Trace-Vergleich zur Feedback-Erzeugung im automatisierten E-Assessment-System JACK

Christoph Olbricht[1]

Abstract: An der Universität Duisburg-Essen wird zur automatischen Korrektur von Programmieraufgaben seit mehr als zehn Jahren erfolgreich das E-Assessment-System JACK verwendet. Bisher war das Feedback von JACK auf vordefinierte Hinweise beschränkt, welche in den statischen und dynamischen Tests formuliert wurden. Für die Prüfung der Einsatztauglichkeit von Trace-Vergleichen zur Feedback-Erzeugung in JACK wurden Traces studentischer Lösungen mit Musterlösungen der Aufgaben automatisch verglichen. Hierzu musste zunächst mittels statischer Checks eine passende Musterlösung gewählt werden. Ziel war die Erzeugung eines Feedback-Markers zur Fehlererkennung, um den Studierenden erweitertes Feedback zu bieten. Es wurde festgestellt, dass bestimmte Anforderungen an Länge und Struktur der Aufgaben erfüllt sein müssen, damit der Algorithmus Feedback-Marker erzeugt.

Keywords: E-Assessment, automatische Bewertung, Trace-Vergleich, JACK

1 Einleitung

An der Universität Duisburg-Essen wird zur automatischen Korrektur von Programmieraufgaben seit mehr als zehn Jahren [GS17] erfolgreich das E-Assessment-System JACK verwendet. Die JACK-Architektur, detailliert in [St16] beschrieben, stellt asynchrone Checker-Komponenten bereit. Mittels der asynchronen Checker werden Programmier- und Modellierungsaufgaben ausgewertet. Dem Lehrenden ist es möglich Aufgaben zu erstellen und diese mit statischen und dynamischen Checkern zu versehen. Eingereichte Lösungen werden mittels dieser Checker auf Korrektheit überprüft. Ein statischer Checker überprüft mittels der Anfragensprache GReQL [Gr19] den Quellcode auf vorhandene oder abwesende Programmstrukturen. Es kann sowohl überprüft werden, ob die bestehenden Konventionen der Programmierung eingehalten wurden, als auch, ob unerwünschte Programmkonstrukte verwendet wurden. Ein dynamischer Checker führt die eingereichte Lösung mit mehreren Testfällen aus und prüft das Programm so auf funktionale Korrektheit. Übliche Programmierfehler können mit gezielten Testfällen erkannt werden, so dass dem Studierenden spezifisches Feedback gegeben werden kann. Tritt ein Fehler auf, wird dem Studierenden zudem ein Programm-Trace der Ausführung präsentiert.

Das Feedback dieser Checker ist auf vordefinierte Hinweise beschränkt. Wird ein Fehler durch keinen für diesen Fehlertyp erstellten Testfall abgedeckt, sondern lediglich durch

[1] University of Duisburg-Essen, paluno - The Ruhr Institute for Software Technology, Gerlingstraße 16, 45127 Essen, christoph.olbricht@paluno.uni-due.de

doi:10.18420/abp2019-2

ein falsches Ergebnis erkannt, kann folglich auch nur eine allgemeine Fehlermeldung als Feedback gegeben werden. Dieser Hinweis hilft dem Studierenden nicht bei der Fehlerfindung. Bisher gab es für diese Fälle keine weiteren Hilfestellungen.

Mit Hilfe des Trace-Vergleichs einer studentischen Lösung mit einer passenden Musterlösung des Lehrenden kann erweitertes Feedback geliefert werden. Der Feedback-Marker zeigt die Codezeile auf, ab der die Traces immer stärker voneinander abweichen, da sich in dieser Zeile mit hoher Wahrscheinlichkeit ein Programmierfehler befindet. Somit kann dem Studierenden eine Hilfestellung gegeben werden, um den Fehler zu identifizieren. Eine solche Hervorhebung möglicher Fehler wurde in [Gr17] laut Experten als nützlich angesehen.

Im Rahmen einer Abschlussarbeit sollte die Einsatztauglichkeit der Feedback-Erzeugung dieser Trace-Vergleiche im System JACK geprüft werden. Die Ergebnisse dieser Arbeit werden hier vorgestellt. Es konnten Erkenntnisse über den Aufwand zur Abdeckung des Lösungsraums gewonnen werden. Zudem wurden Anforderungen an die Aufgaben erarbeitet, die erfüllt sein müssen, damit die Feedback-Marker zuverlässig erzeugt werden können.

