Karlsruher Institut für Technologie

Extrahieren von Code-Änderungen aus einem Commit für kontinuierliche Integration von Leistungsmodellen

Codereview für eine Bachelorarbeit von

Ilia Chupakhin

24. Juli 2020

an der Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation (IPD)

Erstgutachter:	Prof. Dr.-Ing. Anne Koziolek
Zweitgutachter:	Prof. Dr. Ralf H. Reussner
Betreuender Mitarbeiter:	M.Sc. Manar Mazkatli

–

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Inhaltsverzeichnis

1 Einleitung **1**

2 Installationsanleitung **3**
2.1 Software-Installation 3
2.2 Ausführung von Tests 4

3 Architektur **5**
3.1 Plugin tools.vitruv.applications.pcmjava.integrationFromGit 6
3.2 Plugin tools.vitruv.applications.pcmjava.integrationFromGit.test 7

4 Case-Study-Ergebnisse **11**

Literatur **15**

1 Einleitung

Ein Performance-Modell ermöglicht den Software-Entwicklern eine frühzeitige Analyse von programmierten Komponenten in Bezug auf Leistungseigenschaften, wie zum Beispiel Speicherbedarf oder Ausführungsdauer. Das Performance-Modell muss mit allen anderen im System vorhandenen Modellen konsistent gehalten werden. Falls ein Modell geändert wurde, muss auch das Performance-Modell aktualisiert werden. Allerdings ist eine Aktualisierung des Performance-Modells für große Systeme aufwändig. Ein Ansatz für eine effiziente Aktualisierung von Performance-Modellen wurde in [2] vorgestellt und in [3] weiterentwickelt. Wir bezeichnen diesen Ansatz als Continuos Integration of Performance Model (CIPM).
In dieser Bachelorarbeit implementieren wir den ersten Schritt für den CIPM-Ansatz. Wir verknüpfen den CIPM-Ansatz mit einem Git Repository und extrahieren Änderungen aus Commits. Anhand von den extrahierten Änderungen passen wir die existierenden Code-Modelle an. Anschließend werden diese Änderungen zu den Performance-Modellen automatisch propagiert. Für diesen Zweck haben wir die existierenden Change-Propagation-Regeln angepasst und einige neue Regeln implementiert.
Unsere Implementierung haben wir in einer Case-Study evaluiert. Für ein Projekt haben wir unterschiedliche Arten von Änderungen simuliert und sie als Commits in einem Git-Repository gespeichert. Danach haben wir dieses Projekt in Vitruvius integriert, Änderungen aus den Commits gelesen und die entsprechenden Modelle angepasst. Anschließend haben wir die Korrektheit der aktualisierten Code- und Performance-Modelle überprüft.
Das Kapitel 2 enthält eine Installationsanleitung und Instruktionen für die Ausführung von Tests. In dem Kapitel 3 beschreiben wir die Architektur unserer Implementierung und der dazu gehörenden Tests. Die Ergebnisse der durchgeführten Case-Study zeigen wir in dem Kapitel 4.

2 Installationsanleitung

2.1 Software-Installation

- Java, Version 12
- Eclipse Modeling Tools, Version 2020-03
  https://www.eclipse.org/downloads/packages/release/2020-03/r
- In Eclipse im Menü "Help -> Install new software"die folgenden Plugins installieren:
  - Henshin
    http://download.eclipse.org/modeling/emft/henshin/updates/release
  - EMFText, Version 1.4.1
    http://update.emftext.org/trunk/
  - JaMoPP, Version 1.4.1
    http://update.jamopp.org/trunk/
  - Palladio, Version 4.1 (release 1.1.0)
    https://updatesite.palladio-simulator.com/palladio-build-updatesite/releases/1.1.0/
  - Vitruvius
    https://vitruv-tools.github.io/updatesite/nightly/
  - SoMoX
    http://kit-sdq.github.io/updatesite/nightly/somox/jamopp/
- Die folgenden Plugins von https://github.com/vitruv-tools/Vitruv-Applications-PCMJavaAddition
  von dem Branch "modelsUpdateFromGitCommits"herunterladen und in Eclipse workspace
  importieren (File -> Import -> General -> Existing Projects into Workspace):
  - org.palladiosimulator.pcm.modified
  - org.somox.test.gast2seff
  - tools.vitruv.applications.pcmjava.integrationFromGit
  - tools.vitruv.applications.pcmjava.integrationFromGit.test
  - tools.vitruv.applications.pcmjava.linkingintegration
  - tools.vitruv.applications.pcmjava.linkingintegration.change2command
  - tools.vitruv.applications.pcmjava.linkingintegration.ejbtransformations
  - tools.vitruv.applications.pcmjava.seffstatements
  - tools.vitruv.applications.pcmjava.seffstatements.pojotransformations

- Alle Plugins von https://github.com/maxil063/sdq/tree/master/changedPlugins von dem Branch "master"herunterladen und in Eclipse workspace importieren (File -> Import -> General -> Existing Projects into Workspace).

