Efficient Detection of Inconsistencies in a Multi-Developer Engineering Environment

Andreas Demuth
Institute for Software Systems Engineering
Johannes Kepler University
Linz, Austria
andreas.demuth@jku.at

Markus Riedl-Ehrenleitner
Institute for Software Systems Engineering
Johannes Kepler University
Linz, Austria
markus.riedl@jku.at

Alexander Egyed
Institute for Software Systems Engineering
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

ABSTRACT

Software developers work concurrently on different kinds of development artifacts such as requirements, architecture, design, or source code. To keep these development artifacts consistent, developers have a wide range of consistency checking approaches available. However, most existing consistency checkers work best in context of single tools and they are not well suited when development artifacts are distributed among different tools and are being modified concurrently by many developers.

This paper presents a novel, cloud-based approach to consistency checking in a multi-developer/-tool engineering environment. It allows instant consistency checking even if developers and their tools are distributed and hence do not have access to all artifacts. It systematically reuses consistency checking knowledge to keep the memory/CPU cost of consistency checking to a small constant overhead per developer. The feasibility and scalability of our approach is demonstrated by a prototype implementation and through an empirical validation with 22 partly industrial system models.

1. INTRODUCTION

Software engineering is an inherently collaborative discipline with developers working concurrently on a wide range of *development artifacts*—requirements, use cases, design, code, and more. To modify these development artifacts, an equally diverse engineering tool landscape exists—each tool usually specializing on specific kinds of development artifacts (e.g., Eclipse [1] for source code, IBM Rational Software Architect [2] for UML models). Each tool thus only presents a partial view of a software system [3]. After all, developers often do not even need access to all engineering artifacts [4, 5, 6, 7].

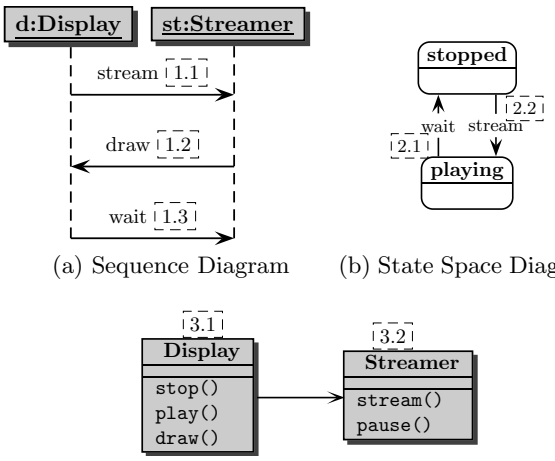
The software engineering landscape is characterized by the distributed and concurrent modification of development artifacts by many developers. This follows a familiar pattern: Typically, developers download development artifacts to their local workstations (i.e., checkout) and modify them there in private before finally uploading the changes to a shared repository for others to see (i.e., commit). In doing so, developers create a local environment where they modify development artifacts separately from the shared repository and where they only have access to a subset of all development artifacts—those that can be modified by the tools they use. For example, the designer will use a modeling tool to modify the models while the programmer will use a programming tool to modify the source code.

Inconsistencies arise if development artifacts contradict. It is easy to see that inconsistencies are particularly hard to spot in situations where different developers modify development artifacts privately within different tools. Here existing work for consistency checking is lacking and efficient means to manage consistency is needed [7]. While many consistency checking approaches exist today (e.g., [3, 6, 8, 9, 10, 11, 12, 13, 14]) they do not focus on the multi-developer/-tool problem. *These approaches either require complete local access to all development artifacts or they do not differentiate between private and public knowledge during consistency checking, which is contradictory to the situation of developers working in private.*

For example, it is hard for a designer working on a checked-out design model to understand the consistency implications of private changes with regard to source code that is not locally available. We refer to the checked out development artifacts as private working copies and there are as many such copies as there are developers/tools working concurrently—each having a partial view only and each reflecting changes that developers made in private. Thus, today it is not possible that the designer receive feedback about inconsistencies between model and source code, if the latter is not available locally.

In this paper, we introduce a novel approach for consistency checking in a multi-developer, multi-tool engineering environment. Our approach relies on a cloud infrastructure to maintain i) a central, public area that contains the entirety of development artifacts shared by all developers and ii) as many private working areas as there are developers to reflect their individual, not yet published modifications. Changes that developers perform locally in their tools are instantly propagated to their respective private working ar-

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235



(a) Sequence Diagram (b) State Space Diagram

(c) Class Diagram

Figure 1: Current Public Version

eas in the cloud. These changes are then instantly checkable against the public area, therefore enabling a complete consistency checking.

The performance savings of our approach are the result of systematic reuse of consistency checking knowledge. Our approach does not have any expensive initialization costs and its benefit increases with the number of developers working concurrently. To validate our approach we analyzed its computational and memory complexity. Furthermore, to show its feasibility, we developed a prototype and conducted an empirical evaluation to assess the scalability of our approach. Finally, we discuss three partly industrial case studies in which we applied our infrastructure.

2. PROBLEM ILLUSTRATION AND BACKGROUND

This work builds on the Model/Analyzer [15, 10] which is a single tool/single developer consistency checker. Like most other consistency checking technologies [3, 11, 16, 17] it cannot provide a holistic consistency check across all development artifacts when a developer only has one development artifact available (e.g., source code or model) To illustrate this, we discuss a model of a *video on demand (VOD)* system [**?**] that is concurrently modified by two developers: Alice and Bob. Note that when referring to elements of the example we will use the syntax `[UML::<type>]<name>`, where `<type>` is a placeholder for the specific UML type and `<name>` for its name.

2.1 Initial State

Figure 1 shows the complete, public model of the VOD system that both Bob and Alice are able to see. The UML model consists of three diagrams: a class diagram showing the entities `Display` and `Streamer`, a sequence diagram outlining the interaction among those two classes, and a state chart depicting the state space of `Streamer`. We limit our illustration to UML for simplicity. However, the public model could also include requirements, use cases, source code, and other development artifacts.

To express desired conditions that a model must satisfy consistency rules are used. Below we show three examples of consistency rules, written in *OCL* [18], that apply to UML models: `CR1`, `CR2`, and `CR3` in Listings 1, 2, and 3.

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

Listing 1: (CR1) Message must be defined as an operation in the receiver's class.

```
UML::Transition t
let classifier:BehavioredClassifier=
  self.owner.oclAsType(Region).stateMachine.
      context in
classifier.oclIsTypeOf(Class) implies
  classifier.oclAsType(Class).ownedOperation
    ->exists(o:Operation|o.name=self.name)
```

Listing 2: (CR2) Action of statechart must be an operation in the owner's class.

```
UML::Class c
-- all parents of UML Class c
let allParents : Set(UML::Classifier) in
not allParents->includes(c)
```

Listing 3: (CR3) No circular inheritance.

