency checker*: The consistency checking mechanism can be triggered through various ways. It can either be started manually, by user input, or automatically, e.g., by catching change events on scope elements.

³OCL: <https://www.omg.org/spec/OCL/About-OCL/>

- *Retrieving the rule:* Once the consistency checker is active, it must first retrieve the rule corresponding with an artifact. For this the artifact triggering the mechanism must first be identified within a consistency rule instance artifact. It does not matter whether it is the context element or a scope element. Both kinds of changes can have an impact on the consistency state of a project. Once the artifact is found within a rule instance the referenced rule definition is retrieved.
- *Traversing the artifact structure:* Once the rule is retrieved, it can be executed. The rule itself describes a path from a context element to property values, which are then compared according to whatever operation is defined within the rule.
- *Write back results:* Once the rule is evaluated, the results are written back onto the Consistency Rule Instance Artifact.
- *Feedback:* Results can be forwarded to tool adapters and other services. The interpretation of the results (e.g., triggering screen messages or service mechanisms) is up to them.

During the evaluation process, the consistency checker also updates the respective consistency rule instances scope. Every artifact the consistency checker passes during the traversal of the artifact structure is stored in the scope for future evaluations.

```
1 Context:    Java::Class
2 Rule:      self.UML.name == self.name
```

Listing B.1: A simple consistency rule comparing names between a JavaClass artifact and the corresponding UML diagram.

3.2 Collaborative Enhancement

With the deployment on the DesignSpace and the utilization of its architecture, the consistency checker can be enhanced through several collaborative features. In our approach, we covered the following:

- Full Activity Awareness
- Multi-Tool Consistency Checking
- Team-Driven Error Handling

In the following, we will discuss how individual features of the DesignSpace were exploited to achieve these features.

Full Activity Awareness

A way to collaboratively enhance consistency checking is to make the corresponding mechanism aware of each engineer's work. For our approach, three forms of awareness were established by deploying the consistency checker as a collaborative service:

1) *Artifact Awareness:* The most basic capability of an engineering cloud is to enable awareness of each others engineering artifacts. The lack of awareness of another engineers' work may lead to conflicts and unnecessary rework. This hinders collaboration as much as it hinders collaboratively enhanced consistency checking, which can in turn help us to reduce conflicts between engineering artifacts. Through the full integration of the consistency checker with the DesignSpace's uniform data representation the mechanism works natively on the cloud's data structure. Artifact awareness is given by providing the

service with full access to the artifact storage. Contrary to the regular users the service does not only overlook a single PWA, but all work areas, including the PR. This also grants the consistency checker the possibility to retrieve a full representation of an engineering artifact within the context of each engineer's individual PWA. Each such context can be seen as an individual view of the consistency checker on engineering data. Such a view can be defined as follows:

$$\begin{aligned} V_{cc}(A) &= V(A(P_m \cup P_n)) \\ P_m &= \{\forall p \mid p \in PWA\} \\ P_n &= \{\forall p \mid p \in PR \wedge p \notin PWA\} \end{aligned}$$

Where V_{cc} is the view of the consistency checker on an artifact A . This is equivalent to the view on two property sets P_m and P_n , both in union making up the artifact A . P_m is a property set of all properties that have been changed within a PWA and are thus only available there in isolation, whereas P_n is a property set of all properties residing *only* within the PR. Since they have not been changed they do not reside within the PWA. In short: In the context of full artifact awareness, the consistency checker can view an artifact comprised of its public version complemented by the individual changes of the engineers.

2) *Change Awareness*: An engineering cloud can provide change awareness so that engineers and collaboration services see changes instantly. This is especially useful when the engineers' works have strong, immediate implications on each other. In such a scenario immediate consistency checking - triggered by change events - can be of high value. For this to happen the collaboration service in our approach directly listens to change events fired by both the PWAs and the PR. Catching these change events triggers the consistency checker in one of two possible modes:

- *Private Change Evaluation*: These are triggered by change events fired in a PWA. The evaluated consistency information is written into the private version of the respective Consistency Rule Instance. The newly computed results are only visible to the engineer using the PWA.
- *Public Change Evaluation*: These are triggered by change events fired in the PR. Since consistency information is also stored privately - and subsequently committed to the PR - change events triggered by users publishing their private changes are ignored. Instead, this form of rule evaluation can only happen if a service changes the PR directly. The newly computed results are visible to all users and services.

Both modes can perform local or global consistency checks, since the extent of a check is dependent on the nature of the written consistency rule. If a consistency rule contains global elements, like a link towards different engineering artifacts not available within the PWA, the consistency checker will automatically retrieve the publicly available version of the corresponding artifacts and integrate it into the rule evaluation.

3) *Error Awareness*: Concluding its rule evaluation process, the consistency checker uses the direct connection between PWAs and tools to send the consistency feedback to all listening tool adapters as illustrated in Figure 1. The tool adapters can then visualize this feedback in a customized manner. In this context it is important to note that anybody can register for the error feedback of any PWA. This allows us to create a full error awareness for engineers, which can also be utilized collaboratively. For example, an electrical engineer and a software engineer could register to receive the consistency information of their respective changes. If, however, they choose to work as a group/team then they could choose to get feedback with regard to the union of the group's changes, while still working in isolation on their own engineering artifacts.

Multi-Tool Consistency Checking

A collaborative engineering cloud, such as the DesignSpace, gives its users the ability to augment tool knowledge with additional information, which tools are normally not able to capture themselves. To our understanding, one of the most crucial bits of additional information is the relationship between artifacts from different tools. Such relationships can be established in the DesignSpace through artifact links as aforementioned in Section 2.4. These links can be extensively utilized by a collaboratively enhanced consistency checker, especially with regards to multi-tool, respectively global consistency. Since links are defined through regular properties on an artifact, it is sufficient to refer to such a property in order to navigate from one artifact to another during the rule-evaluation process. Consider again the consistency rule given in Listing B.1. In it, we refer to a property called “UML”. In the DesignSpace this property can be defined as a reference towards a corresponding UML artifact. During the rule evaluation process the consistency checker will automatically navigate to the referenced artifact and continue the evaluation from there. Subsequently, our approach can rely on such links to define conditions beyond the boundary of a single artifact or even the engineering domain. Additionally, it should be noted, that the consistency checker can also be used to validate such links, e.g., by making sure the properties are not empty. If, alternatively, links are established through separate artifacts featuring both a source and target property, the consistency checker can - given proper rules have been established beforehand - evaluate whether these artifacts have been created correctly.

Team-Driven Error Handling

Concurrent modification of engineering artifacts leads to errors - errors that are increasingly expensive to fix the longer they stay undetected. A collaboratively enhanced consistency checker can support cross workstation/tool/discipline error feedback on a scale that is otherwise not possible today. Especially with the help of change events being fired in both PWAs as well as the PR, the consistency checker can analyze arising inconsistency immediately and send the corresponding feedback to engineers. With this, fixing errors can become a collaborative effort, as the inconsistency feedback contains exact information about broken consistency rules, affected artifacts and the workspaces that issued changes leading to inconsistencies. The latter bit of information is especially important, since it gives the engineers indications on who to communicate with when fixing a problem. As an additional note: Enforcing consistency rules can also be used to prevent errors in the first place, e.g., by pre-checking changes and rejecting them if they violate any rule. However, since a temporarily inconsistent state of the isolated work is normally tolerated, this form of rule enforcement might be regarded as too restrictive by the engineers.

4 Evaluation

The collaborative engineering cloud as well as the proposed collaboratively enhanced consistency checking mechanism have been evaluated via a prototype implementation in the context of an industrial experiment and two major case studies. In this section, we discuss them in more detail.

4.1 Prototype Implementations

The DesignSpace approach is different from much of its related work mentioned in Section 1, because it does not emphasize on certain collaboration styles but is meant to provide flexibility in collaboration. The goal is a freely definable and changeable collaboration approach where engineers can join groups, change artifact awareness, or engage arbitrary

processes. The current implementation supports a range of tool adapters, such as Java Eclipse⁴, IBM Rational Software Architect⁵ (for UML), Microsoft Excel (for calculations), Creo⁶ (for CAD Drawings), Eplan Electric P8⁷ (for electrical layouts), and others. These tools demonstrate convincingly that it is possible to represent arbitrary tool artifacts in a cloud. The current implementation of the consistency checker makes extensive use of linked artifacts from different tools and as such augmented links with error rules to detect inconsistencies on the fly.

4.2 Experiments & Case Studies

Our collaboratively enhanced consistency checking approach has been validated on the basis of an experiment as well as two case studies, which are discussed in the following.

Industrial Experiment

Our consistency checking approach has been beneficially utilized in an extensive industrial experiment with Van Hoecke Automation⁸. There, consistency has been secured between electrical models as well as their software controllers. The corresponding artifacts were made available in the DesignSpace and altered in parallel, while continuously being counter-checked against established consistency rules. Errors, respectively arising inconsistencies, were then fed back to the engineers, who worked with tool adapters for EPlan and Eclipse. With this the application of our approach could achieve full activity awareness. Multi-Tool consistency checking was established through various types of links, which were evaluated during the experiment as well. Artifacts representing spreadsheet cells, code and EPlan model elements have been linked manually through a separate tool. These links were typed artifacts and were not only used by the consistency checker, but also established traceability between engineering artifacts. This confirms not only the feasibility but adds to the usefulness of our approach, as traceability is an additional important issue covered through the DesignSpace. Both the detection of conflicting changes and also the prevention of errors has been done within the experiment. Feedback about inconsistencies was provided instantly after the consistency checker analyzed changes and evaluated them as erroneous. Alternatively, consistency feedback could be requested by the engineers manually.

FMTC Case Study

Full activity awareness was further evaluated during a case study with the Flanders's Mechatronics Technology Center (FMTC⁹). In this case study, Eplan Electric P8 drawings were provided by a third-party company. These drawings had to be kept consistent with their respective code implementations according to user-defined, domain-specific consistency rules. Through customly implemented tool adapters, the respective artifacts were integrated in the DesignSpace, giving validation to our approach with regards to its multi-tool aspect. Links between artifacts were created with a custom DesignSpace tool. After issuing a series of changes, instant consistency feedback was provided to engineers. Detected discrepancies between the drawings and the code were reported, evaluating the notion of full activity awareness as well as team-driven error handling.

⁴Eclipse IDE: <https://www.eclipse.org/>

⁵IBM RSA: <https://www.ibm.com/developerworks/downloads/r/architect/index.html>

⁶Creo: <https://www.ptc.com/de/products/cad/creo>

⁷EPlan: <https://www.eplanusa.com/us/2/>

⁸Van Hoecke Automation: <https://www.vha.be/>

⁹FMTC: <https://www.flandersmake.be/en>

ACCM Case Study

To evaluate our approach within a different engineering domain, a case study was conducted with the Austrian Center of Competence in Mechatronics (ACCM/LCM¹⁰). There, the mechanical calculation for a robot arm was shared and analyzed on the DesignSpace. Full activity awareness was given with particular focus on error awareness, as engineers were notified about changes that have been counter-checked against established consistency rules. With the help of the notifications problem solutions concerning arising inconsistencies could be worked out in a team-driven way. The consistency rules required the established links between artifacts from different tools. In particular, these links involved relations between UML models, CAD drawings and spreadsheet artifacts, validating our approach with regards to its multi-tool capabilities.

5 Conclusion & Future Work

Software systems engineering is a highly complex activity that integrates knowledge from a variety of disciplines. Engineers attempt to manage and break down this complexity by focusing on what they can do individually in the context of tools they are using. However, this results in knowledge fragmented across many tools and an increased risk of inconsistencies. This paper has shown that a collaboratively enhanced consistency checker can rectify this issue. Our proposed mechanism both contributes to and benefits from a cloud-based engineering platform, that integrates engineering artifacts from different disciplines. We discussed various aspects of collaboration, that are affected positively by our approach, most importantly activity awareness, multi-tool consistency checking and team-driven error handling. With regards to the future work, we would like to extend the consistency checker's capabilities under particular consideration of the following scenarios:

Selected Group Awareness: Sometimes consistency information of a specific group context (e.g., a certain team within a project) can be of interest. While engineers might be reluctant to counter-check the unfinished work with publicly available engineering artifacts, they are more likely willing to check work-in-progress against selected sub-results of their own workgroup. This can be achieved by changing the context of consistency checks to a selected set of artifacts. The results can help engineers to integrate their own work with the rest of the team.

Change and Reuse The systematic linking of engineering knowledge also benefits change and reuse. Imagine a project about a robotic system which has similarities with past solutions - for example, parts of a robot arm. With all engineering knowledge available in the cloud and with all its artifacts linked, engineers could now extract the said robot arm's specifications, use cases or associated code. Consistency checking can simplify this process, since reused engineering artifacts must still adhere to the same or similar consistency rules.

6 Acknowledgement

This work is supported by the Austrian Science Fund (FWF), grant no. P 31989-N31, the JKU Linz Institute of Technology (LIT), the state of Upper Austria, grant no. LIT-2016-2-SEE-019, Pro2Future, a COMET K1-Centre of the Austrian Research Promotion Agency (FFG), grant no. 854184 and the Software Competence Center Hagenberg GmbH.

¹⁰ACCM/LCM: <https://www.lcm.at/unternehmen/kompetenzzentrum/>

Paper C

Live and Global Consistency Checking in a Collaborative Engineering Environment

Published SAC '19: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing; April 2019; Pages 1776–1785;

Authors Michael Alexander Tröls, Atif Mashkooor, Alexander Egyed

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

1 Introduction

Contemporary software and systems engineering is a process that involves a wide range of tools, bringing together knowledge of many engineers. In doing so, engineers create a wide variety of engineering artifacts, ranging from requirements to design model specifications and code. With increasing number of tools and engineers involved, the number of interdependent engineering artifacts grows non-linearly. Maintaining consistency among artifacts of these separate tools is a challenging task [36]. We speak of global consistency checking.

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

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

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

Our approach was implemented as a cloud service utilizing the DesignSpace [25] infrastructure. In the cloud, merely one consistency checker implementation is needed rather than many implementations for each and every tool. Given that the cloud computes consistency feedback once for all artifacts, it is in fact more efficient than tool-centric solutions where each tool would have to compute the consistency feedback separately (to the extent they are capable of doing that). Empirical evidence demonstrates scalability and usability across a wide set of tools. Two industrial case studies provide further evidence of usefulness.

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

Figure 1: Sequence diagram of the language selection process

2 Problem Illustration

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

2.1 Initial State

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

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

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

2.2 Subsequent Changes Made by Engineers

The following describes a simple change scenario where the aforementioned engineers adapt their respective artifacts using their corresponding tools. The first implementation reveals shortcomings in the connection process to the server. When a user logs in for the first time, the server should request a language for displaying the catalog of available movies. The server then sends all textual descriptions accordingly. John records this request as an additional requirement. The adjustment of the requirement specification has direct impact on the UML design of the server. John's change makes it necessary for Alex to adapt his

UML design. The new process is illustrated in Figure 1. The framed parts of the sequence diagram are an addition to the existing design; however, the sequence diagram is but one scenario that describes the client/server communication protocol. As their company is progressive and aware of benefits of maintaining traceability between requirements and design, Alex needs to ensure that the changed/added parts of the design are mapped back to the requirements that motivated them.

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

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

2.3 Problem Statement

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

Due to the limited perspectives of engineering tools, current state-of-the-art tool-centric consistency checking mechanisms are not able to detect these inconsistencies. That is, the UML modeling tool representing the sequence diagram would be aware of Alex's addition of the messages and could quickly identify inconsistencies with other parts of the UML model. However, it could not depict how Alex's changes relate to the source code. A document-centric consistency checker would be able to provide more comprehensive consistency feedback but its use would be disruptive and certainly not live and incremental in providing up-to-date inconsistency feedback as engineers make changes available. Inconsistencies across tools may thus go undetected during software and systems engineering. This has several negative implications: 1) the end product may not correctly implement the new requirement, 2) the code may be harder to maintain because the UML model no longer documents the code correctly, and 3) there might be higher costs of fixing resulting problems [33].

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

3 Infrastructure

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

3.1 Overview

To overcome these issue, our approach uses the DesignSpace [25] infrastructure, which provides a central, versioned storage space for engineering artifacts. It provides a fine-grained, uniform, model / metamodel-style artifact representation. The DesignSpace infrastructure not only acts as a storage space for engineering artifacts but it also holds the links that connect engineering artifacts from different tools. By linking artifacts through a cloud environment both the problem of physical and logical separation of engineering information can be overcome. A pre-requisite for linking artifacts is a syntactic common-ground onto which links can build. This is achieved through a common uniform artifact representation into which artifacts are translated. As a result, it is possible to, e.g., link Java classes with the UML model elements that implement them. Once links are known, engineers may formulate consistency rules that define correctness conditions for artifacts from different tools. These rules could be written by the same engineers using the tools; though, it is more likely that they are predefined by domain engineers. Once linked in their common representation, engineering artifacts can furthermore be analyzed and manipulated by services such as a consistency checker.

An illustration for our global consistency checking mechanism interlinked with the DesignSpace's core concepts can be seen in Figure 2. Engineers are working with their respective tools: Eclipse for Java and IBM's Rational Software Architect (RSA¹) for UML models. There are adapters for these tools in order to automate the mapping of engineering artifacts to the uniform data representation. Furthermore, adapters automate the synchronization with the cloud environment. Links and consistency rules can be created and modified with separate tools. In our case the linking tool directly modifies the artifacts synchronized with the cloud. With the links set up our global consistency checking approach then validates the defined consistency rules. Inconsistencies are shared with the engineers in the form of network messages from the cloud, which can be directly fed into the engineers tool through the tool adapter.

3.2 Cloud Environment

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

- **Uniform Data Representation:** To provide a common format for distinct kinds of engineering artifacts, such as source code (which is usually textual), or models

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

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

(which is usually graphical), the DesignSpace stores data in a uniform representation.

- **Typing:** To provide a well-defined language (metamodel) and language constructs for data, the DesignSpace offers custom artifact types that can be instantiated dynamically.
- **Incremental Live Synchronization:** Tools may continuously synchronize changes made by the engineers, ensuring that the global consistency checker is always up-to-date.
- **Conflict Handling:** Mutually changed artifacts may stand in conflict with each other. The DesignSpace provides conflict handling strategies for such situations.

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

Uniform Data Representation

In the DesignSpace engineering artifacts (respectively parts thereof) are translated into a uniform data representation. The data structure itself is a generic mapping between keys and values, representing arbitrary properties of engineering artifacts. The keys are simple strings, while the values are primitive data types or references to other artifacts. For example, if we wish to translate a Java class into the uniform data representation we could use field names as keys and their default values as values. Something important to note is that such a mapping can as well represent links between artifacts, since artifacts in the uniform data representation carry a unique ID to be identifiable. This ID is then simply referenced in the property value. Any structured data that can be represented in such a manner can, furthermore, be uploaded to the DesignSpace. It should be noted that in no

way our uniform data representation is solely laid out for software engineering artifacts. It can as well be used for, e.g., electrical engineering artifacts. Hence, we do not use notation or terminology from a specific field. To simplify conversion into the artifact format the DesignSpace's API provides appropriate methods. All created artifacts, respectively their mappings, must correspond to a defined type. This means that artifacts have a fixed base structure.

Typing

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

Incremental Live Synchronization

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

For our running example John's tool adapter creates and instantiates types corresponding to the requirements he wishes to synchronize with the DesignSpace. The artifact type for requirements defines the necessary properties, e.g., requirement name, requirement ID and description. These types are then instantiated by the requirements tool adapter, whenever a new requirement is added. A part of this example is illustrated in Figure 1, where two instantiations of a requirement artifact type are created.

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

Conflict Handling

Engineers mostly do not modify the same artifacts concurrently. In case mutually overlapping changes exist, the DesignSpace applies user preferences specifically set for such situations. Users can either overwrite or ignore changes from other users. These options can be set for each user individually. Users could, for example, overwrite their own changes with information from one user, yet ignore conflicting artifacts from another one, keeping

Figure 3: Exemplary types and instantiations for requirements

their own changes intact. In the latter case, the user’s data would only be complemented by changes that are not in conflict.

3.3 Linking Engineering Artifacts

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

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

Engineers can use this linking tool to create arbitrary links. Principally, engineers define links manually through the linking tool. We refer to this as capturing explicit links.

Concerning our running example it may be interesting for John, which of his requirements affect which parts of Alex’s class diagrams. Likewise, Alex may be interested in the Java classes which are affected by his UML classes. In the type definition of their engineering artifacts they may even introduce a specific property to save such references, as given in Figure 4. A responsible engineer could then fill these references manually. Naturally, one could create a DesignSpace service software analyzing and manipulating engineering artifacts to automatically establish links between them [46]. However, this is beyond the scope of the approach presented in this paper.

4 Global Consistency Checking

With artifacts stored and linked in the cloud environment, we can proceed with checking their consistency. When engineers change artifacts through their tools, the consistency checker is notified. It then automatically identifies and validates the changes according to corresponding consistency rules. These consistency rules are provided by the users, e.g., a domain expert. The results of the consistency checker are then returned to the engineers respectively their tools. The tools may then provide specialized feedback to the engineers who benefit from the live consistency checking on their changes. Consistency checking is done in the cloud and engineers do not have to interrupt their work. Our implementation is similar to the incremental consistency checker discussed in [29]. The key differences are its integration with the unified data structure, its ability to change arbitrary artifacts, its ability to handle new types and consistency rules during runtime, and its ability to work seamlessly with tool artifacts and links.

4.1 Data Structure

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

Rule Definition Artifacts

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

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

Rule Evaluation Artifacts

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

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

4.2 Evaluation Process

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

Artifact Creation (see Algorithm 1)

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

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

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

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

Algorithm 1: Artifact Creation

```

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

```

Artifact Modification (see Algorithm 2)

If an artifact is modified, the cloud sends a notification to the consistency checking service. As a modification is always about a change to an artifact property, this notification contains

the changed artifact instance (its ID) and property name. If the artifact respectively property is listed in the scopes of rule evaluations then these rule evaluations are re-evaluated.

Algorithm 2: Artifact Modification

```

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

```

Artifact Deletion (see Algorithm 3)

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

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

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

4.3 Inconsistency Feedback

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

4.4 Example

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

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

Algorithm 3: Artifact Deletion

```

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

```

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

Consider the artifact structure depicted in Figure 4: To describe consistency rule CR1, we define the UML lifeline type as the context (TypeArtifact(ID: 20)). The corresponding condition then compares two sets, more precisely, the names of outgoing messages from the lifeline to the names of methods in the affected Java class. Note that in Section 3.2, we mentioned that artifact properties could also hold collections of values. In this case both message and method names could be held in a string collection. For all message names there must be a matching method name. The fully realized consistency rule can be seen in Listing C.1.

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

```

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

```

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

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

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

Consistency Rule CR2 is handled in a similar fashion. To describe consistency rule CR2, we define the Requirement Type as the context (TypeArtifact(ID: 25)). The corresponding condition utilizes the “Affects” link, making sure that it points towards an artifact. The condition could also be extended to ensure that the target of the “Affects” link corresponds to the right type - in this case a UML lifeline (ID: 20). In that regard, consistency rules can also cover basic well-formedness checks. One thing to consider here is, that requirements might affect several UML elements. So, if specified, we may want to check the super type (UML diagram) of UML Lifeline instead. On the requirements side we now need a value to compare this type to. Mind that the type of a field can be specified in the field definitions of an artifact (see Figure 1; e.g., the field “Name” is of type string, while the requirement ID is an integer). This can be utilized as a comparative value in our rule’s condition, if the aforementioned super type (UML diagram) is specified as the type of the “Affects” field (remember that types can be linked to other types and, therefore, reference them as type for their fields). The fully realized consistency rule can be seen in Listing C.2.

