LlsChecker, ein CAA-System für die Lehre im Bereich Programmiersprachen

D. Rösner, M. Amelung, M. Piotrowski
Institut für Wissens- und Sprachverarbeitung
Otto-von-Guericke-Universität Magdeburg
email: <nachname>@iws.cs.uni-magdeburg.de

Abstract: Wir berichten über Entwurf, Implementierung und Einsatz des Systems LlsChecker. LlsChecker ist eine in ein Content-Management-System (CMS) für Lehr- und Lernmaterialien integrierte Komponente zur automatischen Überprüfung studentischer Lösungen für Programmieraufgaben in unterschiedlichen funktionalen Programmiersprachen. Das System ist so generisch organisiert, dass die Ausweitung der Dienste auf weitere Sprachen – zumindest für funktionale Programmiersprachen – allein durch eine XML-basierte Deklaration möglich ist.

1 Einleitung

Seit einigen Semestern gibt es vermehrt Klagen darüber, dass zahlreiche Studierende der Informatik und verwandter Studienangebote auch nach dem Vordiplom noch nicht über ausreichende Erfahrungen im Programmieren verfügen. Das bisherige Lehrangebot scheint zu viele „Schlupflöcher“ offen zu lassen, so dass sich Studierende davor drücken können, wirklich Programmiererfahrungen zu sammeln. Hier setzten die Überlegungen an, mit Hilfe von E-Learning einerseits Studierenden zu mehr Programmierpraxis zu verhelfen, andererseits aber durch ‘computer-assisted assessment’ (CAA) den Betreuungsaufwand in vertretbarem Rahmen zu halten.

Die Grundidee für das realisierte und seit dem Wintersemester 2003/04 eingesetzte CAA-System LlsChecker lässt sich wie folgt charakterisieren: Es ist aus der Theoretischen Informatik bekannt, dass es im allgemeinen Fall nicht möglich ist, automatisch die Korrektheit von Programmen zu bestimmen. Andere Überprüfungen sind aber möglich. Steht zu einer Aufgabenstellung eine sorgfältig überprüfte oder gar verifizierte Musterlösung zur Verfügung, so kann diese auf Testdaten angewendet werden. Verhält sich eine studentische Lösung auf Testdaten anders als die Musterlösung, wird sie als inkorrekt eingestuft. Eine studentische Lösung, die sich auf allen Testdaten so verhält wie die Musterlösung, ist vermutlich korrekt, wobei das Vertrauen in die Korrektheit um so größer ist, je mehr unterschiedliche Testdaten in die Überprüfung einfließen. Selbstverständlich dürfen die kompletten Testdaten den Programmerstellern nicht vorab bekannt sein, um Triviallösungen auszuschließen, die lediglich über ein Case-Statement auf die Eingaben die richtigen

Ausgaben liefern.

Ein wichtiges Argument für den Wunsch nach Computerunterstützung bei Übungsaufgaben und ihrer Bewertung ist die erwartete höhere Effizienz. In Zeiten steigender oder immerhin noch hoher Studierendenzahlen und gleichzeitigem Personalabbau ist es um so dringlicher, die für die Betreuung von Studierenden zur Verfügung stehende Arbeitszeit besonders effektiv zu nutzen und automatisierbare Teile des Übungsbetriebs auch tatsächlich maschinell zu unterstützen.

Unser Beitrag ist wie folgt organisiert: Zunächst diskutieren wir in Kapitel 2 Anforderungen an das System aus Sicht der Lehrenden und der Studierenden, dann skizzieren wir die realisierte generische Implementierung. Danach diskutieren wir didaktische Fragen des E-Assessment bei Programmieraufgaben. Über unsere Erfahrungen beim Einsatz in der Lehre berichten wir in Kapitel 4. Kapitel 5 stellt verwandte Arbeiten vor. Den Abschluss bildet ein Ausblick auf in Arbeit befindliche Erweiterungen und weitere Einsatzmöglichkeiten des LlsChecker.

2 Architektur

2.1 Anforderungen

Das CAA-System sollte so konzipiert und implementiert werden, dass für den Lehrenden, der es einsetzen möchte, ein minimaler Zusatzaufwand für die Einarbeitung in die Anwendung entsteht.

Ebenso sollte die Benutzung des Systems zur Einreichung und automatischen Auswertung von Programmierlösungen auch für Studierende keinen zusätzlichen Lernaufwand nach sich ziehen. Die Integration in die vertraute Umgebung, in der andere Vorlesungsmaterialien und Informationen zur Vorlesung bereit gestellt werden, ist daher unabdingbar.