2.2 Ausführung von Tests

In Eclipse Run Configuration öffnen (Run -> Run Configuration). In der Liste 'JUnit Plug-in Test' das Plugin tools.vitruv.applications.pcmjava.integrationFromGit.test wählen. Es wird empfohlen, die Tests einzeln auszuführen. 'Run a single test' aktivieren, als 'Test Class' einen Test aus der Liste wählen (Die Namen von den ausführbaren Tests fangen mit 'IA' oder mit 'NIA' an), als 'Test runner' JUnit 4 wählen und die Option 'Run in UI Thread' deaktivieren (das ermöglicht eine Interaktion mit dem Benutzer mithilfe von User Dialogs).
Während der Ausführung wird in manchen Tests ein User Dialog mehrmals gezeigt, wo der Benutzer eine Wahl treffen muss. Falls der Benutzer keine Wahl trifft, wird der Test nach dem zweiten Timeout weiter ausgeführt und liefert falsche Ergebnisse oder schlägt fehl. Die folgenden Wahlmöglichkeiten werden zur Zeit unterstützt:

- 'create Basic Component' falls ein Package oder eine Klasse erstellt wurde
- 'create interface' falls ein Interface erstellt wurde

In manchen Tests werden Timeouts ausgelöst, obwohl es kein User Dialog angezeigt wurde. In diesem Fall wird der Test nach zwei Timeouts weiter korrekt ausgeführt.

3 Architektur

Unsere Implementierung ist in dem Plugin tools.vitruv.applications.pcmjava.integrationFromGit. Die Tests befinden sich in dem Plugin tools.vitruv.applications.pcmjava.integrationFromGit.test. Die unvollständigen Klassendiagramme für die beiden Plugins sind in den Abbildungen 3.1 und 3.2 dargestellt.

