Konzeption universeller .NET-Prüfkomponenten für das E-Assessment-System JACK

Mario D’Amico

Abstrakt

Das E-Assessment-System JACK der Universität Duisburg-Essen bietet eine server-basierte, interaktive Übungs- und Prüfungsplattform zur Unterstützung von Programmier-Lehrveranstaltungen an. Mittels modularer sprachspezifischer Prüfkomponenten (Checker) können automatisiert Quellcodedateien korrespondierender Programmiersprachen auf ihr Laufzeitverhalten und ihre interne Struktur untersucht werden. Durch die Ausnutzung technischer Fähigkeiten des .NET-Frameworks war es möglich, Prüfkomponenten für das JACK-System zu entwickeln, welche nicht an eine Programmiersprache gebunden sind, sondern Prüfungen von Quellcodedateien aller Programmiersprachen des .NET-Frameworks ermöglichen und selbst multilinguale .NET-Applikationen untersuchen können.

1. Einleitung

Seit dem Wintersemester 2006/2007 nutzt die Universität Duisburg-Essen das E-Assessment-System JACK (Java Checker) als server-basierte, interaktive Übungs- und Prüfungsplattform zur Unterstützung der Java-Lehrveranstaltung für Erstsemester. Das System ist durch integrierte Prüfkomponenten in der Lage Java-Quellcodedateien auf ihr Laufzeitverhalten und ihre interne Struktur bezüglich vorgegebener Anforderungen zu testen. Wegen seiner modularen Architektur kann JACK durch weitere Prüfkomponenten ergänzt und auf diesem Weg in unterschiedlichste Lehrveranstaltungen als unterstützendes Medium integriert werden. Die notwendigen Anpassungsarbeiten zur Adaption des Systems bestehen in der Entwicklung und Integration sprachspezifischer Checker-Komponenten [GSB08].

Alle bisherigen in JACK integrierten Prüfkomponenten fokussieren stets eine Programmiersprache. Im Gegensatz dazu ermöglichen technische Merkmale des von Microsoft entwickelten .NET-Frameworks es, Checker zu entwickeln, welche universell Lösungen aller Programmiersprachen des Frameworks kontrollieren können. Zudem werden Lösungen akzeptiert, die in mehreren Programmiersprachen generiert wurden. Die

Konzeptionierung und Integration derartiger Prüfkomponenten würde folglich das JACK-System für Lehrveranstaltungen sämtlicher Programmiersprachen des .NET-Frameworks qualifizieren.

Dieses Paper beschreibt die Konzeptionen zweier solcher universell einsetzbarer .NET-Prüfkomponenten, die im Rahmen eines Projektes für das E-Assessment-System JACK entwickelt und prototypisch implementiert wurden. In Kapitel 2 wird zu diesem Zweck auf das JACK-System näher eingegangen. Das Kapitel 3 befasst sich mit Mechanismen des .NET-Frameworks, welche die sprachunabhängige Nutzung der .NET-Prüfkomponenten ermöglichen. Die Kapitel 4 und 5 beschreiben schließlich die beiden Konzepte der sprachunabhängigen .NET-Checker. Das Paper endet mit einem Fazit und Ausblick in Kapitel 6.

2. Das E-Assessment-System JACK

JACK stellt Studierenden eine umfangreiche Übungs- und Prüfungsplattform zur Verfügung. Über ein systemeigenes Webportal können sie auf Java-Übungsaufgaben zugreifen, welche die praktische Umsetzung der in der Vorlesung behandelten theoretischen Inhalte fordern. Lösungen werden vom System auf syntaktische Richtigkeit und korrektes Laufzeitverhalten bezüglich der in der Aufgabe beschriebenen Inhalte überprüft. Im Falle eines Fehlers generieren die Prüfkomponenten fehlerbeschreibende Kommentare, welche die Studierenden in der selbstständigen Findung der korrekten Lösung unterstützen sollen. Des Weiteren ermöglicht JACK Lehrenden die Erstellung und Verwaltung von vorlesungsbegleitenden Prüfungen und Testaten, an denen Studierende über Systemschnittstellen teilnehmen können. JACK realisiert somit eine interaktive Übungs- und Prüfungsumgebung, die mit wenig administrativen Personalressourcen ein Lehrangebot für eine große Menge von Studierenden ermöglicht [GSB08].

