Ein Schnittstellen-Datenmodell der Variabilität in automatisch bewerteten Programmieraufgaben

Robert Garmann
Fakultät IV – Wirtschaft und Informatik
Hochschule Hannover
30459 Hannover, Germany
robert.garmann@hs-hannover.de

***Zusammenfassung*—Automatisch bewertete, variable Programmieraufgaben stellen besondere Schnittstellenanforderungen an Autobewerter (Grader) und Lernmanagementsysteme (LMS). Um Wiederverwendung von Aufgaben über Systemgrenzen hinweg zu begünstigen, schlagen wir vor, Aufgabenschablonen durch eine von allen beteiligten Systemen genutzte Middleware zu instanziieren und dabei Variabilitätsinformationen in einem Schnittstellen-Datenmodell zu transportieren. Wir stellen ein solches Datenmodell vor, welches für die Grader-unabhängige Kommunikation mit LMS ausgelegt ist und beispielhaft im Autobewerter Graja implementiert wurde. Zudem wird eine Dialogkomponente für die manuelle Werteauswahl vorgestellt, die auch bei großen Wertemengen effizient und Grader-unabhängig einsetzbar ist. Die Eignung des Dialogs und des Datenmodells wird anhand eines typischen Bewertungsszenarios diskutiert.**

***Abstract*—Automatically graded, variable programming tasks put special interface requirements on auto-graders and learning management systems (LMS). To promote reuse of tasks across system boundaries, we propose that all involved systems should instantiate task templates through a middleware, while transporting variability information in an interface data model. We present such a data model, which is designed for grader-independent communication with LMS and was implemented exemplarily in the Graja auto-grader. In addition, a dialogue component for manual value selection is presented, which can be used efficiently and grader-independently, even with large value sets. We discuss the suitability of the dialogue and the data model for a typical grading scenario.**

Stichwörter—individuelle Programmieraufgaben; Grader; Autobewerter; E-Assessment; Variabilität

I. Einleitung

Im formativen E-Assessment bearbeiten Studierende verpflichtende Programmieraufgaben[1] im Selbststudium. Die über ein Lernmanagementsystem (LMS) eingereichten Lösungen werden entweder direkt oder über eine spezielle Middleware [6] an einen Autobewerter (Grader) gegeben, der die Einreichung hinsichtlich korrekter Funktion und hinsichtlich weiterer Qualitätsaspekte untersucht.

Es spricht einiges dafür, Aufgaben als Aufgaben*schablonen* mit variablen Stellen, sog. *Variationspunkten* (Vp) zu konzipieren. Durch Einsetzen individueller Variablenwerte entstehen automatisch sehr viele Aufgaben*instanzen*, deren Schwierigkeitsgrad[2] je nach Zielsetzung entweder vergleichbar ist oder intentionsgemäß variiert. So lassen sich etwa Zusatzaufgaben gleicher Schwierigkeit für intensives Üben generieren oder aber am individuellen Lernstand ausgerichtete Aufgabeninstanzen unterschiedlicher Schwierigkeit. Im summativen Assessment können Aufgabenvarianten gleicher Schwierigkeit individuell jedem Studierenden zugeordnet werden und so helfen, Plagiate zu unterbinden.

Die variablen Stellen einer Aufgabe und deren konkrete Werte müssen an den Schnittstellen zwischen LMS, Middleware und Grader übertragen werden (Abschnitt IV). Dieser Beitrag stellt ein hierfür geeignetes Datenmodell vor (Abschnitt V) und plausibilisiert dieses anhand einer Beispielaufgabe (Abschnitt III). Das Datenmodell wurde in einer unabhängig von einem konkreten Grader implementierten Klassenbibliothek realisiert. Die grundsätzliche praktische Eignung konnten wir anhand einiger variabler Programmieraufgaben für den Grader Graja belegen (Abschnitt VI).