Probleme der Nutzung, der Präsentation oder der Interaktion sollen den Lehrenden weitgehend abgenommen werden, so dass sie sich auf die Inhalte und die inhaltliche Strukturierung der Übungsaufgaben konzentrieren können.

Typische Aufgaben, wie das Erstellen und Publizieren von Programmieraufgaben und letztlich die Auswertung der studentischen Programmlösungen müssen daher einfach und schnell durchzuführen sein und idealerweise durch einen Workflow unterstützt werden.

Der effiziente Einsatz des LlsSystems setzt darüber hinaus die Sammlung sämtlicher Übungsaufgaben in einem zentralen Datenpool voraus – denn nur so ist es möglich, gezielt und mit geringem Aufwand nach vorhandenen Lehrmaterialien zu suchen und diese auch wiederzuverwenden.

Schon beim System von Forsythe und Wirth [FW65] wurde ein direktes Ausführen studentischer Einreichnungen als potentielles Sicherheitsproblem betrachtet. Studentischer Code darf auf keinen Fall zu einem Systemabsturz oder – noch schlimmer – zu einer Veränderung des Bewertungssystems führen. Daher sollte der LlsChecker die studentischen Einreichnungen jeweils in einer ‘sand box’ ausführen. Potentiell kritische Aktionen sind

hier nicht zugelassen und Programme werden gestoppt, wenn sie länger als eine vorgegebene Zeit arbeiten, da dann vermutlich eine Endlosschleife oder Endlosrekursion vorliegt.

Weitere Voraussetzungen für das System sind die Zugänglichkeit rund um die Uhr (insbesondere während der Einreichungsperiode), Robustheit und Erweiterbarkeit.

2.2 Aufbau des Systems

Eine unmittelbare Konsequenz aus den Anforderungen war, die Benutzeroberfläche komplett in das Content-Management-System (CMS) zu integrieren, das auch für Vorlesungsmaterialien usw. verwendet wird. Somit finden alle Arbeitsschritte, die sowohl der Lehrende als auch der Lernende mit dem System durchführen kann, in einer ihm bereits vertrauten Arbeitsumgebung statt.

Für unsere Lehrveranstaltungen [1] verwenden wir das auf dem quelloffenen (open source) Anwendungsserver Zope [zop04] aufgesetzte Content-Management-Framework Plone [plo04]. Die Präsentation der Aufgaben sowie die Auswertungen wurden als Plone-Produkte implementiert und sind somit vollständig in Plone integriert.

Um bereits erarbeitete Programmieraufgaben mit ihren zugehörigen Metadaten und Musterlösungen wiederverwenden zu können, bedient sich der Checker eines XML-basierten Repository mit Übungsaufgaben. Dieses Repository ist in Form einer relationalen Datenbank umgesetzt, auf die von Zope aus unproblematisch zugegriffen werden kann.

Will ein Lehrender eine neue Aufgabe in das Repository einbringen, so erfordert dies, dass er mit Hilfe eines komfortablen Erfassungswerkzeuges (Autorenwerkzeug) eine Musterlösung in der Programmiersprache der Aufgabenstellung und zugehörige Testdaten bereitstellt. Der sorgfältige Entwurf von Testdaten wie auch die Sicherstellung der Korrektheit der Musterlösung sind je nach Komplexität der Aufgabenstellung ein nicht zu vernachlässigender Aufwand, aber durch das Repository ergibt sich ein hoher Gewinn bei der Wiederverwendung.

Alle anderen Aufgaben im Zusammenhang mit der automatisierten Überprüfung der studentischen Einreichungen, also die Entgegennahme der Lösungen, ihre Überprüfung, die Benachrichtigung der Studierenden, die Ablage von Lösungen und Lösungsversuchen und verschiedene statistische Auswertungen erfolgen durch das System.

Das Gesamtsystem besteht zusammengefasst also aus

- dem Autorenwerkzeug, das die Definition von Aufgaben durch Lehrende ermöglicht,
- dem Web-Interface, in dem zum einen die Studierenden ihre Lösungen eingeben und die Auswertungen ansehen können und zum anderen die Lehrenden die Aufgaben bereitstellen,
- dem eigentlichen ‘Checker’, der die Programmierlösungen überprüft und
- dem Repository, das als zentrale Schnittstelle zwischen diesen Komponenten steht (s. Abbildung 1).

