9. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы. Построение и анализ / Пер. с англ. – 2-е изд. – М.: Вильямс, 2005. – 1296 с.
10. Holland J.P. Adaptation in Natural and Artificial Systems. – The University of Michigan Press, 1975.
11. Mitchell M. An Introduction to Genetic Algorithms. – MA: MIT Press, 1996.
12. Timus Online Judge. Архив задач с проверяющей системой [Электронный ресурс]. – Режим доступа: http://acm.timus.ru, свободный. Яз. рус., англ. (дата обращения 21.09.2010).
13. Задача «Ships. Version 2» [Электронный ресурс]. – Режим доступа: http://acm.timus.ru/problem.aspx?space=1&num=1394, свободный. Яз. рус., англ. (дата обращения 21.09.2010).
14. Pisinger D. Algorithms for Knapsack Problems: PhD. Thesis. – University of Copenhagen, 1995.
15. Koza J.R. Genetic programming: On the Programming of Computers by Means of Natural Selection. – MA: The MIT Press, 1998.

Буздалов Максим Викторович – Санкт-Петербургский государственный университет информационных технологий, механики и оптики, студент, mbuzdalov@gmail.com

УДК 004.05

СОВМЕСТНОЕ ПРИМЕНЕНИЕ КОНТРАКТОВ И ВЕРИФИКАЦИИ ДЛЯ ПОВЫШЕНИЯ КАЧЕСТВА АВТОМАТНЫХ ПРОГРАММ

А.А. Борисенко, В.Г. Парфенов

При создании систем со сложным поведением важную роль играет контроль качества разрабатываемых программ. Цена ошибки в таких системах может быть слишком велика, поэтому важно не просто проверить соответствие создаваемой программы всем предъявленным к ней требованиям, но и сделать этот процесс эффективным, максимально автоматизировав его. На практике этого можно добиться, формализовав все требования к программе и храня полученную исполнимую спецификацию непосредственно вместе с кодом программы.

Рассмотрены существующие методы контроля качества современных программных систем и автоматных программ, а также описан процесс создания среды, позволяющей поддержать сразу три подхода к проверке качества программ с явным выделением состояний: проверку на модели, модульное тестирование и контракты. Предложенный подход позволяет сохранить корректность записи сформулированных требований при изменении самой программы, а также интерактивно контролировать ее качество.

Ключевые слова: контроль качества, соответствие спецификации.

Введение

Качественное программное обеспечение (ПО) – это, прежде всего, надежное программное обеспечение. Зачастую системы, требующие высокого уровня надежности, представляют собой системы со сложным поведением [1], а цена ошибки в таких проектах может быть слишком высокой [2]. При разработке систем со сложным поведением важное место занимает стадия тестирования, а одним из распространенных методов разработки таких систем является автоматное программирование [1].

Другой важной чертой современных программных проектов является их частое изменение: модифицируются требования к системе, находятся и исправляются ошибки. Для контроля качества ПО, соответствия его реализации и спецификации, в современных проектах используется ряд методов: ручное и автоматизированное тестирование, контрактное программирование и верификация [3].

Контроль качества автоматных программ

К автоматным программам могут быть успешно применены следующие методы анализа корректности [4]: тестирование, верификация (проверка на модели и доказательная верификация), контракты. Рассмотрим последовательно каждый из них, выделяя при этом характерные для данного метода преимущества и недостатки.

Тестирование. Тестирование – процесс выявления ошибок в ПО. Запуск программы на определенных входных данных, а также проверка различных сценариев выполнения позволяют достаточно быстро (по сравнению с другими методами поиска ошибок) убедиться в корректности обработки заданных сценариев [5].

Тестирование, применяемое после окончательного написания программы, не способно найти все ошибки. Как заметил Э. Дейкстра, если при тестировании ошибки в программе не найдены, это еще не означает, что их там нет.

Верификация. *Артефактами* жизненного цикла ПО называются различные информационные сущности, документы и модели, создаваемые или используемые в ходе разработки и сопровождения ПО [6].