II. Verwandte Arbeiten

Insbesondere in den Fachgebieten Mathematik und Physik werden individualisierbare Aufgaben schon länger eingesetzt (s. etwa [8]). Aber auch in der Informatik gibt es Ansätze. Die im Folgenden beispielhaft genannten Lösungen sind allerdings entweder nur in einem eingeschränkten Systemumfeld einsetzbar oder fokussieren auf einfache Codeanalyse-Aufgaben. Der Beitrag [12] beschreibt ein sehr einfaches System zur Generierung von Varianten für C++-Programmieraufgaben. In [9] wird ein Ansatz zur Generierung individueller Aufgaben für ein Teilgebiet der Programmierung (Auswertung von Ausdrücken) diskutiert. Der Aufsatz [1] berichtet über das QuizPACK-System für parametrisierte Aufgaben für die Sprache C, das später für Java portiert wurde. Dieses System stellt „parameterized code-execution exercises“, bei denen Studierende ein gegebe-

[1] Mit Programmieraufgaben bezeichnen wir Aufgaben, in denen ein Programm in einer Programmiersprache erstellt werden muss (Synthese). Einfachere Aufgaben, die sich mit der Analyse eines gegebenen Quelltextes befassen, sind für das hier betrachtete unüberwachte Selbststudium nur bedingt geeignet.

[2] Für Ideen zur systematischen Ermittlung der Variantenschwierigkeit s. z. B. [10].

nes Programm durchdenken müssen und dann Fragen wie „Welchen Wert hat die Variable x?“ beantworten müssen. Codesynthese-Aufgaben werden nicht unterstützt.

In der Produktlinienentwicklung werden Variabilitätsmodelle genutzt, um Varianten und deren sog. Constraints darzustellen. Beispielhaft seien die Common Variability Language (CVL, [7]) und das Orthogonal variability model [11] genannt. Die Produktlinienentwicklung hat mit individualisierbaren Programmieraufgaben gemeinsam, dass Variablen, Wertausprägungen und teilweise komplexe Randbedingungen formuliert werden müssen. Die Möglichkeiten solcher Ansätze gehen weit über die Erfordernisse einer variablen Programmieraufgabe hinaus. Das verwundert nicht, da in der Produktlinienentwicklung Modelle mit hunderten von Variablen keine Seltenheit sind [11], während wir in variablen Programmieraufgaben mit kaum mehr als einer kleinen zweistelligen Anzahl von Variablen rechnen.

III. Beispielaufgabe

Die in Abb. 1 gezeigte Programmieraufgabe liege im ProFormA-Format [13] als automatisch bewertete Java-Aufgabenschablone vor. Zur Aufgabe gehören weitere Artefakte. Beispielsweise definiert der Grader Graja [3] u. a. JUnit-Testmethoden, Regeln zur statischen Codeanalyse und Bewertungsvorschriften als Teil einer Aufgabe.

Die grauen Stellen in Abb. 1 definieren sieben sog. *Variationspunkte* (kurz Vp), deren Wertemengen in TAB. I. gelistet sind. Ein wichtiger Parameter ist die *domain* (werden Buchstaben oder Ziffern gezählt?), von der weitere Vp-Wertemengen abhängen. Bspw. hängt der *className* von *domain* ab.

Write a `main` method in class `%vp{className}` in the default package that reads a series of characters from user input. Your program should output a statistic of %vp{domain} in the input, i. e. the percentage of characters in { %vp{set} }. Example (user input is **highlighted**):

```
Give me characters, please: %vp{input}
%vp{output} % are %vp{domain}
```

Your program should print %vp{precision} decimal places of the percentage value. %vp{lowerSentence}

Abb. 1. Beispielaufgabe mit variablen Stellen (Variationspunkte)

TAB. I. Variationspunkte der Beispielaufgabe

Name: Typ	Sinnvolle Werte
input: String	Die Beispieleingabe {„x.ÜL:0€ÁHDú/7Ú“}. Hier nicht variiert aus Platzgründen.
domain: String	{„letters“, „numbers“}
className: String	{„Letters“}, falls domain=„letters“ {„Digits“, „Numbers“, „Figures“}, sonst
set: String	{„A-Z“, „A-Z, Ä, Ö, Ü“, „A-Z, Œ“}, für domain=„letters“ {„0-9“, „0-9, I, V, X, L, C, D, M“}, sonst
countLower: Boolean	{null}, falls set=„0-9“ {true, false}, sonst
lowerSentence: String	{„“}, falls countLower=null, {„Lowercase should be counted.“}, falls countLower=true {„Lowercase should not be counted.“}, sonst
precision: Integer	{1-9}
output: String	Die erwartete Ausgabe wie sie im Aufgabentext steht.