Each consistency rule is written for a specific *context* and the rule must be evaluated for each instance of this context. Since we focus on UML in this illustration, the context is a specific UML element. CR3 with the context `UML::Class` checks whether a given `UML::Class` is part of a circular inheritance. For CR3 in the class diagram in Fig. 1 we find two `UML::Class` instances (`[UML::Class]Display` and `[UML::Class]Streamer`) and thus two evaluations of CR3 are necessary—one for each class. We will refer to such an evaluation as *consistency rule instance (CRI)* throughout the rest of the paper. Similarly, CR1 and CR2 check whether messages in sequence diagrams and transitions in statecharts have names that correspond to operations in the class diagrams (i.e., a message action must have an equally named operation in a class).

Figure 1 depicts the entire UML model, and thus any consistency checker applicable to UML models would be able to determine its consistency with regard to the three consistency rules. For example, the Model/Analyzer would first identify the CRIs needed by searching for all UML model instances of the context elements and instantiating a CRI for every instance found. For the VOD example, seven CRIs are needed:

- three instantiations of CR1 corresponding to the three messages in the sequence diagram: `CRI.1.1` for `[UML::Message]stream`, `CRI.1.2` for `[UML::Message]draw` and `CRI.1.3` for `[UML::Message]wait`;
- two instantiations of CR2 corresponding to the two transitions in the statechart diagram: `CRI.2.1` for the transition `[UML::Transition]wait` and `CRI.2.2` for `[UML::Transition]stream`; and
- two instantiations of CR3 corresponding to the two classes in the class diagram: `CRI.3.1` for `[UML::Class]Display` and `CRI.3.2` for `[UML::Class]Streamer`.

Except for CRI.2.1 and CRI.1.3, all instances are consistent. CRI.2.1 is inconsistent because there exists no operation in the class diagram for [UML::Transition]wait in the statechart diagram. Similarly, CRI.1.3 is inconsistent because there exists no operation for the "wait" message in the sequence diagrams.

2.2 Multi-Developer Consistency Checking

Consider now that Alice and Bob modify the UML model separately. Each developer checks out the model to his or her local workstation that runs a modeling tool and a consistency checker. Normally, the public model would not only have a UML model but also other kinds of development artifacts such as requirements, use cases. In analogy, let us assume that there are two modeling tools involved and Alice uses a modeling tool that can only modify class and state chart diagrams whereas Bob uses a modeling tool that can modify all three kinds of diagrams. Thus, Alice—and her consistency checker—only has partial knowledge available. Note that in the following, we append the starting letter of the developer's name to the consistency rule instances to distinguish them (e.g., CRI.1.3.B is Bob's private consistency rule instance of CRI.1.3).

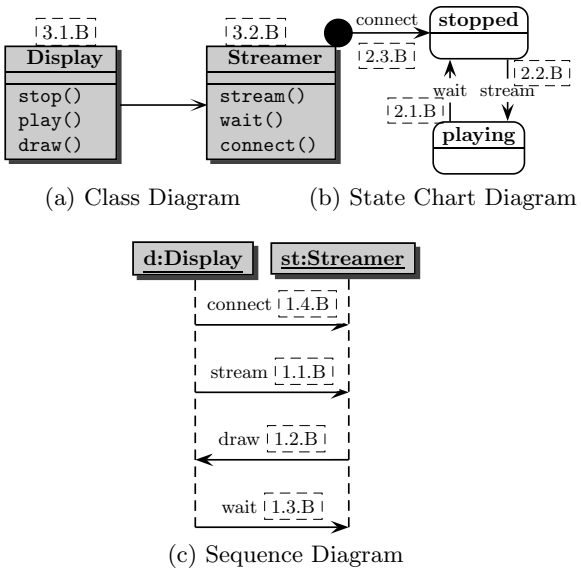
2.2.1 Adaptations by Bob

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

- an operation called "connect" is added to the [UML::Class]Streamer in the class diagram,
- a start state and transition is added in the state chart diagram, and
- a connect message is added to the sequence diagram.

Furthermore, Bob decides to resolve the two inconsistencies CRI.1.3 and CRI.2.1 by renaming the [UML::Operation]pause in the class diagram to "wait". The new state of Bob's working copy is depicted in Fig. 2. These changes alter the state of the UML model and thus have implications on the CRIs. Incremental consistency checkers, such as the Model/Analyzer, are able to react to changes in a fine-grained manner without re-evaluating the entirety of the model. The Model/Analyzer defines the concept of a *scope* which is a set of elements that, if changed, should trigger the re-evaluation of CRIs. This set is created at the first evaluation of the CRI and is simply the list of model elements accessed during the evaluation of the CRI on the UML model. Only changes to these accessed elements can cause the CRI state to change. A CRI is thus re-evaluated if an element in its scope changes—details are discussed by Egyed [**?**]. Please note that most incremental consistency checker have similar concepts (e.g., *critical node* [19] *impact matrix* [16]).

The first change triggers a re-evaluation of CRI.1.1.B and CRI.1.3.B because a new operation was added to the class [UML::Class]Streamer, which may affect their results. However, the re-evaluation did not cause the CRIs to change their state (i.e., change from consistent to inconsistent or vice versa). The CRIs CRI.2.1.B and CRI.2.2.B need to be re-evaluated as well. Furthermore, the second and third change each require a new CRI, CRI.2.3.B and CRI.1.4.B



(a) Class Diagram (b) State Chart Diagram

(c) Sequence Diagram

Figure 2: Bob's Private Version

respectively because they introduce new UML model elements whose types match the context of a consistency rule (CR2 and CR1, respectively). This consistency check is indeed necessary. Finally, the renaming of [UML::Operation]pause to "wait" in the class diagram affects nearly all instances of the interaction diagram (except 1.2.B) and all instances of the state chart, amounting to 6 CRI evaluations in total. This is because the changed name could affect any defined transition in the state chart or any operation called on [UML::Lifeline]Streamer in the interaction diagram. Indeed, this change does resolve the inconsistency CRI.1.3.B. In fact, all inconsistencies are resolved at this point and a new feature (connect) is added.

2.2.2 Adaptations by Alice

Alice also wants to resolve the inconsistency between the class diagram and the state chart. Unaware of Bob's work she does so in a different way: by renaming the [UML::Transition]wait to "pause". The state of her working copy is depicted in Fig. 3. This change resolves the inconsistency and requires the re-evaluation of CRI.2.1.A. However, recall that Alice's tool does not have available the entire UML model. This demonstrates a situation where a tool and its corresponding consistency checker does not have access to all development artifacts—which is the norm [4, 7]. Her local consistency checker finds that she indeed removed the inconsistency with the state chart. However, her local consistency checker cannot know about the inconsistency with the interaction diagram, which remains (i.e., there is still no [UML::Operation]wait for the [UML::Class]Streamer).

