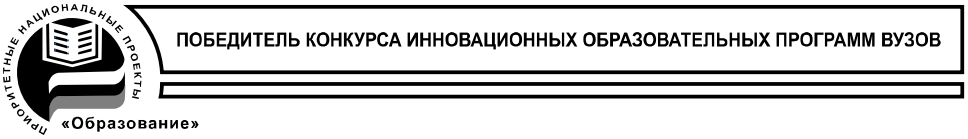
МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ



НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК

САНКТ-ПЕТЕРБУРГСКОГО ГОСУДАРСТВЕННОГО УНИВЕРСИТЕТА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

Выпуск 53

АВТОМАТНОЕ ПРОГРАММИРОВАНИЕ

САНКТ-ПЕТЕРБУРГ

2008

ISSN 1819-222X

Адрес: 197101, Санкт-Петербург, Кронверкский пр., 49, СПбГУ ИТМО

Телефон: (812) 233 12 70

Факс: (812) 233 12 70

http: //books.ifmo.ru/ntv

E-mail:karmanov@mail.ifmo.ru

Подписано к печати 10.04.2008 Тираж 100 экз. Заказ № 1(53)

Отпечатано в Центре распределенных издательских систем
Адрес: 197101, Санкт-Петербург, Кронверкский пр., 49

---

1 ВВЕДЕНИЕ В АВТОМАТНОЕ ПРОГРАММИРОВАНИЕ

УДК 004.4’2

ПАРАДИГМА АВТОМАТНОГО ПРОГРАММИРОВАНИЯ

А.А. Шалыто

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В статье приводятся основные положения автоматного программирования и обосновываются преимущества его использования при разработке программного обеспечения. Описываются инструментальные средства для поддержки автоматного программирования и приводятся примеры успешного применения предлагаемого подхода на практике. В статье описывается процедурное программирование с явным выделением состояний и объектно-ориентированное программирование с явным выделением состояний. Автоматные программы могут быть эффективно верифицированы, а для их построения в ряде случаев могут быть применены генетические алгоритмы.

Ключевые слова: парадигма программирования, автоматное программирование, программирование с явным выделением состояний

Введение

Большинство программистов-практиков считает, что в программировании нет особых проблем. «Отсутствие» проблем приводит к тому, что на практике при создании программного обеспечения (ПО) в большинстве случаев используются частные (*ad hoc* – экспромт или спонтанное решение) подходы, основанные на опыте программистов. Если трудности при создании программ и возникают, то их смиренно считают «неизбежным злом профессии». Тот факт, что при таком подходе достаточно много проектов заканчивается неудачно, не изменяет точку зрения большинства.

Принципиально другое мнение у теоретиков программирования, которые еще в 1968 г. «открыто признали кризис программного обеспечения» [1]. Однако в настоящее время это иногда оспаривается. Так, например, профессора Н. Вирт и Ю. Гутхнехт на пресс-конференции, посвященной избранию в 2005 г. создателя «Паскаля» Почетным доктором СПбГУ ИТМО (http://is.ifmo.ru/belletristic/_wirth_poch.pdf), утверждали, что они не видят проблем в программировании, оговорившись, правда, что сказанное не относится к программированию драйверов, которые обладают сложным поведением (отметим, что именно вопросам реализации такого поведения и посвящена настоящая статья). Несмотря на наличие у некоторых влиятельных ученых таких взглядов, многие теоретики считают, что указанный кризис продолжается, и они стали искать выход из него в переходе от «искусства программирования» [2] к *программной инженерии (Software Engineering)* [3, 4], которой, в частности, активно занимается «наследник» Н. Вирта по кафедре в *ETH* (Цюрих) Б. Мейер (http://is.ifmo.ru/belletristic/_meyer.pdf).

Несмотря на большое число работ по методологиям разработки ПО [5], проводимых, в том числе, и в настоящее время [6], считается [7], что указанный кризис не миновал. Он во многом связан с тем, что специалисты по программной инженерии «варятся в собственном соку» и почти не используют подходы, разработанные в других инженерных областях. Это привело к созданию сообщества исследователей и практиков, озабоченных будущим программной инженерии, которые особое внимание уделяют междисциплинарным исследованиям и подходам, в особенности тем из них, которые созданы в «не–ИТ»-дисциплинах задолго до появления компьютеров (*Interdisciplinary Software Engineering Network – ISEN*). При этом, например, по аналогии с архитектурой, родились паттерны проектирования.

В ходе указанных исследований сформировалось мнение [8], что при разработке ПО, видимо, может быть полезен *опыт создания систем автоматического управления* и, возможно, целесообразно сделать шаг назад и обратиться к трудам основоположников кибернетики, таких как, например, Н. Винер, Д. фон Нейман, У. Эшби. В подтверждение сказанному родился термин *программная кибернетика (Software Cybernetics)* [8], а первый международный семинар в этой области (*The First International Workshop on Software Cybernetics*), который после этого *стал ежегодным*, прошел в *2004* г.

Ниже излагаются основы *автоматного программирования,* разрабатываемого автором с *1991* г. [9]. *Исследования в области автоматного программирования относятся как к программной инженерии, так и к программной кибернетике* и базируются на идеях теории автоматов и теории автоматического управления – двух из трех составляющих, на которых, по мнению Н. Винера, базируется кибернетика [10]. При этом особо подчеркнем, что под автоматным программированием автором понимается *не* программирование с применением автоматов, а соответствующая технология программирования, направленная на создание систем со сложным поведением [11]. В этом смысле уместно провести параллель между автоматами и автоматным программированием, с одной стороны, и *UML* (*Unified Modeling Language*) и *RUP* (*Rational Unified Process*) – с другой. Так, автоматы и *UML* – это нотации, в то время как автоматное программирование и *RUP* – это процессы, использующие указанные нотации.

В заключение раздела отметим, что излагаемый подход близок к подходу Д. Харела [12], о котором Ф. Брукс в работе [5], сказал, что «он может оказаться революционным».

Автоматное программирование как стиль программирования

Автоматное программирование является одним из стилей программирования [13]. В работе [13] также отмечено, что этот термин был предложен в работе [11].

Упрощенная трактовка автоматного программирования состоит в том, что это стиль программирования, при использовании которого поведение программ предлагается описывать автоматами, которые в дальнейшем преобразуются в код. При этом отметим, что автоматы давно и успешно применяются в программировании, например, при построении компиляторов [14, 15]. Автоматы используются также и при решении многих других задач, например, при реализации протоколов.

В работе [16] подход с использованием автоматов для описания поведения программ был определен как стиль программирования, названный «программирование от состояний». В этой работе отмечалось, что «программирование от состояний – пожалуй, самый старый стиль программирования, так как на него наталкивает само устройство существующих вычислительных машин, которые представляют собой гигантские конечные автоматы». В плане выделения указанного подхода в качестве самостоятельного стиля программирования на авторов работы [16] повлияла работа [11], на которую они ссылаются как на современную методику программирования от состояний. В работе [11] было предложено применять автоматы в программировании не от случая к случаю, как одну из моделей дискретной математики, а как универсальный подход, который целесообразно использовать практически всегда, когда программа должна обладать достаточно сложным поведением, и, в особенности, реактивным [17] – реагировать по-разному на события в зависимости от состояний, в которых находится программа.

Отметим, что, несмотря на работы Д. Харела (лауреата *ACM Software Systems Award 2007*), даже реактивные системы проектируются автоматно далеко не всегда. Об этом, в частности, свидетельствует появление только в 2007 г. работ ведущего специалиста *IBM* Э. Принга [18, 19] о реализации медленно всплывающих подсказок с приме-

нением автоматов. Однако даже в этом случае нельзя говорить о применении технологии автоматного программирования, так как переход от модели к программе выполнялся эвристически, что привело к их расхождению.

В заключение раздела отметим, что «программирование с автоматами» нельзя рассматривать как парадигму программирования, так как при этом остается неясным, как с использованием автоматов проектировать и реализовывать программы в целом.

Автоматное программирование как парадигма программирования

Очень многие системы, которые для внешнего наблюдателя ведут себя достаточно осмысленно, являются автоматизированными объектами управления. *Автоматизированный объект* управления *представляет собой совокупность системы управления (СУ) и объекта управления (ОУ), охваченных обратными связями* (рис. 1).

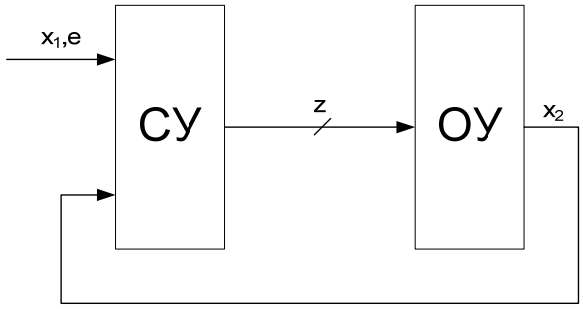


Рис. 1. Автоматизированный объект управления

Задача построения автоматизированных объектов управления рассматривается в любом курсе теории автоматического управления применительно к объектам различных типов. Удивительно, что это почти никак не коснулось практики программирования, несмотря на то, что в теории алгоритмов в качестве одной из основных моделей используется машина Тьюринга (рис. 2).

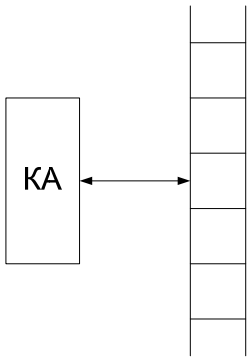


Рис. 2. Машина Тьюринга

Машина Тьюринга, по сути, является автоматизированным объектом управления [20], в котором система управления – конечный автомат, а объект управления – лента (ее ячейки памяти) (рис. 3). Сложность программирования на машине Тьюринга (например, умножение 2 на 3 требует 66 команд [21]) определяется тем, что ней используются очень простые объекты управления (ячейки памяти), которые могут выполнять только простейшие действия (операции) по сдвигу головки и записи и стиранию отдельных символов. В этой ситуации вычисления, в некотором смысле, прихо-

дится выполнять конечному автомату, который для этой цели не приспособлен, так как его предназначение – управление. Другая особенность машины Тьюринга, которая может резко усложнять программы – это использование только одного автомата, которого достаточно для проведения теоретических исследований, но бывает недостаточно при практическом применении.

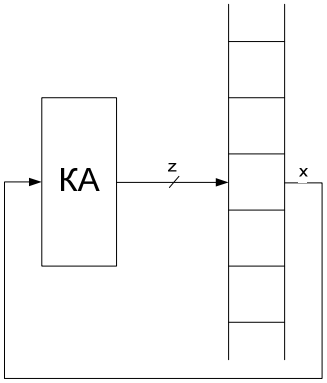


Рис. 3. Машина Тьюринга как автоматизированный объект управления

Переход от тьюрингова программирования к практическому (автоматному) программированию [20] осуществляется за счет усложнения объектов управления, которые могут выполнять сколь угодно сложные действия (операции), и применения в качестве системы управления системы взаимодействующих автоматов. Из изложенного следует, что универсальность предлагаемого подхода определяется тем, что он основан на расширении машины Тьюринга, которая позволяет реализовать произвольные алгоритмы. При этом теоретические положения о том, что автоматы позволяют распознавать регулярные языки [22], а магазинные автоматы – языки с контекстно-свободными грамматиками, отходят на второй план, так как *в рассматриваемом подходе используются не автоматы, а автоматизированные объекты управления*, в которых объекты управления и их сложность не фиксированы, как и число автоматов.

Учитывая изложенное, в работе [23] была сформулирована основная особенность автоматного программирования – *программы предлагается создавать так же, как производится автоматизация технологических (и не только) процессов*. При этом на основе анализа предметной области выделяются источники входных воздействий и автоматизированные объекты управления, каждый из которых содержит систему управления (систему взаимодействующих конечных автоматов) и объекты управления. Эти объекты реализуют выходные воздействия и формируют значения еще одной разновидности входных воздействий, которые по обратным связям передаются системе управления.

Все перечисленные составляющие каждого автоматизированного объекта управления отображаются на *схеме связей*, которая может совмещаться со *схемой взаимодействия* автоматов. На схеме связей для каждого входного и выходного воздействия указывается *полное название* и краткое символьное обозначение, которое в дальнейшем и используется в качестве пометки в графах переходов автоматов и идентификатора соответствующей переменной в программе. Использование кратких символьных обозначений позволяет даже весьма сложные алгоритмы отражать так компактно, что часто граф переходов удается размещать на одном экране монитора, позволяя человеку «охватить» весь граф одним взглядом, что резко упрощает понимание таких графов.

Использование символьных, а не смысловых идентификаторов, являющихся обычно сокращенными английскими словами, смысл которых обычно забывается через некоторое время, не ухудшает понятности автоматных программ, так как в рамках рас-

сматриваемого подхода их построение и внесение изменений в них должно производиться формально (вручную или автоматически) и только по графам переходов. При этом смысл переменных можно понять по схеме связей.

Объекты управления могут быть реальными или виртуальными – реализованными программно. В первом случае их логика изменена быть не может, а во втором – она (при необходимости) практически вся может быть вынесена в автоматы.

Парадигма автоматного программирования состоит в представлении и реализации программ как систем автоматизированных объектов управления [23].

Основные положения автоматного программирования

Основным понятием в автоматном программировании является *состояние* [11].

При написании автоматных программ предлагается (в отличие от работ [24, 25]) разделять состояния на два класса: *управляющие* и *вычислительные*. При этом с помощью небольшого числа управляющих состояний, как и в машине Тьюринга, можно управлять сколь угодно большим числом вычислительных состояний [26]. Во введенной классификации управляющие состояния могут быть названы *качественными*, а вычислительные – *количественными*. В рамках автоматного программирования основное внимание уделяется управляющим состояниям, которые, если это не оговаривается особо, и рассматриваются в дальнейшем. При этом справедливо соотношение: «Состояния + входные воздействия = конечный автомат без выхода». Справедливо также: «Автомат без выхода + выходные воздействия = автомат».

Автоматы могут быть абстрактными (входные и выходные воздействия формируются последовательно) и структурными (входные и выходные воздействия формируются «параллельно») [27]. В автоматном программировании, в отличие, например, от программирования компиляторов, обычно применяются структурные автоматы [28].

Время в автоматах в явном виде не используется. При необходимости применения элементов задержки они рассматриваются как объекты управления. При этом задержки запускаются и сбрасываются из автоматов, а информация об истечении времени поступает в них в виде входных воздействий.

Автоматы могут задаваться в различном виде, однако при их проектировании и использовании человеком они должны обладать *когнитивными свойствами* [29], что достигается при задании поведения автоматов в виде графов переходов (диаграмм состояний). Если автоматы генерируются автоматически [30], то они могут задаваться иначе, например, в табличной форме. При этом даже при сравнительно небольшом числе состояний и переходов такое задание затрудняет понимание работы автоматов человеком, так как отражение переходов в них ненаглядно. Понятность графов переходов достигается во многом за счет того, что *состояния декомпозируют множество всех входных воздействий* соответствующего автомата на группы, каждая из которых определяет переходы из рассматриваемого состояния.

Достоинства автоматного программирования

В рамках автоматного программирования предполагается, что собственно написание (генерация) программы начинается только после ее проектирования. При этом *в инженерной практике (в отличие от традиционного программирования) проект или его этап обязательно завершается выпуском проектной документации.* Поэтому при автоматном программировании, основной областью использования которого являются встроенные системы (чисто инженерная область), должна выпускаться *проектная* документация [31], а не только документация пользователя, как это обычно принято в

программных проектах. При этом для автоматных программ необходимой компонентой, входящей в состав проектной документации, должны быть графы переходов.

Проектирование автоматов, описывающих логику программ, которая при традиционном программировании не упорядочена и поэтому сложна, а также *формальный и изоморфный* переход от автоматов к реализующим их программам приводят к тому, что программы *либо сразу работают, либо требуют минимальной отладки*. Это также связано с тем, что реализация функций входных и выходных воздействий при излагаемом подходе, как отмечалось выше, почти не содержит логики.

При необходимости проведения отладки для автоматных программ могут генерироваться отладочные протоколы, которые отражают поведение программ в терминах автоматов (состояний, переходов, значений входных и выходных воздействий), так как в автоматном программировании автоматы являются не картинками [32], а частью программ, представленных в нетрадиционной для программистов визуальной, а не текстовой форме. При этом отметим, что увеличение времени создания программ при использовании автоматного подхода компенсируется сокращением времени их отладки. Это приводит к тому, что для программ средней сложности трудоемкости разработки на основе автоматного и традиционного подходов практически совпадают. Однако в первом случае остаются диаграммы, понятные человеку, по которым программа была построена формально, а во втором – только программа, понимание логики которой для представителей заказчика или даже для ее автора через некоторое время часто представляет большую проблему.

Создание программ со сложным поведением без использования диаграмм приводит к трудностям на всех этапах жизненного цикла. Особенно сложно в этом случае реализовывать программы, так как все особенности их поведения приходится держать в голове в течение всего времени их написания, вместо того чтобы отобразить их на диаграмме и на время забыть. Еще одним преимуществом автоматных программ является простота внесения изменений в них специалистами в предметной области, не являющимися профессиональными программистами.

Следующее преимущество автоматного подхода – эффективность верификации автоматных программ на основе метода *Model Checking* (верификация на модели) [33]. Это объясняется тем, что модель для верификации программ этого класса может строиться автоматически по графу переходов и иметь относительно небольшой размер, так как в графах переходов используются только управляющие состояния.

Для автоматных программ *отсутствует семантический разрыв* между требованиями к программе и к модели, так как он устраняется в ходе разработки графов переходов на этапе проектирования. Это позволяет считать автоматные программы приспособленными к верификации. Таким образом, при верификации программ имеет место ситуация, аналогичная контролю схем со сложной логикой – эти схемы не удается проверить, если они не спроектированы специальным образом для обеспечения контролепригодности.

В заключение раздела отметим, что для автоматных программ естественен параллелизм, что особенно важно при применении многоядерных процессоров [34].

Разновидности автоматного программирования

Автоматное программирование развивается в трех основных направлениях: логическое управление, программирование с явным выделением состояний и объектно-ориентированное программирование с явным выделением состояний.

Логическое управление

Наиболее просто и естественно применять автоматы в системах логического управления, в которых входные и выходные переменные двоичны. Естественность

применения автоматов при программировании этого класса систем объясняется тем, что программы в них заменяют логические схемы, проектирование которых с использованием автоматов широко распространено и развивается давно, начиная с релейно-контактных схем [28]. Однако простота и естественность применения автоматов даже для этого класса систем, видимо, далеко не очевидна, так как даже при программировании логических контроллеров [35] практически никто не предлагал сначала проектировать автоматы, а затем их реализовать на выбранном языке программирования.

В работе [36] было показано, что плохого в неавтоматном программировании контроллеров, а работах [37–40] описано применение технологии автоматного программирования для систем логического управления. Эта технология была использована при создании ряда систем управления, в том числе судовыми техническими средствами [11]. В работах [41–43] в качестве примеров показано, как применять предложенную технологию для языка функциональных блоков, широко используемого в программируемых логических контроллерах, а также в инструментальном средстве *LabVIEW*.

Программирование с явным выделением состояний

Значительно более широким классом по сравнению с системами логического управления являются реактивные системы. Для описания поведения таких систем, к которым относится большинство встроенных систем, применение автоматов также естественно [17], как и для систем логического управления. Однако далеко не все программисты и для таких систем используют автоматы, что приводит к множеству проблем, о которых говорилось выше.

Реактивные системы являются более сложными по сравнению с системами логического управления. В них:

- в качестве входных воздействий, наряду с входными переменными, используются события;
- запуск программ осуществляется по событиям, а не циклически;
- в качестве выходных воздействий могут использоваться не только двоичные, но и другие функции, что позволяет называть автоматы, применяемые при этом, *гибридными* [44];
- автоматы могут содержать не только вложенные состояния [17], но и вложенные автоматы;
- автоматы могут взаимодействовать не только за счет проверки номеров состояний, как было предложено для систем логического управления [11], но также и за счет вложенности, вызываемости и обмена событиями (сообщениями).

Рассматриваемый класс систем обычно реализуется на процедурных языках программирования. Поэтому традиционно используемый процесс написания программ в этом случае может быть назван *процедурным программированием* или просто *программированием*. В таких программах состояния существуют, но они обычно явно не выделяются. Это отличает их от автоматных программ, создание которых в этом случае может быть названо *программированием с явным выделением состояний*.

Технология программирования с явным выделением состояний создавалось в ходе выполнения работ по разработке системы управления судовыми дизель-генераторами [45]. Эта технология подробно описана в работах [46–51].

Объектно-ориентированное программирование с явным выделением состояний

Уже два десятилетия объектно-ориентированное программирование (ООП) является наиболее широко используемым стилем программирования в мире. При применении объектной парадигмы программы строят из объектов, взаимодействующих за счет обмена сообщениями. С развитием методов проектирования таких программ [52] в вопрос описания и реализации поведения объектов ясность не была внесена, а применять

автоматы предлагалось лишь от случая к случаю, а не «как руководство» к действию, пригодное для использования во многих системах при описании сложного поведения. Даже в тех редких случаях, когда автоматы применялись для описания поведения программ (например, при проектировании программного обеспечения метеостанции в работе [24]), это делалось неубедительно и не способствовало их широкому применению в ООП.

Появление унифицированного языка моделирования *UML* и даже языка *UML 2.0* [54] эту проблему не решило. Во-первых, в этом языке, кроме диаграмм состояний, для описания поведения предлагается использовать и другие типы диаграмм и не говорится, когда и какие диаграммы следует применять. Во-вторых, в рамках унифицированного процесса разработки программ [55], как, впрочем, и при использовании других методологий их создания (например, описанной в работе [56]), не было предложено подходов для совместного использования диаграмм, описывающих статические и динамические свойства программ. В-третьих, диаграммы для описания поведения в основном использовались как язык общения между участниками разработки и для документирования программ, в то время как для кодогенерации использовались только диаграммы классов. Лишь в последние годы в рамках концепции *исполняемого UML* [57] вопрос о кодогенерации был поставлен и для остальных диаграмм.

Для решения указанной проблемы под руководством автора были выполнены исследования по совместному использованию объектной и автоматной парадигм программирования. При этом такой стиль программирования был назван *объектно-ориентированное программирование с явным выделением состояний* [58].

Возможны различные подходы к решению этой проблемы. Автоматы можно использовать, например, как методы классов [59] или как классы [60]. Более «глубокое проникновение» автоматов в объектно-ориентированное программирование происходит при применении паттернов проектирования. При этом отметим, что применение паттерна *State*, предназначенного для реализации объектов, поведение которых зависит от состояния, как ни странно, обычно вызывает трудности [61]. Решению этого вопроса посвящена работа [62]. Для устранения недостатков, присущих указанному паттерну, в работах [63, 64] был предложен паттерн *State Machine,* на базе которого создан язык с тем же названием [65], являющийся расширением языка *Java* и позволяющий достаточно эффективно реализовывать автоматы.

В работе [66] был предложен еще один подход к реализации объектно-ориентированных программ с явным выделением состояний, который позволяет обеспечить повторное использование программных компонентов, параллельные вычисления и автоматическое протоколирование работы системы. В качестве основы для разработки «автоматной» части программ в этой работе была предложена библиотека, реализованная на языке *C++*. При использовании этой библиотеки остальная часть программ (контекст) разрабатывается традиционным образом.

Особенности проектирования и свойства автоматных программ позволяют отнести автоматное программирование к одной из разновидностей синхронного программирования [67, 68], которое активно развивается в Западной Европе для создания программного обеспечения ответственных систем.

Высокое качество автоматных программ может быть достигнуто не только за счет автоматного расширения универсальных языков программирования, но в тех случаях, когда для написания автоматных программ разрабатываются языки, учитывающие их специфику [69, 70].

Существуют и другие подходы к совместному использованию объектной и автоматной парадигм программирования. Классификация таких подходов приведена в работах [71, 72].

Работы в указанном направлении продолжаются. При этом, например, в работе [73] предложена удобная графическая нотация для отображения наследования в автоматных программах, а в работе [74] на основе использования понятия автоматизированный объект управления предложен новый подход к проектированию сложных объектно-ориентированных программ с явным выделением состояний.

Верификация автоматных программ

Выше отмечалось, что автоматные программы, в отличие от программ, написанных традиционно, сравнительно легко верифицируются на основе метода *Model Checking*, за разработку которого Э. Кларку, Э. Эмерсону и Д. Сифакису была присуждена *A.M. Turing Award 2007*. Упрощение формальной верификации для рассматриваемого класса программ по сравнению с построенными традиционным образом очень важно для построения ответственных систем (например, самолетов, вертолетов, ядерных реакторов и т. д.). Поэтому автор надеется, что со временем в технических заданиях на разработку программного обеспечения таких систем будет записываться требование использовать автоматное программирование.

В настоящее время активно ведутся работы в указанном направлении [33, 75–78]. В частности, проводятся исследования по государственному контракту «Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода», выполняемому в рамках Федеральной целевой программы «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007–2012 годы».

Автоматизация построения автоматных программ

Из изложенного следует, что основная трудоемкость построения автоматных программ связана с проектированием автоматов. Однако существуют задачи, про которые известно, что они могут быть решены с использованием автоматов, но эвристически построить автоматы в этих задачах крайне трудно или даже невозможно. В этих случаях можно пытаться применять формализованные методы. Например, в работе [79] предложено использовать *динамическое программирование для построения автоматов*. Однако такой подход имеет ограниченную область применения.

Значительно более универсально применение генетического программирования для генерации автоматов. В настоящее время активно ведутся работы по этой тематике [80–87], в частности, проводятся исследования по государственному контракту «Технология генетического программирования для генерации автоматов управления системами со сложным поведением», выполняемому в рамках указанной выше Федеральной целевой программы.

Генерация автоматов позволяет до 80% кода автоматных программ строить почти автоматически, так как в программах этого класса объем кода, порождаемого автоматами, может достигать указанной величины.

В указанных работах на основе генетического программирования выращивались автоматы, а в работе [88] одновременно использовались различные модели и методы искусственного интеллекта. Во-первых, проектировалась мультиагентная система, во-вторых, система управления каждым объектом состояла из нейронной сети и автомата, в-третьих, для того, чтобы не потерять информацию о взаимодействии объектов, в качестве особи выбиралось две указанные системы управления, и, наконец, в-четвертых, к такой сложной особи применялось генетическое программирование. В результате была построена система управления, в которой автомат имеет шесть состояний и 48 пере-

ходов. Эксперименты показали, что автоматически построенная система управления обеспечивает более эффективную работу мультиагентной системы, по сравнению со случаем, когда автоматы системы управления (их было семь) строились эвристически [89]. При этом отметим, что поведение системы, построенной человеком, однако, значительно более понятно, и в нее существенно проще вносить изменения.

Технология автоматного программирования

На основании указанных выше работ была разработана технология автоматного программирования, которая охватывает все этапы жизненного цикла программ рассматриваемого класса. Эта технология описана в работах [90–92], а в работе [93] она изложена для массового читателя.

Инструментальные средства для поддержки автоматного программирования

Рассмотрим средства для поддержки автоматного программирования.

Для *процедурных автоматных программ* в работе [94] было разработано инструментальное средство, которое по графам переходов, представленным в нотации, предложенной в работе [48], и изображенным с помощью пакета *Visio*, генерирует код, изоморфный реализуемым графам переходов, который основан на конструкциях `switch` языка *C*. Это средство применяется в настоящее время при создании программного обеспечения одного класса ответственных систем реального времени. При этом в качестве программ используются реализованные вручную на языке *C* функции входных и выходных воздействий, которые практически не содержат логики, а также графы переходов, по которым исходный код генерируется автоматически. По отзывам разработчиков этих систем, их уже долгое время не покидает удивление оттого, что в каждой новой системе программы, спроектированные указанным образом, работают практически без отладки, а расширение функциональности обычно обеспечивается «малой кровью», и все это достигается при использовании процедурного, а не объектного программирования.

Это направление исследований получило развитие в работе [95], в которой показано, что аналогичный подход может быть использован для реализации автоматов на любом наперед заданном языке программирования. Для поддержки такого подхода было создано инструментальное средство *MetoAuto*.

Переходя к инструментальному средству для поддержки построения *объектно-ориентированных автоматных программ*, отметим, что если для генерации программ по автоматам, кроме средств, рассмотренных выше, известны также и многие другие [96], то решение задачи об автоматизации построения объектно-ориентированных программ в целом в открытых источниках не излагалось. Решение этой задачи было предложено в ходе выполнения работ по теме «Автоматное программирование: применение и инструментальные средства», которая выполнялась в рамках Федеральной целевой научно-технической программы «Исследования и разработки по приоритетным направлениям науки и техники» на 2002–2006 гг. [97].

В результате было создано инструментальное средство *UniMod* [98–102], которое автоматизирует процесс построения объектно-ориентированных автоматных программ. При его использовании структура программ задается диаграммами классов, которые изображаются не традиционным путем, а в форме схемы связей автоматов с поставщиками событий и объектами управления. Динамика программ описывается с помощью диаграмм состояний в нотации языка *UML*, в которых могут использоваться не только вложенные состояния, но и вложенные автоматы. При этом имеется возможность проверить ряд свойств этих диаграмм, например, полноту и непротиворечивость. Функции

входных и выходных воздействий, которые практически не содержат логики, пишутся на языке *Java* вручную. После этого автоматически может быть скомпилирован код программы в целом или программа может выполняться в режиме интерпретации. Описанное инструментальное средство находится в свободном доступе и является единственным открытым и бесплатным средством http://unimod.sourceforge.net/intro.html, поддерживающим концепцию *исполняемого UML* [103].

Апробация технологии автоматного программирования

Эта технология применялась не только при разработке программного обеспечения систем управления, но и для ряда других задач, например, головоломок [104], игр [105, 106], информационных систем [107] и учебных программ [108].

Автоматы применялись также и в ходе выполнения исследований в области искусственного интеллекта. Так, например, в работах [109, 110] рассматривалось применение автоматов при построении мультиагентных систем, а в работе [111] – вопрос о совместном использовании автоматов и нейронных сетей.

Кроме этих работ, на сайте [112], посвященном автоматному программированию, постоянно публикуются проекты, для каждого из которых созданы программное обеспечение на основе автоматного подхода и проектная документация (раздел «Проекты»). На сайте также имеется раздел, посвященный *UniMod*-проектам.

Разработка визуализаторов алгоритмов дискретной математики

Технология автоматного программирования продемонстрировала свою эффективность при решении различных задач, но при реализации алгоритмов дискретной математики автоматы используются крайне редко. Известны лишь несколько задач этого класса, в которых применять автоматы целесообразно. Это, например, поиск подстрок [113], подсчет длины слов в строке [114] и обход деревьев [115]. Однако оказалось, что если реализовывать алгоритмы дискретной математики на автоматах часто нецелесообразно, то строить их визуализаторы с применением автоматов крайне полезно всегда, так как при этом визуализацию можно проводить в состояниях. При решении задачи построения таких визуализаторов был выполнен ряд работ как теоретического [116–120], так и прикладного характера [121–124]. В результате было разработано инструментальное средство *VIZI* для автоматизированного построения визуализаторов рассматриваемого класса. Визуализаторы, построенные на основе автоматного подхода, и проектная документация к ним опубликованы на сайте [112] в разделе «Визуализаторы».

Несмотря на то, что лекции по курсу «Алгоритмы дискретной математики» читаются практически в каждом университете мира, где обучают информационным технологиям, а в некоторых из них в учебном процессе используются визуализаторы, эффективную методику их построения без применения автоматного подхода создать не удавалось.

Отличие предлагаемого подхода от известных

Автоматы все чаще применяются в программировании. Если раньше они обычно использовались при построении компиляторов и протоколов, то теперь их от случая к случаю применяют и в других областях [18, 19] При этом был разработан ряд инструментальных средств, которые поддерживают программирование с автоматами [96]. Одним из наиболее распространенных инструментальных средств в этой области является *Stateflow* (http://www.mathworks.com/products/stateflow/) – расширение пакета *MatLab*. Это средство позволяет строить автоматы, моделировать работу их и выполнять кодо-

генерацию на языке *СИ*. В 2007 г. корпорация *Microsoft* разработала программный продукт *Windows Workflow Foundation* (http://itc.ua/node/23217), в котором конечные автоматы (State Machines) в форме диаграмм состояний используются в качестве языка программирования.

Отличие предлагаемого подхода от известных состоит в том, что *автором предлагается применять автоматы в программировании не от случая к случаю, а везде и всегда, где требуется обеспечить сложное поведение* (сложным считается поведение, при котором выбор выходного воздействия зависит не только от входного воздействия, но и от предыстории). Это позволяет уменьшить число «прозрений», аналогичных произошедшим в 2007 г., когда ведущий специалист корпорации *IBM* (!) вдруг обнаружил, что автоматы целесообразно использовать и при разработке виджетов [18, 19]. Автоматный подход поддержан парадигмой, технологией и инструментальными средствами. Его применение практически не зависит от используемых программных [70] и аппаратных средств [42, 43], в то время как в известных работах либо решается та или иная задача с применением автоматов, либо описывается конкретное инструментальное средство для поддержки программирования с автоматами.

Если при использовании традиционного подхода для реализации последовательностного поведения программы обычно применяется множество двоичных переменных, называемых флагами, то при автоматном программировании в процесс реализации вводится этап, который известен из теории автоматов и называется «кодирование состояний». Термин «кодирование» при традиционном подходе заменял термин «программирование», так как при его использовании состояния явно обычно не выделялись. Явное выделение состояний позволяет *кодировать (различать) состояния автомата с любым числом состояний с помощью одной многозначной переменной, что позволило ввести в программирование понятие «наблюдаемость»* [11], которое широко применяется в теории автоматического управления. Это дает возможность наблюдать за поведением каждого автомата в пространстве его состояний по изменению значения только *одной переменной.* Если же для автоматных программ ввести ограничение, состоящее в том, что в каждом программном цикле или при каждом запуске программы в каждом автомате выполняется не более одного перехода, то все используемые значения переменной, кодирующей состояния автомата, становятся доступными для других автоматов системы и могут использоваться во взаимных блокировках, не требуя введения для этой цели дополнительных переменных.

Проектирование автоматов и многозначное кодирование их состояний позволяют *формально и изоморфно* реализовывать автоматы с помощью конструкции `switch` языка *С* или ее аналогов в других языках программирования. При этом в теле этих конструкций нецелесообразно реализовывать функции входных и выходных переменных, а достаточно указывать только сами эти переменные, которые осуществляют вызов соответствующих функций, которые практически не содержат логики и реализуются отдельно. Тем самым осуществляется отделение логики поведения (условий, определяющих необходимость выполнения тех или иных действий) от описания его семантики (смысла каждого из действий), что резко упрощает понимание программ.

Изложенный подход позволяет практически исключить отладку построенных таким образом программ. Использование предлагаемого подхода не всегда обеспечивает уменьшение времени создания программы по сравнению с традиционным подходом. Однако программы, построенные с использованием предлагаемого подхода, обладают многими достоинствами, которые были описаны выше. Есть основание предполагать, что для ответственных объектов, автоматизация которых требуется верификации программ, применение автоматного программирования, как отмечалось выше, *может стать неизбежным*. Исследования в этом направлении активно проводятся [125, 126].

Заключение

В ходе исследований по автоматному программированию необходимо было решить задачи в области проведения научных исследований, разработки технологии для его поддержки и образования. Все эти задачи были решены в результате педагогического эксперимента, описанного в работе [127]. Со многими работами, указанными в списке литературы, можно ознакомиться на сайте http://is.ifmo.ru/. Там же приведены и некоторые результаты внедрения автоматного программирования в практику проектирования различных систем.

Идеи автоматного программирования, изложенные в статье, могут в том или ином виде использоваться не только для текстовых и визуальных языков программирования, как описано выше, но и для программируемых логических контроллеров [128, 129], а также различных средств автоматизации [43] и имитационного моделирования [44, 130, 131]. Автор предполагает, что области применения автоматного программирования будут еще расширяться, так как «более 99% всех микропроцессоров, проданных в 1998 г., использовались во встраиваемых системах, а в 2000 г. число микроконтроллеров в высококачественном автомобиле достигало 60» [132].

Рассматриваемая технология важна и для образования. В частности, на ее основе можно проводить *первоначальное обучение проектированию программ* не только в университетах [133], но и в средних школах [134]. При этом отметим, что, поскольку концепции автоматного программирования существенно отличаются от традиционных, начинать обучение программированию в этом стиле следует как можно раньше [135].

Автор выражает надежду, что технология использования автоматов при разработке программных систем со сложным поведением будет развиваться: появятся новые модели, нотации, инструментальные средства. Например, при участии авторов ведутся работы по созданию текстовых языков автоматного программирования [136–138], декларативных методов описания автоматов на императивных языках программирования [139], методов динамической верификации автоматных программ [140], инструментальных средств на основе концепции предметно-ориентированных языков программирования [141]. Также проводятся работы по применению автоматов при создании программного обеспечения для мобильных роботов [142] и автоматизации документооборота [143].

Автор надеется, что парадигма автоматного программирования, изложенная в этой статье, станет «каркасом» для дальнейших исследований в области использования автоматов в программировании.

В заключение работы отметим, что создание автоматных программ предполагает их проектирование, названное во введении к книге [11] «автоматным проектированием программ». Так как при проектировании этого класса программ основное внимание уделяется управлению, то можно говорить о новой парадигме управления, названной автором в работе [11] «автоматное управление». Эта парадигма, как отмечалось выше, неоднократно апробировалась на практике, в том числе и при проектировании программного обеспечения сложных систем [45, 128, 129].

Отметим также, что использование автоматов при проектировании систем управления [146] до последнего времени в основном рассматривалось в рамках применения гибридных автоматов [147]. Дополнительный интерес к использованию автоматов в управлении возник у специалистов после пленарного доклада Р. Брокетта на конгрессе IFAC [148], в котором обсуждались вопросы упрощения проектирования сложных систем управления.

Автор признателен *Дмитрию Александровичу Поспелову*, в ходе беседы с которым в 1998 г. на конференции по мультиагентным системам, проходившей в поселке Ольгино под Санкт-Петербургом, родился термин (см. введение к работе [11]), который

использован в названии настоящей статьи. На английский язык этот термин переводится как «Automata-Based Programming». Он был предложен в работе [128], а парадигма автоматного программирования – в работе [129].

Литература

1. Дейкстра Э. Смиренный программист – Лекции лауреатов премии Тьюринга за первые двадцать лет. 1966–1985. М.: Мир, 1993.
2. Кнут Д. Искусство программирования. Том. 1. Основные алгоритмы. – М.: Вильямс, 2000.
3. Software Engineering. Germany: NATO Science Committee, 1968. – Режим доступа: http://www.europrog.ru/book/nato1968e.pdf
4. Software Engineering Techniques. Italy: NATO Science Committee, 1969. – Режим доступа: http://www.europrog.ru/book/nato1969e.pdf
5. Брукс Ф. Мифический человеко-месяц или как создаются программные системы. – СПб: Символ, 2000.
6. Software Engineering 2006. ETH Zurich. Chair of Software Engineering.
7. Черняк Л. Адаптируемость и адаптивность // Открытые системы. – 2004. – № 9.
8. Cai Kai-Yuan, Chen T. Y., Tse T. H. Towards Research on Software Cybernetics / Proceedings of 7th IEEE International on High-assurance Systems Engineering (HASE 2002). Los Alamitos. IEEE Computer Society Press, 2002.
9. Шалыто А. А. Программная реализация управляющих автоматов // Судостроительная промышленность. – Серия «Автоматика и телемеханика». – 1991. – Вып. 13.
10. Яковлев В.Б. Автоматика, кибернетика, информатика, синергетика // Труды конференции «Пятьдесят лет развития кибернетики». СПбГТУ, 1999.
11. Шалыто А.А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. – СПб: Наука, 1998.
12. Harel D. Biting the Silver Bullet: Toward a Brighter Future for System Development // Computer. – 1992. – № 1.
13. Непейвода Н.Н. Стили и методы программирования. – М.: Интернет-университет информационных технологий, 2005.
14. Ахо А., Сети Р., Ульман Д. Компиляторы. Принципы, технологии, инструменты. – М.: Вильямс, 2001.
15. Хопкрофт Д., Мотвани Р., Ульман Д. Введение в теорию автоматов, языков и вычислений. – М.: Вильямс, 2002.
16. Непейвода Н.Н., Скопин И.Н. Основания программирования. – М.–Ижевск: Институт компьютерных исследований, 2003.
17. Harel D. et al. Statemate: A Working Environment for the Development of Complex Reactive Systems // IEEE Trans.Software Eng. – 1990. – № 4.
18. Принг Э. Конечные автоматы в JavaScript, Часть 1: Разработаем виджет. – Режим доступа: http://www.ibm.com/developerworks/ru/library/wa-finitemach1/index.html
19. Принг Э. Конечные автоматы в JavaScript, Часть 2: Реализация виджета. – Режим доступа: http://www.ibm.com/developerworks/ru/library/wa-finitemach2/wa-finitemac_ru.html
20. Шалыто А.А., Туккель Н.И. От тьюрингова программирования к автоматному // Мир ПК. – 2002. – № 2.
21. Хопкрофт Д. Машины Тьюринга // В мире науки. – 1984. – № 7.
22. Карпов Ю.Г. Теория автоматов. – СПб: Питер, 2002.

23. Шалыто А.А. Автоматное программирование // Тезисы докладов Международной научной конференции памяти профессора А.М. Богомолова «Компьютерные науки и информационные технологии». – Саратов: СГУ, 2007.
24. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. – М.: Бином, СПб.: Невский диалект, 1998.
25. Лавров С.Н. Программирование. Математические основы, средства, теория. – СПб: БХВ-Петербург, 2001.
26. Туккель Н.И., Шамгунов Н.Н., Шалыто А.А. Ханойские башни и автоматы // Программист. – 2002. – № 8.
27. Глушков В.М. Синтез цифровых автоматов. – М: Физматгиз, 1962.
28. Гаврилов М.А., Девятков В.В., Пупырев Е.И. Логическое проектирование дискретных автоматов. – М.: Наука, 1977.
29. Shalyto A.A. Cognitive Properties of Hierarchical Representations of Complex Logical Structures / Proceeding of the 1995 International Symposium on Intelligent Control (ISIC). Workshop. 1995. Monterey. California.
30. Фогель Л., Оуэнс А., Уолш М. Искусственный интеллект и эволюционное моделирование. – М.: Мир, 1969.
31. Шалыто А.А. Новая инициатива в программировании. Движение за открытую проектную документацию // Информационно-управляющие системы. – 2003. – № 4.
32. Зюбин В.Е. Программирование информационно-управляющих систем на основе конечных автоматов. – Новосибирск: НГУ, 2006.
33. Кузьмин Е.В., Соколов В.А. Моделирование, спецификация и верификация «автоматных» программ // Программирование. – 2008. – № 1. – С. 38–60.
34. Любченко В., Тяжлов Ю. Осторожно: многоядерный процессор // Открытые системы. – 2007. – № 6.
35. International Standard IEC 1131-3. Programmable controllers. Part 3. Programming languages. International Electrotechnical Commission. 1993.
36. Вавилов В.К., Шалыто А.А. Что плохого в неавтоматном подходе к программированию контроллеров? // Промышленные АСУ и контроллеры. – 2007. – № 1.
37. Шалыто А.А. Технология программной реализации алгоритмов логического управления как средство повышения живучести // Тезисы докладов научно-технической конференции «Проблемы обеспечения живучести кораблей и судов». – СПб: Судостроение, 1992.
38. Антипов В.В., Шалыто А.А. Алгоритмизация и программирование задач логического управления техническими средствами. – СПб: Моринтех, 1996.
39. Шалыто А.А. SWITCH-технология. Алгоритмизация и программирование задач логического управления // Промышленные АСУ и контроллеры. – 1999. – № 9.
40. Шалыто А.А. Автоматное проектирование программ. Алгоритмизация и программирование задач логического управления // Известия РАН. Теория и системы управления. – 2000. – № 6.
41. Шалыто А.А. Реализация алгоритмов логического управления программами на языке функциональных блоков // Промышленные АСУ и контроллеры. – 2000. – № 4.
42. Альтерман И.З., Шалыто А.А. Формальные методы программирования логических контроллеров // Промышленные АСУ и контроллеры. – 2005. – № 10.
43. Вавилов В.К., Шалыто А.А. LabVIEW и SWITCH-технология // Промышленные АСУ и контроллеры. – 2006. – № 6.
44. Колесов Ю.Б., Сениченков Ю.Б. Моделирование систем. Динамические и гибридные системы. – СПб.: БХВ–Петербург, 2006.