Einige Vp verändern intentionsgemäß die Schwierigkeit der Aufgabe. Z. B. fordert *set*=„A-Z, Œ“ im Gegensatz zu „A-Z“ i. d. R. die Berücksichtigung von Zeichenkodierungen. Andere Vp wie *precision* dienen der möglichst zahlreichen Variantenbildung bei gleichbleibender Schwierigkeit. Weitere Vp werden je nach Wahl anderer Vp sogar überflüssig: der letzte Satz der Aufgabe, der die Behandlung von Kleinbuchstaben fordert, ist bei *set*=„0-9“ zu streichen[3]. Die Tabelle enthält auch einen achten „versteckten“ Vp *countLower*, der zwar nicht im Aufgabentext, aber in weiteren Artefakten auftaucht [4].

IV. Systemumfeld und Anforderungen

In einem typischen Szenario lädt die Lehrkraft im LMS eine im ProFormA-Format vorliegende Aufgabenschablone hoch und reicht zum Test eine Musterlösung ein. In einem Dialog gibt sie für jeden Vp Werte ein, die zur Musterlösung passen. Der Dialog leitet die Lehrkraft und verhindert die Eingabe ungültiger Vp-Wertkombinationen. Das LMS erstellt mit den Werten eine Aufgaben*instanz*, sendet diese und die Musterlösung an eine Grading-Middleware bzw. an den Grader und zeigt abschließend das zurück erhaltene Feedback an (Abb. 2).

Für einen Studenten erstellt das LMS im Moment des erstmaligen Aufrufs eine individuelle Aufgabeninstanz. Es nimmt eine zufällige, die Vp-Abhängigkeiten berücksichtigende Wertbelegung vor und lässt ggf. weitere Faktoren wie die Lernhistorie des Studenten in die Auswahl einfließen. Das LMS speichert die für den Studenten geltende Variante persistent ab. Der eigentliche Einreichungs- und Bewertungsvorgang über die Middleware unterscheidet sich nicht von einer „normalen“ Aufgabe. Ggf. kann das LMS, wenn die Aufgabe gelöst wurde, eine individuelle neue Variante zu Übungszwecken anbieten.

In dem Szenario ist das LMS für die Auswahl von Wertbelegungen und Aufgabeninstanziierungen alleine verantwortlich. Das LMS wird jedoch kaum ohne die Unterstützung des Graders auskommen. Ggf. liegen die von den Vp betroffenen Aufgabenartefakte in proprietären Binärformaten vor. Nur der Grader wird letztverantwortlich aus einer Aufgabenschablone eine Instanz erzeugen können. Einfache Aufgabenteile wie den Aufgabentext kann ggf. das LMS autark instanziieren. Abb. 2 stellt dar, wie das LMS auf einen sog. Instanziierungsdienst zurückgreift, welcher aus einer Aufgabenschablone und einer vorgegebenen „Auflösung“ der Variationspunkte (cvr) eine Aufgabeninstanz erzeugt. Ggf. bedient sich der Instanziierungsdienst dabei weiterer Unterstützung des betroffenen Graders, der hierfür eine spezielle Instanziierungsfunktion anbietet.

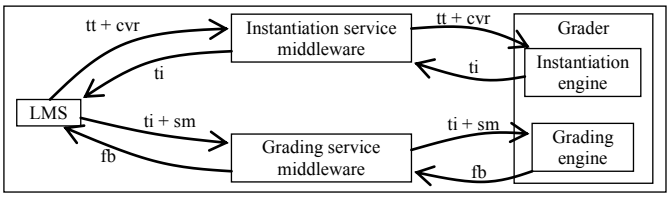

Abb. 2. Nutzung eines Instanziierungsdienstes. Abkürzungen: tt = task template (Aufgabenschablone), ti = task instance (Aufgabeninstanz), cvr = composite variation resolution (Auflösung aller Variationspunkte zu konkreten Werten), sm = submission (Einreichung), fb = Feedback.

[3] Die in [11] beschriebene Abhängigkeit „Variant excludes variation point“ ist hiermit vergleichbar.

Der Instanziierungsdienst entscheidet auch über den Zeitpunkt, in dem variable Artefakte einer Aufgabe in konkrete Artefakte überführt werden. Diese sog. *binding time* kann von Artefakt zu Artefakt variieren. Details und weitere Szenarien, in denen ein Instanziierungsdienst nutzbringend eingesetzt werden kann, sind in [4] beschrieben.