3. PROBLEM STATEMENT

The Model/Analyzer, like other existing consistency checking approaches, suffers from incomplete knowledge in private working copies. The following issues exist:

- **Incomplete information.** Approaches such as the Model/Analyzer [15, **?**] or CLIME [3] are capable of

checking private working copies efficiently and incrementally. However, they require all development artifacts to be available locally, which is not often the case in multi-developer, multi-tool development scenarios and should also not be the case just for the sake of complete consistency checking and separation of concerns.

- **No support for private working copies.** Approaches that allow for incomplete local knowledge (e.g., [20, 6, 8, 9, 21]) do not support the idea of private adaptations (i.e., every change a developer performs is immediately considered public). This limits the applicability of those approaches because not every developer would like changes to be publicly visible immediately (i.e., trial and error).

4. GOAL

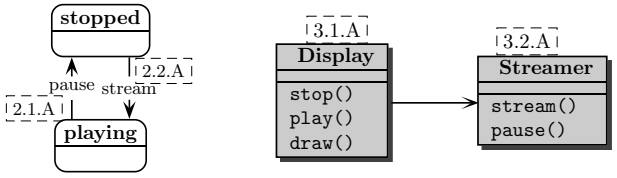
To address the problems described in Section 3, our goal is to develop an infrastructure that is capable of checking the impact of changes performed by many concurrently working developers in private (i.e., many private work areas) with respect to publicly available knowledge (i.e., one shared, public area). The infrastructure's computational effort and memory consumption should be scalable and it should not replicate all development artifacts to all private working areas.

5. APPROACH

Our approach to scalable consistency checking in a multi-developer, multi-tool environment combines the advantages of version control systems, private working areas, and incremental consistency checkers. Our solution systematically reuses computed consistency checking knowledge, and provides complete consistency checking for all developers at all times.

5.1 Overview

Figure 4 depicts an overview of our approach. The cloud-based infrastructure is depicted on the right-hand side of Fig. 4. The infrastructure maintains a version history of all public development artifacts and their corresponding consistency data (`public area`). The CRIs are thus persisted alongside the development artifacts, which is a key factor of our approach. The version history is fine-granular, similar to the version control of *Resource Description Framework(RDF)* [22] or operation-based version control of EMF [23]. To represent arbitrary development artifacts, we use a unified representation, which we will refer to simply as the *model*. In this unified representation individual elements of development artifacts and their properties are subject to version control [24].



(a) State Chart Diagram (b) Class Diagram

Figure 3: Alice's Private Version

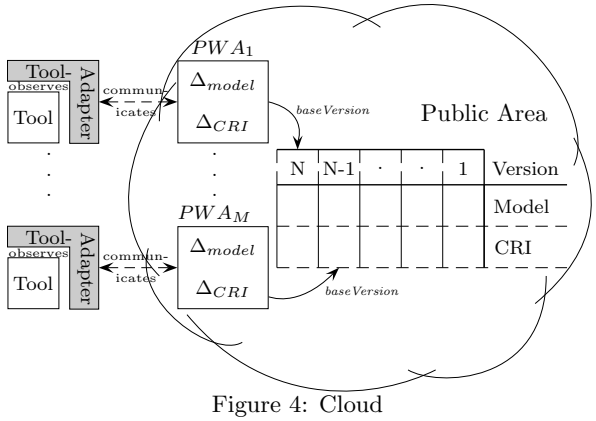


Figure 4: Cloud

The left-hand side of Fig. 4 depicts development tools used by developers (rectangles labeled as `Tool`).[1] The cloud maintains *private work areas (pwa)* (`PWA`$_1$ to `PWA`$_M$) to capture the differences between the development artifacts in the tools and the shared, public development artifacts in the cloud. It is important to note that we do not expect development tools to run within the cloud as developers typically use them on their local workstations. However, PWAs do reside in the cloud together with the public area. There are as many PWAs as there are development tools used by developers. Each PWA reflects what its developer has changed in its respective tool as compared to the public area. This is achieved by the means of `tool-adapter`s, depicted as grey colored polygons that integrate the various tools with the cloud-based infrastructure[2]. Tool-adapters support a typical SVN [28]-style workflow (i.e., check-out, update, commit) and, most importantly, they communicate development artifact changes made by developers in tools to their respective PWAs in the cloud to ensure that the PWAs are always up-to-date.

A tool is typically only concerned with the development artifacts that it can edit (e.g., a modeling tool may not edit source code). Yet, to perform a complete consistency check, a consistency checker needs access to all development artifacts. While PWAs reflect the ongoing works of their corresponding developers/tools. Placing a PWA in the cloud provides it with access to all public, version controlled development artifacts. Therefore, by extension a consistency checker operating on the PWAs in the cloud has access to all development artifacts—private and public. This satisfies the basic requirement for complete consistency checking. However, this alone does not guarantee scalability. Simply executing the Model/Analyzer—or any other consistency checker—in PWAs in the cloud would lead to unnecessary CRI evaluations and unnecessary memory consumption because each PWA might replicate same/similar consistency information. Consider the problem illustration again where after obtaining the model, both private working areas of Alice and Bob have available local copies of their CRIs to reflect their private state of consistency. For example, neither

[1]As proof of concept, we already support a wide range of such tools such as Eclipse for source code [1], IBM RSA [2], Microsoft Excel and Visio, Eplan [25], Creo Elements Pro [26], generated ECore Editors.

[2]For more information on tool-adapters refer to Demuth et al. [27].

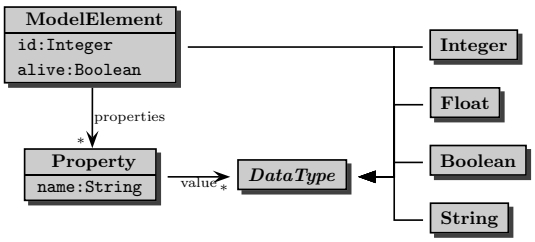


Figure 5: Model Element

Bob's nor Alice's changes affected the third consistency rule (circular inheritance). Thus, `CRI.3.1.A/B` and `CRI.3.2.A/B` were initially needlessly evaluated by both Alice's and Bob's consistency checker.