Eingereichte Lösungen analysiert das System anhand zweier unterschiedlicher Prüfungsverfahren. Zum einen durchlaufen sie statische Analysen, die sich mit der inneren Struktur ihres Quellcodes befassen. Zum anderen werden Lösungen dynamisch geprüft, indem das Programmverhalten während der Laufzeit mit erwarteten Mustern verglichen wird. Zur vollständigen Analyse einer eingereichten Lösung werden beide Prüfungsverfahren durch unterschiedliche, autark agierende Checker-Komponenten durchgeführt [St14].

Die in diesem Paper vorgestellten Konzepte der .NET-Checker realisieren sowohl die statische als auch die dynamische Analyse von .NET-Applikationen. Zur Erläuterung ihrer multilingualen Einsatzfähigkeit soll im Kapitel 3 ein kurzer Abriss struktureller und technischer Eigenschaften des .NET-Frameworks erfolgen.

3. Sprachübergreifende Unterstützung des .NET–Frameworks

Im Jahr 2002 stellte Microsoft das .NET-Framework als eine umfangreiche funktionserweiternde Entwicklerplattform für das eigene Betriebssystem Windows vor. Grundlegende Ziele bei der Entwicklung waren die Gewährleistung der Interoperabilität der diversen Microsoft-Programmiersprachen und Plattformunabhängigkeit [Be06].

Zur Unterstützung der Interoperabilität interner Programmiersprachen definiert das .NET-Framework einen von jeder Sprache einzuhaltenden minimalen Regelsatz (Common Language Spezifikation CLS) bezüglich Deklarations-, Verwaltungs- und Verwendungsregeln seiner Datentypen. Jede Programmiersprache, welche den Regelsatz der CLS erfüllt, ist in der Lage, im .NET-Framework interoperabel mit anderen Sprachen zu agieren. Folglich unterstützt das Framework .NET-Applikationen mit internen Komponenten, die in unterschiedlichen Programmiersprachen erstellt wurden [Be06].
Zur Programmausführung stellt das Framework eine virtuelle Laufzeitumgebung (Common Language Runtime CLR) bereit, welche mit der Common Intermediate Language (CIL) eine eigene Befehlssatzsprache besitzt. CIL-Code beschreibt eine objektorientierte, vollständig Stack-basierte Zwischensprache, in welche jeder Quellcode, der in einer Programmiersprache der .NET-Sprachfamilie generiert wurde, vor seiner Ausführung übersetzt wird. Den Übersetzungsprozess übernehmen sprachspezifische, im .NET-Framework integrierte Compiler. Das Resultat des Kompilierungsprozesses ist eine Datei, welche zusätzlich zum CIL-Code weitere für die Ausführung relevante Informationen, wie z.B. Metadaten beinhaltet. Eine solche Datei wird Assembly genannt [Be06].

Die beschriebenen Aspekte verdeutlichen, dass nach dem Kompilierungsprozess alle .NET-Applikationen für die CLR gleich aussehen, unabhängig davon, in welcher Programmiersprache oder in welchen Kombinationen von Programmiersprachen sie generiert wurden. In Folge dessen muss eine dynamische Analyse von .NET-Applikationen stets sprachunabhängig sein. Für die statische Untersuchung liefert der CIL-Code zudem eine universelle Grundlage, anhand derer syntaktische Eigenschaften der .NET-Applikationen kontrolliert werden können.