[1] http://wwwai.cs.uni-magdeburg.de:8080/wdok/lehre/sommer2005/FP/uebung

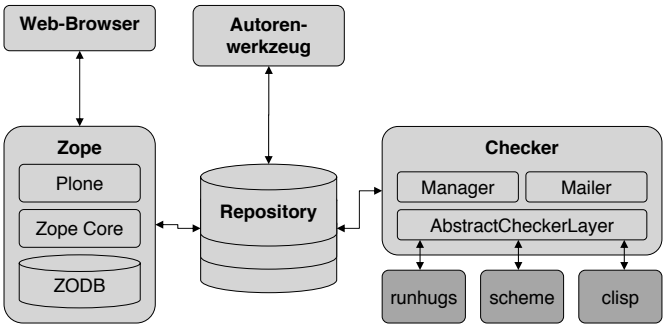

Abbildung 1: Systemarchitektur des LlsChecker-Systems

Aktuelle Überlegungen gehen dahin, die Zope-Datenbank (ZODB) als Repository zu benutzen und die Übungsaufgaben direkt dort abzulegen sowie das Autorenwerkzeug als Plone-Produkt umzusetzen. Die Vorteile lägen in einer noch engeren Integration des LlsChecker in das CMS sowie in der einfacheren Installation und somit wiederum vereinfachten Weitergabe des Systems. Durch entsprechende Import-/Exportschnittstellen würde dann sichergestellt, dass die Lehrmaterialien auch weiter in XML-Form verwendbar sind und somit das Potential der XML-Technologie voll genutzt werden kann.

2.3 Generische Implementierung

Die weiteren Komponenten des LlsChecker-Systems sind - ebenso wie Zope/Plone - in Python implementiert. Die eigentliche Checker-Komponente bietet derzeit die Überprüfung studentischer Lösungen zu Programmieraufgaben in Haskell, Scheme und CommonLisp. Um zu arbeiten, benötigt das System Informationen zur jeweiligen Aufgabe aus dem Repository und Informationen zum zu verwendenden Interpreter aus einer Konfigurationsdatei im XML-Format.

Die Metadaten zu einer Aufgabe im Repository beinhalten u. a. Angaben zur Programmiersprache, zur Hauptfunktion, die Musterlösung, die Testdaten, die zu verwendende Vergleichsfunktion.

In der Konfigurationsdatei sind diejenigen Informationen in einem deklarativen XML-Format zu hinterlegen, die zusätzlich vom LlsChecker benötigt werden, um mit einem konkreten Interpreter zur gewählten Programmiersprache arbeiten zu können. Zu diesen Informationen, die es erlauben, die Musterlösung und die studentische Einreichung im Interpreter auf die Testdaten anzuwenden und die Meldungen des Interpreter auszuwerten, gehören im einzelnen:

- Angaben zur Syntax von Funktionsaufrufen in der Programmiersprache,
- Angaben zum Starten und zum Beendigen des Interpreters und zu seinen unterschiedlichen Rückgaben im Fehler- und Erfolgsfall,
- Angaben zu gegebenenfalls zusätzlich erforderlichem ‘Wrapper-Code’, in den auszuführen-

der Code „verpackt“ werden sollte (z. B. für Vergleiche),

- Angaben zur Strategie des Checkers bei der Abarbeitung.

Die Angaben zur Strategie sind nur erforderlich, wenn vom vordefinierten Standardablauf abgewichen werden soll.

Dieser generische Ablauf des LlsChecker sieht wie folgt aus:

- Laden der Musterlösung,
- Laden der Testdaten,
- Anwenden der Musterlösung auf die Testdaten,
- Laden der studentischen Lösung, danach:
  - Überprüfen auf Ausführbarkeit im Interpreter,
  - Überprüfen auf Termination innerhalb eines Zeitlimit,
  - Überprüfen auf Fehler zur Laufzeit,
  - Sammeln der Ergebnisse für alle Testdaten,
  - Abgleich mit den gesammelten Ergebnissen der Musterlösung.

Wenn Überprüfungen oder Abgleiche negativ ausfallen, wird die Bearbeitung der aktuellen Einreichung beendet und der/die jeweilige Studierende wird per E-Mail informiert.