Wir wollen im ProFormA-Format vorliegende Aufgaben LMS- und Grader-unabhängig individualisieren. Wir benötigen dazu eine Grader-unabhängige Sprache zur Beschreibung der Vp, die es LMS in verschiedenen technischen Umgebungen erlaubt, das oben beschriebene Szenario umzusetzen. Die Beschreibung soll Wertemengen und gegenseitige Vp-Abhängigkeiten enthalten und dabei sowohl komplexe als auch geringe Abhängigkeiten redundanzarm abbilden. Die Auswahl gültiger Vp-Wertkombinationen muss effizient unterstützt werden.

V. Ein Datenmodell der Variabilität

Wir haben ein in XML repräsentierbares Datenmodell entwickelt, welches eine Teilmenge des von mehreren Vp aufgespannten mehrdimensionalen Raums beschreibt. Neben Grundoperationen wie kartesisches Produkt und Vereinigungsmenge werden weitere Operationen unterstützt, die es erlauben, die Vp redundanzarm und damit wartbar zu spezifizieren. Kontinuierliche Wertebereiche werden mit einer vorzugebenden Schrittweite diskretisiert. Zudem kann man Vp-Werte von anderen Vp-Werten ableiten[4], indem eine Javascript-Funktion in die XML-Beschreibung eingebettet wird. Wir wählten Javascript wegen der weiten Verbreitung auf Serverplattformen und im Browser.

A. *Benutzersicht*

Wir beginnen mit der Benutzersicht des weiter unten beschriebenen Datenmodells, um dessen effiziente und Grader-unabhängige Einsetzbarkeit zu plausibilisieren. Abb. 3 zeigt die Benutzersicht für die Werteauswahl.

Variable	Select	Value
input		x.ÜL:0€ÁHDú/7Ú
domain		letters
className		Letters
set		A-Z
countLower		false
lowerSentence		Lower case letters should not be coun...
precision		2
output		21.43

Variable	Select	Value
input		x.ÜL:0€ÁHDú/7Ú
domain		numbers
className		Figures
set		0-9, I, V, X, L, C, D, M
countLower		true
lowerSentence		Lower case letters should be counted.
precision		6
output		35.714286

Abb. 3. Dialog in zwei verschiedenen Zuständen zur Auswahl einer den Abhängigkeiten genügenden Wertbelegung

[4] Ableitungen ähneln dem Konzept der *VSpec derivations* in [7]

Der Benutzer arbeitet sich oben beginnend durch alle Vp durch. Wenn zu einer getätigten Auswahl Abhängigkeiten zu weiter unten stehenden Vp bestehen, schränkt der Dialog die unten auswählbaren Werte geeignet ein. Vp, die keine Auswahl erlauben (bspw. *output*) werden ohne Eingabemöglichkeit dargestellt. Abb. 3 zeigt, wie nach Wechsel der *domain* zu „numbers“ der Regler für *className* mit drei Auswahloptionen erscheint (erkennbar an den Einrastpositionen), während *set* nur noch zwei statt drei Optionen anbietet. Wir haben einen Schieberegler als Interaktionselement gewählt, weil dieser ressourcenschonend auch bei großen Wertemengen zu Beginn lediglich die Anzahl der Optionen kennen muss, um dann bei jedem Schiebeereignis den zur aktuellen Reglerposition zugehörigen Wert on-the-fly zu berechnen.

Bedingung für die Eignung des Schiebereglers ist, dass sich die Werte jedes Vp durch einen fortlaufenden Index aufzählen lassen und dass der zu einem gegebenen Index gehörige Wert effizient ermittelbar ist. Die im weiteren Verlauf beschriebene Spezifikation der Wertemenge erfüllt diese Bedingung.

B. *Variabilitätsmodell der Beispielaufgabe*

In Abb. 4 veranschaulichen wir für die obige Beispielaufgabe das im nachfolgenden Abschnitt C allgemein spezifizierte Datenmodell.