Die in den folgenden Kapiteln beschriebenen Konzeptionen bauen auf diesen Erkenntnissen auf. Eine zentrale Rolle spielen dabei Assemblies, die aus der Menge der eingereichten Lösungsdateien generiert werden müssen. Die dynamische .NET-Prüfkomponente betrachtet dabei das Laufzeitverhalten der Assemblies und vergleicht deren Ausgaben mit Musterlösungen. Die statische .NET-Prüfkomponente fokussiert Strukturen des CIL-Codes der Assemblies, um Informationen über den inneren Aufbau der ursprünglichen Lösungsdateien zu erhalten.

4. Konzeption der universellen dynamischen .NET-Prüfkomponente

Um Laufzeitprüfungen von .NET-Applikationen zu realisieren, nutzt die dynamische .NET-Prüfkomponente das aus der Softwareentwicklung bekannte Unit-Test Verfahren. [Os10, S.24] definiert Unit-Tests folgendermaßen:

„Ein Unit-Test ist ein Stück Code (meist eine Methode), das ein anderes Stück Code aufruft und anschließend die Richtigkeit einer oder mehrerer Annahmen überprüft. Falls sich die Annahmen als falsch erweisen, ist der Unit-Test fehlgeschlagen. Eine Unit ist eine Methode oder Funktion.“

Zur Durchführung der Tests nutzt die Prüfkomponente das NUnit-Framework, das eine Unit-Test-Plattform für alle .NET-Programmiersprachen realisiert und deren Interoperabilität unterstützt [Os10].

Die Grundlage für die Ausführung von NUnit-Tests bilden Assemblies. Diese müssen neben NUnit-Testdefinitionen ebenfalls Informationen über den zu testenden Quellcode beinhalten. Folglich muss die Prüfkomponente aus der Menge der vom Studierenden eingereichten Lösungsdateien und der vom Lehrenden erstellten aufgabenspezifischen NUnit-Testdatei ein Gesamtprojekt generieren. Zu diesem Zweck stellt die Komponente einen Rahmen aus sich gegenseitig referenzierenden, in unterschiedlichen .NET-Sprachen generierten, leeren Bibliotheksprojektmappen (Containerbibliotheken) bereit, in welche sprachgleiche Lösungsdateien und NUnit-Testdefinitionen abgelegt werden. Durch Kompilierung dieses Rahmenwerks wird ein Gesamtprojekt aus sich gegenseitig referenzierenden Assemblies generiert, auf dessen Basis alle NUnit-Tests ausgeführt werden können.

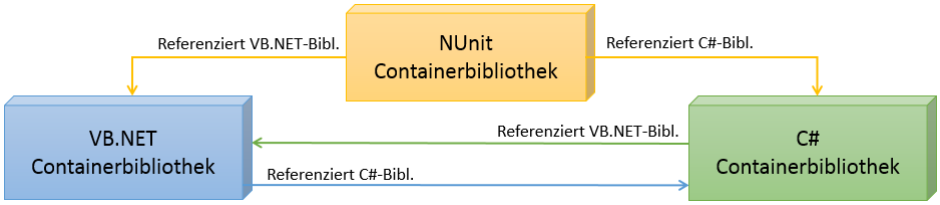

Abb. 1: Infrastruktur zur Realisierung von NUnit-Tests, die auf VB.NET und C# Quellcode-Dateien angewendet werden können

Abb. 1 visualisiert die konzeptuelle Architektur des Bibliothek-Rahmenwerks. Das zentrale Herzstück bildet eine in einer beliebigen .NET-Sprache realisierte NUnit-Bibliothek, in welche ausschließlich NUnit-Testdefinitionen abgelegt werden. Die Testdefinitionen müssen stets der gleichen Sprache entsprechen, in welcher auch die NUnit-Bibliothek generiert wurde. Die Gewährleistung der Interoperabilität der Programmiersprachen im .NET-Framework ermöglicht in dieser Konfiguration zwei

essentielle Features, welche die sprachunabhängige Nutzung der Prüfkomponente realisieren: Zum einen ist es möglich, NUnit-Testdefinitionen auf anderssprachige .NET-Quellcodedateien anzuwenden. Somit können sich Testmethoden der NUnit-Bibliothek auf Objekte aller Klassen der referenzierten, programmiersprachen-spezifischen Containerbibliotheken beziehen. Zum anderen erlaubt das gegenseitige Referenzieren der sprachspezifischen Containerbibliotheken untereinander, dass multilinguale Lösungen aufgenommen und geprüft werden können. Existiert folglich für jede Programmiersprache des .NET-Frameworks eine korrespondierende Containerbibliothek im Bibliotheksrahmenwerk der Prüfkomponente, können auf diesem Weg Laufzeittests auf Lösungen angewendet werden, welche in einer oder beliebig vielen .NET-Sprachen generiert wurden.