3 Didaktische Überlegungen

3.1 Alternative Lösungen

Beim Entwurf des Checkers drehte sich ein Teil der Diskussionen um die Frage nach alternativen Lösungen für typische Programmieraufgaben und um mögliche „typische“ Fehler. Anhand des folgenden Beispiels werden die relevanten Argumente verdeutlicht.[2]

Angenommen, die Studierenden haben die folgende Aufgabe zu lösen:

> *Implementieren Sie in Haskell eine Funktion namens ‘remDuplicates’, die zu einer gegebenen Liste als Eingabe eine Liste zurückgibt, in der alle Mehrfachvorkommen von Elementen entfernt sind.*

Der Aufgabensteller hat vielleicht an eine Lösung gedacht wie:

```
remDuplicates [] = []
remDuplicates (x:xs) = if (elem x xs) then remDuplicates xs
                       else x : remDuplicates xs
```

[2]Die Diskussion verwendet Beispiele aus Haskell, aber die Argumente gelten für andere Programmiersprachen ebenso. Der Kern der Argumentation sollte sich auch Lesern ohne Vorkenntnisse in Haskell (cf. [Bir98]) erschließen.

Eine studentische Lösung (hier zur Unterscheidung `remDuplicatesB` genannt) könnte auch sein:

```
remDuplicatesB [] = []
remDuplicatesB (x:xs) = x : remDuplicatesB (removeOccs x xs)

removeOccs _ [] = []
removeOccs y (x:xs) = if (y == x) then removeOccs y xs
                      else x : removeOccs y xs
```

Auf dem Beispiel `[1,2,1,3,2,4,1]` liefern die alternativen Versionen von `remDuplicates` die folgenden Ergebnisse:

```
RemDups> remDuplicates [1,2,1,3,2,4,1]
[3,2,4,1]

RemDups> remDuplicatesB [1,2,1,3,2,4,1]
[1,2,3,4]
```

Beide Lösungen sind korrekt, aber die produzierten Ergebnisse würden als verschieden eingestuft, wenn als Vergleichsfunktion nur Gleichheit (von Listen) verwendet würde. Da in der Spezifikation der Aufgabe `remDuplicates` keine Forderungen bezüglich der Reihenfolge der Elemente gestellt werden, die in der Liste nach Entfernen von Duplikaten verbleiben, sollten also Listen, die sich aus unterschiedlichen Lösungen ergeben, dann als gleich betrachtet werden, wenn sie Permutationen voneinander sind.

Ein weiteres Beispiel für eine solche 'Unterspezifikation' des Ergebnisses einer listenwertigen Funktion liefert die Aufgabe:

> *Implementieren Sie in Haskell eine Funktion namens 'permutations', die zu einer gegebenen Liste l als Eingabe die Liste mit allen Permutationen von l zurückgibt.*

3.2 Hilfestellung durch Testdaten

Für das folgende Beispiel wird angenommen, dass der/die Studierende den Text der Programmieraufgabe `remDuplicates` eventuell nicht sorgfältig genug gelesen hat und daher beim Versuch einer Lösung von der Vorstellung ausgeht, nur unmittelbar benachbarte Duplikate seien zu entfernen. Allerdings – so die weitere Annahme – wird selbst die falsch verstandene Aufgabe auch noch fehlerhaft 'gelöst'. Hier der studentische Versuch:

```
remDuplicates [] = []
remDuplicates [x] = [x]
remDuplicates (x:y:xs) = if (x == y) then x : remDuplicates xs
                         else x : remDuplicates (y:xs)
```

Der Checker kann das Programm ausführen und es terminiert. Der/die Studierende erhält einerseits Rückmeldung über E-Mail (s. Abbildung 2), kann andererseits Informationen zum Ergebnis auch über seine Benutzerberechtigung im CMS einsehen.

In der Rückmeldung werden jeweils einige der Testdaten, die von der Einreichung erzielten Resultate und die erwarteten Ergebnisse einander gegenübergestellt:

```
...
  Testdaten: [1,2,1,3,2,4,1]
  Ihr Ergebnis: [1,2,1,3,2,4,1]
  Erwartetes Ergebnis: [3,2,4,1]

  Testdaten: [1,2,2,2,3]
  Ihr Ergebnis: [1,2,2,3]
  Erwartetes Ergebnis: [1,2,3]
...
```

Die Erwartung ist, dass der/die Studierende mit diesen Informationen den Lösungsversuch überdenkt. Das erwartete Ergebnis für die Liste `[1,2,1,3,2,4,1]` sollte helfen, den anfänglichen konzeptuellen Fehler zu erkennen: alle Duplikate und nicht nur unmittelbar aufeinanderfolgende müssen entfernt werden. Der Vergleich der Ergebnisse für `[1,2,2,2,3]` hilft hoffentlich, den Fehler bei der Rekursion in der eingereichten Lösung zu erkennen.

