8. Web Application Description Language. – Rezhim dostupa: http://www.w3.org/ Submission/wadl, svobodniy. – Zaglavie s ekrana. – Yaz. angl.

9. Web Service Modeling Ontology. – Rezhim dostupa: http://www.wsmo.org, svobodniy. – Zaglavie s ekrana. – Yaz. angl.

10. WSDL W3C specification. – Rezhim dostupa: http://www.w3.org/TR/wsdl20, svobodniy. – Zaglavie s ekrana. – Yaz. angl.

УДК 004.031.2:37+004.424

СПОСОБЫ ПРОВЕРКИ РЕШЕНИЙ ЗАДАНИЙ ПО ПРОГРАММИРОВАНИЮ В ОБУЧАЮЩИХ СИСТЕМАХ

***Катаев Александр Вадимович**, ассистент, Волгоградский государственный технический университет, 400131, Россия, г. Волгоград, пр. Ленина, 28, e-mail: alexander.kataev@gmail.com.*

***Шабалина Ольга Аркадьевна**, кандидат технических наук, доцент, Волгоградский государственный технический университет, 400131, Россия, г. Волгоград, пр. Ленина, 28, e-mail: O.A.Shabalina@gmail.com.*

***Камаев Валерий Анатольевич**, доктор технических наук, профессор, Волгоградский государственный технический университет, 400131, Россия, г. Волгоград, пр. Ленина, 28, e-mail: kamaev@cad.vstu.ru.*

При реализации обучающих систем одна из наиболее сложных возникающих задач – задача тестирования и оценки результатов обучения. В системах, обучающих программированию, решения заданий представляют собой фрагменты программного кода, записанные на некотором языке программирования. Для верификации, анализа и тестирования программ существует специализированные инструменты, однако их использование для проверки решений в обучающих системах не всегда приемлемо, поскольку они ориентированы на анализ промышленного кода. В статье изложены особенности программного кода как вида решений заданий и предложены способы проверки такого вида решений. Рассмотрены применение способов проверки текста (исходного кода) решений и результаты работы кода решения. Описаны особенности программного кода, которые снижают эффективность применения методов обработки текста, традиционно применяемых для проверки решений в обучающих системах. Предложены методы предварительной обработки исходного кода (нормализации текста решения), позволяющие повысить эффективность работы алгоритмов проверки текста решения. Предложено применять регулярные выражения для хранения множества эталонных решений и выполнения проверки текста решения. Рассмотрены варианты реализации проверки результата работы программного кода – выполнение кода с использованием существующего интерпретатора или же с использованием специально созданного интерпретатора, ориентированного на применение в обучающей системе. Предложен способ дополнения фрагментов программного кода до полноценной программы с использованием заготовленных фрагментов кода (контекста трансляции).

***Ключевые слова:** автоматизированные обучающие системы, проверка заданий, оценка уровня знаний, программный код, верификация программного обеспечения, модульное тестирование, регулярные выражения, нормализация программного кода, контекст трансляции, выполнение программного кода.*

ПРИКАСПИЙСКИЙ ЖУРНАЛ:
управление и высокие технологии № 3 (15) 2011

METHODS OF PROGRAMMING CODE CHECKING IN LEARNING SYSTEMS

***Kataev Alexander V.**, Lecturer, Volgograd State Technical University, 28 Lenin avenue, Volgograd, 400131, Russia, e-mail: alexander.kataev@gmail.com.*

***Shabalina Olga A.,** Cand.in Technics, Associate Professor, Volgograd State Technical University, 28 Lenin avenue, Volgograd, 400131, Russia, e-mail: O.A.Shabalina@gmail.com.*

***Kamaev Valery A.,** D.Sc. in Technology, Full Professor, Volgograd State Technical University, 28 Lenin avenue, Volgograd, 400131, Russia, e-mail: kamaev@cad.vstu.ru.*

One of the most complicated problems of educational systems developers is the problem of testing and evaluation of learning results. In educational systems aimed at learning programming languages, learning tasks require writing programming code. There exist specialized tools for programming code verification, analysis and testing, but they are mostly oriented on industrial software level. Peculiarities of programming code as a class of educational task solution are considered and approaches of programming code evaluation in educational systems are suggested. Approaches based on evaluation of a text of solution (source code) and based on code execution results are described. Peculiarities of programming code that lead to low effectiveness of using traditional methods of text verification are considered. Methods of preprocessing of source code (code normalization) for using in algorithms of evaluation of the text of program are suggested. Different ways of evaluation of code execution results are considered, based on ready-to-use interpreters, or development of special tools for using in educational system. The method of completing programming code with prepared program fragments (translation context) is suggested.