To address this issue, each PWA not only maintains the Δ of the model with respect to the base version (i.e., `pwa.`Δ`model`) but also its corresponding consistency Δ (i.e., the added/modified/removed CRIs stored in `pwa.`Δ`CRI`). Whenever a developer modifies a development artifact, the consistency checker re-evaluates the affected CRIs. When CRIs change then they are stored as CRI deltas in the PWA, which constitutes the difference between the tool's consistency state and public version's consistency state. The version controlling of consistency data also avoids expensive, initial batch consistency checking as the CRI can be checked-out together with the artifacts.

5.2 Unified Representation

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

A model element has a unique `id`. Furthermore, the boolean flag `alive` indicates whether the model element is alive or not (i.e., whether it has been deleted or not). Moreover, model elements may have a variable number of `properties`, which are key-value pairs. Values of properties are either of basic data types such as `Boolean`, `Integer`, `Float`, and `String` or references to other model elements (notice that the inheritance of `DataType` extends to `ModelElement`). Any tool that stores its development artifact in a graph-like structure of nodes and edges can be mapped to such a simple, unified representation (e.g.,ECore [29], ASTs). The semantics do not change.

5.3 Check-Out

To follow the illustrative example, Alice at first needs to obtain (i.e., check-out) the class and state-chart diagrams. Alice thus uses the check-out function in her modeling tool—provided by the tool-adapter—where she specifies the model and version she wants to check out. Listing 4 describes the check-out algorithm.

```
checkOut(baseVersion, model)
  pwa = create PrivateWorkingArea(baseVersion)
  return pwa, getModelElements(model, baseVersion)
```

Listing 4: CheckOut

The algorithm first creates a PWA for her tool, which references a version of the public area as base version. The PWA's Δ of the model is empty upon check-out because Alice has not have made any changes yet (`pwa.`Δ`model`. Likewise, `pwa.`Δ`CRI` is also empty. The algorithm returns a tuple of the PWA and the checked-out model elements to the tool adapter. The tool adapter then translates the model elements to her tool's internal language—which is tool-specific and therefore not depicted.

5.4 Changes

After checking out a model, developers may modify it within their development tool. Changes made by a developer are observed by the tool adapter and propagated to the developer/tool's corresponding PWA as a change of model elements. There the changes are stored as deltas with regard to the base version. Less straightforward is computing the consistency implications of these changes. Affected CRIs have to be re-evaluated and the new state needs also be stored in the PWA. However, this process varies with regard to the kind of changes: `modification`, `addition`, and `deletion`.

5.4.1 Modification

Recall that Alice renames the transition from "wait" to "pause". She does this in her tool and the tool adapter will update Alice's `pwa.`Δ`model` with the modified name. Listing 5 describes the algorithm to handle modifications of model element properties in the PWA. The changed model element property is added in line 2 as a model delta. The syntax of this line is to be understood as follows: `pwa.`Δ`model` holds the Δ for all model elements. The Δ for an individual model element with identifier i can be accessed through `pwa.`Δ`model[`i`]`. Recall, that each model element is identified by a unique id, which is utilized for this access. Further, the Δ of individual properties of model elements can be accessed by `pwa.`Δ`model[`i`].[`p`]`, p again being the identifier for the property In this case, the name of a property is the unique identifier for the access of property Δs.

```
modify(pwa, modelElement, property, oldValue,
    newValue)
  pwa.Δmodel[modelElement].[property] = newValue
  CRIs = {CRI ∈ pa[baseVersion].CRI |
       CRI.alive == true and CRI ∉ pwa.ΔCRI}
  CRIs ∪= pwa.ΔCRI
  for CRI ∈ CRIs
    if (modelElement, property) ∈ CRI.scope
      pwa.ΔCRI[CRI] = evaluate(CRI)
```

Listing 5: Modification of a Model Element

An example of such an access could be `pwa.`Δ`[[UML::Transition]wait].[name]`. This model element/property does not exist in the PWA before the modification and is set to the value "pause"—thus overwriting the value "wait" from the same model element/property found in the public model. Each model element/property may have at most one entry in `pwa.`Δ`model`. Should Alice later overwrite this name again, then the newer name would overwrite the previous change in the PWA.

For each modification, all CRIs affected by the modification must be re-evaluated and the result must be persisted in `pwa.`Δ`CRI`. Similar to model elements for tool-adapters, a consistency checker has an internal language for CRIs. Thus, CRIs are similarly translated into the unified represen-

tation and also uniquely identified and accessible as model elements. Since these CRIs are as well subject to version control, we must distinguish two cases: CRIs that already are in pwa.ΔCRI and CRIs that are in the public area. The variable CRIs is the union of the private CRIs (line 5) and the public CRIs (lines 3–4). Please note that in line 3 a new syntax is introduced to access the public area (pa). Therefore, pa[version] accesses a specific version of the public area. Computing the private CRIs is straightforward because it is pwa.ΔCRI itself. Computing the public CRIs is more complex for two reasons: we must not include any public CRIs that 1) have been deleted (i.e., certain changes require CRIs to be deleted in pwa.ΔCRI) and 2) have newer counterparts in the PWA (i.e., changes may require CRIs to be re-evaluate the new result of an evaluation must then overwrite their public counterparts).

Once collected, we need to re-evaluate those CRIs whose scopes contain the changed model element property (lines 6-8). The new evaluation result is then stored in pwa.ΔCRI (line 8)—which either overwrites a previous private result or, if there was none, creating a new entry and therefore effectively overwriting the public result. This explains the need to filter the overwritten public CRIs as discussed above. In case of Alice's change, pwa.ΔCRI was initially empty. Of the seven CRIs that existed in the public area during check-out (Fig. 1), only CRI.2.1 had a scope that included the tuple ([UML::Transition]wait, name). Therefore, CRI.2.1 needs to be re-evaluated, resulting in the private CRI.2.1.A stored in her PWA. This private CRI exists for Alice only and overwrites the public CRI.2.1. The public CRI.2.1 cannot be replaced physically because it corresponds to the public model element. For example, Bob's PWA still needs to see the public CRI.2.1 as he does not see Alice's changes at this point and has not made his changes either. The evaluation mechanism evaluate(CRI) is not discussed at this point for brevity—please refer to Reder and Egyed [15].

5.4.2 Addition

In case of an addition, the new model element and all its properties need to be inserted in pwa.Δmodel (line 2 in Listing 6). The addition of a model element cannot cause a re-evaluation because a new model element cannot yet be part of the scope of any CRI (the addition often coincides with a modification in which case the modify algorithm takes care of the latter). However, if a consistency rule exists whose context matches the type of the model element then a CRI must be created and evaluated (lines 5–8). For example, recall that Bob adds a new [UML::Message]connect in his sequence diagram (Fig 2).

```
add(pwa, modelElement)
  for property of modelElement
    pwa.Δmodel[modelElement].[property] =
      modelElement.[property]
  for CR ∈ CRs
    if cr.context == modelElement.type then
      CRI = create instance
      pwa.ΔCRI[CRI] = evaluate(CRI)
```

Listing 6: Addition of a Model Element

After Bob's tool adapter sends this newly added model element, the add algorithm adds the new message to pwa.Δmodel and adds a newly instantiated CRI.1.4.B to pwa.ΔCRI—recall that a CRI is instantiated for every instance of a matching context element.