Die Testausführung vollzieht die dynamische .NET-Prüfkomponente mit Hilfe des Test-Runner-Tools NUnit-Console.exe[1]. Dem Tool wird das Assembly der NUnit-Bibliothek übergeben, worauf es alle in der Bibliothek abgelegten Testdefinitionen auf die Inhalte der involvierten Containerbibliotheks-Assemblies anwendet. Die Testergebnisse dokumentiert das Tool in einer XML-Datei, deren Inhalt schlussendlich die Grundlage für die Rückmeldung an den Studierenden bildet.

5. Konzeption der universellen statischen .NET-Prüfkomponente

Neben der Betrachtung des Laufzeitverhaltens verlangt eine vollständige Analyse der eingereichten Lösungsdateien ebenfalls die Kontrolle inneren Programmstrukturen. Um dieses sprachübergreifend durchführen zu können, fokussiert die statische .NET-Prüfkomponente Strukturen des zum Lösungscode korrespondierenden CIL-Codes. Mit Hilfe der Abb. 2 sollen beispielhaft einige charakteristische Muster des CIL-Codes verdeutlicht werden, anhand derer Eigenschaften des internen Aufbaus der eingereichten Quellcodedateien abgeleitet werden können.
Auf der linken Seite der Abb. 2 befindet sich die C#-Implementierung einer Klasse, zur Rechten ein Ausschnitt ihres korrespondierenden CIL-Codes. In den Zeilen 1 und 2 des CIL-Codes ist zu erkennen, dass sowohl Rückschlüsse auf Benennungen und Zugriffsmodifikationen als auch auf die Hierarchie der integrierten Klassen abgeleitet werden können. Des Weiteren beschreiben die Zeilen 4 und 5 des CIL-Codes, dass Variablendeklarationen inklusive Benennung, Datentyp und Zugriffsmodifikation vorhanden und somit überprüfbar sind. Gleiches gilt, wie in Zeilen 6 und 7 ersichtlich, für Methodendeklarationen, welche zusätzlich Informationen über Rückgabewerte liefern.

[1] NUnit-Console.exe ist ein textbasierter NUnit-Test-Runner, der standardmäßig vom NUnit-Framework angeboten wird.