45. Туккель Н.И., Шалыто А.А. Проектирование программного обеспечения системы управления дизель-генераторами на основе автоматного подхода // Системы управления и обработки информации. – 2003. – Вып. 5.
46. Туккель Н.И., Шалыто А.А. Применение SWITCH-технологии для программирования в событийных системах // Труды международной научно-технической конференции «Пятьдесят лет развития кибернетики». – СПб: СПбГТУ, 1999.
47. Туккель Н.И., Шалыто А.А. SWITCH-технология – автоматный подход к созданию программного обеспечения «реактивных» систем // Промышленные АСУ и контроллеры. – 2000. – № 10.
48. Туккель Н.И., Шалыто А.А. SWITCH-технология – автоматный подход к созданию программного обеспечения «реактивных» систем // Программирование. – 2001. – № 5.
49. Туккель Н.И., Шалыто А.А. Программирование с явным выделением состояний // Мир ПК. – 2001. – № 8, № 9.
50. Туккель Н.И., Шалыто А.А. Реализация автоматов при программировании событийных систем // Программист. – 2002. – № 4.
51. Шалыто А.А. Алгоритмизация и программирование для систем логического управления и «реактивных» систем (обзор) // Автоматика и телемеханика. – 2001. – № 1.
52. Грэхем И. Объектно-ориентированные методы. Принципы и практика. – М.: Вильямс, 2004.
53. Рамбо Дж., Якобсон А., Буч Г. UML. Специальный справочник. – СПб: Питер, 2001.
54. Рамбо Дж., Блаха М. UML 2.0. Объектно-ориентированное моделирование и разработка. – СПб: Питер, 2007.
55. Рамбо Дж., Якобсон А., Буч Г. Унифицированные процесс разработки программного обеспечения. – СПб: Питер, 2002.
56. Ауер К., Миллер Р. Экстремальное программирование: постановка процесса. – СПб: Питер, 2003.
57. Mellor S. et al. Executabl UML: A Foundation for Model Driven Architecture. – MA: Addison-Wesley, 2002.
58. Туккель Н.И., Шалыто А.А. Объектно-ориентированное программирование с явным выделением состояний // Материалы Международной научно-технической конференции «Искусственный интеллект». – Т.1. – Таганрог. ТГРУ; Донецк. Донецкий гос. ин-т искусств. интеллекта, 2002.
59. Туккель Н.И., Шалыто А.А. Танки и автоматы // BYTE/Россия. – 2003. – № 2.
60. Наумов Л.А., Шалыто А.А. Искусство программирования лифта. Объектно-ориентированное программирование с явным выделением состояний // Информационно-управляющие системы. – 2003. – № 6.
61. Гранд М. Шаблоны проектирования в Java. – М.: Новое знание. 2004.
62. Шопырин Д.Г., Шалыто А.А. Применение класса STATE в объектно-ориентированном программировании с явным выделением состояний // Труды X Всероссийской научно-методической конференции «Телематика-2003». – СПб: СПбГИТМО (ТУ). 2003. – Т.1.
63. Корнеев Г.А., Шамгунов Н.Н., Шалыто А.А. Паттерн State Machine для объектно-ориентированного проектирования автоматов // Информационно-управляющие системы. – 2004. – № 5.
64. Shamgunov N., Korneev G., Shalyto A. State Machine Design Pattern // .NET Technologies 2006 – Shot communication papers conference proceedings. – 4-th International Conference in Central Europe on .Net Technologies. University of West Bohemia. 2006.

65. Корнеев Г.А., Шамгунов Н.Н., Шалыто А.А. Язык State Machine – расширение языка Java для эффективной реализации автоматов // Информационно-управляющие системы. – 2005. – № 1.
66. Шопырин Д.Г., Шалыто А.А. Объектно-ориентированный подход к автоматному программированию // Информационно-управляющие системы. – 2003. – № 5.
67. Туккель Н.И., Шалыто А.А. Автоматное и синхронное программирование // Искусственный интеллект. – 2003. – № 4.
68. Шопырин Д. Г., Шалыто А. А. Синхронное программирование // Информационно-управляющие системы. – 2004. – № 3.
69. Гуров В.С., Мазин М.А., Шалыто А.А. Текстовый язык для автоматного программирования // XIV Всероссийская научно-методическая конференция «Телематика-2007». – СПб.: СПбГУ ИТМО. 2007. – Т.2.
70. Степанов О.Г., Шопырин Д.Г., Шалыто А.А. Предметно-ориентированный язык автоматного программирования на базе динамического языку RUBY // Информационно-управляющие системы. – 2007. – № 4.
71. Наумов Л.А., Шалыто А.А. Методы объектно-ориентированной реализации реактивных агентов на основе конечных автоматов // Искусственный интеллект. – 2004. – № 4.
72. Naumov L., Korneev G., Shalyto A. Methods of Object-Oriented Reactive Agents Implementation on the Basis of Finite Automata // 2005 International Conference on «Integration of Knowledge Intensive Multi-Agent Systems: Modeling, Exploration and Engineering» (KIMAS-05). – Boston: IEEE. 2005.
73. Шопырин Д.Г., Шалыто А.А. Графическая нотация наследования автоматных классов // Программирование. – 2007. – № 4.
74. Поликарпова Н.И., Шалыто А.А. Автоматное программирование. – СПб: СПбГУ ИТМО, 2007. – Режим доступа: http://is.ifmo.ru/books/_umk.pdf
75. Вельдер С.Э., Шалыто А.А. О верификации простых автоматных программ на основе метода Model Checking // Информационно-управляющие системы. – 2006. – № 3.
76. Корнеев Г.А., Парфенов В.Г., Шалыто А.А. Верификация автоматных программ // Тезисы докладов Международной научной конференции, посвященной памяти профессора А.М. Богомолова «Компьютерные науки и технологии». – Саратов: СГУ, 2007.
77. Корнеев Г.А., Шалыто А.А. Верификация управляющих программ со сложным поведением, построенных на основе автоматного подхода // Материалы международной научно-технической мультиконференции «Проблемы информационно-компьютерных технологий и мехатроники» (ИКТМ–2007). – Таганрог: НИИ МВС ЮФУ, 2007.
78. Гуров В.С., Шалыто А.А., Яминов Б.Р. Технология верификации автоматных моделей программ без их трансляции во входной язык верификатора // Материалы международной научно-технической мультиконференции «Проблемы информационно-компьютерных технологий и мехатроники» (ИКТМ–2007). – Таганрог: НИИ МВС ЮФУ, 2007.
79. Оршанский С.А., Шалыто А.А. Применение динамического программирования при решении задач на конечных автоматах // Компьютерные инструменты в образовании. – 2007. – № 4.
80. Лобанов П.Г., Шалыто А.А. Использование генетических алгоритмов для автоматического построения конечного автомата в задаче о флибах / 1-я Российская мультиконференция по проблемам управления. Сборник докладов четвертой науч-

ной конференции «Управление и информационные технологии». – СПбГУ ЭТУ «ЛЭТИ», 2006.
81. Лобанов П.Г., Шалыто А.А. Использование генетических алгоритмов для автоматического построения конечных автоматов в задаче о флибах // Известия РАН. Теория и системы управления. – 2007. – № 5.
82. Мандриков Е.А., Кулев В.А., Шалыто А.А. Построения автоматов с помощью генетических алгоритмов для решения задачи о флибах // Сборник X международной конференции по мягким вычислениям и измерениям. – СПбГУ ЭТУ «ЛЭТИ». – 2007. – Т.1.
83. Мандриков Е.А., Кулев В.А., Шалыто А.А. Применение генетического программирования при решении задачи о флибах // Информационные технологии. – 2007. – № 12.
84. Царев Ф.Н., Шалыто А.А. Применение генетического программирования для генерации автоматов в задаче об «умном муравье» // Сборник научных трудов. IV-я Международная научно-практическая конференция «Интегрированные модели и мягкие вычисления в искусственном интеллекте. – Коломна: – 2007.
85. Царев Ф.Н., Шалыто А.А. О построении автоматов с минимальным числом состояний для задачи об «умном муравье» // Сборник докладов X международной конференции по мягким вычислениям и измерениям. – СПбГУ ЭТУ «ЛЭТИ». – 2007. – Т.2.
86. Поликарпова Н.И., Точилин В.Н., Шалыто А.А. Применение генетического программирования для реализации систем со сложным поведением // Сборник научных трудов. IV-я Международная научно-практическая конференция «Интегрированные модели и мягкие вычисления в искусственном интеллекте. – Коломна. – 2007.
87. Поликарпова Н.И., Точилин В.Н., Шалыто А.А. Разработка библиотеки для генерации автоматов методом генетического программирования // Сборник докладов X международной конференции по мягким вычислениям и измерениям. – СПбГУ ЭТУ «ЛЭТИ». – 2007. – Т.2.
88. Царев Ф.Н., Шалыто А.А. Применение генетического программирования для построения мультиагентной системы одного класса // Материалы международной научно-технической мультиконференции «Проблемы информационно-компьютерных технологий и мехатроники» (ИКТМ–2007). – Таганрог: НИИ МВС ЮФУ, 2007.
89. Paraschenko D., Tsarev F., Shalyto A. Modeling Technology for One Class of Multi-Agent Systems with Automata Based Programming // Proceedings of 2006 IEEE International Conference on Computational Intelligence for Measurement Systems and Application (IEEE CIMSA- 2006). Spain, 2006.
90. Шалыто А.А. Технология автоматного программирования // Труды первой Всероссийской научной конференции «Методы и средства обработки информации». – М.: МГУ, 2003.
91. Korneev G. A., Shalyto A. A. State-Driven Programming / Материалы Евразийского научного симпозиума. – Корея. Сеул% Политехнический университет, 2007. – Режим доступа: www.eurasia.re.kr
92. Шалыто А.А. Автоматное программирование / Тезисы докладов Международной научной конференции, посвященной памяти профессора А.М. Богомолова «Компьютерные науки и технологии». – Саратов: СГУ, 2007.
93. Шалыто А.А. Технология автоматного программирования // Мир ПК. – 2003. – № 10.
94. Головешин А. Использование конвертора Visio2Switch. http://is.ifmo.ru
95. Канжелев С.Ю., Шалыто А.А. Автоматическая генерация автоматного кода // Информационно-управляющие системы. – 2006. – № 6.
96. List of UML tools. – Режим доступа: http://en.wikipedia.org/wiki/List_of_UML_tools

97. О проекте «Технология автоматного программирования: применение и инструментальные средства» // Информационные технологии. – 2006. – № 2.
98. Гуров В.С., Мазин М.А., Нарвский А.С., Шалыто А.А. Разработка UML. SWITCH-технология. Eclipse // Информационно-управляющие системы. – 2004. – № 6.
99. Gurov V. S., Mazin M. A., №arvsky A. S, Shalyto A. A. UniMod: Method and Development of Reactive Object-Oriented Programs with Explicit States Emphasis // Proceedings 2005 of St. Petersburg IEEE Chapters. International Conference «110 Anniversary of Radio Invention SPb ETU «LETI». 2005.
100. Гуров В.С., Мазин М.А., Шалыто А.А. Ядро автоматного программирования // Свидетельство об официальной регистрации программы для ЭВМ. № 2006 613249 от 14.09.2006.
101. Гуров В.С., Мазин М.А., Шалыто А.А. Встраиваемый модуль автоматного программирования для среды разработки // Свидетельство об официальной регистрации программы для ЭВМ. № 2006 613817 от 07.11.2006.
102. Гуров В.С., Мазин М.А., Нарвский А.С., Шалыто А.А. Инструментальное средство для поддержки автоматного программирования // Программирование. – 2007. – № 6.
103. Гуров В.С., Нарвский А.С., Шалыто А.А. Исполняемый UML из России // PC Week/RE. – 2005. – № 26.
104. Мазин М.А., Шалыто А.А. Преступники и автоматы // Мир ПК. – 2004. – № 9.
105. Беляев А.В., Суясов Д.И., Шалыто А.А. Компьютерная игра «Космонавт». Проектирование и реализация // Компьютерные инструменты в образовании. – 2004. – № 4.
106. Корнеев Г.А., Петрошенко П.А., Шалыто А.А. Реализация игры «Морской бой» на основе автоматного подхода // Компьютерные инструменты в образовании. – 2005. – № 6.
107. Гуров В.С., Нарвский А.С., Шалыто А.А. ICQ и автоматы // Технология «Клиент-Сервер». – 2004. – № 3.
108. Мазин М.А., Парфенов В.Г., Шалыто А.А. Анимация. FLASH-технология. Автоматы // Компьютерные инструменты в образовании. – 2003. – № 4.
109. Naumov L., Shalyto A. Automata Theory for Multi-Agent Systems Implementation // International Conference on «Integration of Knowledge Intensive Multi-Agent Systems: Modeling, Exploration and Engineering». (KIMAS-03). Boston: IEEE. 2003.
110. Yartsev B., Korneev G., Kotov V., Shalyto A. Automata-Based Programming of the Reactive Multi-Agent Control Systems / 2005 International Conference on «Integration of Knowledge Intensive Multi-Agent Systems: Modeling, Exploration and Engineering» (KIMAS-05). Boston: IEEE. 2005.
111. Кретинин А. В., Солдатов Д. В., Шостак А. В., Шалыто А. А. Ракеты. Автоматы. Нейронные сети // Нейрокомпьютеры: разработка и применение. – 2005. – № 5.
112. Сайт по автоматному программированию. – Режим доступа: http://is.ifmo.ru
113. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы. Построение и анализ. – М.: МЦНМО, 1999.
114. Лобанов П.Г., Шалыто А.А. Подсчет длины слов в строке // Мир ПК. – 2005. – № 7.
115. Корнеев Г.А., Шамгунов Н.Н., Шалыто А.А. Обход деревьев на основе автоматного подхода // Компьютерные инструменты в образовании. – 2004. – № 3.
116. Туккель Н.И., Шалыто А.А. Реализация вычислительных алгоритмов на основе автоматного подхода // Телекоммуникации и информатизация образования. – 2001. – № 6.
117. Туккель Н.И., Шалыто А.А. Преобразование итеративных алгоритмов в автоматные // Программирование. – 2002. – № 5.

118. Туккель Н.И., Шамгунов Н.Н., Шалыто А.А. Реализация рекурсивных алгоритмов на основе автоматного подхода // Телекоммуникации и информатизация образования. – 2002. – № 5.
119. Казаков М.А., Корнеев Г.А., Шалыто А.А. Метод построения логики работы визуализаторов алгоритмов на основе конечных автоматов // Телекоммуникации и информатизация образования. – 2003. – № 6.
120. Корнеев Г.А., Шалыто А.А. Преобразование программ в систему взаимодействующих конечных автоматов / Труды Второй Всероссийской научной конференции «Методы и средства обработки информации». – М.: МГУ, 2005.
121. Казаков М.А., Шалыто А.А. Использование автоматного программирования для реализации визуализаторов // Компьютерные инструменты в образовании. – 2004. – № 2.
122. Казаков М.А., Шалыто А.А. Реализация анимации при построении визуализаторов алгоритмов на основе автоматного подхода // Информационно-управляющие системы. – 2005. – № 4.
123. Корнеев Г.А., Шалыто А.А. VIZI – язык описания визуализаторов алгоритмов // Научно-технический вестник СПбГУ ИТМО. – 2005. – Вып. 23. Высокие технологии в оптических и информационных системах.
124. Корнеев Г.А., Шалыто А.А. Построение визуализаторов алгоритмов дискретной математики // Научно-технический вестник СПбГУ ИТМО. – 2005. – Вып. 23. Высокие технологии в оптических и информационных системах.
125. Риган П., Хемилтон С. NASA: миссия надежна // Открытые системы. – 2004. – № 3.
126. Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. – СПбГУ ИТМО. 2007. 2008. – Режим доступа: http://is.ifmo.ru/verification/_2007_01_report-verification.pdf
127. Шалыто А.А. Триединая задача одного педагогического эксперимента в области IT-образования // Инженерное образование. – 2007. – № 4.
128. Вавилов К.В. Программируемые логические контроллеры SIMATIC S7-200 (SIEMENS). Методика алгоритмизации и программирования задач логического управления. – СПб: 2005. – Режим доступа: http://is.ifmo.ru/progeny/_metod065.pdf
129. Вавилов К.В. Контроллеры SIMATIC S7-300 (SIEMENS). Организация взаимодействия локальных систем управления на основе автоматного подхода и функционального разделения автоматов управления. – СПб, 2005. – Режим доступа: http://is.ifmo.ru/progeny/_s7300.pdf
130. Карпов Ю.Г. Имитационное моделирование систем. Введение в моделирование с AnyLogic 5. – СПб: БХВ-Петербург, 2006.
131. Дьяконов В.П. Simulink 5/6/7. Самоучитель. – М.: ДМК Пресс, 2008.
132. Ослэндер Д., Риджли Д., Ринггенберг Д. Управляющие программы для механических систем. Объектно-ориентированное проектирование систем реального времени. М.: БИНОМ. Лаборатория знаний. 2004.
133. Васильев В.Н., Казаков М.А., Корнеев Г.А., Парфенов В.Г., Шалыто А.А. Применение проектного подхода на основе автоматного программирования при подготовке разработчиков программного обеспечения // Труды первого Санкт-Петербургского конгресса «Профессиональное образование, наука, инновации в XXI веке». – СПбГУ ИТМО. – 2007. – С. 98–100. – Режим доступа: http://is.ifmo.ru/download/2008-02-25_comp_proekt.pdf
134. Красильников Н.Н., Парфенов В.Г., Царев Ф.Н., Шалыто А.А. Виртуальная лаборатория для первоначального обучения проектированию программ // Компьютерные инструменты в образовании. – 2007. – № 5. – С. 62–67.

135. Кузнецов Б.П. Психология автоматного программирования // BYTE/Россия. – 2000. – № 11. – С. 28–32. – Режим доступа: http://www.softcraft.ru/design/ap/ap01.shtml
136. 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.
137. Степанов О.Г., Шалыто А.А., Шопырин Д.Г. Предметно-ориентированный язык автоматного программирования на базе динамического языка Ruby // Информационно-управляющие системы. – 2007. – № 4. – С. 22–27.
138. Лагунов И.А. Разработка текстового языка автоматного программирования и его реализация для инструментального средства UniMod на основе автоматного подхода. – СПб: СПбГУ ИТМО, 2008. – Режим доступа: http://is.ifmo.ru/papers/fsml
139. Астафуров А.А., Шалыто А.А. Декларативный подход к вложению и наследованию автоматных классов при использовании императивных языков программирования // Материалы конференции «Software Engineering Confernece (Russia) – SEC(R) 2007». – М.: ТЕКАМА, 2007. – С. 230–238.
140. Stepanov O., Shalyto A. A Method for Automatic Runtime Verification of Automata-Based Programs // Proceeding of the Second Spring Young Researchers' Colloquium on Software Engineering (SYRCoSE'2008). – SPbSU. – 2008. – Vol.2. – P.19–23.
141. Решетников Е.О. Инструментальное средство для визуального проектирования автоматных программ на основе Microsoft Domain-Specific Language Tools. – СПбГУ ИТМО. – 2007. – Режим доступа: http://is.ifmo.ru/papers/reshetnikov_bachelor
142. Клебан В.О., Шалыто А.А. Использование автоматного программирования для построения многоуровневых систем управления мобильными роботами // Сборник тезисов 19 Всероссийской научно-технической конференции «Экстремальная робототехника». – СПб: ЦНИИ РТК, 2008. – С. 85–87.
143. Клебан В.О., Новиков Ф.А. Применение конечных автоматов в документообороте // Настоящий сборник.
144. Shalyto A. A. Technology of Automata-Based Programming. 2004. – Режим доступа: http://www.codeproject.com/KB/architecture/abp.aspx?print=true
145. Шалыто А.А. Парадигма автоматного программирования // Международная научно-техническая мультиконференция «Проблемы информационно-компьютерных технологий и мехатроники». Материалы международной научно-технической конференции «Многопроцессорные вычислительные и управляющие системы» (МВУС`2007). – Таганрог: НИИМВС, 2007. – Т.1. – С.191–194.
146. Гудвин Г.К., Гребе С.Ф., Сальгадо М.Э. Проектирование систем управления. М.: Бином, 2004.
147. Alur R., Courcoubetis C., Henzinger A., Ho P. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems // Lecture notes in computer science. 1993. – V. 736. – P. 209–229.
148. Brokett R. Reduced Complexity control systems / Proceedings of the 17 th World Congress The International Federation of Automatic Control. Seoul. 2008. – Эл. диск.

2 ПОСТРОЕНИЕ АВТОМАТОВ ДЛЯ УПРАВЛЕНИЯ БЕСПИЛОТНЫМИ УСТРОЙСТВАМИ НА ОСНОВЕ ПРИМЕНЕНИЯ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ

УДК 004.4’242

ПРИМЕНЕНИЕ ГЕНЕТИЧЕСКОГО ПРОГРАММИРОВАНИЯ ДЛЯ ГЕНЕРАЦИИ АВТОМАТОВ С БОЛЬШИМ ЧИСЛОМ ВХОДНЫХ ПЕРЕМЕННЫХ

Н.И. Поликарпова, В.Н. Точилин, А.А. Шалыто

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

Эффективность известных методов автоматической генерации конечных автоматов на основе генетического программирования экспоненциально снижается с ростом числа входных воздействий автомата. В работе предложен метод, который не имеет указанного недостатка. Предпочтительность применения предложенного метода при большом числе входных воздействий обоснована теоретически. Метод был применен в инструментальном средстве для автоматизации разработки системы управления самолетом на высоком уровне абстракции.

Ключевые слова: генетический алгоритм, автоматное программирование

Введение

В литературе описывается ряд методов построения конечных автоматов с помощью генетических алгоритмов, генетического программирования и других эволюционных подходов. Большинство работ в этой области посвящено построению автоматов-*распознавателей*, описывающих грамматику некоторого языка. Задача такого автомата состоит в определении принадлежности заданной строки языку. Распознаватель не производит выходных воздействий, результат определяется состоянием автомата после обработки входной последовательности. В данном направлении как наиболее значимые можно выделить работы [1–8]. Более сложная форма конечного автомата – *преобразователь* – отображает множество входных строк на множество выходных, возможно над другим алфавитом. Примером преобразователя может служить компилятор, а примером распознавателя – синтаксический анализатор, определяющий соответствие кода программы грамматике языка программирования. Эволюционному построению преобразователей посвящена работа [9].

В распознающих и преобразующих автоматах все условия переходов имеют вид сравнения входного символа с заданным. Таким образом, подмножество входных воздействий, удовлетворяющее условию конкретного перехода, всегда состоит из единственного входного символа.

В области генетического программирования традиционно используются модели вычислений в виде графов, которые можно интерпретировать как графы переходов конечных автоматов. Построению программ в виде графов посвящено большое число работ, например [10–14]. Применение автоматов в играх описано в работах [15–17].

Небольшое число работ затрагивают вопросы построения *управляющих автоматов* – автоматов, описывающих логику сложного поведения сущности или системы. Например, в статьях [18–20] рассматривается автоматическое построение компонент

программного обеспечения логических контроллеров в виде автоматов, а в работе [21] оценивается эффективность автоматных моделей применительно к различным задачам.

Практически во всех рассмотренных исследованиях автомат в каждый момент времени обрабатывает только одну входную переменную. Исключения составляют лишь работы [16] (четыре параллельных троичных входа) и [21] (в качестве условия перехода допускается сравнение значений двух регистров). Теоретически любое число параллельных входов сводится к одному, в качестве воздействий которого выступают комбинации сигналов исходных параллельных входов. Однако размер алфавита полученного таким образом входа растет экспоненциально с увеличением числа исходных параллельных входов. В упомянутых работах параллельные входы не приводят к недопустимо большому алфавиту, но для реальных систем эта проблема крайне актуальна.

Также в рассмотренных работах автомат на каждом шаге может выполнить не более одного действия из заданного множества. В таком случае любая комбинация действий, которые может потребоваться выполнить одновременно, также должна считаться элементарным действием. При этом требуется априорная информация обо всех возможных комбинациях действий либо задание вместе с элементарными действиями всех их наборов, что приводит к экспоненциальному росту числа действий. Отметим, что в работе [21] допускаются действия с аргументами, а также параллельно выполняемые автоматы, отвечающие за различные действия, что значительно ослабляет проблему.

В настоящей работе рассматривается задача автоматического построения на основе генетического программирования управляющих автоматов, графы переходов которых в общем случае помечаются булевыми формулами. Из всех перечисленных работ наилучшие результаты по этой тематике получены в статьях [19, 20], посвященных созданию управляющей программы робота. Однако и эти работы не лишены упомянутых выше недостатков, относящихся к входным и выходным воздействиям. Насколько известно авторам, исследования в области эволюционного построения логики систем со сложным поведением, отличных от роботов, ранее не проводились.

При разработке предлагаемого авторами метода сокращенных таблиц наибольшее внимание уделялось специфике управляющих автоматов – возможности построения автоматов с произвольным числом параллельных входов и выходов. Кроме того, для сокращения пространства поиска метод использует *концепцию автоматизированного объекта управления* [22]: логика сложного поведения описывается автоматом или системой автоматов на высоком уровне абстракции, а объект управления порождает входные и выходные данные и оптимизации не подвергается. Объект управления может быть произвольным (и, вообще говоря, сколь угодно сложным). Таким образом, предлагаемый метод решает задачу об использовании сложных структур данных в рамках генетического программирования, поставленную основателем генетического программирования *J. Koza* в работе [23].

Постановка задачи

Сформулируем решаемую в настоящей работе *задачу построения управляющего автомата*. Пусть задан объект управления $O = \langle V, v_0, X, Z \rangle$, где V – множество вычислительных состояний (или *значений*), v_0 – *начальное значение*, $X = \{x_i : V \to \{0,1\}\}_{i=1}^{n}$ – множество предикатов, $Z = \{z_i : V \to V\}_{i=1}^{m}$ – множество действий. Также задана оценочная функция $\varphi : V \to \mathbf{R}^+$ и натуральное число k.

Объект O может управляться автоматом вида $A = \langle S, s_0, \Delta \rangle$, где S – конечное множество управляющих состояний, s_0 – *стартовое состояние*, $\Delta : S \times \{0,1\}^n \to S \times Z^*$

– *управляющая функция*. Управляющую функцию можно разложить на две компоненты: функцию выходов $\zeta : S \times \{0,1\}^n \to Z^*$ и функцию переходов $\delta : S \times \{0,1\}^n \to S$.

Пусть перед началом работы объекту управления соответствует начальное значение v_0, а автомат находится в стартовом состоянии s_0. Назовем *шагом работы* автоматизированного объекта следующую последовательность операций.

1. Объект управления вызывает все предикаты из множества X и формирует из их значений вектор *входного воздействия* $in \in \{0,1\}^n$.
2. Автомат вычисляет значение вектора *выходного воздействия* $out = \zeta(s,in)$, где s – текущее состояние автомата, и переходит в новое управляющее состояние $s_{new} = \delta(s,in)$.
3. Объект управления по очереди вызывает действия $z \in out$, изменяя при этом текущее вычислительное состояние [22].

Задача построения управляющего автомата состоит в том, чтобы найти автомат заданного вида такой, что за k шагов работы под управлением этого автомата объект O перейдет в вычислительное состояние с максимальной пригодностью ($\varphi(v) \to \max$).

В связи с использованием генетического программирования для этой задачи возникают следующие подзадачи: выбор представления конечного автомата в виде особи; адаптация генетических операторов (мутации и скрещивания) для выбранного представления; настройка параметров генетической оптимизации.

В классической интерпретации генетического алгоритма особь представляется в виде набора хромосом. Управляющий автомат можно представить как набор состояний, в каждом из которых его поведение определяется сужением управляющей функции $\Delta_s : \{0,1\}^n \to S \times Z^*$, $s \in S$. Таким образом, удобно сопоставить каждому состоянию хромосому. Следовательно, от задачи представления автомата в виде особи перейдем к более конкретной постановке проблемы – представлению управляющего состояния автомата в виде хромосомы.

Представление в виде хромосомы

Основная проблема, возникающая при использовании представления состояния автомата в виде таблицы переходов и действий – это экспоненциальный рост размерности хромосомы с увеличением числа предикатов объекта управления (напомним, что число строк в таблице 2^n, где n – число предикатов). Опыт показывает, что в реальных задачах управляющие автоматы, построенные вручную, имеют гораздо меньше переходов. Причина этого состоит, видимо, в том, что в большинстве задач предикаты имеют «локальную природу» по отношению к управляющим состояниям. В каждом состоянии *значимым* является лишь определенный, небольшой поднабор предикатов, остальные же не влияют на значение управляющей функции. Именно это свойство позволяет существенно сократить размер описания состояний. Кроме того, использование этого свойства в процессе оптимизации позволяет получить результат, более похожий на автомат, построенный вручную, а, следовательно, и более понятный человеку.

Свойство локальности предикатов можно использовать для сокращения описания управляющего состояния разными способами. Авторами выбран один из подходов, при котором число значимых в состоянии предикатов ограничивается некоторой константой r. К стандартной таблице, задающей сужение управляющей функции на данное состояние, в этом случае добавляется битовый вектор, описывающий множество значимых предикатов (рис. 1). Число строк таблицы в этом случае 2^r. При этом константа r обычно не-

велика. Ее выбор зависит от сложности задачи. Как показывает опыт, для большинства автоматизированных объектов среднее по всем состояниям значение r не больше пяти.

x_0	x_1	x_2	x_3	x_4	x_5
0	1	0	1	0	0

x_1	x_3	s	z_0	z_1	z_2
0	0	0	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	2	1	1	1

Рис. 1. Хромосома состояния: сокращенная таблица (n = 6, r = 2).

Генетические операции

Опишем генетические операторы над хромосомами состояний, записанными в виде сокращенных таблиц.

Алгоритм 1. Мутация сокращенных таблиц. По сравнению со стандартным представлением добавилась возможность мутации множества значимых предикатов. При этом каждый из значимых предикатов с некоторой вероятностью заменяется другим, который не принадлежит множеству (рис. 2).

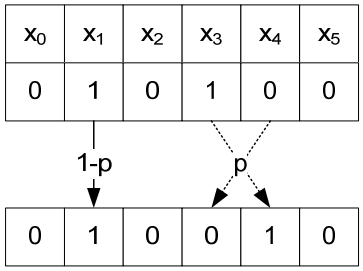


Рис. 2. Пример мутации множества значимых предикатов.

Мутация самой сокращенной таблицы происходит так же, как мутация полной таблицы. Описание алгоритма приведено в листинге 1 приложения.

Алгоритм 2. Скрещивание сокращенных таблиц. Это наиболее сложный из предлагаемых алгоритмов. Основная последовательность его шагов отражена в листинге 2 приложения.

Поскольку родительские хромосомы, представленные сокращенными таблицами, могут иметь разные множества значимых предикатов, сначала необходимо выбрать, какие из этих предикатов будут значимы для хромосом детей. Функция `ChoosePreds`, осуществляющая этот выбор, представлена в листинге 3 приложения. В результате работы функции `ChoosePreds` предикаты, значимые для обоих родителей, наследуются обоими детьми, а каждый из тех предикатов, которые были значимы лишь для одной родительской особи, равновероятно достанется любому из двух детей. Пример работы функции для родительских хромосом, представленных на рис. 3, проиллюстрирован на рис. 4.

После выбора значимых предикатов заполняются таблицы обоих детей. Алгоритм заполнения представлен в листинге 4 приложения.

Иллюстрация примера заполнения первой строки таблицы одного из детей приведена на рис. 5. В данной реализации оператора скрещивания на значения каждой стро-

ки таблицы ребенка влияют значения нескольких строк родительских таблиц. При этом конкретное значение, помещаемое в ячейку таблицы ребенка, определяется «голосованием» всех влияющих на нее ячеек родительских таблиц.

x_0	x_1	x_2	x_3	x_4	x_5
0	1	0	1	0	0

x_0	x_1	x_2	x_3	x_4	x_5
1	1	0	0	0	0

x_1	x_3	s	z_0	z_1	z_2
0	0	0	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	1	2	1	1	1

x_0	x_1	s	z_0	z_1	z_2
0	0	1	1	1	0
0	1	2	0	0	1
1	0	2	0	1	0
1	1	0	0	1	0

Рис. 3. Родительские хромосомы, представленные сокращенными таблицами.

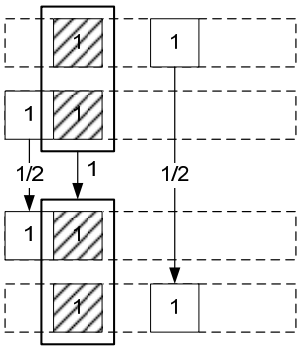


Рис. 4. Пример выбора значимых предикатов детей.

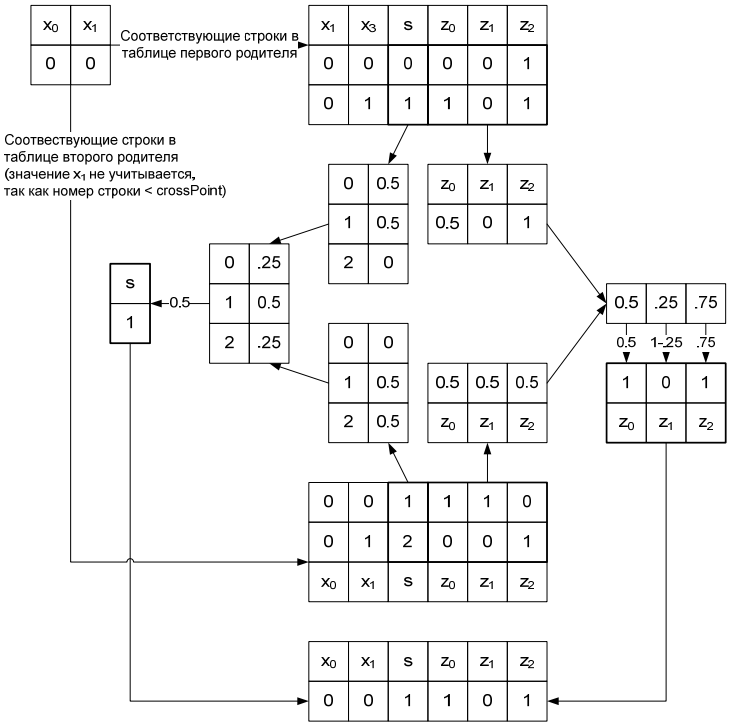


Рис. 5. Пример заполнения строки таблицы ребенка при скрещивании сокращенных таблиц

В описанном выше варианте алгоритма все состояния автомата имеют равное число значимых предикатов (r – константа для всего процесса оптимизации). Однако предложенный алгоритм скрещивания легко расширяется на случай разного числа значимых предикатов у пары родителей.

Эффективность метода сокращенных таблиц

Для оценки характеристик метода сокращенных таблиц был проведен ряд экспериментов, в которых эффективность данного метода сравнивалась с эффективностью метода полных таблиц [24]. Сравнение проводилось по следующим критериям: объем занимаемой памяти, время, затрачиваемое на обработку каждого поколения, скорость роста функции пригодности (от номера поколения и от времени).

Как упоминалось выше, основное достоинство метода сокращенных таблиц состоит в том, что он решает проблему экспоненциального роста размера описания автомата с увеличением числа входных переменных (предикатов объекта управления). Это свойство метода подтвердилось при экспериментальной проверке. Во время эксперимента описания автоматов хранились в памяти компьютера. Поэтому объем занимаемой памяти в этом случае прямо пропорционален размеру описания автомата. Результаты эксперимента приведены на рис. 6, a.

Из рассмотрения графика следует, что объем занимаемой памяти при использовании метода полных таблиц растет экспоненциально, в то время как при использовании метода сокращенных таблиц объем занимаемой памяти с ростом числа предикатов практически не изменяется. В действительности он зависит от числа предикатов линейно, однако коэффициент линейной зависимости настолько мал, что в экспериментах она не отражается. Аналогичный характер имеет зависимость времени, требуемого на обработку каждого поколения, от числа предикатов (рис. 6, b).

Теперь перейдем к оценке скорости роста функции пригодности. На рис. 6, c, приведены зависимости значения оценочной функции от номера поколения при использовании метода полных таблиц и метода сокращенных таблиц (измерения приводились при небольшом числе предикатов). Из последнего графика следует, что оптимизация методом полных таблиц требует вычисления меньшего числа поколений. Это означает, что при небольшом числе предикатов метод полных таблиц обладает более высоким быстродействием. Однако с ростом числа предикатов стремительно растет время обработки одного поколения методом полных таблиц. Кроме того, из-за экспоненциального роста объема требуемой памяти применение метода полных таблиц, начиная с некоторого числа предикатов, становится не просто неэффективным, а практически невозможным. В экспериментах авторам не удалось построить методом полных таблиц автомат с более чем 14 входными переменными. Отметим также, что автоматы, которые построены методом полных таблиц, практически невозможно изобразить и понять, так как в них присутствует большое число избыточных переходов, а условия на переходах громоздки. Напротив, автоматы, построенные методом сокращенных таблиц, могут быть сравнительно легко поняты человеком.

Чтобы адекватно оценить быстродействие обоих методов, необходимо установить зависимость значения оценочной функции, достигаемого с помощью каждого метода за определенное время, от числа предикатов. Такая зависимость для промежутка времени, равного пяти минутам, приведена на рис. 6, d. Как и ожидалось, приведенные зависимости показывают, что при небольшом числе предикатов метод полных таблиц имеет более высокое быстродействие, однако с ростом числа предикатов его быстродействие резко падает. В то же время быстродействие метода сокращенных таблиц с ростом числа предикатов уменьшается незначительно.

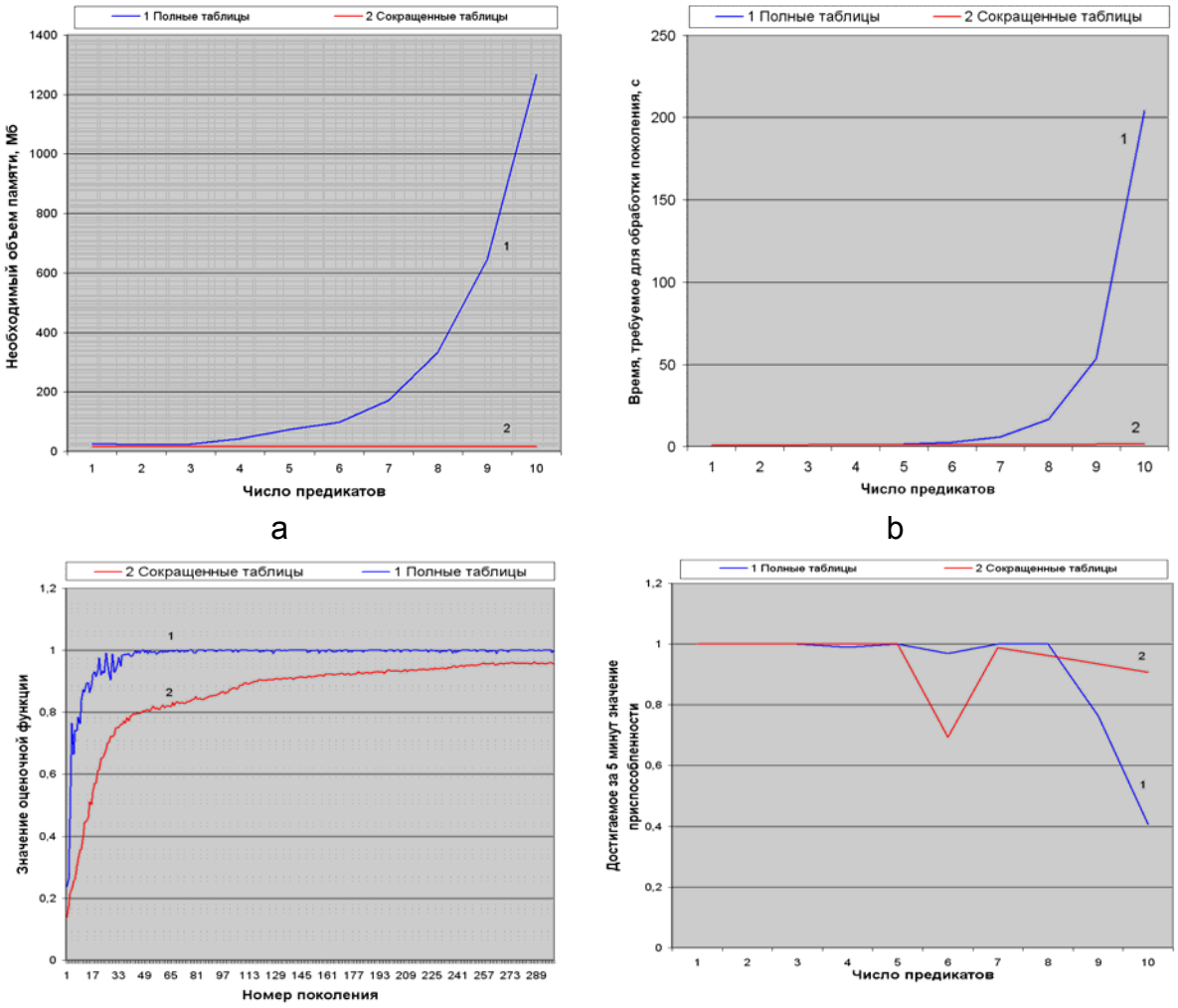


Рис. 6. Зависимости объема занимаемой памяти (a), времени обработки поколения (b), значения оценочной функции для заданной продолжительности работы метода (d) от числа предикатов и значения оценочной функции от номера поколения (c)

.

Из приведенного исследования характеристик методов полных и сокращенных таблиц можно сделать следующие выводы:

- при небольшом числе предикатов метод полных таблиц является более эффективным по времени и достаточно эффективным по памяти, однако построенные этим методом автоматы непонятны человеку;
- начиная с некоторого числа предикатов применение метода полных таблиц практически невозможно, в то время как метод сокращенных таблиц остается достаточно эффективным и по времени и по памяти.

Для генерации автоматов на основе предложенного метода авторами было разработано инструментальное средство, которое опубликовано на сайте http://is.ifmo.ru в разделе «Работы».

Применение метода сокращенных таблиц для автоматического управления самолетом

Предложенный метод был применен для разработки системы автоматического управления моделью самолета на высоком уровне абстракции. Эта система должна быть реализована конечным автоматом. При этом предполагается, что на более низком

уровне абстракции используется автопилот, который управляет самолетом в каждом состоянии конечного автомата. Этот автопилот в настоящей работе не строится, а считается заданным. Разработанная система должна обеспечить выполнение полета без участия пилота и не должна была предъявлять нестандартных требований к наземному оборудованию.

Ниже представлена последовательность действий, выполняемых при применении метода сокращенных таблиц для автоматического управления самолетом.

1. Разработка (выбор) эмулятора самолета.
2. Построение преобразователя интерфейсов.
    1. Выбор и реализация входных воздействий.
    2. Выбор и реализация выходных воздействий.
3. Выбор функции приспособленности.
4. Генерация автомата.
    1. Настройка эмулятора.
    2. Порождение популяции псевдослучайных автоматов.
    3. Взаимодействие автоматов с эмулятором через преобразователь интерфейсов.
    4. Оценка каждого автомата с помощью функции приспособленности.
    5. Использование генетических операций для генерации автоматов с вероятно более высоким значением функции приспособленности.
    6. Если максимальная полученная оценка функции приспособленности меньше целевого значения, то переход к пункту 4.4.
    7. Запись автомата с максимальной оценкой функции приспособленности.
    8. Ручное упрощение графа переходов для повышения «понятности» автомата.
5. Проведение эксперимента по управлению моделью самолета сгенерированным автоматом.
    1. Выполнение написанных вручную блоков прокладки маршрута и настройки навигации.
    2. Запуск системы автомат – преобразователь интерфейсов – эмулятор.
    3. Формирование результатов моделирования.
    4. Анализ результатов моделирования.

Разработка (выбор) эмулятора самолета

В рассматриваемой задаче объектом управления является самолет, находящийся в некоторой среде, включающей воздух, возможно движущийся, взлетно-посадочные полосы, ландшафт, службу управления полетами, радиомаяки и многое другое. Требуется эмулировать аэродинамику, механику и работу оборудования самолета. Ввиду трудоемкости разработки необходимого эмулятора представляется естественным использование эмулятора стороннего производителя. Для этой цели был выбран авиационный симулятор *X-Plane* [25] . Данный симулятор представляет собой игру для персонального компьютера, обладающую достаточной точностью физической эмуляции, высокой скоростью работы, возможностью взаимодействия с другими программами и недорогой лицензией, не препятствующей ее использованию в данном проекте.

Выбор и реализация входных воздействий

Выбранные входные воздействия приведены в табл. 1.

Таблица 1. Входные воздействия автомата

Идентификатор	Описание значения
x1	Самолет движется
x2	Скорость самолета достаточна для полета с выпущенными за-

	крылками
x3	Скорость самолета достаточна для полета с убранными закрылками
x4	Самолет летит
x5	Самолет находится рядом с поверхностью земли
x6	Высота полета соответствует рекомендованной высоте эшелона
x7	Самолет находится рядом с опорной точкой GPS-приемника

Выбор и реализация выходных воздействий

Выбранный набор действий приведен в табл. 2.

Таблица 2. Выходные воздействия автомата

Идентификатор	Описание действия
z1	Настройка навигационного приемника на частоту посадочного маяка аэропорта отправления
z2	Настройка навигационного приемника на частоту посадочного маяка аэропорта прибытия
z3	Перевод GPS-приемника в режим следования к опорной точке
z4	Установка переключателя управления полетом в положение “АВТО”
z5	Переключение автопилота в режим полета на точку в горизонтальной плоскости
z6	Переключение автопилота в режим полета по курсу в горизонтальной плоскости
z7	Переключение автопилота в режим снижения по маяку
z8	Переключение автопилота в режим постоянной высоты
z9	Переключение автопилота в режим постоянной вертикальной скорости
z10	Включение колесного тормоза
z11	Выключение колесного тормоза
z12	Установка максимальной подачи топлива
z13	Установка средней подачи топлива
z14	Установка минимальной подачи топлива
z15	Убирание закрылков
z16	Установка закрылков во взлетное положение
z17	Установка закрылков в посадочное положение
z18	Выпуск шасси
z19	Убирание шасси
z20	Использование навигационного приемника в качестве источника сигнала индикатора горизонтальной ситуации
z21	Использование GPS-приемника в качестве источника сигнала индикатора горизонтальной ситуации
z22	Установка скорости набора высоты и высоты полета на автопилоте
z23	Управление рулем направления в соответствии с сигналами автопилота
z24	Установка руля направления в центральное положение

Соответствующие входным переменным приборы и соответствующие действиям органы управления эмулятором самолета показаны на рис. 7. Неотмеченные приборы не используются.