Ergibt sich aus diesen Einsichten eine erneute Einreichung, die akzeptiert werden kann, dann erhält der/die Studierende eine entsprechende Bestätigung, andernfalls eine erneute Aufforderung, die Lösung zu verbessern.

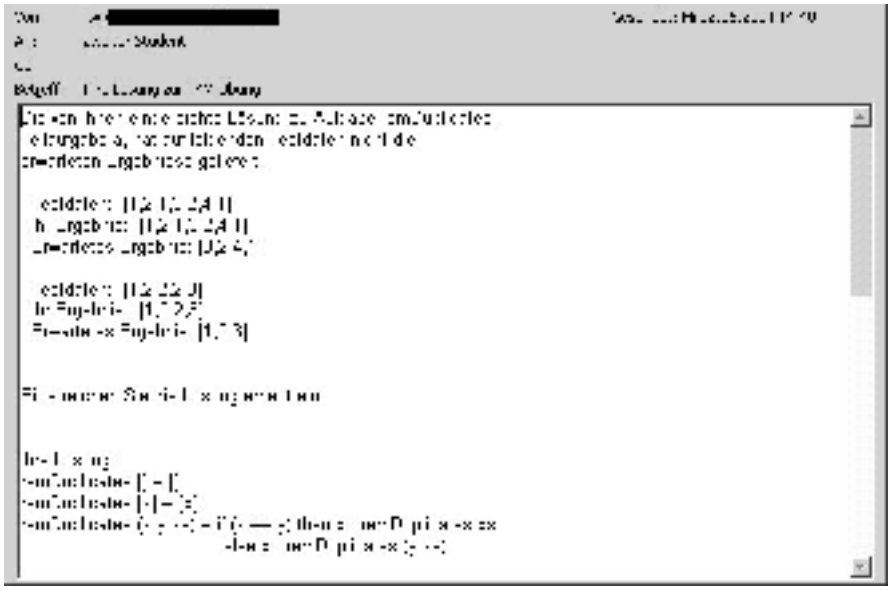

Abbildung 2: Rückmeldung per E-Mail.

3.3 Zum Umgang mit Plagiaten

Ein Lerneffekt stellt sich bei Aufgaben nur dann ein, wenn man versucht, sie selbstständig zu lösen. Wer die Lösungen eines Anderen lediglich kopiert, bringt sich um diesen Lerneffekt. Leider sind sich nicht alle Studierenden dieser Tatsache bewusst oder handeln nicht dieser Einsicht entsprechend. Wer die Vergabe von Leistungspunkten von elektronischen Einreichungen abhängig macht, kommt daher nicht umhin, sich Gedanken zum Umgang mit Plagiaten zu machen.

Aufgabenstellungen, die, wie es bei funktionalen Sprachen oft der Fall ist, mit sehr kompaktem Code – manchmal in Form eines 'Einzeilers' – gelöst werden können, bieten kaum Möglichkeiten zur Variation bei der Lösung. Auch unabhängig arbeitende Studierende können so zu exakt der gleichen Lösung mit genau dem gleichen Aussehen kommen, ohne voneinander abgeschrieben zu haben. Dies gilt um so mehr, wenn sie sich an die Konventionen für die Benennung von Bezeichnern für Funktionsnamen, Parameter und so weiter halten.

Werden die Aufgaben komplexer und erlauben sie unterschiedliche Arten der Problemdekomposition, so sind exakt übereinstimmende Gesamtlösungen zunehmend unwahrscheinlich. Für solche Aufgaben erscheint das folgende Vorgehen praktikabel: Wenn für eine komplexe Aufgabe bei einer Einreichung ein Programm übergeben wird, das exakt mit einem bereits eingereichten Programm übereinstimmt, so soll der Zweiteinreicher nicht nur darauf hingewiesen, sondern aufgefordert werden, eine geänderte Lösung einzureichen. Eine Folgefrage ist allerdings, ab wann eine erneute Einreichung dann als hinreichend unterschiedlich betrachtet wird. Leider ist es auch möglich, ohne wirkliches Verständnis eines Programms durch konsequentes Umbenennen von Identifikatoren eine Variante zu erzeugen, die sich von der Vorlage zwar unterscheidet, aber keine eigenständige Lösung darstellt.