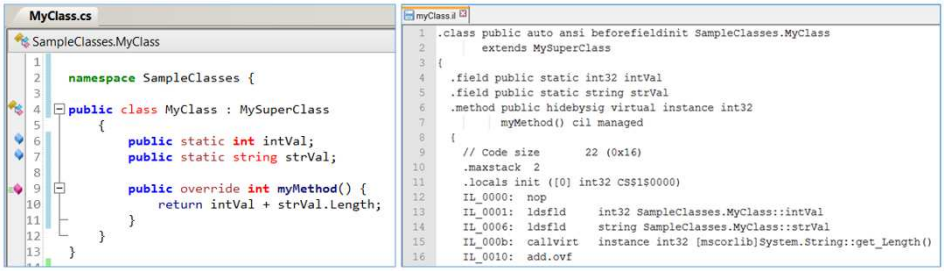

Abb. 2: Eine C# Klasse und ein Ausschnitt ihres korrespondierenden CIL-Codes

Der für die statische Prüfung benötigte CIL-Code wird aus den Assemblies einer Containerbibliotheks-Infrastruktur gewonnen, welche analog zum dynamischen auch im statischen .NET-Checker integriert ist. Der Extrahierungsprozess des CIL-Codes aus den Assemblies wird mittels des Tools Ildasm.exe[2] vollzogen. Zur syntaktischen Untersuchung wird im Anschluss der extrahierte CIL-Code aller Assemblies zusammengeführt und applikationsübergreifend in die formale Repräsentation eines abstrakten Syntaxbaumes (AST) übersetzt. Der erforderliche AST-Parser wurde mittels des ANTLR-Frameworks[3] generiert. Er basiert auf einer regulären CIL-Grammatik, welche der CIL-Spezifikation der zweiten Edition des ECMA-335 Standards [ECMA02] folgt und auf der ANTLR-Webseite frei zur Verfügung gestellt wird.

Zur Überprüfung struktureller Eigenschaften der CIL-AST-Repräsentation wird die Graph-Anfragesprache GReQL2 (**G**raph **Re**pository **Q**uery **L**anguage 2) verwendet. Sie besitzt eine SQL-ähnliche Syntax und ermöglicht durch Abfragen und reguläre Pfadausdrücke, auf die Eigenschaften und Strukturen von TGraphen zuzugreifen. Ein TGraph entspricht einer gerichteten, typisierten, attribuierten und geordneten Graph-Repräsentation [Ma06]. Folglich muss der CIL-AST in einen TGraphen übersetzt werden. Grundlage für diese Umwandlung ist ein Schema, welches sämtliche potentiell auftretenden Knoten und Kanten eines CIL-ASTs erfasst und Typen zuordnet. Auf Basis dieses Schemas und der Funktionen der JGraLab-Bibliothek[4] können automatisiert Klassen sämtlicher beschriebener Knoten- und Kantentypen generiert werden, mit deren Hilfe für jedes Element des ASTs Objekte erzeugt und einem TGraphen übergeben werden können [Ka06]. Alle Eigenschaften des CIL-Codes, die Konklusionen auf den internen Aufbau seines ursprünglichen Quellcodes erlauben, sind im resultierenden TGraphen durch attribuierte Knoten und Kanten erfasst und können mittels GReQL2-Anfragen untersucht werden. Auf diesem Weg können übergreifend für alle Programmiersprachen der .NET-Sprachfamilie statische Prüfungen realisiert werden.

[2] Ildasm.exe ist ein standardmäßig im .NET-Framework integriertes CIL-Disassembler-Tool.
[3] ANTLR (**An**other **T**ool for **L**anguage **R**ecognition) beschreibt eine von der Universität San Francisco entwickelte Plattform, die es ermöglicht, sprachspezifische AST-Parser zu generieren [Pa07].
[4] Die JGraLab-Bibliothek ist eine von der Universität Koblenz erstellte Java-API, die Klassen und Funktionen für das Erstellen und Verarbeiten von TGraphen bereitstellt [Ka06].

6. Fazit und Ausblick

Durch Verwendung von Mechanismen des .NET-Frameworks sind die vorgestellten Prüfkomponenten in der Lage, sowohl das Laufzeitverhalten als auch den internen Aufbau von .NET-Applikationen zu überprüfen. Ihre Implementierung und Integration ermöglichen, JACK als sprachunabhängiges E-Assessment-System im Rahmen von Lehrveranstaltungen, welche sich mit den Programmiersprachen der .NET-Sprachfamilie befassen, einzusetzen.

Die dynamische .NET-Prüfkomponente liefert dabei die gleiche Informationstiefe wie sprachspezifische dynamische Komponenten, deren Entwicklung zudem wenig Sinn ergeben würde, da die Ausführung von .NET-Applikationen stets über die CLR abgewickelt wird und diese keine andere Sprache kennt als die CIL.
Kritisch zu betrachten ist der Black-Box-Charakter der vorgestellten dynamischen Prüfung. Die Unit-Tests liefern keine Informationen darüber, wie Ausgaben der getesteten Assemblies erzeugt werden. Folglich eruieren sie im Falle des Auftretens von Unregelmäßigkeiten nicht den Grund oder den Zeitpunkt ihres Erscheinens. Um die didaktische Aussagekraft der JACK-Rückmeldungen zu steigern, sollte die dynamische Betrachtung durch Funktionalitäten zur Ablaufverfolgung (Tracing) der Quellcodeausführung erweitert werden, um im Falle des Auftretens von Unregelmäßigkeiten die Fehlerquelle gezielt identifizieren zu können.