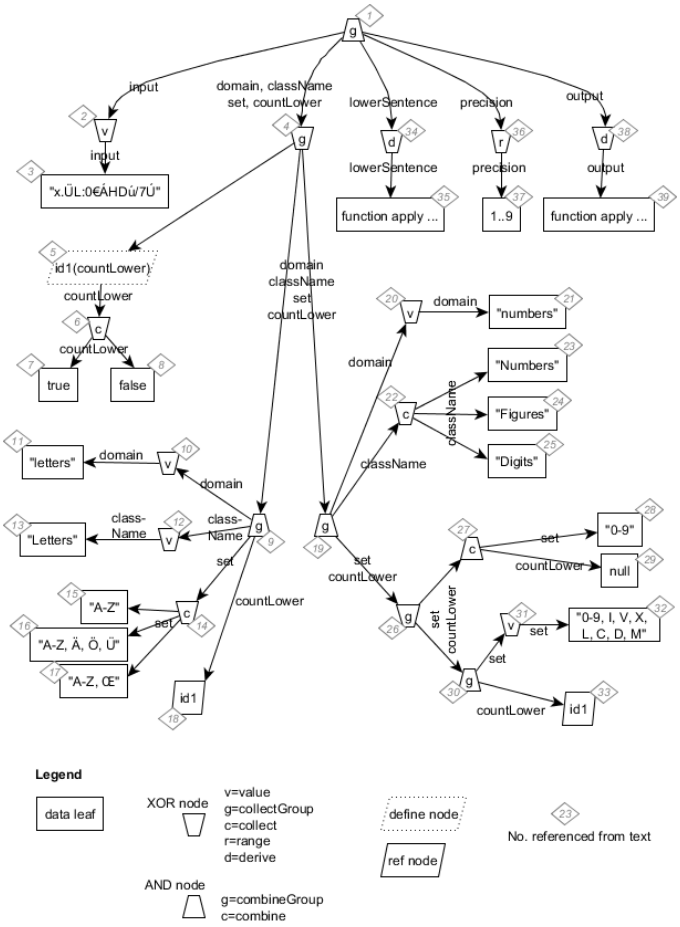

Abb. 4. AND-XOR-Baum zur Beschreibung der Vp der Beispielaufgabe. AND-Knoten repräsentieren kartesische Produkte, XOR-Knoten stehen für Vereinigungsoperationen und Einzelwertangaben.

Die fünf Kinder des Wurzel-AND-Knotens (Nr. 1) besagen, dass sich alle Werte aufzählen lassen, wenn man Wertemengen für die Vp (*input*), (*domain*, *className*, *set*, *countLower*), (*lowerSentence*), (*precision*) und (*output*) aufzählt und diese geeignet zu achtdimensionalen Tupeln zusammensetzt. Die Zusammensetzung muss alle Kindknoten einbeziehen, da es sich um einen AND-Knoten handelt. Sie beginnt links mit einer einelementigen und eindimensionalen Menge S_2 für *input* (XOR-Knoten 2). Die Menge S_2 wird (da Knoten 1 ein AND-Knoten ist) kartesisch mit der vierdimensionalen Menge S_4 für (*domain*, *className*, *set*, *countLower*) multipliziert ($S_2 \times S_4$). Der Inhalt von S_4 entsteht als Vereinigungsmenge (XOR) der Mengen $S_9 \cup S_{19}$, die ihrerseits kartesische Produkte sind. Besonders sind die Knoten 18 und 33, die ihren Wertebereich $S_{18}=S_{33}=\{true, false\}$ durch Verweis auf die Wertemenge des Knotens 5 erhalten, der die Wertemenge vorab unter dem Symbol „id1“ definiert[5].

Bewegen wir uns zu den Geschwistern des Knotens 4 nach rechts weiter. Der XOR-Knoten 34 repräsentiert eine Operation, die $S_2 \times S_4$ mit dem folgenden Javascript-Fragment (Knoten 35) für *lowerSentence* „kombiniert“:

```
function apply(o) {
  if (o.countLower===null) return "";
  if (o.countLower===true)
                return "Lowercase should be counted.";
  return "Lowercase should not be counted.";
};
```

Die Vorschrift zur Bildung der resultierenden sechsdimensionalen Menge ist kein einfaches kartesisches Produkt, da die Menge S_{34} nicht vorab bekannt ist. Die Bildungsvorschrift, die wir als ⓓerivation-Operation notieren, ist wie folgt: Für jedes Element e aus ($S_2 \times S_4$), füge (e, apply(e)) zum Ergebnis $(S_2 \times S_4)ⓓS_4$ hinzu. Weiter geht es mit dem XOR-Knoten 36. Die hierin als Wertebereich definierte eindimensionale Menge $S_{36}=\{1, …., 9\}$ für *precision* wird wieder kartesisch multipliziert ($S_2 \times S_4ⓓS_4 \times S_{36}$). Die letzte Operation für Knoten 38 gleicht der für *lowerSentence* mit einer hier aus Platzgründen nicht wiedergegebenen Javascript-Funktion für *output* ($S_2 \times S_4ⓓS_4 \times S_{36}ⓓS_{38}$).