2 Trace-Vergleich

Traces können vielseitig Verwendung finden. In [ZY18] wurden aus dynamischen Testfällen gewonnene Traces genutzt, um Regeln der Programmierung zu gewinnen. In [Vi03] wurden zum „runtime model checking“ in Java Traces automatisch generiert und analysiert. Zur Verbesserung der Lehre wird in [GH17] die manuelle Erstellung von Traces von Studierenden als Übung zum besseren Verständnis von Programmieraufgaben besprochen. Eine werkzeugunterstützte automatische Korrektur von Programmierfehlern mittels Trace-Vergleich für erweitertes Feedback wird in [Su17] vorgestellt.

Beim Trace-Vergleich zwischen der studentischen Lösung und der Musterlösung des Lehrenden muss zunächst sichergestellt werden, dass beide Lösungen denselben Lösungsansatz besitzen. Weichen die Lösungen zu sehr in ihrem Vorgehen voneinander ab, kann kein sinnvoller Trace-Vergleich stattfinden. Um dieses Problem zu lösen, wurden mehrere Musterlösungen für die vorhandenen Aufgaben erstellt. Mittels statischer Codeprüfung konnte der studentischen Lösung eine ähnliche Musterlösung zugeordnet werden. Eine automatische Zuordnung passender Musterlösungen zu studentischen Lösungen und die daraus resultierenden Herausforderungen wird in [Gr17] besprochen. Die Möglichkeit mittels Syntaxanalyse und Programmtransformation von einzelnen Musterlösungen viele Lösungsvarianten zu erzeugen und diese mit normalisierten studentischen Lösungen zu vergleichen wird in [GJH12] besprochen.

Der vorhandene Algorithmus von Ukkonen [Uk85] vergleicht alle Variablen beider Lösungen miteinander, um eine korrekte Zuordnung der Variablen zu erreichen. Besitzt eine Lösung mehr Variablen, wird die kleinere Variablenmenge betrachtet. Die

Kombination aller Variablen mit der höchsten Übereinstimmung in Variablen-typ und –wert wird für die folgenden Schritte verwendet. Es folgt eine Bewertung jedes einzelnen Schritts der Traces. Sind die gleichen (zuvor zugeordneten) Variablen mit identischem Wert im Schritt beider Traces vorhanden, führt dies zu einer Erhöhung des *Alignment*-Werts. Wird in einem Schritt eine Diskrepanz festgestellt, wird der *Alignment*-Wert reduziert. Eine ausführliche, technische Beschreibung des Verfahrens ist in [St15] nachzulesen.

Ein Schritt, in dem das *Alignment* reduziert wird, wird als *candidate step* vorgemerkt. Sinkt der Wert des *Alignments* für einen Schritt und steigt anschließend wieder, wird von einer weniger performanten Lösung ausgegangen und der *candidate step* verworfen. Sinkt der Wert dauerhaft, muss von einem Fehler in der studentischen Lösung ausgegangen werden und der *candidate step* wird mit einem Feedback-Marker versehen. Das Verhältnis von *candidate steps* zur Länge des Trace wird als *average match* bezeichnet. Der *average match* dient als Kenngröße, ob studentische Lösung und Muster ähnlich zueinander sind. Er kann Werte von 0 bis 1 annehmen und bietet somit eine prozentuale Bewertung der Ähnlichkeit beider Lösungen zueinander. Liegt der Wert unter 0.8, muss von unterschiedlichen Lösungsansätzen ausgegangen werden, womit die Erzeugung eines Feedback-Markers als zufällig anzusehen ist. Bei einem Wert von 1 wurde dagegen kein *candidate step* gefunden, womit auch kein Marker erzeugt werden konnte. Dies lässt jedoch keine Aussage über die Korrektheit des Trace zu, da auch die Möglichkeit besteht, dass keine verarbeitenden Variablen einander zugeordnet wurden und somit ein Fehler nicht erkannt werden konnte.

Da ein *candidate step* im nächsten Schritt bestätigt oder widerlegt werden muss, können bestimmte Fehler nicht durch den Trace-Vergleich aufgezeigt werden. Hierzu gehören Exceptions und Fehler, welche im letzten Schritt des Trace auftreten. Außerdem können Aufgaben, die lediglich logische Abfragen verlangen nicht ausgewertet werden, da der Algorithmus Variablen und ihre Belegung vergleicht. Folglich werden verarbeitende Variablen in den Aufgaben benötigt.