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

The first adjustment is John recording an additional requirement in his requirements tool. Like all tools in this example, John’s tool is incrementally synchronized with the cloud environment (see Section 3.2). This means, whenever he adds a requirement in the GUI of his tool, an artifact of the type “requirement” (see Figure 1 respectively Section 3.2) is automatically instantiated in the cloud. This action triggers the consistency checker’s “Artifact Creation” algorithm (see Algorithm 1) through a notification, since the “Requirement” type is a context in a rule definition (see Listing C.2). Subsequently the consistency checker will create a new Rule Evaluation artifact, with the newly instantiated requirement artifact as context element. This Rule Evaluation artifact will be evaluated immediately. Since the requirement is completely new and not yet represented within the UML design, the result of the consistency check will indicate an inconsistency and consistency rule CR2 as broken. This is reported to John through the GUI of his synchronized tool, e.g., by marking the corresponding requirement in an overview.

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

lifeline, respectively its operations) in Alex’s UML tool will be marked as inconsistent.

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

5 Validation

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

5.1 Existing Tool Adapters

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

Tool adapters have been written for software engineering tools (e.g., Eclipse, RSA and Papyrus), electrical engineering tools (e.g., EPlan Electric P8 [22] [32] and PTC Creo [32]), as well as for common purpose tools (e.g., Microsoft Excel and Visio). All of these tools produce very different engineering artifacts, including Java code, UML models, electrical models, CAD drawings, data tables and regular diagrams. Engineering artifacts synchronized with the DesignSpace adhere to different metamodels. Among others these include various tool-internal metamodels (EPlan and Creo), the Ecore metamodel and extensions thereof (Papyrus and RSA) as well as the Java metamodel. These metamodels, or parts thereof, are synchronized with the DesignSpace as well. Tool Adapters have been written in Java, C# and C++. Thanks to the adaptability of the DesignSpace API the range of potential tool adapter languages can easily be extended.

5.2 Links

In previous applications of the DesignSpace several different types of engineering artifacts have been linked together. For example, EPlan artifacts have been linked to Java code [22] and Creo CAD hardware components - respectively geometric shapes - have been linked to UML elements [32]. Note that these are very different engineering artifacts, coming from different engineering domains. Their linking acts as a proof of concept, that the semantic gap between engineering artifacts can be bridged with the help of the DesignSpace. Furthermore, this work demonstrated how requirements could be linked to UML elements and in turn to Java code.

5.3 Empirical Study

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

Figure 5: Comparison of total processing time

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

Table 1: Case study projects

the size of model and code respectively. Projects ranged from small, like an implementation of the “Game of Life” [44], to big multi-developer projects like the ArgoUML tool². The evaluation was based on 14 consistency rules, which can be found in [75].

We validated the computational scalability of our approach by systematically applying changes to all elements of our projects and by capturing the time required to re-evaluate all rule evaluation artifacts. We ensured that each element present in the scope of a rule evaluation artifact was changed. Mean and median times were observed and used for our analysis. Figure 5 shows the mean processing time per affected rule evaluation artifact. The mean observed processing time was 10.7 ms, while the median was 8.8 ms. The total processing time remain below 50 ms on average, which is an acceptable time for tool users [69].

5.4 Case Studies

Our approach was validated on the basis of two case studies. One was performed with the help of the Austrian Center of Competence in Mechatronics (ACCM³), another with the Flanders’s Mechatronics Technology Center (FMTC⁴). These case studies suggest the usability of our approach.

²ArgoUML: <http://www.argouml.tigris.org/> (last accessed: 06.12.2018)

³ACCM/LCM: <https://www.lcm.at/> (last accessed: 06.12.2018)

⁴FMTC: <http://www.flandersmake.be/en> (last accessed: 06.12.2018)

ACCM Robot Arm

This case study provides a live global consistency checking for the design, mechanical calculation and partial implementation of a robot arm [32]. Engineering artifacts involved were Excel sheets, CAD drawings, UML models based on IBM RSA and source code written in the Eclipse IDE. For explicit linking, the case study's implementation also provides the ability to define and check traceability links between different involved engineering artifacts.

FMTC

This case study provides live global consistency checking between electrical circuit diagrams as well as an implementation [22]. The artifacts involved were EPLAN Electric P8 drawings as well as source code. Both were provided by a third-party company.

5.5 Threats to Validity

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

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

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

6 Related Work

Currently a multitude of consistency checking mechanisms exists (e.g., [37] [38] [74]). In practice, consistency checking mechanisms are either integrated into standalone tools (very common with UML modeling tools) or executed on documents (very commonly XML). While the consistency checking mechanisms for standalone tools tend to be agile and support live, often instant inconsistency feedback with changes, their consistency checking is limited to the kinds of artifacts available within the tool. Document-centric consistency checking approaches let engineers combine artifacts from multiple tools and hence support global consistency checking; however, at the expense of agility. Document-centric approaches are not change driven and hence expensive and disruptive to use. In all cases, consistency checking mechanisms require a single, well-defined metamodel. For example, the consistency of a UML modeling tool is checked within the context of the UML metamodel or the consistency of a Java programming environment is checked within the context of the Java metamodel. Similarly, the consistency of a merged document is checked within the context of that document's grammar (e.g., XMI in case of XML documents). Nonetheless, the limitations of all mechanisms are always the same. Every approach today 1) either focuses on the limited knowledge available within their respective

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

One document based approach is ArchJava, developed by Aldrich et al. [1]. It couples an architecture description language with implementation code. For this it utilizes an extended version of Java, featuring a unique type system that guarantees communication integrity between the code and the architecture. Archface [91] acts as both a programming-level interface as well as an architectural description language. It relies on architectural constraints within the implementation to realize traceability. Likewise, DiaSpec [15] uses an architectural description language to describe allowed interactions between components. Inconsistencies are detected with the help of the Java compiler. The problem of ArchJava, Archface and DiaSpec are the limitation to a specific programming language, respectively its compiler. Our approach relies on a generic data format into which a multitude of different engineering artifacts can be translated. Another document based approach is discussed by Nentwich et al. [66]. It generates links between distributed XML-based web artifacts and checks their consistency. In contrast our approach is not based on distributed documents but centralized engineering artifacts. This removes the network communication delay from the consistency checker's processing time. Furthermore, our approach delivers live consistency feedback to engineers.

Koenig et al. [51] presented an approach that is closer to the ideals of global consistency checking. This approach allows heterogeneous multimodels to be analyzed as locally as possible by only comparing model elements relevant to a global constraint. Doing so, significantly reduces the effort required for model matching respectively merging. TReMer+ [77] is a tool for model merging and global consistency checking. It includes merging operations for requirements, behavior models, design and implementation. Both Koenig et al. [51] and TReMer+ [77] rely on partial merging respectively the comparison of metamodels. Our approach relies on links between engineering artifacts, which act as a lightweight alternative to model merging.

Finally, Egyed et al. [32] as well as Demuth et al. [22] [23] [25] covered certain aspects of live, global consistency checking but with key differences. The solution proposed by Demuth et al. [25] alike ArchJava or DiaSpec provided a solution where Eclipse was used to represent both model and code. Egyed et al. [32] already introduced a cloud to provide a central place of artifact storage with a uniform representation. However, this paper focused on reducing the memory and computational footprint of checking the same consistency rules separately for each tool. Our approach checks the rules once centrally and then makes the results available to all relevant tools. Finally, Demuth et al. [22] reports on an industrial application of their proposed approach in the form of an experience report. The experience report focused primarily on the need for live, global consistency checking and how a cloud could support it. However, that approach required engineers to be aware of the cloud, whereas our approach is able to effectively hide the existence of the cloud, such that engineers remain mostly unaware of it.

7 Conclusion

This paper discusses a novel approach towards live global consistency checking, with the help of an engineering cloud environment. It presents the functionality and architecture of the consistency checking mechanism and demonstrates its feasibility with two industrial case studies. The approach uses the DesignSpace as a cloud environment. In future, the consistency checking mechanism will be adapted to consider private views, respectively

different versions of engineering artifacts. We will investigate a way to compute consistency checking information for differently versioned artifacts, including those that might be in conflict with those of other private views. We will also analyse how the results of rule evaluations could be provided to users in a helpful manner (e.g., through mutable pop-up warnings integrated in tool adapters).

8 Acknowledgement

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

Paper D

Multifaceted Consistency Checking of Collaborative Engineering Artifacts

Published ME '19: Proceedings of the ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C); September 2019; Pages 278-287;

Authors Michael Alexander Tröls, Atif Mashkooor, Alexander Egyed

Abstract In modern day engineering projects, different engineers collaborate on creating a vast multitude of different artifacts such as requirements, design specifications and code. While these artifacts are strongly interdependent, they are often treated in isolation and with little regard to their semantical overlappings. Automatic consistency checking approaches between these artifacts are rare and often not feasible. Therefore, artifacts become inconsistent and the consequences are costly errors.

This work proposes a multifaceted consistency checking approach for different kinds of engineering artifacts, with the help of a collaborative engineering platform. The proposed approach enables engineers to automatically check the consistency of their individual artifacts against the work results of other engineers, without using different tools than the established ones of their fields and without merging their artifacts with those of others.

1 Introduction

Engineering is a complex process involving different engineers from a multitude of varied disciplines. Each engineer produces artifacts such as code, requirements, models, hardware specifications and many more. The complexity and number of these artifacts is constantly growing as engineering projects become an increasingly important part of many different industries. A modern engineering project is no longer just concerned with the details of, e.g., a software solution or a mechatronical system – it acts as an inter-disciplinary process incorporating the knowledge from many different areas such as medicine, agriculture, logistics and more.

Likewise, the interdependency among engineering artifacts is strongly increasing as well. With the engineering process becoming more and more complex, the necessity to integrate and propagate changes from highly interdependent artifacts within a project becomes a more elaborate and time costly endeavour. For example, if hardware specifications change, all related engineering artifacts, first and foremost those using the specified hardware, must propagate such a change in one way or another. Frameworks may need to be updated, implementations might need to be adapted and designs may require restructuring in the light of new technological circumstances.

This combination of complexity, high interdependency and constant availability of the increasing amounts of engineering artifacts, has made the maintenance of the involved information a difficult task. As a result, keeping consistency among engineering artifacts is a critical aspect of each engineering project [36]. However, the available software solutions are lacking in this regard. While a regular engineering project may share its engineering artifacts among engineers in a multitude of different ways, this is not a sufficient solution to overcome the increasing complexity of the artifacts and their interdependencies. Despite its major industrial importance and constant reminders by the research community (e.g., [37, 38, 72, 74, 76, 89, 94]), maintaining consistency among artifacts is treated as an afterthought.

Therefore, this work proposes a novel, multifaceted way of checking consistency between engineering artifacts within a collaborative environment. The consistency checks are automatically triggered by changes engineers perform on their artifacts and happen under the consideration of the semantical overlappings of the different types of engineering artifacts involved in a project. This enables engineers to approach the problem of the strongly interdependent nature of engineering artifacts and simplifies the engineers' task of maintaining their work results' consistency. We tackle the artifacts' physical, virtual and syntactical separation by synchronizing them into a collaborative cloud environment, where they share a common representation. There they can be arbitrarily linked to each other, which allows us to establish semantical equivalencies between the different artifacts. These links can subsequently be used for consistency checking on the basis of a cloud service and a set of user-determined consistency rules, which are automatically checked on the synchronization of a change with the cloud.

The rest of this paper is organized as follows: In Section 3 we discuss the current state of collaborative engineering and address problems that may commonly occur during modern engineering projects. In Section 3 we introduce our approach for tackling these problems. In Section 5 we discuss details on the realization of our approach. We conclude this paper with a discussion of the approaches benefits in Section 5.

2 Problem Statement

A single engineering project may involve a multitude of different collaboration tools, many of which are used in parallel. Such tools include repositories, e.g., Git¹ or SVN², time

¹Git: <https://git-scm.com/>

²SVN: <https://subversion.apache.org/>

planning tools, communication tools or various team management solutions. Each of these tools incorporates engineering artifacts in one way or the other. Artifacts may be stored in a repository or shared through communication tools - yet, while engineering artifacts are passed around through and stored within these collaboration solutions, they are often not truly integrated within the tool. A traditional repository, for example, may store different engineering artifacts, but does not offer ways to natively incorporate the relationships between them. As a result, different engineering artifacts remain logically isolated from each other, even when they are virtually stored in the same space. This naturally makes the maintenance of consistency among engineering artifacts a highly complex task, as it has to be treated in an equally separate way for each type of artifact. No true incorporation of different engineering artifacts in a single consistency check is feasibly possible. Yet lacking such consistency checking may lead to a multitude of different problems. In the following we discuss some of the most critical problems, as well as other circumstances within current collaboration solutions that hinder feasible consistency checking.

- **Syntactic differences of engineering artifacts:** Most modern collaboration tools allow the storage of a single type of engineering artifacts. For example most repository solutions are text- respectively line-based and therefore, mostly used for code. Different engineering artifacts, such as design models, can technically be stored on these repositories as well, however, their formats - especially if they are stored in binary files - are often not natively supported. As a result, meaningful analysis of different engineering artifacts in one space is very difficult.
- **Artifact interdependency:** Despite the high interdependency between different engineering artifacts, collaboration solutions often provide no way of properly representing this circumstance efficiently. This complicates the effort of maintaining consistency among artifacts, as a consistency checker – even if it can interpret all engineering artifacts – has no possibility of making use of the semantic overlappings between different artifacts. Therefore, similar concepts, represented in different kinds of engineering artifacts, become isolated from each other and the task of keeping them consistent might fall short.
- **Limited collaboration techniques:** Due to the large variety of engineering artifacts, there is a limited support for comprehensive, native integration within most collaboration solutions. While collaboration techniques like merging or branching are common for text-based artifacts – e.g., code in Git or SVN – such operations are not possible for many other kinds of engineering artifacts. For example, design models or circuit diagrams are rarely meaningfully supported in this regard.
- **Difficult merging process:** Current collaboration solutions provide only limited conflict identification for engineering artifacts. Once the work of one engineer has to be integrated with the work of others, this may lead to a difficult merging process. This may further lead to unnecessary delays, which can become a crucial problem, especially when the integration process of engineering artifacts is tackled close to deadlines. As a result, hasty resolutions may often be incorporated into the final product, which potentially lead to severe errors.
- **Refactoring efforts:** When the merging process leads to the identification of problems, the consequence is often an extensive refactoring phase. Engineers may have to carefully refactor their work into a state where it is consistent with the work of others again. For example, both design models and code of a software engineering project may become out of sync, which makes it necessary to update one or both sides, in order to keep the project's documentation up-to-date.

- **Tolerated inconsistencies:** It often occurs that even though inconsistencies are identified during the merging process, they are being tolerated to a certain extent. This is, for example, done to simplify development processes during engineering projects or out of organizational reasons. However, at some point the identified inconsistencies must be corrected. To guarantee this, not only the identification, but the documentation of inconsistencies is a critical task. Engineers must know which issues arose in which context at what time to meaningfully merge the final engineering artifacts and subsequently remove the documented inconsistencies. Often, since cross-documentation of different types of engineering artifacts and their relationships rarely exist, the removal of older inconsistencies is entirely based on the memory of individual engineers. This, of course, holds massive potential for errors in the final product, especially if an engineering project has long development cycles or engineers are replaced during the project. Consequently, many undocumented inconsistencies remain hidden for the rest of the development. As a result, costly errors can end up in the final product, which are – if they are detected at all – hard to fix.
- **Error propagation:** Artifacts which undergo constant changes by several engineers also produce constant consequences for related artifacts. So, while a single change may not cause an inconsistency in itself, the change may lead to errors once it is propagated through into other engineering artifacts. If that is the case, it is important to know the exact change history of engineering artifacts and the associated consistency state. Lacking the possibility to fall back on such documentation diminishes the possibility of identifying the original source of a problem. Even if an inconsistency can be found, the unawareness of its further consequences may nullify the attempt to repair it and turn the repairs into sources of new errors.
- **Symptomatic inconsistency repairs:** Given that a consistency checker finds an inconsistency, repairing the inconsistency does not necessarily eradicate the problem. If the inconsistency is correlating with previous changes, the actual root of the problem can not be gotten rid of by fixing the latest effect. As such, engineering projects enter a cycle of fighting the symptoms of old, largely invisible problems.
- **Follow-up errors:** Naturally, the practice of fixing symptomatic problems can lead to follow-up errors. Especially if later project specifications start to clash with the original problem. These decisions may propagate through one or several engineering projects for a very long time, before the real problems are finally recognized. A typical example of this are long lasting errors in software solutions based on architectural mistakes. If the same base architecture is re-used during the production of new software, respectively new versions of same software, old errors may lead to new work arounds and new errors.
- **Late or no recognition of inconsistencies:** If, at last, the original problem causing an inconsistency is – perhaps by chance – identified, the damage is most likely already done. The effects are potentially immense costs, radical changes of established systems or product lines and possibly even legal consequences. This makes the early recognition of inconsistencies – and thus the recognition of correlating changes – not just a side issue, but an existential necessity for the smooth execution of engineering projects.

As can be seen, the support of different engineering artifacts, as well as the documentation of their relationships with each other, is an important aspect of engineering. Yet, the current landscape of collaboration solutions is severely lacking in these regards. Therefore, the identification of inconsistencies becomes very difficult.

3 Multifaceted Consistency Checking

The issues discussed in Section 3 show a wide array of different aspects that are affected or could be averted by a multifaceted consistency checking approach fit for the growing complexity, number and required availability of engineering artifacts. As such, we suggest a consistency checking mechanism featuring five core characteristics in this work:

- Cloud Deployment
- Global Constraints
- Live Capabilities
- Context Views
- Group Orientation

Each of these characteristics offers individual advantages for the engineer, which are discussed in the following.

3.1 Cloud Deployment

To meet the requirement of constant availability, storing engineering artifacts in a single, network-connected storage space comes as a natural practice in most engineering projects. Considering consistency checking, this means, that the respective mechanism could be deployed on such a storage, instead of being run individually for each tool, on each machine contributing to a project. The core issue with this is not the technical limitation of data storage, e.g., by synchronizing artifacts into a cloud environment. The problem is to natively store them in an integrated way that allows us to handle their complexity, especially with regards to their syntactic differences. A consistency checker must either operate on a single uniform engineering artifact representation or be able to interpret all different engineering artifact formats that are synchronized with the storage. Naturally the former is a more feasible solution.

In this work we settle for a uniform artifact representation, which can be further exploited for advantageous concepts that aid us in tackling the complexity of engineering artifacts. In fact, a core issue that encompasses many of the problems discussed in Section 3 is the syntactic difference of engineering artifacts. While a multitude of artifacts is stored in various textual representation (coding languages, plain text, HTML, XML, etc) many others are, for example, stored in a binary format (images, CAD drawings, etc). Regardless of their format these artifacts share interdependencies and semantical overlappings that need to stay consistent. A classic example would be the names of a class diagram and its concrete implementation in code. To keep these artifacts consistent requires either a consistency checker that is able to read all the involved formats and interpret them, or a transformation of the artifacts into a single uniform representation in which they can be read and analyzed on a syntactic common ground. With such a uniform representation present, the deployment of a consistency checker within a cloud environment becomes both feasible and advantageous.

3.2 Global Constraints

One way to advantageously exploit the uniform representation of artifacts synchronized with the cloud environment, is the possibility to represent their interdependencies in the same abstracted manner. While engineering artifacts feature many different semantic overlappings, the resulting interdependencies between them are only rarely comprehensively documented. Even with traceability being a common challenge for each engineering

project, most tools concerned with it, are limited to parallel representations of the matter in further separate tools. The approach presented in this paper suggests a more integrated way of representing the interdependencies of artifacts. In our approach we represent semantic overlappings, respectively the interdependency of engineering artifacts, through the manipulation of the uniform artifact representation directly in the artifact storage space. This allows us to arbitrarily extend the engineering artifacts with additional meta-information, e.g., the relationship to other artifacts. If, for example, a certain piece of code realizes a certain UML diagram, this relationship can be expressed through the addition of a simple link on the respective code artifact, pointing towards the uniform artifact representation of the UML diagram artifact. Alternatively, a new artifact can be created which then points towards both the code and the UML artifact as source and target respectively.

These links can then be used by the consistency checker to analyze the impact of an engineering artifact's change beyond the boundaries of a single discipline. This can be done by formulating consistency rules in a way that builds on the uniform artifact representation and therefore utilizes the links that can be set therein. This gives engineers the possibility to implement global constraints for entire engineering projects, which can be applied in multiple scenarios, from model-code consistency checking, to checking the integrity between implementation and corresponding circuit diagrams ([22,89]).

3.3 Live Capabilities

To prevent the late recognition of inconsistencies and to tackle the aspect of ever changing artifacts having an impact on artifacts related to them, a consistency checker capable of reacting live towards changes becomes a necessity. Exploiting the cloud deployment and the uniform artifact representation therein can help tackle this problem. As a single, cloud-based consistency checker, familiar with the uniform artifact representation can be responsible for all engineering artifacts within an engineering project, the computation of consistency states can be entirely server sided. This way consistency checks can be done in parallel to engineers working. Their work is not interrupted. At the same time, the server-sided computation can apply load-balancing strategies that help minimizing the computation time for a consistency check. As a result consistency feedback can be given through the network directly to the engineers, live as they synchronize their artifact changes with the cloud. Immediate feedback of such nature can be utilized to foster a more team-driven approach towards error handling, as well as awareness of collaborative activities within the team of engineers.

Handling the amount of computations and feedback is no trivial task in this context. The organization of the cloud's artifact storage into several substorages can be advantageous in this context. In this work we apply both a private and a public work area - the earlier holding only work of a single engineer, which is then pushed to the latter and made publicly available for every engineer using the cloud environment. Every change within one of the substorages fires an event. Feedback that is not related to a certain work area can easily be filtered out for the connected engineer. Further our data model allows us to minimize the number of (necessary) executed consistency checks. This adds to the scalability of our consistency checking approach.

3.4 Context Views

An engineer's work can always be regarded in two different consistency contexts: By itself - e.g., code corresponding to certain syntax - and integrated with the work of others - e.g., code corresponding to UML designs. To prevent merging conflicts as well as extensive refactoring processes, our approach can regard the consistency state of an engineer's work from the context of a private work area. This way, the engineer's work is projected on top of publicly available engineering artifacts. The consistency checker can then deliver

information on what would be the consistency state of both the private and the public artifacts, would the engineer decide to push his work to the public area. Since feedback can be given live, the engineer can always be aware of this information.

3.5 Group Orientation

Adding on top of context views, multifaceted consistency checking can also regard the consistency of engineering artifacts within a group context. Then the engineers' work, held in their private work area, is not primarily projected on top of publicly available artifacts, but on top of the work present within a group of other private work areas. The consistency state computed from this context allows engineers to be aware of their work's consistency integrated with the ongoing work of other engineers. This further prevents a complex merging process as conflicts can be detected before engineers decide to make their engineering artifacts publicly available.