C. UML-Klassenmodell

Nachdem wir das Beispiel durchgesehen haben, stellen wir nun das Datenmodell als UML-Klassendiagramm vor (Abb. 5). Wo inhaltlich Parallelen bestehen, haben wir uns bei der Benennung von Entitäten an [11] und [7] orientiert. Ein Variationspunkt (Vp) besitzt einen Namen (key) und einen Datentyp (VpT) mit optionaler Vergleichsgenauigkeit (accuracy). Eine Ausprägung des Vp ist eine Variante (V) mit einem Wert. Alle Vp einer Aufgabe zusammen sind in einem composite variation point (CVp) zusammengefasst. Eine Aufgabenschablone (tt, Abb. 2) enthält eine Komposition der Spezifikationen aller Vp der Aufgabe (CVSpec, composite variation specification). Ein CVSpec-Objekt ist Wurzelknoten einer Hierarchie von CVSpecNode-Objekten. Die Hierarchie beschreibt alle Vp-Werte durch geschachtelte Vereinigungsmengen (Collect…), kartesische Produkte (Combine…) und Wertangaben (Val).

[5] Mit dem Define-Knoten verwandt ist das Konzept von feature model references [2].

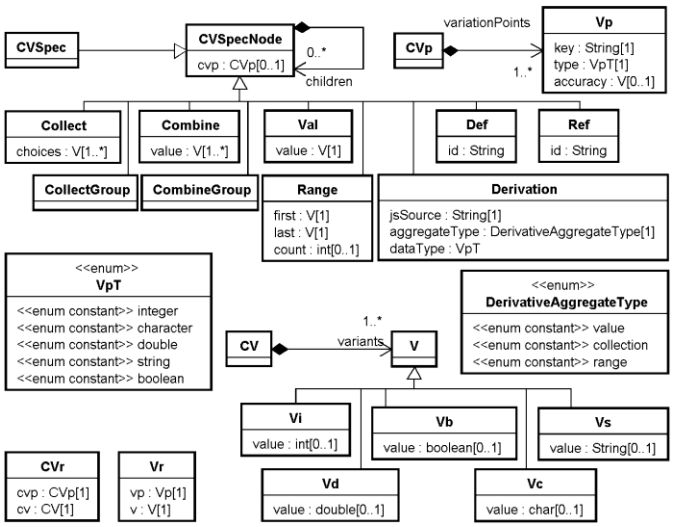

Abb. 5. UML-Diagramm des Datenmodells. Die Menge unterstützter Datentypen (VpT) ist beliebig erweiterbar.

Um Redundanz und lange Auflistungen zu vermeiden, ergänzen wir spezielle Definitions- und Referenzierungsknoten (Def, Ref), Bereichsknoten (Range) und Ableitungsknoten (Derivation[6]). Der Wurzelknoten besitzt mit dem Attribut cvp eine Spezifikation aller Vp der Aufgabe. Nachfolgerknoten speichern die für den jeweiligen Teilraum geltenden Vp[7].

Um alle Vp zwecks Erzeugung einer Aufgabeninstanz (ti, Abb. 2) aufzulösen, wird ein CVr-Objekt (composite variant resolution) benötigt. Dieses beinhaltet einen Vektor vom Typ CV, der je Vp die gewählte Variante speichert.