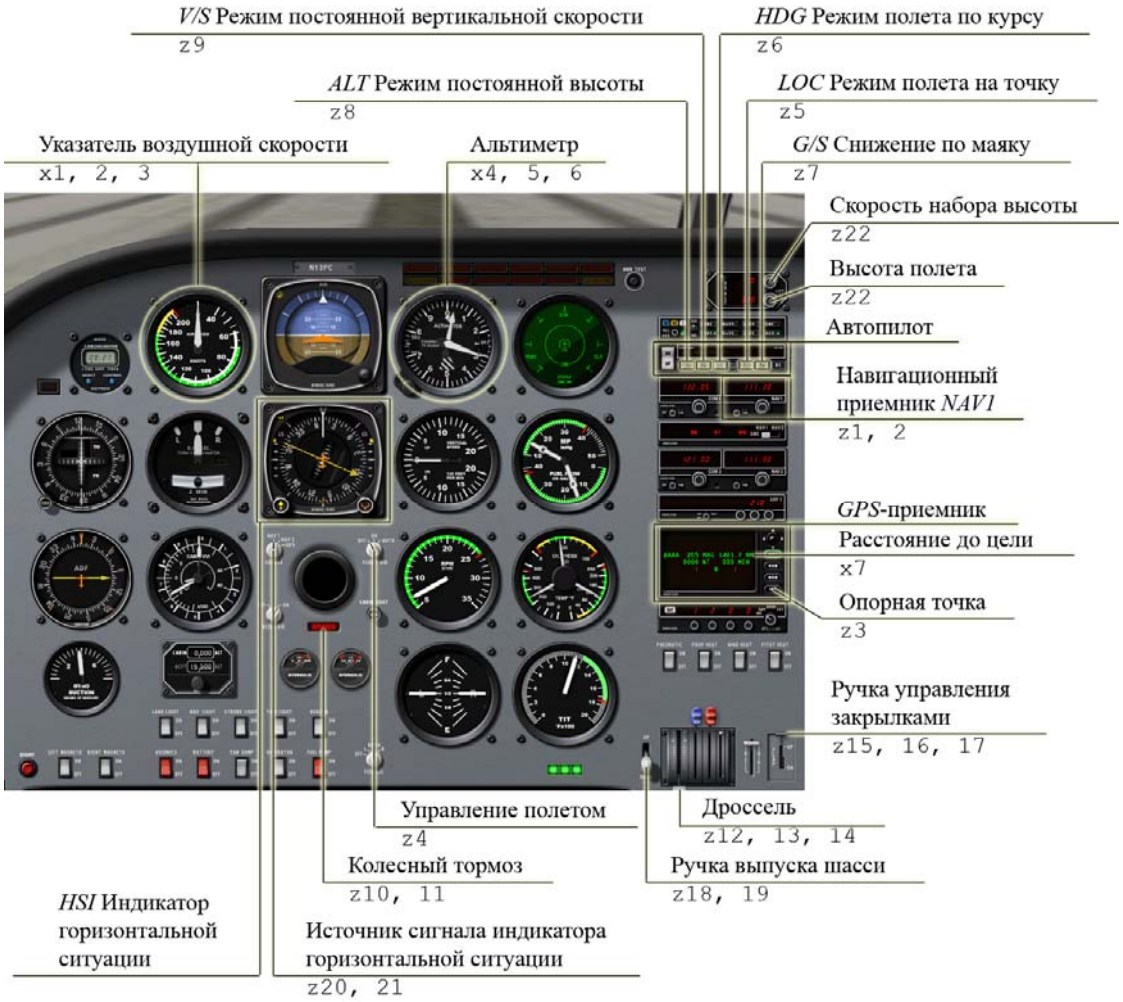


Рис. 7. Используемые приборы и органы управления самолетом.

Функция приспособленности

Переформулируем задачу в виде функции приспособленности. Выберем существенные характеристики, которые будут влиять на значение функции приспособленности и использоваться при ее вычислении. От автопилота требуется провести самолет по маршруту и не разбить его. Следовательно, можно выделить два значимых фактора: отклонение от маршрута и сохранность самолета. Кроме того, важно, чтобы после прохождения маршрута самолет остановился (или снизил скорость до безопасной) на взлетно-посадочной полосе. Совокупное отклонение от маршрута как по положению, так и по скорости будем вычислять в виде интегралов модулей соответствующих отклонений по времени:

$$P = \int_{t_0}^{t_1} |p(t)| * dt ,$$

где P – совокупное отклонение от маршрута по положению, t_0 – время начала эмуляции, t_1 – время окончания эмуляции, $p(t)$ – отклонение по положению в момент времени t;

$$V = \int_{t_0}^{t_1} |v(t)| * dt ,$$

где V – совокупное отклонение от оптимальной скорости, а $v(t)$ – отклонение от оптимальной скорости в момент времени t. Требуется оптимизировать систему управления по трем параметрам. Для того чтобы свести многокритериальную оптимизацию к однокритериальной, выберем функцию приспособленности вида

$$f = \frac{t_1 - t_0}{\left(100 + \frac{P+V}{t_1 - t_0}\right) * 0{,}77 * (1 + b * 9)},$$

где b – переменная, равная нулю при целом самолете и единице при разбитом.

Построение управляющего автомата

После описания всех необходимых элементов программы можно проверить ее работоспособность. Основными показателями являются динамика значений функции приспособленности в процессе работы программы и качество автомата, получаемого после завершения работы. График изменения оценки приспособленности с ростом номера поколения автоматов приведен на рис. 8. Единичное значение функции приспособленности на двух последних поколениях свидетельствует об успешном завершении процесса оптимизации.

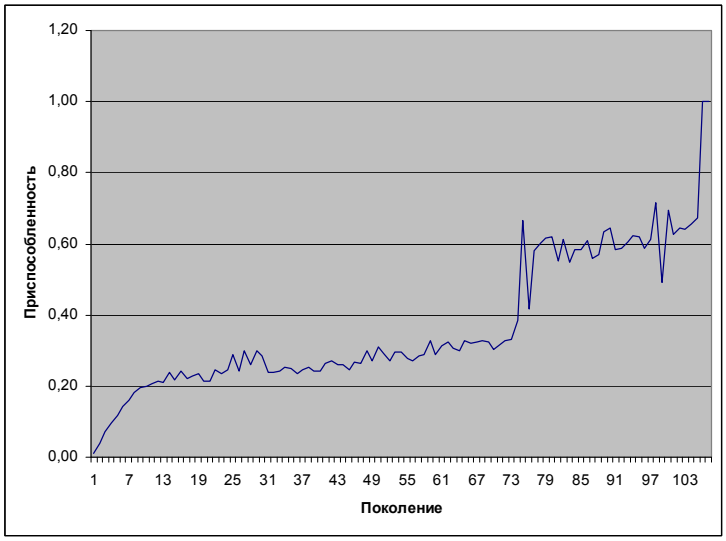


Рис. 8. Изменение приспособленности в процессе эволюции

При генерации автомата было наложено ограничение на число переменных `n`, проверяемых на переходах из состояний. Даже при `n` = 1 четырехпроцессорный кластер строил автомат около месяца (значение функции приспособленности вычислялось около пяти минут). При этом построение автомата, корректно управляющего самолетом, при принятом ограничении не гарантировалось.

Полученный в результате работы автомат, несмотря на небольшое число состояний и переходов, достаточно сложен для восприятия человеком ввиду большого числа вызываемых на переходах действий. Для упрощения автомата было произведено формальное, но не изменяющее поведение автомата, сокращение числа вызываемых на переходах действий. Кроме того, состояния были перенумерованы и снабжены названиями. Полученный автомат приведен на рис. 9.

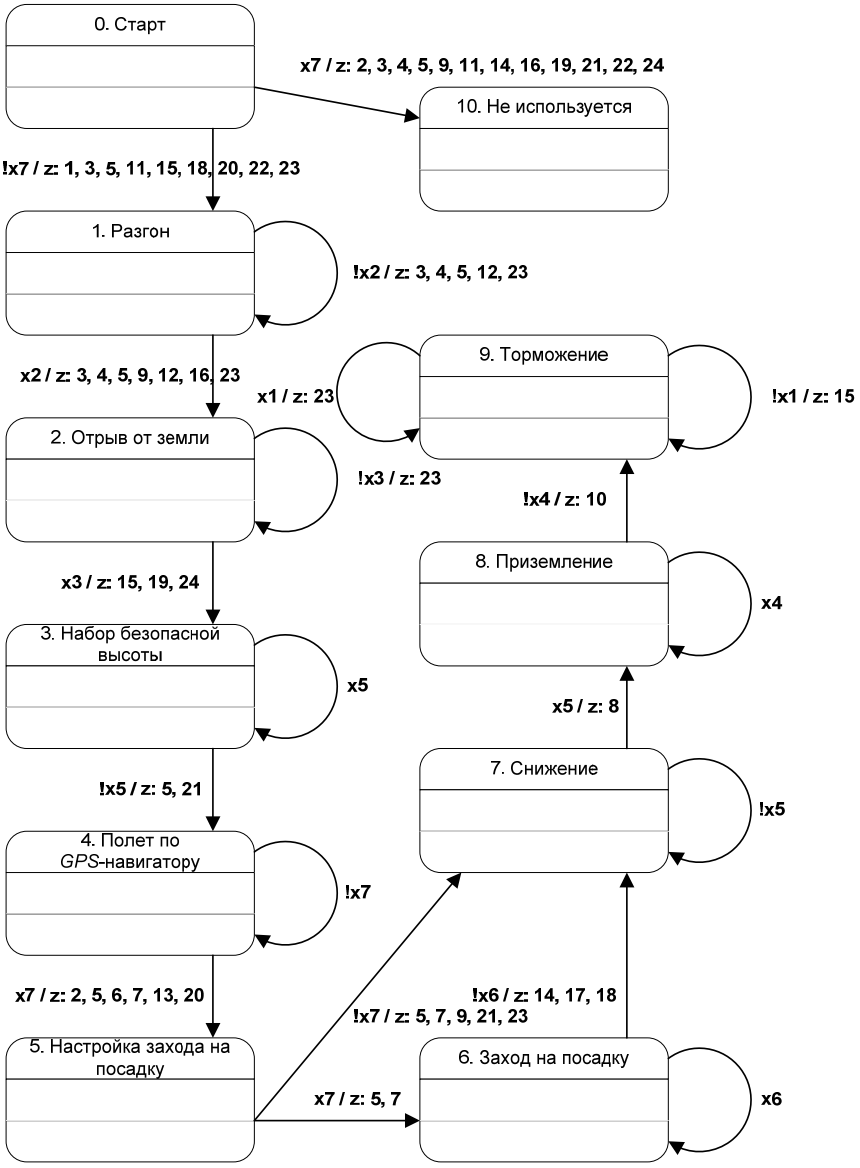
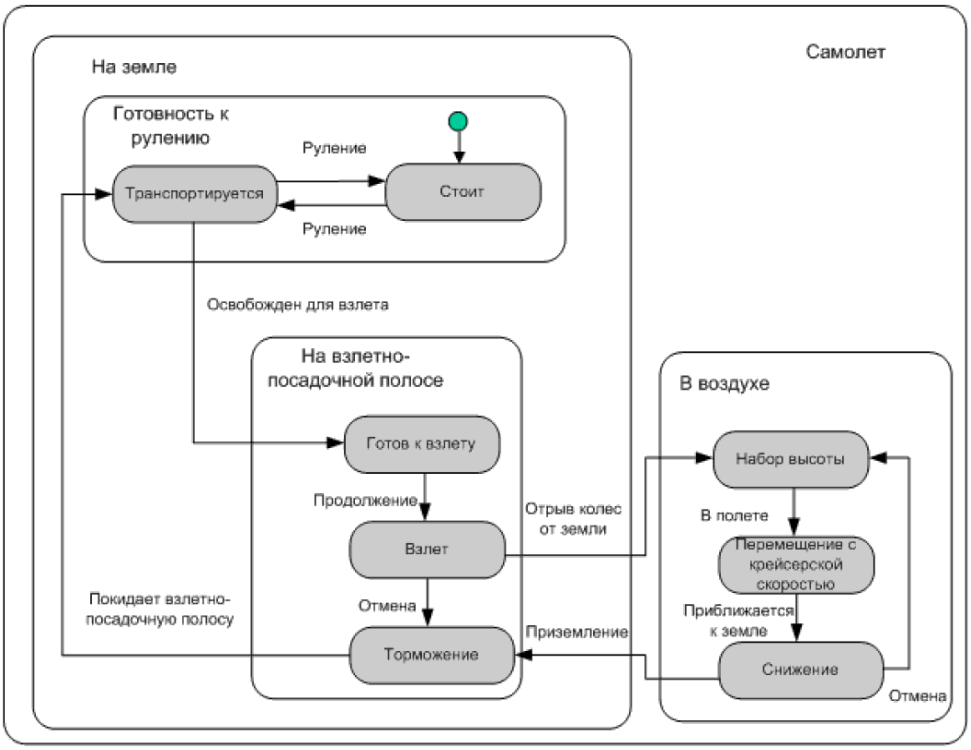


Рис. 9. Граф переходов полученного автомата

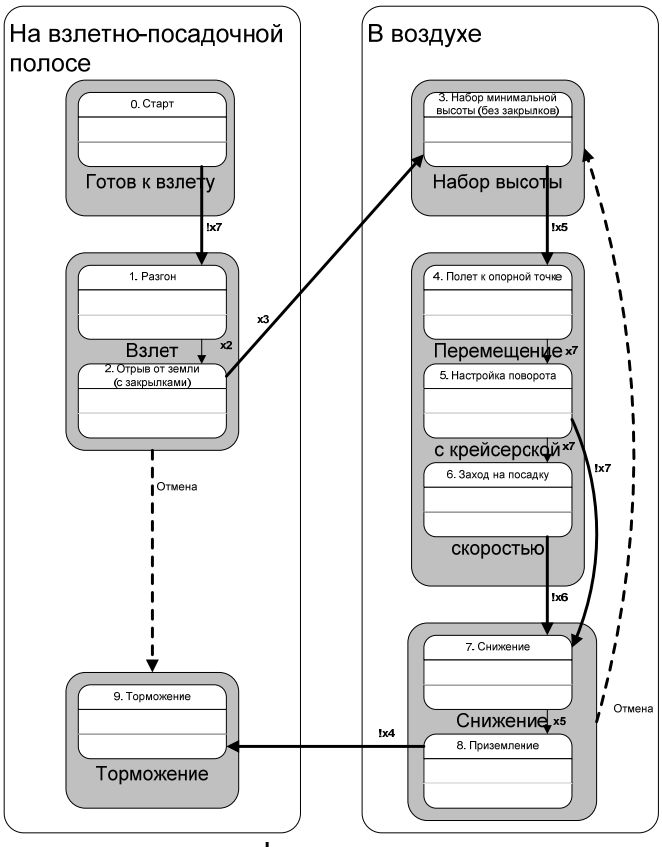
Простота структуры полученного управляющего автомата объясняется использованной конфигурацией метода сокращенных таблиц. Число анализируемых в каждом состоянии входных воздействий, как отмечалось в предыдущем разделе, было выбрано равным единице. Следовательно, из каждого состояния могло быть не более двух переходов. В рассматриваемом случае выразительной силы практически линейной структуры автомата было достаточно в результате двух факторов:

- высокоуровневые входные и выходные воздействия автомата;
- ограниченные требования к функциональности системы управления.

Изображенный на рис. 9 автомат может быть сравнительно легко построен эвристически (без использования генетического программирования). Однако в общем случае сложность результата не всегда находится в прямой зависимости от сложности процесса его получения. Построенный эвристически автомат часто оказывается неоправданно сложным по сравнению с полученным автоматически. Кроме того, подход на основе автоматической генерации характеризуется лучшей масштабируемостью.



a



b

Рис. 10. Диаграмма состояний для полета самолета из книги [27] (a) и ее сопоставление с автоматически построенным автоматом (b)

Сравним полученный автомат с автоматом, построенным эвристически. В книге [27] в качестве примера диаграммы состояний приведен граф переходов, описывающий полет самолета (рис. 10, a). На рис. 10, b, состояниям и переходам этого автомата сопоставляются состояния и переходы автоматически построенного автомата с рис. 9. Автоматически построенный автомат отличается отсутствием состояния готовности к рулению и связанных с ним переходов, а также отсутствием переходов отмены между состояниями «Взлет» и «Торможение» и между состояниями «Снижение» и «Набор высоты». Отсутствующие элементы выделены на рисунке пунктиром. Различия между автоматами объясняются отсутствием этапа руления и возможности отмены взлета или снижения в процессе обучения.

Проведение эксперимента по управлению моделью самолета сгенерированным автоматом

Работоспособность автомата в качестве системы управления моделью самолета проверяетс0я с помощью того же инструментального средства, на котором был сгенерирован автомат. Работа инструментального средства в этом режиме аналогична режиму генерации автомата с единственным автоматом в поколении и без применения генетических модификаций. Автомат, положение самолета и показания приборов в каждом состоянии показаны на рис. 11.

1

2


10.
Не используется
0. Старт
1. Разгон
2. Отрыв
от земли
3. Набор
безопасной
высоты
4. Полет по
GPS-навигатору
5. Настройка
захода на
посадку
6. Заход
на посадку
7. Снижение
8. Приземление
9. Торможение

3


10.
Не используется
0. Старт
1. Разгон
2. Отрыв
от земли
3. Набор
безопасной
высоты
4. Полет по
GPS-навигатору
5. Настройка
захода на
посадку
6. Заход
на посадку
7. Снижение
8. Приземление
9. Торможение

4


10.
Не используется
0. Старт
1. Разгон
2. Отрыв
от земли
3. Набор
безопасной
высоты
4. Полет по
GPS-навигатору
5. Настройка
захода на
посадку
6. Заход
на посадку
7. Снижение
8. Приземление
9. Торможение

5


10.
Не используется
0. Старт
1. Разгон
2. Отрыв
от земли
3. Набор
безопасной
высоты
4. Полет по
GPS-навигатору
5. Настройка
захода на
посадку
6. Заход
на посадку
7. Снижение
8. Приземление
9. Торможение

6

7

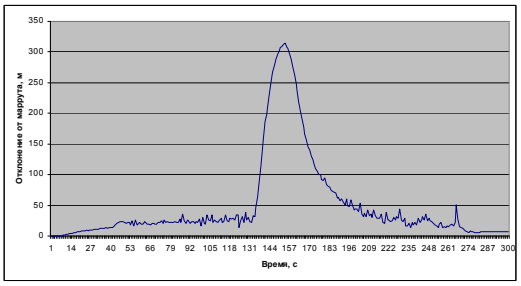
8

9

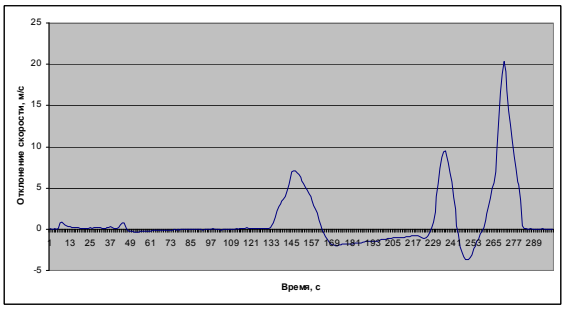
Рис. 11. Автомат и самолет в различных состояниях

Формирование результатов моделирования

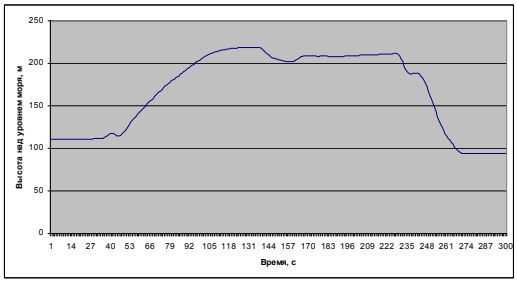
В процессе тестового использования сгенерированного автомата для управления самолетом фиксировались отклонения от маршрута и отклонения от рекомендованной скорости. Так как допустимые отклонения зависят от этапа полета, также фиксировалась высота полета, позволяющая определить моменты отрыва от земли при взлете и касания земли при приземлении. Кроме того, записывались действующие в процессе полета перегрузки, которые не учитывались при выводе автомата. Все записанные данные показаны на рис. 12.



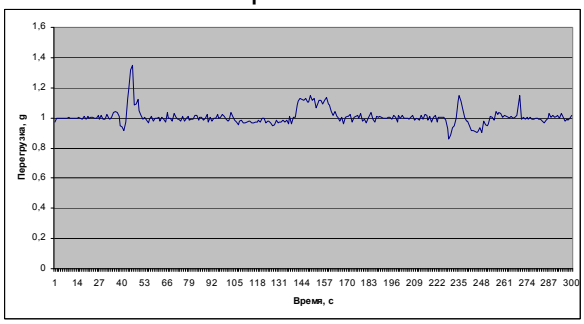
Отклонения от маршрута



Отклонения от рекомендованной скорости



Высота полета над уровнем моря



Перегрузки

Рис. 12. Результаты тестирования

Анализ результатов моделирования

Из графика следует, что максимальное отклонение самолета от маршрута за все время полета составило 314 м. При этом отклонение от маршрута не превышало 60 м на протяжении всего полета, кроме минутного участка в середине графика, когда отклонения наименее критичны. По графику высоты полета можно определить время отрыва и касания земли: 35 с и 271 с соответственно. Из установленного времени и графика отклонений от маршрута следует, что максимальное отклонение при разгоне составило 12 м, а при торможении – 10 м. Существенные, но не выходящие за пределы взлетно-посадочных полос отклонения позволяют считать, что сгенерированный автомат провел самолет по маршруту с достаточной точностью.

Из графика рекомендованной скорости следует, что скоростной режим соблюден, за исключением превышения скорости на 20,34 м/с в момент касания земли, которое было компенсировано при торможении и не привело к существенному превышению тормозного пути. Совокупное отклонение положения самолета после остановки, включающее поперечное отклонение от центра полосы и превышение тормозного пути, составляет 6,5 м. Из графика перегрузок следует, что ускоренное торможение не привело к существенному дискомфорту пассажиров. Проведенный анализ позволяет сделать заключение об удовлетворительном качестве автоматически построенного автомата.

Заключение

В работе была показана низкая эффективность существующих методов генерации автоматов с большим числом входных воздействий и предложен метод, который не имеет указанных недостатков. Изменение соотношения эффективностей известного и предложенного методов с ростом числа входных воздействий в пользу предложенного метода было обосновано теоретически и проверено на модельной задаче. Получение работоспособного и эффективного автомата в результате применения предложенного

метода для генерации системы управления самолетом показало его применимость для решения практических задач.

Литература

1. Gold E. M. Language Identification in the Limit // Information and Control. – 1967. – №10. – P.447–474.
2. Belz A. Computational Learning of Finite-State Models for Natural Language Processing. PhD thesis. – University of Sussex. 2000.
3. Clelland C. H., Newlands D. A. Pfsa modelling of behavioural sequences by evolutionary programming // Complex'94 – Second Australian Conference on Complex Systems. – IOS Press, 1994. – P.165–172.
4. Das S., Mozer M. C. A Unified Gradient-Descent/Clustering Architecture for Finite State Machine Induction. // Advances in Neural Information Processing Systems. – 1994.
5. Lankhorst M. M. A Genetic Algorithm for the Induction of Nondeterministic Pushdown Automata. // Computing Science Report. – University of Groningen Department of Computing Science. – 1995.
6. Belz A., Eskikaya B. A genetic algorithm for finite state automata induction with an application to phonotactics // ESSLLI-98 Workshop on Automated Acquisition of Syntax and Parsing. – Saarbruecken. – 1998. – P. 9–17.
7. Ashlock D., Wittrock A., Wen T-J. Training finite state machines to improve PCR primer design // Congress on Evolutionary Computation (CEC'02). – 2002. – P.13–18.
8. Ashlock D. A., Emrich S. J., Bryden K. M. et al. A comparison of evolved finite state classifiers and interpolated markov models for improving PCR primer design // 2004 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB'04). – 2004. – P.190–197.
9. Lucas S. M. Evolving Finite State Transducers: Some Initial Explorations // Genetic Programming: 6th European Conference (EuroGP'03). – Berlin: Springer, 2003. – P.130–141.
10. Teller A., Veloso M. PADO: A New Learning Architecture for Object Recognition. Symbolic Visual Learning. – New York: Oxford University Press. 1996. P. 81–116.
11. Banzhaf W., Nordin P., Keller R. E., Francone F. D. Genetic Programming – An Introduction. // On the automatic Evolution of Computer Programs and its Application. – San Francisco: Morgan Kaufmann Publishers, 1998.
12. Kantschik W., Dittrich P., Brameier M. Empirical Analysis of Different Levels of Meta-Evolution // Congress on Evolutionary Computation. – 1999.
13. Kantschik W., Dittrich P., Brameier M., Banzhaf W. Meta-Evolution in Graph GP // Genetic Programming: Second European Workshop (EuroGP'99). – 1999.
14. Teller A., Veloso M. Internal Reinforcement in a Connectionist Genetic Programming Approach // Artificial Intelligence. North-Holland Pub. Co. – 1970. – P.161.
15. Miller J. H. The Coevolution of Automata in the Repeated Prisoner's Dilemma. // Working Paper. Santa Fe Institute. – 1989.
16. Spears W. M., Gordon D. F. Evolving Finite-State Machine Strategies for Protecting Resources // International Syposium on Methodologies for Intelligent Systems. – 2000.
17. Ashlock D. Evolutionary Computation for Modeling and Optimization. – New York: Springer. 2006.
18. Frey C., Leugering G. Evolving Strategies for Global Optimization. A Finite State Machine Approach // Genetic and Evolutionary Computation Conference (GECCO-2001). – Morgan Kaufmann. – 2001. – P. 27–33.
19. Petrovic P. Simulated evolution of distributed FSA behaviour-based arbitration // The Eighth Scandinavian Conference on Artificial Intelligence (SCAI'03). – 2003.

20. Petrovic P. Evolving automatons for distributed behavior arbitration. // Technical Report. – Norwegian University of Science and Technology. – 2005.
21. Petrovic P. Comparing Finite-State Automata Representation with GP-trees. // Technical report. – Norwegian University of Science and Technology. – 2006.
22. Поликарпова Н.И., Шалыто А.А. Учебно-методическое пособие по дисциплине «Автоматное программирование». – СПб: СПбГУ ИТМО, 2007. – Режим доступа: http://is.ifmo.ru/books/_umk.pdf
23. Koza J. R. Future Work and Practical Applications of Genetic Programming. Handbook of Evolutionary Computation. – Bristol: IOP Publishing Ltd. 1997.
24. Поликарпова Н.И., Точилин В.Н., Шалыто А.А. Применение генетического программирования для реализации систем со сложным поведением // Сборник трудов IV-ой Международной научно-практической конференции «Интегрированные модели и мягкие вычисления в искусственном интеллекте». – Том 2. – М.: Физматлит, 2007. – С. 598–604. – Режим доступа: http://is.ifmo.ru/genalg/_polikarpova.pdf
25. Сайт X-Plane by Laminar Research – Режим доступа: http://www.x-plane.com/
26. Koza J.R. Genetic Programming: On the Programming of Computers by Means of Natural Selection. – The MIT Press. 1992.
27. Халл Э., Джексон К., Дик Д. Разработка и управление требованиями. Практическое руководство пользователя. – 2005.
28. Халл Э., Джексон К., Дик Д. Разработка и управление требованиями. – Режим доступа: http://download.telelogic.com/download/article/eBook_RU_Requirements_Engineering.pdf
29. Поликарпова Н.И., Точилин В.Н., Шалыто А.А. Разработка библиотеки для генерации управляющих автоматов методом генетического программирования // Сборник докладов X Международной конференции по мягким вычислениям и измерениям. – Том 2. – СПбГЭТУ «ЛЭТИ». – 2007. – С. 84–87. – Режим доступа: http://is.ifmo.ru/download/polikarpova(LETI).pdf

УДК 004.4’242

СОВМЕСТНОЕ ПРИМЕНЕНИЕ ГЕНЕТИЧЕСКОГО ПРОГРАММИРОВАНИЯ, КОНЕЧНЫХ АВТОМАТОВ И ИСКУССТВЕННЫХ НЕЙРОННЫХ СЕТЕЙ ДЛЯ ПОСТРОЕНИЕЯ СИСТЕМЫ УПРАВЛЕНИЯ БЕСПИЛОТНЫМ ЛЕТАТЕЛЬНЫМ АППАРАТОМ

Ф.Н. Царев

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В работе рассматривается применение генетических алгоритмов для построения системы управления беспилотным летательным аппаратом. Система управления строится как совокупность искусственной нейронной сети и конечного автомата. Нейронная сеть преобразует входные вещественные переменные в логические, которые подаются на вход конечному автомату. Он, в свою очередь, вырабатывает выходные воздействия. Для оптимизации этой модели используется генетическое программирование.

Ключевые слова: генетическое программирование, нейронные сети, конечный автомат, беспилотный летательный аппарат, автоматное программирование

Введение

В последнее время все чаще применяется автоматное программирование [1], в рамках которого поведение программ описывается с помощью конечных детерминированных автоматов. В ряде задач автомат удается построить эвристическими методами,

однако часто такое построение требует больших затрат времени или вообще невозможно. Примером такой задачи является управление командой беспилотных летательных аппаратов [2, 3] в соревнованиях с другой командой. Полный перебор крайне трудоемок, а эвристическое построение не всегда дает приемлемые результаты. Поэтому для построения автоматов в задачах такого рода целесообразно применять генетические алгоритмы и генетическое программирование [4–10].

С другой стороны, большая часть известных автору методов построения автоматов с помощью генетических алгоритмов [11–13] пригодна только для случая логических входных переменных. Чтобы применять такой алгоритм, необходимо разработать способ перехода от произвольных входных переменных к логическим входным переменным (или, хотя бы, к переменным, множество значений которых конечно и содержит небольшое число элементов).

Одним из вариантов решения этой задачи является введение соответствующих переменных. Например, если исходно были две вещественные переменные x и y, то, например, можно ввести две новые логические переменные A равную $x > 100$ и B равную $y < 200$.

Второй вариант решения состоит в том, чтобы разбить множество значений входных переменных на несколько областей и использовать в качестве значения входной переменной номер области, в которой лежат текущие значения входных переменных. Таким образом, перед тем, как подавать данные на вход автомату, необходимо определять, в какой из областей лежит набор текущих значений входных переменных. Это – *задача классификации*. Если для ее решения применять автоматический классификатор (нейронная сеть, дерево принятия решений, и т.д.), то возникает идея настраивать этот классификатор совместно с автоматом, с которым он связан.

В настоящей работе применяется второй подход. В качестве классификатора используется нейронная сеть. Ее настройка и построение автомата производится совместно с помощью генетического программирования.

Целью настоящей работы является построение с помощью генетического программирования системы управления беспилотным летательным аппаратом, состоящей из конечного автомата и нейронной сети.

Постановка задачи

Проводится соревнование [2, 3] между двумя командами беспилотных летательных аппаратов. Цель соревнований состоит в том, чтобы один из летательных аппаратов команды переместился на максимальное расстояние от линии старта. Состязание проходит на трассе, представляющей собой полубесконечную (бесконечную в одну сторону) полосу шириной 40 метров. Маневры, связанные с изменением высоты полета, не допускаются (таким образом, трасса соревнования двумерна).

Каждая команда состоит из N летательных аппаратов. В дальнейшем, кроме термина «соревнование», будем использовать термин «гонка». В начале гонки аппараты первой команды располагаются в воздухе случайным образом на некотором расстоянии от линии старта в левой половине трассы, которая на экране расположена горизонтально. Вторая команда размещается симметрично первой на правой половине трассы. Для каждого аппарата задана начальная скорость и начальное направление движения. В простейшем случае начальные скорости всех аппаратов одинаковы, а направления – строго вперед. Летательные аппараты в процессе полета могут поворачивать. Каждый летательный аппарат имеет определенный запас топлива, расходуемого в процессе движения. По команде «Старт» все аппараты начинают движение с целью максимально удалиться от линии старта. Они в процессе полета могут изменять скорость своего движения за счет изменения расхода топлива. Беспи-

лотные летательные аппараты, покинувшие трассу, считаются прекратившими гонку. Выходом за пределы коридора считается пересечение центром аппарата границы трассы.

Управление каждой командой выполняет программа, написанная на языке программирования *Java*.

Правила соревнований

В каждом соревновании каждая из команд на старте имеет N беспилотных летательных аппаратов с полным запасом топлива. Исходно аппараты первой команды случайным образом располагаются на первых 25 метрах левой половины трассы. Летательные аппараты второй команды располагаются симметрично им в правой половине трассы (рис. 1).

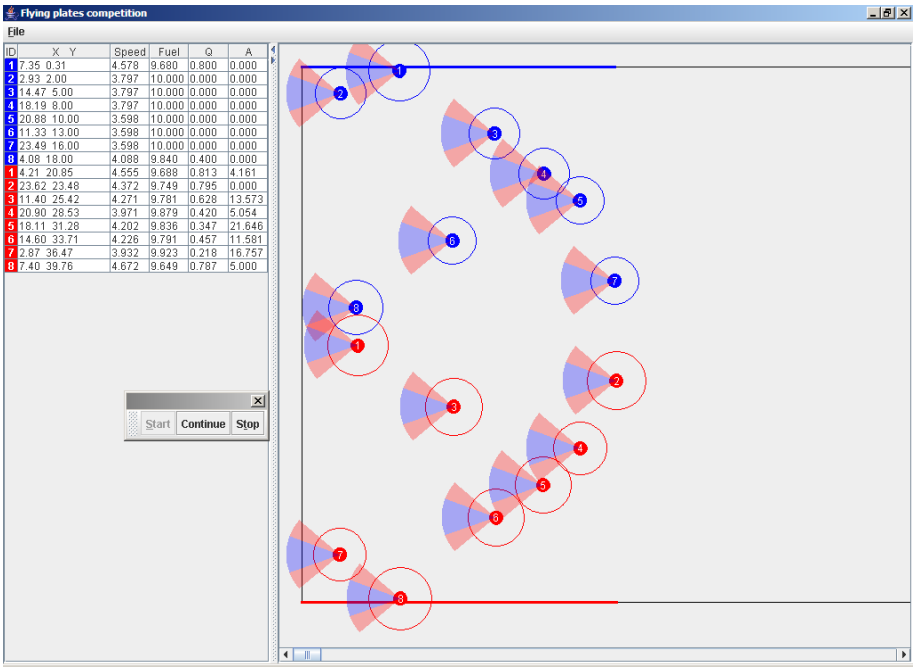


Рис. 1. Беспилотные летательные аппараты на старте

Жизненный цикл беспилотного летательного аппарата в рассматриваемой задаче может быть описан графом переходов автомата с тремя состояниями: «Полет», «Нормальное завершение гонки», «Аварийное завершение гонки» (рис. 2). Обозначения, используемые на рис. 2, приведены в табл. 1.

Таблица 1. Используемые обозначения

Обозначение	Описание
v1	Летательный аппарат покинул пределы трассы (его центр пересек границу трассы)
v2	Скорость летательного аппарата стала меньше, чем один м/с
v3	Летательный аппарат столкнулся с другим летательным аппаратом
v_rel	Относительная скорость столкновения летательных аппаратов
fuel	Количество топлива, которое осталось у летательного аппарата

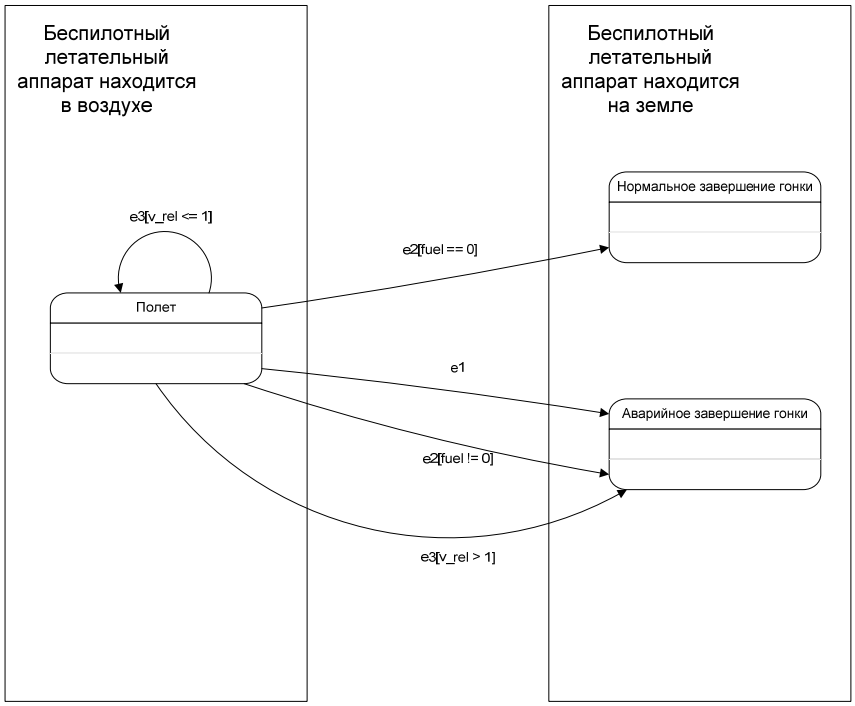


Рис. 2. Возможные состояния беспилотного летательного аппарата и переходы между ними

Поясним поведение беспилотного летательного аппарата. В начале гонки он находится в воздухе, исправен и способен продолжать участие в гонке. Этому соответствует состояние «Полет». При выходе летательного аппарата за пределы трассы (событие *e1*) он завершает гонку аварийно. Если скорость летательного аппарата падает ниже 1 м/с (событие *e2*), и его топливный бак не пуст (условие *fuel != 0*), то он завершает гонку аварийно. Если же при падении скорости ниже 1 м/с (событие *e2*) топливный бак летательного аппарата пуст (условие *fuel == 0*), то он нормально завершает гонку.

Если летательный аппарат сталкивается с другим аппаратом (событие *e3*), то при относительной скорости столкновения, большей 1 м/с (условие *v_rel > 1*), он аварийно завершает гонку. При относительной скорости столкновения, не превышающей 1 м/с (условие *v_rel <= 1*), беспилотный летательный аппарат продолжает полет. Заметим, что поскольку начальный запас топлива у каждой летательного аппарата конечен, то рано или поздно все беспилотные летательные аппараты обеих команд завершат гонку.

При подведении итогов гонки учитываются только результаты летательных аппаратов, нормально ее завершивших. Результатом команды считается наибольшее из расстояний, на которое удалились от линии старта ее летательные аппараты, которые нормально завершили гонку. Если все летательные аппараты команды вышли из гонки аварийно, результат команды считается равным нулю. Победителем признается команда, прошедшая наибольшее расстояние. В случае равенства результатов гонка считается завершившейся вничью.

Динамика беспилотного летательного аппарата

Беспилотный летательный аппарат представляет собой дискообразное «летающее крыло» радиусом 1 м. На рис. 3 представлен вид сверху описываемого летательного аппарата.

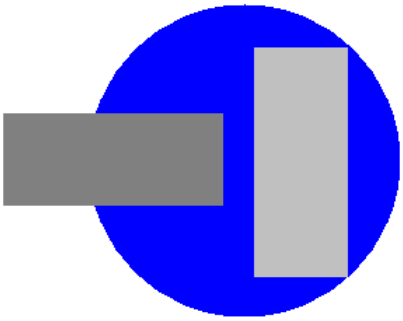


Рис. 3. Беспилотный летательный аппарат

Беспилотный летательный аппарат имеет реактивный двигатель (горизонтальный прямоугольник на рис. 3), топливный бак (вертикальный прямоугольник), аэродинамические рули и бортовой компьютер, способный регулировать расход топлива (и, как следствие, тягу двигателя) и положение аэродинамических рулей. Рули позволяют летательному аппарату маневрировать. Летательный аппарат может передвигаться со скоростями от 1 м/с. Его максимальная скорость зависит от запаса топлива и сопротивления воздуха. Ограничение в 1 м/с вызвано тем, что летательный аппарат с меньшей скоростью не может держаться в воздухе.

Беспилотный летательный аппарат движется в соответствии со вторым законом Ньютона. Его движение определяется двумя силами: сопротивлением воздуха F и тягой двигателя T. Если тяга не равна сопротивлению воздуха, то летальный аппарат движется с ускорением, которое может быть положительным (если тяга больше сопротивления воздуха) или отрицательным (если сопротивление воздуха больше тяги). Ускорение определяется по формуле (1):

$$a = \frac{T - F}{m}, \quad (1)$$

где m – масса беспилотного летательного аппарата. При этом считается, что изменение его массы за счет выгорания горючего пренебрежимо мало. Сопротивление воздуха определяется по формуле (2):

$$F = c_1 + c_2 v^2, \quad (2)$$

где v – скорость беспилотного летательного аппарата, а коэффициенты *c1* и *c2* определяются его аэродинамическими характеристиками и одинаковы для всех аппаратов обеих команд. Тяга двигателя определяется по формуле (3):

$$T = c_4 q, \quad (3)$$

где q – расход топлива [см^3/с]. Расход топлива находится под контролем бортового компьютера летательного аппарата, что позволяет изменять расход от нуля до единицы. Константа c_4 определяется характеристиками двигателя беспилотного летательного аппарата и одинакова для всех аппаратов обеих команд.

Аэродинамические рули позволяют летательному аппарату поворачивать относительно его текущего направления движения на угол, не превышающий *25°*.

Аэродинамическое взаимодействие между летательными аппаратами

При полете беспилотного летательного аппарата от траектории его полета в направлениях назад и в стороны под углом около 30° распространяются конические вихревые потоки воздуха. Если другой аппарат попадет в этот вихрь, то сопротивление воздуха его полету резко *снизится* (рис. 4). Отметим, что летательный аппарат, находящийся за хвостом

(два сектора по 20°) другого беспилотного летательного аппарата, испытывает *дополнительное сопротивление* движению, обусловленное реактивной струей.

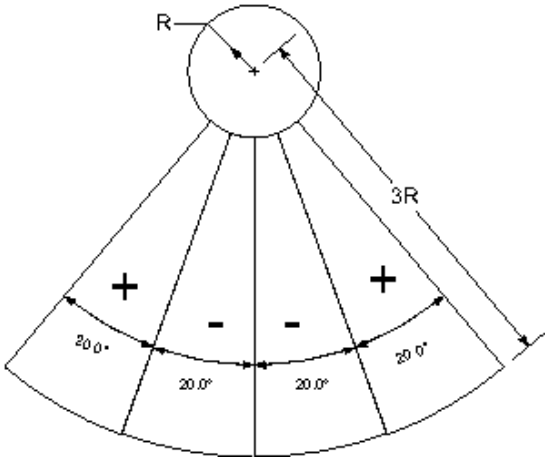


Рис. 4. Зоны повышенного и пониженного сопротивления воздуха

Поясним, как учитывается изменение сопротивления воздуха. Если центр второго летательного аппарата находится в областях, отмеченных на рис. 4 знаком «+», сопротивление воздуха его движению падает на 50%. Если же центр второй летательного аппарата находится в области, помеченной знаком «–», сопротивление воздуха возрастает на 50%. Аэродинамические воздействия от нескольких летательных аппаратов складываются так, что в зоне, отмеченной на рис. 5 знаками «++», сопротивление воздуха вообще отсутствует, а в зонах, помеченных знаком «0», воздействия компенсируют друг друга. При этом в результате наложения зон воздействия от трех и более летательных аппаратов сопротивление воздуха не может стать отрицательным.

Учитывая изложенное, вычисление сопротивления воздуха происходит следующим образом. Пусть *N*+ – число летательных аппаратов, уменьшающих сопротивление воздуха в этой области, а *N*- – число аппаратов, увеличивающих сопротивление воздуха. Пусть $\Delta N = N+ - N-$. Если $\Delta N = 0$, то в этой области нормальное аэродинамическое сопротивление, если $\Delta N = 1$ или $\Delta N = 2$, то сопротивление понижается на $50\Delta N$ процентов. Если ΔN отрицательно, то сопротивление в этой области повышается на $50|\Delta N|$ процентов.

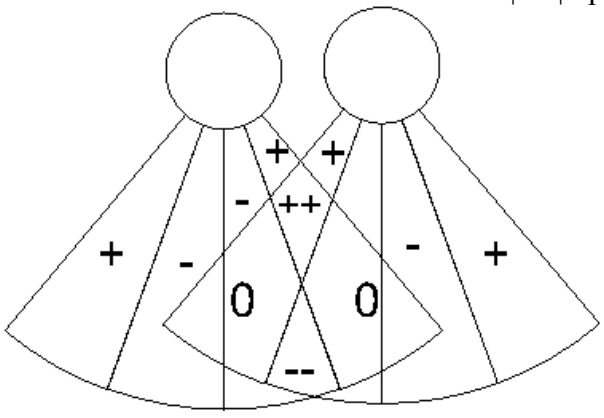


Рис. 5. Наложение областей аэродинамического взаимодействия двух летательных аппаратов

Столкновение беспилотных летательных аппаратов

При столкновении двух беспилотных летательных аппаратов происходит их абсолютно упругое соударение без передачи вращательного момента. Если относительная скорость столкновения была более 1 м/с, то оба участвовавшие в столкновении беспилотных летательных аппарата повреждаются и аварийно завершают гонку. В рамках рассматриваемой модели поврежденные летательные аппараты не взаимодействуют между собой и аппаратами, продолжающими полет.