4 Realization

The approach discussed in this work is a live consistency checker of global consistency rules, representing the interdependent aspects of engineering artifacts. It reacts towards the synchronization of artifact changes onto a collaborative engineering cloud environment, where all engineering artifacts are stored in full or partially. The synchronization is handled by tool adapters, which are used with the regular engineering tools the engineers use in their respective fields. Our approach has no need of switching tools. Engineers may work with what they already know. In this Section we outline how the approach tackles the problems discussed in Section 3, respectively the concepts discussed in Section 3 with regards to their realization.

4.1 Uniform Data Representation

A problem that has to be addressed before realizing multifaceted consistency checking of varied engineering artifacts, is the equally varied landscape of different artifact formats. A large portion of engineering tools store their outputs in unique data structures, respectively file formats. This is a major hurdle, when it comes to the meaningful integration of engineering artifacts in a single storage space and subsequently, the analysis of their semantically overlapping aspects. Therefore, this approach applies a uniform data representation, into which engineering artifacts - or parts thereof - are transformed, before they are stored in a cloud environment. This allows a cloud service to analyse the stored data, without knowing the specifics of the original artifact format.

The uniform data representation engineering artifacts are transformed into, is a typed, uniquely identifiable mapping of named values with timestamps. These named values are referred to as properties. A single property always contains a list of values, which represent the property's change history.

Every artifact in the cloud environment is created according to a corresponding type. Such types can either be created manually by users, or automatically by tool adapters (e.g. during the adapter's initialization process). Further, the type contains a set of definition fields that describe the name, cardinality and primitive data type of an artifact's properties. Once artifacts are synchronized with the cloud environment, the type is automatically instantiated and the respective properties are filled with values. The mapping process is done on the side of the synchronizing engineering tool, with the help of a tool adapter - which can be written as a plugin, using the cloud environment's API.

The data representation is willfully kept abstract to allow the biggest possible number of different engineering artifacts to be stored in the cloud environment. A figurative

Figure 1: Exemplary types and instantiations for requirement artifacts.

transformation of engineering artifacts can be seen in Figure 1, where a requirement type is present in the cloud environment and two instances have been created with corresponding values.

4.2 Linking

To represent the interdependencies between different engineering artifacts, the applied uniform data representation allows us to link different engineering artifacts together, by referencing their unique identifier within a property. This way, semantical overlappings between engineering artifacts from different fields, can be easily expressed. In the current approach, the linking process is handled manually - however, automated approaches can easily be deployed on the cloud environment in the form of a further service. The linking tool used in this approach can be seen in Figure 4, showing an overview of the artifact structure, as well as a detailed view on a selected artifact. The links themselves can be set as simple references on artifacts, or be instantiated as artifacts themselves - with source and target properties pointing towards the respective linked engineering artifacts. This allows our approach to apply different link types and enhance them with further meta-information. Such structures can again be used by a consistency checker for the analysis of relationships between artifacts. Furthermore, the timestamped history of property values gives information on which artifacts were linked together at what point in time, which automatically documents re-arrangements within the relationships of artifacts.

4.3 Artifact Storage

Once artifacts are transformed into a uniform representation, they can persistently be stored in the cloud environment. More precisely, the artifact storage stores a set of property changes, the sum of which describes the full representation of an artifact. The artifact

storage itself is separated into two types of work areas.

- **Public Work Area:** There is only a single public work area in the cloud environment. Engineers can push their changes into the public work area, once they are done editing them in their respective private work area. The public work area can be accessed by any engineer at any time. It acts as the root within a hierarchy of work areas.
- **Private Work Area:** There are several private work areas within the artifact storage. Each tool adapter synchronizing engineering artifacts with the cloud environment is bound to a single, unique work area. When synchronizing an engineering artifact, the private work area only stores a change, respectively a delta of the artifact with respect to the publicly already available information on the said artifact. This restricts the contents of the private work area to a relatively small set of changes. The private work areas can be parented to each other, with the public work area always acting as the highest parent. The parenting concept allows us to provide specific, change-oriented views on engineering artifacts. Tool adapters, retrieving engineering artifacts from the cloud, always receive the changes they synchronized with their bound private work area projected on top of whatever artifact information is available in the private work area's parents. An example for such information retrieval can be seen in Figure 3, where the public area of the artifact storage contains an instance of a "Robot Arm" engineering artifact, describing length and variant of the hardware. From the perspective of private work area "WA1", the robot arm has the length of 0.75 meters, since its contents are projected on top of the information within the public area. Work area "WA2" on the other hand still retrieves the length as 1.00 meters. If requested properties cannot be found within a private work area, they are retrieved from the parents - in this case the public work area. Vice versa "WA1" can not retrieve the latest name and variant description of the robot arm, as changes on these properties are only available within work area "WA2".

Since many engineering projects are organized in development teams, the artifact storage also allows for the grouping of work areas. These groups can be established by the users and alter the data retrieval process. When a property can not be found within a grouped private work area, the cloud environment will first check the other work areas within the group instead of the private work area's direct parents. If the property is found in multiple members of the work area group, the cloud environment retrieves the latest version of it, based on its value's timestamp. This way - when a full artifact is retrieved from the artifact storage - the changes of a private work area are first projected onto whatever relevant properties can be found within the group, which are then projected onto what can be found in the public work area. Coming back to Figure 3, would "WA1" and "WA2" be grouped together, the latest name and variant changes - present in "WA2" - could be retrieved from the perspective of work area "WA1". Vice versa, the latest length - present in "WA1" - could be retrieved from the perspective of work area "WA2". Property duplicates would have no effect, since there are only two work areas and the work area from whose perspective data is retrieved overrules the group.

4.4 Live Services

With engineering artifacts being persisted within the artifact storage, our approach can now access the said storage to analyze and potentially alter its content. This is done through live services, which are running in parallel to the cloud's activities. Every cloud activity, for example the synchronization of a change, triggers a corresponding event, which can be listened to by the services. As a result, every service can immediately react towards changes synchronized by the engineers. This holds the advantage that it allows

Figure 2: An overview of an artifact as represented in the context of a work area hierarchy.

us live consistency checking, which supports the effort of keeping inconsistent states of an engineering project as short as possible, and which prevents the late recognition of engineering artifact inconsistencies.

4.5 Consistency Checking

With the discussion of the problems listed in Section 3 the importance of comprehensive consistency checking becomes very clear. Maintaining consistency and keeping the complexity of engineering artifacts in check is a critical aspect of every large engineering project - not just for the sake of documentation, but to ensure a level of quality in the final product by preventing unnecessary flaws. Our approach builds on the cloud environment's concepts introduced beforehand. As such our consistency checking approach is implemented in the form of a live service. It listens to changes synchronized with the artifact storage, analyses these changes for their relevancy and re-evaluates the altered artifacts in parallel to other cloud activities. The results are fed back to the users through their tool adapters. This process is illustrated in Figure 2. This section goes into detail on the specifics of the consistency checker's data model within the cloud environment, as well as its concrete functionality whenever a change happens within the artifact storage.

Data Model

To realize the consistency checking service, our approach fully utilizes the uniform data representation by creating its own artifact types as well as the corresponding instantiations for them. This first and foremost comes with the advantage that the service handles its own documentation as it stores its results - which in themselves are engineering artifacts again - alongside other engineering artifacts. It references the concrete values that lead to the stored results and offers information on the involved artifacts.

For the storage of its own internally used data, the consistency checker creates two different engineering artifact types:

- **Consistency Rule Definition Artifact:** A Consistency Rule Definition Artifact defines a certain rule for a single, specific artifact type. It contains a name, a description, a reference to the specific type artifact and the concrete rule in a simple string, each of which are provided by an engineer. Every time an instance of the

Figure 3: Tool adapters synchronize artifacts with private work areas, from where they can be pushed to the public area. The artifact storage is observed by the consistency checker, which provides feedback to the tools.

type is created, the consistency checker automatically realizes the definition in the form of a Consistency Rule Evaluation Artifact.

- **Consistency Rule Evaluation Artifact:** A Consistency Rule Evaluation Artifact realizes a consistency rule definition for the instantiation of a certain artifact type. For example, if an engineer creates a Consistency Rule Definition Artifact for a requirements artifact type, as seen in Figure 1, every instantiation of the requirement type (ID:35 and ID:36) will receive its very own Consistency Rule Evaluation Artifact. This artifact contains a reference to its definition, a reference to the artifact type instantiation (the so called context element), the latest consistency checking result of this Evaluation Artifact and a so called scope, which keeps track of all properties involved in the computation of the result.

Both of these artifact types, respectively their instantiations are utilized during the consistency rule evaluation process. Within this process the Consistency Rule Definition Artifacts have an organizational role, while the Consistency Rule Evaluation Artifacts are concerned with documenting the results. The consistency rules themselves are stored as a string that is parsed and executed during the evaluation. These strings have the following structure:

$$Conjunction(Operator(ExpressionA, ExpressionB)^{1...*})$$

Expression pairs are compared via operations. Many operations can be connected into a logical conjunction, formalizing the full rule. The expressions themselves are held in an OCL-like language³, which describes the navigation path from a context element towards a certain property value, based on the uniform artifact representation. Consider again the structure of the requirements artifact in Figure 1 and the following expression:

$$self.@type.TypeName$$

This expression, for example, would retrieve the name of the context element's type (the context element is always the first point of reference within a rule and referred to as "self"). The result sets of two of such expressions would be compared via the enclosing operation. In our approach we implemented regular operations (equals, greater than, etc) for both sets and single values. A set of values may be retrieved when the cardinality of a property is defined accordingly in the artifact type's field definitions.

Rule Evaluation

Our consistency checking approach reacts immediately towards changes synchronized within the cloud environment's artifact storage. In the following we discuss every step from the fired change event to the concrete consistency state results.

- **Change Analysis:** When engineers perform a change within their tools, the corresponding tool adapter transforms the change into a form that can be incrementally added onto the uniform data representation. This change is then uploaded to the cloud environment and stored within the artifact storage on the respective artifact's property. Each change fires a change notification event, which can be listened to by the cloud's live services. The change notification event contains information on the identifier of the corresponding artifact, the name of the changed property, the nature of the change (Creation, Update, Deletion), as well as the new value of the property. This way, the consistency checker receives all required changes performed on the synchronized artifacts and can immediately react by computing a new consistency

³OCL: <https://www.omg.org/spec/OCL>

state. Naturally, not every single change is relevant, nor would it be performant to re-evaluate with every change notification event. Therefore, the consistency checker only reacts towards three types of change events:

a) *Creation of an artifact*: When an engineer creates a new artifact in the cloud environment, the consistency checker must first analyze whether the type of the created artifact is referenced in an existing Consistency Rule Definition Artifact. If that is the case, the service automatically creates the corresponding Consistency Rule Evaluation Artifact and sets the context element.

b) *Update of a scoped property*: When an engineer changes an existing artifact, the cloud environment stores this change on the respective property and fires a corresponding change notification event. To minimize the number of consistency rule re-evaluations, the consistency checker only reacts towards changes on properties that are part of a scope. This is possible, since a scope (which is built during the first re-evaluation of a Consistency Rule Evaluation Artifact) references all properties that are relevant for a consistency rule. Naturally, when one of these scoped properties changes, the Consistency Rule Evaluation Artifact containing the scope must be retrieved.

c) *Deletion of an artifact*: When an engineer deletes an artifact, the cloud fires a corresponding change notification. The consistency checker then analyzes whether the deleted artifact was a context element or referenced through a scoped property. If the earlier is the case, the corresponding Consistency Rule Evaluation Artifacts are removed as well. If the latter is the case, the corresponding Consistency Rule Evaluation Artifacts' results are set to "invalid" until a change on a related scope element (referencing the deleted artifact) triggers the re-evaluation process again.

The created, respectively retrieved Consistency Rule Evaluation Artifacts are marked down for "Data Gathering". The following steps must be executed for each Consistency Rule Evaluation Artifact individually.

- **Data Gathering**: For the data gathering process, the consistency checker first retrieves the stored consistency rule from the Consistency Rule Definition Artifact referenced in the Consistency Rule Evaluation Artifact. The said rule must then be parsed. Each step within the rule represents a navigation step through the artifact structure, starting at the context element. Each part of an expression refers to a certain property on a certain artifact. The navigated artifacts, respectively their properties are stored within the scope. Once a full expression has been navigated through, the final results (the property values of the last step) are marked down for the "Rule Evaluation" process. This is done for every expression within the rule.

It should be noted, that when work area groups exist within the artifact storage, this step is slightly different. If a property can not be found within a work area, it is first looked for within the group. If more than one result can be found (i.e., if more than one work area has changed the property) these are compared via their timestamp. The latest version of the respective property is retrieved. If the property can not be found in the group, the consistency checker falls back on the work area's parent hierarchy and eventually the Public Work Area.

- **Rule Evaluation**: For the concrete rule evaluation process, the gathered property values are substituted for the consistency rules expressions and the encapsulating operations are executed. The resulting boolean values are connected in the overall logical conjunction. The boolean result of this conjunction denotes whether the consistency rule holds or not.

- **Storing Results:** Once a rule evaluation result has been computed, the said result must be stored within the artifact storage for the documentation of consistency states. In our approach, the result is stored as a simple change on the according property of the corresponding Consistency Rule Evaluation Artifact. This change is of course only stored within the context work area that triggered the original rule re-evaluation. Once the change is pushed to the public work area, the latest result is simply added on the corresponding artifact's result property with a new timestamp. With the help of timestamps the history of consistency states is stored within this property.
- **Providing Feedback:** Once the result is added to the Consistency Rule Evaluation Artifact a new change notification event is fired within the cloud environment. This change event can be used by further live services that provide detailed feedback. Such feedback can then be provided to the engineers by contacting the tool adapter plugin bound to the work area, for which the latest re-evaluation was conducted. Alternatively tool adapters can simply listen to change notification events based on the Consistency Rule Evaluation Artifact type. This way, engineers will immediately be notified if their work area, respectively the changes therein, are in a new consistency state. This information can then of course be used for further visualizations by the tools. For example, potential inconsistencies can be marked on the according engineering artifacts (e.g. classes within a UML diagram that do not correspond to a naming convention, formulated in a rule definition, could be colored red - the concrete realization of feedback is up to the tools themselves).

5 Discussion

The application of multifaceted consistency checking on different engineering artifacts comes with several benefits. However, it also comes with the drawback that engineering artifacts must first be transformed into a syntactically equivalent form. This means additional effort for programmers - implementing tool adapters - as well as domain experts - consulting the programmers during the implementation process. We believe, that the benefits are well worth the additional effort, as the uniform data representation also allows engineers to perform arbitrary enhancements on their data, e.g., through the addition of properties (or referenced artifacts) holding meta-information. In the case of the consistency checker, such added information would be the consistency states of an engineering artifact, which add further to the documentation of an engineering project.

This flexible way of adding information helps us tackle the complexity of the growing number of engineering artifacts, as it also allows us to document the various interdependencies between artifacts in a single location. In our approach this is done via a specific cloud tool, which can be seen in Figure 4. Further the uniform artifact representation allows the sharing and merging of engineering artifacts in a fine-grained way that is not possible in most modern collaboration solutions. This simplifies the integration process of one engineers work with the work of other engineers.

The consistency checker, with its ability to check the consistency of private changes against public knowledge (as the earlier are projected on top of the latter), allows engineers to stay consistent with the work that has already been done and simplifies the eventual merging process. With the work area grouping mechanism this advantage can not only be drawn from public knowledge, but from knowledge about the private changes of other engineers as well. This way, engineers are notified about inconsistencies in their work, before a merging process even starts. This minimizes potential refactoring efforts and prevents late recognition of major conflicts in ongoing work.

Since the consistency checker, respectively its Consistency Rule Evaluation Artifacts,

keeps track of the consistency state history of a certain engineering artifact, the consistency checker provides documentation for temporarily tolerated inconsistencies within a project, which may otherwise be forgotten and go unnoticed until the final product.

With an overview over the fully connected, navigatable artifact structure of an engineering project, the consistency checker can further help tracking down the origin of various errors, as it keeps track of all engineering artifacts that are involved in the computation of a consistency state. This helps engineers to avoid symptomatic inconsistency repairs and potential follow-up errors.

All in all, consistency checking in a cloud environment provides several highly advantageous aspects. We believe, these aspects outweigh the additional effort of writing custom tool adapters, as our approach can help us tackling some of the major issues arising through the constantly increasing complexity and number of engineering artifacts in a modern engineering project. Further, a range of tool adapters has already been implemented, such as Java Eclipse⁴, IBM Rational Software Architect⁵ (for UML), Microsoft Excel (for calculations), Creo⁶ (for CAD Drawings), Eplan Electric P8⁷ (for electrical layouts), and others. Our approach has been evaluated in multiple ways. There has been an empirical study with regards to its computational feasibility [89]. Further, two case studies have been conducted to evaluate its applicability within different engineering projects ([22, 32]), as well as within an industrial environment [22].

6 Related Work

While a multitude of consistency checking approaches exist in the current landscape of engineering solutions (e.g., [37, 38, 72, 74, 76, 94]) most of them do not consider the core aspect of this work, which is the multifaceted application of the mechanism on a varied set of different engineering artifacts. Nevertheless, individual concepts of this work have been the focus of various research efforts in the past. Consistency checking itself, as well as the collaborative aspect of engineering, with the strong interdependency between various engineering artifacts in mind, have gotten attention in various works.

Some consistency checking approaches rely on the partial merging of metamodels to incorporate different model elements with each other. This way they can check consistency on the overlapping, respectively interdependent parts of engineering artifacts without performing a full integration. Koenig et al. [51] considered various engineering artifacts by incorporating individual model elements of heterogeneous multimodels. These elements were then compared on the basis of global constraints. Minimizing the set of compared elements significantly lowers the efforts required for the matching and merging process between different types of artifacts. The rest of the artifacts - which did not overlap with other artifacts - were checked as locally as possible. Likewise, Sabetzadeh et al. [77] proposed global consistency checking by the partial merging and the comparison of metamodels. For their approach they provided merging operations for requirements, behaviour models, design and implementation. Contrary to these two approaches, our approach transforms existing engineering artifacts into a different format, which can be arbitrarily linked. This is a very lightweight alternative to the sometimes extensive effort of model merging.

Collaborative approaches towards model-driven engineering have also been the focus of various works [13, 39, 41–43, 57, 96]). A mapping study conducted by Franzago et al. [40] documented available solutions for collaborative modeling, which showed that only very few approaches allow for the editing of model artifacts in an asynchronous way. A live

⁴Eclipse IDE: <https://www.eclipse.org/>

⁵IBM RSA: <https://www.ibm.com/developerworks/downloads/r/architect/index.html>

⁶Creo: <https://www.ptc.com/de/products/cad/creo>

⁷EPlan: <https://www.eplanusa.com/us/2/>

Figure 4: An overview of the linking tool used in this approach. Both a part of the full artifact structure as well as a detailed view of a link artifact are depicted.

capability for various applications, such as a consistency checker, is partly hindered by this fact. This issue is overcome by our application of a collaborative engineering cloud which directly integrates the consistency checker and forwards activity events towards it.

Another major way in which our approach deviates from related work is the fact that its live change-adaptive capabilities are based on live synchronization through a tool plugin. This - depending on the concrete plugin - means that our consistency checker is reacting live towards changes directly handled in the engineering tools, which are immediately propagated to the cloud environment. This way, the cloud always receives engineering artifacts directly from the internal data structure that is present within the tool itself, which is in contrast to most other approaches. In most consistency checking approaches the engineering artifacts are synchronized from a file- respectively document-base. One such approach was developed by Aldrich et al. [1]. In their approach architecture description language was coupled with the concrete implementation in a Java project. It utilizes a type system guaranteeing the integrity between architecture and code. Similarly Ubayashi et al. [91] secures the integrity between architectural design and code, by providing both a programming interface, as well as a description language for architectures. In their approach, architectural constraints within the implementation guarantee the traceability and integrity between model and code. An approach that also considers the aspect of artifact availability was suggested by Nentwich et al. [66]. In it, XML-based engineering artifacts are checked for consistency after links have been generated between them. The artifacts are web distributed, which is in contrast to our own approach which centralizes artifacts in a single storage place.

7 Conclusion & Future Work

In this paper we discussed the various aspects of modern day collaborative engineering and the major problems that arise through the constantly growing complexity in engineering artifacts as well as with their number and required availability. To tackle these problems, this work introduced the novel way of multifaceted consistency checking within an engineering cloud environment. In contrast to most related work, our approach takes the diverse nature of strongly interdependent engineering artifacts into account, by synchronizing them into a single space and offering a way to represent their relationships. These relationships can then be exploited for consistency checking. Our approach further takes full advantage of the cloud environment's structural concepts. Thanks to this, the results of a consistency state computation are directly integrated with the rest of the engineering artifacts, which further adds to the documentation of an engineering project. The consistency checking mechanism itself strongly builds on the availability of various work areas within the cloud environment. It takes advantage of various context views on engineering artifacts. These views can be constructed by combining different work areas during the data gathering process of the consistency checker, e.g., in the form of a parent hierarchy, where the changes of one work area are projected on top of other changes – or in the form of work area groups, where the latest version of a certain property within a group is retrieved for the consistency check. We concluded this paper with a discussion of various benefits of our approach, as well as with a discussion of the related work.

For future work we would like to expand on the mechanism for distributing consistency feedback. Currently a lot of consistency feedback is produced, not all of which may be interesting to the engineers. Therefore, we would like to introduce a way of propagating only information that is relevant to the engineers' current focus.

8 Acknowledgements

This work is partially supported by the Austrian Science Fund (FWF): FWF P 31989-N31, the LIT Secure and Correct Systems Lab, and the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy and the Province of Upper Austria in the frame of the COMET center SCCH.

Paper E

Timestamp-based Consistency Checking of Collaboratively Developed Engineering Artifacts

At the time of writing the thesis at hand, this paper was still under submission.

Authors Michael Alexander Tröls, Atif Mashkoor, Andreas Demuth, Alexander Egyed

Abstract Engineering projects typically involve many engineers who work concurrently on a multitude of engineering artifacts such as requirements, models, code, or even non-software artifacts e.g., drawings or computations. These projects require tight collaboration within and across engineering disciplines in order to guarantee a consistent engineering process. Existing collaborative engineering tools provide limited functionality in this regard. Inconsistencies among engineering artifacts are currently identified only after all involved engineers merge their artifacts. This is problematic because considerable amount of time may elapse between these merges and engineers inadvertently may spend considerable time working under wrong/changed assumptions. This leads to more inconsistencies. This paper proposes an approach for checking the consistency of the engineers' artifacts against the continuously evolving, timestamped version history of a public artifact storage. This avoids costly reworks as engineers understand the implications of their work long before they merge their engineering artifacts. To evaluate our work we conducted an empirical study utilizing the proposed approach.

1 Introduction

Engineering is a highly collaborative discipline, requiring tight interactions among many engineers, possibly from different domains. Unfortunately, the landscape of available software tools often does not support such collaboration. Collaborative version control systems, such as Git¹ or Subversion² were originally only developed for textual engineering artifacts (e.g., code). As such, they are not fit to meet the basic requirements imposed by collaborative engineering with different kinds of artifacts (e.g., both code and UML models) and their synchronous editing. This makes it difficult for engineers to keep their works consistent.