D. XML-Fragment

Um einen Implementierungseindruck zu zeigen, skizzieren wir in Abb. 6 einen Teil der XML-Repräsentation des Datenmodells der Beispielaufgabe aus Abschnitt III. Das vollständige Beispiel wird in [4] besprochen.

```
 9:  <v:combineGroup>
  :    <v:cvp>
  :      <v:vp key="domain" type="string"></v:vp>
  :      <v:vp key="className" type="string"></v:vp>
  :      <v:vp key="set" type="string"></v:vp>
  :      <v:vp key="countLower" type="boolean"></v:vp>
  :    </v:cvp>
10:    <v:val>
11:      <v:string value="letters"></v:string>
  :    </v:val>
12:    <v:val>
13:      <v:string value="Letters"></v:string>
  :    </v:val>
14:    <v:collect>
15:      <v:string value="A-Z"></v:string>
16:      <v:string value="A-Z, Ä, Ö, Ü"></v:string>
17:      <v:string value="A-Z, Œ"></v:string>
  :    </v:collect>
18:    <v:ref id="id1"></v:ref>
  :  </v:combineGroup>
```

Abb. 6. Fragment der XML-Repräsentation der CVSpec der Beispielaufgabe. Die führenden Zeilennummern korrespondieren mit den rautenförmig umrandeten Ziffern der Abb. 4.

[6] Details zum DerivativeAggregateType: s. [4].

[7] Das cvp-Attribut ist optional, da jeder Knoten die zugehörigen Vp leicht vom Mutterknoten ermitteln lassen kann. Verpflichtend ist cvp, wenn lokale Umordnungen von Vp in Kindknoten gewünscht sind, um die Spezifikation kompakter zu gestalten. Mehr Details: s. [5].

VI. Einsatz

Das Datenmodell wurde bereits für mehrere Java-Aufgaben für den Grader Graja prototypisch eingesetzt. Dabei wurden aus unserer bisherigen Lehrpraxis entnommene Aufgaben mit Vp ausgestattet und deren Wertemengen und Constraints als AND-XOR-Baum beschrieben. Das entsprechende, den Baum repräsentierende XML-Dokument wurde in die Graja-Aufgabe aufgenommen, die damit zu einer Aufgabenschablone wurde. Um einem „proof of concept“ nahe zu kommen, wurden Aufgaben unterschiedlichen Umfangs umgesetzt: Programmierung eines einzelnen Ausdrucks, einer (main-)Methode sowie einer ganzen Klasse. Dabei wurden alle in Abschnitt V beschriebenen Konzepte praktisch eingesetzt: Wertebereich, Werteaufzählung, kartesisches Produkt, Vereinigungsmenge, Ableitungsvorschrift, Definition und Referenzierung.

Zur dialogbasierten Eingabe (vgl. Abb. 3) sowie zur zufallsbasierten Auswahl einer gültigen Wertebelegung (cvr) wurde eine eigens implementierte, kleine Bibliothek eingesetzt, die unabhängig von der in der Aufgabe geforderten Programmiersprache und dem Grader ist. Die Bibliothek implementiert effiziente Datenstrukturen: Intervallbäume für intervallskalierte Vp, balancierte Suchbäume für ordinalskalierte Vp. Unter der praxisnahen Annahme, dass die Anzahl der Knoten im AND-XOR-Baum deutlich kleiner als die Anzahl gültiger Vp-Wertkombinationen ist, diskutiert [4] ausführlich die Effizienz dieser Datenstruktur und liefert damit einen Hinweis auf die prinzipielle Eignung des vorgeschlagenen Datenmodells für die effiziente Eingabe bzw. Auswahl von Vp-Werten.

Um die Aufgaben praktisch einsetzen zu können, wurde eine speziell auf Graja zugeschnittene „Instantiation engine“ implementiert [4]. Diese erzeugt derzeit Aufgabeninstanzen (ti) u. a. durch einfache Such- und Ersetzungsoperationen. Die Graja-„Grading engine“ bewertet Einreichungen zu Aufgabeninstanzen i. w. wie gewöhnliche Aufgaben.

Derzeit liegen Einsatzerfahrungen mit Graja im Labor vor, die sich nach unserer Einschätzung unmittelbar auf einen Einsatz im Feld übertragen lassen. Die Anbindung der Graja-Instantiation engine an das an der HS Hannover eingesetzte LMS Moodle und damit der Einsatz der variablen Aufgaben in einer realen Lehrveranstaltung mit vielen Studierenden steht noch aus. Ein dazu benötigter Grader-unabhängiger Instanziierungsdienst als Middleware soll demnächst entstehen.