Под *относительной скоростью столкновения* понимается величина проекции векторной разности скоростей летательных аппаратов на прямую, проходящую через центры аппаратов в момент столкновения (рис. 6). На этом рисунке вектор *Vrel* соответствует относительной скорости.

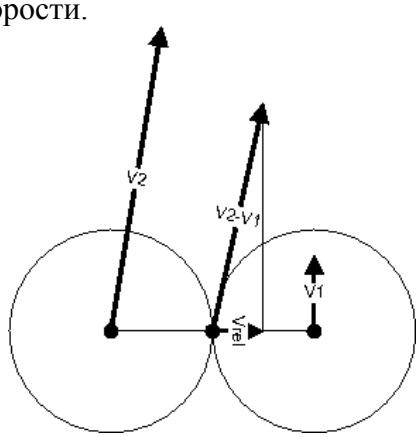


Рис. 6. Относительная скорость столкновения двух летательных аппаратов

Моделирование гонки

Моделирование гонки происходит по ходам, каждый из которых занимает *t* миллисекунд. В начале каждого хода игроки обладают информацией о координатах и скоростях всех беспилотных летательных аппаратов. Каждому игроку предоставляется возможность установить расход топлива и угол поворота аэродинамических рулей каждого летательного аппарата своей команды. Каждые *t* миллисекунд (один ход) происходит обновление параметров. Покажем, как выполняется моделирование полета беспилотных летательных аппаратов за время одного хода.

Во-первых, производится снятие с соревнования летательных аппаратов, движущихся со скоростью, меньшей 1 м/с. При завершении полета летательными аппаратами с пустыми баками пройденные ими расстояния засчитываются в результат команды.

Во-вторых, производится расчет ускорений летательных аппаратов в соответствии с установленными расходами топлива и углами поворотов, а также аэродинамическим сопротивлением.

В-третьих, производится расчет новых скоростей летательных аппаратов по формуле (4):

$$\vec{V}_{temp} = \vec{V}_{old} + \vec{a} \cdot \Delta t, \quad (4)$$

где V_{temp} – вектор скорости летательного аппарата после учета ускорения, V_{old} – вектор старой скорости летательного аппарата, a – вектор ускорения. После этого происходит поворот вектора скорости на угол, равный углу поворота аэродинамических рулей (рис. 7). В результате поворота получается вектор новой скорости беспилотного летательного аппарата V_{new}.

В-четвертых, производится снятие с соревнования беспилотных летательных аппаратов, движущихся медленнее 1 м/с. Как и ранее, при завершении полета летательными аппаратами с пустыми баками пройденные ими расстояния засчитываются в результат команды.

В-пятых, происходит равномерное прямолинейное движение летательных аппаратов (считается, что за время шага моделирования их скорости не меняются). Если при этом происходит соударение аппаратов, т.е. расстояние между центрами каких-либо двух аппаратов становится меньше 2 м, то их скорости и координаты изменяются в соответствии с законами сохранения импульса и энергии. При этом летательные аппараты, относительная скорость столкновения которых превосходила 1 м/с, выбывают из гонки.

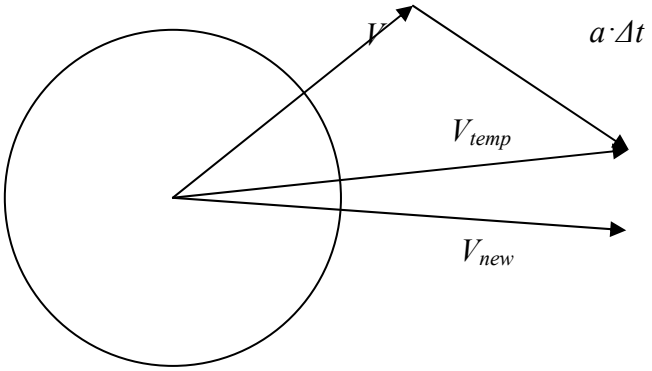


Рис. 7. Пересчет скорости летательного аппарата на шаге моделирования

В-шестых, производится проверка того, что все беспилотные летательные аппараты находятся в пределах трассы. При выходе центра аппарата за пределы трассы он выбывает из гонки.

Гонка продолжается до тех пор, пока ее не завершил хотя бы один летательный аппарат. После того, как ее закончит и этот аппарат, гонка завершается.

Структура системы управления беспилотным летательным аппаратом

Каждый беспилотный летательный аппарат управляется системой, состоящей из нейронной сети и конечного автомата. Таким образом, можно говорить, что используется *мультиагентный подход* [4] – каждый летательный аппарат представляет собой агента, взаимодействующего с внешней средой и другими агентами. При этом, как отмечалось выше, нейронная сеть используется для классификации значений вещественных входных переменных и выработки входных логических переменных для автомата, а автомат – для выработки выходных воздействий на беспилотный летательный аппарат (рис. 8).

Структура нейронной сети и способ ее взаимодействия с конечным автоматом показаны на рис. 9. Символами *S* на рис. 9 обозначены нейроны с сигмоидальной функцией активации, символом *L* – нейроны с пороговой функцией активации. Рядом с нейронами указаны их номера (они используются при описании операции скрещивания нейронных сетей). На каждый из трех выходов нейронной сети поступает число равное нулю или единице. Таким образом, существует восемь вариантов комбинаций выходных сигналов нейронной сети (000, 001, 010, 011, 100, 101, 110, 111), подаваемых на вход конечного автомата.

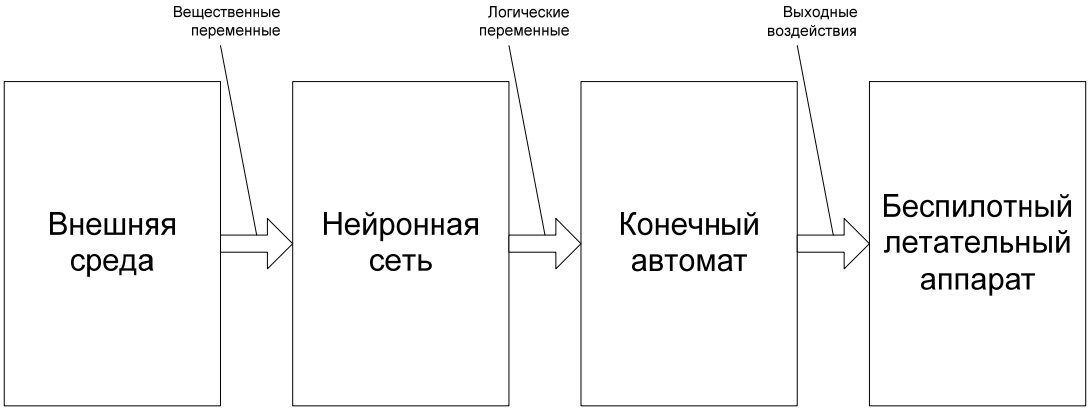


Рис. 8. Структурная схема системы управления беспилотным летательным аппаратом

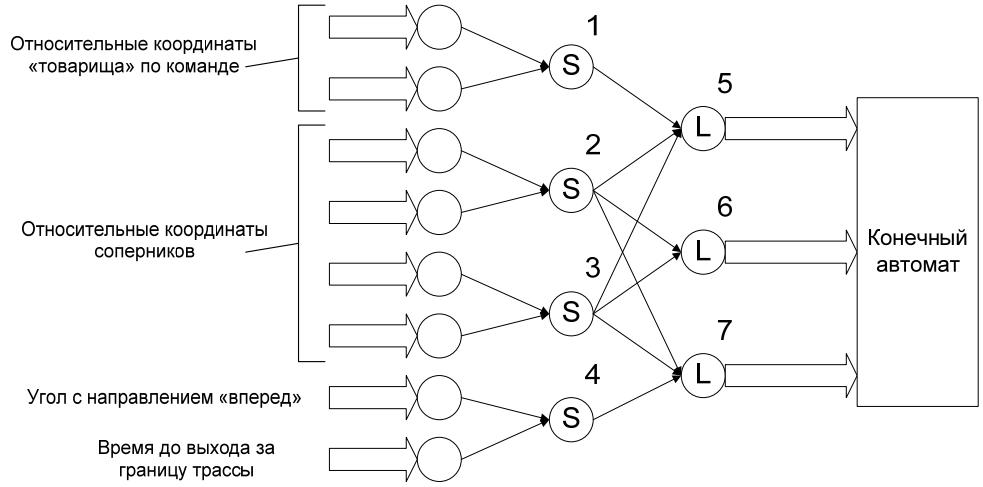


Рис. 9. Нейронная сеть и ее взаимодействие с конечным автоматом

Алгоритм генетического программирования для построения системы управления беспилотным летательным аппаратом

В настоящем разделе описан алгоритм генетического программирования, используемый для построения системы управления беспилотным летательным аппаратом. *Алгоритм генетического программирования* состоит из пяти частей:

- создание начального поколения;
- мутация;
- скрещивание (кроссовер);
- отбор особей для формирования следующего поколения;
- вычисление функции приспособленности (фитнес-функции).

Структура особи в используемом алгоритме

В связи с тем, что генетическое программирование эффективно только в случае небольшого размера особи, а число летательных аппаратов в каждой команде может быть достаточно велико, возникла идея совместно строить управляющие системы только для двух аппаратов. При этом предполагается, что беспилотный летательный аппарат может достаточно «хорошо» управляться даже при наличии небольшого объема информации о внешней среде. Поэтому была выбрана структура нейронной сети, ука-

занная в предыдущем разделе. При этом отметим, что два летательных аппарата были выбраны только для генерации систем управления с помощью генетического программирования, а в дальнейшем построенные системы управления будут «размножены» в количестве, необходимом для формирования команды.

Таким образом, особь в описываемом алгоритме генетического программирования состоит из двух систем управления беспилотным летательным аппаратом (рис. 10).

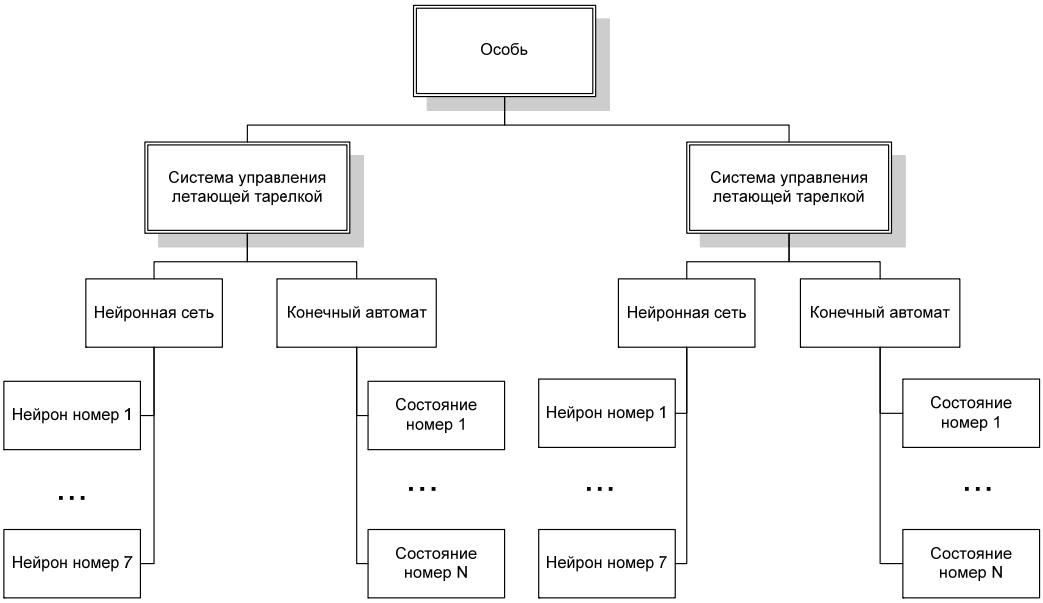


Рис. 10. Структура особи

Каждая система управления беспилотным летательным аппаратом состоит из нейронной сети и конечного автомата. Нейронная сеть состоит из четырех нейронов с сигмоидальной функцией активации и трех нейронов с пороговой функцией активации. Каждый из нейронов характеризуется порогом активации и весами связей, которые соединяют другие элементы сети с рассматриваемым. На языке программирования *Java* нейрон представляется следующим образом:

```
public abstract class Neuron {
  protected Neuron[] inputs;
  protected int inputsCnt;
  protected double[] w;
}
```

Описание конечного автомата состоит из номера начального состояния и описания состояний. Описание состояния состоит из описаний восьми переходов, соответствующих восьми вариантам значений входных переменных автомата. Описание каждого перехода состоит из номера состояния, в которое ведет этот переход, и двух действий, которые выполняются при выборе этого перехода – изменения расхода топлива и изменения угла поворота аэродинамических рулей. Каждое из этих действий характеризуется одним вещественным числом – соответственно, новым расходом топлива и углом поворота аэродинамических рулей:

```
public class Individual {
  protected PlateControlSystem[] pcs;
}
public class PlateControlSystem {
  private NeuralNet neuralNet;
  private Automaton automaton;
}
```

Создание начального поколения

Начальное поколение заполняется случайно сгенерированными системами управления. При этом в каждой системе управления случайным образом генерируется конечный автомат и нейронная сеть – веса связей в ней инициализируются случайными числами от -1 до 1.

Операция мутации

Мутация особи. При мутации особи с равной вероятностью мутирует либо одна система управления летательным аппаратом, либо вторая.

Мутация системы управления беспилотным летательным аппаратом. При мутации системы управления беспилотным летательным аппаратом мутирует либо нейронная сеть, либо конечный автомат

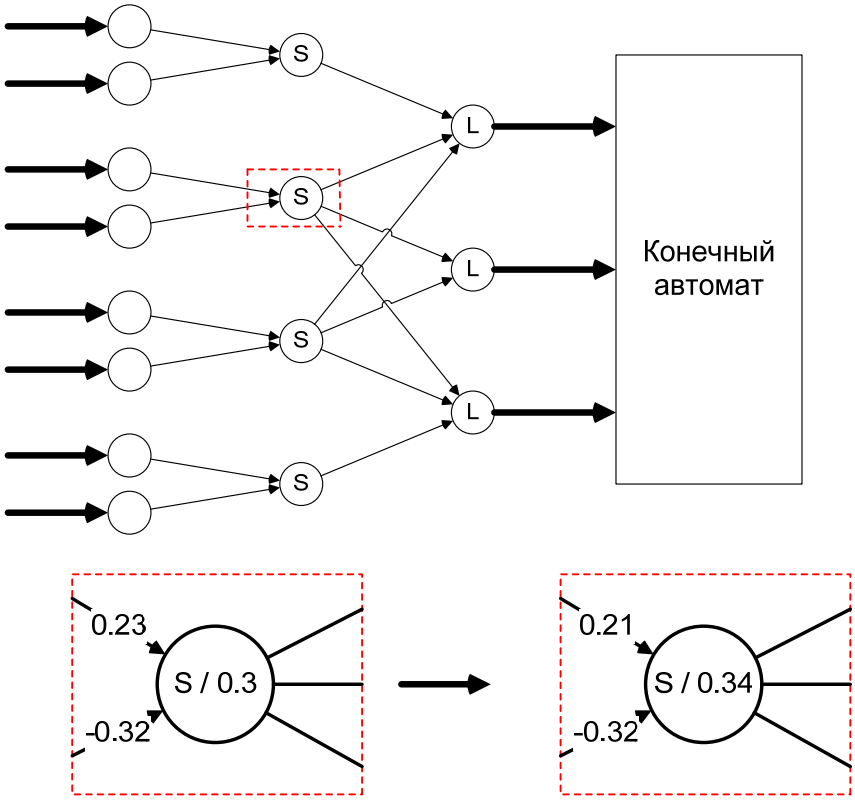


Рис. 11. Мутация нейронной сети

Мутация нейронной сети. При мутации нейронной сети случайно и равновероятно выбирается один элемент (искусственный нейрон) сети и мутирует. Мутация нейронной сети проиллюстрирована на рис. 11.

Мутация элемента сети. При мутации элемента сети случайно выбирается один из весов связей, и к нему прибавляется случайное число из отрезка [-0.05; 0.05]. Кроме этого, с вероятностью 0.5 аналогичная операция производится с лимитом активации нейрона.

Мутация конечного автомата. При мутации конечного автомата равной вероятностью производится либо изменение начального состояния, либо мутация случайно выбранного перехода.

Изменение начального состояния. Начальное состояние изменяется на случайно выбранное состояние автомата.

Мутация перехода. При мутации перехода с равной вероятностью происходит либо изменение номера состояния, в которое ведет переход, либо мутация одного из действий, связанных с переходом – может мутировать действие, связанное с изменением расхода топлива (прибавляется случайное число от -0.05 до 0.05) или с изменением угла поворота аэродинамических рулей (уменьшается или увеличивается на 5°).

Операция скрещивания

Оператор скрещивания получает на вход две особи (*P1*, *P2*) и выдает две особи (*S1*, *S2*). Пусть *X* – некоторая особь. Пусть *X.s1* и *X.s2* – системы управления беспилотными летательными аппаратами, входящие в эту особь. Пусть *s* – некоторая система управления аппаратом. Обозначим как *s.ns* входящую в нее нейронную сеть, а как *s.a* – входящий в нее автомат.

Скрещивание особей. При скрещивании особей происходит скрещивание систем управления летательными аппаратами: *P1.s1* и *P2.s1*, *P1.s2* и *P2.s2*. Обозначим системы, получившиеся в результате первого скрещивания, *s11* и *s12*, а в результате второго –*s21* и *s22*. Тогда для особей-потомков будет справедливо: *S1.s1 = s11*, *S1.s2 = s21*, *S2.s1 = s21*, *S2.s2 = s22*.

Скрещивание систем управления беспилотными летательными аппаратами. При скрещивании систем управления беспилотными летательными аппаратом *s1* и аппаратом *s2* происходит скрещивание автоматов *s1.a* и *s2.a* и скрещивание нейронных сетей *s1.ns* и *s2.ns*. Обозначим получающиеся в результате описанных скрещиваний автоматы *a1* и *a2*, а нейронные сети – *ns1* и *ns2*. В результате скрещивания системы управления летательными аппаратами получаются системы управления *s3* и *s4*, содержащие следующие элементы: *s3* содержит *a1* и *ns1*, *s4 – a2* и *ns2*.

Скрещивание автоматов. Обозначим автоматы, поступающие на вход оператора скрещивания автоматов, *A1* и *A2*. Начальное состояние некоторого автомата *A* обозначим *A.is*, а переход из состояния *i* по значению входной переменной *j* как *A(i, j)*. Обозначим автоматы, получающиеся в результате скрещивания, *A3* и *A4*. Для их начальных состояний справедливо:

- либо *A3.is = A1.is* и *A4.is = A2.is*;
- либо *A3.is = A2.is* и *A4.is = A1.is*.

Опишем переходы автоматов *A3* и *A4*. Скрещивание производится отдельно для каждого состояния *i* и для каждого значения *j* входной переменной. В каждом случае возможно два равновероятных варианта:

- *A3(i, j) = A1(i, j)* и *A4(i, j) = A2(i, j)*;
- *A3(i, j) = A2(i, j)* и *A4(i, j) = A1(i, j)*.

Проиллюстрируем скрещивание автоматов на примере случая одной входной переменной. Обозначим переход из состояния номер *i* в автомате *A1* по значению входной переменной «1» как *A1(i, 1)*, а по значению «0» как *A1(i, 0)*. Аналогичный смысл придадим обозначениям *A2(i, 0)* и *A2(i, 1)*. Тогда для переходов из состояния с номером *i* в автоматах-потомках *A3* и *A4* будет справедливо одно из четырех соотношений:

- *либо* A3(i, 0) = A1(i, 0), A4(i, 1) = A2(i, 1) *и* A4(i, 0) = A2(i, 0), A4(i, 1) = A1(i, 1);
- *либо* A3(i, 0) = A2(i, 0), A4(i, 1) = A1(i, 1) *и* A4(i, 0) = A1(i, 0), A4(i, 1) = A2(i, 1);
- *либо* A3(i, 0) = A1(i, 0), A4(i, 1) = A1(i, 1) *и* A4(i, 0) = A2(i, 0), A4(i, 1) = A2(i, 1);
- *либо* A3(i, 0) = A2(i, 0), A4(i, 1) = A2(i, 1) *и* A4(i, 0) = A1(i, 0), A4(i, 1) = A1(i, 1).

Все четыре варианта соотношений равновероятны. Возможные варианты переходов изображены на рис. 12. В левой части этого рисунка показаны переходы из состояния номер *i* автоматов, поступающих на вход операции скрещивания, а в правой части – четыре возможных варианта переходов автоматов, которые будут получены в резуль-

тате ее применения. Переходы в левой части рисунка пронумерованы числами от одного до четырех. Переходы в правой части также пронумерованы, причем нумерация переходов соответствует нумерации в левой части рисунка.

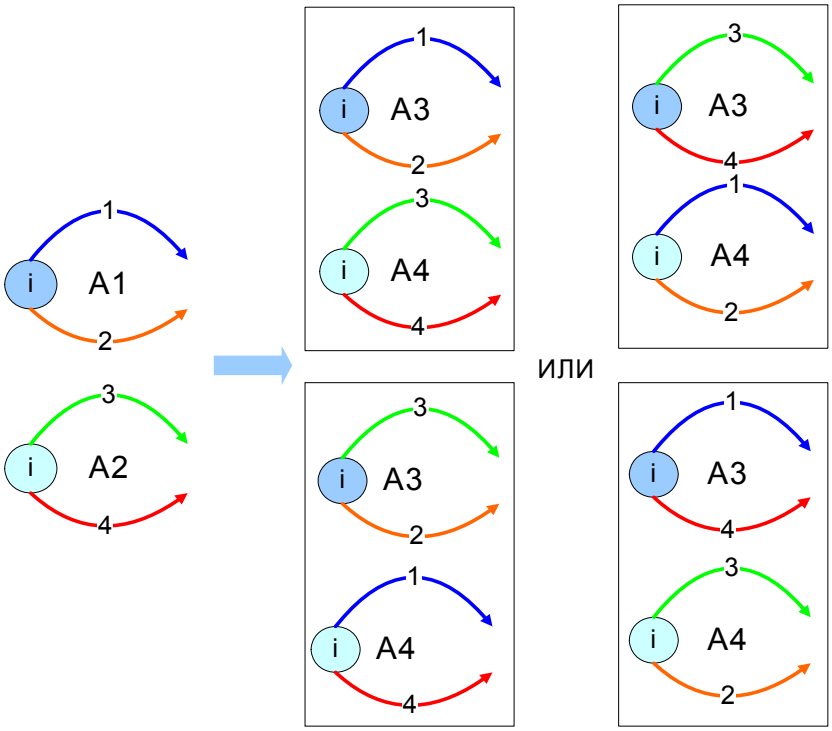


Рис. 12. Варианты переходов при скрещивании

Скрещивание нейронных сетей. Обозначим нейронные сети, поступающие на вход оператора скрещивания нейронных сетей, *NS1* и *NS2*, а получающиеся в результате его применения –*NS3* и *NS4*. Опишем их устройство. Обозначим *NS(i)* нейрон с номером *i* сети *NS* (рис. 9). Для нейронов с номерами *NS3(i)* и *NS4(i)* возможны два варианта:

- *NS3(i)* = *NS1(i)* и *NS4(i)* = *NS2(i)*;
- *NS3(i)* = *NS2(i)* и *NS4(i)* = *NS1(i)*.

Скрещивание нейронных сетей проиллюстрировано на рис. 13.

Формирование следующего поколения

В качестве основной стратегии формирования следующего поколения используется элитизм. При обработке текущего поколения отбрасываются все особи, кроме нескольких наиболее приспособленных. Доля выживающих особей постоянна для каждого поколения и является одним из параметров алгоритма. Эти особи переходят в следующее поколение. После этого оно дополняется до требуемого размера следующим образом: пока оно не заполнено, выбираются две особи из текущего поколения, и они с некоторой вероятностью скрещиваются или мутируют. Обе особи, полученные в результате мутации или скрещивания, добавляются в новое поколение.

Кроме этого, если на протяжении достаточно большого числа поколений не происходит увеличения приспособленности, то применяются «малая» и «большая» мутации поколения. При «малой» мутации поколения ко всем особям, кроме 10% лучших, применяется оператор мутации. При «большой» мутации каждая особь либо мутирует, либо заменяется на случайно сгенерированную.

Число поколений до «малой» и «большой» мутации постоянно во время работы алгоритма, но может быть различным для разных его запусков.

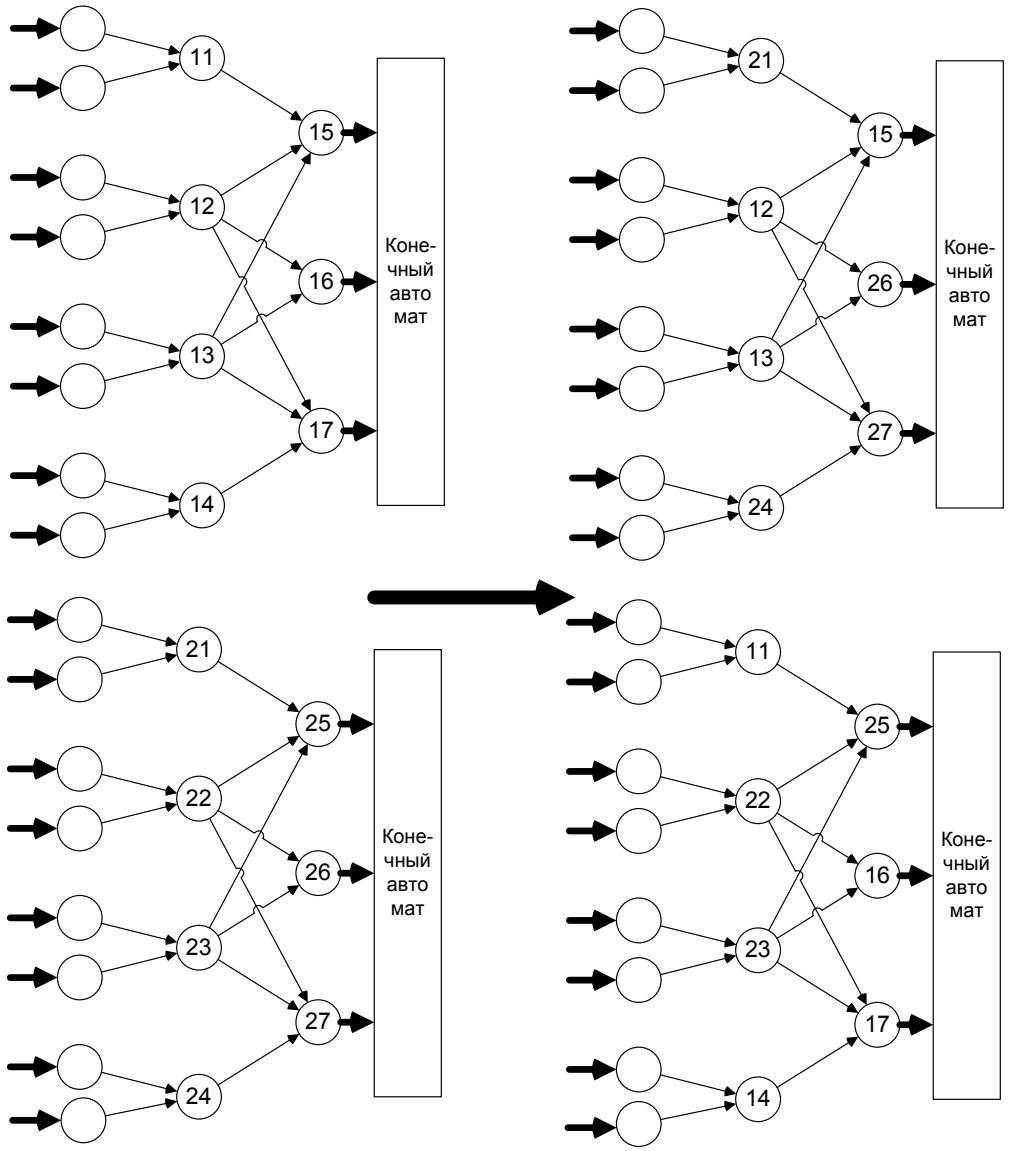


Рис. 13. Скрещивание нейронных сетей

Функция приспособленности

Функция приспособленности особи вычисляется в ходе соревнований команды, аппараты которой управляются описываемыми особью системами управления беспилотным летательным аппаратом, с некоторыми командами, управляемыми системами управления, реализующими выбранную стратегию поведения. В качестве таких систем в настоящей работе используются системы, реализующие «агрессивную» и «простую» стратегию [3]. Проводилось по пять соревнований с каждой из стратегий при следующих параметрах летательных аппаратов:

$c1 = 0.625;$ (5)

$c2 = 0.025;$ (6)

$c4 = 3.125;$ (7)

$\Delta t = 0.3;$ (8)

$L = 7.$ (9)

Результатом вычисления функции приспособленности является сумма результатов команды, аппараты которой управляются описываемыми особью системами управления беспилотным летательным аппаратом, во всех соревнованиях, к которой прибавлено число побед, деленное на число соревнований, увеличенное на единицу.

Результаты применения метода совместного применения нейронных сетей, генетического программирования и конечных автоматов

Системы управления беспилотным летательным аппаратом строились с помощью описанного алгоритма генетического программирования, а далее тестировались в среде, разработанной в работе [3]. Тестирование проводилось с помощью соревнования построенной системы с командой, аппараты которой управляются системой, также описанной в работе [3]. Соревнования проводились при числе летательных аппаратов в каждой команде, равном 8. Чтобы построенные с помощью генетического программирования системы управления беспилотным летательным аппаратом могли работать в этом случае, на первые два входа нейронной сети (рис. 9) подавались относительные координаты ближайшего летательного аппарата из «своей» команды, а на входы с третьего по шестой подавались координаты двух ближайших аппаратов из обеих команд. При этом летательные аппараты с нечетными номерами управлялись системой, построенной для первого аппарата, а с четными номерами – построенной для второго аппарата.

С помощью описанного алгоритма генетического программирования была построена особь, содержащая две системы управления беспилотными летательными аппаратами, каждая из которых содержит автомат с шестью состояниями. Их построение заняло около суток на компьютере с процессором *Intel Celeron 2.53 GHz*.

Таблица 2. Функция переходов и функция действий автомата, управляющего первым беспилотным летательным аппаратом

	000		001		010		011		100		101		110		111	
0	0		5		3		5		4		2		5		4	
	5	0,8	25	0,4	-5	0,8	15	0,8	5	0,4	-5	0,8	-10	0,4	25	0,8
1	3		2		3		5		4		2		0		5	
	-20	0,4	25	0,4	15	0,8	-15	0,4	5	0,4	20	0,4	-10	0,8	0	0,8
2	2		3		4		5		1		5		2		0	
	20	0,4	-5	0,8	15	0,8	10	0,4	20	0,8	10	0,8	5	0,4	-20	0,4
3	5		3		5		4		5		2		0		4	
	-20	0,4	-20	0,8	10	0,8	-10	0,4	-25	0,8	5	0,4	-5	0,8	-25	0,8
4	0		2		5		2		5		0		1		2	
	0	0,4	5	0,4	-10	0,8	-10	0,44	-10	0,8	10	0,8	-15	0,4	-5	0,8
5	2		1		2		3		1		4		2		2	
	-25	0,78	-15	0,4	-20	0,4	10	0,4	-20	0,8	-5	0,8	0	0,8	-20	0,4

Функция переходов и действий автомата, входящего в систему управления первым беспилотным летательным аппаратом, приведена в табл. 2. Строки этой таблицы

соответствуют состояниям автомата (пронумерованы числами от 0 до 5), столбцы – возможным комбинациям значений трех входных переменных. Серым цветом отмечена строка, соответствующая начальному состоянию. Ячейки имеют формат, показанный в табл. 3: в верхней строке указано новое состояние, а в нижней – выполняемое действие. В табл. 4 приведены веса нейронной сети, входящей в построенную систему управления первым беспилотным летательным аппаратом.

Таблица 3. Формат ячеек табл. 2

Новое состояние	
Изменение угла поворота	Новый расход топлива

Таблица 4. Веса связей и смещенные веса нейронов, входящих в нейронную сеть, управляющую первым беспилотным летательным аппаратом

Номер нейрона	Номера нейронов, соединенных с данным	Вес связи	«Смещенный вес»
1	Вход-1	*0,7820135561918911*	*0,0000823910892*
	Вход-2	*0,9843639098567366*	
2	Вход-3	*0,8621015893426324*	*0,0000342189215*
	Вход-4	*0,49004984673483654*	
3	Вход-5	*0,31478285017906643*	*0,0000238490014*
	Вход-6	*0,9111794712756234*	
4	Вход-7	*0,9383238994033571*	*0,0000341289013*
	Вход-8	*0,5244096391670159*	
5	1	*-0,9357005374202394*	*0,4971125943935544*
	2	*-0,4004385439577506*	
	3	*0,97490945048913*	
6	2	*0,9562669142389417*	*-0,568426590601512*
	3	*-0,2903272439699007*	
7	2	*0,1400255181142766*	*-0,859430022111886*
	3	*-0,315932752555524*	
	4	*-0,5112115605370626*	

Таблица 5. Функция переходов и функция действий автомата, управляющего вторым беспилотным летательным аппаратом

	000		001		010		011		100		101		110		111	
0	2		3		3		4		0		4		0		3	
	-15	0,4	5	0,4	10	0,4	-20	0,4	-5	0,4	15	0,4	15	0,4	-25	0,4
1	0		3		3		3		3		5		2		5	
	-25	0,8	-15	0,4	25	0,4	-10	0,4	-25	0,8	-5	0,4	-15	0,8	-10	0,4
2	4		0		0		0		2		0		3		2	
	-15	0,8	5	0,8	25	0,4	15	0,4	-25	0,4	-5	0,4	0	0,4	10	0,8
3	3		3		2		3		3		5		2		1	
	0	0,4	20	0,8	20	0,8	-25	0,8	-15	0,8	20	0,4	5	0,4	15	0,4
4	4		4		5		0		4		2		3		2	
	5	0,8	-20	0,4	25	0,8	25	0,4	0	0,38	-15	0,4	0	0,8	-15	0,36
5	3		4		0		0		4		4		4		2	
	-15	0,4	-20	0,4	0	0,8	10	0,4	-10	0,8	10	0,35	0	0,4	0	0,8

Таблица 6. Веса связей и смещенные веса нейронов, входящих в нейронную сеть, управляющую вторым беспилотным летательным аппаратом

Номер нейрона	Номера нейронов, соединенных с данным	Вес связи	«Смещенный вес»
1	Вход-1	*0,23234941730420006*	*0,0003223789*
	Вход-2	*0,07485700920671275*	
2	Вход-3	*0,3056025308692568*	*0,0030210123478*
	Вход-4	*0,30661997660895013*	
3	Вход-5	*0,17219120661378778*	*0,0000728973223*
	Вход-6	*0,03018455303178147*	
4	Вход-7	*0,23636134465917025*	*0,00430412789122*
	Вход-8	*0,806175323870498*	
5	1	*0,9180350210858039*	*-0,8415486736799431*
	2	*-0,4638322712743177*	
	3	*-0,5253829213469569*	
6	2	*-0,8334567897770782*	*0,05661834053390668*
	3	*-0,7931512598511974*	
7	2	*0,3096107059232607*	*0,2372057991691212*
	3	*0,13115163005946284*	
	4	*-1,0*	

В табл. 5 показаны значения функций переходов и действий для автомата, управляющего вторым беспилотным летательным аппаратом. Обозначения и формат ячеек в этой таблице такие же, как и в табл. 2. В табл. 6 приведены веса связей нейронной сети, входящей в построенную систему управления вторым беспилотным летательным аппаратом.

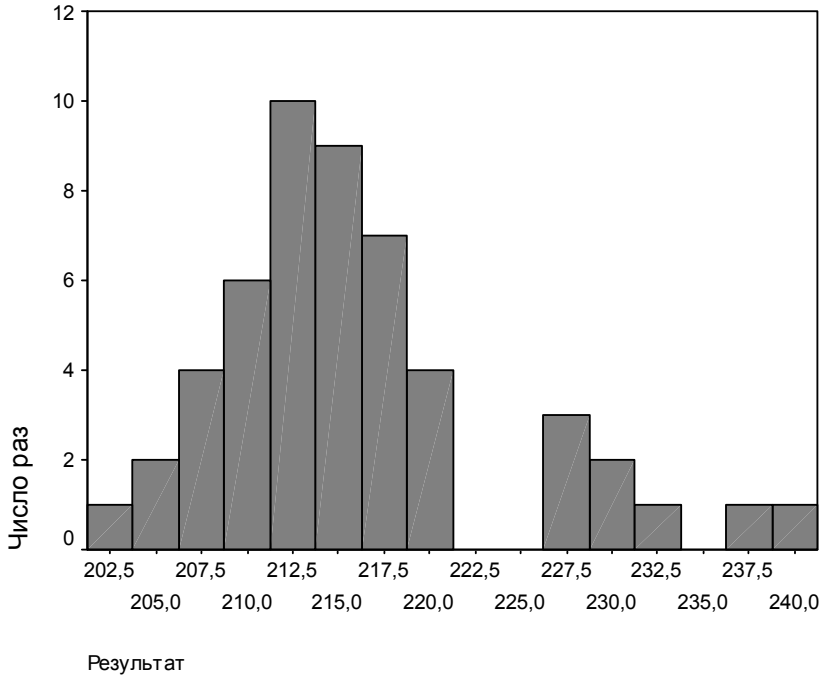


Рис. 14. Распределение результатов, обеспеченных системой, построенной при помощи генетического программирования

Приведем результат сравнения построенной с помощью генетического алгоритма системы управления с системой, построенной вручную в работе [3]. На рис. 14 показано распределение результатов, обеспеченных системой управления, построенной в настоящей работе с помощью генетического программирования. На рис. 15 показано распределение результатов, обеспеченных системой управления, построенной в работе [3].

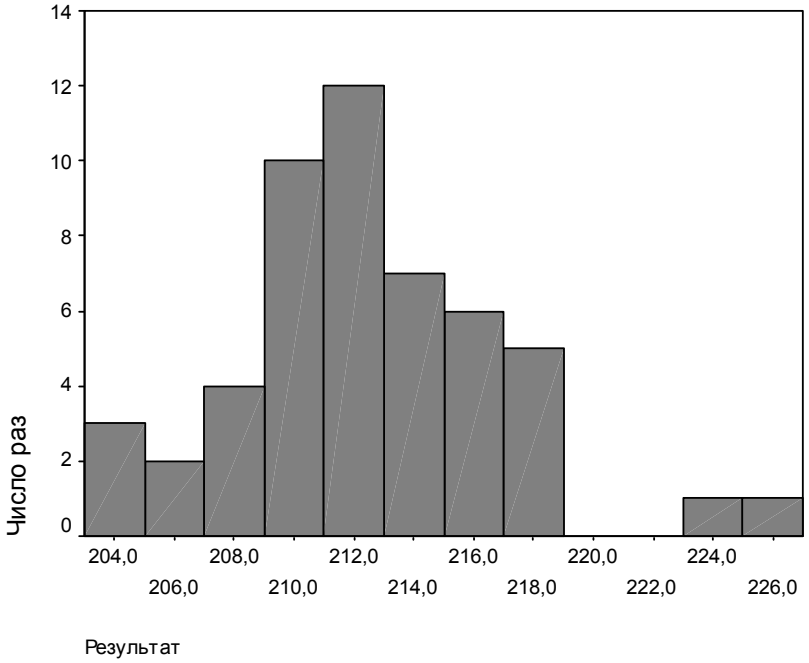


Рис. 15. Распределение результатов, обеспеченных системой, построенной в работе [3]

Выводы

В работе предложен алгоритм генетического программирования, осуществляющий построение системы управления беспилотным летательным аппаратом. Система управления строится как совокупность нейронной сети и конечного автомата. Нейронная сеть служит для преобразования вещественных входных переменных от внешней среды в логические переменные. Эти переменные подаются на вход конечного автомата, который вырабатывает выходные воздействия.

Проведено сравнение системы управления с системой автоматов, построенной в работе [3]. Результаты этого сравнения позволяют сделать вывод о том, что с помощью генетического программирования была построена система управления, превосходящая по результатам соревнований построенную вручную в работе [3]. Это доказывает эффективность используемой модели и метода ее оптимизации.

Литература

1. Шалыто А.А. Технология автоматного программирования // Труды первой Всероссийской научной конференции «Методы и средства обработки информации». – М.: МГУ. – Режим доступа: 2003. http://is.ifmo.ru/works/tech_aut_prog/
2. Заочный тур всесибирской олимпиады 2005 по информатике. – Режим доступа: http://olimpic.nsu.ru/widesiberia/archive/wso6/2005/rus/1tour/problem/problem.html
3. Паращенко Д.А., Царев Ф.Н., Шалыто А.А. Технология моделирования одного класса мультиагентных систем на основе автоматного программирования на примере иг-

ры «Соревнование летающих тарелок». Проектная документация. – СПбГУ ИТМО. 2006. – Режим доступа: – Режим доступа: http://is.ifmo.ru/unimod-projects/plates/
4. Рассел С., Норвиг П. Искусственный интеллект. Современный подход. – М.: Вильямс, 2006.
5. Jefferson D., Collins R., Cooper C., Dyer M., Flowers M., Korf R., Taylor C., Wang A. The Genesys System. 1992. – Режим доступа: www.cs.ucla.edu/~dyer/Papers/AlifeTracker/Alife91Jefferson.html
6. Angeline P. J., Pollack J. Evolutionary Module Acquisition // Proceedings of the Second Annual Conference on Evolutionary Programming. 1993. – Режим доступа: http://www.demo.cs.brandeis.edu/papers/ep93.pdf
7. Chambers L. Practical Handbook of Genetic Algorithms. Complex Coding Systems. Volume III. CRC Press, 1999.
8. Гладков Л.А., Курейчик В.В., Курейчик В.М. Генетические алгоритмы. – М.: Физматлит, 2006.
9. Koza J. R. Genetic programming: on the programming of computers by means of natural selection. – MIT Press, 1992.
10. Яминов Б. Генетические алгоритмы – Режим доступа: http://rain.ifmo.ru/cat/view.php/theory/unsorted/genetic-2005
11. Царев Ф. Н., Шалыто А. А. О построении автоматов с минимальным числом состояний для задачи об "умном муравье" // Сборник докладов X международной конференции по мягким вычислениям и измерениям. – СПбГЭТУ "ЛЭТИ". – Т.2. – 2007. – С. 88–91. http://is.ifmo.ru/download/ant_ga_min_number_of_state.pdf
12. Данилов В.Р. Технология генетического программирования для генерации автоматов управления системами со сложным поведением. Бакалаврская работа / СПбГУ ИТМО. – 2007.. – Режим доступа: http://is.ifmo.ru/papers/danilov_bachelor/
13. Поликарпова Н.И., Точилин В.Н. Применение генетического программирования для реализации систем со сложным поведением // Научно-технический вестник СПбГУ ИТМО. – 2007. – Выпуск 39. – С.276–293.

УДК 004.4’242

ПРИМЕНЕНИЕ ГЕНЕТИЧЕСКОГО ПРОГРАММИРОВАНИЯ И МЕТОДОВ СОКРАЩЕННЫХ ТАБЛИЦ ПЕРЕХОДОВ И ДЕРЕВЬЕВ РЕШЕНИЙ ДЛЯ ПОСТРОЕНИЯ АВТОМАТОВ УПРАВЛЕНИЯ МОДЕЛЬЮ БЕСПИЛОТНОГО ЛЕТАТЕЛЬНОГО АППАРАТА

А.А. Давыдов, Д.О. Соколов, Ф.Н. Царев

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В работе рассматривается применение генетического программирования для построения конечных автоматов, управляющих системами со сложным поведением. Для представления конечных автоматов используются два метода: метод сокращенных таблиц переходов и метод представления автоматов деревьями решений. Применение этих методов иллюстрируется на примере задачи об управлении моделью беспилотного летательного аппарата.

Ключевые слова: генетическое программирование, конечный автомат, беспилотный летательный аппарат, автоматное программирование

Введение

В последнее время все чаще применяется автоматное программирование [1], в рамках которого поведение программ описывается с помощью конечных детерминиро-

ванных автоматов. В ряде задач автомат удается построить эвристическими методами, однако часто такое построение требует больших затрат времени или вообще невозможно. Примером такой задачи является управление командой беспилотных летательных аппаратов [2, 3] в соревнованиях с другой командой. Полный перебор крайне трудоемок, а эвристическое построение не всегда дает приемлемые результаты. Поэтому для построения автоматов в задачах такого рода целесообразно применять генетические алгоритмы и генетическое программирование.

Целью настоящей работы является построение с помощью генетического программирования автоматов Мили для управления беспилотными летательными аппаратами.