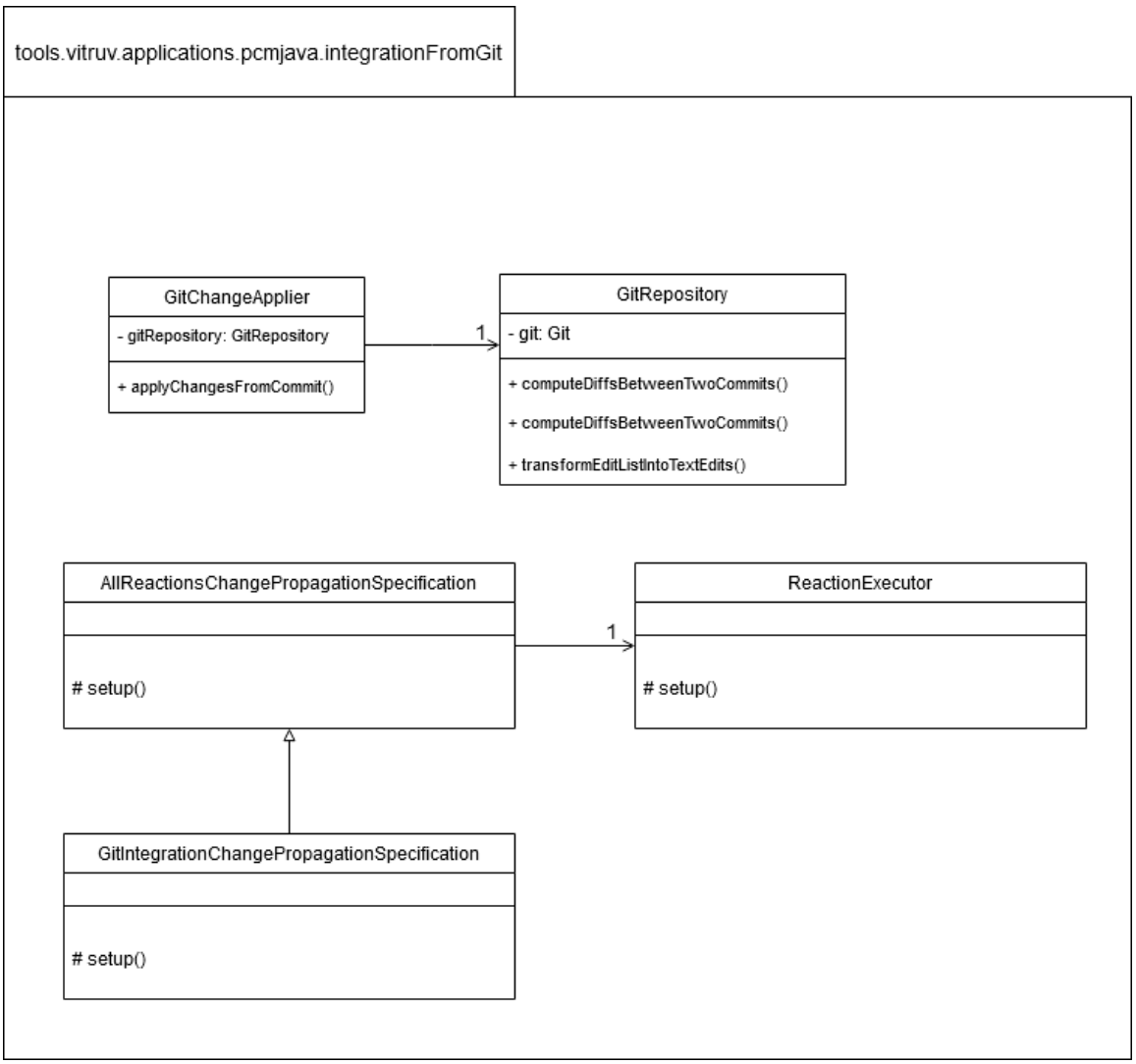

Abbildung 3.1: Klassendiagram für das Plugin tools.vitruv.applications.pcmjava.integrationFromGit. Das Klassendiagramm ist unvollständig und dient nur einer Darstellung der einzelnen Klassen, Methoden und Variablen.

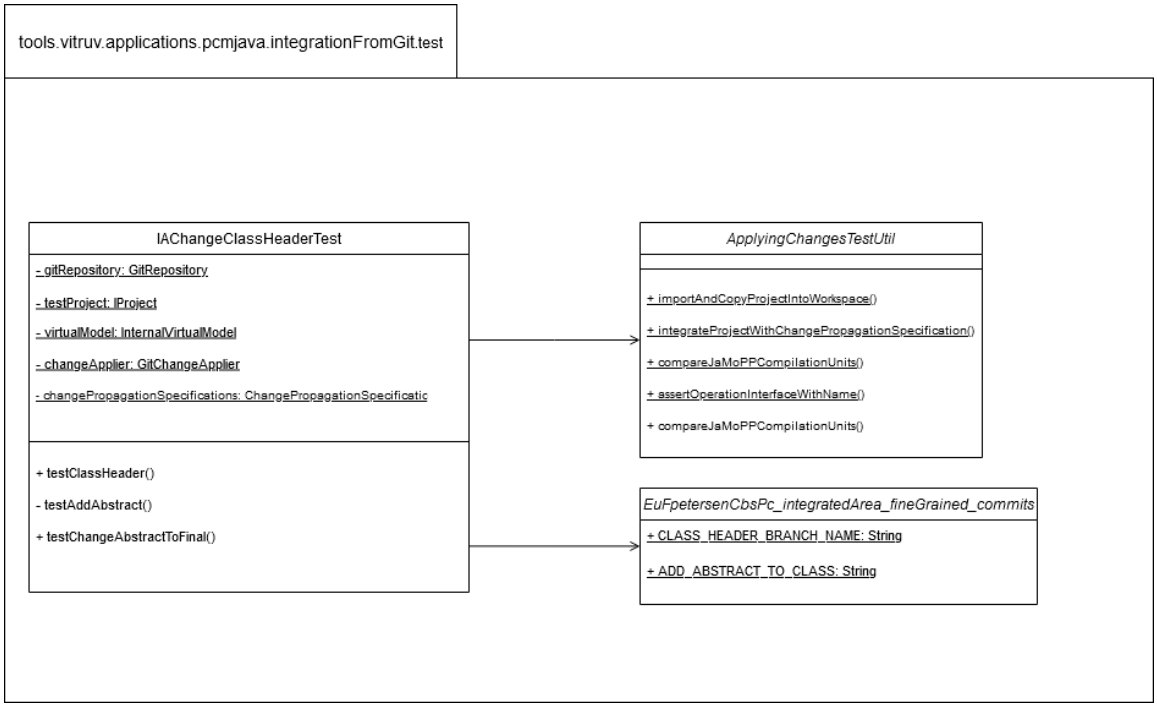

Abbildung 3.2: Klassendiagram für das Plugin tools.vitruv.applications.pcmjava.integrationFromGit.test. Das Klassendiagramm ist unvollständig und dient nur einer Darstellung der einzelnen Klassen, Methoden und Variablen.