To counter this issue, Demuth et al. [25] proposed the integration of engineering tools into a collaborative engineering environment (CEE). In such an environment, different kinds of engineering artifacts can be represented in a fine-grained, uniform artifact storage. This kind of storage can be analysed by a consistency checking mechanism. Hence, it is possible to cross-check artifacts from different engineering tools/engineers against each other according to freely-definable consistency rules. The CEE allows the analysis of heterogeneous engineering artifacts, independent from their origin. For example, it allows to check the consistency between UML models and the code that implements them. This is known as “**global consistency checking**” [89].

The advantages of utilizing a CEE have been demonstrated [23, 88, 90]. However, the downside of the current state of the art is that engineers do not understand their individual works in relation to the CEE - the “**public**” knowledge. Current works detect inconsistencies with regard to public knowledge only after engineers publish changed artifacts. This late consistency checking is problematic because engineers then may need to immediately rework published artifacts to address inconsistencies they have caused; and/or other engineers might get distracted by those inconsistencies. Indeed, inconsistencies do not arise at the time artifacts are published but at the time they are created or modified by engineers in their respective tools. We speak of artifact changes made in “**private**” - aka unpublished in their tool. A “**change**” is defined as the creation or deletion of artifacts (e.g., requirement, model element, Java class) or the modification of their properties (e.g., the class inheritance, the model element name). Such changes are initially only visible to the engineers who made them (usually in context of the tools they are using) but these changes have consistency implications as soon as they are made. Similarly, changes published to the “**public**” knowledge have consistency implications to other engineers. While a given engineer’s publication may well be consistent with the public knowledge, it may be inconsistent with private changes made by other engineers in their tools. Hence, collaborative engineering must understand the consistency of each engineer’s private knowledge with regard to the public knowledge. There are thus $n+1$ consistency perspectives for n collaborating engineers.

This paper proposes an approach for checking consistency between private changes and *any state* of the public knowledge. Our approach understands changes as timestamped events that impact consistency according to the moment of the change’s concrete occurrence within the CEE. Since the public knowledge can change frequently (as other engineers publish changes), our consistency checking approach keeps of each private perspective relative to a timestamp (aka version) of the public knowledge.

The rest of this paper is structured as follows. In Section 2, we present an illustrative example showing the advantages of consistency checking in a CEE. We illustrate our contribution to such an environment. In Section 3, we formulate the corresponding research questions. In Section 5, we discuss the technical pre-requisites. We then outline our proposed approach and the corresponding consistency checking mechanism in Section

¹Git: <https://git-scm.com>

²Apache Subversion: <https://subversion.apache.org/>

Figure 1: Working with an automated consistency checking mechanism provided by a CEE

5. In Section 6, we validate our approach on the basis of an empirical study. Section 7 covers related works within the field of collaborative engineering. The paper is concluded in Section 9.

2 Illustrative Example

Imagine the development of an industrial robotic arm that is capable of grabbing and relocating objects in front of it. A wide variety of engineers is required for such a task, but for simplicity let us focus on two engineers in particular. There is Alice (a software engineer) who describes the robotic arm's behavior of grabbing objects using a UML-style activity diagram and there is Bob (a mechanical engineer) who models the necessary hardware components and specifications (e.g., data sheets for the mounting platform of the robot arm and corresponding CAD drawings).

The left portion of Figure 1 depicts the current state of Alice's activity diagram and Bob's CAD drawing at time t_1 (before a series of changes is to be performed). Both diagrams have been committed beforehand into a shared version control repository. They are thus publicly visible within the involved group of engineers. In principle, both Alice and Bob can see both drawing and diagram albeit they cannot necessarily manipulate them (i.e., it requires extensive tool knowledge to manipulate CAD drawings that Alice does not possess). While these artifacts appear to be independent – having different underlying tools, syntax & semantics, and even reflect different engineering domains – there are nonetheless dependencies among them. For example, Alice's activity diagram refers to a “range”, describing the maximum distance at which the robotic arm can grab objects. Naturally, this range is dependent on the actual, physical length of the robotic arm. A range reflected in the CAD drawing as depicted in the lower part of Figure 1.

In steps 1 and 2 these artifacts are checked-out by Alice and Bob into their respective engineering tools (as indicated by the green check marks in the first column, the artifacts are still consistent at this point).

Assume now, that a stakeholder requests a different mounting platform is to be used for the robotic arm. As illustrated in step 3 of Figure 1, Bob implements this change within his CAD drawing tool first. The new specification reduces the robot arm's range by 0.25m. From the perspective of a shared version control repository, this change is considered "private" because it is performed by Bob within his CAD tool and (for the time being) remains uncommitted. Neither Alice nor Alice's tool know about this change (unless communicated explicitly by Bob). Hence, from this point on, Bob's change is inconsistent with the public knowledge reflected in the repository; and it is inconsistent with Alice's diagram.

Bob's change causes an intentional inconsistency (i.e., he cannot possibly perform this change without it causing an inconsistency). This demonstrates the first advantage of consistency checking in a CEE: ensuring that engineers are aware of inconsistencies they cause even if their changes are still private (uncommitted with the public repository). Hence, even though Bob has not yet published his changes and even though Bob's tool is not capable of understanding knowledge other than CAD drawings, Bob should be informed about the inconsistency with the activity diagram. Doing so informs Bob of the impact of his change (steps 5 and 6), as indicated by the red check mark in his tool's GUI. Alice does not need to know about the inconsistency yet because Bob might make further changes in private (perhaps even resolving inconsistencies if possible). However, as this is an intentional inconsistency and Bob knows that additional changes need to be performed by others, he commits his changes (i.e., publishes them as in step 7). By committing his CAD drawing with the inconsistency, he communicates that he can not solve this problem.

The public version (which changed because of Bob's commit) is now inconsistent with the diagram Alice checked out previously. This leads us to the second advantage of consistency checking in a CEE: ensuring that engineers are aware of inconsistencies caused by others once they have been committed. Alice is now informed about the inconsistency between her activity diagram and the current version of the CAD drawing in the repository (as of time t_3 , steps 7 to 10). Alice benefits from such inconsistency feedback in two fundamental ways: 1) she becomes aware of a problem early on rather than being surprised by it during later commits, and 2) she could see this published inconsistency as a trigger to continue a change that was started by others. Since the inconsistency identifies all artifacts involved (meaning hers and Bob's), she could even consult with Bob if needed.

In our illustration, Alice changes her diagram next (step 11). Once she made her changes and convinced herself that her changes are consistent with Bob's changes (analogous to t_2), she commits her changes to the public repository (step 12). At this point, Alice can be certain that her inconsistencies are indeed resolved. At this point, Bob would also become aware that the inconsistency he caused was resolved (steps 13 to 15). Note that between Bob committing the inconsistent CAD drawing and Alice resolving the problem, other changes may be merged with the public repository. Naturally, with every passing change, the source of the inconsistency will become harder to detect. The original problem may become buried within the version history of artifact changes. This illustrates the importance of not only being able to continuously check consistency against the latest state of engineering artifacts, but against several, historical versions. As such, engineers should be able to locate the exact version in which an inconsistency first arose and the changes that caused it (t_2 in Figure 1).

3 Problem Statement

As demonstrated by the example in Section 2, merely considering the consistency state of the published artifacts (the public knowledge) does not help engineers in understanding the implications of their changes. Consequently, this may lead to expensive refactoring, symptomatic inconsistency repairs, and flawed end products. While platforms for the versioned integration of heterogeneous engineering artifacts exist (e.g., in the form of CEEs), how to perform consistency checks with regard to the many private workspaces and their public counterpart is an unsolved problem. Since the different private workspaces may have been checked out at different times, we must think of the public workspace as a timestamped artifact storage whereby each private workspace may relate to a different version (i.e., timestamp) of the public workspace. This leaves us with the following research questions:

- RQ1: Which public/private workspace configurations may exist under consideration of the various timestamp differences between private workspaces and the public workspace?
- RQ2: How can we perform individualized consistency checking for every private workspace regardless of its timestamp-based difference to the public workspace?
- RQ3: What is the practical benefit of providing individualized consistency checking for every private workspace and hence for every engineer?

4 Technical Prerequisites

We implement our approach on the foundation of a cloud-based CEE. This environment provides us with several advantageous concepts, some of which are directly relevant to the proposed solution of the issues outlined in Section 3. These concepts are now discussed in more detail:

4.1 Uniform Data Representation

In the CEE, data is stored in a uniform representation. Data entities are referred to as “artifacts” – these artifacts are uniquely identifiable mappings between property names and a respective list of past (timestamped) values of the said property (i.e., changes). An artifact adheres to a certain user-defined structure, which we refer to as “artifact type” (e.g., for requirement artifacts, code artifacts, and model artifacts). Realizations of these types, referred to as “artifact instances”, come with a set of pre-defined, empty mappings, which are filled during the translation of tool data into artifacts. Translation of data is handled by custom implemented tool adapters. These tool adapters act as synchronization software between the engineers’ tools and the CEE. Any type of data that can be read and translated towards such a format can be fed into the CEE. An exemplary synchronization of three different artifact types (Java code, UML models and requirements) is shown in the supplementary material³. In our illustrative example, both Alice and Bob require an adapter to transform their engineering artifacts into the CEE’s data representation. The tool adapter would, for example, translate an activity diagram into several artifacts representing both the activities and relationships among them as well as the conditions that describe them.

³Artifact Synchronization: <https://tinyurl.com/yxfwpfoy>

Figure 2: An overview of the CEE with a tool synchronizing engineering artifacts to its respective private work area

4.2 Private Work Areas and Public Area

One further concept provided within the CEE are private work areas (PWAs) and the public area (PA). The PA incrementally stores all artifact changes published by engineers. The sum of these changes represents the full engineering artifacts, stored in a generally accessible public repository. When new changes are synchronized with the CEE, they are first stored in a PWA. Therefore, when an artifact is retrieved from a tool perspective, the changes stored within the PWA are “layered” upon the full, public artifact from the PA. The PWA changes *override* the current PA state. This process of layering is illustrated in Figure 2. Once all changes are done, the content of a PWA is pushed to the PA. The merging process then adds the published changes to their respective artifacts/properties within the PA (note again that a “Property” carries a “List of Changes”, representing its values – the last entry being the latest value). Every PWA is assigned with a timestamp corresponding to the time of its creation. The PA is assigned with a timestamp corresponding to the latest published change. Taking a look back at our example, the changes published by Bob would first reside in his PWA, before they are pushed into the PA, where they are visible to Alice.

4.3 Consistency Checking in a Cloud-based CEE

To simplify the exchange and maintenance of consistency information in the artifact storage, the consistency checker uses the CEE’s internal data structure. All information required for consistency checks is stored in two kinds of artifacts (see Figure 3 for illustration):

- **Consistency rule definition (CRD):** Stores a consistency rule and the type of artifact for which that rule should hold (context).
- **Consistency rule evaluation (CRE):** Realizes a CRD for a concrete artifact instance (context element), for which the consistency checker evaluates the rule. It also stores the result of the evaluation and a scope. A scope reflects all artifacts that are traversed during the evaluation process of a CRE.

To evaluate a rule, the consistency checker retrieves the CRD and all its CREs. For each CRE (respectively, the artifact instance referenced therein), it then navigates towards the property values described in the rule. The retrieved property values are then compared to whatever operator is used in the consistency rule. A consistency rule is formulated as an OCL string. The expression is written in an object-access like manner that reflects the artifact/property structure of the CEE. An exemplary operation could be written in the following way:

Figure 3: Illustration of consistency checking artifacts and how they reference regular artifacts

$$self.name \neq \{\}$$

Denoting that the property *name* of a specific artifact instance must not be empty (the original artifact instance of a rule evaluation is referred to as *self* in consistency rules). The rules are specified by engineers in a custom built client-tool and loaded into the CEE through a standard tool adapter. In the CEE, the corresponding CRDs are established. This way, a PWA can, for example, specify rules that are only relevant in its own, isolated perspective. Committing newly created CRDs to the PA then makes the rules relevant to all other PWAs. The consistency checker is activated through change events, fired by changes on relevant artifacts, i.e., artifact instances that are referenced by a CRE or which are part of a scope. Regarding the implementation of the concrete rule evaluation process, our approach builds upon an adaptation of an established incremental consistency checker [72]. Our mechanism provides this adaptation with the necessary data for re-evaluations. An exemplary demonstration of consistency checking in the CEE can be seen in the supplementary material⁴.

5 Timestamp-based Consistency Checking

In our approach, we provide different configurations to filter consistency checking by timestamps. The users can freely select a timestamp and inspect the correspondingly computed consistency state of their engineering artifacts. This is done to support collaborative work and narrow down past changes that may be responsible for an inconsistency. It also gives engineers the possibility to review their own work in relation to the work of others.

Timestamp-based consistency checking makes use of two concepts available in the CEE. The first concept we use are the layered artifact storage areas, i.e., PWAs and the PA. We utilize the PWA's creation timestamp to compute consistency information at the time when an engineer originally started manipulating the artifact basis (i.e., global consistency checking between the PWA and the PA but without any further changes that may have happened since the PWA was created). We further utilize the PA's latest timestamp to provide consistency information on the current state of the artifact storage (i.e., global consistency checking between the PWA and the PA at the time of the latest change).

⁴Consistency Checking in the CEE: <https://tinyurl.com/y4pzx23s>

The second concept we use is the timestamps of property values to give insight into the incremental change of consistency information.

As a result, Alice could, for example, check her artifacts only with regards to her own changes, respectively her own changes in relation to publicly available engineering artifacts. Furthermore, she could now configure her consistency information according to a certain time-frame, by setting a timestamp. This way artifacts can be compared in relation to each other, at any state of their respective change history.

All select-able configurations concerning timestamps have their own implications on consistency information. In the following, we present a discussion of different configurations and how consistency information is computed for them.

5.1 Timestamp Configurations

Each of the timestamp configurations defined in the following puts the public and private workspaces into a specific relationship, each representing a different perspective on engineering artifacts during the consistency checking process (addressing RQ1). Each configuration has its own implications on the consistency information that can be computed. Users of the CEE can freely switch between different configurations. From discrepancies between the computed consistency information, engineers may derive shortcomings in their own or other engineer's artifacts. We define the following three timestamp configurations for consistency checking in a CEE (also shown in Figure 4):

- PWA Timestamp
- PA Timestamp
- Intermediate Timestamp

Consistency is always checked from the perspective of a PWA. If there is an overlap between properties, as illustrated with the *Diagram Artifact* in Figure 4 (the property *range* is available in both PWA and PA), the PA version will always be overruled by the PWA version. The consistency checker will automatically choose the latter for the computation of consistency information.

PWA Timestamp

The PWA timestamp configuration only considers changes issued within a PWA, respectively publicly available artifacts at the time of the PWA's creation. Consider again the illustrative example as depicted in Figure 4. A consistency rule may require that the properties *dist* and *range* are equal. With a PWA timestamp configuration the consistency checker would consider all public artifacts until the timestamp of the PWA's creation (t_1). This means, it would retrieve 1.00 for the property *range* (as the PWA value overrules the public value) and 1.25 for *dist*. As a result, the rule would not hold within the PWA timestamp configuration.

Formally, this configuration can be described as follows:

$$\begin{aligned}
 C(A) &= A(P_m \cup P_n) \\
 P_m &= \{\forall p \mid p \in PWA \wedge p \in A\} \\
 P_n &= \{\forall p \mid p \in PA \wedge p \in A \wedge p \notin PWA \wedge f(p)\} \\
 f(p) &= p.t \leq PWA.t
 \end{aligned}$$

Where the configuration C on an artifact A is equivalent to an artifact made up of the union of two sets: P_m and P_n . The set P_m represents the property values p of A that reside within the PWA, whereas P_n consists of all property values p that are not in the PWA but in the PA, with their timestamp t being earlier than or equal to that of the PWA. This is expressed in the timestamp function $f(p)$.

Figure 4: An illustration of different timestamp configurations consisting of artifacts and their properties

PA Timestamp

The PA timestamp configuration considers changes within a PWA in addition to the latest changes in the PA. Consider again the illustrative example as depicted in Figure 4. Further, consider again a consistency rule that requires the properties *dist* and *range* to be equal. Given the PA timestamp configuration our approach would retrieve 1.00 for the property *range*. Again, the PWA overrules any changes on the same property within the PA. For the property *dist*, the consistency checker would retrieve 0.75 as this is the latest public change on the respective property value. The consistency rule would again not hold.

A consistency check with a PA timestamp configuration is especially useful *before* publishing changes in the PA. We can see inconsistencies and start figuring out solutions before making our changes visible to other engineers. In other words, we can check consistency with the latest project state, before starting the merging process of our engineering artifacts.

Intermediate Timestamp

The latest changes of the PA may not always be of interest to engineers, e.g., during error localization. For this, our approach offers a specialized form of the PA timestamp configuration. The intermediate timestamp configuration provides insight into private changes complemented by public changes up to a certain point in time. The engineers are given the possibility to limit consistency checks by a specified timestamp. This timestamp must be between the PWA timestamp and the PA timestamp.

Consider yet again the illustrative example in Figure 4, as well as the consistency rule requiring equality between *dist* and *range*. As with the other configurations the consistency

checker would retrieve 1.00 for *range*. For the property *dist*, it would retrieve the latest public property value via a certain defined timestamp. If this timestamp were t_2 the consistency checker would retrieve the property value 1.00, utilizing the property's change history. As a result, the consistency rule would hold. Both the PA and the intermediate timestamp configuration can be defined through an alteration of the timestamp function $f(p)$ on the property set P_n (the other definitions stay the same as in the PWA timestamp configuration):

$$f(p) = \begin{cases} p.t \leq T & \text{if } T > -1 \\ \max(p.t) & \text{otherwise} \end{cases}$$

where T is a user-defined timestamp. If T is set (i.e., it is greater than -1) the property value is only taken over, if its timestamp is equal or earlier than the specified T (intermediate timestamp). Otherwise the property value's timestamp must equal the maximum possible timestamp of this property to be taken over into the property set P_n . With regards to Figure 4 this means that all the latest property values of the publicly available artifacts are taken into consideration (PA timestamp).

5.2 Consistency Checking Mechanism

This subsection defines what alterations have to be made on the consistency checking mechanism in order to provide individualized consistency information for the defined timestamp configurations (addressing RQ2). After receiving a change event, the consistency checker follows six steps: 1) Analyze whether the change was relevant, 2) Retrieve the corresponding consistency rules, respectively their artifacts, 3) Gather data for the rule evaluations, 4) Evaluate rule expressions, 5) Write results back to the CRE, and 6) Provide consistency feedback to tool adapters. Each of these steps is discussed in the following.

Change Analysis

A consistency rule has to be re-evaluated once the state of the artifact storage changes. However, many of these changes are not relevant to the consistency state held by the defined consistency rules within a project. Therefore, consistency checking in the CEE filters a substantial part of these changes and only initiates a re-evaluation when it is really necessary, i.e., once a concrete CRE is affected, e.g., through the change of a referenced artifact instance, respectively a scope element of the CRE. The algorithm for this step is illustrated in Listing F.1 In the timestamp-based consistency checking approach, we also apply the same principle.

```
foreach (change in changeInformation)
  switch(change.ChangeType)
    case InstanceCreation
      foreach (context-corresponding CRD in PA/PWA)
        if(corresponding CRE is NOT element of PA/PWA)
          create CRE for context element in PWA
    case InstanceUpdate
      foreach(context-element-corresponding CRE in PWA)
        re-evaluate CRE for context element in PWA
    case InstanceDeletion
      foreach (context-element-corresponding CRE in PA/PWA)
        if(instance is NOT element of PWA)
          remove CRE for CE in PWA
    case RuleCreation
      foreach (CE in PA/PWA)
        if(corresponding CRE is NOT element of PA/PWA)
```

```

    create CRE for CE in PWA
case RuleUpdate
  foreach(corresponding CRE in PWA)
    re-evaluate in PWA
case RuleDeletion
  foreach(corresponding CRE in PA/PWA)
    if(Rule is NOT element of PWA)
      remove CRE in PWA

```

Listing E.1: Algorithm for change analysis

Consistency Rule Retrieval

Once a CRE corresponds to a certain change, the respective rule has to be retrieved from the CRD. This step is unaffected in timestamp-based consistency checking.

Data Gathering

Providing different timestamp configurations requires the consideration of the data gathering process for the rule evaluation. During the evaluation of a rule, the consistency checker retrieves all properties referenced by the rule. However, during the timestamp-based consistency checking this retrieval is limited to properties of artifacts associated with the currently selected configuration. The current configuration is selected by the engineers and stored in their respective PWA in the form of a timestamp T . When the consistency checker receives a relevant change event from a PWA, it first retrieves the according configuration:

$$C(PWA) = \begin{cases} PWA \text{ Timestamp} & \text{if } T = PWA.t \\ PA \text{ Timestamp} & \text{if } T = PA.t \\ Intermediate \text{ Timestamp} & \text{if } T > PWA.t \wedge T < PA.t \end{cases}$$

where $C(PWA)$ is the configuration of a PWA according to a set timestamp T . This timestamp is then used by the consistency checker for the further retrieval of properties.

$$\begin{aligned}
P_{CC}(CRE, T) &= P_m \cup P_n \\
P_m &= \{\forall p \mid g(p) \wedge p \notin PWA_{0...n} \wedge p \in PA\} \\
P_n &= \{\forall p \mid g(p) \wedge p \in PWA_i\} \\
g(p) &= \{\forall p \mid p \in CRE.scope \wedge p.t \leq T\}
\end{aligned}$$

meaning, that the set of properties P_{CC} , necessary for the re-evaluation of a consistency rule at the timestamp T and referenced through a CRE, consists of a union of property sets P_m and P_n . In both sets all properties p must be included in the respective scope of “CRE” and correspond to the retrieved timestamp T , as expressed in the function $g(p)$. The properties of the unified set may reside in the PA and not in any “ $PWA_{0...n}$ ”, meaning the public property values at timestamp T (P_m). Furthermore, properties may reside in a specific “ PWA_i ”, meaning all relevant private properties (P_n).

Going back to the illustration in Figure 4, assume a scope made up of the elements *Artifact:87:dist* as well as *Artifact:75:range* and a timestamp T corresponding to t_2 . For a re-evaluation of such a consistency rule the consistency checker would now retrieve *range* at the value of 1.00 and *dist* at 1.00.

Rule Evaluation

Once all engineering artifacts, respectively their configuration-related properties have been gathered, the consistency checker solves the rule expression under substitution of the gathered values. This procedure is unaffected by the timestamp-based consistency checking.

Consistency Information Storage

For timestamp-based consistency checking, the concept of consistency information storage must be extended. Originally, there was only one result to consider: The consistency checking result within the current work area of the consistency checker. Now that several consistency checking results can be computed for a single work area, storing the result in a single property is no longer sufficient. For this reason, we transform the result property of a CRE into a reference towards a “*Consistency Rule Result*” artifact, a third consistency checking artifact type complementing the ones described in Section 4.3, i.e., CRD and CRE. This artifact type holds the following fields:

- PA Timestamp Result: The consistency results under full consideration of the latest changes.
- Last Initiator: The last initiator (work area) of the rule evaluation.
- PWA Timestamp Results: A reference towards a mapping between PWAs and the last respective result under consideration of their timestamp PWA.t.