5.4.3 Deletion

The deletion of a model element (Listing 7) requires the removal of all CRIs whose context elements match the deleted element, as opposed to additions that cause the creation of CRIs. Similar to addition, deletion often coincides with a modification in which case the modify algorithm takes care of the latter. A CRI residing in the private work area can be deleted simply by removing it from pwa.ΔCRI. However, CRIs residing in the public area cannot be deleted because they should continue to exist in the public base version. These CRIs must be flagged as "not alive" in further versions. This is done by adding/modifying a CRI to pwa.ΔCRI with the alive flag being false. Model elements are handled likewise. For example, should Bob delete [UML::Message]wait then the model element and its CRI must be flagged deleted and added to the pwa.Δmodel and pwa.ΔCRI.

```
deletion(pwa, modelElement)
  CRIs = ... //as defined in change(..)
  for CRI ∈ CRIs
    if modelElement ∈ CRI.scope then
      if CRI ∉ pa[baseVersion].CRI then
        remove CRI from pwa.Δ.CRI
      else
        CRI.alive = false
  if modelElement ∉ pa[baseVersion].model then
    for property of modelElement
      remove pwa.Δmodel[modelElement].[property]
  else
    pwa.Δmodel[modelElement].alive = false
```

Listing 7: Deletion of a Model Element

5.5 CRI Evaluations and Property Access

During the evaluation of a CRI the consistency checker accesses model element properties (e.g., the name of [UML::Class]Display). Since a PWA possibly overwrites any public version of a property, these accesses need to be handled differently from traditional approaches. For example, when the consistency checker re-evaluates CRI.2.1.A then the transition's name should be "pause", not "wait". Algorithm getProperty (Listing 8) shows that we first need to look for a model element property in pwa.Δmodel and return its value, if found.

```
getProperty(pwa, modelElement, property)
  if ¬modelElement.alive then
    throw Exception
  if (modelElement, property) ∈ pwa.Δmodel then
    return pwa.Δmodel[modelElement].[property]
  else
    return pa[baseVersion].model[modelElement].[
        property]
```

Listing 8: Property Accesses

However, if the model element is marked as "dead" an exception will be thrown. Otherwise, we return the value of the property of the public model's base version. Therefore, the consistency checker has access to all model elements and properties (public and private) but the private ones have precedence over the public ones. Each PWA thus reflects its own view of the complete model.

5.6 Commit

Once Alice and Bob are done with their changes, they will want to commit them. Committing (Listing 9) requires determining possible version conflicts. Algorithm conflicts

in Listing 10 defines how to obtain conflicts. A model element conflict (`conflictME`) exists for those model elements in `pwa.Δ.model` for which a newer version (higher than the base version) exists. To determine the existing conflicts, the deltas of a range of versions need to be accessed (`pa[lowerVersion, higherVersion]`). The same holds for CRI conflicts (`conflictCRI`) if a new version of a CRI exists upon commit. If conflicts are detected then the commit is aborted and developers are expected to resolve them analogous to SVN and other repositories. Once no conflicts are detected then the PWA changes are moved to the public area and given a new version number—a cheap operation since they already reside in the cloud. Finally, `pwa.Δmodel` and `pwa.ΔCRI` are emptied because after a commit there is again no difference between the tool artifacts and the public area.

```
commit(pwa)
  conflictsME, conflictsCRI = conflicts(pwa,
      version)
  if conflictsME ≠ ∅ ∨ conflictsCRI ≠ ∅ then
    return
  pa[HeadRevision()+1].model = pwa.Δ.model
  pa[HeadRevision()+1].CRI = pwa.Δ.CRI
  pwa.Δ.model = ∅
  pwa.Δ.CRI = ∅
```

Listing 9: Commit

```
conflicts(pwa, version)
  conflictsME = {modelElement ∈ pwa.Δ.model |
    modelElement ∈ pa[pwa.baseVersion, version].
        model
    ∨∃ prop ∈ modelElement.[property] :
      prop ∈ pa[pwa.baseVersion, version].model }
  conflictsCRI = {CRI ∈ pwa.Δ.CRI |
    CRI ∈ pa[pwa.baseVersion, version].CRI}
  return (conflictsME, conflictsCRI)
```

Listing 10: Conflicts

5.7 Update

Developers may update the model they are currently modifying, if a newer version or conflicts exists. Before updating, version conflicts (if any) must be determined and resolved manually. Updating a private working area (Listing 11) consists of the following steps: i) determine the differences between the PWA's base version and the desired version, ii) determining the currently existing version conflicts, iii) updating the base version to the new higher version, and finally returning the conflicts and the differences to the tool. Finally, the differences between base version, new desired version result, and already existing deltas in the PWA may require additional consistency checks.

```
update(pwa, version)
  diff = pa[pwa.baseVersion, version].model
  conflictsME, conflictsCRI = conflicts(pwa,
      version)
  pwa.baseVersion = version
  CRIs = {CRI ∈ pa[baseVersion].CRI |
      CRI.alive == true and CRI ∉ pwa.ΔCRI}
  CRIs ∪= pwa.ΔCRI
  for (modelElement, property) ∈ diff
    for CRI ∈ CRIs
      if (modelElement, property) ∈ CRI.scope
        pwa.ΔCRI[CRI] = evaluate(CRI)
  return conflictsME, conflictsCRI, diff
```

Listing 11: Update

6. VALIDATION

In Section 5 we discussed how our approach integrates PWAs with a public area in a cloud-based environment to enable comprehensive, complete consistency checking in a multi-developer environment. To validate our approach, we i) demonstrate its feasibility by developing a prototype implementation, ii) analyzed the computational complexity of our approach, iii) performed an empirical evaluation to assess the actual overhead imposed by our approach compared to the Model/Analyzer applied to UML on a single tool, iv) analyzed the memory consumption of storing the CRIs, v) discuss the advantages of our approach in terms of memory consumption for each PWA, and vi) conducted three case studies to demonstrate the applicability of our approach.

6.1 Prototype Implementation

This section provides a short overview of the implementation details of the version control system and the incremental consistency checker.

Version Control System - DesignSpace The version control proof of concept implementation, called *DesignSpace (DS)* [13, 27], is a cloud-based infrastructure that allows to version control arbitrary development artifacts on a fine-granular basis.