3.1 Plugin tools.vitruv.applications.pcmjava.integrationFromGit

Die Klasse GitChangeApplier wendet extrahierte Änderungen auf die Code-Modelle an. Die Hauptmethode applyChangesFromCommit nimmt als Parameter zwei Commits, zwischen denen Änderungen ausgerechnet werden müssen, sowie ein JDT-Modell von einem Java-Projekt. Die Methode ermittelt Änderungen, klassifiziert sie und ruft eine passende Routine auf. Zum Beispiel, für eine als 'ADD' klassifizierte Änderung wird die Methode addElementToProject aufgerufen. Nach einer erfolgreichen Ausführung sind die Änderungen auf das JDT-Modell von dem Java-Projekt angewandt. Falls dieses Projekt in Vitruvius integriert ist, werden auch die JaMoPP-Modelle in VSUM automatisch angepasst und gegebenenfalls auch die korrespondierenden PCM-Modelle.
Anders als in dem GitChangeReplay-Tool [1] werden in unserer Implementierung die aus einem Commit extrahierten Änderungen nicht in atomare Änderungen zerlegt. Für unseren Zweck ist es nicht notwendig. Darüber hinaus würde das die Performance unserer Implementierung negativ beeinträchtigen.
Änderungen an den JaMoPP-Modellen in VSUM werden von dem Monitor detektiert. Anschließend werden die entsprechenden Listener benachrichtigt. Dieser Mechanismus ermöglicht eine Aktualisierung von den korrespondierenden Performance-Modellen. Als

[1] GitChangeReplay-Tool: https://github.com/vitruv-tools/GitChangeReplay

Performance-Modelle haben wir die Palladio Component Models (PCM) benutzt. Um die Performance-Modelle automatisch anpassen zu können, muss eine Change Propagation Specification (CPS) und Change Propagation Rules (CPR) definiert werden. Unsere CPS ist in der Klasse GitIntegrationChangePropagationSpecification definiert. Die CPR haben wir in der Reaction-Sprache definiert. Sie befinden sich in den Dateien PackageAndClassifiers.reactions und ClassifierBody.reactions. Die meisten CPR haben wir aus den folgenden Projekten übernommen und angepasst:

- tools.vitruv.applications.pcmjava.linkingintegration.change2command.internal.response.PackageMappingIntegration.reactions
- tools.vitruv.applications.pcmjava.pojotransformations.java2pcm

Die CPR für die Änderungen innerhalb der Methoden haben wir unverändert gelassen. Sie stammen aus dem folgenden Projekt:

- tools.vitruv.applications.pcmjava.seffstatements.pojotransformations.Java2PcmPackageMappingMethodBodyChangePreprocessor.xtend

Außerdem haben wir auch neue CPR definiert. Zum Beispiel eine Reaktion auf löschen einer Klasse und eines Packages. Die Anpassungen in den bereits existierenden CPR waren notwendig, weil sie für unsere Case-Study nicht (korrekt) funktioniert haben. Ein Grund dafür war, dass die ursprünglichen CPR nicht für die integrierten Projekte gedacht sind, sondern für die Projekte, die von Anfang an mit Vitruvius entwickelt wurden. Zum Beispiel, wenn ein neues Package erstellt wurde, wurden automatisch Subpackages 'datatypes' und 'contracts' erstellt. Das hat aber zu Inkonsistenzen zwischen der Projektstruktur in dem Git Repository und der in Vitruvius geführt, weil in Git Repository keine Subpackages 'datatypes' und 'contracts' existieren.