Note here, that there is only one PA timestamp result, which is equivalent to a consistency check within the PA, after publishing changes in it. PWA timestamp results are stored as well for later review. Intermediate results are only relevant for intermediate review and are therefore not persisted. The result artifact is stored within the PWA for which the consistency checker evaluated a rule. If this artifact is now published alongside other changes of the PWA, the results will be added to the PA and represent the current consistency state of public artifacts. Going back to the illustrative scenario in Figure 4, if the PWA changed the latest value on property *range*, a corresponding CRE and its result artifact have to be changed as well. The result is first stored within the PWA as a delta towards the last (public) result and then published.

Providing Consistency Feedback

There can be several reasons for selectively considering consistency in different timestamp configurations. Engineers may not want to be constantly notified about the changes on engineering artifacts related to their own artifacts, as it is the case with the PA timestamp configuration. Therefore, our consistency checking mechanism provides two modes for feedback: Retrieving consistency information immediately when a change is recognized (instant feedback) or when it is demanded (triggered feedback). This way, the work of engineers is not unnecessarily interrupted. Consistency information is still computed and stored in the background. Published re-evaluation results still correspond to the consistency state of the artifacts published together with them. Any visualization of consistency feedback (e.g., marking inconsistent models in a UML tool), is up to the respective tool adapter. The consistency checking mechanism merely computes consistency information and distributes the results to the respective PWAs.

6 Evaluation

In previous works on the CEE (e.g., [23,88,90]), consistency checking was only handled on the latest version of the artifact storage. Our approach, on the other hand, can distinguish between different timestamps. To illustrate this functionality and its practical benefit, we conducted an empirical study.

In this study we document, how many consistency checking re-evaluations, on different timestamps, may lead to a different result. For this, we compare the number of re-evaluations in both the PWA and the PA timestamp configuration. Every time a

consistency rule cannot be fully re-evaluated in the PWA timestamp configuration our implementation falls back to the PA timestamp configuration, hence producing a potential discrepancy between the results.

We applied the timestamp configurations during the building process of ten differently sized UML projects, respectively, their UML models. Each change during the building process was performed incrementally and represents a certain model state, from which a corresponding consistency state can be computed. Consistency checking was based on ten different consistency rules, covering several artifact types.

The presented approach has been evaluated via a prototype implementation⁵, providing the technical pre-requisites to realize the concepts proposed in this work. The implementation sets up the necessary data structures and operates on the basis of ten UML files, storing the ten selected projects.

6.1 Model Data

Our empirical study is based on ten independent real-world UML models, including class diagrams, sequence diagrams and state-charts. The type of model data is only of relevance during the transformation process of the tool adapter. Any type of model that is organized in a map-able structure can be taken over into the uniform artifact representation and, therefore, be checked for consistency. Size-wise the models range from small to large, with the number of model elements (UML classes, transitions, lifelines, etc.) ranging between 88 and 11,629. Each model is incrementally loaded into the CEE. For this, the smallest model requires 585 commands, while the largest requires 83,599. Commands were extracted by parsing the UML files. Each command represents either the creation of an artifact – based on a parsed model element – or the update of one of its properties – based on a model element’s parsed attributes (e.g., its type and name). Properties may store one or many values (i.e., lists), as well as references to other artifacts. The numerical overview of the conducted empirical study and its results are shown in Table 1.

6.2 Procedure

Before the model data is synchronized with the CEE, the consistency checking mechanism is provided with ten different consistency rules (see Table 2). While the rules themselves describe simple well-formedness conditions for UML, note that in our approach these conditions can be secured across many different instantiations of the same or different UML modeling tools. Consistency checking is handled on an abstract level. A tool adapter forwards the incremental model adaptations as commands to the CEE. With each issued command a PWA within the CEE is changed. Every change leads to a notification towards a set of listeners, i.e., the consistency checking mechanism. The mechanism then instantiates CREs according to the description in Section 4.3. This leads to a set of re-evaluations, depending on the artifact type of the created, or updated, artifact. Each re-evaluation can either be fully executed within the PWA timestamp configuration (e.g., on the re-evaluation of a single isolated model element at the time of the PWA creation) or requires the access to the PA timestamp configuration (e.g., on the re-evaluation of multiple interdependent model elements – some of which may not be available at the time of the PWA creation), essentially setting private engineering artifact changes into relation with publicly available, merged changes. The number of re-evaluations, as well as the PWA and PA timestamp configuration accesses per project, can be seen in Table 1. This evaluation deliberately does not state any information on whether consistency rules hold or break. A different timestamp does not automatically mean “better” consistency rule evaluations (i.e., rules hold), but different results. Engineers draw conclusions from

⁵Prototype Implementation: <https://tinyurl.com/y27gxanm>

Table 1: The numerical overview of the conducted empirical study and its results.

<i>Project</i>	<i>ModelElements</i>	<i>Commands</i>	<i>Creations</i>	<i>Updates</i>	<i>Evaluations</i>	<i>PWAAccesses</i>	<i>PAAccesses</i>	<i>PWAAccesses(%)</i>	<i>PAAccesses(%)</i>	<i>Time(ms)</i>
PR01	88	585	88	497	281	313	109	74.17	25.83	2.80
PR02	211	1,543	211	1,332	343	917	95	90.61	9.39	3.48
PR03	764	5,488	764	4,724	1,741	4,298	733	85.43	14.57	8.69
PR04	1,296	9,441	1,296	8,145	2,716	5,118	1,874	73.20	26.80	9.04
PR05	1,745	12,971	1,745	11,226	2,441	4,754	1,303	78.49	21.51	5.79
PR06	2,294	17,086	2,294	14,792	8,457	12,225	2,648	82.20	17.80	17.63
PR07	2,625	19,660	2,625	17,035	7,304	20,964	3,531	85.58	14.42	13.73
PR08	3,109	22,913	3,109	19,804	6,856	22,367	2,888	88.56	11.44	11.96
PR09	9,875	73,494	9,875	63,619	37,522	28,653	11,939	70.59	29.41	16.01
PR10	11,629	83,599	11,629	71,970	43,748	32,384	16,618	66.09	33.91	15.56

discrepancies. The consistency results on their own are dependent on the respective models. The practical benefit of our approach is illustrated in Table 1 by the number of PA timestamp configuration accesses, which highlight the potential for discrepancies between the PWA and PA state.

6.3 Discussion

On average 20.50 percent of re-evaluations had to be issued on the PA timestamp configuration to come to full conclusion. This means, that in one out of five re-evaluations, there is the potential for discrepancies between the global consistency results of different project states. We believe this lends credibility to the assumption, that individualized consistency checking on the basis of different timestamps proposes a beneficial concept (addressing RQ3). As shown in Table 1, the minimum number of accesses is 95 in project “PR02”, whereas the maximum accesses are 16,618 in project “PR10”. This illustrates the number of potential changes that may interfere with model elements within the PWA configuration. More precisely this means, every time a PA timestamp access happens, the consistency checking mechanism must check data, that is required for the evaluation of a consistency rule but not available within the PWA timestamp configuration. This further shows us that a PWA alone, or equivalently a versioned private working copy of artifacts, may be insufficient for the computation of consistency information. This is especially noteworthy with regards to the strong interdependence of different engineering artifacts. However, with our approach, even within a PWA, the latest public changes can be considered during the re-evaluation of a consistency rule. This provides us with consistency information, that will necessarily be relevant during the later merging of the engineers’ work with the work of others. Engineers can preemptively fix and adapt their private changes to fit the public artifacts.

The complexity, correctness and applicability of our approach need to be discussed in particular:

Complexity

We analyze the run-time complexity of our mechanism as it is outlined in section 5.2. Since most read and write processes on the CEE are executed on mappings, the complexity of rule retrieval, information storage and feedback is relatively low (on average $O(1)$, respectively $O(n)$ at worst). Change analysis and data gathering, however, should be discussed in more detail:

- **Change Analysis:** A growing number of artifacts (n) may be relevant to a growing number of CRDs (m). This results in $n * m$ CREs to be created or re-evaluated. Usually the number of consistency rules is limited to a constant (i.e., it is not growing beyond a certain point). This leads to a run-time complexity of $O(n)$.
- **Data Gathering:** For the retrieval of individual timestamped artifacts, the same complexity applies as for the retrieval of any other artifact. In the data gathering process, we collect every element from the artifact storage that is related to a corresponding CRE. The complexity of this process is not affected by the selected timestamp configuration, since it only further filters possible selections from the artifact storage. This results in a runtime complexity of $O(n)$.

To summarize, the run-time complexity of our approach is mainly dependent on the change analysis, which executes at a complexity of $O(n)$. However, it should be noted that usually only a small subset of changes triggers a re-evaluation process. Irrelevant changes are filtered out with the help of scope elements (i.e., if the change's respective artifact is not contained in any scope it is immediately skipped). This minimizes the amount of changes that has to be considered by our approach significantly.

Correctness

To assure the correctness of our approach, we thoroughly test the implementation with the help of an exhaustive input simulation. In this simulation, we test all valid input sequences of a child and parent work area for a subset of consistency checking-relevant artifacts. Such input sequences involve, e.g., the creation of a context element, the modification of one of its properties, and its eventual deletion. We generate the sum of all input sequences by organizing valid inputs in a graph structure. Starting at all valid initial inputs, a full depth first traversal of this graph structure results in all valid input sequences. These input sequences can then be executed on the CEE. After each step, we check the CEE's artifact storage against a pre-defined post-condition. The post-condition defines the correct amount of CREs that must be present. In total, our exhaustive input simulation creates 2994 scenarios. We believe this lends credibility to the correctness of our implementation and re-assures that our approach can be used within a collaborative environment.

Applicability

To validate the applicability of our approach to different types of engineering artifacts, we conducted a case study, which incorporated different types of engineering artifacts [87]. In the said study, we used our approach to maintain the consistency between electrical models and their corresponding software controller, respectively their implementation. We integrated the corresponding artifacts into our CEE and manually established traceability links between them. Whenever a change happened on an artifact, the affected engineers would be notified about potential inconsistencies by the means of an on-screen warning.

6.4 Threats to Validity

We simulate the building process of models with the execution of a sequence of commands. While this sequence of commands is in line with the hierarchical structure of the built models, a real-life engineer may take a differently organized approach towards the building process. Most likely that approach would allow for more temporary inconsistencies within the PWA timestamp configuration. Such inconsistencies are normally tolerated during the creation of a single model element [2], which is then linked up to the rest of the model and brought into a consistent state. In our study, this is done immediately as all necessary

Table 2: An overview of applied consistency rules

Rule	Description	Artifact
CR01	Every lifeline must have a corresponding class.	uml:Lifeline
CR02	Every transition has to have a corresponding message.	uml:Transition
CR03	Statechart action must be defined as an operation in the owner’s class.	uml:Transition
CR04	Message actions must be defined as operation in receiver’s class.	uml:Message
CR05	Operation parameter names must be unique.	uml:Operation
CR06	An Operation has at most one return parameter.	uml:Operation
CR07	Interfaces can have at most one generalization.	uml:Interface
CR08	Interfaces can only contain public operations and no attributes.	uml:Interface
CR09	Class operations signatures must be unique.	uml:Class
CR10	Field names must be unique.	uml:Class

model elements become available, keeping inconsistencies and PA timestamp accesses at a minimum. Even with a minimization of these accesses, the study still demonstrates their relative commonness.

Another threat to validity is the choice of consistency rules. However, the rules applied in this study have already been used before by Kretschmer et al. [55] in an established and evaluated consistency checking approach. These rules, in fact, stem from the work of Torre et al. [86] – a systematic mapping study on UML consistency rules – in which rules were gathered based on their general applicability to a wide range of available UML models.

7 Related Work

To the best of our knowledge, no other CEE approach provides freely selectable timestamps on engineering artifacts, while computing the associated consistency information. However, certain aspects of our approach are partially covered in some other approaches as well. The following section gives an overview of relevant works concerning such aspects.

Global consistency checking: Global consistency checking plays an integral part in our approach. While various approaches towards regular consistency checking have been researched in the past (e.g., [37, 38, 66, 74]), global consistency checking generally plays a smaller role, mostly due to the lack of proper infrastructure (as provided with the CEE in this work). Nevertheless, some works concerning global consistency checking have been realized. In their work, Sabetzadeh et al. [77] propose their tool “TReMer+” for global consistency checking. The tool first merges the distributed models of different developers before checking consistency on it. The downside – and big difference to our approach – is the fact that it only focuses on homogeneous models. To unify heterogeneous models in a common representation both Herzig et al. [48] and Vangheluwe [93] rely on the application of directed graphs. In these, models are expressed through sets of triples. A triple consists of an object, an attribute and a value. Implicitly, our approach also represent models in the form of graphs – with artifacts being equivalent to objects, properties to attributes and references stored in properties to values. The direction of an edge can be expressed through the interposition of a typed link artifact, providing source and target properties. One advantage of our approach, however, is additional meta-information enriching property values (e.g. timestamps). The matter of heterogeneous models was also the focus of Koenig et al. [51]. In their work, they address the feasibility of local consistency checks with regards to the definitions of global consistency. The problem addressed is mostly due to the costly merging of different meta-models, since all different engineering artifacts adhere to different meta-models. In our approach, this problem is addressed through the application of a uniform data representation and custom tool adapters transforming engineering artifacts.

Collaborative modeling: There has been extensive research towards collaboration mechanisms within model-driven engineering (e.g., [13, 39, 41–43, 57, 96]). Franzago et

al. [40] conducted a mapping study, investigating currently available methods and techniques for collaborative modeling. In their findings, they conclude that only a minor part of the examined approaches covered multi-views of models and even less (only four approaches out of 106 selected papers) regard the collaborative editing of model artifacts in an asynchronous way, as it would be possible with our approach. Our interpretation of this number is, that – while interest in both the investigation and the execution of such approaches exists – the lack of a proper platform for the realization of asynchronous editing is a major hindrance to research in this regard; an issue that is overcome through the application of a cloud-based CEE in this work. Bartelt et al. [3] introduced a model repository enabling collaborative modeling on the basis of the jazz development platform. While their approach supports an abstract internal representation of models, the consideration of heterogeneous engineering artifacts in relation to these models is not covered. In their work, Di Rocco et al. [26] state to develop an extensible web-based modeling platform, providing a community-based modeling repository. While considering heterogeneous models, this approach does not further focus on global consistency checking. The relevancy of this aspect is illustrated by the deep interdependence of model artifacts with the rest of the development process. Our approach abstracts artifacts to a point, where semantic equivalencies between arbitrary engineering fields can be established and consistency can be checked globally. Collaborative modeling within a single environment has also been the focus of multiple literature reviews (e.g., [9, 16]). The unification, presentation and manipulation of model artifacts, as well as the management of consistency, plays a major role in the presented research. In relation to our own work, Model Join [12], Vitruvius [54], and Viatra [20] stand out most notably. These works consider multiple model artifact types (respectively, meta-models) within a single view. Engineers can then enhance the concrete artifacts with further information. However, only Viatra is giving the incremental recomputation of consistency information similar attention. The Viatra framework performs model queries and re-evaluations on the basis of changes. However, it does not natively implement these features on the foundation of a cloud infrastructure. The subsequent multi-user aspect is therefore not considered in this approach, as it is in ours. Similarly, the Vitruvius approach and ModelJoin create flexible views on artifacts by combining meta-models. Heterogeneous model artifacts can be unified in a non-intrusive way by combining legacy meta-models. In our approach, this is taken over by tool adapters, which dynamically add their type information at runtime.

Version-Control Tools: While our approach is implemented on a CEE, it is important to note that in principle any repository-based artifact storage, e.g., Git, could realize the presented work. Going by the example of Git, private changes – normally stored in the PWA – could be represented by changes stored in a private branch. Pushing to the Git repository’s master branch would be equivalent to committing the PWA to the PA. Different timestamp configurations could be realized by automatically merging the private branch’s changes with different versions of the master branch. However, it must also be noted, that the internal data representation of our CEE strongly simplifies the process of merging. By layering private changes of PWAs on public knowledge from the PA, the retrieval process for different timestamp configurations is significantly more straight-forward to implement. An automatic merging process in a Git repository may encounter various difficulties when overlapping changes exist. Naturally, this complexity also propagates into any further analysis or manipulation of the retrieved artifacts such as consistency checking and the storage of consistency information in the respective branches.

8 Conclusion

This work presented an approach for providing timestamp-based consistency checking in a collaborative engineering environment (CEE). Our approach enables engineers to

preemptively check whether their work is consistent with the latest or any previous changes published by other engineers. We considered three research questions: We asked which private/public version differences could occur in practice to ensure that our approach is able to handle all possible situations (RQ1). This is addressed in Section 5.1 as we discuss the PWA, the PA and the intermediate timestamp configuration. We answered how consistency checking in a CEE functions where every collaborating engineers obtains up-to-date consistency feedback about their private (local) changes with regard to the global (public) knowledge (RQ2). This is addressed in Section 5.2 by discussing the steps of a consistency checking mechanism under consideration of a timestamp configuration. Finally, we assessed the practical benefits (scalability, usefulness, and correctness) of a timestamp-based private/public consistency checking mechanism (RQ3). To this end, we also evaluated the approach on the basis of ten differently sized UML projects. Each project was analysed with regards to the consistency checker's accessing both the PWA and the PA configuration. We discovered that a substantial number of consistency checks requires access to the PA configuration (on average 20.5 percent). This demonstrates that global consistency checking based on a single version of engineering artifacts alone – as provided by the state-of-the-art tools – is insufficient for a holistic evaluation of consistency rules.

Paper F

Instant Distribution of Consistency-relevant Change Information in a Hierarchical Multi-developer Engineering Environment

At the time of writing the thesis at hand, this paper was still under submission.

Authors Michael Alexander Tröls, Atif Mashkooor, Alexander Egyed

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.

1 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.

Various works have proposed ways of unifying engineering artifacts in a collaborative engineering environment (CEE) ([87–90]). 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 Git¹ or SVN²) this allows the engineers to *natively* store heterogeneous engineering artifacts. On this foundation, the artifact’s interdependent properties can be compared and 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. Naturally, every set of changes has its own implications on the consistency of the artifacts. We compute the consistency information concerning these private changes before they are committed upwards in the hierarchy. Naturally, this makes it necessary to first distribute all consistency-relevant change information down the hierarchy for potential consistency re-evaluations. This way engineers can continuously check their work 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 2, we describe a typical consistency checking scenario. In Section 3, we outline the issues arising within the illustrative example. In Section 4, 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 5. We evaluate our work with an experiment and an exhaustive scenario simulation, which are discussed in Section 6. The related work is discussed in Section 7. The paper is concluded in Section 9.

¹Git: <https://git-scm.com/> (last accessed: 02.09.2020)

²SVN: <https://subversion.apache.org/> (last accessed: 13.08.2020)

Figure 1: An overview of the CEE and its functionality.

2 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 adaptations: 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 personal private work area to the joined private 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 must be 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 private work area, the said work area can be committed to the public area (step 6). After this commit, Carol will be made aware of the new specification and the changed activity diagram (step 7). She can now adapt her implementation accordingly (step 8) and commit her finalized 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.

3 Problem Statement

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

- **Merging delays:** In commonly used collaboration systems (e.g., Git or SVN), conflict identification is usually tackled when the work of different engineers is brought together (e.g., while merging branches on Git). Effectively, this means that 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 properly 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. An example would be flawed architectural implementations arisen through discrepancies with the original design.
- **Tolerated inconsistencies:** During the engineering process some temporary inconsistencies may be tolerated (e.g., between designs and implementation to guarantee a flexible workflow) [70]. However, in many cases the documentation of these inconsistencies is lacking. This holds the potential for flaws in the final product as tolerated inconsistencies may be forgotten, e.g., if certain engineers leave a company. Subsequently, the inconsistencies often remain hidden for the rest of the development. The results are errors that are – if at all – recognized much later, which makes them substantially harder to correct [84].

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 properly 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 potential refactoring phases, whose issues may otherwise already be deeply rooted within the engineering artifacts at the time of merging. Tolerated inconsistencies can be kept in check as consistency checking results – involving the said inconsistencies – are stored as a part of the propagated consistency information.

Figure 2: An overview of the data model utilized by the CEE. Artifact Storage Elements are represented in blue, Artifact Declaration Elements are represented in green and Consistency Checking Elements are represented in orange.

4 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.

4.1 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 Ar-

³Exemplary synchronization: <https://tinyurl.com/y7pny6ao>

tifact 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. 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 linked to exactly one work area. 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 artifacts, which consist of properties, which, in turn, consist of a sequence of changes. 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 highest 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 simply 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.

4.2 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.

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

- **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.

4.3 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:

$$\text{Conjunction}(\text{Operator}(\text{ExpressionA}, \text{ExpressionB})^{1...*})$$

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

self.type.name

would navigate from a certain artifact instance (“self”) through a link to its 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:** The consistency checker analyzes a triggered change event. The change event contains the changed property, the associated artifact and the initiator of the change. 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 for the said CRE.
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 from the results.
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 consistency results. This is done through the cloud environment. Of course writing the result into the CRE is another change event in the artifact storage, which can be caught by further mechanisms. These may provide additional feedback to the engineers.

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 material⁴.

5 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.

5.1 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.

5.2 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.

⁴Consistency Checking in the CEE: <https://tinyurl.com/y4pzx23s>

- **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 outdated. This can happen in the following scenarios SC1-7 (supporting illustrations for the scenarios can be found in the supplementary material available online⁵):

- **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).