Постановка задачи

Проводится соревнование [2, 3] между двумя командами беспилотных летательных аппаратов. Цель соревнований состоит в том, чтобы один из летательных аппаратов команды переместился на максимальное расстояние от линии старта. Состязание проходит на трассе, представляющей собой полубесконечную (бесконечную в одну сторону) полосу шириной 40 м. Маневры, связанные с изменением высоты полета, не допускаются (таким образом, трасса соревнования двухмерна).

Каждая команда состоит из N летательных аппаратов. В дальнейшем, кроме термина «соревнование», будем использовать термин «гонка». В начале гонки аппараты первой команды располагаются в воздухе случайным образом на некотором расстоянии от линии старта в левой половине трассы. Вторая команда размещается симметрично первой на правой половине трассы. Для каждого аппарата заданы начальная скорость и направление движения. В простейшем случае начальные скорости всех аппаратов одинаковы, а направления – строго вперед. Летательные аппараты в процессе полета могут поворачивать. Каждый летательный аппарат имеет определенный запас топлива, расходуемого в процессе движения. По команде «Старт» все аппараты начинают движение с целью максимально удалиться от линии старта. Они в процессе полета могут изменять скорость своего движения за счет изменения расхода топлива.

Беспилотные летательные аппараты, покинувшие трассу, считаются прекратившими гонку. Выходом за пределы коридора считается пересечение центром аппарата границы трассы. Аппарат также считается аварийно прекратившим соревнование еще в двух случаях: во-первых, если его скорость падает ниже определенной величины, а топливный бак не пуст; во-вторых, если при столкновении двух аппаратов их относительная скорость больше определенной величины; в противном случае происходит абсолютно упругое столкновение.

Динамика беспилотных летательных аппаратов подчиняется второму закону Ньютона и подробно описана в работе [3]. Для нас важно то, что на каждый летательный аппарат влияет расположение остальных аппаратов, замедляя (благодаря реактивной силе двигателя) или, наоборот, ускоряя его (благодаря уменьшению силы сопротивления воздуха). Это дает простор для разработки различных стратегий управления командой беспилотных летательных аппаратов, что и является решаемой задачей.

В работе [3] была предложена система управления беспилотными летательными аппаратами, основанная на мультиагентном подходе [4]. При таком подходе каждый аппарат рассматривается в отдельности от других, а искусственный интеллект каждого летательного аппарата реализуется на основе конечных автоматов (для всех аппаратов одной команды он одинаков). Целью данной работы является построение управляющего конечного автомата с помощью генетических алгоритмов [5–11].

В рамках работ по применению генетических алгоритмов для построения конечных автоматов, проводимых в СПбГУ ИТМО, предложено несколько методов пред-

ставления управляющего автомата в виде хромосомы, используемой в генетическом алгоритме [12–13]. В настоящей работе применяются метод представления автоматов деревьями решений [12] и метод сокращенных таблиц [13].

Структура системы управления беспилотным летательным аппаратом

В течение соревнования каждый летательный аппарат испытывает на себе воздействие среды и других летательных аппаратов. Для описания этих воздействий служат следующие логические входные переменные:

1. Граница трассы справа,
2. Граница трассы слева,
3. Другой аппарат слева,
4. Другой аппарат справа,
5. Другой аппарат спереди,
6. Другой аппарат сзади.

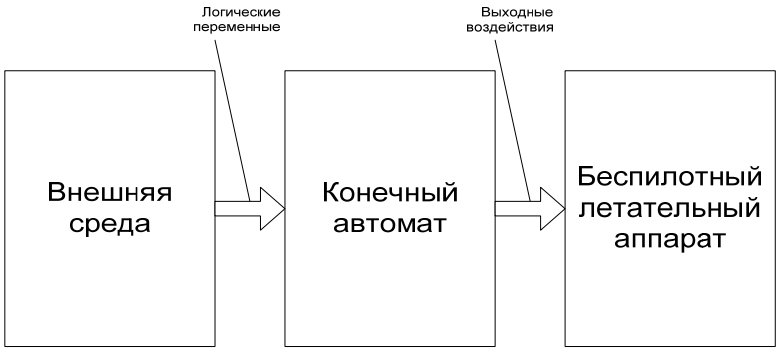


Рис. 1. Структурная схема системы управления беспилотным летательным аппаратом

Эти переменные подаются на вход управляющего автомата, который, в свою очередь, формирует последовательность выходных воздействий (рис. 1). Приведем список выходных воздействий:

1. установить нормальный расход топлива,
2. увеличить расход топлива на фиксированную величину,
3. уменьшить расход топлива на фиксированную величину,
4. сделать расход топлива максимально возможным,
5. изменить направление аэродинамического руля на фиксированный угол налево,
6. изменить направление аэродинамического руля на фиксированный угол направо,
7. лететь прямо.

Представление особи в генетическом алгоритме при помощи сокращенных таблиц переходов

При использовании метода сокращенных таблиц переходов каждая особь хранит следующие параметры:

1. массив состояний,
2. число состояний,
3. число обрабатываемых входных переменных,
4. число возможных действий.

Каждое состояние хранит сокращенную таблицу переходов, которая содержит:

1. массив значимых входных переменных.

2. массив состояний для всех переходов.
3. массив выполняемых действий для всех переходов.

На рис. 2 приведен пример сокращенной таблицы для одного из состояний автомата. Здесь столбец `S` – массив состояний, `Z1-Z4` массивы действий, `Z1[i]` равно единице, если данное действие выполняется. Массив `variables` указывает, какие переменные значимы, если переменной соответствует единица, то переменная значима.

S	Z1	Z2	Z3	Z4
1	0	0	1	0
3	1	0	1	1
5	1	0	1	0
1	1	0	1	1

variables	A	B	C	D	E	F	G
	0	0	1	0	0	1	0

Рис. 2. Пример сокращенной таблицы переходов

Значимыми называются входные переменные, значение которых обрабатываются при нахождении в данном состоянии (значения других переменных игнорируются).

Восстановление связей между состояниями

В данном разделе описан алгоритм восстановления связей между состояниями автомата, который обеспечивает достижение большего числа состояний автомата из начального за счет изменения конечного состояния некоторых переходов. Для поиска достижимых состояний используется алгоритм *поиска в ширину (Breadth First Search)*. Данный алгоритм описан ниже с помощью псевдокода.

```
TableAutomaton repairedAutomaton() {
  for (повторить N раз) {
    Запустить поиск в ширину от стартового состояния.
    Массив пометок mark содержит информацию о том, было
    ли посещено состояние.
    for (для всех i: состояний автомата) {
      if (!mark[i]) {
        State st = случайное достижимое состояние;
        Установить случайный переход из st в
        состояние i;
      }
    }
    if (не изменилось ни одно из состояний) {
      выйти из цикла;
    }
  }
  return this;
}
```

Данная процедура не гарантирует достижимость всех состояний из начального состояния для графа переходов, а лишь восстанавливает ее с вероятностью, возрастающей с константой N (в данной работе N = 10). Рис. 3 иллюстрирует работу данного алгоритма, серым цветом отмечены состояния, достижимые из стартового.

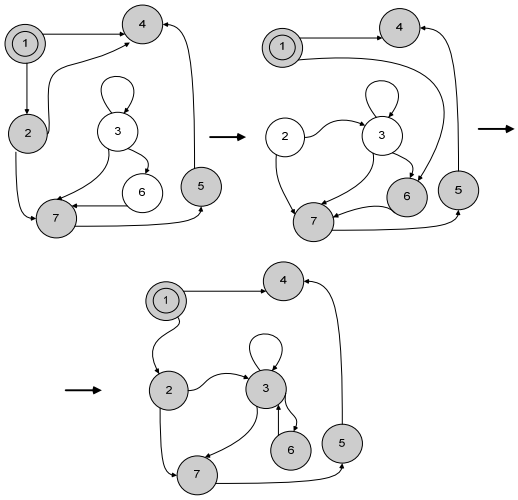


Рис. 3. Пример работы алгоритма восстановления связей между состояниями

Особенности применения сокращенных таблиц

Опишем преимущества метода сокращенных таблиц переходов применительно к рассматриваемой задаче:

- несовместность некоторых входных переменных (граница трассы слева, граница трассы справа) делает неиспользуемыми некоторые переходы. При использовании сокращенных таблиц вероятность такого события, по сравнению с полными таблицами, значительно снижается;
- сокращение используемой памяти, а также ускорение работы алгоритма.

При применении сокращенных таблиц все значимые переменные имеют равный приоритет, так как решение принимается сразу на основе всех переменных, которые используются в данном состоянии (альтернативой являются деревья решений [12], в которых решение принимается поэтапно).

Представление особи в генетическом алгоритме при помощи деревьев решений

При использовании метода представления автоматов деревьями решений каждая особь хранит следующие параметры:

- массив состояний;
- «рекомендуемая» высота дерева.

Каждое состояние представляет собой дерево решений для функции вида

$$f : \{0,1\}^n \to N \times \{0,1\}^k,$$

где n – число входных переменных, k – число возможных действий беспилотного летательного аппарата. Данное дерево по значениям входных переменных выдает номер состояния, в которое необходимо перейти автомату, а также вектор из нулей и единиц. При этом если i-ая компонента этого вектора равна единице, то аппарату необходимо выполнить i-ое действие. Далее будем называть этот вектор *вектором действий*. На рис. 4 приведен пример дерева решений для функции от двух булевых переменных (A, B).

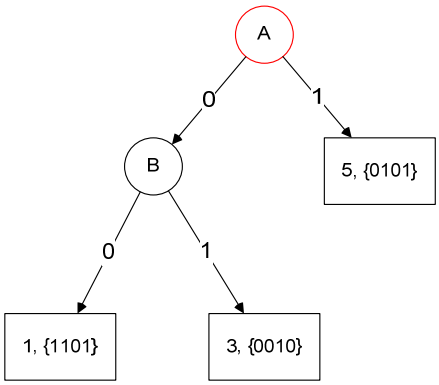


Рис. 4. Пример дерева решений

Каждое состояние представляется деревом решений, которое состоит из *узлов*. Среди узлов выделяется *корень* – стартовый узел дерева решений (на рис. 4 изображен сверху).

Каждый узел дерева решений состоит из следующих частей:

1. указателя на левого ребенка (для листьев этот указатель пуст);
2. указателя на правого ребенка (для листьев этот указатель пуст);
3. переменной, по которой происходит расщепление в данном узле (для листьев это значение не используется);
4. номера состояния автомата, в которое ведет переход из данной вершины (используется только в листьях);
5. вектор действий (для внутренних вершин этот вектор пуст).

Левый ребенок внутреннего узла соответствует нулевому значению переменной расщепления, а правый – единичному.

Алгоритм обрезки недостижимых ветвей дерева

Если по пути из корня до некоторого узла переменная, по которой происходит расщепление в этом узле, встречается дважды, то ветвь, соответствующая значению противоположному тому, которое было выбрано при первом расщеплении, будет *недостижимой*. Рис. 5 поясняет это утверждение. Светло-серым цветом на рис. 5 отмечены вершины с повторяющейся переменной расщепления, а темно-серым – недостижимая ветвь.

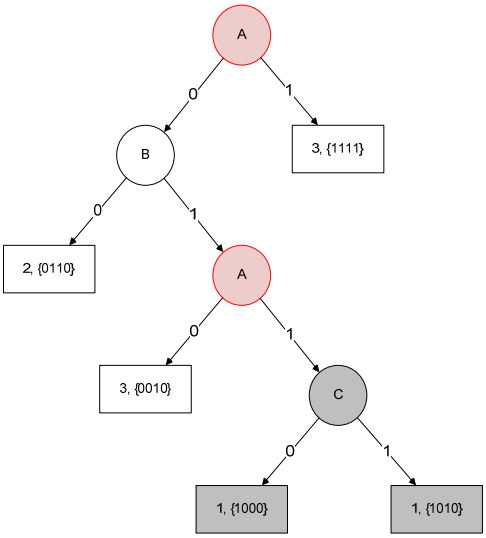


Рис. 5. Недостижимые ветви дерева решений

Для обрезки недостижимых ветвей используется модификация алгоритма *поиска в глубину* (*Depth First Search*). При рекурсивном спуске запоминаются переменные, по которым уже проводилось расщепление на пути из корня в текущий узел, а также значения этих переменных, соответствующие этому пути. Если переменная встречается второй раз, то текущий узел заменяется корнем достижимого поддерева этого узла. Решение о том, какое из поддеревьев достижимо, принимается на основании запомненных значений переменных. Работу данного алгоритма иллюстрирует рис. 6.

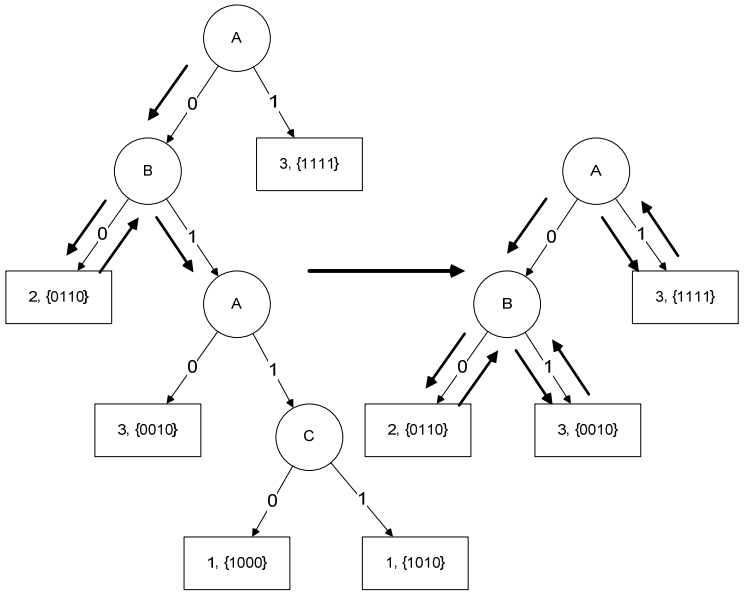


Рис. 6. Удаление недостижимых ветвей дерева решений

Генерация случайного дерева решений

Для генерации внутренней вершинц дерева применяется следующий алгоритм:

- выбрать переменную, по которой в данной вершине дерева будет проводиться расщепление;
- сгенерировать левого ребенка;
- сгенерировать правого ребенка.

Генерация листа происходит следующим образом:

- случайным образом выбрать номер состояния, в которое будет вести переход из данного листа;
- случайный образом сгенерировать вектор действий.

Для генерации случайного дерева решений применяется следующий алгоритм: с некоторой вероятностью (в работе использовалось значение 0.85) генерируется лист или внутренний узел. Данный алгоритм может продолжать свою работу бесконечно долго. Чтобы этого избежать введено ограничение на высоту дерева. Если при генерации высота дерева превысит удвоенное число возможных входных переменных, то обязательно генерируется лист.

Особенности представления при помощи деревьев решений

Дерево решений имеет те же достоинства, что и сокращенные таблицы:

- так же, как и при использовании сокращенных таблиц, могут использоваться не все входные переменные. Это позволяет с большей вероятностью, чем у представления с

помощью битовых строк или полных таблиц переходов, исключить из рассмотрения переходы с несовместными переменными;

- еще большее сокращение объема требуемой памяти по сравнению с сокращенными таблицами. Даже если в конкретном дереве используется *k* входных переменных, то его размер может быть меньше чем *2k* (вплоть до линейного от числа переменных).

Специфика деревьев решений указывает на то, что в каждом состоянии автомата задается свой приоритет входных переменных. Это связано с тем, что чем выше находится лист дерева, в котором происходит расщепление по данной переменной, тем больше вероятность, что в него можно попасть в зависимости от значений других входных переменных.

Генетический алгоритм

В работе для генерации системы управления беспилотным летательным аппаратом используется *островной генетический алгоритм* [8, 11]. Общая схема (рис. 7) алгоритма заключается в том, что существует несколько популяций – *островов*. Большую часть времени развитие популяций на каждом острове происходит независимо. При этом через заданное число поколений происходит *миграция* – часть особей с каждого острова передается другому острову.

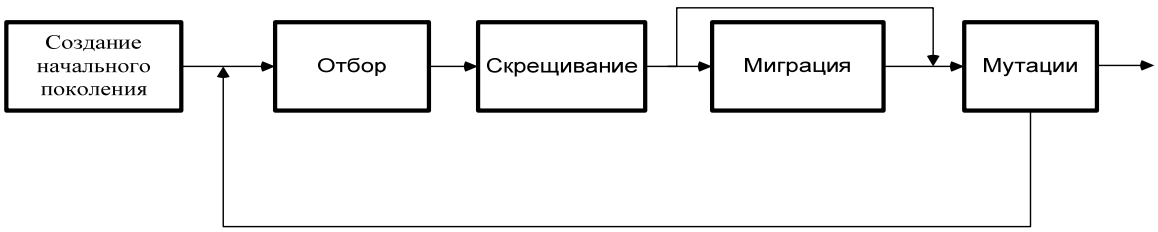


Рис. 7. Схема островного генетического алгоритма

Создание начального поколения

Все острова заполняются случайно сгенерированными особями. Все особи имеют заранее заданное число состояний. При применении сокращенных таблиц для хромосом всех особей задается число значимых переменных, которое одинаково для всех состояний.

Формирование следующего поколения

В качестве основной стратегии формирования следующего поколения используется элитизм. При рассмотрении текущего поколения отбрасываются все особи, кроме некоторой доли наиболее приспособленных – «элиты». «Элита» переходит в следующее поколение напрямую. После этого поколение дополняется в определенной пропорции случайными особями, особями из текущего поколения, которые мутировали, и результатами скрещивания особей из текущего поколения (отдельно отметим, что скрещиваться могут не только элитные особи, а все). Особи, «имеющие право» давать потомство, определяются «в поединке»: выбираются две случайные пары особей, и более приспособленная особь в каждой из них становится одним из родителей.

Оператор скрещивания особей, представленных при помощи сокращенных таблиц переходов

Оператор скрещивания особей для случая сокращенных таблиц переходов представляется следующим образом. Обозначим родительские особи – *P1* и *P2*, а детей – *S1*

и *S2*. Обозначим *k*-ое состояние автомата *A* как *A.a[k]*. Как *c1[k]* и *c2[k]* обозначим результат скрещивания состояний *P1.a[k]* и *P2.a[k]*. Тогда для любого *k* будет верно утверждение *S1.a[k] = c1[k], S2.a[k] = c2[k]*.

Алгоритм скрещивания состояний представляет собой *одноточечное* скрещивание соответствующих столбцов таблицы переходов. Для каждого столбца выполняются следующие действия:

- случайный выбор границы разделения столбца;
- соответствующий столбец состояния *c1* составляется из первой части столбца состояния *P1.a[k]* и второй части столбца особи *P2.a[k]*;
- соответствующий столбец состояния *c2* составляется из второй части столбца особи *P1.a[k]* и первой части столбца особи *P2.a[k]*.

Данный алгоритм проиллюстрирован на рис. 8.

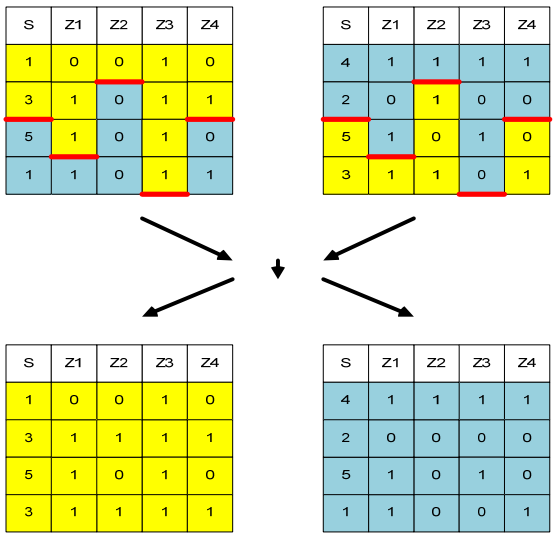


Рис. 8. Скрещивание сокращенных таблиц переходов

Для того чтобы выбрать значимые переменные *c1[k]* и *c2[k]*, выполняется просмотр списка значимых переменных родителей. Обозначим за *v[k] k*-ый элемент в списке значимых переменных. При этом справедливо одно из следующих утверждений:

1. *P1.v[k] = 0, P2.v[k] = 0* → *S1.v[k] = 0, S2.v[k] = 0*;
2. *P1.v[k] = 1, P2.v[k] = 1* → *S1.v[k] = 1, S2.v[k] = 1*;
3. *P1.v[k] = 0, P2.v[k] = 1* или *P1.v[k] = 1, P2.v[k] = 0* → возможен один из трех случаев:
   1. списки значимых переменных обоих детей еще не заполнены (число значимых переменных в обоих списках не превышает число значимых переменных у родителей), тогда с вероятностью *0.5* выполняются соотношения *P1.v[k] = 0, P2.v[k] = 1*, иначе *P1.v[k] = 1, P2.v[k] = 0*;
   2. списки значимых переменных первого ребенка заполнены, тогда *P1.v[k] = 0, P2.v[k] = 1*;
   3. списки значимых переменных второго ребенка заполнены, тогда *P1.v[k] = 0, P2.v[k] = 1*;

Отметим, что списки значимых переменных обоих потомков могут быть заполнены одновременно тогда и только тогда, когда в списках обоих родителей не осталось не просмотренных значимых переменных.

Оператор мутации особей, представленных при помощи сокращенных таблиц переходов

При мутации особи выполняются следующие действия:

- с вероятностью 0.5 случайное изменение начального состояния;
- мутация случайного состояния.

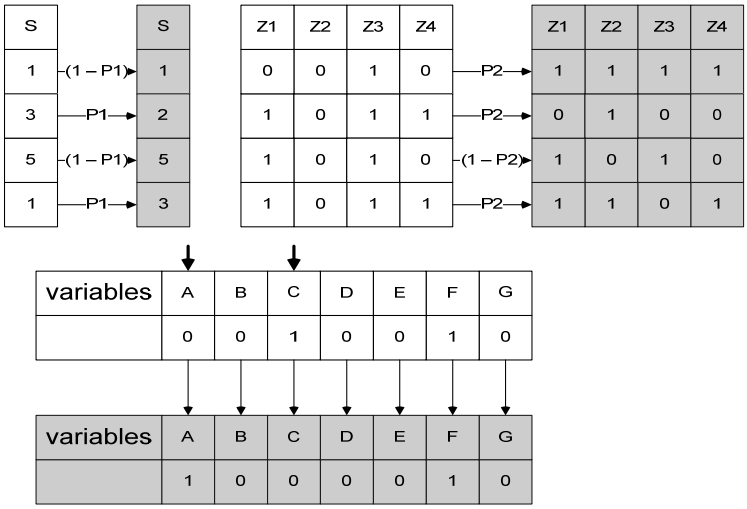


Рис. 9. Мутация сокращенных таблиц

Мутация состояния следующим образом: для каждой строки таблицы переходов выполнить:

- с некоторой заранее заданной вероятностью *p1* изменить переход из данного состояния по текущим значениям входных переменных на случайное значение (из допустимых);
- с некоторой заранее заданной вероятностью *p2* изменить все действия на данном переходе. Каждое действие выполняется на данном переходе с вероятностью *n1/ k*, где *n1* – число выполняемых действий на данном переходе до начала мутации, *k* – число возможных действий.

Выбор *значимых переменных* состояния осуществляется следующим образом:

- выбрать случайным образом две переменные;
- если одна из них значима в данном состоянии, а другая нет, то они меняются местами. На рис. 9 приведен пример мутации. Серым цветом отмечены получившиеся части состояния.

По окончании работы оператора мутации для получившейся особи запускается алгоритм восстановления связей между состояниями.

Оператор скрещивания особей, представленных при помощи деревьев решений

Оператор скрещивания особей представляется следующим образом. Обозначим родительские особи – *P1* и *P2*, а детей – *S1* и *S2*. Обозначим *k*-ое состояние автомата *A*, как *A.a[k]*. Обозначим *c1[k]* и *c2[k]* результат скрещивания состояний *P1.a[k]* и *P2.a[k]*. Тогда для любого *k* будет верно утверждение:

S1.a[k] = c1[k], S2.a[k] = c2[k].

Оператор скрещивания состояний фактически выбирает два поддерева – одно из первого дерева решений, второе из второго – а затем меняет их местами. Этот алгоритм, также как операция мутации и алгоритм обрезки недостижимых ветвей, представляет собой модификацию алгоритма поиска в глубину и имеет рекурсивную структуру. При каждом рекурсивном вызове этого алгоритма выполняются следующие действия:

- если текущий узел является листом, то обязательно, а для внутренних узлов с вероятностью P, вернуть данное поддерево;
- иначе равновероятно пойти в одно из поддеревьев текущего узла.

На рис. 11 приведен пример, в котором необходимо, как и в случае с оператором мутации, провести обрезку недостижимых ветвей. При этом для всех деревьев, соответствующих состояниям автомата, запускается алгоритм обрезки недостижимых ветвей.

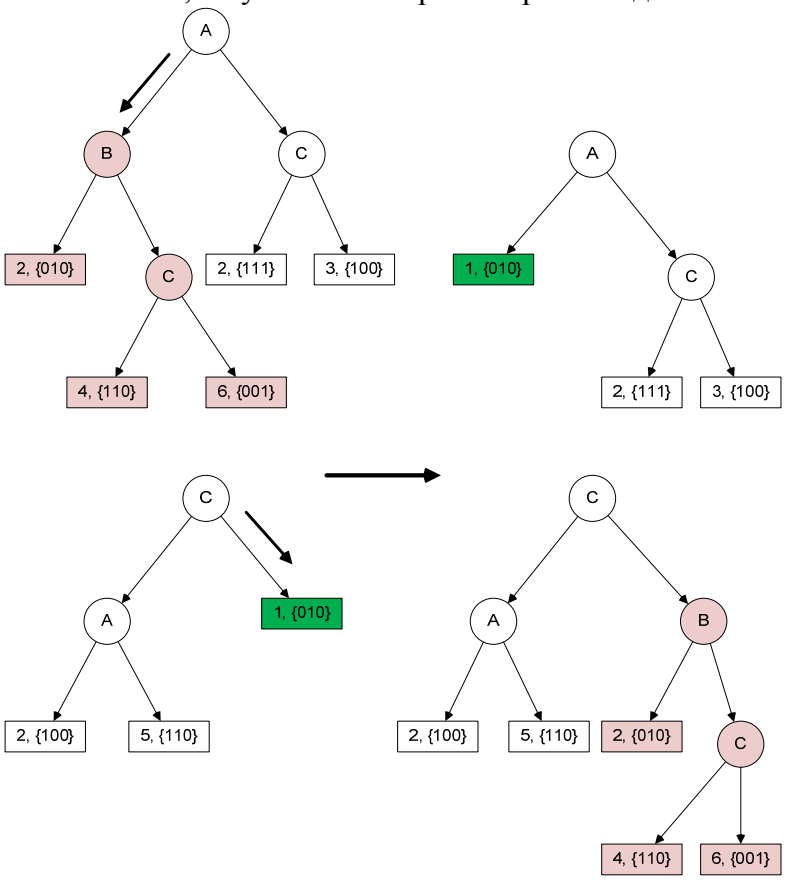


Рис. 11. Пример скрещивание деревьев решений

Оператор мутации особей, представленных при помощи деревьев решений

Оператор мутации выполняет:

- с вероятностью *0.5* случайное изменение стартового состояния;
- мутацию случайного состояния.

Оператор мутации состояния представляет собой модификацию алгоритма поиска в глубину. Выполняются следующие действия:

- если текущий узел является листом, то обязательно, а для внутренних узлов с вероятностью P, вернуть случайно сгенерированное дерево решений;
- иначе равновероятно пойти в одно из поддеревьев.

Фактически данный алгоритм случайно выбирает некоторое поддерево и заменяет его на случайно сгенерированное. Выбор поддерева происходит не равновероятно – чем выше узел, тем больше вероятность выбора его поддерева. По окончании мутации для мутированной особи необходимо запустить алгоритм обрезки недостижимых ветвей дерева. Это иллюстрирует рис. 10.

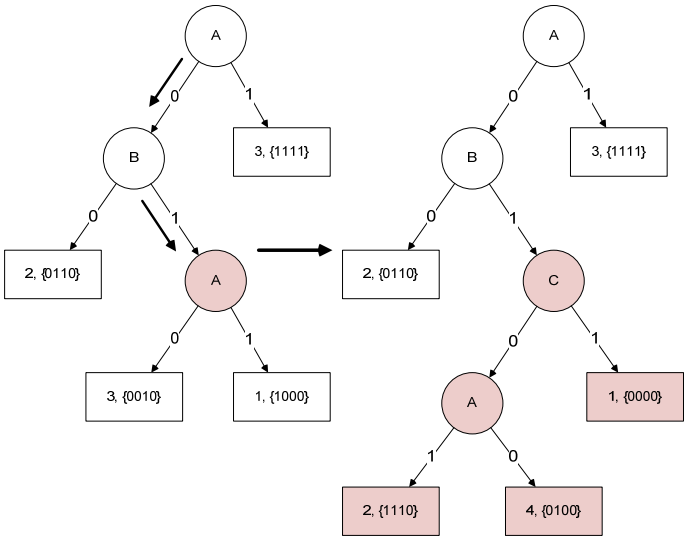


Рис. 10. Мутация деревьев решений

Миграции

Через фиксированное число поколений каждый остров меняется с другим случайным числом случайно выбранных элитных особей. Также через некоторое число поколений происходит *большая мутация*, называемая также *мутацией острова*.

Большие мутации

Для того чтобы избежать «вырождения» автоматов (попадания в локальный максимум функции приспособленности), предлагаемый генетический алгоритм использует описанную ниже процедуру. Через заранее заданное число поколений фиксированная доля островов заменяется островами со случайными особями. Это является причиной провала (рис. 12) на 41-ом поколении.

Проведение мутации в момент, когда функция приспособленности (имеются в виду только элитные особи) изменяется незначительно, невозможно, так как за счет миграции особей между островами среднее значение функции приспособленности постоянно изменяет свое значение.

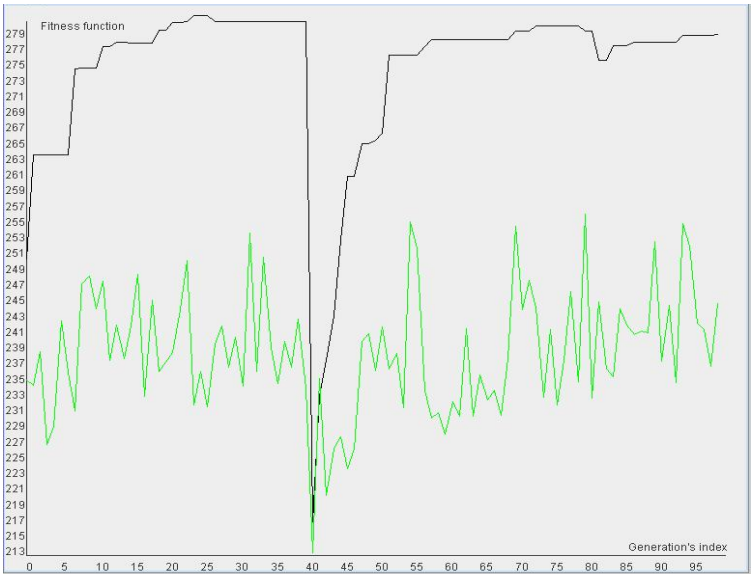


Рис. 12 Большая мутация

Вычисление функции приспособленности

Функция приспособленности особи должна зависеть от результата команды беспилотных летательных аппаратов, использующей стратегию, описываемую этим автоматом. Чтобы задать вид зависимости, требуется определить условия соревнования: начальные координаты летательных аппаратов и команду соперника. В этом-то и состоит проблема, так как, если при каждом запуске автомата начальные координаты и соперника выбирать случайным образом, то функция приспособленности будет «необъективна». Противоположный вариант – зафиксировать соперника и начальные координаты.

Предлагается вычислять функцию приспособленности как среднее арифметическое результатов нескольких соревнований. При этом начальные координаты – случайные, но одинаковые для всех команд, реализующих стратегию, заданную генерируемыми особями. Число соревнований выбиралось равным 10 из соображений уменьшения времени работы алгоритма.

$$F = \frac{\sum_{i=0}^{k} r_k}{k}.$$

Для особей, представленных в виде деревьев решений, формула была несколько изменена:

$$F = \frac{\sum_{i=0}^{k} r_k}{k} - C * Z(h_{\max}, height),$$

где C – некоторая константа, h_{max} – максимальная из высота деревьев решений, соответствующих состояниям данного автомата, *height* – «рекомендуемая» высота дерева решений, а Z – функция, определяемая следующим образом:

$$Z(a,b) = \begin{cases} a-b, a \geq b; \\ 0, a < b. \end{cases}$$

Таким образом, данная функция отвечает за то чтобы дерево решений не сильно «разрасталось». Если дерево решений имеет высоту больше *height*, функция приспособленности такой особи может быть ниже, даже если средний результат ее соревнований выше.

В качестве соперников были выбраны две команды. Первая – команда с «агрессивной» стратегией. Эта стратегия заключается в том, что один летательный аппарат летит только вперед с постоянным нормальным расходом топлива (нормальный расход топлива – такой расход топлива, при котором достигается наибольшая дальность полета). Остальные аппараты пытаются сбить летательные аппараты соперника. В качестве второго соперника была выбрана команда со стратегией, заданной автоматом, построенным с помощью описываемого генетического алгоритма, но при вычислении функции приспособленности, в которой в качестве соперника была взята только команда с «агрессивной» стратегией.

Этот автомат был выбран по двум причинам. Во-первых, он «продемонстрировал свою силу» в соревнованиях с «агрессивной» командой и командой, реализующей стратегию, заданную системой автоматов, построенных вручную (описана в работе [3]). Во-вторых, в отличие от этих двух команд, он реализует детерминированную стратегию. И, наконец, этот автомат реализует «дружелюбную» стратегию, которая заключается в том, что летательные аппараты помогают лететь друг другу, а не пытаются сбить аппараты команды соперника. Это позволяет «тренировать» новые управляющие автоматы сразу на двух принципиально различных стратегиях: «агрессивной» и «дружелюбной».

Проблемой является принципиальная невозможность оценить полученную особь по абсолютной шкале, как уже упоминалось ранее. Это происходит из-за недетермини-

рованности начальных параметров задачи, а также противников. Этот эффект можно наблюдать на рис. 12, где изображены графики функции приспособленности особи, представляющей собой систему управления беспилотными летательными аппаратами от индекса поколения. Светлым цветом на этих графиках изображена средняя дальность летательного аппарата при проведении 30 соревнований (против десяти при вычислении функции приспособленности), а черным – функция приспособленности.

Число поколений работы генетического алгоритма ограничено лишь теми соображениями, что генерируемые на больших поколениях особи не демонстрируют универсального поведения, а приспосабливаются к соперникам и начальным условиям. Однако, если задан конкретный противник, то вырастить автомат, который, по крайней мере, играет не хуже его, не составляет особого труда. Естественно, результат соревнования все равно будет зависеть от начальных условий, но эта зависимость будет не слишком значимой.

Особенности применения островного генетического алгоритма

Островной генетический алгоритм имеет следующие достоинства (перед алгоритмами, в которых поколение развивается как единое целое):

- более быстрая сходимость к максимуму за счет миграций;
- возможность выхода из локального максимума без потери результатов предыдущих вычислений за счет «изолированности» островов.

К недостаткам островного алгоритма можно отнести более частое попадание в локальные максимумы по сравнению с традиционным генетическим алгоритмом.

Результаты

При помощи алгоритма генетического программирования с использованием метода представления автоматов с помощью сокращенных таблиц переходов построен конечный автомат управления моделью беспилотного летательного аппарата. Его граф переходов приведен на рис. 13. Построение этого автомата заняло 62 поколения при следующих значениях параметров алгоритма.

1. Число островов – 10.
2. Размер острова – 100 особей.
3. Процент «элиты» – 10%.
4. Вероятность мутации – 10%.
5. Процент особей, участвующих в миграции между островами – 3%.
6. Число поколений, проходящее между миграциями особей между островами – 10.
7. Число поколений, проходящее между «большими» мутациями – 41.
8. Процент островов, на которых все особи заменяются случайно сгенерированными в процессе «большой» мутации – 85%.

На рис. 13 используются указанные ниже обозначения. Пометки на вершинах имеют вид *номер/массив значимых переменных*. Переменные перечисляются в следующем порядке.

1. Граница трассы справа.
2. Граница трассы слева.
3. Другой аппарат слева.
4. Другой аппарат справа.
5. Другой аппарат спереди.
6. Другой аппарат сзади.

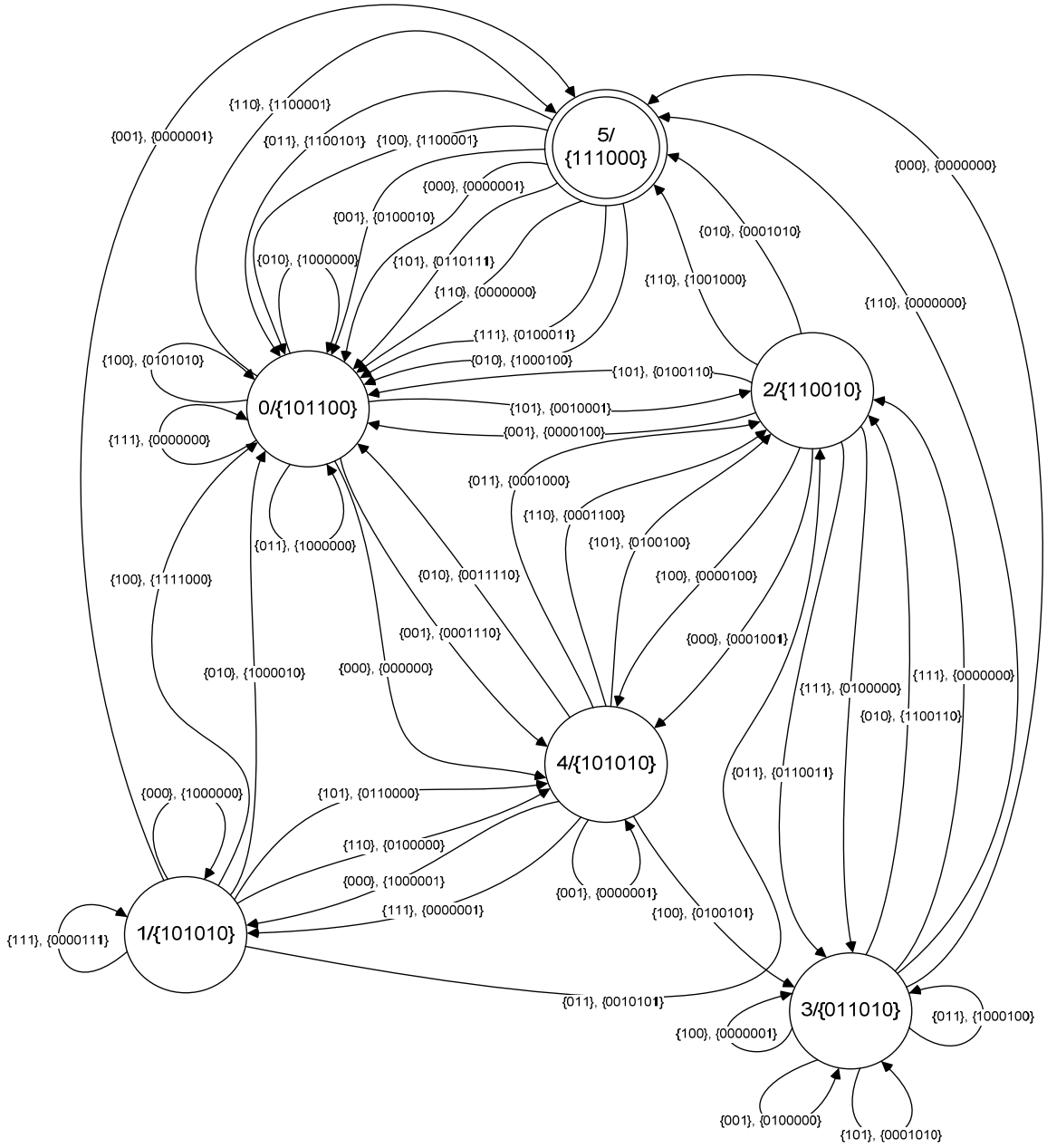


Рис. 13. Управляющий автомат, построенный алгоритмом генетического программирования

Пометки на переходах имеют вид *{значения переменных}/{вектор действий}*. Действия перечисляются в следующем порядке.

1. Установить нормальный расход топлива.
2. Изменить направление аэродинамического руля на фиксированный угол налево.
3. Изменить направление аэродинамического руля на фиксированный угол направо.
4. Лететь прямо.
5. Сделать расход топлива максимально возможным.
6. Увеличить расход топлива на фиксированную величину.
7. Уменьшить расход топлива на фиксированную величину.
8. Двойной окружностью показано начальное состояние автомата.

Команда, реализующая стратегию, описываемую полученным автоматом, из 50 соревнований с «агрессивной» командой выиграла 45, а с командой, построенной вручную [3] – 43.

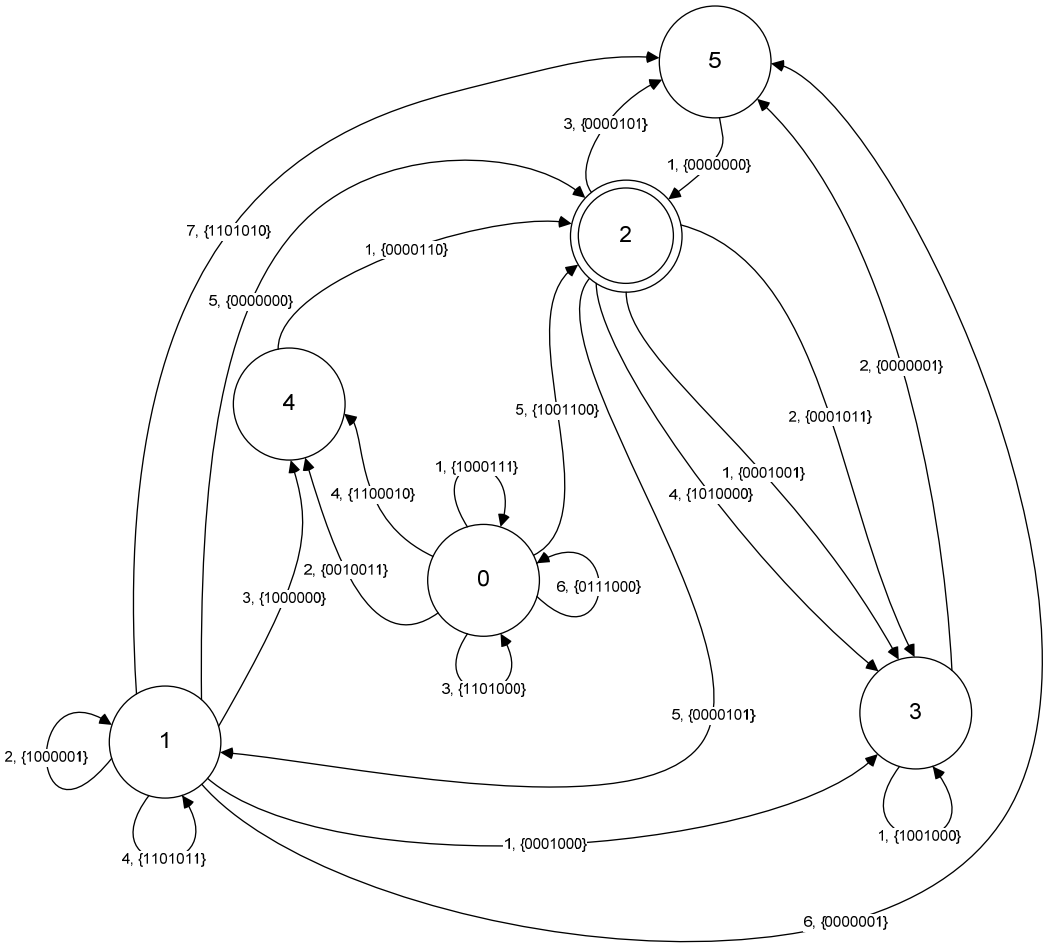


Рис. 14. Граф переходов конечного автомата, построенного с применением метода представления автоматов с помощью деревьев решений

При помощи алгоритма генетического программирования с использованием метода представления автоматов с помощью деревьев решений построен конечный автомат управления беспилотным летательным аппаратом. Его граф переходов приведен на рис. 14. Пометки на переходах имеют формат *номер перехода, {вектор действий}*. Построение этого автомата заняло 32 поколения при тех же значениях параметров, что и в случае с сокращенными таблицами, со следующими специфическими параметрами:

- рекомендуемая высота дерева решений – 4;
- вероятность генерации внутреннего узла дерева решений – 75%.

Опишем семантику каждой компоненты вектора действий.

1. Установить нормальный расход топлива.
2. Изменить направление аэродинамического руля на фиксированный угол налево.
3. Изменить направление аэродинамического руля на фиксированный угол направо.
4. Лететь прямо.
5. Сделать расход топлива максимально возможным.
6. Увеличить расход топлива на фиксированную величину.
7. Уменьшить расход топлива на фиксированную величину.

Переходы из каждого состояния пронумерованы натуральными числами, начиная с единицы. Номера переходов соответствуют номерам листов деревьев решений, показанных на рисунках ниже. Заметим, что в этом автомате состояние с номером 0 недостижимо и может быть удалено.