3.2 Plugin tools.vitruv.applications.pcmjava.integrationFromGit.test

Die Tests haben wir in zwei Packages unterteilt:

- tools.vitruv.applications.pcmjava.integrationFromGit.test.integratedArea
- tools.vitruv.applications.pcmjava.integrationFromGit.test.nonIntegratedArea

Die Tests in tools.vitruv.applications.pcmjava.integrationFromGit.test.integratedArea simulieren Änderungen auf den Teilen des Projekts, die bereits vor der Integration in Vitruv existiert haben und als einen integrierten Bereich zählen. Was als ein integrierter Bereich zählt ist projektspezifisch. In unserem Test-Projekt gelten alle Packages innerhalb des source-Packages als integrierten Bereich. Mehr Details über integrierte und nicht integrierte Bereiche können in [1] gefunden werden. Die Tests in tools.vitruv.applications.pcmjava.integrationFromGit.test.integratedArea simulieren Änderungen auf den Teilen des Projekts, die vor der Integration in Vitruv noch nicht existiert

haben. Außerdem finden diese Änderungen in einem Bereich statt, der als nicht integrierten Bereich zählt. In unserem Test-Projekt zählt ein neues Package innerhalb des source-Packages als ein nicht integrierter Bereich. Das neue Package darf nicht in die anderen schon existierenden Subpackages verschachtelt sein. Sonst würde es als ein integrierter Bereich erkannt.
In unseren Tests wollten wir zeigen, dass die von uns angepassten Change Propagation Rules (CPR) sich gleich verhalten ungeachtet darauf, ob Änderungen in einem integrierten oder nicht integrierten Bereich stattgefunden haben. In der Zukunft sollte diese Unterscheidung komplett irrelevant sein, weil der CIPM-Ansatz [2] auf den Java-Projekten mit beliebigen Strukturen anwendbar sein soll.
In dem Package tools.vitruv.applications.pcmjava.integrationFromGit.test.commits sind die Namen von den Git-Branches und Commit-Hashes für alle Tests gespeichert.
In dem Ordner tools.vitruv.applications.pcmjava.integrationFromGit.test/testProjects ist unser Test-Projekt gespeichert.
In der Klasse ApplyingChangesTestUtil.java sind die Hilfsmethoden für alle Tests enthalten. Die meisten Methoden sind statisch und können deshalb auch in den anderen Projekten benutzt werden.
Jeder Test überprüft die Korrektheit unserer Implementierung für einen bestimmten Änderungstypen. Zum Beispiel der Test IAChangeClassHeaderTest.java simuliert Änderungen eines Klassenkopfes. Das Präfix 'IA' steht für Integrated Area und 'NIA' für Non-Integrated Area. Alle Tests haben eine ähnliche Struktur:
1. Vorbereitung (setUpBeforeClass-Methode):
Zuerst wird ein Projekt in das aktuelle Workspace kopiert und ein JDT-Modell davon erstellt (IProject testProject). Ebenso wird dieses Projekt mit seinem Git-Repository in das Workspace kopiert (in einen separaten Ordner namens 'clonedGitRepositories'). Die Objekte GitRepository und GitChangeApplier verweisen auf 'clonedGitRepositories'. Das erstellte JDT-Modell (IProject testProject) wird in Vitruvius integriert. Dabei entsteht ein Ordner 'vitruvius.meta' in dem Workspace, ein Monitor, der Änderungen zu den korrespondierenden Modellen propagiert, und ein InternalVirtualModel-Objekt. Das InternalVirtualModel-Objekt enthält unter anderem eine Referenz auf die 'vitruvius.meta'-Ordner und GitIntegrationChangePropagationSpecification-Objekt. Für die Tests in den nicht integrierten Bereichen werden auch nicht integrierte Bereiche vorbereitet (Packages, Klassen, Interfaces erstellt).
2. Der eigentliche Test:
Die Abbildung 3.3 zeigt einen Test für das Hinzufügen einer Klassenvariable in einer Klasse, die in einem integrierten Bereich liegt. Zuerst wird eine Änderung aus einem Commit gelesen und auf das in Vitruvius integrierte Projekt angewandt (changeApplier.applyChangesFromCommit(...)) und das geänderte JDT-Modell gefunden (ICompilationUnit compUnitChanged). Als Referenzmodell wird das Git-Repository an dem gleichen Commit ausgecheckt und ein temporäres JDT-Modell von der gleichen (aber nicht der selben) Compilation Unit erstellt (gitRepository.checkoutFromCommitId(...) und ICompilationUnit compUnitFromGit). Anschließend werden die JaMoPP-Modelle für die beiden Compilation Units (compUnitChanged und compUnitFromGit) verglichen (ApplyingChangesTestUtil.compareJaMoPPCompilationUnits(...)). Das JaMoPP-Modell für compUnitFromGit wird neu erstellt während das JaMoPP-Modell für compUnitChanged wird aus