Incremental Consistency Checker - Model/-Analyzer As consistency checker the *Model/Analyzer (M/A)* [30] was adapted and employed. Previously existing implementations already worked with UML, EMF Core (*ECore*) [29], and also RDF. For this work, it was adapted to work with the representation used by the VCS and its change notifications.

6.2 Computational Complexity

The computational complexity of consistency checking is mostly a factor of the number of necessary CRI evaluations. The changes that cause the re-evaluation of CRIs are `add` and `modify`—the `deletion` of model elements does not cause re-evaluations. Previous validations of the Model/Analyzer [**?**] suggested that the average number of evaluations of CRIs per change is between 3 and 11 (depending on the model size). Following, we denote this value as constant c. We define the set of performed changes (which trigger re-evaluations; i.e., add, modify) by an individual developer as *change.am*. Equation 1 defines the total number of CRI evaluations $CRI.E_{approach}$ as the sum of evaluations required for each individual developer.

$$CRI.E_{approach} = \sum_{i=1}^{|developer|} (c * |change.am_i|) \qquad (1)$$

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

To empirically assess the overhead imposed by our approach, we re-enacted a previous evaluation of the Model/-Analyzer using our cloud (i.e., we used the same models and performed random changes as described by Reder and Egyed [15]). For this, we simulated changes for all model elements in each of the used models. Specifically, we captured the time required for re-evaluating all affected CRIs and to persist the new result. Each change was performed

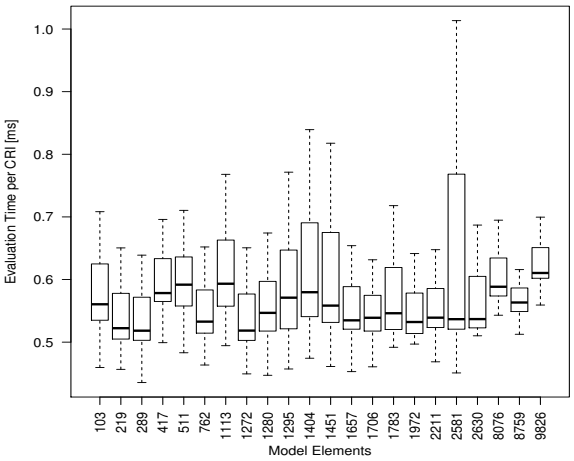


Figure 6: Evaluation Time per affected Instance

Table 1: Regression Results for Total Processing Time

	Total Processing Time (ms)
Model Elements	0.0004581***
	(0.0000068)
Affected Instances (AI)	-0.0017938
	(0.0074605)
Evaluation Time per Instance (ET)	-0.0001086
	(0.0003659)
AI*ET	1.0003651***
	(0.0002575)
Observations	48708
R^2	0.9996

Standard errors in parentheses
* $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

several times and the raw data for each change was recorded. The experiment was performed on a Windows 7 Professional PC with an Intel(R) Core(TM) i7-3770 CPU @ 3,4GHz and 16GB of RAM. Figure 6 depicts the evaluation times per affected CRI on average depending on the project size. We furthermore performed an OLS regression analysis on the total processing time (i.e., the time it takes to find affected CRIs and to re-evaluate them). The obtained results are summarized in Table 1, which shows that the model size has a significant, yet quite small effect on the evaluation time. Increasing the model size by 1,000 elements leads to a total processing time increase of 0.4ms on average. The results furthermore indicate that the number of affected CRIs and evaluation times per CRI individually do not affect the total processing time. However, in combination those factors are significant determinants of the total processing time. On average, an increase of the number of affected instances by one increases the total processing time by 23ms. An increase of the evaluation time per affected instance by 1ms increases the total processing time by 1.5ms on average. Note that more than 99.9% of sample variations are explained by the analyzed factors. Comparing to previous evaluations of the Model/Analyzer [15], we obtained similar result and our framework did not slow down the evaluation times.

6.3 Memory Consumption

The memory consumption of our approach correlates with the number of CRIs that need to be maintained overall and for each developer individually. Equation 2 defines the total number of CRIs that have to be persisted in our approach ($CRI_{approach}$). The first factor describes the set of CRIs needed for the public area, which is the number of CRIs for a given model ($CRI(model)$). Furthermore, since a PWA stores the Δ with respect to the base version, we must consider the effect of modifications, additions, and deletions. Each of these operations adds a new entry to `pwa.`Δ`CRI`. Recall the average number of CRIs affected by a single change c, each performed change adds c CRI entries. Note that the number of CRIs needed for an entire model grows linearly with the model size (factor $CRI(model)$). The CRIs' memory consumption in the public area is thus linear with the model size. Furthermore the CRI's memory consumption in the PWAs is again linearly dependent on the number of changes per developer and the total number of developers.

$$CRI_{approach} = CRI(model) + \sum_{i=1}^{|developer|} (c * |change_i|) \quad (2)$$

This equation again shows the scalability of our approach.

6.4 Version control mechanism

In this section, we discuss the advantages of our approach in terms of memory consumption for each individual PWA. As previously stated, a PWA stores the Δ`model` and furthermore Δ`CRI`. We provide a behavioral analysis of how a PWA's memory footprint depends on model changes compared to using the Model/Analyzer as a plugin.

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

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

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

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

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

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

Equation 6 – Equation 7 state our assumptions about both Δ`model` and Δ`CRI`. The size of Δ`model` is expressible through a function of the model size (Equation 6) (e.g., a developer may always change a fixed number of elements per commit). Furthermore, the size of Δ`CRI` is completely

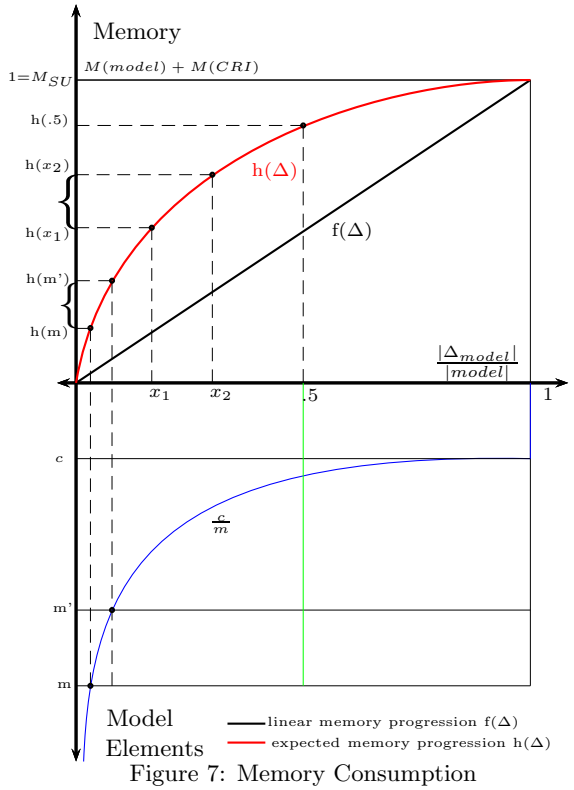


Figure 7: Memory Consumption

determined by the function CRI (Equation 7), which takes as argument Δ`model`.