***Key words**: learning environments, task checking, knowledge level estimation, program code, software verification, unit-testing, regular expressions, program code normalization, translation context, program code execution.*

При реализации обучающих систем вообще и, в частности, обучающих игр одна из наиболее сложных возникающих задач – задача тестирования и оценки результатов обучения. Наиболее полно оценить знания обучаемого можно с помощью открытых тестов, ответы на которые представляют собой произвольный текст. Однако проверка таких тестов является трудной задачей, решение которой требует разработки специализированных методов проверки [1, 4, 8].

В системах, обучающих программированию, решения заданий представляют собой фрагменты программного кода, записанные на некотором языке программирования. Решение может кодировать некоторый алгоритм или описывать фрагмент, не являющийся полноценной программой (объявление переменной, объявление функции, описание типа данных и т.п.).

Задача проверки решения в обучающей системе по программированию может рассматриваться как частный случай задачи тестирования программного обеспечения или задачи формальной верификации [2, 3, 5]. Для выполнения верификации и тестирования программ существуют специализированные инструменты (статические анализаторы кода, модульного тестирования и др.). Однако использование этих инструментов для проверки решений в обучающих системах не всегда приемлемо, поскольку они ориентированы на анализ промышленного кода и не подходят для проверки кода, формируемого в рамках выполнения учебных заданий. Кроме того, такие системы, как правило, предполагают независимое использование (обладают пользовательским интерфейсом) или интеграцию со средами разработки программного обеспечения, и их интеграция в обучающую систему затруднена. Существуют разработки, применяемые для автоматизации проверки лабораторных работ [4], но они не подходят для решения задачи проверки небольших фрагментов кода.

Решения заданий могут вводиться в виде текста или составляться из отдельных блоков (лексем). Практически в обоих случаях могут применяться схожие методы проверки, с той лишь разницей, что для проверки текстового описания алгоритмы проверки должны работать с отдельными символами текста, а для проверки блочного решения – с отдельными лексемами.

Проверка решений может происходить двумя основными способами: проверка текста решений и проверка результатов работы решений. Текстом решений может являться как программный код, так и цепочки лексем.

Проверка текста решений. Простой способ реализации проверки текстовых решений, используемых в автоматизированных обучающих системах (например, в Moodle), основан на подготовленном разработчиком заданий посимвольном сравнении решений, вводимых обучаемым, с правильными решениями. Применение такого способа для проверки программного кода затруднено (и в общем случае не всегда применимо) из-за возможных различий в записи решения. Специфика заданий по программированию такова, что для одного задания могут существовать различные алгоритмы его решения, а один и тот же алгоритм может быть закодирован различными способами.

Кодировка одного и того же алгоритма на языке программирования может различаться форматированием (наличием комментариев, отступов, лишних переводов строк и пробелов), именами идентификаторов, записью математических выражений, порядком следования независимых конструкций кода и др. Пример различных кодировок одного и того же решения приведен в таблице 1.

Таблица 1

Пример задания и возможных решений

Задание: «Напишите код на языке C++, увеличивающий значение целочисленной переменной i на 1»	
Варианты решений	Комментарии
i = i + 1;	Используется оператор сложения
`i ++;`	Используется оператор инкремента
`i = i + 1u;`	Явное указание типа константы
`i = 1 + i;`	Другой порядок слагаемых
`int d = 1;` `i = i + d;`	Используется дополнительная переменная, имя переменной может быть любым

Возможные различия в записи одного и того же решения могут комбинироваться. Приведенные в примере варианты записи являются практически используемыми, поэтому они должны считаться правильными при проверке такого задания.

Учет различных способов записи операторов и величин, а также порядка следования независимых конструкций приводит к конечному количеству получаемых вариантов записи решений, что делает возможным описание и хранение всех таких вариантов. Однако с ростом сложности заданий количество вариантов решений растет экспоненциально (эффект комбинаторного взрыва). Это приводит к высоким затратам ресурсов компьютера для хранения правильных решений и выполнения проверки, а также высоким трудозатратам при составлении множества правильных решений, что ограничивает практическую возможность непосредственного хранения множества правильных решений.

Кроме того, в коде могут присутствовать различия, позволяющие получить практически бесконечное множество вариантов записи решений. К таким отличиям можно отнести форматирование кода и имена идентификаторов. Различия в форматировании, как будет показано ниже, могут быть достаточно легко устранены. Явное указание в задании возможных имен используемых идентификаторов позволяет избавиться от потенциально бесконечного количества решений, но может быть неприемлемым с точки зрения разработчика обучающих заданий.