InternalVirtualModel genommen. Als letztes wird es überprüft, ob ein PCM-Modell für die neu erstellte Klassenvariable existiert (ApplyingChangesTestUtil.assertFieldWithName(...)). Eine explizite Überprüfung der nötigen Korrespondenz ist nicht notwendig, weil sie implizit in der Methode ApplyingChangesTestUtil.assertFieldWithName(...) stattfindet.

```
private void testAddField() throws NoHeadException, GitAPIException, IOException, CoreException, InterruptedException {
    //Apply changes
    changeApplier.applyChangesFromCommit(commits.get(EuFpetersenCbsPc_integratedArea_fineGrained_commits.ADD_IMPORT_FOR_FILED),
            commits.get(EuFpetersenCbsPc_integratedArea_fineGrained_commits.ADD_FIELD), testProject);
    //Checkout the repository on the certain commit
    gitRepository.checkoutFromCommitId(EuFpetersenCbsPc_integratedArea_fineGrained_commits.ADD_FIELD);
    //Create temporary model from project from git repository. It does NOT add the created project to the workspace.
    projectFromGitRepository = ApplyingChangesTestUtil.createIProject(workspace, workspace.getRoot().getLocation().toString()
            + "/clonedGitRepositories/" + testProjectName + ".withGit");
    //Get the changed compilation unit and the compilation unit from git repository to compare
    ICompilationUnit compUnitFromGit = CompilationUnitManipulatorHelper.findICompilationUnitWithClassName("GraphicsCard.java", projectFromGitRepository);
    ICompilationUnit compUnitChanged = CompilationUnitManipulatorHelper.findICompilationUnitWithClassName("GraphicsCard.java", testProject);
    //Compare JaMoPP-Models
    boolean jamoppClassifiersAreEqual = ApplyingChangesTestUtil.compareJaMoPPCompilationUnits(compUnitChanged, compUnitFromGit, virtualModel);
    //Ensure that there is a corresponding PCM model to the field.
    boolean pcmExists = ApplyingChangesTestUtil.assertFieldWithName("field", compUnitChanged, virtualModel);

    assertTrue("In testAddField() the JaMoPP-models are NOT equal, but they should be", jamoppClassifiersAreEqual);
    assertTrue("In testAddField() corresponding PCM model does not exist, but it should exist", pcmExists);
}
```

Abbildung 3.3: Test für das Hinzufügen einer Klassenvariable in einer Klasse, die in einem integrierten Bereich liegt

4 Case-Study-Ergebnisse

Für unsere Case-Study haben wir das Projekt namens 'eu.fpetersen.cbs.pc' [1] benutzt. Für dieses Projekt haben wir ein Git Repository initialisiert und Commits für unterschiedliche Arten von Änderungen erstellt. Anschließend haben wir das Projekt in Vitruvius integriert, Commits auf dem integrierten Projekt angewandt und die Korrektheit von den aktualisierten Modellen überprüft. Die Ergebnisse sind in der Tabelle 4.1 abgebildet.

Test	Subtests	JaMoPP Codebuffer	JaMoPP Modelle	PCM Modelle	Probleme
testClassAnnotation (IA)	testAddClassAnnotation	korrekt	korrekt	nicht betroffen	
	testChangeClassAnnotation	korrekt	korrekt	nicht betroffen	
	testRemoveClassAnnotation	korrekt	korrekt	nicht betroffen	
testClassHeader (IA)	testAddAbstract	korrekt	korrekt	nicht betroffen	
	testChangeAbstractToFinal	korrekt	korrekt	nicht betroffen	
	testRenameClass	nicht überprüft	nicht überprüft	nicht überprüft	Vitruv throws an exception when a class is removed. Because rename class is considered as remove class with old name and create class with new name, that test fails. In tools.vitruv.domains.java.monitorededitor. ChangeResponder.visit(DeleteClassEvent) the visit(...) method tries to get some information from the already removed JDT model.
testExtends (IA)	testAddImport	korrekt	korrekt	nicht betroffen	
	testAddExtends	korrekt	korrekt	keine CPR implementiert	
	testRemoveExtends	korrekt	korrekt	keine CPR implementiert	
testChangeField (IA)	testAddImport	korrekt	korrekt	nicht betroffen	
	testAddField	korrekt	korrekt	korrekt	
	testRenameField	korrekt	korrekt	korrekt	
	testAddFieldModifier	korrekt	korrekt	nicht betroffen	
	testChangeFieldModifier	korrekt	korrekt	nicht betroffen	
	testChangeFieldType	korrekt	nicht korrekt	nicht überprüft	ChangeFieldTypeEventRoutine does not work appropriate
	testRemoveField	nicht überprüft	nicht überprüft	nicht überprüft	remove field event is recognized as InsertEReference, but not as RemoveEReference.
testImplements (IA)	testAddImport	korrekt	korrekt	korrekt	
	testAddImplements	korrekt	korrekt	korrekt	
	testRemoveImplements	korrekt	korrekt	korrekt	
testChangeMethodHeader (IA)	testRenameMethodInInterface	korrekt	korrekt	korrekt	
	testRenameMethodInClass	korrekt	korrekt	korrekt	
	testChangeReturnTypeInInterfaceMethod	korrekt	korrekt	korrekt	
	testChangeReturnTypeInClassMethod	korrekt	korrekt	nicht betroffen	
	testAddReturn0InClassMethod	korrekt	korrekt	nicht betroffen	
	testAddFinalModifierToClassMethod	korrekt	korrekt	nicht betroffen	
	testAddMethodParameterInInterface	korrekt	korrekt	korrekt	
	testAddMethodParameterInClass	korrekt	korrekt	nicht betroffen	
testChangeMethod-Implementation (IA)	testAddInternalAction	korrekt	korrekt	korrekt	
	testAddForLoop	korrekt	korrekt	korrekt	
	testAddIfElse	korrekt	korrekt	korrekt	
	testRemoveIfElse	korrekt	korrekt	korrekt	