Ein anderes Modell kann sein, eine Lösung, die sich von allen anderen deutlich unterscheidet, höher zu bepunkten als eine später eingereichte Lösung, die nur eine Kopie oder Variante davon ist.

Es ist zwar wichtig, die Möglichkeiten für Schummeleien so gering wie möglich zu halten, dennoch sollte man nicht der Illusion verfallen, dass sie gänzlich auszuschließen sind. Es ist daher sinnvoller, Studierenden, die Tests und andere Leistungskontrollen ernst nehmen, für die erkannten Defizite entsprechende Lernhilfen anzubieten. Gegebenenfalls sollten als Ergänzung zum CAA-System Leistungskontrollen vorgesehen werden, bei denen die Individualität der Lösungen besser gewährleistet werden kann, z. B. Klausuren, die in das Gesamtkonzept der Lehrveranstaltung einbezogen werden.

4 Der LlsChecker im Einsatz

4.1 Einsatz in Verbindung mit Präsenzlehre

Die Lehrveranstaltung „Programmierkonzepte und Modellierung (PKM)“ ist an der Fakultät für Informatik (FIN) der Otto-von-Guericke-Universität Magdeburg Pflichtveranstaltung für Studierende der Informatik im 3. Semester und Wahlpflichtveranstaltung für Studierende der Ingenieurinformatik, Computervisualistik und Wirtschaftsinformatik. Darüberhinaus wird die LV auch von Studierenden der Fakultät für Elektro- und Informationstechnik besucht.

In dieser LV haben wir im Wintersemester 2003/04 begonnen, ergänzend zur Präsenzlehre webbasierte E-Learning-Komponenten einzusetzen. Dazu gehörten [RA05]:

- eine Pilotversion des LlsChecker mit vier Aufgaben in Haskell und Scheme,
- ein Werkzeug zur strukturierten Erfassung studentischer Induktionsbeweise (für Korrektur durch Assistenten),
- eine erste Version eines Werkzeugs für interaktive Test im Multiple-Choice-Format (LlsMultipleChoice).

Aufgrund der gemachten Erfahrungen und der Rückmeldungen aus einer Fragebogen-Aktion für Studierende wurden die Werkzeuge weiter ausgebaut und besser in das CMS der Lehrveranstaltung integriert.

Im vergangenen Wintersemester 2004/05 wurde der verbesserte Checker in PKM für eine Programmieraufgabe in jeder Woche eingesetzt (Programmiersprachen: Haskell und Scheme). Damit konnte sichergestellt werden, dass die Studierenden deutlich mehr Programmiererfahrung sammeln als in früheren Semestern. Zusätzlich wurde ein via WWW einzureichender Induktionsbeweis verlangt und die Bearbeitung von Multiple-Choice-Fragen zu Prolog.

Im Sommersemester 2005 wird der LlsChecker zum ersten Mal in den Lehrveranstaltungen „KI-Programmierung und Wissensrepräsentation (KPWR)“ (für Aufgaben in CommonLisp) und „Funktionale Programmierung (FP)“ (für Aufgaben in Haskell, Erlang, Oz) eingesetzt.

4.2 Sicht der Studierenden

Für die Studierenden bestand die Möglichkeit, innerhalb einer Woche ihre Lösung einzureichen bzw. bei einer als falsch eingestuften Lösung, eine verbesserte nachzureichen. Die Zahl der Versuche war nicht limitiert.

Nach erfolgreicher Einreichung, d. h. die Lösung hat auf allen Testdaten das erwartete Ergebnis produziert und stimmt nicht exakt mit einer bereits eingereichten Lösung überein, waren keine weiteren Einreichungen mehr möglich.

Der Lerneffekt von Übungen – so die allgemeine Einschätzung – ist auch davon abhängig, dass Rückmeldungen auf studentische Problemlösungen zeitnah erfolgen. Beim LlsChecker ist die Rückmeldung innerhalb weniger Minuten nach Einreichung sichergestellt. Diese zeitnahe Rückmeldung erleichtert es den Studierenden, ihre (falsche oder nur partiell korrekte) vorige Lösung anhand der rückgemeldeten Abweichungen zwischen erwarteten und tatsächlichen Ergebnissen zu überarbeiten und erneut einzureichen. Die Erfahrungen zeigen, dass Studenten diese Möglichkeit zur Überprüfung ihrer Programme sehr schätzten.