⁵Supporting illustrations: <https://tinyurl.com/y93lo8bk>

- **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. If 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. This means if there is a CRE in a child PWA (e.g., because of a changed scope element), the corresponding CE, as well as the 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 F.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 perspective)
                create CRE for CE
            else
                retrieve CRE for CE
    re-evaluate all created or retrieved CREs
    store results in work area

case MODIFY:
    if (CRD is modified)
        foreach (CRE in CRD.CREs)
            retrieve corresponding CRE
    elseif (CE is modified)
        foreach((context-element-corresponding CRE)
            retrieve corresponding CRE
    elseif (scope element is modified)
        foreach (CRE where CRE.scope contains scope element)
            retrieve corresponding CRE
    re-evaluate all retrieved CREs
    store results in work area

case DELETE:
    if (CRD is deleted)
        if(alive CRD is not in PWA.changes)
            foreach(alive CRE in PWA.changes that is contained in CRD.CREs)
                store deletion of CRE in PWA.changes

        elseif (CE is deleted)
            if (alive CE is not in PWA.changes)
                foreach (alive context-element-corresponding CRE in perspective)
                    store deletion of CRE in PWA.changes

        elseif (scope element's instance is deleted)
            if (alive scope element is not in PWA.changes)
                foreach (CRE in PWA.changes
                    where CRE.scope contains scope element)
                    re-evaluate CRE
            store results in work area
```

Listing F.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.

Table 1: Numerical overview of the empirical case study results

Correctness	Information	No Information	Performance
Correct	59 (41.84%)	22 (15.71%)	268%
Partially Correct	49 (34.75%)	50 (35.71%)	0%
Incorrect	33 (23.4%)	68 (48.57%)	-51.5%
Tasks	141 (100%)	140 (100%)	

6 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.

6.1 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 [50]. The study was conducted with 36 bachelor-level computer science students at a 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 1). 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.

6.2 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 5), which secures the correctness of the outcome. The evaluation is conducted on a CEE implementation with the features presented in Section 4.

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). These inputs can be put into a sequence.
- 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.

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.

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 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 subgraph, 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.

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.

6.3 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.

Figure 4: A graph structure showing multiple possible sequences of inputs, starting with the creation of a CE. One possible sequence is highlighted in orange.

6.4 Threat to Validity

A scenario simulation may not pose a real-world utilization of the proposed mechanism. However, in the case of an exhaustive scenario simulation, such as ours, real-world input scenarios merely represent a subset of all possible scenarios. An exhaustive simulation poses as a holistic evaluation of the functionality of our mechanism and includes all potential input scenarios a regular user could perform. We, therefore, believe that an exhaustive scenario simulation evaluates the correctness of our work in a representative way.

7 Related Work

Collaborative engineering approaches: While various works have been conducted in the field of collaborative engineering (e.g., [13, 39, 41–43, 57, 96]) the literature review of Bruneliere et al. [9] focuses on model-driven engineering 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 [10] Kitalpha, [58], Model Join [12] and Viatra [20] 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 [54] 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 the computed information towards stakeholders using the workbenches.

An approach with stronger focus on collaboration is developed by Obeo⁶. This includes the Sirius project⁷ as well as commercial extensions built on top of it. These solutions

⁶Obeo: <https://www.obeo.fr/en> (last accessed 15.09.2020)

⁷Sirius: <https://www.eclipse.org/sirius/> (last accessed 15.09.2020)

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. [48] and Vangheluwe [93] 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. [19] 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. [32] 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 a relevant 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 relevant 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.

8 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 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. As a future work, we would like to check consistency within arbitrary groups of work areas (i.e., independently from any work area hierarchy). To achieve this, we plan to further expand the data gathering mechanism of the CEE's consistency checker.

Paper G

Ensuring Safe and Consistent Co-Engineering of Cyber-Physical Production Systems: A Case Study

Published / Accepted Journal of Software: Evolution and Process (At the time of writing the respective issue and pages were not yet available.)

Authors Michael Alexander Tröls, Atif Mashkoor, Andreas Demuth, Alexander Egyed

Abstract In today's engineering projects, companies continuously have to adapt their systems to changing customers or dynamic market requirements. This requires a flexible, iterative development process in which different parts of the system under construction are built and updated concurrently. However, concurrent engineering becomes quite challenging in domains where different engineering artifacts from different disciplines come into play, such as safety-critical cyber-physical systems, where the involved engineering artifacts are quite heterogeneous in nature. In such systems, it is of utmost importance that different artifacts remain consistent in order to guarantee a correctly functioning end product. In this article, we discuss our experiences (with a leading company working in the areas of production automation and product processing) in maintaining the consistency between electrical models and the corresponding software controller, when both are subject to continuous changes. The article discusses how we let engineers describe the relationships between electrical models and the corresponding software controller code in the form of links and consistency rules. Additionally, we demonstrate that how our approach, through a process of continuous consistency checking, notifies engineers about the erroneous impact of their changes in various engineering artifacts.

1 Introduction

Modern engineering projects often span across multiple domains and disciplines. It has become common that engineers with diverse backgrounds collaborate in order to develop complex systems. For example, developing a cyber-physical system typically requires software engineers, hardware engineers, electrical engineers, requirements engineers, mathematicians, physicists, and others to work together closely. Typically experts from these different domains are responsible for different engineering artifacts. For instance, the requirements the system under development must fulfill may be negotiated with stakeholders by the requirements engineer. The requirements may then be used by a mathematician to calculate data that must be known to realize the project. This data may then, in turn, be used by an electrical engineer to create a model of the hardware setup. Based on this hardware setup and other calculated data, a software engineer may, finally, develop the source code that controls the hardware components.

During the initial development of the aforementioned artifacts, there is typically communication between experts from different domains. For example, once the hardware design is finished, the hardware engineer informs the software engineer about which hardware components will be used and thus have to be controlled by the control software. Each engineer will also quite likely inform the corresponding supervisor about a finished artifact. Therefore, during the initial development, knowledge is typically passed on quite well between involved engineers and different artifacts are likely to be consistent with respect to each other. However, such communication often happens informally, and the knowledge passed during this communication might not be documented. This becomes a problem especially when an artifact is evolved past its initial stage, for example, in an iterative development process individual artifacts are evolved multiple times until they reach their final state when the project is finished. In this case, the rationale for certain decisions is often no longer available for engineers. For instance, it might no longer be clear why certain source code fragments are needed for implementing the controller of a piece of hardware. If an engineer removes a component from the hardware model, software engineers might have a hard time locating the corresponding source code fragments. As a consequence, changes because of evolution or maintenance may easily introduce inconsistencies between different artifacts and the impact of a change is hard to estimate. This is especially challenging in cyber-physical systems due to the highly interwoven nature of their engineering artifacts. The development of such systems commonly involves engineers from highly diverse backgrounds. The more diverse the engineering disciplines in a development project are, the more difficult it becomes to maintain consistency between artifacts throughout multiple development phases.

The maintenance of consistency is critical for the safety aspect of cyber-physical systems. Since safety plays a major role in modern cyber-physical systems [5, 63, 64], safety analysis must accurately capture the potential harm a system may cause to its operators. With inconsistencies between engineering artifacts and the propagation of arising errors, the results of such analysis may be distorted. From a safety perspective it is therefore highly important to guarantee both a reliable and comprehensive analysis of artifact consistency.

These are exactly the issues that the Belgian company Van Hoescke Automation¹ (VHA) – a leading provider of automation solutions – is facing. In this article, we present the results of a collaboration between VHA, Flanders Make², and Johannes Kepler University³ (JKU) that addresses these issues. In particular, we describe a solution that enables traceability, consistency checking, and change impact analysis between different artifacts in the VHA’s development process. The solution relies on existing principles of traceability

¹<https://www.vha.be/> (last visited: 25.06.2020)

²<https://www.flandersmake.be/> (last visited: 25.06.2020)

³<https://www.jku.at/> (last visited: 25.06.2020)

and incremental consistency checking (e.g., the *DesignSpace Information Integration Platform* [25] and the *Model/Analyzer Consistency Checking Framework* [29,72]). Moreover, new technologies for establishing traceability and the application of consistency checking to different artifacts have been developed.

This article is an extension of our previous work [21]. The extension includes a broad update of definitions and concepts according to the latest implementation standards of the presented solution. We also report improvements on the principles of engineering artifact integration, the adaption of types in the integration platform, the organization of data within the artifact storage as well as the inner workings of the consistency checking mechanism.

The remainder of the paper is organized as follows. In Section 2, we give background information about Van Hoescke Automation and the Flanders' Mechatronics Technology Centre. In Section 3, we present a challenge problem that Van Hoescke Automation actually has been facing. In Section 4, we outline the specific goals and requirements of Van Hoescke Automation regarding a better support of artifact evolution and maintenance. Section 5 then describes the solution that has been developed to address the challenge problem and to meet the stated goals and requirements. Section 6 then shows how the developed solution is applied to the challenge problem. In Section 7, we discuss how the developed solution actually meets the goals and requirements. Moreover, in this section we discuss aspects such as expected applicability and benefits of employing the solution, and the challenges that we encountered during the development. Section 8 discusses the related work. The paper is concluded in Section 9.

2 Background

Van Hoescke Automation is a leading Belgian Family company, founded in 1990, providing advanced solutions for production automation, product processing, and product inspection and tracking. Their main sectors of operation include the make industry with an important international activity in the automotive and OEM sector, and the food industry. Key competences of VHA regarding the discipline of software engineering include the development of production supervisory systems by means of standard tools and the development of customer specific machine applications. As it is typical for companies dealing with cyber-physical systems and production automation, VHA employs experts from diverse domains such as business management, software engineering, mechanical engineering, electrical engineering, mathematics, or physics.

Flanders Make is the strategic research centre for the manufacturing industry with establishments in Lommel and Leuven, Belgium. The centre collaborates with research labs at various Flemish universities as well as with major companies and SMEs. Together, they focus on product and process innovation based on the challenges and needs of the industry. The research focus is on 4 technological domains: power electronics and energy storage, mechatronics and design methods, production processes, and people-driven system development. The primary goal of the collaboration is to yield product and process innovations in 3 fields of application: vehicles, machines, and factories.

In this project, VHA and Flanders Make collaborated with JKU to combine their experience in the development of mechatronical systems with leading research on software traceability and consistency checking. The results are meant to not only be beneficial for the industrial partners with respect to their development process, but also to serve as a case study for the application of research technologies from academia in an industrial context.

Figure 1: Incremental development of the conveyor belt system

3 Problem Statement

VHA produces conveyor belts which are powered by asynchronous motors. The motors are controlled by a programmable logic controller (PLC). The design of the conveyor belt involves four different engineers: i) a systems engineer, ii) an electrical engineer, iii) a safety engineer, and iv) a software engineer. The process is depicted in Fig. 1. Each engineer is using specific tooling during the following individual steps of the scenario:

1) [System concept] The systems engineer creates the specification of the initial system concept (as shown in Fig. 2).

2) [Electrical design] Eplan Electric P8⁴ is used by the electrical engineer to specify the electrical connections between actuators (e.g., a motor) and the PLC (a high-level view of the electrical design is shown in Fig. 3, a snippet from the actual Eplan P8 model is shown in Fig. 4). The Eplan P8 model snippet depicts the control signal of the conveyor motor as the component 21K3 and the normal output of the Siemens PLC as the component Q70.0.

3) [Software design] In the example, the Eclipse IDE with Java is used as a proxy for the regular PLC code in which the control logic of the actuators are implemented (as shown in Fig. 5). Notice that for the two IO ports of the Siemens PLC, there are two variables defined: i) `hw_IO_out0_1` representing the normal output (Q70.0 in Fig. 4), and ii) `hwIO_out1_1` representing the safety output of the Siemens PLC (not depicted in Fig. 4). Moreover, the variable `fb_motor_control` represents the control signal of the motor conveyor (21K3 in Fig. 4). In the last line of Fig. 5, the linking of the components Q70.0 and 21K3 from Fig. 4 is reflected.

4) [Safety analysis] Microsoft Excel is used by the safety engineer for the analysis of potential harms to (human) operators caused while operating the system, using a template specifically designed at VHA (as shown in Fig. 6).

⁴Eplan Electric P8: <https://www.eplanusa.com/us/solutions/product-overview/eplan-electric-p8/> (last visited: 25.03.2020)

5) **[Electrical design update]** The electrical engineer performs updates of the electrical connections to avoid the harms identified in the previous step (as shown in the high-level view in Fig. 7 and the actual EPlan P8 model snippet in Fig. 8). Notice that in Fig. 8 the control signal (component 21K3) is now connected to a different component (Q30.1), which represents the safety output of the Siemens PLC.

6) **[Software design update]** The software engineer performs updates of the software so that it is in sync with the updated electrical connections (as shown in Fig. 9). Notice that in the mapping of IO ports, the control signal (`fb_motor_control`) is now linked to the safety output of the Siemens PLC (`hwIO.out1_1`).

Note that after Step 5, the update of the electrical design, there is an inconsistency between the electrical design and the control software, and therefore an update of the software design is required. Specifically, in the EPlan P8 model snippet in Fig. 8 it can be seen that the component 21K3 is no longer connected to the component Q70.0 (as in Fig. 4), but instead it is now connected to the safety output of the Siemens PLC, which is modeled as the component Q30.1. The source code from Fig. 5 still reflects that 21K3 and Q70.0 are connected. However, VHA has experienced that in such a scenario the inconsistency may be overlooked by engineers and may thus remain undetected, resulting in Step 6 being not performed in practice. This might be caused by a lack of support for traceability, change impact analysis, and inconsistency detection. The consequences of not detecting the inconsistency and not performing Step 6 in the scenario include, for example, delays in the development progress or failure to deliver a functioning system. Therefore, it is crucial for VHA to introduce support for traceability, change impact analysis, and inconsistency detection in their development process.

Figure 2: Initial system concept from Step 1.

Figure 3: Initial electrical design from Step 2.

4 Goals and Requirements

Let us now summarize the goals and specific solution requirements stated by VHA. The main goals of the project presented in this article are:

- to increase awareness about consistency between spreadsheet data, source code, and electrical models,
- to increase traceability between these engineering artifacts, and
- to get feedback about consistency and change impact.

Figure 4: Initial EPlan P8 model from Step 2.

```
//Q70.0
int hwIO_out0_1 = 0xc001;
//Q30.1
int hwIO_out1_1 = 0xc001;
//21K3
int fb_motor_control = 0xc002;

//linking
IO_mapping.put(hwIO_out0_1, fb_motor_control);
```

Figure 5: Initial software design from Step 3.

A key requirement stated by VHA is the support for the following technologies: Microsoft Excel Spreadsheets and Eplan Electric P8. Additionally, in the example, the Eclipse IDE with Java is used as a proxy for the regular PLC code development. Engineers using either tool must be notified about arising inconsistencies. Moreover, it must be possible to analyse the impact of a change and to track dependencies between different artifacts. Additionally, the current workflows of the individual disciplines should not be disrupted significantly (i.e., the specialized engineering tools used currently must not be replaced). As we have already illustrated VHA’s challenge problem and discussed the goals and requirements they stated, now we present the solution that was developed in the project.

5 Approach

One concern about the realization of the outlined goals was the potential interference with the established development process of VHA. Therefore, we decided to apply a solution based on existing approaches. In the following, we first discuss a general overview of our solution’s architecture and then describe its individual parts.

As discussed above, incremental consistency checking and traceability between different artifacts are key features our solution must provide. We abstract these concepts in a way that they are adaptable on a heterogeneous engineering artifact basis. The original engineering artifacts (e.g., Excel files) should remain unchanged. Therefore, they are duplicated in the employed data integration platform. In this work, we exploit the DesignSpace cloud environment [25]. A service running on top of the DesignSpace platform is used for consistency checking, which has been discussed in previous works (e.g., [88–90]). Further, we apply another service enabling traceability between different existing artifacts. We first discuss how artifact integration works in our solution, then we present how traceability can be established using the integrated data. We finally show how consistency checking is realized on the integrated data and existing traceability information.

Danger	Likelihood	frequency	...	Corrections
Pinched in between conveyor belt	Probably	Daily	...	fail safe on conveyor motor

	Required SRP/CS level			
Safety Functions	S	F	P	SILr
Fail safe conveyor motor	S2	F1	P2	2

Figure 6: Initial safety analysis from Step 4.

Figure 7: Updated electrical design from Step 5.

5.1 Artifact Integration

As discussed above, the DesignSpace integration platform is used in this project. This platform provides support for the transparent integration of arbitrary engineering artifacts. In this work, all engineering artifacts remain unchanged at their original location (e.g., on an engineer’s local hard drive) and the DesignSpace only holds a replication of the original information that is synchronized live. Thus, artifact integration with the DesignSpace does not affect existing processes and practices. Engineers are still able to use their favorite engineering tools for developing their artifacts. This is important for two reasons. First, engineers are typically reluctant to change the used tools, which is understandable because engineering tools are typically highly specialized for performing certain tasks as efficient as possible. Thus, trying to replace such highly specialized tools is typically a non-optimal solution with respect to the quality and the efficiency of results. Second, developing a new engineering tool for a specific domain requires years of development, with only small chances of coming up with a tool that leads to better — or even similar — results than the leading existing tools for the domain.

In the DesignSpace, engineering artifacts are represented in the form of property-value mappings. An artifact consists of several (named) properties, which may point toward basic value types (integer, float, boolean, string) or yet more artifacts (references). This data structure is illustrated in Fig. 10. It is of importance to note at this point, that a value, basic or referential, is not merely a single value, but a list of deltas, that describe the change history of the property. Nothing in the DesignSpace is ever truly removed, but merely overwritten or flagged as “not alive”. It is saved for every artifact, on which an engineer performed the last modification along with a timestamp when this modification happened. Further, every artifact is uniquely identifiable with an ID and adheres to a specific user-defined type.

The type of an artifact defines the structural components of its concrete instantiation. This means, an artifact type describes properties, i.e., property names, cardinalities (whether a value is a list or a single value), and the datatype. When an engineering artifact is synchronized with the DesignSpace, it automatically instantiates the corresponding type and fills the properties accordingly.

To synchronize an engineering artifact with the DesignSpace, we may employ one of two principles:

- **File-Adapters:** One option is to use file-adapters that translate file contents into

Figure 8: Updated EPlan P8 model from Step 5.

```
//Q70.0
int hwIO_out0_1 = 0xc001;
//Q30.1
int hwIO_out1_1 = 0xc001;
//21K3
int fb_motor_control = 0xc002;