Верификация проверяет соответствие одних создаваемых в ходе разработки и сопровождения ПО артефактов другим, ранее созданным или используемым в качестве исходных данных, а также соответствие этих артефактов и процессов их разработки правилам и стандартам. В частности, верификация проверяет соответствие между нормами стандартов, описанием требований (техническим заданием) к ПО, проектными решениями, исходным кодом, пользовательской документацией и функционированием самого ПО.

Формальная верификация. Формальная верификация представляет собой процесс доказательства с помощью формальных методов корректности или некорректности алгоритмов, программ и систем в соответствии с заданным описанием их свойств. Она требует высококвалифицированных специалистов в области формальных доказательств и логического вывода. В общем случае задача, решаемая в рамках этого подхода, является алгоритмически неразрешимой. При этом весь процесс формального доказательства связан с огромной ручной работой, что делает его малоприменимым на практике [3].

Верификация на модели. Под верификацией на модели понимают метод формальной верификации, позволяющий проверить, удовлетворяет ли заданная модель системы формальным спецификациям. Применение данного подхода позволяет для заданной модели поведения системы с конечным (возможно, очень большим) числом состояний проверить выполнимость некоторого требования (спецификации), которое обычно формулируется в терминах языка темпоральной логики (LTL, CTL и т.д.). Таким образом, можно проверить не только условия в определенном состоянии системы, но и историю развития во времени [7].

Метод верификации на модели хорошо подходит для проверки поведения автоматных программ. Это связано с тем, что при верификации на модели, обычно, по программе строится модель Крипке [8], которая фактически является специальным видом автомата.

Ощутимым неудобством верификации на модели является необходимость ручного ввода темпоральных спецификаций и описания модели программы на языке, понятном программе-верификатору.

Контракты. Под контрактами обычно понимают совокупность способов формализации и проверки требований к программе, в которую входят инварианты, постусловия и предусловия [9].

С помощью инвариантов удобно отслеживать выполнение требований к работе программы, которые должны исполняться на непрерывном отрезке ее жизненного цикла. При этом сами инварианты остаются недоступными для людей, не участвовавших в разработке кода, а тем более для тех, кто будет заниматься поиском ошибок в программе [10]. Важно отметить, что в современных средах разработки отсутствует поддержка автоматической проверки контрактов для создаваемой автоматной программы. Это усложняет применение данного подхода на практике.

Формализация требований к автоматной программе

Требования, предъявляемые к разрабатываемым программам, обычно формируются словесно. Для автоматической проверки их необходимо формализовать. В качестве примера, наглядно иллюстрирующего проблему формализации вербальных требований к форме, пригодной для автоматической проверки, рассмотрим модель холодильника. Зададим список предъявляемых к ней требований:

1. при закрытой дверце внутренняя лампа не должна гореть;
2. при выходе показаний датчика напряжения за пределы допустимого диапазона управляющее реле выключит питание;
3. в момент отключения охлаждающего элемента показания термостатов не должны превышать допустимые;
4. пока дверца не будет открыта, лампа не будет включена;
5. если открыть дверцу холодильника, прогреть его основной объем и закрыть дверь, то: будет включена лампа; при достижении критической температуры будет включен охлаждающий элемент; лампа будет выключена.

Приведенный список требований можно условно разделить на три группы. В первую группу отнесем требования 1–3. Они подходят для записи контрактов. Первое из них, «При закрытой дверце внутренняя лампочка не должна гореть», является инвариантом – условием, которое должно соблюдаться в течение всего процесса выполнения программы. В требовании 2 проверяется выполнимость некоторого условия при выходе из заданного состояния, а в требовании 3 – при входе в заданное состояние. Формализуем их при помощи пост- и предусловий, оперирующих показаниями термодатчиков. Ко второй группе можно отнести требование 4: «Пока дверца не будет открыта, лампочка не будет включена». Оно содержит выражение, зависящее от последовательности выполнения программы во времени, и его можно компактно записать в качестве темпоральной спецификации: (*Лампа_не_включена* **U** *Дверь_открыта*).