Im Gegensatz zur dynamischen kann die statische .NET-Prüfkomponente nicht die gleiche Informationstiefe liefern wie es vergleichsweise sprachspezifische statische Komponenten ermöglichen. Nicht alle Strukturen des Lösungsquellcodes lassen sich über die syntaktische Analyse seines CIL-Codes ableiten. So ist es z.B. nicht möglich, im CIL-Quellcode zu ermitteln, ob eine Iteration mit Hilfe einer *while*- oder *for*-Schleife realisiert wurde. Die Kontrolle derartiger Strukturen kann folglich nicht auf Basis von CIL-Code erfolgen, sondern muss die Syntax des ursprünglichen Quellcodes beleuchten.
Ihren Mehrwert erlangt die statische .NET-Prüfkomponente durch ihre applikationsübergreifende Perspektive. Dadurch, dass alle Klassen einer multilingualen .NET-Applikation in einem Graphen dargestellt und kontrolliert werden können, leistet der statische .NET-Checker Analysearbeiten, die von sprachspezifischen statischen Prüfkomponenten und selbst Kombinationen von ihnen nicht erbracht werden können. Folglich beschreibt die syntaktische Betrachtung des CIL-Codes eine notwendige Prüfung, um applikationsumfassende Strukturen von multilingualen .NET-Programmen zu kontrollieren. Um letztendlich auch in allen Betrachtungsfeldern die notwendige Informationstiefe zu erlangen, empfiehlt sich eine Kombination aus CIL- und sprachspezifischer Quellcodebetrachtung.
Die Art der syntaktischen Prüfung sprachspezifischen Quellcodes kann analog mittels TGraphen und der GReQL2 Anfragesprache realisiert werden. In Folge dessen bestehen die sprachbedingten Unterschiede im Prüfungsverlauf ausschließlich in der Generierung entsprechender abstrakter Syntaxbäume, deren Transformation in TGraphen und der Formulierung und Anwendung sprachspezifischer GReQL2-Regeln. In Zusammenarbeit

mit dem statischen .NET-Checker sind folglich umfassende syntaktische Prüfungen von multilingualen .NET-Programmen möglich, die sowohl applikationsübergreifende Strukturen prüfen können als auch sprachspezifische Aspekte der Applikationen syntaktisch beleuchten, welche über den CIL-Code nicht betrachtet werden können.

Literaturverzeichnis

[Be06] Beer, Wolfgang; Birngruber, Dietrich; Mössenböck, Hans-Peter; Prähofer, Herbert; Wöß, Albrecht. Die .NET-Technologie, Grundlagen und Anwendungsprogrammierung. Dpunkt.Verlag, Heidelberg, 2006.

[ECMA02] Standard ECMA-335, Common Language Infrastructur (CLI), 2nd Edition, Dezember 2002.

[GSB08] Goedicke, M.; Striewe, M.; Balz, M.: Computer Aided Assessments and Programming Exercises with JACK. ICB-Research Report No.28, Essen 2008.

[Ka06] Kahle, Steffen. JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen. Diplomarbeit, Koblenz 2006.

[Ma06] Marchewka, Katrin. Entwurf und Definition der Anfragesprache GReQL2. Diplomarbeit, Koblenz 2006.

[Os10] Osherove, Roy. The Art of Unit Testing (Deutsche Ausgabe). Mitp-Verlag, Heidelberg 2010.

[Pa07] Parr, Terence. The Definitive ANTLR Reference. Dallas, Texas 2007 und http://www.antlr3.org, Stand: 02.08.2015.

[St14] Striewe, Michael. Automated Analysis of Software Artefacts – A Use Case in E-Assessment. Dissertation, Essen, 2014.