//linking
IO_mapping.put(hwIO_out1_1, fb_motor_control);
```

Figure 9: Updated software design from Step 6.

the DesignSpace’s internal data structure. With file-based integration, the synchronization of the original engineering artifact and its duplicate in the DesignSpace can only be done when the file is saved, requiring the use of diffing tools or inefficient overrides of existing data.

- **Tool-Adapters:** The second option is to use tool-adapters that directly translate and synchronize information from engineering tools with the DesignSpace. Note that this can be done live and incrementally while engineers work on their respective engineering artifacts. Moreover, by using tool-adapters it is possible to present information about consistency or trace information to the engineer directly in the engineering tool. Therefore, the use of tool-adapters is generally preferable.

Figure 10: The uniform artifact representation employed in the DesignSpace

For each of the three tools that are used by VHA in their challenge problem, we developed three different tool adapters, which we present next.

Microsoft Excel

For Microsoft Excel, data integration is performed at cell level through an Excel plugin written in C#. We decided not to duplicate all cells in a spreadsheet with the DesignSpace, but only those cells that are of relevance for other engineering artifacts. This decision is based on the fact that, due to the nature of spreadsheets, a vast majority of existing cells may be either empty or contain information that is required only for some internal calculations but does not have any effect on other artifacts. Instead, those cells in the spreadsheet that are of relevance and should therefore be synchronized with the DesignSpace for later use by the traceability and consistency checking services are annotated by an engineer. This avoids unnecessary communication overhead and allows for easier establishing of traceability, as we will see below. The annotations are added to each cell individually using Excel's comment function. To distinguish comments for data integration from others, the data integration comments must conform to the following syntax: `$name=<name>; $unit=<unit>; $meta=<meta information>`. The `$name` attribute allows engineers to specify a name by which the cell can later be referenced for traceability and consistency checking. The `$unit` attribute defines the unit of the value saved in the annotated cell. The attribute `$meta` allows engineers to specify additional meta information, for example, if a cell is representing some domain concepts. As an example, consider in Fig. 6 the cell with the value **Probably** (2nd row, 2nd column). Using the `$meta` attribute, an engineer can define that this cell's value represents a likelihood. Note that another option would have been to use predefined templates that define which cells should be synchronized. However, using annotations provides more flexibility to engineers in case templates are changed.

Eclipse IDE with Java

For the source code development, in this project we relied on the Eclipse IDE. Therefore, we developed a tool-adapter plugin for the Eclipse IDE and the Java programming language. Notice again that in practice PLCs are not programmed using Java. However, in this project we agreed with VHA to use Java for demonstrating the feasibility of checking consistency between EPlan P8 models and the source code. VHA is confident that if consistency checking is feasible with Java, it is also feasible with an actual PLC programming language. Similarly to the tool-adapter for Microsoft Excel, this adapter does not synchronize the entire source code with the DesignSpace platform, but only those parts of it that are relevant for other artifacts. As for excel, this reduces the amount of information that is available in the DesignSpace for establishing traceability. Engineers can mark relevant parts by adding annotations as comments. Note that such annotations can be made for any element of a program (e.g., classes, methods and functions, and individual statements). The synchronization with the DesignSpace is not performed immediately when an annotation is added, but it is performed each time the Eclipse IDE's compiler is executed, which is typically after each save triggered by the user.

EPlan Electric P8

For electrical models created with EPlan Electric P8, we followed a similar approach as for Microsoft Excel and the Eclipse IDE with Java. A tool-adapter is used that observes the electrical model. Any model elements that should be synchronized with the DesignSpace have to be marked by adding a custom property to the element (e.g., to a modeled component). However, in EPlan it is not possible to select connections between elements and

Figure 11: Wizard for creating a link between EPlan model element and its corresponding source code fragment.

set this property. Therefore, the tool-adaptor also synchronizes all connection elements that connect two (or more) already marked (and therefore synchronized) elements. For instance, if a motor element is connected to a controller element, then the connection between the motor and the controller is also synchronized with the DesignSpace. An example of this behavior is shown in Fig. 11. Specifically, the left-hand side of the figure shows three model elements that are synchronized with the DesignSpace: Q70.0, Q30.1 (only visible partially to the right of Q70.0), and 21K3. The corresponding DesignSpace representation of the EPlan model is shown on the right-hand side of the linking wizard in the center of Fig. 11. Notice that there are four entries: one for each of the marked model elements, and one additional representation of the link between Q70.0 and 21K3. This link has been created in the DesignSpace automatically.

Integrated Type Information of Engineering Artifacts

As discussed above, the DesignSpace platform internally uses a simple, graph-like data structure that represents any information as artifacts and properties. Alongside this graph of artifacts, the DesignSpace also stores corresponding type information. This type information describes the structure of engineering artifacts as they are represented on the DesignSpace.

The type information can be drawn from corresponding metamodels (e.g., the EPlan metamodel, which describes the structure of EPlan models). Since not all engineering artifacts have defined metamodels, type information can also be created from scratch by a domain expert. While the engineering artifacts are modeled as logical instantiations of their types, the representations of both instantiations and their types, are runtime elements of the DesignSpace's architecture. Therefore, it must be distinguished between linguistic and ontological levels. Linguistic levels define technical instantiation at runtime, whereas ontological levels describe logical instantiation.

In Fig. 12, the linguistic and ontological levels are depicted. Notice that we use the same notation for representing the individual engineering artifacts at the ontological level 00 and their respective types at the ontological level 01. This indicates that both levels are runtime instances of the linguistic level L1. The distinction between the two ontological

Figure 12: Engineering artifacts and their types integrated in the DesignSpace.

levels is purely virtual and based on interpretation of the runtime data structures. This means, that both instantiations and types are based on the same abstract entity within the data structure (i.e., everything is treated as an artifact - even type information). For each integrated engineering artifact its corresponding type remains unchanged in principle – the type may only be extended with additional information but the original concepts are retained. This means that any such issue would be avoided that typically arise when trying to merge different metamodels into a single one [34, 53]. In this project, we used metamodels to create the corresponding typing information. However, the typing information of synchronized engineering artifacts has been simplified. This was not only done to simplify the development of tool adapters but also to reduce the complexity of linking and consistency checking from the perspective of the involved engineers. Specifically, the typing information of EPlan models contains only two entities: **EPlanModelElement** and **EPlanModelLink**. The former is used to represent any first-class entities of EPlan models (e.g., components), whereas the latter is used for representing any links that are drawn between first-class entities (e.g., when an input of a component is linked to another component’s output). For the source code, the typing information contains also two classes: **SourceCodeFragment** and **SourceCodeLink**. A **SourceCodeFragment** can be any part of source code, e.g., a class, a method, an individual statement, or even a simple comment. A **SourceCodeLink** is used to model linking and mapping of the source code fragments. For instance, a **SourceCodeLink** between two variables is used to model that the two variables are used as a key-value pair in a map. For Microsoft Excel spreadsheets, the tool adapter only synchronizes individual cells. Therefore, a single typing element, **SpreadsheetCell** is sufficient for this project.

At the ontological level O0 in Fig. 12, the state of the DesignSpace after step 4 from the challenge problem in Section 3 is shown. Note that the relevant elements from each of the three involved engineering artifacts have been synchronized with the DesignSpace automatically by tool-adapters after they have been annotated by engineers in their respective engineering tools. For the source code, the three variable and the IO-port mapping from Fig. 5 are represented. For the EPlan P8 model, the three components that model the control signal of the motor conveyor and the two outputs of the Siemens PLC are repre-

sented. The links between the elements, which have been generated automatically by the respective tool adapters, are drawn as boxes with grey background. For the Excel sheet from Fig. 6, there are no annotated cells, meaning that nothing has to be synchronized with the DesignSpace.

5.2 Work Areas

Once synchronized, the engineering artifacts are stored in the DesignSpace's artifact storage. The artifact storage consists of both a public area (PA) and a set of private work areas (PWAs). The PA contains all information on artifacts that have been synchronized and published by the engineers. The PWAs contain only deltas (changes) of artifacts with regards to their publicly available version. The work areas are arranged in a tree structure, where the PA is the root. PWAs are attached as leaves below the root. Retrieving engineering artifacts from a PWA layers the work area's deltas on top of the respective property values in its parent. This way an artifact can always be retrieved from a certain engineer's perspective, incorporating their changes. This holds the major advantage that changes can be seen in relation to publicly available (and potentially changing) properties without integrating these changes into the public version of the artifact. This concept can be exploited in consistency checking, as it allows us to check the consistency of changes that have yet to be published. The full concept of layered work areas can be seen in Fig. 13. In this illustration, all changes in the PWA overwrite their respective equivalent in the PA. The link represented as the artifact EML1 holds a property *in* towards Q70.0 and a property *out* towards 21K3. Both PWA1 and PWA2 hold different changes on these two properties respectively. When the artifact EML1 is retrieved in full, the DesignSpace will return the public version, complemented by the changes of the PWA used for retrieval. For example, PWA1 will retrieve EML1 with an *in* property towards Q71.0 and an *out* property towards 21K3. The PWA2 will retrieve the same properties with Q70.0 and 31K3.

This concept of keeping engineering artifacts in a layered hierarchy of work areas comes with various advantages. There is a separation of personal changes and public knowledge. This gives engineers an individualized view on their engineering artifacts. Changes are always kept separate from public knowledge until they are published. Yet a full engineering artifact is always retrieved on the basis of its potentially changing public version. This means the individual private perspective of the engineer is always kept up to date (unlike in most contemporary version control systems, where being up to date with a public repository requires checking out its latest state). With regards to consistency checking this provides the advantage, that a private change can always be checked with regards to publicly available knowledge. This allows consistency checking from the perspective of a specific engineer's work, without publishing the said work. Supporting such private perspectives puts this work into contrast with traditional consistency checking approaches (e.g., [37, 38, 68, 77]), which are either entirely focused on publicly available knowledge or on an incomplete subset thereof.

5.3 Traceability

A major goal of VHA for this project was to allow for traceability between engineering artifacts. After a change has been performed it should be possible to easily identify other artifacts that might be affected by the change. Further it should be possible to identify the associated engineers.

A major challenge with regards to traceability is the heterogeneous nature of engineering artifacts. Different kinds of engineering artifacts underlie different syntactic and semantic rules. Connecting these artifacts for the purpose of traceability has to overcome these semantic and syntactic differences. This problem is often tackled by managing traceability information in separate traceability tools. In these tools, traceability links as well as

Figure 13: A layered hierarchy of work areas.

Figure 14: Extension of type information for traceability and linked engineering artifacts.

engineering artifacts are captured independently from the actual engineering artifacts. In other words, information is duplicated and analysis is done on a minor subset of the original engineering artifacts. This complicates a comprehensive analysis of the resulting traceability graph. It further comes with the disadvantage that traceability links have to be maintained independently. This means, whenever an engineering artifact changes in a way that affects traceability, this change has to be propagated through various tools. Not only does this cause extra costs for maintenance, but without highly detailed documentation this process may quickly become a source of error.

We avoid these problems in our approach by live synchronizing changes of engineering artifacts with the DesignSpace. Every potential change to traceability information is immediately realized in the platform holding this information. Further, we can analyse any change through a consistency checker, making sure that traceability information is correct and complete. Further, the constructed traceability graph can be used to analyze the propagation of inconsistencies identified in an engineering artifact.

To realize traceability maintenance on the DesignSpace, we first need to integrate all involved engineering artifacts in the uniform artifact representation as described in Section 5.1. After integrating all involved artifacts with the DesignSpace data integration framework, its linking tool can be used to establish traceability between (parts of) the different artifacts. Two artifacts can either be linked together by referencing each other through a property, or by referencing an interposed link artifact. This typed link artifact stores both a source and a target, as well as additional information (e.g., name and description). Using an additional artifact for linking has the advantage that the links themselves become concrete engineering artifacts. For traceability, this means that traceability links can easily be retrieved for further analysis and manipulation.

For creating, updating, and deleting links between the different artifacts, we developed a standalone desktop application, the DesignSpace *Linking Tool*. While the creation of links may be done (semi-)automatically using state-of-the-art heuristics to identify the parts that are logically connected (e.g., a cell in a spreadsheet that is used for defining a constant in the source code), the solution we developed relies on manual linking performed by engineers.

The application visualizes any information present in the DesignSpace uniformly and traces can be established between arbitrary pieces of information. For example, one cell in an Excel sheet may be linked to another cell in the same spreadsheet, to another cell in a different spreadsheet, to a constant or also to a method in the source code, or to an element of the electrical model. To create a link, an engineer simply has to specify the two pieces of information that are logically connected. Different kinds of links can be used to express additional meaning or to restrict the information that is eligible as a source or a target of the link. For example, for this project, in addition to the linking artifacts already presented above - which are responsible for linking information between engineering artifacts of the same type - we provide by default three kinds of links that may be used to link different types of engineering artifacts: **Excel-to-Code**, **Excel-to-EPlan**, and **EPlan-to-Code**. These kinds of links simply restrict the source and target information to specific artifact types. For instance, an **Excel-to-Code** link allows only cells from an Excel spreadsheet as a source and only elements of the source code as a target. These three kinds of links are considered bidirectional; they just express that there exists a connection between the linked elements. Fig. 14 depicts the type information for linking artifacts used in this project. Note that it contains one artifact type for each of the pre-defined types of traces at the ontological level 01. In Fig. 11, the creation of an **EPlan-to-Code** link is shown. Specifically, the left-hand side of the figure shows the EPlan model, whereas on the right-hand side the Eclipse IDE with the Java source code is shown. The component named Q70.0 in the EPlan model (selected and drawn in blue), which models the **Normal Output** IO-port of the **Siemens PLC** in

Fig. 3, should be linked to the variable in the source code that reflects this IO-port `hwIO_out0_1`. In the center of Fig. 11, the linking tool for creating such a link is shown. In this tool, the engineer has opened the information available in the DesignSpace of the EPlan model (right-hand side of the tool) and the source code (left-hand side of the tool). The engineer simply selects the DesignSpace representation of the IO-port (`Q70:0`) and the corresponding source code fragment representation (`hwIO_out0_1`) to establish the desired link. After linking each EPlan P8 model element to its corresponding source code fragment, the resulting DesignSpace representation is depicted in the bottom part (i.e., the ontological level 00) of Fig. 14.

Note that additional kinds of links can be defined by engineers freely and on-demand. For example, an engineer may want to connect certain elements in the electrical model to a requirement, while another engineer may want to use a link to express that a certain constant in the source code exactly reflects a value that is defined in a spreadsheet. Note that it is possible for engineers to express link semantics through additional link constraints. This allows, for instance, for different units in different artifacts. For example, a link may be enriched with a constraint that ensures correct translation between a metric value in a spreadsheet (e.g., 2,400 mm) to an imperial value in an electrical model (e.g., 94.48819 inches), or 2.4 kW to 2,400 W. The *Object Constraint Language (OCL)*⁵ is used for defining these constraints. Using OCL, more complex and sophisticated constraints can also be stated and validated. Next, we present in detail how constraints are written and validated in this work.

5.4 Consistency Checking

As discussed in Section 4, the main objective of this project was to support consistency checking among VHA's major engineering artifacts. Above, we have seen how these artifacts are integrated with the DesignSpace platform and how they can be connected in the DesignSpace to establish traceability. Besides its obvious benefits, traceability is also a requirement for enabling the employed consistency checking approach. Specifically, our solution relies on a variation of the Model/Analyzer consistency checking framework [30, 72] adapted for the DesignSpace platform. The Model/Analyzer framework allows for highly efficient, incremental consistency checking of any information that is available in the DesignSpace.

In our variation of the framework, the Model/Analyzer is directly integrated into the DesignSpace as a platform-sided service. This allows for the propagation of consistency information, directly through the internal artifact structure of the DesignSpace. This gives us the possibility to treat consistency information as an additional type of engineering artifact that is maintained alongside the other engineering artifacts. Naturally this also requires us to introduce Model/Analyzer concepts in the form of typed artifacts. For this work, we introduced two types of artifacts:

- **Consistency Rule Definition Artifacts (CRDs):** A Consistency Rule Definition holds both a consistency rule (in the form of an OCL expression) and a reference to the artifact type for which the rule should hold (the context of the rule). These artifacts are provided by the users of the DesignSpace.
- **Consistency Rule Evaluation Artifacts (CREs):** A Consistency Rule Evaluation Artifact realizes a CRD for a specific engineering artifact and stores a result and a scope. These artifacts are automatically instantiated by the DesignSpace consistency checking service.

When a CRD is created, the consistency checking service automatically creates the corresponding CREs for each artifact related to the defined context.

⁵OCL: <https://www.omg.org/spec/OCL/About-OCL/> (last visited: 26.03.2020)

It is important to note that both CRDs and CREs underlie the same conditions of layered representation as all other engineering artifacts in the DesignSpace. This means if somebody changes a CRD, the change will be stored within that person's PWA. Naturally, this has an effect on the distribution of CREs as well. When a change happens in a work area, the CRE must be re-validated for the said change. The result of this re-validation is stored as a change on the CRE artifact in the PWA from which the re-validation was triggered. This enables comprehensive consistency information for every PWA containing consistency-relevant changes in the DesignSpace. This allows to provide a work-area-specific perspective on the consistency state of engineering artifacts stored on the DesignSpace. To enable incremental re-validation after changes of information, our consistency checking service reacts to changes in the data structure of the DesignSpace. A change in the DesignSpace causes an event that can be listened to by a service. Therefore, a re-validation via the consistency checker is triggered whenever the information available in the DesignSpace changes (i.e., whenever an engineer performs a change in his or her local tool and the change is synchronized with the DesignSpace). However, the re-validation works incrementally, meaning that only those consistency rules are re-validated that are potentially affected by the change. This is done with the help of a **scope** that is attached to each CRE. The scope documents every property – respectively artifact – that is involved in the re-validation of a consistency rule. If one of these properties is changed, the consistency checker can identify associated scopes with the help of the change event. The CREs carrying these scopes are then re-validated.

It has been shown that this incremental re-validation of consistency rules after changes to the checked data is typically performed within milliseconds. Thus, the information about changes regarding consistency between engineering artifacts can be provided instantly to engineers. All tasks regarding consistency checking (e.g., definition of consistency rules or the notification about the visualization of inconsistencies) are done in a custom tool developed for this project.

Indeed, in order to check consistency among information from different artifacts, besides the links between these artifacts, there must exist consistency rules that express their desired relation. These consistency rules are written in OCL, as already briefly discussed above. For each consistency rule, a *context* must be chosen. This context defines which artifact types are affected by the rule. Specifically, the consistency rule is validated individually for each logical instantiation of the defined context (i.e., a rule context at the ontological level L1 means that each instantiation of the rule context at L0 is validated). When using our variation of the Model/Analyzer and the DesignSpace platform, any piece of information available in the DesignSpace may be used as a consistency rule's context. For instance, a consistency rule with the context `Class::Java` means that the rule is validated for every occurrence of a Java class in the DesignSpace. In the following, we discuss four relevant consistency rules used in this project.

EPlan-to-CodeLink

Let us first show how consistency rules can be used to ensure that the traceability features discussed above are used correctly. We must make sure that both the `EPlanModelElements` and the `SourceCodeFragments` can refer to the respective links between them. As an example, consider the link created between the electrical component `Q70.0` and the source code fragment `hwIO_out0_1` in Fig. 11. This particular link should be navigable from both `Q70.0` and `hwIO_out0_1` by using the reference named `ECLink` of the respective DesignSpace representations of the two components (i.e., `(Q70.0).ECLink` and `(hwIO_out0_1).ECLink` should point to collections of **EPlan-to-Code** links that the respective elements are part of, and these collections should both include the specific link). This can be ensured with the consistency rule shown in Listing G.1. The context of this consistency rule is

```

1 [Context: EPlan-to-CodeLink]
2 self.source.e2c->contains(self) and
3 self.target.e2c->contains(self)

```

Listing G.1: Consistency Rule for EPlan-to-CodeLink

```

1 [Context: SpreadsheetLink]
2 self.source <> self.target and
3 self.source.link->contains(self) and
4 self.target.link->contains(self)

```

Listing G.2: Consistency Rule for SpreadsheetLink

EPlan-to-CodeLink. Therefore, the consistency rule is validated individually for each occurrence of such a link, including the **EPlan-to-CodeLink** between `Q70.0` and `hwIO_out0_1`.

SpreadsheetLink and SourceCodeLink

Similarly to the **EPlan-to-CodeLink**, we want to ensure that links connecting different spreadsheet cells or source code fragments are used correctly. As shown in the type information for linking in Fig. 14, a spreadsheet cell's representation in the DesignSpace may be linked to another cell by a **SpreadsheetLink**. However, it is not desired that a cell can be linked to itself. Therefore, the consistency rule in Listing G.2 does not only check that the link is also accessible from the cells it connects, but also that it really connects distinct cells. The same checking can be done for links between source code fragments. The corresponding consistency rule is equivalent to the one shown in Listing G.2 but with **SourceCodeLink** as a context.

EPlanModelLink

Finally, for the sake of completeness, links between elements in EPlan P8 models should also only exist between distinct elements, which is ensured with the consistency rule shown in Listing G.3.

Now that we have described consistency rules that ensure correct traceability, let us move on to consistency rules for ensuring correct relations between different engineering artifacts such as EPlan P8 models and Java source code. Specifically, we want to make sure that whenever two components in an electrical model are linked, their respective representations in source code are also linked. The corresponding consistency rule, which is instantiated for every **EPlanModelLink**, is shown in Listing G.4. Indeed, this can only be checked if both of the linked model elements (i.e., the link's **in** and **out** **EPlanModelElements**) are represented by a source code fragment (i.e., the target of a **EPlan-to-Code** link). This can be ensured by checking whether both elements are part of at least one **EPlan-to-Code**

```

1 [Context: EPlanModelLink]
2 self.in <> self.out and
3 self.in.link->contains(self) and
4 self.out.link->contains(self)

```

Listing G.3: Consistency Rule for EPlanModelLink