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

$$|\Delta\texttt{CRI}| = CRI(\Delta\texttt{model}) \quad (7)$$

Memory consumption of PWAs is mostly influenced by Δ`model`. In Fig. 7 a behavioral analysis for possible cases of $DeltaModel$ is presented, as described below.

6.4.1 *Memory Consumption per distinct Model Elements changed*

First, we discuss memory consumption per distinct changed model elements, which is plotted on the positive y-axis in Fig. 7. Values on the positive y-axis are normalized as follows, $(M(\Delta_{model}) + M(\Delta_{CRI}))/M_{SU}$ (i.e., $\frac{\text{actual memory consumption}}{\text{possible memory consumption}}$).

The x-axis shows that the ratio of distinct model element changes to the model size (1 implies the entire model changed).

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

6.4.2 *Memory Consumption depending on Model Size*

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

6.5 Case Studies

Our infrastructure was employed in three user studies, which we briefly discuss next. One of these case studies was performed with the *Austrian Center of Competence in Mechatronics (ACCM)*, another with the *Flanders' Mechatronics Technology Center (FMTC)*, and a third study involved students of the Johannes Kepler University.

6.5.1 *ACCM Robot Arm*

This case study involved the design, mechanical calculation and partial implementation of a robot arm. The ACCM group provided the mechanical calculation of the robot arm in several Excel sheets and CAD drawings. UML models of the robot arm were drawn in IBM RSA, Eclipse provided the source code. The mechatronic design followed a trial-and-error cycle where most changes were instantly propagated in a synchronous collaboration style. The scope of the project exceeds the discussion in this paper. However, we provided instant consistency checking and the ability to define and check cross-tool traceability links.

6.5.2 *FMTC*

This FMTC case study involved EPLAN Electric P8 [25] drawings that had to be kept consistent with source code. The drawings and code were provided by a third-party company. The goal was to enable instant consistency checking across tool boundaries in an efficient manner.

6.5.3 *Student Experiment*

Finally, our approach was applied in a student survey, conducted as part of a research project by a K2-Center of the COMET/K2 program. The goal was to study the benefits of different representations for expressing dependencies in mechatronical designs. In particular, whether graph visualization are inferior to matrix representations. Participants of the experiment were 32 students who had to perform a design refactoring based on a requirement from a customer. The nature of the case study required that all development artifacts resided in the same private work area, which enabled a quick trial-and-error cycle given the provided consistency information.

6.6 Threats to Validity

In terms of computational complexity, we demonstrated that for each developer only a small set of CRIs needs to

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

7. RELATED WORK

Both consistency checking and version control are active fields of research, following research related to our approach is discussed.

Global Consistency Checking: These approaches address the general problem of consistency checking across possibly distributed development artifacts of a system. Finkelstein et al. [8] introduced consistency checking in multi perspective specifications. Each developer owns a perspective (i.e., a *viewpoint* [5]) of the system according to his or her knowledge, responsibilities or commitments. Multiple viewpoints can describe the same design fragment, leading to overlap and the possibility of inconsistencies. The issues involved in inconsistency handling of multi perspective specifications are outlined in Finkelstein et al.[31]. An important insight for handling consistency is to allow models to be temporarily inconsistent, rather than enforcing full consistency at all times. Despite this advances, no implementation is provided. Further, the approach does not difference between, a public state that is fixed (i.e. the contents of the repository) and private modifications. Nentwich et al. with *xlinkit* [12] present a framework for consistency checking distributed software engineering documents encoded in XML. Sabetzadeh et al. [32] presented global consistency checking by model merging. The approach focuses on handling inconsistencies between multiple models expressed in a single language. Although both approaches consider distributed models, at the time of the consistency check there is no distinction between private adaptations and public knowledge.

Consistency Checking in General: Numerous approaches exist for consistency checking, specializing on specific artifact types or across artifacts [33]. Many of these approaches can also be used to check consistency across several development artifacts. Finally, two approaches need to be highlighted, as these could have been replacements for the Model/Analyzer. Blanc et al. [34] look at the sequence of operations used to produce the model rather than looking at the model itself. Thus they can not only verify structural consistency of model but also methodological consistency. Reiss presented an approach (CLIME) to incremental maintenance of software artifact [3]. This approach covers a multitude of development artifacts (presented are source code, models and test cases). As unified representation information is extracted from the development artifact (e.g., symbol table for source code, the class diagram itself) and stored in a SQL Database.

Version control: Version control for text-based development artifacts permeate software engineering and academia. Further, extensive research was conducted on version control of models, a survey is presented by Altmanninger et al. [35]. Research in the area of version control systems analyze their version controlled development artifacts to find inconsistencies. An example of such an approach was presented by Taentzer et al. [36]. The approach considers the abstract syntax of models as graphs. Revisions are graph modifications, based on this, they identify two kinds of conflicts, operation-based and state-based as a result of merged graph modifications. State-based conflicts are concerned with the well-formedness, operation-based conflicts are then concerned on the parallel dependence of graph transformations and the extraction of critical pairs. Cicchetti et al. [37] proposed a meta-model for representing conflicts which can be used for specifying both syntactic as well as semantic conflicts. Finally, two popular version control systems need to be mentioned *GIT* [38] and *Apache Subversion (SVN)* [28]. Both inherently provide capabilities to create a continuous integration, during which source code checks are executed. However, the authors are not aware of any tools to extent this idea to also check consistency across multiple artifacts. Applying our approach to the continuous integration phase, would allow that for a commit or push to a feature branch the consistency checker would verify the impacts of the performed changes with respect to all development artifacts. Another, distributed, revision control and source code management system is *GIT* [38].With GIT, each obtained working directory is a full-fledged repository—a `clone` copies the entire history of the repository to the working directory. Therefore, a developer has the complete standard workflow available without being dependent on network access or a central server. To publish changes a `push` to a remote server is necessary. However, since different implementations of GIT (and extensions to the typical workflow) exists, a push may at first be pushed to a staging area, where it needs to be reviewed, before being approved and becoming part of the main trunk (e.g., Gerrit [39]).