5 Verwandte Arbeiten

Wegen der zentralen Rolle von Programmierübungen in der Ausbildung der Informatik wundert es nicht, dass erste Systeme zur Unterstützung bei Bepunktung ('marking') und Benotung ('grading') von studentischen Lösungen schon um 1960 entwickelt und eingesetzt wurden. Als Klassiker eines solchen CAA-Systems gilt das in Stanford entstandene System von Forsythe und Wirth [FW65]. Der 'grader' war in ALGOL geschrieben. Obwohl bereits in der Zeit der Lochkarten entstanden, sind die Motive (u. a. 'teach programming ... to masses of students') und die diskutierten Themen relevant geblieben: Arten der Bewertung, Speichern der Bewertungen, Kontrolle der Laufzeit, Sicherheit, Beenden fehlerhafter oder 'bösartiger' Einreichungen usw.

Das System BAGS (Basser Automatic Grading Scheme, [JW69]) war zwar noch batchbasiert, aber es war bereits in das Betriebssystem integriert und erforderte keine Spezialkenntnisse mehr seitens der Nutzer. Es nahm Lösungen zu Aufgaben in ALGOL, in MINIGOL und in KDF9-Assembler entgegen und bewertete sie. Zu den Design-Vorgaben gehörte, dass Nutzer sich nicht an das System sollten anpassen müssen.

KASSANDRA von Urs von Matt von der ETH Zürich war netzbasiert mit einer Client-Server-Architektur ([vM94]). Das System war ausgelegt für den Abgleich studentischer Lösungen mit Musterlösungen zu Aufgaben für Maple und Matlab. Client-Code nahm nur die studentischen Lösungsversuche entgegen, denn der Zugang zur Implementation, den Testdaten und den bewerteten Ergebnissen musste selbstverständlich ausgeschlossen werden.

TRAKLA2 und das Vorgängersystem TRAKLA erlauben Studierenden, ihr Wissen über Algorithmen und Datenstrukturen zu demonstrieren und zu vertiefen, indem sie in einem Simulationswerkzeug mit GUI-Operationen Visualisierungen von Datenstrukturen manipulieren [LSKM04]. Die Beispieldaten (z. B. für Sortieren) können zufällig erzeugt werden, so dass nahezu beliebig viele unterschiedliche Aufgabeninstanzen möglich sind.

CourseMaster ist der Nachfolger des bereits 1988 begonnenen und an vielen Universitäten weltweit eingesetzten Ceilidh-Systems [HST02]. Dieses System bietet Unterstützung für eine ganze Palette von Aufgaben bei Lehrveranstaltungen. Dazu gehören:

- Aufgaben der Kursadministration,
- Präsentation von Informationen und Materialien für Studierende und
- automatische Bewertung studentischer Einreichungen.

Begonnen hatten die Arbeiten zur Bewertung von studentischen Programmen mit imperativen Sprachen (C, C++, Pascal). Für dieses Paradigma scheinen sie auch am ausgereiftesten zu sein. In Ceilidh/CourseMaster wird nicht nur versucht, die funktionale Korrektheit studentischer Programme zu überprüfen (durch Abgleich mit Testdaten), einen breiten Raum nimmt auch der Versuch ein, die Qualität der Lösungen mit unterschiedlichen Software-Metriken zu bewerten. Es wurde auch versucht, diese Metriken auf Programme in Prolog und auf Aufgaben der objektorientierten Analyse und des objektorientierten Entwurfs zu übertragen, allerdings war dies auch aus Sicht der Autoren zumindestens für die letzteren Aufgaben nicht wirklich erfolgreich.

Das Ceilidh/CourseMaster-System beurteilt und bewertet studentische Programme anhand der Ausgaben, die diese Programme auf Testdaten produzieren. Da die Gestaltung der Programmausgabe nicht zu sehr vorgegeben werden soll, muss versucht werden, mittels regulärer Ausdrücke in den studentischen Ausgaben die wesentlichen Werte der korrekten Musterlösung zu finden. Das kann mühsam und fehleranfällig sein.

Daher hat die Gruppe aus Helsinki, die auch zu den Nutzern von Ceilidh/CourseMaster gehört, für die automatische Bewertung von Aufgaben in Scheme nicht Ceilidh angepasst, sondern mit Scheme-robo [SMK01] eine eigene Lösung entwickelt. Scheme-robo nutzt aus, dass bei funktionalen Sprachen wie Scheme die Programme Funktionen sind, deren Werte direkt mit den Werten der Musterlösung abgeglichen werden können.