```

1 [Context: EPlanModelLink]
2 (self.in.e2c->size()>0 and self.out.e2c->size()>0) implies
3 self.in.e2c->exists(
4   x:EPlan-to-Code|self.out.e2c->exists(
5     y:EPlan-to-Code|x.source.link->exists(
6       z:SourceCodeLink|
7       ((z.source=x.source and z.target=y.source) or
8        (z.source=y.source and z.target=x.source))))

```

Listing G.4: Advanced Semantics Consistency Rule for EPlanModelLink

Figure 15: Engineering artifact representation in the DesignSpace after step 3.

link (see line 2 of Listing G.4). If that is the case, then both linked `EPlanModelElements` must be part of `EPlan-to-Code` links to `SourceCodeFragments` that are connected by a `SourceCodeLink`. Those `EPlan-to-Source` link for the `in` and `out` `EPlanModelElements` are named `x` and `y`, respectively, in the consistency rule (see lines 4–5 of Listing G.4). The existence of an appropriate `SourceModelLink` between the source code fragments is checked in lines 6–8 of Listing G.4. Since `SourceCodeLink` should only express a connection between source code fragments without a prescribed direction but `SourceCodeLinks` have a source and a target `SourceCodeFragment`, it is necessary to check for the two possible directions (see lines 7–8 in Listing G.4). Now that we have presented how engineering artifacts are integrated, which kinds of links can be used to establish traceability between different engineering artifacts, and how this traceability can be used to define consistency rules, we next have a look at how this can be applied to the challenge problem from Section 3.

6 Application to Challenge Problem

We will now revisit the challenge problem from Section 3 and discuss how our solution is applied during the most important steps.

6.1 After Step 3

We start with Step 3. After this step, engineers have defined the initial versions of the EPlan P8 model and the corresponding source code, as depicted in Fig. 4 and Fig. 5, respectively. These two engineering artifacts have already been annotated by the engineers and the corresponding DesignSpace representations have been created automatically by the tool-adapters. Moreover, engineers have established traceability between the engineering artifacts, as shown by the presence of `EPlan-to-Code` links. Figure 15 shows the resulting DesignSpace representation. At this point, there are no inconsistencies. Notice that for the `in` and `out` elements of the `EPlanModelLink`, the corresponding source code fragments are also connected via `SourceCodeLink`. Therefore, at this stage of development all involved artifacts are consistent.

Figure 16: Engineering artifact representation in DesignSpace after step 5.

Figure 17: Engineering artifact representation in DesignSpace after step 6.

6.2 After Step 5

After Step 5, the EPlan P8 model has been changed by the electrical engineer to reflect the requests issued by the safety engineer in Step 4. The DesignSpace representation of the electrical model has been updated automatically by the EPlan P8 tool-adaptor, the result of this update is depicted in Fig. 16. At this point, the consistency rule from Listing G.4 is re-validated for the link that now connects the components 21K3 and Q30.1 (instead of Q70.0). However, since the source code has not been changed, there is no link in the source code representation that connects the corresponding source code fragments. Therefore, an inconsistency is detected and visualized right after the change of the electrical model. The information presented to the engineers shows exactly that the reason for the inconsistency is that the mapping in the source code is not conforming to the mapping in the EPlan P8 model. Moreover, the Workbench also lets the electrical engineer, who performed the change of the electrical model, identify the software engineer(s) responsible for the involved source code fragments. Of course, the software engineers will also be informed in the Workbench about the inconsistency. Therefore, not only the inconsistency is detected, but more specific information about the impact of the changed electrical model is available.

6.3 After Step 6

After reacting to the inconsistency that was detected after Step 5 by performing Step 6, the inconsistency is removed by the software engineer by changing the mapping between source code fragments. In Fig. 17, the resulting representation of engineering artifacts in the DesignSpace is depicted. Notice that the SourceCodeLink SCL1 has changed compared to Fig. 16. Specifically, it now uses as source the safety output of the Siemens PLC, hwIO_out1_1. When re-validating the consistency rule from Listing G.4 for the EPlanModelLink EML1, the consistency checker detects that now a SourceCodeLink exists that connects right source code fragments (i.e., those that represent the EPlanModelElements, which are linked by EML1). Again, the change is processed immediately by the consistency checker and the engineer is informed immediately that the change actually removed the inconsistency.

7 Discussion

Now that we have presented the solution developed in this project and its application to the challenge problem, let us revisit the goals and requirements stated in Section 4 and discuss whether these are met by the solution. Moreover, in this section, we will discuss the major challenges that were encountered during the project.

7.1 Goals

Traceability

As far as traceability is concerned, our solution provides the means to establish traceability between the integrated artifacts (i.e., spreadsheets, electrical models, and source code). The DesignSpace workbench can be used not only to establish traces of the pre-defined types, but also new types of traces with custom semantics can be defined by engineers. Using our solution, engineers at VHA can easily create and maintain traceability between different engineering artifacts as standard tasks that are added to their established development process. Engineers are expected to incrementally build traceability information by creating traces each time they make use of existing information they received from other engineers. While managers at VHA and Flanders Make are confident that this practice can be adopted by engineers, they believe that more support for engineers is still necessary.

Consistency Checking

For consistency checking, we have developed a set of consistency rules that are validated automatically and incrementally by the employed Model/Analyzer consistency checking framework. VHA believes that these consistency rules are sufficient to check electrical models and source code for consistency, which was the main goal of this project. Moreover, feedback about inconsistencies is provided instantly (or at any desired point in time) to engineers. However, more consistency rules may be added to extend the current set of consistency rules to also cover, for instance, spreadsheet data.

Change Impact Analysis

Based on traceability and consistency checking, our solution provides the necessary information for comprehensive change impact analysis. This allows engineers to quickly identify which engineering artifacts may be (or are) affected after a performed change and which engineers are responsible for those engineering artifacts. This provides important information for further analysis, especially with regards to the safety critical components of a cyber-physical system. Therefore, our solution provides the information desired by VHA. Overall, the developed solution helps VHA to realize their goals of establishing traceability and consistency checking in order to improve the efficiency of their development process.

7.2 Requirements

The requirements stated in Section 4 are met by the developed solution. In particular, the stated tools have been integrated with tool-adapters. Moreover, change impact analysis is now possible and dependencies between engineering artifacts are managed and visualized by the DesignSpace Workbench application.

7.3 Challenges and Lessons Learned

One factor that has been underestimated in the early phases of the project was the development of tool-adapters. Developing a tool-adapter for a commercial, closed-source tool

with a defined API required us not only to familiarize ourselves with that API, but it also required all involved stakeholders to analyse and understand exactly which information is actually available to plug-ins or extensions, and in which form. Only after an in-depth analysis of the available information, it is possible to successfully develop a tool-adaptor that is capable of synchronizing the relevant tool information with the DesignSpace. For example, the EPlan P8 tool-adaptor developed in this project has to automatically discover the links between two or more components that are synchronized with the DesignSpace, as it is not possible for a user of the tool to mark these links. Moreover, it can be an additional challenge to observe a tool's internal data structure for incremental changes in order to synchronize its data live with the DesignSpace (or any other cloud solution, for that matter). However, typically tools allow at least for a synchronization at some events during their typical workflow, for example, when the user saves changes. Additionally there are significant differences between tools with respect to their extensibility and the expressiveness of information available for third-party plug-ins or extensions. However, for this project we were able to access the relevant information in all three integrated tools. Moreover, it has been shown that in general commercial engineering applications can be extended appropriately [31].

Overall, the results of the project show that even the basic forms of traceability and consistency checking that our solution provides to VHA may help to improve the efficiency of a development process significantly. Engineers and managers at VHA were generally perceiving the use of the provided tools as straightforward and easy to learn. While only time can tell whether the practice of establishing and managing traces is actually performed continuously by engineers, the involved stakeholders agree that this additional task does not impose a significant amount of extended work that would severely impact their existing workflow.

8 Related Work

Indeed, the topics of traceability and consistency checking have been discussed extensively in literature and they have also been addressed by commercial tool vendors. In this section, we will discuss those approaches and tools that are closest to the solution developed in this project and highlight the differences between existing approaches and tools and the results of this project.

For consistency checking, various approaches have been proposed by academics and several commercial tools exist that perform some sort of consistency checking (e.g., [6, 7, 18, 28, 30, 67, 74, 76, 77, 94, 97]). Unfortunately, to the best of our knowledge, there is no commercial or academic tool available that supports consistency checking of EPlan P8 models, let alone support for checking consistency between EPlan P8 models and other engineering artifacts. For academic approaches, as with commercial tools, they do not provide out-of-the-box support for EPlan P8 models, meaning that an adaptation of any existing approach was necessary. We opted for employing the Model/Analyzer consistency checker [30, 72] as it has been shown that it scales well and provides instant feedback about inconsistencies, even for large-scale industrial models [73]. Moreover, the Model/Analyzer not only provides engineers with information about inconsistencies, but it also provides features that automatically, based on individual inconsistencies, provide engineers with suggestions on how to repair them [24]. Of course, there also exist other technologies that allow for automatic model repairs (e.g., [67]). However, such approaches typically select and execute repairs automatically. Discussions with VHA and Flanders Make showed that such automatically performed changes are not desired as engineers wish to have ultimate control about the engineering artifacts and do not want the artifacts to be changed by fully automated technologies. Thus, the use of the Model/Analyzer, which only informs about inconsistencies and, if desired, possible repair options, is a valid choice.

Another crucial aspect of the consistency checking approach applied in this work is the heterogeneity of engineering artifacts. The possibility to check consistency across different engineering domains has been researched by various other works (e.g. [48, 51, 93]). One major problem is the integration of type information in the platform used for consistency checking. Most approaches do not provide automated synchronization of artifacts in a uniform representation, but rely on costly meta-model merges. In line with this problem, an extension of type information with further meta-information (e.g., properties for linking) poses a problem as well. Our approach easily allows for such extensions. Artifact type information can be extended at will and may contain properties that are not available in the engineering tool the artifacts originate from. These properties can then be filled by services running on the DesignSpace (e.g., traceability links that are filled by the traceability service).

The heterogeneity of engineering artifacts plays a similarly important role in the field of systems engineering, where modeling languages such as SysML⁶ are used for the specification, analysis, design, verification and validation of large systems. These languages are supported in various tools, such as Papyrus⁷ or Eclipse PolarSys⁸. However, most system engineering tools do not support a holistic, fine-grained integration of engineering artifacts for team-oriented manipulation. This is largely due to the fact, that SysML - being a variation of UML - is mostly focused on design diagrams. Engineering artifacts representing concrete implementation (e.g., code) find little to no consideration in many SysML-related tools (with the exception of code skeleton generation). Our approach not only considers the consistency between heterogeneous engineering artifacts, but also produces immediate consistency feedback, in relation to recently adapted changes. This collaborative aspect of passing information about inconsistencies to engineers in a timely manner is absent from currently available SysML tools.

In model-driven engineering on the other hand, multiple tools and approaches partly incorporate collaborative concepts, some of which were covered in the literature review of Bruneliere et al. [9]. The main focus of these approaches is the unified consideration of different models within a singular environment (e.g., hardware design and software implementation). This is related to our own consideration of heterogeneous engineering artifacts. Similarly, models corresponding to multiple meta-models are manipulated together. Managing consistency in such an environment is a major factor in these works. EMF Views [10] Kitalpha, [58], Model Join [12] and Viatra [20] are similar to our own work in this regard. However, only Viatra recomputes consistency information in an incremental way. This is done on a change basis. The difference to our work is the application of these principles in a cloud environment with immediate feedback. Similar differences can be identified in approaches such as Vitruvius [54] and ModelJoin. While conceptually these works are similar, the resulting collaboration possibilities differ, mostly due to the lack of immediate consistency feedback.

For traceability, different approaches have been proposed in the past, mainly focusing on automatic creation and management of traces [17], and tools often support some level of traceability, for example, through change tracking (e.g., Excel). However, discussions with VHA and Flanders Make showed that automatic discovery of traces was not desired as there is always a chance of false positives that must be investigated. Therefore, we decided to rely on manually managed traces. Relying solely on different tools' traceability features was indeed not an option as none of the involved tools allowed to specify any kind of connection to engineering artifacts of other tools.

⁶SysML: <http://www.omg.sysml.org/> (last accessed: 22.06.2020)

⁷Papyrus: <https://www.eclipse.org/papyrus/> (last accessed: 22.06.2020)

⁸Eclipse PolarSys: <https://projects.eclipse.org/projects/polarsys> (last accessed: 22.06.2020)

9 Conclusions and Future Work

In this paper, we presented the results of a collaboration between Van Hoescke Automation, Flanders Make, and JKU to establish traceability, consistency checking, and impact analysis in a company that focuses on the development of automation solutions. During the project, a software solution was developed that effectively helps the company to improve its development process by providing traceability capabilities for different engineering artifacts and instant consistency checking between them. Both traceability and consistency checking allowed for a comprehensive change impact analysis. This directly influences the safety-critical aspect of the development process by supporting safety analysis. We found that one of the major challenges was the adaptation of existing technologies to support the domain-specific commercial tools used by the company. For future work, we plan to investigate possibilities of making the integration of such tools easier so that the effort for establishing traceability and consistency checking technologies can be reduced and the acceptance and application of these technologies increase.

Acknowledgements

The research reported in this article has been partly funded by the Austrian Science Fund (FWF) grant # P 31989-N31, the LIT Secure and Correct Systems Lab, LCM, Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK), the Federal Ministry for Digital and Economic Affairs (BMDW), and the Province of Upper Austria in the frame of the COMET - Competence Centers for Excellent Technologies - program managed by Austrian Research Promotion Agency FFG. We would also like to extend our gratitude to Van Hoescke Automation NV and Flanders Make for their cooperation. In particular, we would like to thank Davy Maes, from Flanders Make and Roland Kretschmer, from JKU, for their contribution to the previous work [21].

Bibliography

- [1] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning archjava. *ECOOP*, pages 334–367, 2002.
- [2] Robert Balzer. Tolerating inconsistency. In *Proceedings of the 13th International Conference on Software Engineering*, ICSE '91, pages 158–165, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.
- [3] Christian Bartelt, Georg Molter, and Tim Schumann. A model repository for collaborative modeling with the jazz development platform. In *2009 42nd Hawaii International Conference on System Sciences*, pages 1–10. IEEE, 2009.
- [4] Victor R Basili, Forrest Shull, and Filippo Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, 1999.
- [5] Miklós Biró, Atif Mashkoor, Johannes Sametinger, and Remzi Seker. Software safety and security risk mitigation in cyber-physical systems. *IEEE Software*, 35(1):24–29, 2018.
- [6] Xavier Blanc, Alix Mougenot, Isabelle Mounier, and Tom Mens. Incremental detection of model inconsistencies based on model operations. *Advanced Information Systems Engineering*, pages 32–46, 2009.
- [7] Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. Detecting model inconsistency through operation-based model construction. *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 511–520, 2008.
- [8] Barry W Boehm. Software engineering economics. *IEEE transactions on Software Engineering*, (1):4–21, 1984.
- [9] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. A feature-based survey of model view approaches. *Software & Systems Modeling*, 18(3):1931–1952, 2019.
- [10] Hugo Bruneliere, Jokin Garcia Perez, Manuel Wimmer, and Jordi Cabot. EMF views: A view mechanism for integrating heterogeneous models. In *International Conference on Conceptual Modeling*, pages 317–325. Springer, 2015.
- [11] Antonio Bucchiarone, Jordi Cabot, Richard F Paige, and Alfonso Pierantonio. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, 19(1):5–13, 2020.
- [12] Erik Burger, Jörg Henss, Martin Küster, Steffen Kruse, and Lucia Happe. View-based model-driven software development with modeljoin. *Software & Systems Modeling*, 15(2):473–496, 2016.

- [13] Hong-ming Cai, Xiao-feng Ji, and Feng-lin Bu. Research of consistency maintenance mechanism in real-time collaborative multi-view business modeling. *Journal of Shanghai Jiaotong University (Science)*, 20(1):86–92, 2015.
- [14] Giuliano Casale, Cristina Chesta, Peter Deussen, Elisabetta Di Nitto, Panagiotis Gouvas, Sotiris Koussouris, Vlado Stankovski, Andreas L Symeonidis, Vlassis Vlassiou, Anastasios Zafeiropoulos, et al. Current and future challenges of software engineering for services and applications. *Cloud Forward*, 97:34–42, 2016.
- [15] D. Cassou, E. Balland, C. Consel, and J. Lawall. Leveraging software architectures to guide and verify the development of sense/compute/control applications. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 431–440, New York, NY, USA, 2011. ACM.
- [16] Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio. Multi-view approaches for software and system modelling: a systematic literature review. *Software and Systems Modeling*, 18(6):3207–3233, 2019.
- [17] Jane Cleland-Huang, Orlena CZ Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. Software traceability: trends and future directions. *Proceedings of Future of Software Engineering*, pages 55–69, 2014.
- [18] Marcos Aurélio Almeida da Silva, Alix Mougenot, Xavier Blanc, and Reda Bendraou. Towards automated inconsistency handling in design models. *International Conference on Advanced Information Systems Engineering*, pages 348–362, 2010.
- [19] István Dávid, Joachim Denil, and Hans Vangheluwe. Process-oriented inconsistency management in collaborative systems modeling. In *16th annual Industrial Simulation Conference, ISC'2018, June 6-8, 2018, Ponta Delgada, Portugal/Machado, José [edit.]; et al.*, pages 54–61, 2018.
- [20] Csaba Debrececi, Ákos Horváth, Ábel Hegedüs, Zoltán Ujhelyi, István Ráth, and Dániel Varró. Query-driven incremental synchronization of view models. In *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, pages 31–38, 2014.
- [21] Andreas Demuth, Roland Kretschmer, Alexander Egyed, and Davy Maes. Introducing traceability and consistency checking for change impact analysis across engineering tools in an automation solution company: An experience report. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 529–538. IEEE, 2016.
- [22] Andreas Demuth, Roland Kretschmer, Alexander Egyed, and Davy Maes. Introducing traceability and consistency checking for change impact analysis across engineering tools in an automation solution company: an experience report. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 529–538. IEEE, 2016.
- [23] Andreas Demuth, Markus Riedl-Ehrenleitner, and Alexander Egyed. Efficient detection of inconsistencies in a multi-developer engineering environment. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, pages 590–601, New York, NY, USA, 2016. ACM.
- [24] Andreas Demuth, Markus Riedl-Ehrenleitner, Roberto E. Lopez-Herrejon, and Alexander Egyed. Co-evolution of metamodels and models through consistent change propagation. *Journal of Systems and Software*, 111:281–297, 2016.

-
- [25] Andreas Demuth, Markus Riedl-Ehrenleitner, Alexander Nöhrer, Peter Hehenberger, Klaus Zeman, and Alexander Egyed. Designspace: an infrastructure for multi-user/multi-tool engineering. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1486–1491. ACM, 2015.
 - [26] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Collaborative repositories in model-driven engineering [software technology]. *IEEE Software*, 32(3):28–34, 2015.
 - [27] Peter K Dunn, Michael D Carey, Michael B Farrar, Alice M Richardson, and Christine McDonald. Introductory statistics textbooks and the gaise recommendations. *The American Statistician*, 71(4):326–335, 2017.
 - [28] S. Easterbrook and B. Nuseibeh. Using viewpoints for inconsistency management. *Software Engineering Journal*, 11(1):31–43, 1996.
 - [29] A. Egyed. Automatically detecting and tracking inconsistencies in software design models. *IEEE Transactions on Software Engineering*, 37(2):188–204, 2011.
 - [30] Alexander Egyed. Instant consistency checking for the UML. *Proceedings of the 28th international conference on Software engineering*, pages 381–390, 2006.
 - [31] Alexander Egyed and Robert Balzer. Integrating cots software into systems through instrumentation and reasoning. *Autom. Softw. Eng.*, 13(1):41–64, 2006.
 - [32] Alexander Egyed, Klaus Zeman, Peter Hehenberger, and Andreas Demuth. Maintaining consistency across engineering artifacts. *Computer*, 51(2):28–35, 2018.
 - [33] Michael Fagan. *Design and Code Inspections to Reduce Errors in Program Development*, pages 575–607. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.
 - [34] Muhammad Fahad, Nejib Moalla, and Abdelaziz Bouras. Towards ensuring satisfiability of merged ontology. *Procedia Computer Science*, 4:2216–2225, 2011.
 - [35] Robert Feldt and Ana Magazinius. Validity threats in empirical software engineering research-an initial survey. In *Seke*, pages 374–379, 2010.
 - [36] A. Finkelstein. A foolish consistency: Technical challenges in consistency management. In Mohamed Ibrahim, Josef Küng, and Norman Revell, editors, *Database and Expert Systems Applications*, pages 1–5. Springer Berlin Heidelberg, 2000.
 - [37] Anthony CW Finkelstein, Dov Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.
 - [38] Pascal Fradet, Daniel Le Métayer, and Michaël Périn. Consistency checking for multiple view software architectures. In *Software Engineering - ESEC/FSE’99*, pages 410–428. Springer, 1999.
 - [39] Chiara Di Francescomarino, Chiara Ghidini, Marco Rospocher, Luciano Serafini, and Paolo Tonella. A framework for the collaborative specification of semantically annotated business processes. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(4):261–295, 2011.
 - [40] Mirco Franzago, Davide Di Ruscio, Ivano Malavolta, and Henry Muccini. Collaborative model-driven software engineering: a classification framework and a research map. *IEEE Transactions on Software Engineering*, 44(12):1146–1175, 2018.

- [41] Mirco Franzago, Henry Muccini, and Ivano Malavolta. Towards a collaborative framework for the design and development of data-intensive mobile applications. In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*, pages 58–61. ACM, 2014.
- [42] Jesús Gallardo, Crescencio Bravo, and Miguel A Redondo. A model-driven development method for collaborative modeling tools. *Journal of Network and Computer Applications*, 35(3):1086–1105, 2012.
- [43] Alfonso García Frey, Jean-Sébastien Sottet, and Alain Vagner. Ame: an adaptive modelling environment as a collaborative modelling tool. In *Proceedings of the 2014 ACM SIGCHI symposium on Engineering interactive computing systems*, pages 189–192. ACM, 2014.
- [44] M. Gardner. The fantastic combinations of john conway’s new solitaire game ”life”. *Scientific American*, 223:120–123, 1970.
- [45] P GE. Box, wg hunter, and js hunter. statistics for experimenters, 2005.
- [46] A. Ghabi and A. Egyed. Exploiting traceability uncertainty between architectural models and code. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pages 171–180, Aug 2012.
- [47] Marilyn Healy and Chad Perry. Comprehensive criteria to judge validity and reliability of qualitative research within the realism paradigm. *Qualitative market research: An international journal*, 2000.
- [48] Sebastian JI Herzig, Ahsan Qamar, and Christiaan JJ Paredis. An approach to identifying inconsistencies in model-based systems engineering. *Procedia Computer Science*, 28:354–362, 2014.
- [49] Byron Jones and Michael G Kenward. *Design and analysis of cross-over trials*. CRC press, 2014.
- [50] Georgios Kanakis, Djamel Eddine Khelladi, Stefan Fischer, Michael Tröls, and Alexander Egyed. An empirical study on the impact of inconsistency feedback during model and code co-changing. 2019.
- [51] Harald König and Zinovy Diskin. Advanced local checking of global consistency in heterogeneous multimodeling. In *European Conference on Modelling Foundations and Applications*, pages 19–35. Springer, 2016.
- [52] Philip Koopman and Michael Wagner. Autonomous vehicle safety: An interdisciplinary challenge. *IEEE Intelligent Transportation Systems Magazine*, 9(1):90–96, 2017.
- [53] Konstantinos Kotis, George A. Vouros, and Konstantinos Stergiou. Towards automatic merging of domain ontologies: The hccone-merge approach. *J. Web Sem.*, 4(1):60–79, 2006.
- [54] Max E Kramer, Erik Burger, and Michael Langhammer. View-centric engineering with synchronized heterogeneous models. In *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, pages 1–6, 2013.

-
- [55] Roland Kretschmer, Djamel Eddine Khelladi, and Alexander Egyed. An automated and instant discovery of concrete repairs for model inconsistencies. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, pages 298–299. ACM, 2018.
 - [56] Mayuram S. Krishnan and Marc I Kellner. Measuring process consistency: Implications for reducing software defects. *IEEE Transactions on Software Engineering*, 25(6):800–815, 1999.
 - [57] Stephan Krusche and Bernd Bruegge. Model-based real-time synchronization. In *International Workshop on Comparison and Versioning of Software Models (CVSM’14)*, 2014.
 - [58] Benoît Langlois, Daniel Exertier, and Boubekeur Zendagui. Development of modelling frameworks and viewpoints with kitalpha. In *Proceedings of the 14th Workshop on Domain-Specific Modeling*, pages 19–22, 2014.
 - [59] Per Lenberg, Robert Feldt, and Lars Göran Wallgren. Human factors related challenges in software engineering—an industrial perspective. In *2015 ieee/acm 8th international workshop on cooperative and human aspects of software engineering*, pages 43–49. IEEE, 2015.
 - [60] Patrick Mäder and Alexander Egyed. Assessing the effect of requirements traceability for software maintenance. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 171–180. IEEE, 2012.
 - [61] Patrick Mäder and Alexander Egyed. Do developers benefit from requirements traceability when evolving and maintaining a software system? *Empirical Software Engineering*, 20(2):413–441, 2015.
 - [62] Carsten Marohn, Pepijn Schreinemachers, Dang Viet Quang, Thomas Berger, Prakrit Siripalangkanont, Thanh Thi Nguyen, and Georg Cadisch. A software coupling approach to assess low-cost soil conservation strategies for highland agriculture in vietnam. *Environmental Modelling & Software*, 45:116–128, 2013.
 - [63] Atif Mashkoor, Miklós Biró, Richard Messnarz, and Ricardo Colomo Palacios. Selected functional safety and cybersecurity concerns in system, software, and service process improvement and innovation. *J. Softw. Evol. Process.*, 30(5), 2018.
 - [64] Atif Mashkoor, Johannes Sameting, Miklós Biro, and Alexander Egyed. Security- and safety-critical cyber-physical systems. *Journal of Software: Evolution and Process*, 32(2):e2239, 2020.
 - [65] Madanlal Musuvathi, David YW Park, Andy Chou, Dawson R Engler, and David L Dill. CMC: A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review*, 36(SI):75–88, 2002.
 - [66] Christian Nentwich, Licia Capra, Wolfgang Emmerich, and Anthony Finkelstein. xlinkit: A consistency checking and smart link generation service. *ACM Transactions on Internet Technology (TOIT)*, 2(2):151–185, 2002.
 - [67] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Consistency management with repair actions. *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 455–464, 2003.
 - [68] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelstein, and Ernst Ellmer. Flexible consistency checking. *ACM Trans. Softw. Eng. Methodol.*, 12(1):28–63, 2003.

- [69] J. Nielsen. Usability engineering. 1993.
- [70] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Making inconsistency respectable in software development. *Journal of systems and software*, 58(2):171–180, 2001.
- [71] Claudia Pagliari. Design and evaluation in ehealth: challenges and implications for an interdisciplinary field. *Journal of medical Internet research*, 9(2):e15, 2007.
- [72] Alexander Reder and Alexander Egyed. 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*, pages 347–348. ACM, 2010.
- [73] Alexander Reder and Alexander Egyed. Incremental consistency checking for complex design rules and larger model changes. *International Conference on Model Driven Engineering Languages and Systems*, pages 202–218, 2012.
- [74] Steven P Reiss. Incremental maintenance of software artifacts. *IEEE Transactions on Software Engineering*, 32(9):682–697, 2006.
- [75] M. Riedl-Ehrenleitner. Model-and-code consistency checking. Master’s thesis, Johannes Kepler University, 2013.
- [76] Markus Riedl-Ehrenleitner, Andreas Demuth, and Alexander Egyed. Towards model-and-code consistency checking. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 85–90. IEEE, 2014.
- [77] Mehrdad Sabetzadeh, Shiva Nejati, Steve Easterbrook, and Marsha Chechik. Global consistency checking of distributed models with TReMer+. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 815–818. IEEE, 2008.
- [78] Vivek Sakhrani, Paul S Chinowsky, and John E Taylor. Grand challenges in engineering project organization. *Engineering Project Organization Journal*, 7(1):15, 2017.
- [79] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. Are students representatives of professionals in software engineering experiments? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 666–676. IEEE, 2015.
- [80] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.
- [81] George W Snedecor and Witiiam G Cochran. Statistical methods, 8thedn. *Ames: Iowa State Univ. Press Iowa*, 54:71–82, 1989.
- [82] George Spanoudakis and Andrea Zisman. Inconsistency management in software engineering: Survey and open research issues. In *Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals*, pages 329–380. World Scientific, 2001.
- [83] Mikael Svahnberg, Aybüke Aurum, and Claes Wohlin. Using students as subjects-an empirical evaluation. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 288–290, 2008.
- [84] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, 2002.

-
- [85] R Core Team et al. R: A language and environment for statistical computing, 2013.
 - [86] Damiano Torre, Yvan Labiche, and Marcela Genero. UML consistency rules: a systematic mapping study. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 6. ACM, 2014.
 - [87] Michael Alexander Tröls, Atif Mashkoor, Andreas Demuth, and Alexander Egyed. Ensuring safe and consistent coengineering of cyber-physical production systems: A case study. *Journal Of Software: Evolution And Process*, 2020.
 - [88] Michael Alexander Tröls, Atif Mashkoor, and Alexander Egyed. Collaboratively enhanced consistency checking in a cloud-based engineering environment. In *The 11th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '19)*. ACM, 2019.
 - [89] Michael Alexander Tröls, Atif Mashkoor, and Alexander Egyed. Live and global consistency checking in a collaborative engineering environment. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19)*, pages 1762 – 1771. ACM, 2019.
 - [90] Michael Alexander Tröls, Atif Mashkoor, and Alexander Egyed. Multifaceted consistency checking of collaborative engineering artifacts. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 278–287. IEEE, 2019.
 - [91] N. Ubayashi, J. Nomura, and T. Tamai. Archface: A contract place where architectural design and code meet together. *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 1:75–84, 01 2010.
 - [92] Muhammad Usman, Aamer Nadeem, Tai-hoon Kim, and Eun-suk Cho. A survey of consistency checking techniques for UML models. In *2008 Advanced Software Engineering and Its Applications*, pages 57–62. IEEE, 2008.
 - [93] Hans LM Vangheluwe. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *Cacsd. conference proceedings. IEEE international symposium on computer-aided control system design (cat. no. 00th8537)*, pages 129–134. IEEE, 2000.
 - [94] Michael Vierhauser, Paul Grünbacher, Alexander Egyed, Rick Rabiser, and Wolfgang Heider. Flexible and scalable consistency checking on product line variability models. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 63–72. ACM, 2010.
 - [95] J Christopher Westland. The cost of errors in software development: evidence from industry. *Journal of Systems and Software*, 62(1):1–9, 2002.
 - [96] Dustin Wüest, Norbert Seyff, and Martin Glinz. Flexisketchteam: collaborative sketching and notation creation on the fly. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 685–688. IEEE Press, 2015.
 - [97] Chang Xu, Shing-Chi Cheung, and Wing-Kwong Chan. Incremental consistency checking for pervasive context. *Proceedings of the 28th international conference on Software engineering*, pages 292–301, 2006.
 - [98] Michael Zhivich and Robert K Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2):87–90, 2009.