VII. Ausblick

Das vorliegende Datenmodell soll für Grader-spezifische Erweiterungen geöffnet werden. LMS, Instanziierungsdienst und Grader könnten auf dieser Grundlage die Verantwortung für die *variation resolution* (Vr) jedes Vp aushandeln. So könnte das LMS die zufallsbasierte Vr eines Vp, dessen Wertemenge installationsabhängig nur dem Grader bekannt ist, jenem überlassen.

Zudem sind die Auswirkungen von Vp auf instanziierte Artefakte derzeit noch „ad hoc“ realisiert. Die Notation variabler Stellen in Artefakten könnte zumindest für einige häufig in Programmieraufgaben vorkommende Artefakte standardisiert werden.

Der derzeit in JavaFX realisierte Auswahldialog (Abb. 3) soll nach Javascript portiert werden, um einen breiten Einsatz in verschiedenen LMS zu erleichtern. Zudem soll er um die Möglichkeit einer manuellen Texteingabe erweitert werden, damit auch solche Vp effizient manuell aufgelöst werden können, die auf große, nur dem Grader bekannte, proprietäre Wertemengen zurückgreifen.

Schließlich sollen sukzessive weitere Aufgaben aus unserer Lehrpraxis auf der Basis des hier beschriebenen Datenmodells variabel umgestaltet und im Feld erprobt werden. Dabei sollen andere Grader und Programmiersprachen einbezogen werden.

References

[1] Brusilovsky, P.; Sosnovsky, S.: Individualized exercises for self-assessment of programming knowledge: An evaluation of QuizPACK, ACM Journal of Educational Resources in Computing, Vol. 5, No. 3, 2005.

[2] Czarnecki, K.; Kim, C.H.P.: Cardinality-based feature modeling and constraints: a progress report. In: Proceedings of the International Workshop on Software Factories at OOPSLA, San Diego, California, USA, 2005.

[3] Garmann, R.: Der Grader Graja. In (Bott, O. J. et. al. Hrsg): Automatisierte Bewertung in der Programmierausbildung. Waxmann, Münster, 2017.

[4] Garmann, R.: Spezifikation von Variabilität in automatisch bewerteten Programmieraufgaben, Bericht, Hochschule Hannover, http://nbn-resolving.de/urn:nbn:de:bsz:960-opus4-11893, 2018.

[5] Garmann, R.: Eine Java-Bibliothek zur Spezifikation von Variabilität in automatisch bewerteten Programmieraufgaben. Bericht, Hochschule Hannover. http://nbn-resolving.de/urn:nbn:de:bsz:960-opus4-11874, 2018.

[6] Garmann, R.; Heine, F.; Werner, P.: Grappa – die Spinne im Netz der Autobewerter und Lernmanagementsysteme. In (Pongratz, H. et. al. Hrsg.): DeLFI 2015: Die 13. e-Learning Fachtagung Informatik. Gesellschaft für Informatik, Bonn, S. 169-182, 2015.

[7] Haugen, Ø.: Common Variability Language (CVL) – OMG Revised Submission. OMG document ad/2012-08-05, 2012.

[8] Kashy, D.A.; Albertelli, G.; Ashkenazy, G.; Kashy, E.; Ng, H.-K.; Theonnessen, M.: Individualized interactive exercises: A promising role for network technology. In: Proc. of the 31st ASEE/IEEE Frontiers in Education Conference (Reno, NV), 2001.

[9] Krishna, A.K.; Kumar, A.N.: A problem generator to learn expression: evaluation in CSI, and its effectiveness, Journal of Computing Sciences in Colleges 16.4, S. 34-43, 2001.

[10] Otto, B.; Goedicke, M.: Auf dem Weg zu variablen Programmieraufgaben: Requirements Engineering anhand didaktischer Aspekte. In: Proc. of the ABP 2017. CEUR Workshop Proceedings, ceur-ws.org/Vol-2015/#ABP2017_paper_12, 2017

[11] Pohl, K.; Böckle, G.; van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, 2005.

[12] Radosevic, D.; Orehavocki, T.; Stapic, Z.: Automatic On-line Generation of Student's Exercises in Teaching Programming. In: Central European Conference on Information and Intelligent Systems, Varazdin, S. 87-93, 2010.

[13] Strickroth, S.; Striewe, M.; Müller, O.; Priss, U.; Becker, S.; Rod, O.; Garmann, R.; Bott, O.J.; Pinkwart, N.: ProFormA: An XML-based exchange format for programming tasks, in: eleed, Nr. 11, 2015.