Der LlsChecker nutzt diesen fundamentalen Vorzug funktionaler Sprachen ebenso, geht aber über Scheme-robo hinaus, indem er nicht nur eine einzelne Sprache unterstützt, sondern aufgrund seines generischen Ansatzes für unterschiedliche Programmiersprachen einsetzbar ist. Weitere Unterschiede zu anderen Arbeiten zum E-Assessment von Programmen liegen darin, dass der LlsChecker XML-basiert und in ein CMS (mit u. a. Unterstützung für Internationalisierung) integriert ist.

6 Ausblick

In der jetzigen Implementierung des LlsChecker ist zwar bereits Wert auf die leichte Erweiterbarkeit auf andere Programmiersprachen gelegt worden, dennoch ist nicht das gesamte Spektrum aller Programmierparadigmen in gleicher Weise bereits im Design berücksichtigt. Es ist verhältnismäßig unaufwendig und allein durch Deklaration möglich, eine interpretierte funktionale Sprache oder eine Sprache mit funktionalem Kern für das Checkersystem nutzbar zu machen. Die aktuellen Arbeiten haben zum Ziel, diesen generischen Ansatz insofern noch einmal zu verallgemeinern, dass auch Sprachen, bei denen ein anderes Paradigma dominant ist, ebenso komfortabel integriert werden können. Dazu gehören aktuelle Arbeiten zur Nutzung des Checker für Programmieraufgaben in Prolog als Beispiel für logische Programmierung und Arbeiten zur Nutzung des Checker für Java als objektorientierte und kompilierte Sprache.

In der beruflichen Weiterbildung sollte die Vermittlung moderner Programmierkonzepte eine stärkere Rolle spielen. Die Möglichkeit, einen Teil einer solchen Ausbildung berufsbegleitend und webbasiert durchzuführen, erscheint sehr attraktiv. Bei Gesprächen mit

Wirtschaftsvertretern zu dieser Thematik wurde die Idee eines solchen Angebots begrüßt. Der LlsChecker kann in einer solchen Nutzung ebenfalls wesentliche Dienste leisten. Ein Experiment hierzu ist im Laufe des Jahres 2005 geplant.

Zusatzinformationen

Die Arbeiten am LlsChecker sind Teil des vom Land Sachsen-Anhalt geförderten Projekts „XML-Technologie zur Unterstützung der Entwicklung und Wiederverwendung von Lehr- und Lernmaterialien“ (Förderkennzeichen: `0047M1/0002A`).

Eine Demoversion des CAA-Systems LlsChecker ist öffentlich zugänglich unter:

```
http://lls.cs.uni-magdeburg.de/demo/llschecker
```

Literatur

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Europe, Prentice Hall, 1998.

[FW65] George E. Forsythe und Niklaus Wirth. Automatic grading programs. *Commun. ACM*, 8(5):275–278, 1965.

[HST02] Colin Higgins, Pavlos Symeonidis und Athanasios Tsintsifas. The marking system for CourseMaster. In *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, Seiten 46–50. ACM Press, 2002.

[JW69] J.B.Hext und J.W. Winings. An Automatic grading scheme for simple programming exercises. *Commun. ACM*, 12(5):272–275, 1969.

[LSKM04] Mikko Laakso, Tapio Salakoski, Ari Korhonen und Lauri Malmi. Automatic Assessment of Exercises for Algorithms and Data Structures - A Case Study with TRAKLA2. In *Proceedings of the 4th Finnish/Baltic Sea Conference on Computer Science Education, October 1-3, 2004, Koli, Finland*, Seiten 28–36, 2004.

[plo04] Plone: A user-friendly and powerful open source Content Management System, http://www.plone.org, 2004.

[RA05] Dietmar Rösner und Mario Amelung. A Web-Based Environment to Support Teaching of Programming Paradigms. In *Proceedings of the 4th IASTED International Conference on Web-based Education (WBE 2005), February 21–23, 2005, Grindelwald, Switzerland*, Seiten 655–660. IASTED, ACTA Press, 2005.

[SMK01] Riku Saikkonen, Lauri Malmi und Ari Korhonen. Fully automatic assessment of programming exercises. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, Seiten 133–136. ACM Press, 2001.

[vM94] Urs von Matt. Kassandra: the automatic grading system. *SIGCUE Outlook*, 22(1):26–40, 1994.

[zop04] Welcome to Zope.org, http://www.zope.org, 2004.