На следующих рисунках внутренние узлы дерева решений показаны кругами, а листья – прямоугольниками. Внутренние узлы помечены номером переменной, по которой осуществляется ветвление. Левое поддерево соответствует значению «ложь» переменной, которой помечен узел, а правое – значению «истина». Листья помечены номером перехода, который выполняется при достижении этого листа. Эти номера переходов соответствуют номерам переходов на рис. 15. Приведем перечень переменных, используемых на приведенных в настоящем разделе деревьях.

1. Граница трассы справа.
2. Граница трассы слева.
3. Другой аппарат слева.
4. Другой аппарат справа.
5. Другой аппарат спереди.
6. Другой аппарат сзади.

На рис. 15 показано дерево решений, соответствующее состоянию с номером 2 построенного автомата, на рис. 16 – дерево решений, соответствующее третьему состоянию автомата, на рис. 17 – дерево решений, соответствующее четвертому и пятому состояниям автомата. Наличие в этом дереве только одного узла, который является одновременно и корнем и листом, означает, что из этих состояний всегда выполняется один и тот же безусловный переход.

Приведем результаты соревнований, показанных системами управления, построенными с помощью метода сокращенных таблиц переходов, метода представления автоматов деревьями решений и построенной вручную. На рис. 18 показано распределение результатов, обеспеченных системой управления, построенной при помощи метода сокращенных таблиц переходов. На рис. 19 показано распределение результатов, обеспеченных системой управления, построенной при помощи метода представления автоматов деревьями решений. На рис. 20 показано распределение результатов, обеспеченных системой управления, построенной вручную в работе [3].

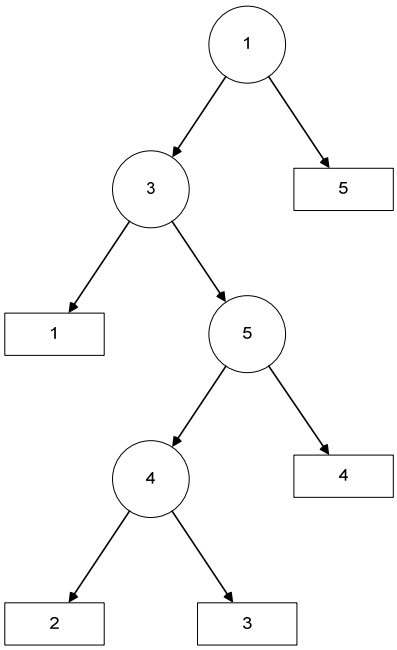


Рис. 15. Дерево решений, соответствующее второму состоянию автомата

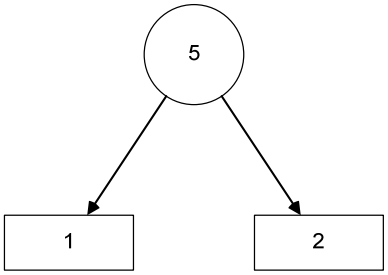


Рис. 16. Дерево решений, соответствующее третьему состоянию автомата

Рис. 17. Дерево решений, соответствующее четвертому и пятому состояниям автомата

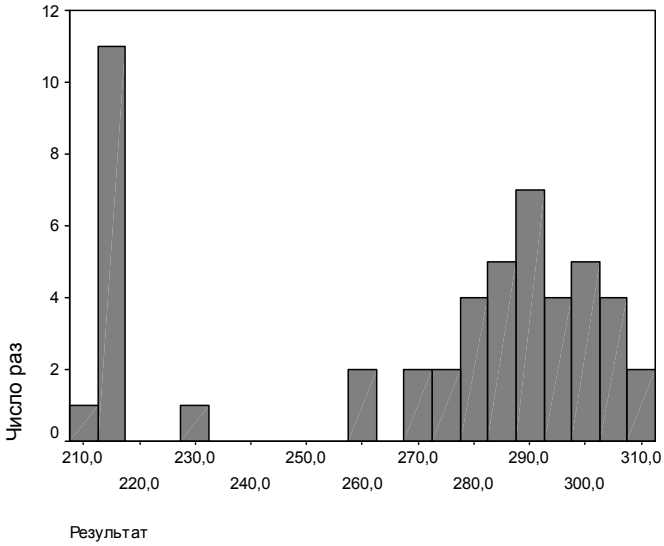


Рис. 18. Распределение результатов, обеспеченных системой, построенной при помощи метода сокращенных таблиц переходов

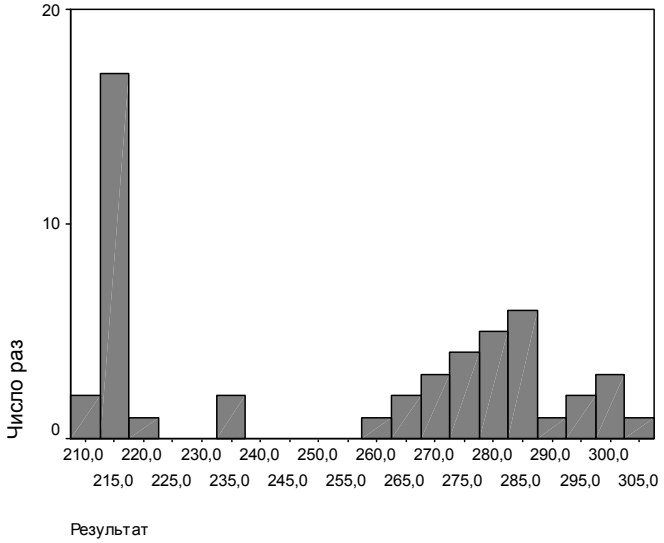


Рис. 19. Распределение результатов, обеспеченных системой, построенной при помощи метода представления автоматов деревьями решений

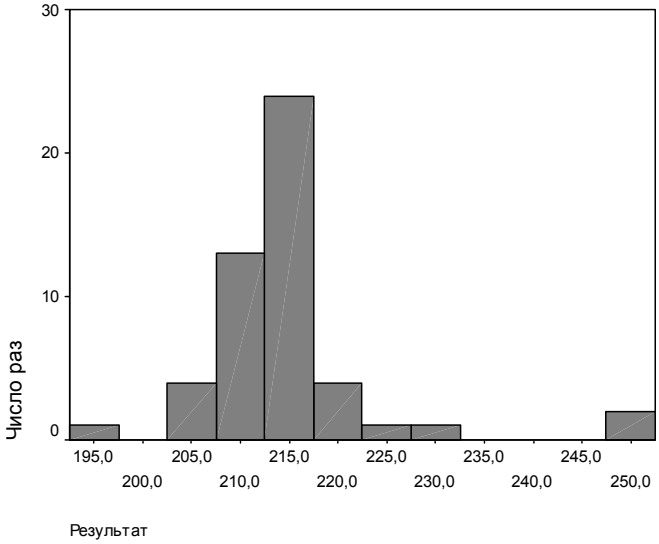


Рис. 20. Распределение результатов, обеспеченных системой из работы [3]

Выводы

В данной работе описано применение островного генетического алгоритма, а также двух способов представления особей (сокращенных таблиц переходов и деревьев решений), для создания управляющего автомата для модели беспилотного летательного аппарата. Проведено сравнение построенных автоматов управления с системой автоматов, построенной в работе [3]. Результаты этого сравнения позволяют сделать вывод о том, что построенные управляющие автоматы показали результаты, значительно превосходящие результаты, полученные с помощью системы автоматов, построенной вручную в работе [3]. Это доказывает эффективность используемых методов представления автоматов и их оптимизации применительно к рассматриваемой задаче.

Литература

1. Шалыто А.А. Технология автоматного программирования // Труды первой Всероссийской научной конференции «Методы и средства обработки информации». – М.: МГУ, 2003. – Режим доступа: http://is.ifmo.ru/works/tech_aut_prog/
2. Заочный тур всесибирской олимпиады 2005 по информатике. – Режим доступа: http://olimpic.nsu.ru/widesiberia/archive/wso6/2005/rus/1tour/problem/problem.html
3. Параценко Д.А., Царев Ф.Н., Шалыто А.А. Технология моделирования одного класса мультиагентных систем на основе автоматного программирования на примере игры «Соревнование летающих тарелок». Проектная документация. – СПбГУ ИТМО. 2006. – Режим доступа: http://is.ifmo.ru/unimod-projects/plates/
4. Рассел С., Норвиг П. Искусственный интеллект. Современный подход. – М.: Вильямс. 2006.
5. Jefferson D., Collins R., Cooper C., Dyer M., Flowers M., Korf R., Taylor C., Wang A. The Genesys System. 1992. – Режим доступа: www.cs.ucla.edu/~dyer/Papers/AlifeTracker/Alife91Jefferson.html
6. Angeline P. J., Pollack J. Evolutionary Module Acquisition // Proceedings of the Second Annual Conference on Evolutionary Programming. 1993. – Режим доступа: http://www.demo.cs.brandeis.edu/papers/ep93.pdf

7. Chambers L. Practical Handbook of Genetic Algorithms. Complex Coding Systems. Volume III. CRC Press, 1999.
8. Гладков Л.А., Курейчик В.В., Курейчик В.М. Генетические алгоритмы. – М.: Физматлит, 2006.
9. Koza J. R. Genetic programming: on the programming of computers by means of natural selection. – MIT Press, 1992.
10. Царев Ф.Н., Шалыто А.А. О построении автоматов с минимальным числом состояний для задачи об «умном муравье» // Сборник докладов X международной конференции по мягким вычислениям и измерениям. – СПбГЭТУ "ЛЭТИ". – Т.2. – 2007. – С.88–91. – Режим доступа: http://is.ifmo.ru/download/ant_ga_min_number_of_state.pdf
11. Яминов Б. Генетические алгоритмы. – Режим доступа: http://rain.ifmo.ru/cat/view.php/theory/unsorted/genetic-2005
12. Данилов В.Р. Технология генетического программирования для генерации автоматов управления системами со сложным поведением. СПбГУ ИТМО. 2007. Бакалаврская работа. http://is.ifmo.ru/papers/danilov_bachelor/
13. Поликарпова Н.И., Точилин В.Н. Применение генетического программирования для реализации систем со сложным поведением // Научно-технический вестник СПбГУ ИТМО. – 2007. – Выпуск 39. – С.276–293. – Режим доступа: http://vestnik.ifmo.ru/ntv/39/ntv_39.3.3.pdf

УДК 004.4’242

ПОСТРОЕНИЕ АВТОПИЛОТА ДЛЯ УПРОЩЕННОЙ МОДЕЛИ ВЕРТОЛЕТА С ПОМОЩЬЮ ГЕНЕТИЧЕСКОГО АЛГОРИТМА

П.Г. Лобанов, С.А. Сытник, А.А. Шалыто

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

Предложен генетический алгоритм построения автопилота для упрощенной модели вертолета. Задачей вертолета является прохождение заданного множества целей в заданном порядке. алгоритм реализован на языке программирования *Java*. Выполнены вычислительные эксперименты, демонстрирующие эффективность этого алгоритма.

Ключевые слова: генетические алгоритмы, конечный автомат, автоматное программирование

Введение

Существует ряд задач, которые эффективно решаются с помощью конечных автоматов. В большинстве случаев построение автоматов выполняется эвристически [1], что достаточно трудоемко. В связи с этим актуальна задача разработки методов автоматизированной генерации автоматов. Генетические алгоритмы [2] представляют собой мощный подход, позволяющий получать точные или достаточно близкие к ним решения для широкого спектра задач. Они применимы и в тех случаях, когда решением задачи является конечный автомат. Для таких задач, как «умный муравей» [2], итерированная дилемма узников [3], задача о флибах [4], синхронизация [5] и классификация плотности для клеточных автоматов [6, 7], известны генетические алгоритмы [2], которые позволяют автоматически строить автоматы. В данной работе рассматривается одна из таких задач – задача построения автопилота для простейшей модели вертолета с помощью генетического алгоритма.

Постановка задачи

Требуется построить автопилот для простейшего вертолета, перемещающегося в двумерной плоскости. За один шаг вертолет может повернуться на некоторый фикси-

рованный угол и изменить скорость своего движения (ускориться или замедлиться). Минимальная скорость, с которой может перемещаться модель вертолета, $V_{min} = 10^{-4}$, максимальная – $V_{max} = 2$, ускорение $a = 0.1$ (все величины и графики в работе приведены в условных единицах). Требуется построить автопилот, который за отведенное время полета проведет вертолет в определенном порядке через максимальное число заранее заданных целей. Цель – это точка, через которую должен пройти вертолет. Цели нумеруются, а вертолет проходит цели в порядке возрастания номеров. При этом вертолет не может пропускать цели, но, так как время полета ограничено, он может не успеть пройти все из них. Считается, что вертолет прошел цель, если он оказался от нее на расстоянии не более 0.5.

Вертолет и окружающая его среда представляются моделями, образующими взаимодействующую систему (рис. 1).

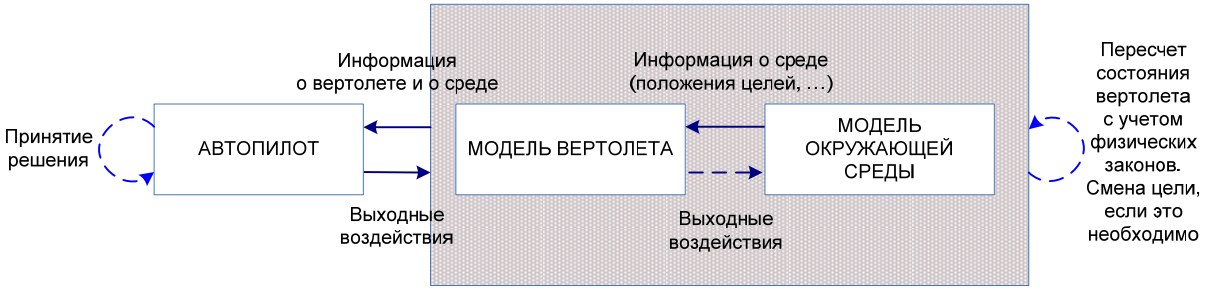


Рис. 1. Структурная схема системы

Модель вертолета, используя полученную от среды информацию, вычисляет необходимые ей данные о текущем состоянии системы. Она использует часть этих данных для их передачи автопилоту в качестве входных воздействий с целью анализа текущей ситуации и принятия решений. Модель окружающей среды «знает», что представляют собой эти воздействия с точки зрения определенных в ней физических законов, и в соответствии с ними пересчитывает положение и скорость вертолета.

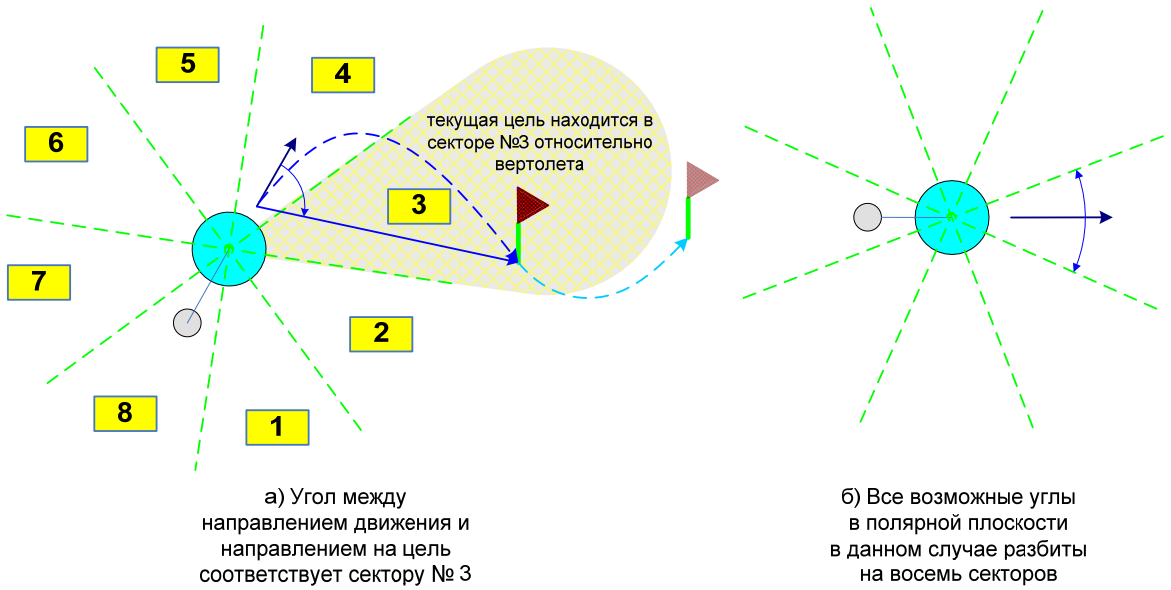


Рис. 2. Входные данные

Входные данные (воздействия) (рис. 2) автопилота представляют собой единственную переменную: положение текущей цели относительно вертолета, заданное углом между направлением движения вертолета и направлением на цель (рис. 2, а). Сектора всегда неподвижны относительно вертолета. Необходимо отметить, что вертолет летит не по границе двух секторов, а посередине одного сектора (рис. 2, б).

В качестве выходных в работе выбраны два воздействия, которые действуют относительно текущего вектора движения вертолета: «Изменить скорость» (придать ускорение в направлении вектора движения) и «Повернуть» (повернуть вектор движения на некоторый угол). Текущие значения ускорения и угла поворота никак не связаны с их предыдущими значениями. В каждый момент времени вертолет знает о положении только одной цели. Таким образом, следующее состояние вертолета зависит от его положения в пространстве, скорости движения и текущей цели. При достижении вертолетом цели текущей становится следующая по порядку. В результате направление на цель изменяется.

В качестве модели автопилота используется конечный автомат. Его входные и выходные воздействия должны быть дискретными. Для этого пространство вокруг вертолета разбивается на сектора обзора вертолета, число которых определяется заранее и является настраиваемым параметром. Автомат содержит некоторое число состояний, ограниченное сверху. Из каждого состояния графа переходов исходят дуги (переходы), число которых равно числу секторов. В качестве значения входной переменной, помечающей соответствующий переход, выступает номер сектора, в котором находится текущая цель вертолета. Каждый переход переводит автомат в новое состояние. Он помечен конкретным набором выходных воздействий. Поскольку автомат не использует памяти для хранения информации о глубокой предыстории (зависит только от предыдущего состояния), выбор следующего состояния производится автоматом лишь на основе входного воздействия и текущего состояния. Состояния *косвенно* отражают информацию о текущем положении вертолета, его скорости и предыстории переходов между состояниями. Рис. 3 иллюстрирует работу автопилота, заданного конечным автоматом.

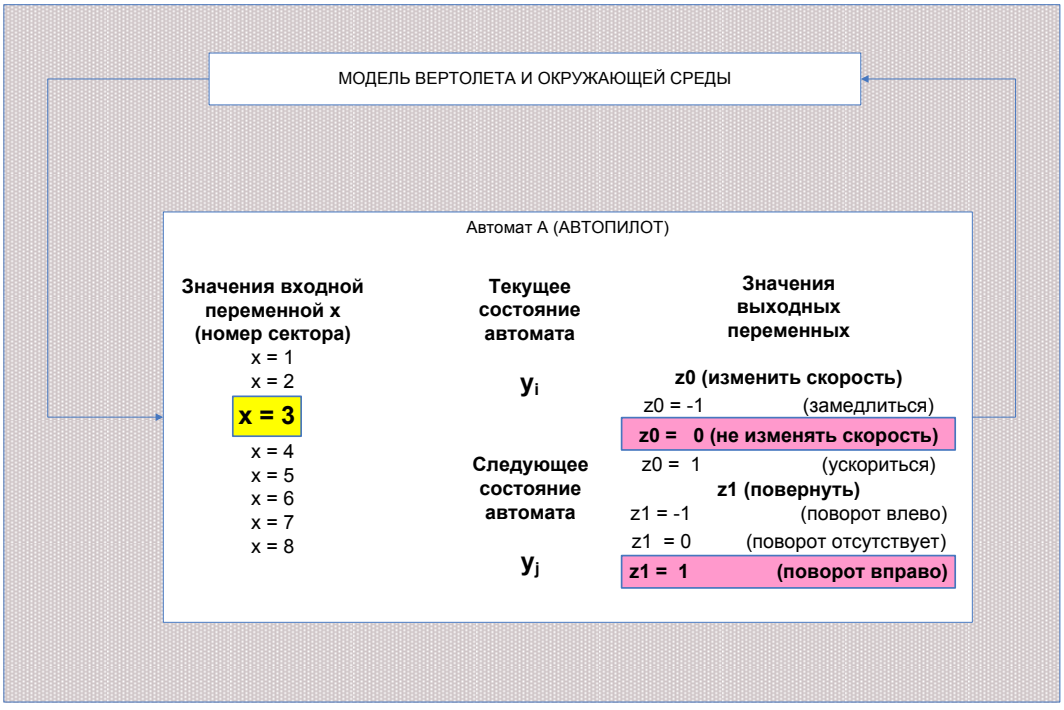


Рис. 3. Работа автопилота, заданного конечным автоматом

Модель вертолета

Модель вертолета содержит данные о положении вертолета в пространстве, его скорости и направления движения. Положение вертолета в пространстве хранится в виде точки $|x_h, y_h|$. Скорость и направление движения вертолета задаются вектором $|v_x, v_y|$, который называется вектором скорости. При этом скорость вертолета определяется соотношением $V = \sqrt{v_x^2 + v_y^2}$. Так как по условию задачи скорость вертолета не может быть меньше 10^{-4}, то вектор скорости всегда однозначно определяет направление движение вертолета.

Модель вертолета отвечает за изменение его положения в пространстве и пересчет скорости. Положение вертолета в следующий момент времени $|x'_h, y'_h|$ вычисляется по формуле $|x'_h, y'_h| = |x_h + v_x, y_h + v_y|$.

Определение вектора скорости $|v'_x, v'_y|$ для следующего момента времени выполняется за два шага. Сначала с помощью формулы

$$|v''_x, v''_y| = \left| v_x \frac{V + 0.1z_0}{V}, v_y \frac{V + 0.1z_0}{V} \right|$$

определятся *промежуточный вектор*, учитывающий изменение скорости вертолета, задаваемое значением выходной переменной `z0`. Затем определяется искомый вектор с помощью поворота *промежуточного вектора*

$$|v'_x, v'_y| = \left| v''_x + \frac{0.1z_1 v''_y}{V''}, v''_y - \frac{0.1z_1 v''_x}{V''} \right|,$$

где z_1 – значение выходной переменной `z1`, а V'' – длина промежуточного вектора $|v''_x, v''_y|$. Если длина полученного вектора V'' оказывается меньше минимальной скорости V_{min}, то вектор скорости $|v'_x, v'_y|$ вычисляется по формуле

$$|v'_x, v'_y| = \left| v'_x \frac{V_{min}}{V'}, v'_y \frac{V_{min}}{V'} \right|.$$

Если V' больше максимальной скорости V_{max}, то $|v'_x, v'_y|$ изменяется по формуле

$$|v'_x, v'_y| = \left| v'_x \frac{V_{max}}{V'}, v'_y \frac{V_{max}}{V'} \right|.$$

Модель окружающей среды

В модели окружающей среды хранятся координаты целей вертолета $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ и номер текущей цели `k`. Когда расстояние от вертолета до текущей цели становится меньше 0.5 ($\sqrt{(x_k - x_h) + (y_k - y_h)} < 0.5$), номер `k` увеличивается на единицу.

Модель среды отвечает за определение номера сектора, в котором находится текущая цель относительно вертолета. Сначала вычисляется угол α между направлением движения вертолета и направлением на текущую цель. Для этого используется формула

$$\alpha = \mathrm{arctg}\left(\frac{v_x(y_k - y_h) - v_y(x_k - x_h)}{v_x(x_k - x_h) + v_y(y_k - y_h)} \right).$$

Номер сектора n вычисляется по формуле:

$$n = \left\lfloor \frac{\frac{\alpha}{\pi} + 1}{2m} \right\rfloor + 1,$$

где m – число секторов обзора.

Алгоритм построения автопилота

Задача автопилота – провести вертолет через максимальное число наперед заданных целей в определенном порядке. Как было отмечено выше, время полета ограничено. Лучшим является автопилот, который провел вертолет через большее число целей за отведенное время. Оценочная функция (функция *fitness*) [2] должна быть максимально гладкой, что позволяет улучшить точность сравнения особей. Если число пройденных целей одинаково, то лучшим считается автопилот, для которого расстояние до следующей цели к моменту завершения полета минимально.

В настоящей работе хромосому будем задавать в виде графа переходов конечного автомата. Такое задание хромосомы определяется тем, что в рассматриваемой задаче переходы помечаются только одной входной переменной. При этом гены хромосомы – компоненты графа переходов (например, значения входной переменной).

Общая схема генетического алгоритма

На рис. 4 приведена общая схема работы генетического алгоритма [2]. В данной работе в качестве стратегии отбора особей для построения следующего поколения используется отбор отсечением [8]. При применении такой стратегии отбора родительские решения выбираются из группы лучших решений текущего поколения. Размер этой группы (порог отсечения) обычно составляет от 1/100 до 1/3 от размера поколения. Все автоматы из группы лучших решений имеют одинаковые шансы быть выбранными в качестве родительских особей.

Рис. 4. Общая схема генетического алгоритма

Приведем описание генетического алгоритма, использующего отбор отсечением.

1. Начальное поколение заполняется случайными решениями.
2. Начальное поколение решений становится текущим.

3. Из текущего поколения отбирается группа лучших решений.
4. Формируется новое поколение решений:
   a. Создается пустое новое поколение.
   b. Из группы лучших решений, выбранных на основе порога отсечения, случайным образом выбирается некоторая пара.
   c. Формируется новое решение с помощью применения оператора скрещивания [2] к двум выбранным решениям.
   d. К новому решению *последовательно* применяются операторы мутации [2].
   e. Полученное решение добавляется в новое поколение решений.
   f. Если размер нового поколения меньше размера текущего поколения, то переходим к пункту b.
5. Новое поколение становится текущим.
6. Если число созданных поколений меньше заданного пользователем, то переходим к шагу 3.

В приведенном алгоритме на шаге 4, d, указано, что к новому решению последовательно применяются операторы мутации, в качестве которых выступают *n*-точечный оператор и предлагаемый авторами оператор мутации – сортировка состояний в порядке использования.

Оператор скрещивания

В данной работе первым используется *n*-точечный оператор *скрещивания*. Для этого оператора задается вероятность, с которой он применения к каждому гену в хромосоме. Приведем описание алгоритма работы оператора скрещивания.

1. В качестве нового автомата формируется копия первого из выбранных решений (шаг 4, b приведенного выше алгоритма).
2. Осуществляется цикл по всем его состояниям. Для каждого состояния:
   a. Выполняется цикл по всем переходам, исходящим из состояния. Для каждого перехода:
      i. Определяется с заданной вероятностью, требуется ли изменить номер состояния, в которое переходит автомат. Если это требуется, то он изменятся на номер состояния из соответствующего перехода второго автомата.
      ii. Определяется с заданной вероятностью, требуется ли изменить значение выходной переменной, генерируемое при переходе автомата. Если это требуется, то значение указанной переменной заменяется значением выходной переменной, полученной из соответствующего перехода второго автомата.

N-точечный оператор мутации

Этот оператор применяется к каждому гену в хромосоме с некоторой заранее заданной вероятностью, в отличие от обычного оператора мутации. Такой оператор может изменить сразу несколько генов. При этом отметим, что значения этой вероятность и вероятности используемой в операторе скрещивания в общем случае отличаются. Приведем описание алгоритма работы рассматриваемого оператора мутации.

3. Осуществляется цикл по всем состояниям автомата:
   a. Выполняется цикл по всем переходам из состояния:
      i. С заданной вероятностью определяется необходимость изменение номера состояния, в которое автомат переходит.
      ii. Если требуется изменить номер состояния, то он изменяется на номер состояния, выбранный случайным образом.

iii. С заданной вероятностью определятся необходимость изменения значения выходной переменной, генерируемой автоматом при переходе.
iv. Если требуется изменить значение выходной переменной, то оно изменяется на одно из возможных значений, выбранное случайным образом.

Новый оператор мутации – сортировка состояний в порядке использования

В большинстве решений, которые перебирает генетический алгоритм, автомат в процессе работы не переходит в часть состояний, число которых в задается исходно. На поведение автомата влияют только те состояния, из которых осуществляется переход по дуге хотя бы один раз. Далее такие состояния будем называть *используемыми*, а все остальные состояния – *неиспользуемыми*.

Приведем алгоритм сортировки состояний в порядке их использования.

1. Создается пустой словарь пар номеров [«старый номер состояния» – «новый номер состояния»].
2. Моделируется полет вертолета. Перед каждым переходом по дуге, если в словаре нет пары, в которой первый элемент равен текущему номеру состояния, то в него добавляется пара [текущий номер состояния – число пар в словаре].
3. Выполняется цикл по всем состояниям автомата. Для каждого состояния, если в словаре нет пары, в которой первый элемент равен номеру состояния, то в него добавляется пара [номер состояния – число пар в словаре].
4. Согласно словарю изменяется порядок состояний.

Состояния, для которых в словарь добавляются пары на шаге 2 – *используемые состояния*. Состояния, для которых в словарь добавляются пары на шаге 3 – *неиспользуемые состояния*.

Построенный автомат в большинстве случаев может быть упрощен за счет исключения *неиспользуемых* состояний и замены переходов в них переходами в начальное состояние (выполняется при необходимости компактного представления).

Описание программы

Для проведения экспериментов по генерации автопилотов была написана программа на языке *Java*, исходные коды которой будут приведены на сайте *http://is.ifmo.ru* в разделе «Статьи». Программа позволяет задавать время полета, верхний предел числа состояний автомата, размер поколения, максимальное число поколений, число секторов, вероятности применения для *n*-точечных операторов мутации и скрещивания. Также в ней можно изменить порог для стратегии отбора отсечением. При проведении экспериментов при необходимости можно изменять расположение целей вертолета.

Для кодирования графов переходов автоматов используется три класса: `StateMachine`, `State` и `Branch`. В качестве главного класса автомата применяется класс `StateMachine`. Классы `State` и `Branch` реализуют его состояния и переходы соответственно. Массив `states` в классе `StateMachine` содержит состояния автомата автопилота. Поле `currentStateIndex` используется для хранения номера текущего состояния автомата. В массив `brunches` класса `State` включены переходы из данного состояния. Поле `stateIndex` в классе `Branch` содержит номер состояния, в которое переходит автомат. В массиве `outputs` хранятся значения выходных переменных, генерируемых при переходе.

При запуске программа создает два файла. В первом из них содержится траектория полета вертолета для автопилота, построенного с помощью генетического алгоритма. Во второй файл выводится таблица переходов и выходов для лучшего автомата. Перед формированием таблицы производится сортировка состояний автомата в порядке использования и удаляются *неиспользуемые состояния*.

Эксперименты

Все эксперименты проводились при размере поколения 300 и пороге отсечения 90. Максимальное число поколений – 200. Вероятность применения оператора скрещивания к переходу автомата – 0.04, а вероятность применения *n*-точечного оператора мутации – 0.02. Число состояний автомата не больше 15. Время полета – 200.

На рис. 5 изображены двадцать целей, пронумерованных в том порядке, в котором через них должен пролететь вертолет. Точка с координатами (0; 0) является исходной. В начале движения вертолет ориентирован вправо и движется в этом направлении с минимальной возможной скоростью. При проведении экспериментов изменялось число секторов обзора вертолета (четыре или шесть), а также использовалась или не использовалась сортировка состояний. Эксперименты с каждым набором этих параметров алгоритма проводились по 10 раз.

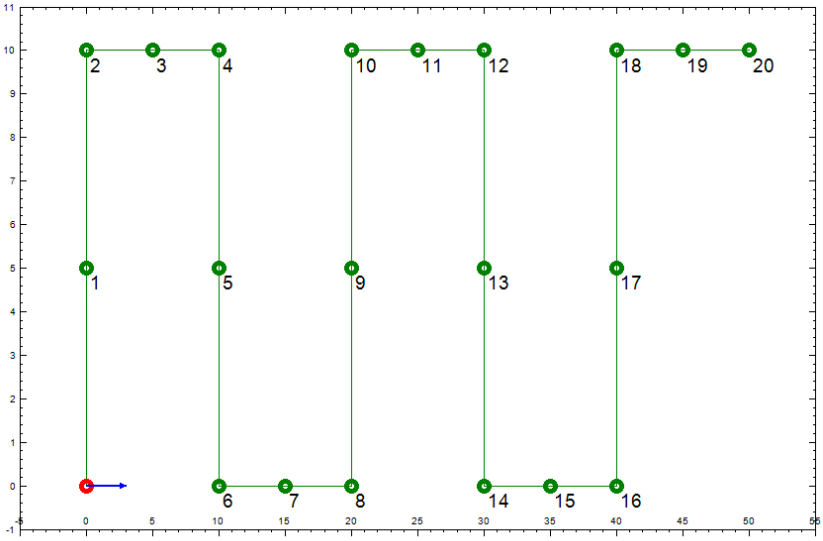


Рис. 5. Цели вертолета

Таблица 1. Результаты экспериментов

Число секторов	Сортировка состояний	Результат		
		худший	средний	Лучший
4	-	12	14.1	16
	+	12	14.7	17
6	-	11	15.6	18
	+	14	16.6	18

В табл. 1 приведены результаты экспериментов. Под результатом эксперимента понимается число целей, через которые пролетел вертолет с автопилотом, построенным с помощью предложенного генетического алгоритма. Столбец «худший» – худший результат из десяти экспериментов, колонка «лучший» – лучший результат этих экспериментов, а

столбец «средний» – усредненный результат. Из табл. 1 видно, что использование алгоритма сортировки состояний улучшает эффективность работы генетического алгоритма.

Таблица 2. Лучший автопилот

	1			2			3			4			5			6		
	s	z0	z1	s	z0	z1	s	z0	z1	s	z0	z1	s	z0	z1	s	z0	Z1
1	2	1	-1	6	0	1	2	-1	-1	12	0	1	2	1	-1	10	1	0
2	8	-1	1	1	0	1	10	1	1	3	1	0	1	1	-1	3	-1	0
3	11	0	0	4	-1	1	3	-1	1	3	1	-1	7	-1	0	8	1	0
4	7	0	0	3	0	1	5	-1	1	2	-1	0	2	1	0	10	-1	-1
5	10	0	-1	2	0	-1	6	1	1	1	1	-1	2	0	0	2	0	1
6	7	0	-1	10	0	0	3	1	1	2	1	-1	1	-1	0	1	-1	-1
7	1	1	0	8	0	-1	8	0	1	7	1	-1	7	-1	-1	1	-1	-1
8	3	-1	-1	1	0	0	6	0	1	9	0	-1	11	0	1	4	0	0
9	1	1	0	7	-1	1	6	0	1	4	0	-1	12	-1	1	7	1	-1
10	1	-1	1	6	-1	-1	7	-1	1	11	-1	1	1	0	0	7	0	0
11	1	1	-1	6	0	0	7	-1	1	11	-1	1	1	0	0	7	0	0
12	1	0	-1	4	1	1	6	0	1	9	0	-1	4	1	-1	12	0	0

Приведем таблицу переходов и выходов автомата для лучшего автопилота (табл. 2). Столбцам этой таблицы соответствуют номера секторов обзора, строкам – состояния автомата. Каждый столбец состоит из трех колонок: колонка `s` – новое состояние, в которое перейдет автомат, колонки `z0`, `z1` – значения выходных переменных. На рис. 6 изображена траектория полета вертолета, управляемого лучшим из построенных автопилотов. Авторами также был выполнен эксперимент по использованию восьми секторов обзора вертолета, однако, несмотря на увеличение времени работы генетического алгоритма, существенного улучшения работы автопилота не наблюдалось

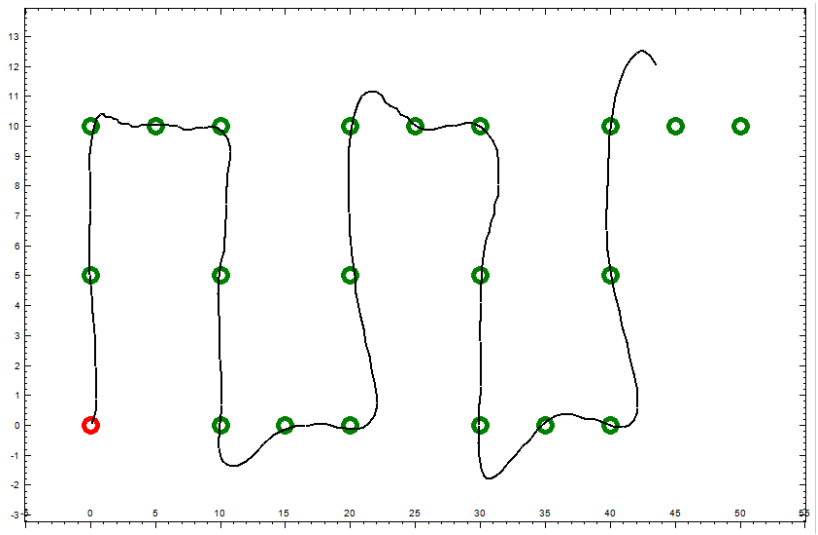


Рис. 6. Траектория полета лучшего вертолета

Заключение

В работе предложен генетический алгоритм, позволяющий *автоматически* построить автопилот для упрощенной модели вертолета. Разработана программа на языке

Java, реализующая указанный алгоритм и позволяющая проводить эксперименты с моделями вертолета и окружающей среды. Предложен дополнительный оператор мутации, позволяющий улучшить генетический алгоритм. Приведены экспериментальные данные, подтверждающие эффективность этого алгоритма.

Построен автопилот с 12 состояниями, который проводит вертолет через первые 18 целей из 20 за отведенное время. Установлено, что при увеличении времени полета на 33 условных единицы полученный автопилот обеспечит прохождение всех 20 целей.

Литература

1. Шалыто А.А. Технология автоматного программирования / Труды первой Всероссийской конференции «Методы и средства обработки информации». – М.: МГУ. 2003. – Режим доступа: http://is.ifmo.ru/works/tech_aut_prog
2. Koza J. R. Genetic programming. On the Programming of Computers by Means of Natural Selection. – The MIT Press, 1998. – Режим доступа: www.ru.lv/~peter/zinatne/ebooks/MIT%20-%20Genetic%20Programming.pdf
3. Mitchell M. An Introduction to Genetic Algorithms. Cambridge. – MA.: MIT Press, 1996.
4. Лобанов П.Г., Шалыто А.А. Использование генетических алгоритмов для автоматического построения конечных автоматов в задаче о флибах // Известия РАН. – Теория и системы управления. – 2007. – № 5. – С127–136.
5. Wolfram S. A New Kind of Science. Champaign. – Wolfram Media, 2002.
6. Mitchell M., Crutchfield J., Hraber P. Evolving cellular automata to perform computations // Physica D. – 1993. – V. 75. – P. 361–391.
7. Бедный Ю.Д. Применение генетических алгоритмов для решения одной задачи на клеточных автоматах. Задача классификации плотности для клеточных автоматов. – Бакалаврская работа / СПбГУ ИТМО. – 2006. – Режим доступа: http://is.ifmo.ru/papers/genalgcelaut/
8. Jefferson D., Collins R., Cooper C., Dyer M., Flowers M., Korf R., Taylor C., Wang A. The Genesys System: Evolution as a Theme in Artificial Life. // Artificial Life II: Proceedings of the Workshop on Artificial Life. – NJ: Addison-Wesley, 1992. – P. 549–578. – Режим доступа: www.cs.ucla.edu/~dyer/Papers/AlifeTracker/Alife91Jefferson.html

УДК 004.4’242

СОЗДАНИЕ СИСТЕМЫ УПРАВЛЕНИЯ ТАНКОМ ДЛЯ ИГРЫ *ROBOCODE* С ИСПОЛЬЗОВАНИЕМ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ

Ю.Д. Бедный, А.А. Шалыто

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В работе предлагается метод создания системы управления танком для игры Robocode, основанный на применении генетических алгоритмов, что позволяет автоматизировать процесс построения системы управления. Кроме того, рассматривается способ улучшения генерируемых систем управления за счет использования более сложного алгоритма управления, реализованного с помощью конечного автомата.

Ключевые слова: генетические алгоритмы, конечный автомат, Robocode, автоматное программирование

Введение

В работе предлагается метод автоматизации построения системы управления танком для популярной компьютерной игры *Robocode* [1]. Этот метод позволяет на основе

использования генетических алгоритмов [2, 3] строить достаточно простые системы управления танком. Работа *не* ставит своей целью создание танка, способного побеждать танки, построенные вручную, которые являются наилучшими из известных на сегодняшний день (например, танк `voidious.Dookious`). Алгоритмы, используемые в таких танках, являются достаточно сложными, и их реализация занимает сотни килобайт, а работа по их созданию крайне трудоемка.

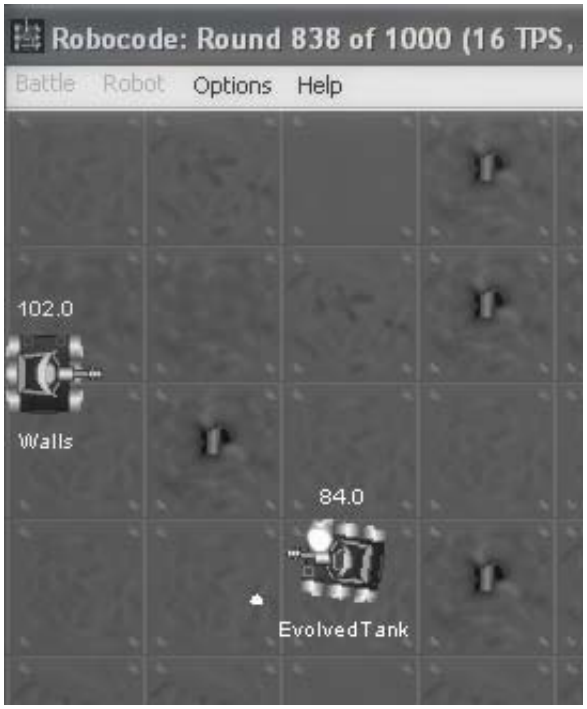


Рис. 1. Пример боя в игре Robocode

Танки, создаваемые предлагаемым методом, должны побеждать в индивидуальном сражении каждого *заранее выбранного соперника* из тестового набора, который включает в себя танки, поставляемые вместе с игрой, а также танк `newCynic.Cynical` [4]. Выбор последнего был связан с тем, что этот танк (как и прототип, на основе которого он построен [5]), в отличие от многих других танков, был предварительно спроектирован, и для него существует достаточно подробная открытая проектная документация. Для каждого танка из тестового набора с помощью предлагаемого метода генерируется танк, который его побеждает. Предлагаемый метод позволяет генерировать танки, системы управления которых построены как с применением автоматного подхода [6], так и без его использования. Если автоматы не используются, то система управления описывается только одной функцией, формирующей выходные воздействия на объект управления. При автоматном подходе для каждого состояния автомата определена соответствующая функция управления.

На рис. 1 в качестве примера приведен внешний вид среды игры *Robocode* во время одного из боев. На этом рисунке разработанный танк назван *EvolvedTank*.

Обзор

Известно большое число танков для рассматриваемой игры [7]. Наиболее простые из них реализованы всего несколькими строками кода, а программы, реализующие наиболее сложные танки, как отмечалось выше, занимают сотни килобайт. Большинство

танков создано вручную. Существуют также работы [8, 9], в которых танки строятся не вручную, а автоматически – с применением методов искусственного интеллекта.

В работе [8] использовано несколько таких методов (обучение с подкреплением, нейронные сети и генетические алгоритмы) для управления различными системами танка (радаром, системой движения и пушкой). Генетические алгоритмы в работе [8] применялись для создания систем управления радаром и движением танка. Однако их использование не привело к желаемому результату – система управления радаром осуществляла его поворот на 360°, а система передвижения задавала перемещение танка по кругу. В работе [9] используется сравнительно сложный способ представления системы управления в виде особи генетического алгоритма – программы на языке *TableRex*. При этом длина программы составила 7776 бит.

Постановка задачи

Устройство танка схематично показано на рис. 2. Необходимо управлять следующими устройствами: радаром (`Radar`), пушкой (`Gun`) и системой движения (`Body`).

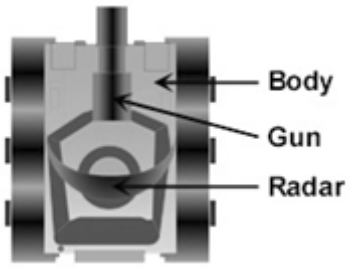


Рис. 2. Схематичное изображение танка

Управление танком осуществляется следующим образом. В каждый момент времени анализируется текущая ситуация (положение танка, его скорость и т.д.). На основе этой информации формируются четыре действия: передвижение (вперед/назад), поворот, поворот пушки, стрельба. Указанные действия формируются в результате интерпретации функции управления, которая генерируется на основе генетических алгоритмов.

Перейдем к формальной постановке задачи. Обозначим текущее время в игре как $t \in N$, а максимальное время, которое возможно для данной игры – M. Обозначим множество позиций как S, а множество действий танка как A. *Позиция* – элемент некоторого множества всех возможных ситуаций в игре в выбранный момент времени. Позиция включает в себя положение каждого танка, его скорость, направление, угол поворота пушки и радара, энергию, информацию о танке соперника и т. д. Задача управления для рассматриваемой игры в общем случае состоит в задании для каждого момента времени t функции управления $f_t : S^t \to A$, которая на основании информации о *позициях* во все предыдущие моменты времени (получаемой из среды *Robocode*) определяет действие на объект управления в текущий момент времени. Таким образом, решение задачи управления танком – вектор $[f_1, f_{2,\ldots,} f_M]^T$.