Ее можно использовать для верификации на модели. Занесем полученные результаты в таблицу, используя в формальной записи инварианты (**Inv**), постусловия (**Post**) и предусловия (**Pred**).

Заметим, что не всякую темпоральную спецификацию удается легко верифицировать. Предположим, что проверке подлежит требование «Лампа в холодильнике зажигается строго после открытия двери». В таком случае придется исключить в темпоральной спецификации все возможные переходы, до-

пустимые заданной автоматной моделью при открытии двери за исключением собственно зажигания лампы. Полученная формула стала бы громоздкой и перестала бы быть удобочитаемой. Более того, полученная таким образом спецификация полностью становилась бы привязанной к конкретной реализации модели, а это обстоятельство затруднило бы процесс контроля качества программы при изменении ее реализации.

1. При закрытой дверце внутренняя лампочка не должна гореть	**Inv** [DoorClosed]: lamp.*isTurnedOff*
2. При выходе показаний датчика напряжения из допустимого диапазона управляющее реле выключит питание	**Post** [VoltageOutOfRange]: powerAdapter.*isTurnedOff*
3. В момент отключения охлаждающего элемента показания термостатов не должны превышать допустимые значения	**Pred** [FreezerTurnedOff]: thermoSensor.*valuesInRange*
4. Пока дверца не будет открыта, лампочка не будет включена	lamp.*isTurnedOff* **U** DoorOpened

Таблица. Примеры записи формальных спецификаций

Для описанного случая хорошо подходит тестирование. Именно к нему целесообразно прибегнуть и при проверке последнего, 5-го требования. Оно представляет собой сценарий исполнения. Его легко записать и проверить, используя, например, модульное тестирование.

Контроль качества автоматных программ

В настоящей работе предлагается объединить достоинства сразу трех подходов для проверки качества программ с явным выделением состояний:

- проверка спецификаций (проверка на модели);
- проверка предусловий и постусловий, инвариантов (контракты);
- unit testing (модульное тестирование).

В результате процесс создания автоматных программ (с внедренными в него указанными подходами) становится эффективным, благодаря сочетанию возможностей современной среды разработки (статические проверки, рефакторинг, автодополнение и т.д.) и семантической проверки автоматного кода.

Был предложен новый подход к разработке автоматных программ [11]. Он использует мультиязыковую среду MPS (http://www.jetbrains.com/mps), для создания новых языков и расширения уже существующих, созданных с ее помощью. Языки, разрабатываемые с помощью MPS, не являются текстовыми в традиционном понимании, так как пользователь не пишет текст программы, а вводит ее в виде абстрактного синтаксического дерева (АСД). Такой подход позволяет обойтись без создания лексических и синтаксических анализаторов при создании новых языков, а также настроить преобразования АСД в код на конкретном языке программирования и задать удобную среду для его редактирования.

Отдельно отметим наличие в MPS языка stateMachine, позволяющего для Java-класса, реализующего сложное поведение, добавить автоматную модель, указав набор состояний и переходов между ними.

Реализация предложенного подхода

Для добавления верификации и проверки контрактов в процесс разработки автоматных программ потребуется:

- создать в среде MPS язык темпоральных спецификаций и контрактов, который был назван stateSpec, и описать операторы языка темпоральной логики LTL;
- настроить систему типов темпоральных операторов;
- внедрить секцию со спецификацией в описание автоматных программ на языке stateMachine;
- разработать плагин для запуска внешнего верификатора NuSMV (http://nusmv.irst.itc.it/);
- указать сочетания клавиш, запускающие верификацию и проверку контрактов;
- преобразовать автоматную модель к форме, пригодной для автоматической верификации, и настроить обработку полученных данных верификации;
- выделить сообщение о результатах верификации;
- в случае обнаружения контрпримера заменить исходные названия состояний и событий в цепочке исполнения программы, приводящей к нарушению спецификации или контракта.

Поддержка контрактов

С помощью языка stateSpec можно переформулировать заданные к автоматной программе контракты на языке LTL, а затем провести их статическую проверку, используя все тот же верификатор NuSMV. Таким образом, пользователь сможет записывать и проверять темпоральные спецификации или контракты в тех случаях, когда это будет удобнее.