3 Vorgehen

Durch die langjährige Nutzung von JACK standen die anonymisierten Lösungen der Übungsaufgaben aller Studierenden aus den letzten Semestern zur Verfügung. Nach ersten Tests fand eine Bewertung der vorhandenen Aufgaben statt, so dass im Rahmen der Abschlussarbeit vier Aufgaben im Detail ausgewertet werden konnten. Alle weiteren Aufgaben stellten sich leider als unbrauchbar heraus, da sie bestimmte Anforderungen des Algorithmus zur Feedback-Erzeugung nicht erfüllten. Von jedem Studierenden mit fehlerhaften Lösungen wurde eine dieser Lösungen, welche wenigstens ein Drittel der Maximalpunktzahl erreicht hatte zufällig ausgewählt. Lösungen mit weniger Punkten waren für den Trace-Vergleich unbrauchbar, da sie keine ernsthaften Lösungsversuche darstellten und somit keine vergleichbaren Musterlösungen existieren konnten.

Insgesamt wurden 160 studentische Lösungen im Detail ausgewertet. Es lagen keine Musterlösungen zu den bereitgestellten Aufgaben vor, daher wurden diese aus den korrekten Lösungen der Studenten ausgewählt. Mittels der Anfragensprache GReQL [Gr19] wurden Auswahlregeln aufgestellt, um den studentischen Lösungen Musterlösungen mit gleichem Ansatz zuzuordnen.

Für jede Lösung wurde der Wert des *Alignments*, sowie der *average match* notiert und geprüft, ob ein Feedback-Marker erzeugt wurde. Anschließend wurde geprüft, ob für die Lösung ein passendes Muster vorlag, oder für einen sinnvollen Trace-Vergleich ein weiteres Muster mit passendem Ansatz notwendig war. Lag ein erzeugter Marker vor wurde überprüft, ob der Marker die Zeile des Fehlers hervorhob. War dies der Fall wurde der Marker als „korrekt“ erfasst. Abschließend wurden die Fehler der Lösung ausgewertet und notiert.

In der ersten Auswertung wurden alle Lösungen einer Aufgabe mit einer Musterlösung verglichen. Anschließend wurde überprüft, ob mehr als 90% der Lösungen einen *average match* von mindestens 0.8 erreicht hatten und somit der Lösungsraum abgedeckt war. War dies nicht der Fall wurde eine zweite Musterlösung hinzugenommen und eine zweite Auswertung vorgenommen. Dieses Vorgehen sollte fortgesetzt werden, bis der Lösungsraum abgedeckt war, oder durch weitere Musterlösungen keine Verbesserung eintrat. Eine hundertprozentige Abdeckung des Lösungsraums ist aufgrund ungewöhnlicher Fehler und Lösungsansätze mittels Musterlösungen nicht zu erreichen. [RK14]

4 Resultate

Die vier Aufgaben waren in ihrem Aufbau sehr ähnlich. Sie besaßen zwei Methoden und ein Array mit bereitgestellten Daten. Es konnten 182 Fehler verzeichnet werden. Diese Fehler wurden zur besseren Übersicht manuell in Fehlertypen gruppiert. Die Anzahl an unterschiedlichen Fehlertypen reichte von fünf bis zwölf vertretenen Fehlertypen in den einzelnen Aufgaben. Die sieben häufigsten Fehlertypen machten 52 % aller Fehler aus. In 7,5 % der Fälle wurde ein Marker erzeugt.

4.1 Voraussetzungen für Feedback-Marker

Die niedrige Rate, mit der der Algorithmus einen Feedback-Marker erzeugt hat, ließ sich bei der Auswertung der Daten auf bestimmte Anforderungen an die Aufgaben zurückführen. Um einen Feedback-Marker zu erzeugen, müssen wenigstens zwei verarbeitende Variablen im Programmcode vorhanden sein. Dies konnte an zwei Aufgaben nachgewiesen werden. Die Aufgabenstellung forderte einen minimalen bzw. maximalen Wert aus einem Array herauszusuchen. In einer der Aufgaben wurde eine weitere Variable benötigt, um die Position der Zahl im Array festzuhalten. Bei dieser Aufgabe wurden verlässlich Feedback-Marker erzeugt, während die Aufgabe mit nur einer verarbeitenden Variable keine Marker erzeugte.

Weiterhin ist bei einfachen Variablen eine Mindestlänge des Trace von sechs Schritten notwendig, damit ein *Alignment* stattfinden kann. Werden die Daten mittels eines Arrays verarbeitet sind mehr Schritte notwendig, um ein *Alignment* zu erreichen. Der exakte Grenzwert ist davon abhängig, wie häufig und in welcher Art die Daten des Arrays verändert werden. Der Grenzwert ist nicht trivial bestimmbar und konnte im Rahmen der Abschlussarbeit nicht erfasst werden. In einer Aufgabe reichten bereits elf Schritte, während bei einer anderen Aufgabe 42 Schritte notwendig waren.