В данной работе решается упрощенная задача управления:

- действие танка в текущий момент времени зависит только от *позиции* в этот момент времени;
- зафиксирован алгоритм управления радаром – вращение по кругу;

- при выборе действия анализируется лишь упрощенная *позиция* – элемент множества $S' = \{< x,\ y,\ dr, tr, w, dh, GH, h, d, e, E >\}$. Здесь:
  - x, y – координаты танка соперника относительно нашего танка;
  - dr – расстояние, которое осталось «доехать» нашему танку (после вызова метода `AdvancedRobot.setAhead`);
  - tr – угол, на который осталось повернуться нашему танку;
  - w – расстояние от нашего танка до края поля;
  - dh – угол между направлением на танк соперника и пушкой нашего танка;
  - GH – угол поворота пушки нашего танка;
  - h – направление движения танка соперника;
  - d – расстояние между нашим танком и танком соперника;
  - e – энергия танка соперника;
  - E – энергия нашего танка;
- множество действий $A' = \{g, p, d, h\}$, где g – угол поворота пушки, p – энергия снаряда (неположительные значения означают, что выстрел не производится), d – расстояние, на которое перемещается танк, h – угол поворота танка.

Учитывая изложенное, задача построения системы управления танком сведена к заданию функции $f : S' \to A'$. Эту функцию будем искать *автоматически* с использованием генетических алгоритмов, а точнее, с помощью их разновидности, обладающей высокой эффективностью – программированием с экспрессией генов (*Gene Expression Programming*) [10]. В отличие как от стандартных генетических алгоритмов, использующих строки фиксированной длины, так и от генетического программирования [11], использующего деревья, хромосомы в генетическом программировании с экспрессией генов представляются в виде строк фиксированной длины, которые преобразуются в так называемые *Karva*-деревья. Генетические операции с такими деревьями (в отличие от генетического программирования) *корректны по определению*. За счет этого при использовании программирования с экспрессией генов не приходится тратить время на проверку корректности деревьев, получаемых в результате скрещивания и мутаций. Это обеспечивает используемому подходу указанные преимущества перед генетическим программированием.

Для решения поставленной задачи с помощью генетических алгоритмов (и, в частности, для программирования с экспрессией генов) необходимо определить способ представления функции в виде хромосомы (особи популяции генетического алгоритма), выбрать функцию приспособленности и определить генетические операции (мутация, скрещивание и отбор).

Как было отмечено во введении, предлагаемый метод позволяет строить системы управления, как с использованием автоматного подхода, так и без его применения. Ниже эти подходы рассматриваются отдельно.

Построение системы управления без применения автоматов

Сначала опишем подход, не предполагающий использования автоматов при создании системы управления.

Общая схема генетических алгоритмов

Схема одного шага работы генетических алгоритмов приведена на рис. 3.

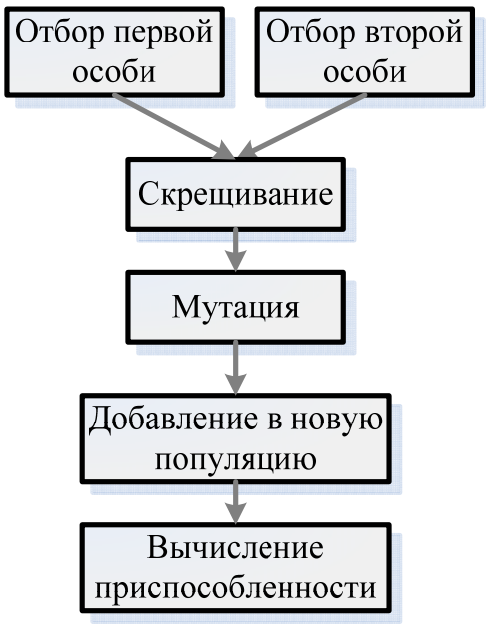


Рис. 3. Один шаг работы генетических алгоритмов

Представление в виде хромосомы

Для описания в виде особи генетического алгоритма искомой функции управления $f : S' \to A'$ предлагается представлять ее в виде четырех функций f_i, каждая из которых возвращает вещественное число: f_1 – угол поворота пушки, f_2 – энергия снаряда, f_3 – расстояние, на которое перемещается танк, f_4 – угол поворота танка. В свою очередь, функция f_i представляется в виде *Karva*-дерева – стандартного способа представления для программирования с экспрессией генов. Таким образом, хромосома состоит из четырех строк – линеаризованных *Karva*-деревьев, для представления которых применяются описываемые ниже функции и терминалы.

Требования, предъявляемые к набору базовых функций, используемых при задании хромосомы, таковы: набор должен быть достаточно полным, чтобы выразить необходимые зависимости действий от позиции, и не слишком избыточным, чтобы работа генетического алгоритма была эффективной.

Предлагается использовать следующий набор базовых функций: $+(x, y) = x + y$, $++(x, y, z) = x + y + z$, $n(x) = –x$, $*(x, y) = xy$, $**(x, y, z) = xyz$, min, $s(x) = \frac{1}{1+e^{-x}}$, $if>(x, y, z, w) = x > y \ ? \ z : w$. Набор терминалов (координаты танка (`x`, `y`), его энергия (`e`) и т.д.) строится исходя из определения позиции (элемента множества S'). Кроме того, этот набор необходимо дополнить некоторым множеством констант. В настоящей работе были выбраны следующие константы: 0, .5, 1, 2, 10.

В табл. 1 приведен пример сгенерированной хромосомы, а в табл. 2 – соответствующая этой хромосоме функция управления $f = (f_1, f_2, f_3, f_4)$, представленная через базовые функции.

На рис. 4 приведен пример *Karva*-дерева для функции *f1* из табл. 3.

Таблица 1. Представление функции в виде хромосомы

f1	4 0 7 1 17 4 5 2 4 1 18 9 10 13 12 10 13 22 23 11 17 21 8 15 16 14 21 13 16 18 22 10 16 19 13 8 21 13 10 8 23
f2	6 4 5 8 0 2 7 0 5 0 20 15 8 19 19 18 8 11 15 16 13 22 23 22 12 20 22 19 13 19 23 15 15 22 11 22 11 14 8 17 23
f3	7 2 1 3 6 5 6 0 6 7 21 16 22 19 21 17 19 23 9 13 17 13 14 15 9 21 12 16 21 14 16 23 16 15 23 16 8 21 15 14 22
f4	2 4 5 0 2 7 1 3 4 1 19 8 15 17 17 9 18 13 13 18 18 21 17 17 17 15 8 9 18 22 17 23 19 21 10 10 11 20 13 19 23

Таблица 2. Функции, соответствующие хромосоме

Функция $f = (f_1, f_2, f_3, f_4)$
*f1 = *(s[n(*(.5, e))], if>[10, *(n(E), dh), *(y, dr, .5), +(w, dr)])*
f2 = min([x, s(*(x, tr, 1))], *[+(s(2), h, if>(1, x, GH, GH), s(dh)])*
f3 = if>(+[(GH, d, 10), min(GH, E)], n[s(y)], +[min(.5, 10), if>(.5, 0, 1, y), d], min[2, e])*
f4 = +([s(*(dh, dh)), +(n(d), GH)], *[if>(x, 1, 10, 10), n(y), +(dh, .5, .5)])*

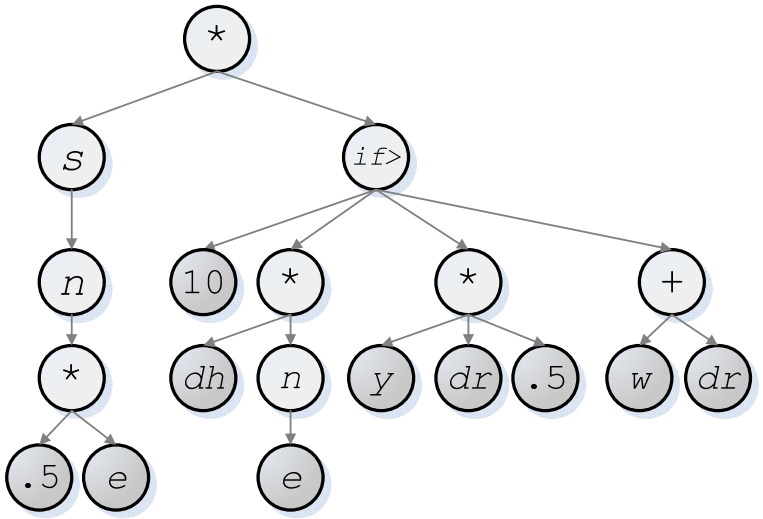


Рис. 4. *Karva*-дерево функции *f1* из табл. 3

Функция приспособленности и критерий остановки процесса эволюции

Опишем, как была выбрана функция приспособленности. Она отражает результат поединков против выбранного соперника. При этом отметим, что, так как результат поединка существенно зависит от начального положения танков, то проводится не один поединок, а двадцать. Увеличение числа сражений позволило бы получить более точный результат, однако это было бы связано с возрастанием времени вычисления функции приспособленности, которое и так велико.

Для того чтобы уравнять шансы соперников, выбор начальных положений танков производится следующим образом. Создается случайный набор пар координат ((*x1*, *y1*), (*x2*, *y2*)). После этого для каждой из них проводится два поединка, в первом из которых танк соперника имеет начальные координаты (*x1*, *y1*), а во втором – (*x2*, *y2*). Обозначим

число очков, набранных нашим танком в бою (состоящим из 20 раундов) как *s*, а число очков, набранных его соперником, как *se*. При этом приспособленность танка будем вычислять следующим образом: $f = 100\frac{s}{s + s_e}$.

Процесс эволюции продолжается до тех пор, пока максимальное значение функции приспособленности для элементов популяции не превзойдет некоторого порогового значения. Например, пороговое значение, равное 70, означает, что танк в среднем «научился» побеждать своего соперника. При этом отметим, что значение функции приспособленности, равное 50, означает, что наилучший из танков *текущей популяции* в ходе двадцати раундов при случайном начальном положении танков, выигрывает половину раундов.

Генетические операции

Для реализации генетического алгоритма в данной работе используются стандартные генетические операции: отбор, мутация и скрещивание. Отбор осуществляется по методу рулетки. Скрещивание хромосом (как наборов из четырех строк, например, приведенных в табл. 2) производится поэлементно. При этом для каждой пары строк с вероятностью *p* в качестве *i*-го элемента результатом скрещивания выбирается *i*-й элемент либо первой, либо второй хромосомы, а с вероятностью (1 – *p*) скрещивание производится стандартным образом с использованием метода битовой маски. Аналогично скрещиванию, мутация выполняется поэлементно. При этом каждый из символов строки с некоторой вероятностью заменяется некоторым другим. Более подробное описание указанных операций приведено в работе [2].

Эксперименты

При реализации генетического алгоритма число особей в популяции было выбрано равным 50. Один шаг эволюции (скрещивание, мутация, отбор, вычисление функции приспособленности) в популяции такого размера выполняется около пяти минут на компьютере *Intel Pentium M* 1.86 *GHz*. При этом практически все время тратится на вычисление функции приспособленности. Данное обстоятельство не позволяет использовать популяции б*о*льших размеров. Так, например, даже при указанном числе особей время работы генетического алгоритма может занимать несколько часов.

На рис. 5 приведен пример графика, иллюстрирующего работу генетического алгоритма для генерации танка против соперника `sample.Walls`. По оси абсцисс отложен номер поколения, а по оси ординат – значение функции приспособленности (`Max` – максимальное значение среди особей популяции, а `Avg` – среднее значение). Из рассмотрения графика максимального значения функции приспособленности (`Max`) следует, что уже на 11-ом поколении эволюцию можно было остановить, так как максимальное значение функции приспособленности превысило пороговое значение 70. Однако процесс был остановлен после 23-го поколения, так как значение функции приспособленности перестало изменяться существенно.

Таким образом, в результате сражений с одним из танков тестового набора строится каждый танк. В табл. 3 приведены описания систем управления, которые сгенерированы в ходе сражений с танками соперников `newCynic.Cynical` и `sample.Walls`. Эти танки использовались и в сражениях, результаты которых перечислены в табл. 4, так как указанные танки являются наиболее сильными в тестовом наборе.После генерации каждого танка, для оценки его эффективности были проведены бои с тем же соперником,

против которого он выводился. Бои состояли из 100 раундов. В табл. 4 приведены результаты этих боев. В этой таблице в столбце «Счет» первое значение – число очков, набранных нашим танком, а второе – танком соперника. В общем случае, как отмечалось выше, увеличение числа раундов приводит к более объективным результатам.

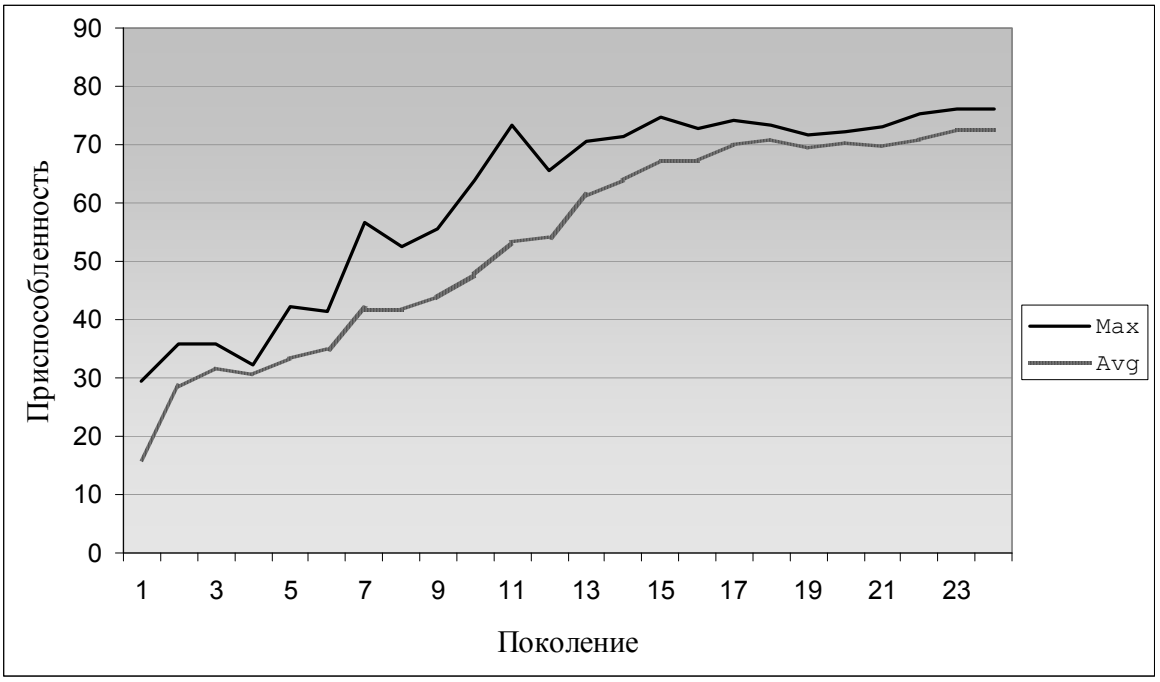


Рис. 5. Процесс эволюционного построения танка

Таблица 3. Представление функции управления сгенерированных танков

Соперник	Описание функции
new-Cynic.Cynical	0 2 1 6 3 6 0 6 13 19 14 13 23 10 28 18 13 17 18 13 19 14 17 23 15 13 18 9 29 30 15 14 13
	0 7 2 0 4 6 1 7 9 22 23 11 23 22 16 23 18 27 10 22 27 16 9 13 15 21 24 29 14 17 16 20 23
	3 4 8 1 6 0 0 5 14 17 21 24 21 17 19 11 19 11 24 18 28 23 23 15 10 24 16 22 14 30 26 14 15
	7 3 3 1 8 0 1 2 18 23 29 15 26 29 16 25 30 25 16 23 26 26 27 17 18 26 19 25 15 10 19 28 20
sample.Walls	7 0 6 8 2 3 8 6 0 5 23 13 18 9 12 20 23 14 20 27 29 12 14 26 14 10 13 26 10 17 22 26 21 18 14 30 14 21 20 23 9
	4 4 4 4 1 6 1 2 2 2 9 22 19 19 29 25 21 26 17 17 14 27 13 14 21 16 22 30 23 21 29 14 21 19 15 9 9 19 22 9 20
	24 0 3 2 1 5 7 1 2 1 16 11 21 15 25 18 25 30 18 17 21 9 25 19 26 10 28 28 13 12 11 28 28 18 14 14 14 26 11 30 28
	2 5 6 6 5 13 6 5 8 12 23 11 28 22 21 14 16 15 21 15 29 29 19 23 14 20 24 27 14 14 22 26 26 10 12 26 15 28 10 15 22

Таблица 4. Результаты поединков (100 раундов)

Соперник	Счет
newCynic.Cynical	10933 : 8108
sample.Walls	9559 : 7240
sample.SpinBot	11414 : 8556
sample.MyFirstRobot	11964 : 5346
sample.Corners	18086 : 2971
sample.Crazy	10797 : 4278
sample.Fire	17610 : 5500
sample.Tracker	18642 : 4844
sample.TrackFire	16269 : 9851
sample.Target	17968 : 5
sample.RamFire	18572 : 3089

Автоматный подход

Эвристическое построение автоматов, управляющих танком, применялось в работе [4, 5]. В то же время попыток автоматически построить автоматы для управления танком ранее не предпринималось.

В рассмотренном выше подходе управление танком в любой момент времени осуществляется единственной функцией $f : S' \to A'$. При использовании автоматов эта функция декомпозируется с помощью состояний следующим образом. В каждом состоянии автомата определим функцию $f_i : S' \to A'$, где i – номер рассматриваемого состояния. Для каждой пары различных состояний с номерами (i, j) зададим *функцию перехода* $f_{ij} : S' \to R$. Построение функций переходов выполняется с помощью генетических алгоритмов, также как это делается для компонент функции управления. При этом, функции представляются в виде *Karva*-деревьев.

Выделим начальное состояние. Далее в каждый момент времени, находясь в состоянии i, последовательно выполняются следующие действия:

- вычисляются значения функций f_{ij} для всех j, $i \neq j$. Как только одно из вычисленных значений оказывается больше нуля $(f_{ij} > 0)$, выполняется переход в состояние j. Если же все $f_{ij} \leq 0$, то состояние не изменяется;
- определяется значение *функции управления* f_i, и выполняется действие, соответствующее найденному значению.

Для представления автомата в виде особи генетического алгоритма предлагается хранить в хромосоме (по аналогии с подходом, описанным выше) 4 * N + N * (N – 1) функций, где N – число состояний автомата. Скрещивание, мутация и отбор выполняются таким же образом, как и ранее. На рис. 6 приведен пример автомата с тремя состояниями, управляющего танком. Табл. 5 иллюстрирует процесс эволюционного построения танка с автоматной структурой. Эволюция была остановлена на 36-ом поколении, так как рост максимального значения функции приспособленности практически остановился.

В табл. 6 приведен пример описания автомата из двух состояний в виде особи генетического алгоритма. Это описание содержит десять строк: восемь (*f1 – f8*) определяются состояниями и две (*f9*, *f10*) – переходами. Данный автомат выводился в результате боев с танком sample.Walls.

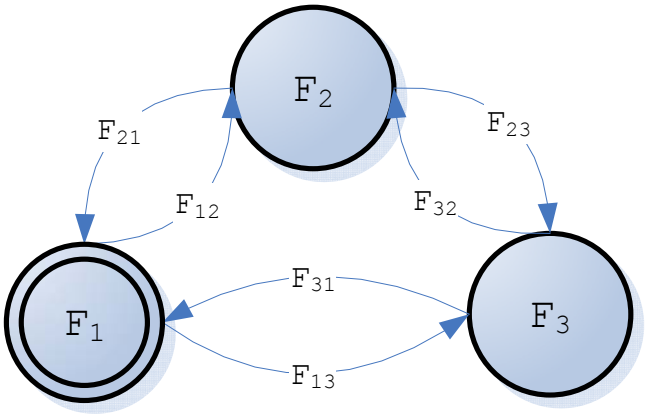


Рис. 6. Автомат управления танком

Таблица 5. Работа генетического алгоритма (процесс эволюции)

Номер поколения	Максимальное значение функции приспособленности	Среднее значение функции приспособленности	Время, с
0	31.15	15.70	122
1	36.71	26.42	138
2	34.93	28.91	152
3	40.30	32.96	168
…	…	…	…
9	52.25	43.74	261
10	55.53	41.15	358
…	…	…	…
21	64.49	59.14	599
22	69.44	60.18	606
23	91.38	62.55	631
24	89.45	72.68	621
…	…	…	…
36	93.73	91.48	778

Таблица 6. Представление функции управления сгенерированных танков

f1	5 2 2 5 6 7 6 1 4 4 8 20 9 8 12 23 22 22 16 13 14 10 12 9 14 16 9 18 11 11 23 23 19 18 23 13 15 16 16 22 14
f2	3 16 2 5 1 2 0 6 3 5 15 8 17 12 22 14 21 11 10 12 13 21 14 14 8 19 11 20 17 15 23 14 22 18 8 10 9 21 20 18 22
f3	3 7 14 3 7 5 4 16 4 1 17 16 9 20 16 22 12 10 17 9 11 8 19 19 16 12 11 8 15 20 9 23 20 19 18 10 19 21 13 10 18
f4	1 7 14 6 2 4 7 6 4 6 9 8 12 21 17 10 13 16 23 18 17 14 15 15 19 17 18 14 21 9 15 17 12 19 11 8 13 9 23 10 18

f5	4 0 7 1 17 4 5 2 4 1 18 9 10 13 12 10 13 22 23 11 17 21 8 15 16 14 21 13 16 18 22 10 16 19 13 8 21 13 10 8 23
f6	6 4 5 8 0 2 7 0 5 0 20 15 8 19 19 18 8 11 15 16 13 22 23 22 12 20 22 19 13 19 23 15 15 22 11 22 11 14 8 17 23
f7	7 2 1 3 6 5 6 0 6 7 21 16 22 19 21 17 19 23 9 13 17 13 14 15 9 21 12 16 21 14 16 23 16 15 23 16 8 21 15 14 22
f8	2 4 5 0 2 7 1 3 4 1 19 8 15 17 17 9 18 13 13 18 18 21 17 17 17 15 8 9 18 22 17 23 19 21 10 10 11 20 13 19 23
f9	20 6 4 3 5 5 3 0 6 2 11 8 22 15 15 19 23 14 20 10 23 17 18 20 17 13 11 20 20 14 18 15 11 10 12 11 8 8 13 21 12
f10	3 7 0 4 1 4 5 1 4 16 22 15 8 14 22 19 18 17 22 10 11 15 21 13 9 13 11 8 8 13 16 11 14 8 10 23 8 23 9 17 18

Сгенерированный танк в 1000 раундах побеждает танк соперника со счетом 180168 на 28278. Таким образом, в среднем наш танк побеждает противника с вероятностью около 86%. Это несколько меньше, чем полученное в табл. 5 максимальное значение функции приспособленности – 93.73. Данное обстоятельство объясняется тем, что результат боя из 20 раундов имеет большую погрешность из-за сравнительно небольшого числа экспериментов.

Применение автоматного подхода для описания поведения танка позволяет генерировать более сложные системы управления по сравнению с первым подходом, и, следовательно, создавать более сильные танки. Однако на практике полученные танки обладают примерно такой же эффективностью, что и танки, построенные без автоматов. Это, видимо, связано с тем, что пространство поиска (множество автоматов с фиксированным числом состояний) достаточно велико для того, чтобы была возможность генерировать «хорошие танки» за разумное время в условиях медленного вычисления функции приспособленности. Можно надеяться, что дальнейшая работа в этом направлении (например, ускорение вычисления функции приспособленности) позволит более эффективно использовать автоматный подход при построении танков с применением генетических алгоритмов.

Программная поддержка метода

Управление танком может осуществляться либо синхронно, либо асинхронно. В первом случае главный класс программы наследуется от библиотечного класса `Robot`, а во втором – от библиотечного класса `AdvancedRobot` [1]. В данной работе был выбран второй вариант. Таким образом, приведенный в приложении код, принадлежит классу, который наследуется от класса `AdvancedRobot`. Метод поддержан двумя типами программ. Первая из них на основе использования генетических алгоритмов строит символическое описание системы управления, пример которого приведен в табл. 2, а вторая – интерпретирует ее и управляет танком (приложение). Первая программа имеет две модификации, соответствующие традиционному и автоматному подходам, а вторая не зависит от используемого подхода и состоит из следующих основных компонентов:

- среда *Robocode*;
- система управления, которая по входным воздействиям, получаемым из среды, формирует выходные воздействия на объект управления;
- объект управления (танк), содержащий систему движения и пушку. Радар реализован тривиально, и поэтому в отдельный класс не выделен.

Эта программа является главной и содержит обращение к некоторым классам, исходный код которых не приведен в приложении. Обе программы написаны на языке *Java*.

Заключение

В результате выполнения настоящей работы предложен метод построения системы управления танком для игры *Robocode*. Достоинство этого метода состоит в том, что при его использовании, построение системы управления происходит автоматически. Структуры генерируемых систем управления достаточно просты, но в то же время танки с этими системами управления побеждают любой фиксированный танк из тестового набора. Предложенный метод также позволяет генерировать системы управления с использованием автоматов.

В заключение отметим, что предлагаемый метод не позволяет генерировать танки, побеждающие наиболее сильных из известных танков, так как в последних используются эвристически созданные сложные и эффективные алгоритмы управления.

Литература

1. Официальный сайт игры Robocode. http://robocode.sourceforge.net/
2. Mitchell M. An Introduction to Genetic Algorithms. MA. The MIT Press. 1996.
3. Гладков Л.А., Курейчик В.В., Курейчик В.М. Генетические алгоритмы. – М.: Физматлит,2006.
4. Кузнецов Д.В., Шалыто А.А. Система управления танком для игры "Robocode". Вариант 2. – СПбГУ ИТМО. 2003. – Режим доступа: http://is.ifmo.ru/projects/robocode2/
5. Туккель Н.И., Шалыто А.А. Система управления танком для игры "Robocode". Вариант 1. Проектная документация. – СПбГУ ИТМО. 2003. – Режим доступа: http://is.ifmo.ru/projects/tanks/
6. Шалыто А.А. SWITCH-технология. Алгоритмизация и программирование задач. – СПб: Наука, 1998. – Режим доступа: http://is.ifmo.ru/books/switch/1
7. История игры Robocode. – Режим доступа: http://robowiki.net/cgi-bin/robowiki?History
8. Gade M., Knudsen M., et al. Applying Machine Learning to Robocode. 2003. – Режим доступа: www.csc.calpoly.edu/~team14fk/F04/dat3_report.pdf
9. Eisenstein J. Evolving robot tank fighters. Technical Report AIM-2003-023, AI Lab, MIT. 2003. – Режим доступа: http://www.ai.mit.edu/people/jacobe/research/robocode/genetic_tanks.pdf
10. Ferreira C. Gene Expression Programming: A New Adaptive Algorithm for Solving Proble ms // Complex Systems. – 2001. – V.13. – № 2. P. 87–129. – Режим доступа: www.gene-expression-programming.com/webpapers/GEP.pdf
11. Koza J.R. Genetic programming. On the Programming of Computers by Means of Natural Selection. – MA. The MIT Press. 1998.

3 ПРИМЕНЕНИЕ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ ДЛЯ ГЕНЕРАЦИИ АВТОМАТОВ

УДК 004.4’242

РАЗРАБОТКА ИНСТРУМЕНТАЛЬНОГО СРЕДСТВА ДЛЯ ГЕНЕРАЦИИ КОНЕЧНЫХ АВТОМАТОВ С ИСПОЛЬЗОВАНИЕМ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ

Е.А. Мандриков, В.А. Кулев

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В рамках проведения научно-исследовательской работы выполняется разработка инструментального средства для генерации автоматов на основе генетического программирования, которое позволяет повысить уровень автоматизации проектирования автоматных программ. На английском языке это инструментальное средство имеет аббревиатуру – GAAP (Genetic Algorithms for Automata-based Programming).

Ключевые слова: генетические алгоритмы, инструментальное средство, конечный автомат, автоматное программирование

Введение

В настоящее время предложено большое число различных реализаций схем генетических алгоритмов [1–3], а также генетических операторов и способов кодирования особей. Многие существующие генетические алгоритмы являются чисто теоретическими разработками, которые не поддержаны программными продуктами. Другие опробованы на тестовых задачах, но их применение в решении практических задач затруднено из-за отсутствия или недостаточной гибкости соответствующих программных систем [4].

Перечислим недостатки, которыми обладают известные программные средства, реализующие генетические алгоритмы.

1. Жесткая привязка разработанного программного обеспечения к задаче на этапах кодирования и декодирования особей.
2. Программная реализация генетического алгоритма производится практически с нуля.
3. Закрытость разработанных программ как для доработки, так и для интеграции с другими программами.
4. Невозможность сохранения результатов поиска и промежуточных состояний популяции и, следовательно, невозможность анализа этой информации в дальнейшем.

Отметим, что в теореме NFL (No Free Lunch) [5] утверждается, что у всех алгоритмов, которые ищут экстремум функции качества, производительность одинакова, если усреднить результаты по всем возможным функциям качества. Практическое значение этой теоремы состоит в том, что не существует панацеи на все случаи жизни, а несомненный успех какого-либо оптимизационного метода в определенной области знаний не гарантирует такого же успеха в другой области. Это означает, что для каждой специфической области необходимо проводить исследования и отыскивать тот оптимизационный метод, который подходит для нее наилучшим образом. При этом можно выделить ряд особенностей систем, использующих генетические алгоритмы.

1. Для каждой оптимизационной задачи целесообразно строить свою функцию приспособленности [1, разд. 2.3 и 2.4]. Следовательно, система должна обладать возможностью использования различных функций приспособленности.
2. В систему должны входить генетические операторы (создание случайной особи, мутация, рекомбинация, переупорядочение). Напомним, что для различных представ-

лений структур хромосом необходимы различные операторы, учитывающие специфику представления.
3. Необходимым условием эффективности работы систем с использованием генетических алгоритмов является ее автоматическая или автоматизированная настройка на объект исследования.

Исходя из изложенного, сформулируем ряд требований к разрабатываемому инструментальному средству для поддержки генетических алгоритмов, генерирующих конечные автоматы:

1. наличие встроенных типов кодирования хромосом и, как следствие, библиотек генетических операторов;
2. наличие блока подбора и адаптации параметров генетического алгоритма и генетических операторов под решаемую задачу;
3. возможность расширения за счет использования внешних функций генетических операторов, способов кодирования и функций пригодности;
4. возможность визуализации получаемых решений;
5. многокомпонентность системы с возможностью включения и отключения требуемых для решаемой задачи компонентов.

Описание разрабатываемого инструментального средства

Для реализации прототипа инструментального средства был выбран язык Java (http://java.sun.com/). Простота интеграции полученной системы с другими системами достигается за счет использования модульности и автоматизированного средства сборки Maven (http://maven.apache.org/).

Инструментальное средство состоит из следующих компонентов.

1. Модуль поддержки генетических алгоритмов, который включает в себя:
    1. репозиторий особей для сохранения промежуточных и конечных результатов работы. Репозиторий можно использовать в качестве общего хранилища особей при реализации метода композитного генетического программирования [6]. Использование нескольких репозиториев позволяет реализовать островную модель генетических алгоритмов [1] с различной топологией и способами обмена особями между островами, которую в дальнейшем предполагается включить в модуль;
    2. реализацию классического генетического алгоритма с блоком подбора и адаптации параметров алгоритма и генетических операторов под решаемую задачу;
    3. средства для различных типов визуализаций и отладки процесса выполнения генетического алгоритма. Пример визуализации приведен на рис. 1;
    4. различные операторы и способы представления хромосом (бинарное и вещественное кодирование [7]) для задач численной оптимизации.
2. Модуль поддержки конечных автоматов, который включает в себя:
    1. различные способы представления конечных автоматов. В настоящее время реализованы два представления (строки фиксированной длины и деревья решений [8], и будет внедрено еще одно (сокращенные таблицы [9]);
    2. поддержку XML-представления, которое позволяет хранить и преобразовывать автоматы. Предлагаемое XML-представление является основой для построения различных визуальных представлений автомата с помощью XSL-преобразования. Указанное представление может быть конвертировано в коды на различных языках программирования [10];
    3. реализацию модели «конечный автомат + объект управления = автоматизированный объект» [11], позволяющей конечному автомату воздействовать на объект управления;

4. механизм визуализации процесса работы автоматизированного объекта с возможностью выполнения по шагам, как в прямом направлении, так и в обратном направлениях. Пример визуализации приведен на рис. 2.

3. Модуль поддержки генетического программирования для конечных автоматов, который включает в себя набор встроенных в систему генетических операторов для различных представлений конечных автоматов в виде хромосом, реализованных в указанном выше модуле. Также планируется внедрение в модуль поддержки генетических алгоритмов функциональности для распределенных вычислений функции приспособленности, так как эти вычисления очень трудоемки.

После реализации базовых компонентов системы полученный программный комплекс был апробирован на ряде тестовых задач: задача о построении «разливочной линии» [9], задача о «флибах» [1, 12, 13], задача об «умном муравье» [1, 14]. В дальнейшем планируется применение полученного программного средства для решения задач программирования мобильных роботов.

Следует также отметить, что проект GAAP разрабатывается в рамках движения за открытую проектную документацию (http://is.ifmo.ru/foundation/?i0=foundation) и является проектом с открытым кодом (сайт проекта GAAP – http://godin.net.ru:8080/projects/gaap/), что способствует его дальнейшему развитию.

Заключение

Ранее реализованное инструментальное средство [14] было ориентированно на исследование лишь одной задачи («Умный муравей»), что затрудняло его дальнейшее расширение и использование на других задачах. Другое инструментальное средство [9] было ориентированно на использование лишь одного представления конечных автоматов, что не давало возможности тщательно изучать решаемую задачу.

Предлагаемое средство является более гибким и универсальным.

Литература

1. Технология генетического программирования для генерации автоматов управления системами со сложным поведением. Отчет по I этапу темы.
2. Технология генетического программирования для генерации автоматов управления системами со сложным поведением. Отчет по II этапу темы.
3. Фогель Л., Оуэнс А., Уолш М. Искусственный интеллект и эволюционное моделирование. – М.: Мир, 1969.
4. Технология генетического программирования для генерации автоматов управления системами со сложным поведением. Отчет о патентных исследованиях.
5. Wolpert D. H., Macready W. G. No free lunch theorems for optimization // IEEE Transactions on Evolutionary Computation. – 1997. – April. – Vol. 1, no. 1. – Pp. 67–82.
6. Бедный Ю.Д. Применение генетических алгоритмов для построения клеточных автоматов.– СПб: СПбГУ ИТМО, 2006.
7. Herrera F., Lozano M., Verdegay J. L. Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis // Artifcial Intelligence Review. – 1998. – Vol. 12. – № 4. – P. 265–319
8. Данилов В.Р., Шалыто А.А. Метод генетического программирования для генерации автоматов, представленных деревьями решений. // Научное программное обеспечение в образовании и исследованиях. – СПбГУ ПУ. – 2008. – С. 175–181.

9. Поликарпова Н.И., Точилин В.Н., Шалыто А.А. Разработка библиотеки для генерации управляющих автоматов методом генетического программирования. – 2007 – С. 84–87.
10. Непейвода Н.Н., Скопин И.Н. Основания программирования. – Ижевск-Москва: Институт компьютерных исследований, 2003.
11. Поликарпова Н.И., Шалыто А.А. Автоматное программирование. Учебно-методическое пособие. – СПб.: СПбГУ ИТМО, 2007.
12. Лобанов П.Г., Шалыто А.А. Использование генетических алгоритмов для автоматического построения конечных автоматов в задаче о «флибах» // Известия РАН. Теория и системы управления. – 2007. – №5. – С. 127–136.
13. Мандриков Е.А., Кулев В.А., Шалыто А.А. Применение генетических алгоритмов для создания управляющих автоматов в задаче о «флибах» // Информационные технологии – 2008. – №1. – С. 42–45.
14. Бедный Ю.Д., Шалыто А.А. Применение генетических алгоритмов для построения автоматов в задаче «Умный муравей». – СПбГУ ИТМО, 2007.

УДК 004.4’242

МЕТОД ПРЕДСТАВЛЕНИЯ АВТОМАТОВ ДЕРЕВЬЯМИ РЕШЕНИЙ ДЛЯ ИСПОЛЬЗОВАНИЯ В ГЕНЕТИЧЕСКОМ ПРОГРАММИРОВАНИИ

В.Р. Данилов

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В работе предлагается новый метод представления автоматов в виде особей эволюционного алгоритма, основанный на деревьях решений. Предлагаемый метод представления автоматов позволяет существенно сократить объем требуемой для работы эволюционного алгоритма памяти. Разработан метод генетического программирования, основанный на предложенном представлении. Эффективность разработанного метода демонстрируется на примере решения одного из вариантов задачи об «Умном муравье».

Ключевые слова: генетическое программирование, дерево решений, конечный автомат, автоматное программирование

Введение

Автоматное программирование [1] – парадигма программирования, использующая представление программ в виде формальной модели – конечных автоматов. Применение генетического программирования [2] для генерации автоматов позволяет существенно сократить цикл разработки некоторых автоматных программ.

Для генерации автоматов могут применяться различные модификации эволюционных алгоритмов [3]. Однако для большинства из этих методов размер хромосомы, требуемый для хранения автомата, экспоненциально зависит от числа входных переменных.

При решении задачи классификации плотности Дж. Коза [4] применил представление функции переходов автомата с помощью деревьев разбора [5], описывающих булевы функции. Это представление обладает существенно большей выразительностью по сравнению с традиционными табличными методами, что позволило эффективно решить задачу. Однако этот метод может быть применен только к автоматам, имеющим два состояния. Целесообразно разработать метод эффективного представления автоматов, который может быть применен к решению более сложных задач.

Представление автомата с помощью деревьев решений

Дерево решений [6] является удобным способом задания дискретной функции, зависящей от конечного числа дискретных переменных. Оно представляет собой помеченное дерево, метки в котором расставлены по следующему правилу:

- внутренние узлы помечены символами переменных;
- ребра – значениями переменных;
- листья – значениями функции.

Для определения значения функции по значениям переменных необходимо спуститься от корня до листа, и получить значение, которым помечен найденный лист. При этом из вершины, помеченной переменной x, переход производится по ребру, помеченному значением переменной x. На рис. 1 показано дерево, реализующее булеву функцию $f = a \wedge c \vee \neg a \wedge \neg b$

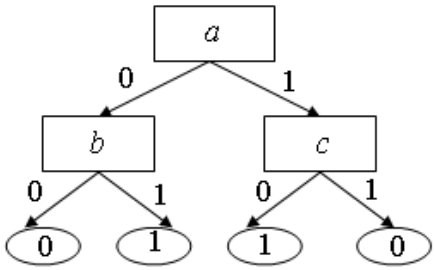


Рис. 1. Пример дерева решений

Опишем предлагаемый метод представления автомата. Для задания автомата необходимо выразить его функции переходов и выходов с помощью деревьев решений. Осуществим следующее преобразование автомата: вместо задания функций переходов и выходов для автомата в целом представим эти функции для каждого состояния. Более формально это выглядит следующим образом: зададим для каждого состояния $q \in Q$ функцию $\sigma_q : X \to Q \times Y$, такую что $\sigma_q(x) = (\delta(q,x), \lambda(q,x))$ для $\forall x \in X$, где Q – множество состояний, X – множество входных воздействий.

Функции σ_q соответствуют функциям переходов и выходов из состояния q. Каждая из этих функций может быть выражена с помощью дерева решений. В этих деревьях переменными являются входные переменные автомата, а множеством значений – все возможные пары (Новое Состояние, Действие). Таким образом, автомат в целом задается упорядоченным набором деревьев решений.

Генетические операции

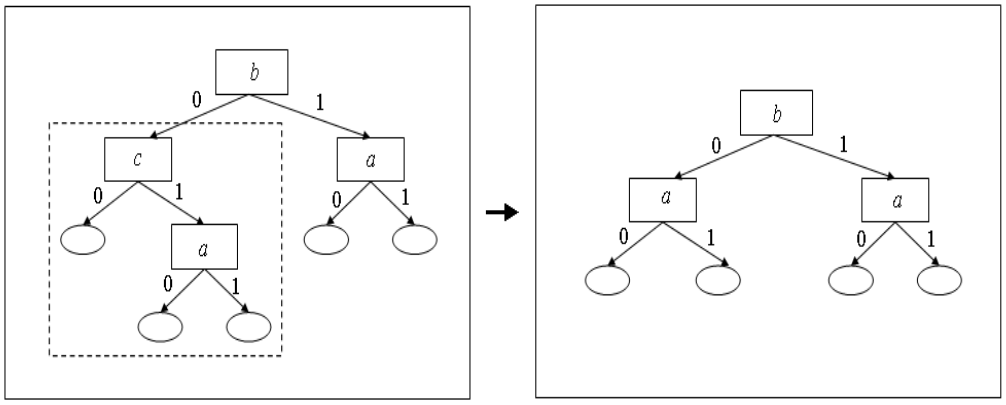
Для использования представления автоматов в виде набора деревьев решений в генетических алгоритмах определим следующие операции:

- случайное порождение автомата – в каждом состоянии создается случайное дерево решений;
- скрещивание автоматов – скрещиваются деревья решений в соответствующих состояниях;
- мутация автомата – в случайном дереве решений производится мутация.

Здесь считается, что число состояний в автомате фиксировано. Поэтому противоречий при выполнении определенных таким образом операций не возникнет.

После определения операций над набором деревьев решений определим генетические операции над отдельными деревьями решений. Их можно определить следующим образом.

- Случайное порождение дерева решений. При этом случайным образом выбирается метка: либо одно из возможных значений функции переходов, либо одна из переменных. После этого создается вершина, помеченная выбранной меткой. Если была выбрана переменная, то рекурсивно генерируются дети вершины, иначе вершина становится листом дерева.
- Мутация – выбирается случайный узел в поддереве. После этого поддерево, соответствующее выбранному узлу, заменяется на случайно сгенерированное (рис. 2, а).
- Скрещивание – в скрещиваемых деревьях выбираются два случайных узла. После этого поддерево, соответствующее выбранному узлу в первом дереве, заменяется поддеревом, соответствующим узлу второго дерева (рис. 2, б).



а б

Рис. 2. Мутация деревьев решений

Отметим, что заданные таким образом операции могут порождать деревья, в которых некий атрибут встречается на пути от корня до листа дважды (например, дерево, полученное в результате скрещивания, приведено на рис. 2, б). Таким образом, необходимо ввести операцию обрезки – удаление недостижимых ветвей. Операция может быть выполнена следующим образом: узел, одна из дочерних вершин которого недостижима, заменяется достижимой дочерней вершиной. Операция обрезки должна выполняться после генетических операций скрещивания и мутации.

Апробация

Разработанный подход был протестирован на задаче «Умный муравей-2» [7]. Приведем описание задачи. Муравей находится на случайном игровом поле. Поле представляет собой тор размером 32×32 клетки. При этом муравей видит перед собой некоторую область (рис. 3).

Еда в каждой клетке располагается с некоторой вероятностью μ. Значение μ является параметром задачи. Игра длится 200 ходов, за каждый из которых муравей может сделать одно из трех действий: поворот налево или направо, шаг вперед. Если после хода муравей попадает на клетку, где есть еда, то еда съедается. Целью задачи является построение стратегии поведения муравья, при которой математическое ожидание съеденной еды максимально.

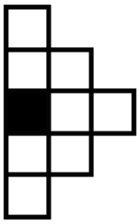


Рис. 3. Видимая муравью область

Автомат управления муравьем в этой задаче имеет восемь (число видимых клеток) входных переменных, каждая из которых определяет, есть ли еда в клетке, соответствующей переменной. Все входные переменные имеют логический тип.

Таблица. Результаты экспериментов

μ	0.01			
	Фитнесс-функция			
Число состояний	2	4	8	16
Битовые строки	2.74	2.93	2.83	2.89
Таблица переходов	2.68	3.49	3.74	3.78
Предложенный метод	2.71	2.92	2.88	3.69
μ	0.02			
	Фитнесс-функция			
Число состояний	2	4	8	16
Битовые строки	7.66	8.38	7.95	6.98
Таблица переходов	6.12	7.32	7.24	7.28
Предложенный метод	7.68	8.04	7.32	8.25
μ	0.03			
	Фитнесс-функция			
Число состояний	2	4	8	16
Битовые строки	14.46	13.81	13.23	11.93
Таблица переходов	12.48	12.17	11.72	11.15
Предложенный метод	14.14	13.86	13.77	14.18
μ	0.04			
	Фитнесс-функция			
Число состояний	2	4	8	16
Битовые строки	19.11	18.68	17.47	15.10
Таблица переходов	17.18	15.94	15.03	13.68
Предложенный метод	18.28	20.28	18.60	20.18

Предложенный подход сравнивался с генетическими алгоритмами над битовыми строками и генетическими алгоритмами, оперирующими над таблицами переходов. Эксперимент заключался в сравнении полученных значений фитнесс-функции (объема съеденной еды) за фиксированное число шагов с одинаковыми настройками. Запуск алгоритмов производился со следующими настройками: стратегия отбора – элитизм, для размножения отбираются 25 % популяции, имеющих наибольшее значение фитнесс-функции; частота мутации – 2 %; размер популяции – 200 особей; число популяций – 100; фитнесс-функция – среднее значение съеденной еды на 200 случайных картах, карты внутри одной популяции совпадают, карты различных популяций различны.