Следует отметить, что верификатор NuSMV работает со своим языком описания автоматных программ. При этом, как уже отмечалось выше, преобразование модели к этому языку вручную чревато появлением привнесенных ошибок. Поэтому преобразование автоматной модели, созданной в среде MPS к форме, понятной верификатору NuSMV, необходимо автоматизировать.

Внедрение полученных результатов

Идея интеграции сразу трех подходов к контролю качества автоматных программ, предложенная в данной работе, была реализована в мультиязыковой среде MPS в виде плагина (дополнительной функциональности, доступной при нажатии сочетания клавиш). Для практического внедрения разработанного в рамках данной работы инструментального средства была выбрана программа учета дефектов YouTrack, разработанная в компании ООО «ИнтеллиДжей Лабс» (работает на мировом рынке под брендом JetBrains).

Программа YouTrack представляет собой Интернет-приложение для работы с базами дефектов. Она реализуется на базе системы JetBrains MPS. При генерации автоматов, работающих на сервере используется Java, в то время как пользовательский интерфейс для работы через браузер основан на JavaScript. Это обстоятельство привело к тому, что язык stateSpec получил отдельную реализацию для работы с программами на JavaScript. С его помощью были формализованы и проверены требования, описывающие логику поведения интерфейса программы YouTrack.

Заключение

В современных проектах вся сопутствующая информация должна обновляться в процессе разработки программы. Это относится и к спецификации. Хранить ее «рядом» с кодом недостаточно. Необходимо сделать ее исполнимой, т.е. внедрить автоматический контроль качества программы в процесс ее разработки. Спецификация, которая не отвечает данному требованию, не является эффективной еще и потому, что она никак не реагирует на изменение модели программы и быстро устаревает.

В данной работе предложен метод, позволяющий разрабатывать надежные объектно-ориентированные системы с явным выделением состояний. При этом, в частности, получены следующие результаты:

- предложен подход к программированию по контрактам (контрактному программированию) с явным выделением состояний. При этом отдельно рассмотрены внешние и внутренние контракты;
- разработан метод, позволяющий выбирать оптимальный способ формализации спецификаций к автоматным системам в зависимости от характера спецификации и особенностей автоматной системы;
- предложен способ интеграции действий по обеспечению соответствия реализованной автоматной системы спецификации в процесс разработки программного обеспечения;
- реализована часть функциональности инструмента, позволяющего проводить верификацию автоматной системы во время разработки.

В целях создания среды разработки надежных программ, реализующих системы со сложным поведением, в работе предложено совместить сразу несколько подходов к проверке качества ПО. Тестирование и верификация на модели были внедрены в процесс разработки автоматных программ в среде MPS. Также была изучена роль контрактов в преобразовании требований к программе, сформулированных словесно, к форме, пригодной для автоматической проверки.

В качестве направления дальнейших исследований можно предложить итеративную верификацию, позволяющую существенно ускорить получение результата, опираясь на данные предыдущих процессов верификации.

Исследование выполнено по Федеральной целевой программе «Научные и научно-педагогические кадры инновационной России на 2009–2013 годы» в рамках государственного контракта П2373 от 18 ноября 2009 года.

Литература

1. Поликарпова Н.И., Шалыто А.А. Автоматное программирование. – СПб: Питер, 2009 [Электронный ресурс]. – Режим доступа: http://is.ifmo.ru/books/_book.pdf, своб.
2. Риган П., Хемилтон С. NASA: миссия надежна // Открытые системы. – 2004. – № 3. – С. 12–17. [Электронный ресурс]. – Режим доступа: http://www.osp.ru/text/302/184060.html, своб.