Letztlich müssen die Testfälle der Aufgaben einzeln erstellt werden, damit sie durch den Trace-Vergleich sinnvoll verarbeitet werden können und für den Studierenden einen Mehrwert darstellen. Werden die Testfälle durch eine Schleife aneinandergereiht und in einem Stück verarbeitet, wird die Ausgabe für den Studierenden unübersichtlich und der Trace-Vergleich liefert keine sinnvollen Ergebnisse.

In einer Aufgabe, die diese Anforderungen erfüllt wurde in 25,8 % der Lösungen ein Marker erzeugt. Wird nur die Methode betrachtet, welche die Anforderungen erfüllte, liefert der Algorithmus in 42 % der Lösungen einen Marker. Zudem waren 100 % der Marker korrekt, haben also die Zeile des vorhandenen Fehlers markiert.

4.2 Lösungsraum der Aufgaben

Die Abdeckung des Lösungsraums einer Aufgabe konnte mit maximal drei Musterlösungen erreicht werden (vgl. Tab. 1). Bei der Aufgabe „Aufgaben mit int Arrays“ wurde keine bessere Abdeckung erreicht, da die Lösungen die notwendige minimale Trace-Länge unterschritten.

Aufgabe	**1 Muster**	**2 Muster**	**3 Muster**
Aufgaben mit int Arrays	62 %	72 %	76 %
Elemente eines Arrays vertauschen	56 %	77 %	93 %
Testat Gruppe 1 WS18	98 %	/	/
Arrays und Schleifen	90 %	/	/

Tab. 1: Lösungsraumabdeckung

Warum bereits drei Muster zur Abdeckung des Lösungsraums ausreichen lässt sich nur vermuten. Zum einen sind die Aufgaben mit dem Ziel erstellt worden, dass sie durch JACK auswertbar und mit wenigen Testfällen überprüfbar sind. Dies führt zu kurzen Methoden, die weniger Raum für starke Abweichungen bieten. Zum andern stammen alle Lösungen von Studierenden, welche dieselbe Vorlesung besucht haben und demnach einen ähnlichen Programmierstil erlernen. In [RK14] wird dies ebenfalls als Grund genannt, warum der Lösungsraum nicht unendlich groß ist. Es ist davon auszugehen, dass diese beiden Faktoren den Lösungsraum eingeschränkt haben.

Ein größerer Lösungsraum führt zu weiterer Arbeit des Lehrenden bei der Erstellung von Aufgaben, da er mehr Musterlösungen mit verschiedenen Ansätzen für den Trace-Vergleich bereitstellen muss. Es erscheint schwierig, dass ein Lehrender den gesamten kreativen Lösungsraum seiner Studenten abdecken kann. Insbesondere, wenn auch fehlerhafte Ansätze bedacht werden müssen.

Dies kann auf zwei Arten abgemildert werden. Zum einen bietet JACK die Möglichkeit mittels statischer Checks den Lösungsraum der Studierenden einzuschränken. Hierbei geht es nicht darum die Kreativität der Studierenden zu beschneiden, sondern unnötige oder sogar kontraproduktive Programmkonstrukte auszuschließen. Somit wird der Lösungsraum eingegrenzt und gleichzeitig den Studierenden eine Hilfestellung gegeben, damit sie sich nicht in falschen Lösungsansätzen verrennen.

Zum anderen kann der Lehrende die Ergebnisse einer Übungsaufgabe in JACK einsehen und Lösungen, für die kein passendes Muster vorlag, auswerten. Ihm stehen drei Möglichkeiten zur Verfügung. Erstens kann die Lösung in die Menge der Musterlösungen aufgenommen werden, womit für alle folgenden Semester ein passendes Muster für diesen Ansatz vorliegt. Zweitens kann ein Testfall für diesen Fehlertypen erstellt werden, damit dem Studierenden eine Hilfestellung gegeben wird. Drittens kann der Lösungsraum wie oben beschrieben eingegrenzt werden, falls sich der Ansatz als unbrauchbar herausstellt.