К бесконечному количеству вариантов записи решений может привести использование некоторых других приемов. Например, можно подставить вместо константы некоторое математическое выражение («i=i+2-1») или использовать свойства математических операций («i=i+(i-i)+1»). Получаемые записи решений, хотя и корректны с точки зрения языка программирования, тем не менее могут считаться неправильными, поскольку не являются практически используемыми.

Усовершенствование проверки текста решения может достигаться за счет проведения предварительной обработки исходных данных – нормализации проверяемых решений и свертки множества правильных (эталонных) решений.

Нормализация текста решения состоит в применении к нему односторонних преобразований, не изменяющих его семантики. Поскольку к проверяемому и эталонному решению применяются одни и те же преобразования, то различия в записи этих решений сводятся к минимуму. Простейшая процедура нормализации программного кода состоит в удалении лишних пробельных символов и комментариев. Эти операции могут быть выполнены достаточно просто, например, с использованием регулярных выражений [6]. Достаточно просто могут быть заменены и некоторые идентичные конструкции языка (например, конструкция «<переменная>++;» может быть заменена на «<переменная>=<переменная>+1;»). Более сложные преобразования текста решения состоят в переименовании идентификаторов, преобразовании математических выражений, изменении порядка следования независящих друг от друга конструкций и др. Однако реализация таких преобразований намного сложнее, поскольку требует реализации практически полноценного анализатора кода.

Процедура нормализации должна применяться к решению непосредственно перед его сопоставлением с множеством эталонных решений. Множество эталонных решений может быть нормализовано один раз, в процессе разработки заданий. После нормализации множества эталонных решений в нем могут появиться совпадающие решения, которые могут быть удалены.

Для хранения эталонных решений и повышения скорости сопоставления могут быть использованы специальные алгоритмы и структуры данных, учитывающие схожесть большинства правильных решений – методы свертки. Простой, но достаточно эффективный метод свертки, основанный на использовании регулярных выражений, состоит в том, что для каждого задания строится шаблон, кодирующий весь набор эталонных решений. Построение такого шаблона может быть выполнено автоматически по набору эталонных решений. Сложность (и, соответственно, требования к памяти и вычислительным ресурсам) шаблонов сильно зависит от количества эталонных решений и алгоритма построения шаблона по набору решений. Основное преимущество использования регулярных выражений состоит в том, что для проверки соответствия решения регулярным выражениям могут использоваться существующие средства (например, библиотека PCRE или функции стандартной библиотеки среды исполнения .NET).

Проверка результата выполнения решения является другим возможным способом проверки решений, который применим только для решений, описывающих некоторый алгоритм. В зависимости от задания результатами работы могут быть либо данные (результаты математических расчетов), либо действия (последовательность вызова функций). Проверка работы может происходить один раз, по окончании выполнения программы или в процессе выполнения (проверка промежуточных значений переменных, контроль вызываемых подпрограмм).

В тестировании программного обеспечения такой подход называется модульным, или unit-тестированием [3, 4]. Существуют системы модульного тестирования промышленного кода и лабораторных работ. Эти системы в основном выполняют проверку возвращаемых функциями значений и не могут быть применены для контроля процесса работы решения.

Для выполнения программного кода может быть использован интерпретатор (или компилятор), используемый при практическом программировании, или же разработан специальный, ориентированный на применение в обучающей системе. Реализация своего интерпретатора может быть полезна, если необходимо в процессе выполнения программы формировать более подробные сообщения об ошибках, чем те, которые формируют существующие интерпретаторы, а также если в обучающей системе используется собственный язык программирования. С другой стороны, использование существующего интерпретатора для проверки заданий в обучающих системах позволит гарантировать, что написанные программы и описания возможных ошибок в них будут полностью соответствовать стандарту языка, что может быть полезно с точки зрения обучения.

В некоторых случаях проверяемый фрагмент может не являться корректным описанием программы, и его непосредственное выполнение не представляется возможным. Например, если требуется проверить решение задания, приведенного в таблице 1, то для реализации такой проверки необходимо дополнить код решения, как минимум, фрагментом, объявляющим переменную i и присваивающим ей начальное значение.

Фрагменты кода, дополняющие решение до полноценной программы, будем называть *контекстом трансляции*. В общем случае контекст трансляции должен включать предваряющий код (пролог), завершающий код (эпилог) и проект.