8. CONCLUSION

This paper presented a novel approach to multi-developer consistency checking. The approach eliminates consistency checking redundancies in a multi-development environment to reduce the CPU and memory footprint to a relative constant per developer. We demonstrated that our approach completely eliminates the model size as a scalability factor and is fast. For future work, further development tools will be integrated with our infrastructure (i.e., create tool-adapters).

Acknowledgments

The research was funded by the Austrian Science Fund (FWF): P25513-N15 and P25289-N15, and the Austrian Center of Competence in Mechatronics (ACCM): C210101.

9. REFERENCES

[1] "Eclipse http://www.eclipse.org/ ,2013.."

[2] "IBM Rational Software Architect https://www.ibm.com/developerworks/rational/products/rsa/ ,2013.."

[3] S. P. Reiss, "Incremental Maintenance of Software Artifacts," *IEEE Trans. Software Eng.*, vol. 32, no. 9, pp. 682–697, 2006.

[4] P. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, vol. 12, no. 6, pp. 42–50, 1995.

[5] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development," *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, no. 1, pp. 31–57, 1992.

[6] P. Fradet, D. L. Métayer, and M. Périn, "Consistency Checking for Multiple View Software Architectures," in *ESEC / SIGSOFT FSE*, pp. 410–428, 1999.

[7] R. B. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," in *FOSE*, pp. 37–54, 2007.

[8] A. Finkelstein, D. M. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency Handling in Multi-Perspective Specifications," in *ESEC*, pp. 84–99, 1993.

[9] M. Sabetzadeh, S. Nejati, S. M. Easterbrook, and M. Chechik, "Global consistency checking of distributed models with TReMer+," in *ICSE*, pp. 815–818, 2008.

[10] A. Egyed, "Automatically detecting and tracking inconsistencies in software design models," *IEEE Trans. Software Eng.*, vol. 37, no. 2, pp. 188–204, 2011.

[11] Michael Vierhauser and Paul Grünbacher and Alexander Egyed and Rick Rabiser and Wolfgang Heider, "Flexible and scalable consistency checking on product line variability models," in *ASE*, pp. 63–72, 2010.

[12] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein, "xlinkit: a Consistency Checking and Smart Link Generation Service," *ACM Trans. Internet Techn.*, vol. 2, no. 2, pp. 151–185, 2002.

[13] M. Riedl-Ehrenleitner, A. Demuth, and A. Egyed, "Towards Model-and-Code Consistency Checking," in *COMPSAC*, pp. 85–90, 2014.

[14] J. C. Grundy, J. G. Hosking, K. N. Li, N. M. Ali, J. Huh, and R. L. Li, "Generating Domain-Specific Visual Language Tools from Abstract Visual Specifications," *IEEE Trans. Software Eng.*, vol. 39, no. 4, pp. 487–515, 2013.

[15] A. Reder and A. Egyed, "Incremental Consistency Checking for Complex Design Rules and Larger Model Changes," in *MoDELS*, pp. 202–218, 2012.

[16] X. Blanc, A. Mougenot, I. Mounier, and T. Mens, "Incremental Detection of Model Inconsistencies Based on Model Operations," in *CAiSE*, pp. 32–46, 2009.

[17] "Eclipse OCL http://projects.eclipse.org/projects/modeling.mdt.ocl, 2015.."

[18] OMG, *ISO/IEC 19507 Information technology - Object Management Group Object Constraint Language (OCL)*. ISO, 2012.

[19] C. Xu, S. Cheung, and W. K. Chan, "Incremental consistency checking for pervasive context," in *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pp. 292–301, 2006.

[20] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer, "Flexible consistency checking," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 1, pp. 28–63, 2003.

[21] D. Beyer, "Relational programming with CrocoPat," in *ICSE*, pp. 807–810, 2006.

[22] "RDF/XML Syntax Specification (Revised)."

[23] M. Koegel and J. Helming, "EMFStore: a model repository for EMF models," in *ICSE (2)*, pp. 307–308, 2010.

[24] J. Estublier, T. Leveque, and G. Vega, "Evolution control in MDE projects: Controlling model and code co-evolution," in *Fundamentals of Software Engineering, Third IPM International Conference, FSEN 2009, Kish Island, Iran, April 15-17, 2009, Revised Selected Papers* (F. Arbab and M. Sirjani, eds.), vol. 5961 of *Lecture Notes in Computer Science*, pp. 431–438, Springer, 2009.

[25] "EPlan Electric P8 http://www.eplanusa.com/us/solutions/product-overview/eplan-electric-p8/."

[26] "PTC, Inc. http://www.ptc.com/product/creo, 2015."

[27] A. Demuth, M. Riedl-Ehrenleitner, A. Nöhrer, P. Hehenberger, K. Zeman, and A. Egyed, "DesignSpace – An Infrastructure for Multi-User/Multi-Tool Engineering," in *SAC*, pp. 1486–1491, 2015.

[28] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, *Version control with subversion - next generation open source version control*. O'Reilly, 2004.

[29] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd ed., 2009.

[30] A. Reder and A. Egyed, "Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML," in *ASE* (C. Pecheur, J. Andrews, and E. D. Nitto, eds.), pp. 347–348, ACM, 2010.

[31] A. Finkelstein, D. M. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency handling in multperspective specifications," *IEEE Trans. Software Eng.*, vol. 20, no. 8, pp. 569–578, 1994.

[32] M. Sabetzadeh, S. Nejati, S. Liaskos, S. M. Easterbrook, and M. Chechik, "Consistency checking of conceptual models via model merging," in *15th IEEE International Requirements Engineering Conference, RE 2007, October 15-19th, 2007, New Delhi, India*, pp. 221–230, IEEE Computer Society, 2007.

[33] F. J. Lucas, F. Molina, and J. A. T. Álvarez, "A systematic review of UML model consistency management," *Information & Software Technology*, vol. 51, no. 12, pp. 1631–1645, 2009.

[34] X. Blanc, I. Mounier, A. Mougenot, and T. Mens, "Detecting model inconsistency through operation-based model construction," in *ICSE*, pp. 511–520, 2008.

[35] K. Altmanninger, M. Seidl, and M. Wimmer, "A survey on model versioning approaches," *IJWIS*, vol. 5, no. 3, pp. 271–304, 2009.

[36] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer, "Conflict Detection for Model Versioning Based on Graph Modifications," in *ICGT*, pp. 171–186, 2010.

[37] A. Cicchetti, D. D. Ruscio, and A. Pierantonio, "Managing Model Conflicts in Distributed Development," in *MoDELS*, pp. 311–325, 2008.

[38] J. Loeliger, *Version Control with Git - Powerful techniques for centralized and distributed project management*. O'Reilly, 2009.

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