3. Отчет по государственному контракту о верификации автоматных программ. Второй этап. – СПбГУ ИТМО, 2007 [Электронный ресурс]. – Режим доступа: http://is.ifmo.ru/verification/_2007_02_report-verification.pdf, своб.
4. Поликарпова Н.И. Объектно-ориентированный подход к моделированию и спецификации сущностей со сложным поведением. – СПб: СПбГУ ИТМО, 2006 [Электронный ресурс]. – Режим доступа: http://is.ifmo.ru/papers/oosuch, своб.
5. Веденеев В.В. Автоматизация тестирования использования программных интерфейсов приложений на основе моделирования конечными автоматами. – СПбГУ ИТМО, 2007 [Электронный ресурс]. – Режим доступа: http://is.ifmo.ru/testing/vedeneev/, своб.
6. Винниченко И.В. Автоматизация процессов тестирования. – СПб: Питер, 2005.
7. Курбацкий Е.А., Шалыто А.А. Верификация программ, построенных при помощи автоматного подхода // Материалы международной научно-методической конференции «Высокие интеллектуальные технологии и инновации в образовании и науке». – СПбГПУ. 2008. – С. 293–296 [Электронный ресурс]. – Режим доступа: http://is.ifmo.ru/download/2008-02-25_politech_verification_kurb.pdf, своб.
8. Кларк Э.М., Грамберг О., Пелед Д. Верификация моделей программ. Model Checking. – М.: МЦНМО, 2002.
9. Мейер Б. Объектно-ориентированное конструирование программных систем. – М.: Интернет-университет информационных технологий, 2005.
10. Мейер Б. Семь принципов тестирования программ // Открытые системы. – 2008. – № 7. – С. 13–29 [Электронный ресурс]. – Режим доступа: http://www.osp.ru/os/2008/07/5478839/, своб.
11. Кузьмин Е.В., Соколов В.А. Моделирование спецификация и верификация «автоматных» программ // Программирование. – 2008. – № 1. – С. 2–5 [Электронный ресурс]. – Режим доступа: http://is.ifmo.ru/download/2008-03-12_verification.pdf, своб.
12. Гуров В.С., Мазин М.А., Шалыто А.А. Текстовый язык автоматного программирования // Тезисы докладов Международной научной конференции, посвященной памяти профессора А.М. Богомолова «Компьютерные науки и технологии». – Саратов: СГУ. 2007. – С. 66–69 [Электронный ресурс]. – Режим доступа: http://is.ifmo.ru/works/_2007_10_05_mps_textual_language.pdf, своб.

Борисенко Андрей Андреевич – Санкт-Петербургский государственный университет информационных технологий, механики и оптики, студент, Andrey.Borisenko@gmail.com

Парфенов Владимир Глебович – Санкт-Петербургский государственный университет информационных технологий, механики и оптики, доктор технических наук, профессор, декан, parfenov@mail.ifmo.ru

УДК 004.4’242

ВИРТУАЛЬНАЯ ЛАБОРАТОРИЯ ОБУЧЕНИЯ МЕТОДАМ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА ДЛЯ ГЕНЕРАЦИИ УПРАВЛЯЮЩИХ КОНЕЧНЫХ АВТОМАТОВ

А.С. Тяхти

Описывается структура и возможности виртуальной лаборатории для обучения генетическому и автоматному программированию, реализованной на языке C#. Описываются основные этапы создания собственных подключаемых модулей лаборатории.

Ключевые слова: автоматное программирование, виртуальная лаборатория.

Введение

Парадигма автоматного программирования [1, 2] была предложена в 1991 году в России. В рамках данной концепции программа рассматривается как набор конечных автоматов, объектов управления и поставщиков событий.

Зачастую построение конечных автоматов эвристическими методами является затруднительным. В связи с этим для их генерации можно использовать автоматизированные методы, в том числе генетические алгоритмы и генетическое программирование [3].

Для обучения генетическому программированию ранее была создана виртуальная лаборатория для языка программирования Java [4]. Однако в указанной виртуальной лаборатории отсутствовала возможность применения других методов искусственного интеллекта, таких, как, например, метод имитации отжига [5–9]. Эта техника оптимизации использует случайный поиск на основе аналогии с процессом образования в веществе кристаллической структуры с минимальной энергией, происходящем при охлаждении этого вещества.

На основании изложенного можно сделать вывод о том, что актуальной является разработка виртуальной лаборатории для обучения методам искусственного интеллекта. При этом важным требованием