Пролог и эпилог контекста трансляции представляют собой фрагменты кода программы, дописываемые перед проверяемым фрагментом и после него соответственно. В совокупности с проверяемым фрагментом кода они формируют синтаксически корректную минимальную единицу трансляции (модуль, файл исходного кода), которая может быть скомпилирована (или интерпретирована). Пролог и эпилог могут содержать объявление переменных, получение исходных данных, описание дополнительных функций, включение модулей и другие элементы.

Проект контекста трансляции включает описание дополнительных единиц трансляции (модулей) программы, которые могут содержать функции и данные, необходимые для корректной работы проверяемого фрагмента программы. В отличие от пролога и эпилога, которые обязательно должны быть представлены в виде исходного кода, проект может состоять из предварительно транслированных модулей.

Для проверки приведенного выше примера (таблица 1) может быть использован контекст трансляции, приведенный в таблице 2.

Элементы контекста трансляции могут быть повторно использованы, однако в общем случае для каждого задания может потребоваться свой контекст трансляции. Необходимость подготовки контекста задания повышает трудоемкость описания заданий при использовании данного метода. Кроме того, для качественной проверки решений необходимо для каждого задания подготовить несколько наборов тестовых данных. С другой стороны, трудоемкость описания заданий при использовании проверки результата выполнения решения может оказаться существенно ниже, чем при проверке исходного текста, так как нет необходимости описывать все возможные алгоритмы решения и учитывать возможные различия в их записи.

Таблица 2

Пример контекста трансляции

Программный код	Описание

`#include "testdata.h"`
`int do_test() {`
`  int i, r;`
`  while (next_data(i,r)) {`
	Пролог. Содержит заголовок функции «do_test», которая получает тестовые данные, объявляет переменные и присваивает им начальные значения

`    i = i + 1;`	*Проверяемый фрагмент кода*

`    if ( i != r )`
`      return 1;`
`  }`
`  return 0;`
`}`
	Эпилог. Завершает описание функции, содержит проверку результата работы фрагмента. В общем случае результат может проверяться с использованием специальных правил

`#include "testdata.h"`
`int main() {`
`  init_data();`
`  return do_test();`
`}`
	Часть проекта – модуль проверки. Содержит главную функцию программы, которая инициализирует модуль получения данных (не показан) и вызывает функцию «do_test»

Рассмотренные способы могут быть реализованы при разработке автоматизированных обучающих и тестирующих систем, в том числе обучающих компьютерных игр. Выбор способа зависит от предполагаемых заданий, используемого языка программирования, в частности, в одной системе могут быть реализованы несколько способов проверки заданий. Рассмотренные способы используются для проверки решений заданий в обучающих играх по программированию [7, 8].

Список литературы

1. Воробкалов П. Н. Оценка качества электронных обучающих систем / П. Н. Воробкалов, В. А. Камаев // Управление большими системами : сб. тр. – 2009. – № 24. – С. 99–111.

2. Кларк Э. М. Верификация моделей программ. Model Checking / Э. М. Кларк, О. Грамберг, Д. Пелед. – М. : МЦНМО, 2002. – 416 с.

3. Криспин Л. Гибкое тестирование. Практическое руководство для тестировщиков ПР и гибких команд / Л. Криспин, Дж. Грегори. – М. : Вильямс, 2010. – 464 с.

4. Литовкин Д. В. Библиотека модульного тестирования, используемая при обучении программированию / Д. В. Литовкин, О. А. Сычев // Известия ВолгГТУ : межвуз. сб. науч. ст. – Волгоград, 2010. – Вып. 9, № 11. – С. 106–109. – (Сер. Актуальные проблемы управления, вычислительной техники и информатики в технических системах).

5. Синицын С. В. Верификация программного обеспечения / С. В. Синицын, Н. Ю. Налютин. – М. : Бином, 2008.– 368 с.

6. Фридл Дж. Регулярные выражения / Дж. Фридл. – СПб. : Питер, 2001. – 352 с.

7. Шабалина О. А. Обучение разработчиков программного обеспечения: применение компьютерных игр и процесса их разработки / О. А. Шабалина, А. В. Катаев, П. Н. Воробкалов // Известия ВолгГТУ : межвуз. сб. науч. ст. – Волгоград, 2010. – Вып. 9, № 11. – С. 117–124. – (Сер. Актуальные проблемы управления, вычислительной техники и информатики в технических системах).