5 Fazit

Die Einsatztauglichkeit von Trace-Vergleichen zur Feedback-Erzeugung in JACK konnte im Rahmen der Abschlussarbeit nur teilweise bewertet werden. Da die vorhandenen Daten zum Großteil aus Aufgaben stammten, welche für den Trace-Vergleich eine schlechte oder unbrauchbare Struktur aufwiesen, musste eine Differenzierung in der Bewertung vorgenommen werden. Mit den aktuellen Aufgaben lag die Erfolgsrate einen Feedback-Marker zu erzeugen bei 7,5 %. Dies ist zwar ein sehr geringer Prozentsatz, allerdings kann mit wenig Aufwand dafür gesorgt werden, dass der Algorithmus wesentlich bessere Ergebnisse liefert.

Werden die Aufgaben an die Anforderungen für den Trace-Vergleich angepasst und Musterlösungen zur Abdeckung des Lösungsraums erstellt, kann der Trace-Vergleich, obwohl bestimmte Fehlertypen aus technischen Gründen nicht erkannt werden können, eine Erfolgsquote von 42 % vorweisen und damit zur Feedback-Erzeugung sinnvoll eingesetzt werden. Somit wird den Studierenden ohne viel Aufwand eine weitere Hilfestellung zur Fehlererkennung geboten. Weiterhin können die Testfälle angepasst werden. Spezielle kurze Testfälle für spezifische Fehler mit fehlerbezogenen Hinweisen kombiniert mit einem ausführlichen Testfall, um nicht abgedeckte Fehler mittels Trace-Vergleich abzufangen, würden den Studierenden in allen Fällen gute Hinweise auf ihre Fehler liefern.

Mit der Information, dass sieben Fehlertypen für 52 % aller Fehler in den Aufgaben verantwortlich waren, lässt sich darauf schließen, dass bestimmte Programmierkonzepte für die Studierenden schwer zu verstehen sind. Lehrende könnten speziell auf diese Fehler eingehen und dafür Sorge tragen, dass diese Fehler nicht mehr so häufig auftreten.

Hierzu muss JACK um die Möglichkeit erweitert werden, automatisch fehlerhafte Lösungen nach Fehlertypen zu gruppieren. Da gleiche Fehler meistens zu einem gleichen *Alignment*-Wert führen, kann dieser als Anhaltspunkt genutzt werden. Diese gruppierten Fehler können jeweils mit einer Bezeichnung versehen werden. Damit wäre es für den Lehrenden in jedem Semester möglich sich die häufigsten Fehlertypen der Übungsaufgaben anzeigen zu lassen und diese nochmals in der Vorlesung zu behandeln.

Literatur

[GJH12] Gerdes, A.; Jeuring, J.; Heeren, B.: An Interactive Functional Programming Tutor. Annual Conference on Innovation and Technology in Computer Science Education, 2012.

[GH17] Goltermann, R.; Höppner, F.: Internalizing a Viable Mental Model of Program Execution in First Year Programming Courses. ABP, 2017.

[Gr17] Gross, S.; Mokbel, B.; Hammer, B.; Pinkwart, N.: Feedback provision strategies in intelligent tutoring systems based on clustered solution spaces. DeLFI 2012: Die 10. e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V.. Gesellschaft für Informatik, Bonn, S. 27-38, 2012.

[Gr19] Graph Repository Query Language (GReQL), https://www.uni-koblenz-landau.de/de/koblenz/fb4/ist/rgebert/research/graph-technology/GReQL, besucht: 2019-06-10.

[GS17] Goedicke, M.; Striewe, M.: 10 Jahre automatische Bewertung von Programmier-aufgaben mit JACK - Rückblick und Ausblick. INFORMATIK 2017. Gesellschaft für Informatik, Bonn, S. 279–283, 2017.

[RK14] Rivers, K.; Koedinger, K.R.: Automating Hint Generation with Solution Space Path Construction. In Proceedings of the 12th International Conference on Intelligent Tutoring Systems, S. 329-339, 2014.

[St15] Striewe, M.: Automated analysis of software artefacts - a use case in e-assessment. Dissertation, 2015.

[St16] Striewe, M.: An architecture for modular grading and feedback generation for complex exercises. Science of Computer Programming 129. Special issue on eLearning Software Architectures, S. 35–47, 2016.

[Su17] Suzuki, R. et.al.: TraceDiff: Debugging unexpected code behavior using trace divergences. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) S. 107-115, 2017.

[Uk85] Ukkonen, E.: Finding approximate patterns in strings. Journal of Algorithms 6.1 S. 132–137, 1985.

[Vi03] Visser, W. et.al.: Model Checking Programs. Automated Software Engineering 10, S. 203-232, 2003.

[ZY18] Zaman, T.S.; Yu, T.: Extracting Implicit Programming Rules: Comparing Static and Dynamic Approaches. In Proceedings of the 7th International Workshop on Software Mining, 2018.