[1] eu.fpetersen.cbs.pc

Test	Subtests	JaMoPP Codebuffer	JaMoPP Modelle	PCM Modelle	Probleme
	testRemoveForLoop	korrekt	korrekt	korrekt	
	testRemoveInternalAction	korrekt	korrekt	korrekt	
testCreateDelete- NonJavaFile (IA)	testCreateFolderAndFile	nicht betroffen	nicht betroffen	nicht betroffen	
	testRenameFile	nicht betroffen	nicht betroffen	nicht betroffen	
	testChangeFileContent	File-Inhalt korrekt	nicht betroffen	nicht betroffen	
	testCopyFile	File-Inhalt korrekt	nicht betroffen	nicht betroffen	
	testRemoveFile	nicht betroffen	nicht betroffen	nicht betroffen	
testCreateDelete- CompilationUnit (IA)	testCreateClass	korrekt	korrekt	korrekt	
	testCreateInterface	korrekt	korrekt	korrekt	
	testRemoveClass	nicht überprüft	nicht überprüft	nicht überprüft	testRemoveClass() and testRemoveInterface() doesn't work appropriate. The problem could be in the method tools.vitruv.domains.java.monitorededitor.ChangeResponder.visit(DeleteClassEvent) and tools.vitruv.domains.java.monitorededitor.ChangeResponder.visit(DeleteInterfaceEvent). The method visit(...) tries to get some information from the already removed JDT model.
	testRemoveInterface	nicht überprüft	nicht überprüft	nicht überprüft	The same problem as discribed above
testCreateDeleteField (IA)	testAddImport	korrekt	korrekt	nicht betroffen	
	testAddField	korrekt	korrekt	korrekt	
	testRenameField	korrekt	korrekt	korrekt	
	testRemoveCreatedField	nicht überprüft	nicht überprüft	nicht überprüft	Somehow remove field event is recognized by Vitruv as InsertEReference, but not as RemoveEReference. Therefore, a wrong correspondence is created
	testRemoveIntegratedField				The same problem as discribed above
testCreateDeleteMethod (IA)	testCreateMethodInInterface	korrekt	korrekt	korrekt	
	testCreateMethodInClass	korrekt	korrekt	korrekt	
	testRemoveMethodInClass	korrekt	korrekt	korrekt	
	testRemoveMethodInInterface	korrekt	korrekt	korrekt	
testCreateDeletePackage (IA)	testCreatePackage	korrekt	korrekt	korrekt	
	testRenameCreatedPackage	korrekt	korrekt	korrekt	
	testRemoveCreatedPackage	korrekt	korrekt	korrekt	
testClassAnnotation (NIA)	testAddClassAnnotation	korrekt	korrekt	nicht betroffen	
	testChangeClassAnnotation	korrekt	korrekt	nicht betroffen	
	testRemoveClassAnnotation	korrekt	korrekt	nicht betroffen	
testClassHeader (NIA)	testRemovePublicClassModifier	korrekt	korrekt	nicht betroffen	
	testAddFinalClassModifier	korrekt	korrekt	nicht betroffen	
	testChangeFinalToAbstractClassModifier	korrekt	korrekt	nicht betroffen	
	testChangeAbstractToPublicClassModifier	korrekt	korrekt	nicht betroffen	
testExtends (NIA)	testAddFirstImportForExtends	korrekt	korrekt	nicht betroffen	
	testAddExtends	korrekt	korrekt	keine CPR implementiert	
	testAddSecondImportForExtends	korrekt	korrekt	nicht betroffen	
	testChangeExtends	korrekt	korrekt	keine CPR implementiert	
	testRemoveExtends	korrekt	korrekt	keine CPR implementiert	
	testRemoveSecondImportForExtends	korrekt	korrekt	nicht betroffen	
	testRemoveFirstImportForExtends	korrekt	korrekt	nicht betroffen	
testField (NIA)	testAddSecondClassForField	korrekt	korrekt	korrekt	
	testAddImportForField	korrekt	korrekt	nicht betroffen	
	testAddField	korrekt	korrekt	Nicht korrekt, kein PCM-Field erstellt	Die Klasse SecondClass hat kein korrespondierendes Interface =>in der Methode mir.routines.classifierBody.FieldCreatedCorrespondingToRepositoryComponentRoutine.ActionUserExecution.callRoutine1 (Classifier, Field, RepositoryComponent, RepositoryComponent, RoutinesFacade) werden keine operationProvidedRoles gefunden =>keine OperationRequiredRole wird ertellt