8. Шабалина О. А. Разработка обучающих компьютерных игр для использования в вузе / О. А. Шабалина, П. Н. Воробкалов, А. В. Тарасенко, А. В. Катаев // Качество. Инновации. Образование. Европейский центр по качеству. – 2008. – № 4. – С. 14–16.

References

1. Clark E. M. Verifikaciya modelei programm. Model Checking / E. M. Clark, O. Gramberg, D. Peled. – M. : MCNMO, 2002. – 416 p.

2. Friddle J. Regulyarnye vyrazheniya / J. Friddle. – SPb. : Piter, 2001. – 352 p.

3. Krispin L. Gibkoe testirovanie. Prakticheskoe rukovodstvo dlya testirovshikov PR i gibkih komand / L. Krispin, J. Gregori. – M. : Vil'yams, 2010. – 464 p.

4. Litovkin D. V. Biblioteka modul'nogo testirovaniya, ispol'zuemaya pri obuchenii programmirovaniyu / D. V. Litovkin, O. A. Sychev // Izvestiya VolgGTU : mezhvuz. sb. nauch. st. – Volgograd, 2010. – Vyp. 9, № 11. – P. 106–109. – (Ser. Aktual'nye problemy upravleniya, vychislitel'noi tehniki i informatiki v tehnicheskih sistemah).

5. Shabalina O. A. Obuchenie razrabotchikov programmnogo obespecheniya: primenenie komp'yuternyh igr i processa ih razrabotki / O. A. Shabalina, A. V. Kataev, P. N. Vorobkalov // Izv. VolgGTU : mezhvuz. sb. nauch. st. – Volgograd, 2010. – Vyp. 9, № 11. – P. 117–124. – (Ser. Aktual'nye problemy upravleniya, vychislitel'noi tehniki i informatiki v tehnicheskih sistemah).

6. Shabalina O. A. Razrabotka obuchayushih komp'yuternyh igr dlya ispol'zovaniya v vuze / O. A. Shabalina, P. N. Vorobkalov, A. V. Tarasenko, A. V. Kataev // Kachestvo. Innovacii. Obrazovanie Evropeiskii centr po kachestvu. – 2008. – № 4. – P. 14–16.

7. Sinitzyn S. V. Verifikaciya programmnogo obespecheniya / S. V. Sinitzyn, N. Yu. Nalyutin. – M. : Binom, 2008. – 368 p.

8. Vorobkalov P. N. Ocenka kachestva elektronnyh obuchayushih sistem / P. N. Vorobkalov, V. A. Kamaev // Upravlenie bol'shimi sistemami : sb. tr. – 2009. – № 24. – P. 99–111.

УДК 004.428.4

ДЕМОНСТРАЦИЯ ПРОЦЕССА ПОИСКА ИНФОРМАЦИИ С ИСПОЛЬЗОВАНИЕМ ХЕШИРОВАНИЯ

***Смирнова Марина Олеговна**, кандидат педагогических наук, Астраханский государственный университет, 414056, Россия, г. Астрахань, ул. Татищева, 20а, e-mail: apsmir@yandex.ru.*

Представлен программный продукт, с помощью которого можно продемонстрировать применение хеширования, являющегося самым быстродействующим методом программного поиска. Хеширование применяется при работе с наборами данных большого размера (браузеры, словари, компиляторы и т.п.) и в криптографии.

Дано описание основных компонентов программного продукта, реализованного на языке программирования Object Pascal в среде визуального программирования Delphi 7, и возможностей использования при изучении разделов, связанных с алгоритмами поиска, построенных на основе хеш-таблиц. Моделирование процесса поиска информации с применением хеш-таблиц в данном программном продукте реализовано на примере построения телефонного справочника, при этом используются таблицы с закрытой и открытой адресацией как примеры открытого и закрытого хеширования.

Типами данных, на которых основана реализация алгоритмов, являются массивы и записи, предназначенные для хранения как самих данных, так и ключей к ним. При этом первый тип хеширования строится на взаимосвязи массива и односвязных списков. Односвязные списки реализуются с помощью указателей на записи. Каждому ключу, хранящемуся в массиве, соответствует указатель на свой список с данными. Второй тип хеширования целиком строится на использовании массива, предположительно имеющего элементов больше возможного количества данных и замкнутого в кольцо.

Программный продукт снабжен справочной системой, позволяющей познакомиться с основами хеширования и принципами работы самой программы.

Разработанный программный продукт обеспечивает наглядными материалами поддержку тем, связанных с алгоритмами поиска, построенных на основе хеш-таблиц.