Test	Subtests	JaMoPP Codebuffer	JaMoPP Modelle	PCM Modelle	Probleme
	testRenameField	korrekt			
	testRemoveField	korrekt			
	testRemoveImportForField	korrekt			
testImplements (NIA)	testAddFirstInterface	korrekt	korrekt	korrekt	
	testAddMethodInFirstInterface	korrekt	korrekt	korrekt	
	testAddFirstImport	korrekt	korrekt	nicht betroffen	
	testAddImplementsAndMethod	korrekt	korrekt	korrekt	
	testAddSecondInterface	korrekt	korrekt	korrekt	
	testAddMethodInSecondInterface	korrekt	korrekt	korrekt	
	testAddSecondImport	korrekt	korrekt	nicht betroffen	
	testChangeImplementsAndAddMethod	korrekt	korrekt	korrekt	
	testRemoveImplements	korrekt	korrekt	korrekt	
	testRemoveBothMethods	korrekt	korrekt	korrekt	
	testRemoveBothImports	korrekt	korrekt	nicht betroffen	
testChangeMethodHeader (NIA)	testRenameMethodInInterface	korrekt	korrekt	korrekt	
	testRenameMethodInClass	korrekt	korrekt	korrekt	
	testChangeReturnTypeInInterfaceMethod	korrekt	korrekt	korrekt	
	testChangeReturnTypeInClassMethod	korrekt	korrekt	nicht betroffen	
	testAddReturn0InClassMethod	korrekt	korrekt	nicht betroffen	
	testAddFinalModifierToClassMethod	korrekt	korrekt	nicht betroffen	
	testAddMethodParameterInInterface	korrekt	korrekt	korrekt	
	testAddMethodParameterInClass	korrekt	korrekt	nicht betroffen	
testMethodImplementation (NIA)	testAddImport	korrekt	korrekt	nicht betroffen	
	testAddField	korrekt	korrekt	korrekt	
	testAddExternalCall	korrekt	korrekt	korrekt	
	testAddInternalAction	korrekt	korrekt	korrekt	
	testAddFor	korrekt	korrekt	korrekt	
	testAddIfElse	korrekt	korrekt	korrekt	
	testRemoveIfElse	korrekt	korrekt	korrekt	
	testRemoveFor	korrekt	korrekt	korrekt	
	testRemoveInternalAction	korrekt	korrekt	korrekt	
	testRemoveExternalCall	korrekt	korrekt	korrekt	

Tabelle 4.1: Ergebnisse der Case-Study. Abkürzungen: IA - Integrated Area, NIA - Non-Integrated Area, CPR - Change Propagation Rules.

Literatur

[1] Michael Langhammer. “Automated Coevolution of Source Code and Software Architecture Models”. Diss. Karlsruher Institut für Technologie, 2017.

[2] Manar Mazkatli und Anne Koziolek. “Continuous Integration of Performance Model”. In: (2018).

[3] Manar Mazkatli u. a. “Incremental Calibration of Architectural Performance Models with Parametric Dependencies”. In: (2020).