Результаты эксперимента приведены в таблице. Последнее измерение фитнесс-функции осуществлялось на случайном наборе из 2000 карт.

Анализ показывает, что в случаях, когда важны значения почти всех предикатов (при $\mu = 0.02$), предлагаемый метод работает хуже известных. Это можно объяснить тем, что искомые автоматы плохо описываются деревьями решений. Однако при больших значениях μ предложенный метод работает лучше, особенно при большом числе состояний. Результаты экспериментов приведены в таблице.

Таким образом, выяснено, что метод работает лучше известных в большинстве случаев, за исключением ситуаций, когда функция переходов не может быть эффективно выражена деревом решений.

Заключение

В работе предложен метод высокоуровневого представления автоматов. Он позволяет значительно сократить требуемую память для большинства практических задач. Другим важным достоинством предлагаемого подхода является упрощение представления полученного автомата для человека. Дальнейшее развитие исследований предполагает адаптацию подхода для генерации систем взаимодействующих автоматов и использование двоичных разрешающих диаграмм для представления автоматов.

Литература

1. Шалыто А.А. Технология автоматного программирования. // Труды первой Всероссийской научной конференции «Методы и средства обработки информации», МГУ. 2003. С. 528–535. – Режим доступа: http://is.ifmo.ru/works/tech_aut_prog/
2. Koza J. Genetic programming: On the Programming of Computers by Means of Natural Selection. – MA: The MIT Press, 1992.
3. Гладков Л.А., Курейчик В.В., Курейчик В.М. Генетические алгоритмы. – М.: Физматлит, 2006.
4. Andre D., Bennet F., Koza J. Discovery by Genetic Programming of a Cellular Automata Rule that is Better than any Known Rule for the Majority Classification Problem. – Режим доступа: http://citeseer.ist.psu.edu/33008.html
5. Хопкрофт Дж., Мотвани Р., Ульман Дж. Введение в теорию автоматов, языков и вычислений. – М.: Вильямс, 2002.
6. Шеннон К. Работы по теории информации и кибернетике. – М.: Изд-во иност. литер., 1963.
7. Бедный Ю.Д., Шалыто А.А. Применение генетических алгоритмов для построения автоматов в задаче «Умный муравей». – СПбГУ ИТМО. 2007. – Режим доступа: http://is.ifmo.ru/works/ant

УДК 004.4’242

ПРИМЕНИЕ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ ДЛЯ ПОСТРОЕНИЯ АВТОМАТОВ МУРА И СИСТЕМ ВЗАИМОДЕЙСТВУЮЩИХ АВТОМАТОВ МИЛИ НА ПРИМЕРЕ ЗАДАЧИ ОБ «УМНОМ МУРАВЬЕ»

А.А. Давыдов, Д.О. Соколов, Ф.Н. Царев
(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В работе предлагается генетический алгоритм, осуществляющий построение конечных автоматов Мура. Аналогичный алгоритм предлагается применять и для построения систем взаимодействующих автоматов Мили. Для указанных типов автоматов разработаны генетические операции скрещивания и мутации. Реализация генетических алгоритмов выполнена на языке программирования *Java*. Применение этих алгоритмов иллюстрируется на примере задачи об «Умном муравье»
Ключевые слова: генетические алгоритмы, автомат Мура, автомат Мили, автоматное программирование

Введение

В последнее время все чаще применяется автоматное программирование [1], в рамках которого поведение программ описывается с помощью конечных детерминированных автоматов. В ряде задач автомат удается построить эвристическими методами, однако часто такое построение требует больших затрат времени. В ряде случаев автоматы эвристически и вовсе не построить. Примером такой задачи является задача об «Умном муравье» [2]. Даже для этой относительно простой задачи полный перебор крайне трудоемок. Поэтому для построения автоматов в задачах такого рода целесообразно применять генетические алгоритмы [2–9].

Все известные авторам работы в этой области [10] посвящены построению одного автомата Мили. Однако в автоматном программировании допускается использование автоматов Мура и систем взаимодействующих автоматов [1]. При этом отметим, что одним из основных способов взаимодействия автоматов является их вложенность. Целью настоящей работы является построение с помощью генетического программирования [7] пары вложенных автоматов Мили и автомата Мура для задачи об «Умном муравье».

Постановка задачи

Ниже приведено описание рассматриваемой задачи [2]. Игра происходит на поверхности тора размером 32×32 клетки (рис. 1).

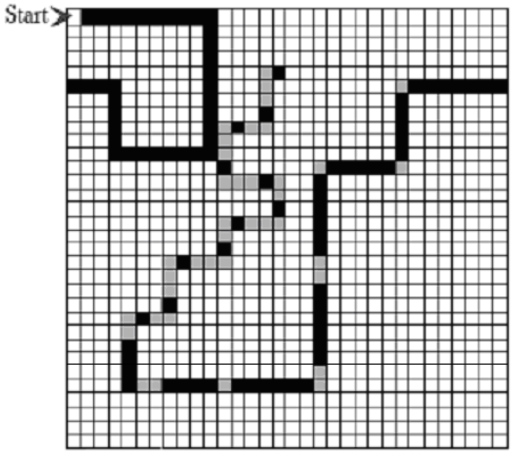


Рис. 1. Игровое поле

В некоторых клетках находится еда (на рисунке эти клетки обозначены черным цветом). Муравей начинает движение из клетки, помеченной меткой *Start*. За ход муравей может выполнить следующие действия:

- повернуть налево;
- повернуть направо;
- сделать шаг вперед, и если в новой клетке есть еда, то съесть ее;
- ничего не делать.

Игра длится 200 шагов. Цель игры – создать муравья «с минимальным числом состояний», который за минимальное число шагов съест как можно больше яблок.

Ранее эта задача была решена [3] при помощи автомата Мили без использования взаимодействующих автоматов. Для построения таких автоматов в работе [8] применялись генетические алгоритмы. В настоящей работе рассматриваемая задача решается сначала с помощью системы из двух автоматов Мили, взаимодействующих за счет вложенности, а затем – с помощью автомата Мура.

Алгоритм генетического программирования

Для генерации пары вложенных автоматов Мили и автомата Мура предлагается использовать один и тот же *островной генетический алгоритм* [5, 9], но с различными структурами хромосом, операторами мутации и скрещивания и функциями приспособленности. Отметим, что при необходимости этот алгоритм может быть адаптирован и для других типов систем автоматов.

В первом случае особь представляется собой пару вложенных автоматов Мили, один из которых будем называть внешним, а другой – внутренним. Хромосома особи состоит из описаний двух автоматов, каждое из которых содержит номер начального состояния и описания каждого состояния.

Описание состояния содержит описание двух переходов – перед муравьем есть еда, и ее нет. Для внешнего автомата любой из этих переходов может быть не определен. В такой ситуации переход совершается во вложенном автомате, у которого, в свою очередь, определены все переходы. Описание перехода состоит из номера состояния, в которое он ведет, и действия, выполняемого при выборе этого перехода.

Во втором случае особь представляет собой автомат Мура. Хромосома особи состоит из номера начального состояния и описания каждого состояния. Описание состояния содержит действие, выполняемое при переходе в состояние, и описание двух переходов – перед муравьем есть еда, и ее нет. Описание перехода состоит из номера состояния, в которое он ведет. Хромосома не кодируется битовой строкой, как это делается традиционно, а представляет собой объект на языке *Java* вида

```
class Automaton {
  int initialState;
  Transition[][] transition;
  char[] stateAction[]; // только для автоматов Мура
  Automaton nestedAutomaton;
}
```

Островной алгоритм традиционно состоит из следующих этапов:

- создание начального поколения;
- мутация;
- скрещивание (кроссовер);
- вычисление функции приспособленности (фитнесс-функции);
- обмен особями между островами;
- отбор особей для формирования следующего поколения.

Опишем этапы этого алгоритма применительно к рассматриваемой задаче. При этом сначала опишем этапы, которые являются общими для обеих моделей описания поведения муравья.

Создание начального поколения

Все острова заполняются случайно сгенерированными автоматами. Все автоматы имеют заранее заданное число состояний.

Мутация (малая или мутация особи)

Каждое поколение особь с заданной вероятностью может мутировать. Для пары вложенных автоматов Мили это означает:

- случайное изменение стартового состояния особи;
- выбор случайного перехода и случайное изменение состояния, в который он ведет;
- удаление (добавление) случайного перехода для внешнего автомата;
- изменение действия на случайном переходе;
- мутация вложенного автомата.

Для автоматов Мура применяется аналогичный оператор мутации:

- случайное изменение стартового состояния особи;
- выбор случайного перехода и случайное изменение состояния, в который он ведет;
- изменение действия в случайном состоянии.

Каждый из перечисленных действий выполняется с некоторой априори заданной вероятностью.

Мутация (большая или мутация острова)

Через заранее заданное число поколений фиксированная доля островов заменяется островами со случайными особями. Проведение мутации в момент, когда функция приспособленности (имеются в виду только элитные особи) изменяется незначительно, невозможно, так как за счет миграции особей между островами фитнесс-функция постоянно изменяет свое значение.

Скрещивание

Оператор скрещивания получает на вход две особи и выдает две особи (потомки входных особей). Оператор скрещивания пары вложенных автоматов аналогичен описанному в работе [8]. Вложенные автоматы равновероятно либо скрещиваются, либо один ребенок наследует вложенный автомат одного родителя, а другой – другого.

Оператор скрещивания автоматов Мура также аналогичен описанному в работе [8], но имеет особенность в связи с наличием действия в состоянии. Обозначим родительские особи – *P1* и *P2*, а детей – *S1* и *S2*. Обозначим действие в *k*-ом состоянии автомата *A* как *A.a[k]*. Тогда для любого *k* будет верно одно из следующих утверждений:

- S1.a[k] = P1.a[k], S2.a[k] = P2.a[k];
- S1.a[k] = P2.a[k], S2.a[k] = P1.a[k];

Оба этих варианта равновероятны.

Вычисление функции приспособленности

В настоящей работе для генерации автоматов Мура применяется функция приспособленности вида

$$F - \frac{T}{200},$$

где F – количество съеденной еды, T – номер последнего хода, на котором муравей съел еду. Как показали вычислительные эксперименты, такая функция не подходит для генерации пары взаимодействующих автоматов Мили, так как при ее использовании генерируются слабо определенные внешние автоматы, функциональность которых, в основном, состоит в передаче управления вложенному автомату. Поэтому для пары вложенных автоматов Мили применялась функция приспособленности вида

$$F - \frac{T}{200} + C \cdot Z ,$$

где F – количество съеденной еды, T – номер последнего хода, на котором муравей съел еду, Z – число посещенных состояний у внешнего автомата, C – некоторый коэффициент. Подбор коэффициента C является ключевым при генерации средствами генетического программирования систем автоматов, взаимодействующих за счет вложенности автоматов.

Формирование следующего поколения

В качестве основной стратегии формирования следующего поколения используется элитизм [11]. При рассмотрении текущего поколения отбрасываются все особи, кроме некоторой доли наиболее приспособленных – элиты. Эти особи переходят в следующее поколение. После этого оно дополняется в определенной пропорции случайными особями, мутировавшими особями из текущего поколения и результатами скрещивания особей из текущего поколения. Особи, «имеющие право» давать потомство, определяются «в поединке»: выбираются две случайные пары особей, и более приспособленная в каждой паре становится одним из родителей. Эта эволюция происходит независимо на каждом из островов.

Через фиксированное число поколений каждый остров меняется с другим случайным числом выбранных случайно элитных особей. Также через некоторое число поколений происходит большая мутация.

В процессе разработки алгоритма исследовалась также возможность использования еще одного средства отбора – «войны между островами», в которой победитель (например, остров с большим суммарным значением функции приспособленности) истреблял население проигравшего (возможно, кроме элиты) и заселял остров своими «жителями». Однако это не ускоряет общую эволюцию, приводя к уменьшению разнообразия живущих особей.

Настраиваемые параметры генетического алгоритма

Алгоритм имеет следующие настраиваемые параметры:

- число островов;
- размер поколения на одном острове;
- число поколений между большими мутациями;
- доля уничтожаемых островов во время большой мутации;
- процент элиты;
- число поколений между обменом особями между островами;
- число обмениваемых особей;
- параметры малой мутации;
- параметры скрещивания;
- отношение «мутантов», случайных особей и детей особей из текущего поколения при формировании следующего поколения;
- C – коэффициент влияния внешнего автомата.

Соотношение этих параметров, при котором достигается оптимальный результат, неизвестно, однако при генерации систем автоматов существенное значение имеет коэффициент влияния внешнего автомата *C* в функции приспособленности.

Результаты

С помощью разработанного алгоритма построен автомат Мура с 10 состояниями, который позволяет муравью съесть всю еду за 198 шагов (рис. 2).

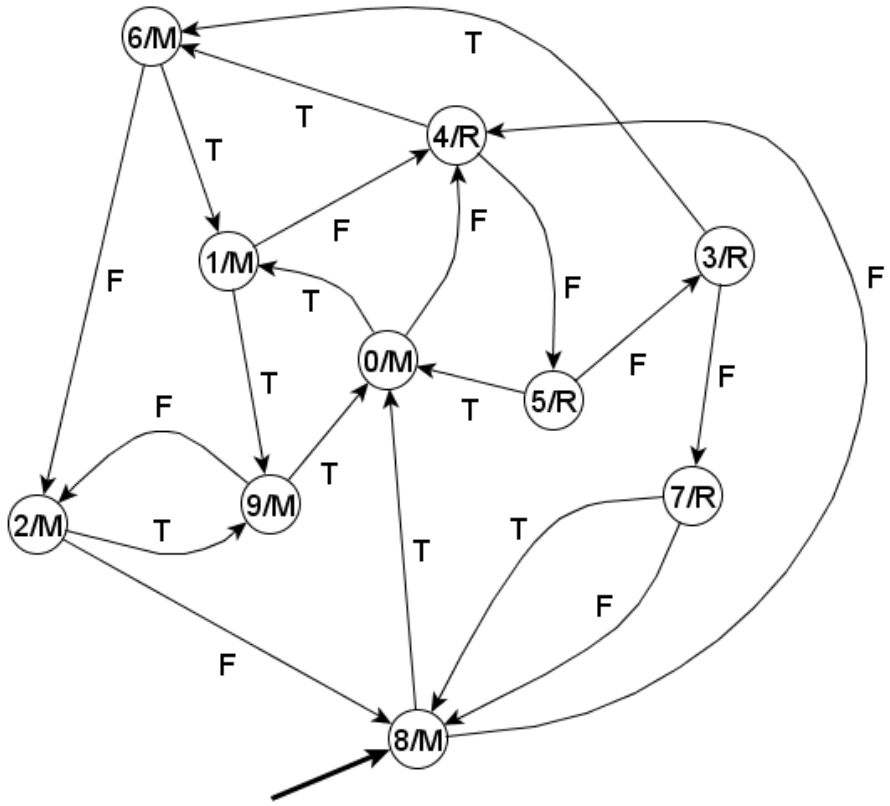


Рис. 2. Автомат, позволяющий муравью съесть всю еду за 198 шагов

Поясним условные обозначения на рис. 2. Пометки на вершинах имеют вид: *номер / действие*. Действия обозначаются следующим образом:

- L – поворот налево;
- R – поворот направо;
- M – сделать шаг, и если в новой клетке есть еда, то съесть ее.

Условия на переходах обозначаются следующим образом:

- T – перед муравьем есть еда;
- F – перед муравье нет еды.

Стартовое состояние отмечено внешней стрелкой. Лучшим достигнутым авторами результатом *для девяти состояний* является автомат, который позволяет муравью съесть 86 единиц еды также за 198 ходов.

Для взаимодействующих автоматов Мили не удалось построить пару, которая позволяет муравью съесть всю еду, и каждый автомат из которой имеет меньше семи состояний, как это имеет место в случае использования одного автомата Мили [8].

Лучшей построенной авторами системой является система автоматов (4, 6) (четыре состояния во внешнем автомате, шесть – во внутреннем), позволяющая муравью съесть 87 единиц еды за 185 ходов (рис. 3). Данный результат был получен при значении параметра $C = 0.01$. Из изложенного можно сделать вывод о том, что автомат, получаемый в результате решения задачи о муравье, не декомпозируем либо плохо декомпозируем.

Поясним условные обозначения на рис. 3. Пометки на вершинах имеют вид: *номер*, а на переходах: *условие / действие*. Условия, действия и стартовые состояния обозначаем так же, как и на рис. 2.

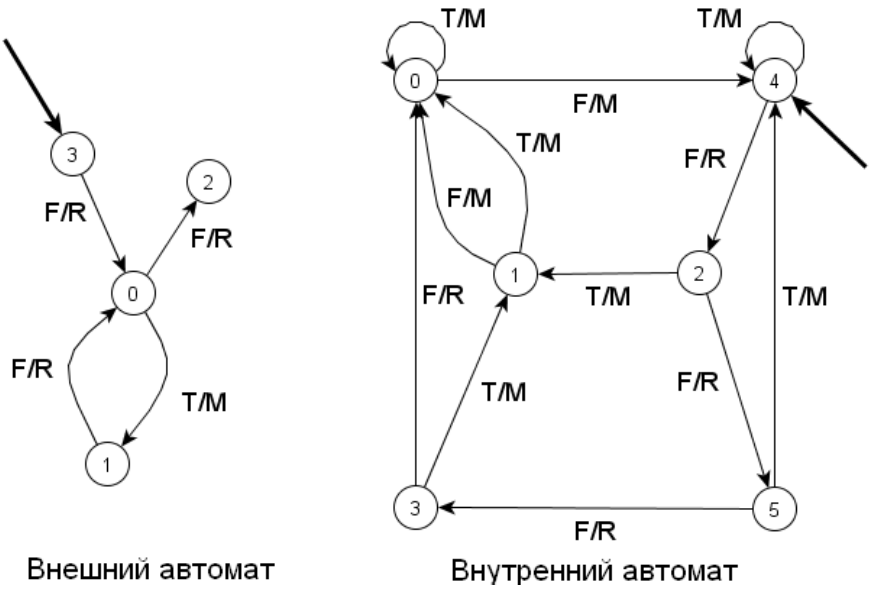


Рис. 3. Пример системы взаимодействующих автоматов Мили

Заключение

В работе предложен островной генетический алгоритм, осуществляющий построение конечных автоматов Мура и систем из двух взаимодействующих за счет вложенности автоматов Мили. Эти алгоритмы применены для задачи об «Умном муравье», и построен решающий эту задачу автомат Мура с десятью состояниями.

В дальнейшем возможно применение сокращенных таблиц [10], что должно позволить использовать предложенный алгоритм для решения задач с большим числом входных переменных.

Литература

1. Шалыто А.А. Технология автоматного программирования /Труды первой Всероссийской научной конференции «Методы и средства обработки информации». М.: МГУ. 2003. – Режим доступа: http://is.ifmo.ru/works/tech_aut_prog/
2. Jefferson D., Collins R., Cooper C., Dyer M., Flowers M., Korf R., Taylor C., Wang A. The Genesys System. 1992. – Режим доступа: www.cs.ucla.edu/~dyer/Papers/AlifeTracker/Alife91Jefferson.html
3. Angeline P., Pollack J. Evolutionary Module Acquisition / Proceedings of the Second Annual Conference on Evolutionary Programming. 1993. – Режим доступа: http://www.demo.cs.brandeis.edu/papers/ep93.pdf

4. Chambers L. Practical Handbook of Genetic Algorithms. Complex Coding Systems. Volume III. CRC Press, 1999. – Режим доступа: http://www.eknigu.com/info/Cs_Computer%20science/CsGn_Genetic,%20neural/Chambers%20D.L.%20(ed.)%20Vol.%203.%20Handbook%20of%20genetic%20algorithms.%20Complex%20coding%20systems%20(CRC,%201999)(ISBN%200849325390)(T)(659s).djvu
5. Гладков Л.А., Курейчик В.В., Курейчик В.М. Генетические алгоритмы. – М.: Физматлит, 2006.
6. Рассел С., Норвиг П. Искусственный интеллект: современный подход. – М.: Вильямс, 2006.
7. Koza J. R. Genetic programming: on the programming of computers by means of natural selection. MA: MIT Press, 1992. – Режим доступа: http://www.eknigu.com/info/Cs_Computer%20science/CsGn_Genetic,%20neural/Koza%20J.R.%20Genetic%20programming%20(MIT,%201998)(T)(ISBN%200262111705)(609s).djvu
8. Царев Ф.Н., Шалыто А.А. О построении автоматов с минимальным числом состояний для задачи об «умном муравье» / Сборник докладов X международной конференции по мягким вычислениям и измерениям. СПбГЭТУ "ЛЭТИ". Т.2, 2007, с. 88–91. – Режим доступа: http://is.ifmo.ru/download/ant_ga_min_number_of_state.pdf
9. Яминов Б. Генетические алгоритмы. – Режим доступа: http://rain.ifmo.ru/cat/view.php/theory/unsorted/genetic-2005
10. Поликарпова Н.И., Точилин В.Н., Шалыто А.А. Применение генетического программирования для реализации систем со сложным поведением / Научно-технический вестник СПбГУ ИТМО. – 2007. – Выпуск 39. – С 276–293.
11. Режим доступа: http://vestnik.ifmo.ru/ntv/39/ntv_39.3.3.pdf
12. De Jong K. An analysis of the behaviour of a class of genetic adaptive systems. PhD thesis. Univ Michigan. Ann Arbor, 1975.

УДК 004.4’242

МЕТОДЫ ОПТИМИЗАЦИИ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ ДЛЯ ПОСТРОЕНИЯ КОНЕЧНЫХ АВТОМАТОВ

П.Г. Лобанов

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

Предложены три метода, позволяющие повысить скорость сходимости стандартных генетических алгоритмов для класса задач, в которых решением является конечный автомат. Первый из этих методов позволяет сократить число состояний автомата за счет использования флагов. Метод восстановления связей между состояниями позволяет сократить число неиспользуемых состояний автомата. Метод сортировки состояний в порядке использования позволяет уменьшить число состояний автомата.

Ключевые слова: генетические алгоритмы, автоматное программирование

Введение

Генетические алгоритмы применяются при решении широкого круга задач. Однако при использовании классического генетического алгоритма часто требуются очень большие вычислительные ресурсы, чтобы получить решение с необходимой степенью точности. В ряде случаев классический алгоритм может не справиться с поставленной задачей за разумное время. Для повышения скорости сходимости генетического алгоритма и улучшения устойчивости его работы, как правило, применяются модификации, учитывающие особенности решаемой задачи. В работах [1, 2] рассматриваются оценочные функции и способы их адаптации для улучшения эффективности генетических алгоритмов. В работе [3] на примере задачи коммивояжера рассматривается, как зави-

сит эффективность эволюционных методов от выбора структуры хромосом. Необходимо отметить, что выбор структуры хромосом определяет, какие классы операторов скрещивания и мутации могут использоваться.

В данной работе рассматривается три новых модификации генетического алгоритма для класса задач, в которых решением является конечный автомат. Эти модификации могут использоваться как независимо друг от друга, так и совместно.

Описание методов

Приведем описание методов, предложенных в данной работе.

Использование автоматов с флагами

Поведение конечного автомата зависит от его состояний и переходов между ними. Каждый переход определяет, в какое состояние попадет автомат при некотором условии (функция, принимающая значения входных переменных и возвращающая булевское значение) и какие действия выполнятся при переходе. Иногда в качестве входных переменных на переходе используются значения выходных переменных, генерируемых самим автоматом. Такие переменные в работе [4] названы флагами. Значений флагов также влияют на окружение, в котором работает автомат.

Чем более сложную задачу решает автомат, тем, как правило, больше число его состояний. В книге [4] определен класс автоматов с флагами, в которых используется вектор многозначных флаговых переменных `F`, который несет информацию о предыдущих состояниях автомата. Если автоматы без флагов зависят от предыстории (предыдущего состояния), то автоматы с флагами зависят от глубокой предыстории (не только от предыдущего состояния). Автоматы данного класса взаимодействуют с внешними ячейками многозначной памяти.

Флаговые переменные одновременно используются и в качестве входных, и в качестве выходных переменных. Вместе с входными переменными и текущим состоянием автомата они определяют, в какое состояние перейдет автомат в следующий момент времени и какие выходные переменные он сформирует. Как показано в [4], использование флаговых переменных часто позволяет уменьшить число состояний в автомате. Это, правда, приводит к усложнению его графа переходов. Становится труднее понять алгоритм работы автомата. Однако этот недостаток не является существенным для случая автоматической генерации автоматов.

Рассмотрим, как можно использовать автоматы с флагами при решении задачи о флибах [5]. Поведение флиба описывается с помощью граф переходов. На рис. 1 в качестве примера изображен флиб с тремя состояниями *A*, *B* и *C*.

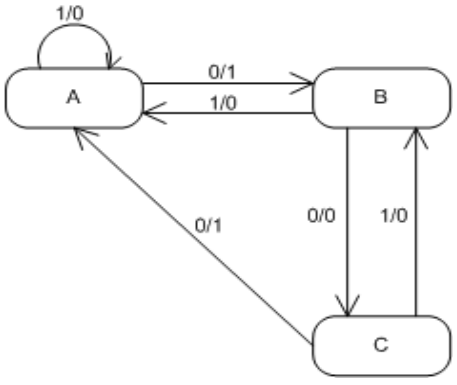


Рис. 1. Граф переходов для флиба с тремя состояниями

Во флибе флаговые переменные можно использовать для хранения предыдущих состояний окружающей среды. Пусть x и $F = (f_1, \ldots, f_n)$ – состояние среды и вектор флаговых переменных на текущий момент времени, соответственно. Тогда вектор флаговых переменных для следующего момента времени будет иметь вид $F' = (f_1' = x, f_2' = f_1, \ldots, f_n' = f_{n-1})$. На рис. 2 изображен граф переходов для флиба с тремя состояниями и одной флаговой переменной – это переменная, сформированная автоматом на предыдущем шаге, на этом шаге используется в качестве одной из входных переменных.

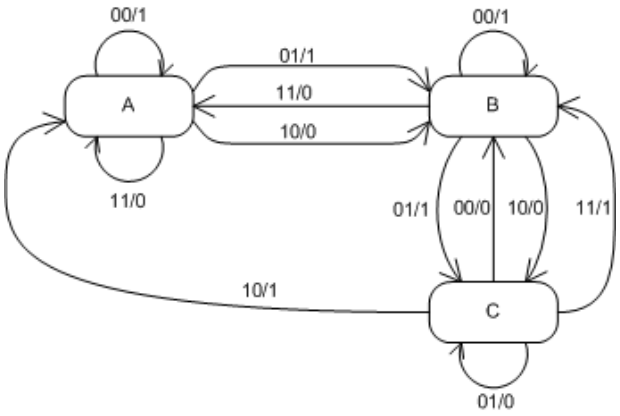


Рис. 2. Граф переходов для флиба с тремя состояниями и одной флаговой переменой

В общем случае входная переменная может принимать произвольное число значений. Пусть существует m допустимых комбинации значений входных переменных, перенумерованных от `0` до m–1. Тогда при `n` флаговых переменных число переходов из каждого состояния автомата будет равно m^{1+n}. Номер перехода, который должен использоваться в текущий момент времени, можно вычислить по формуле $x + m^1 f_1 + m^2 f_2 + \ldots + m^n f_n$, где x – номер текущей комбинации значений входных переменных, а $F = (f_1, \ldots, f_n)$ – вектор флаговых переменных.

Флаговые переменные позволяют существенно уменьшить число состояний в автомате. Рассмотрим эту особенность на простом примере. Пусть требуется построить флиб для простейшей среды, заданной с помощью повторяющейся битовой маски: `11100`. При использовании обычного автомата понадобится, по крайней мере, три состояния, чтобы флиб смог предсказывать изменения среды со сто процентной точностью. Пример такого флиба приведен на рис. 3.

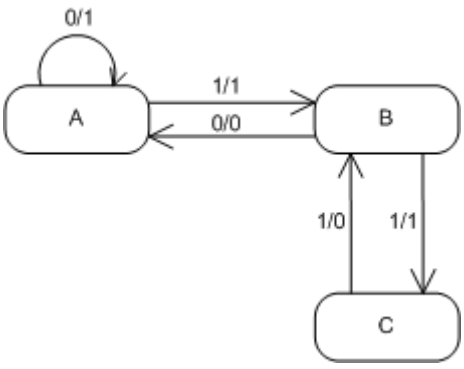


Рис. 3. Граф переходов флиба для среды, заданной битовой маской 11100

Начальное состояние автомата – *A*. Для упрощения понимания переходы между состояниями, которые не используются в случае рассматриваемой среды, были удалены. При использовании двух флаговых переменных достаточно одного состояния, чтобы построить флиб с точностью предсказания, равной ста процентам. Граф переходов для такого флиба с флагами изображен на рис. 4. Как и в предыдущем случае, все неиспользуемые переходы были удалены, чтобы упростить рисунок.

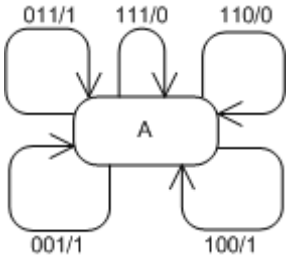


Рис. 4. Граф переходов флиба с двумя флаговыми переменными для среды, заданной битовой маской 11100

Как видно из приведенного примера, использование флаговых переменных позволяет уменьшить число состояний автомата за счет увеличения числа переходов между ними.

Алгоритм восстановления связей между состояниями

При генерации генетическим алгоритмом нового поколения решений, применении операторов скрещивания и мутации переходы в автоматах изменяются случайным образом. При таком изменении переходов в автомате, как правило, возникают состояния, в которые невозможно попасть из начального состояния при любой последовательности значений входных переменных. Будем называть такие состояния *недостижимыми*. Состояния, в которые можно попасть из начального состояния при некоторой последовательности значений входных переменных, будем называть *достижимыми*. Алгоритм восстановления связей между состояниями изменяет переходы в автомате таким образом, чтобы в нем не было *недостижимых состояний*.

Рассмотрим алгоритм восстановления связей между состояния на примере задачи о флибах. Предлагаемый алгоритм имеет следующий вид.

1. Формируется список *достижимых состояний*. Для этого можно, например, использовать обхода графа в глубину.
2. Рассматривается каждое состояние. Если рассматриваемое состояние не входит в число *достижимых состояний*, то для него выполняются следующие операции.
    1. Случайным образом выбирается состояние из списка *достижимых состояний*.
    2. Выбирается случайным образом один переход из выбранного состояния.
    3. В рассматриваемом состоянии заменяется один переход из этого состояния на переход, который ведет в то же состояние, что и переход, выбранный в пункте b.
    4. Выбранный в пункте b переход заменяется переходом, который ведет в рассматриваемое состояние.
    5. Обновляется список *достижимых состояний*. В него добавляется рассматриваемое состояние и все состояния, в которые можно попасть из него.

Рассмотрим работу описанного выше алгоритма на примере. На рис. 5 изображен граф переходов для флиба, полученного в результате работы генетического алгоритма. Начальное состояние автомата – *A*.

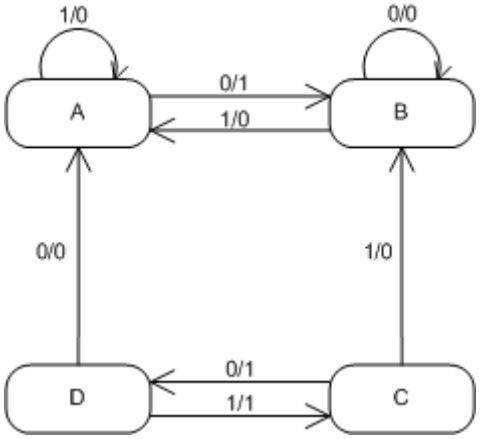


Рис. 5. Граф переходов для флиба, полученного в результате работы генетического алгоритма

Для автомата, граф переходов которого изображен на рис. 5, состояния *A* и *B* являются *достижимыми состояниями*, в то время как состояния *C* и *D* – *недостижимыми*. На рис. 6 *достижимые состояния* закрашены светло-серым цветом, *недостижимые состояния* изображены темно-серыми.

Алгоритм восстановления связей между состояниями перебирает все недостижимые состояния. Первым идет состояние *C*. Случайным образом выбирается состояние из списка *достижимых состояний* (например, состояние *B*). Выбирается случайным образом переход из состояния *B*. В качестве примера выберем переход, который соответствует значению входной переменной `1` и ведет в состояние *A*. Он меняется на переход, соответствующий такому же значению выходной переменной и ведущий в состояние *C*. Один из переходов, ведущих из состояния *C* (для примера возьмем переход, соответствующий значению входной переменной `1`), заменяется аналогичным переходом, ведущим в состояние *A*.

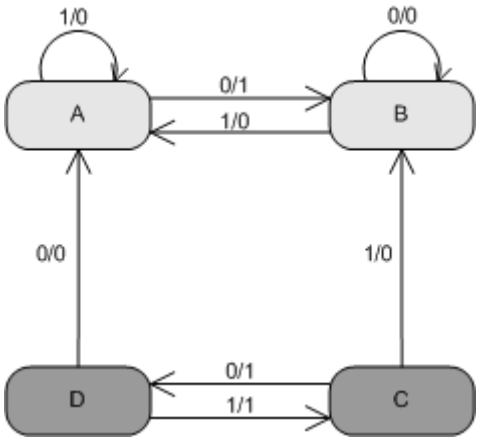


Рис. 6. *Достижимые* (светло-серые) и *недостижимые* (темно-серые) состояния автомата

На рис. 7 показан результат добавления состояния *C* в список *достижимых состояний*. Серым пунктиром показаны переходы, которые были удалены в результате этой операции. Черным пунктиром обозначены новые переходы. В состояние *C* автомат

попадет из начального состояния *A*, если на его вход, например, будет подана последовательность значений входной переменной вида `01`.

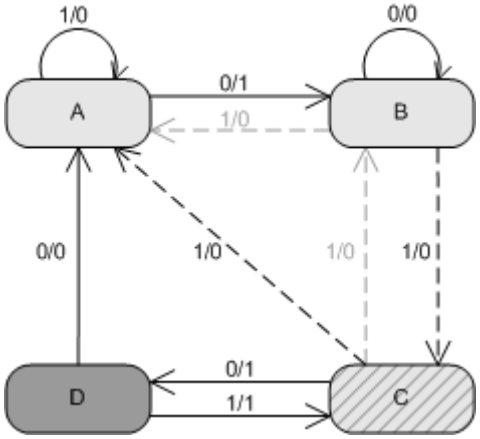


Рис. 7. Добавление состояние *C* в список *достижимых состояний*

После замены переходов состояние *C* стало *достижимым*. В состояние *D* можно попасть из состояния *C*, следовательно, состояние *D* также стало *достижимым*. Действительно, в состояние *D* можно попасть из начального состояния *A*, если подать, например, последовательность значений входной переменной вида `0011`. Так как после обновления списка *достижимых состояний* все состояния стали достижимыми, то алгоритм завершает свою работу.

Алгоритм восстановления связей между состояниями можно использовать как в качестве дополнительного, так и в качестве основного оператора мутации. Иногда может оказаться целесообразным проверять, не ухудшилась ли приспособленность особи после выполнения восстановления связей между состояниями. Если приспособленность особи ухудшилась, то граф переходов возвращается в исходное состояние.

Часто может оказаться целесообразным увеличить число *достижимых состояний* автомата, оставив часть состояний недостижимыми. Такой подход требует меньше вычислительных ресурсов и вносит меньше изменений в поведение, описываемое этим автоматом.

Алгоритм сортировки состояний в порядке использования

Во многих задачах в большинстве решений, которые перебирает генетический алгоритм, автоматы в процессе вычисления значения оценочной функции не попадают в часть состояний. Далее состояния, из которых выполняется переход хотя бы на одном шаге моделирования (вычисления оценочной функции), будем называть *используемыми состояниями*, а все остальные состояния – *неиспользуемыми состояниями*.

Если при вычислении оценочной функции на вход автомата подаются все возможные последовательности входных значений, то *неиспользуемые состояния* и переходы, связанные с ними, не влияют на поведение автомата. Примерами таких задач могут служить задача об умном муравье [6] и задача о флибах [5].

Приведем алгоритм сортировки состояний в порядке их использования.

1. Создается пустой словарь пар номеров: [«старый номер состояния» – «новый номер состояния»].
2. Моделируется работа автомата. Перед каждым переходом выполняем следующее: если в словаре нет пары, в которой первый элемент равен текущему номеру состояния, то в него добавляется пара [текущий номер состояния – количество пар в словаре].

3. Выполняется цикл по всем состояниям автомата. Для каждого состояния: если в словаре нет пары, в которой первый элемент равен номеру состояния, то в него добавляется пара [номер состояния – количество пар в словаре].
4. Согласно словарю, изменяется порядок состояний и номера состояний, в которые ведут переходы.

Состояния, для которых в словарь добавляются пары в пункте 2, и есть *используемые состояния*. Состояния, для которых в словарь добавляются пары в пункте 3 – это *неиспользуемые состояния*.

Рассмотрим работу алгоритма сортировки состояний в порядке использования на примере задачи о флибах. Пусть среда задана с помощью повторяющейся битовой маски: `11100`. В процессе работы генетического алгоритма был получен флиб, граф переходов для которого изображен на рис. 8.

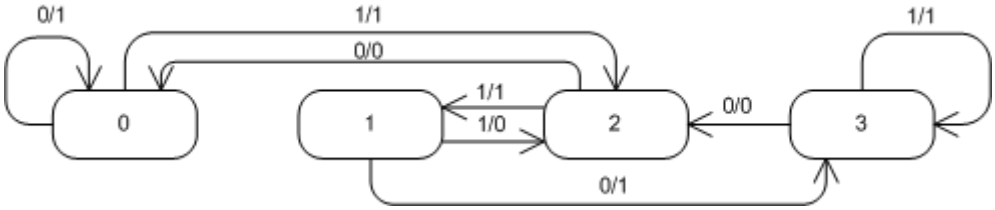


Рис. 8. Граф переходов флиба, полученного в процессе работы генетического алгоритма (среда задана битовой маской 11100)

Состояния флиба пронумерованы цифрами от нуля до трех. Начальное состояние – *0*.

Создается пустой словарь пар [«старый номер состояния» – «новый номер состояния»].

Начинается моделирование работы флиба. На вход автомату подается *1*. Автомат переходит в состояние *2,* и в словарь добавляется пара *[0, 0]*. На следующем шаге моделирования автомат переходит в состояние *1,* и в словарь добавляется пара *[2, 1]*. Далее автомат переходит в состояние *2,* и в словарь добавляется пара *[1, 2]*. При переходе из состояния *2* в состояние *0* в словарь ничего не добавляется, так как для состояний *2* в словаре уже есть пара. При дальнейшем моделирование работы флиба пары в словарь не добавляются, так как при среде, заданной битовой маской `11100`, флиб никогда не попадет в состояние *3*, для всех остальных состояний пары в словаре уже есть.

После моделирования работы флиба словарь пар номеров будет иметь следующий вид: *{[0, 0], [1, 2], [2, 1]}*.

Осуществляется цикл по всем состояниям. Так как в словаре нет пары только для состояний с *3*, то в него добавляется пара *[3, 3]*. Теперь словарь принимает следующий вид: *{[0, 0], [1, 2], [2, 1], [3, 3]}*.

Осталось изменить порядок состояний и номера состояний в переходах согласно словарю. Состояния *1* и *2* меняются местами и номерами. Граф переходов для флиба, получившегося в результате работы алгоритма, приведен на рис. 9.

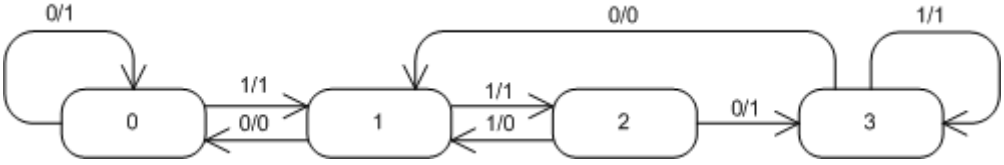


Рис. 9. Граф переходов флиба после применения алгоритма сортировки состояний (среда задана битовой маской 11100)

После применения алгоритма сортировки состояний легко построить автомат, имеющий такое же поведение, как и автомат, найденный с помощью классического генетического алгоритма, но имеющий меньшее количество состояний. Для этого в автомате, состояния которого отсортированы в порядке использования, достаточно удалить все *неиспользуемые состояния* и заменить переходы в них переходами в начальное состояние. В нашем случае одно *неиспользуемое состояние – 3*. В состояние *3* ведет только один переход из состояния *2*. Удалим состояние *3* и заменим переход в него на переход в состояние *0*. Результат изображен на рис. 10.

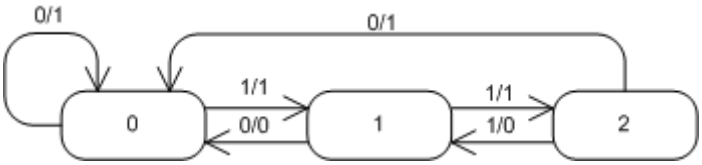


Рис. 10. Граф переходов флиба после удаления *неиспользуемых состояний* (среда задана битовой маской 11100)

Так как при моделировании работы автомата переход из состояния *2,* соответствующий значению входной переменной *0* ни разу не выполнялся, то поведение автомата не изменилось.

Особенности методов оптимизации генетических алгоритмов при построении конечных автоматов

Использование автоматов с флагами вместо обычных автоматов позволяет увеличить число переходов между состояниями и уменьшить число состояний. Благодаря этому в автоматах уменьшается число *недостижимых состояний*. Поведение автоматов с флагами меньше меняется при использовании оператора мутации, чем поведение автоматов без флагов.

Использование алгоритма восстановления связей между состояниями позволяет увеличить среднее число состояний в автоматах поколения. Чем больше состояний у автомата, тем меньшему числу изменений он подвергается, и, как следствие, тем меньше меняется его поведение. В итоге генетический алгоритм начинает отдавать предпочтение автоматам с большим числом состояний и, соответственно, с более сложным поведением. Это полезно, если приближенным решением задачи является автомат с небольшим числом состояний, а любое лучшее решение должно иметь значительно большее число состояний. В этом случае генетический алгоритм может «застрять» в локальном оптимуме. Эта проблема возникает из-за того, что стандартный оператор мутации (изменяет значение выходной переменной на переходе или состояние, в которое осуществляется переход) при однократном применении крайне редко значительно увеличивает число состояний в автомате. Алгоритм восстановления связей между состояниями решает эту проблему.

Алгоритм сортировки состояний в порядке использования приводит к одному виду автоматы с одинаковым поведением, но разным порядком нумерации состояний. Таким образом, использование алгоритма сортировки состояний автомата в порядке использования позволяет сократить пространство поиска, в котором генетический алгоритм перебирает решения.

При n флаговых переменных число переходов из каждого состояния автомата будет равно m^{1+n}, где m – число допустимых комбинации значений входных переменных.

Объем памяти, необходимой для хранения графа переходов автомата, пропорционален числу переходов из каждого состояния автомата. Создание новой особи из двух родительских также выполняется за время, пропорциональное числу переходов из каждого состояния автомата. Чтобы генетический алгоритм перебирал решения для автоматов с флагами со скоростью не меньшей, чем для автоматов без флагов, число состояний в автоматах с флагами должно не больше s/m^{1+n}, где s – максимальное число состояний для автомата без флагов.

Все описанные в настоящей работе методы оптимизации генетических алгоритмов подразумевают, что решения, с которыми ведется работа, хранятся в виде графа переходов. Эти методы также можно использовать, если структура хромосомы особи позволяет преобразовать ее в граф переходов, а затем выполнить обратную операцию. Например, преобразуем битовую строку в граф переходов, выполняем восстановление связей между состояниями и кодируем получившийся граф переходов с помощью битовой строки. Необходимо отметить, что такой подход связан с увеличением требований к вычислительным ресурсам.

Использовать алгоритм сортировки состояний нецелесообразно, если при вычислении значений оценочной функции на вход автоматам одного поколения подаются различные последовательности значений входных переменных.

Заключение

В работе предложены три новых модификации генетических алгоритмов, которые могут быть использованы для задач, решением которых является конечный автомат. Подробно рассмотрены их алгоритмы.

Рассмотрены особенности предложенных методов. Эти методы могут использоваться как независимо друг от друга, так и совместно.

Литература

1. Huang D. MS Thesis Preproposal: Adaptive Incremental Fitness Evaluation in Genetic Algorithms. 2005. NY: Rochester. – Режим доступа: http://www.cs.rit.edu/~dxh6185/downloads/MS_Thesis/Documents/Presentation.pdf
2. Linton R. Adapting binary fitness functions in genetic algorithms / Proceedings of the 42nd annual Southeast regional conference. – NY: ACM Press. 2004. – P. 391–395.
3. Bryant K. Genetic Algorithms and the Traveling Salesman Problem. Harvey Mudd College: Department of Mathematics, 2000. – Режим доступа: http://www.math.hmc.edu/math197/archives/2001/kbryant/kbryant-2001-thesis.pdf
4. Шалыто А.А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. – СПб: Наука, 1998. – 1026 с.
5. Лобанов П.Г., Шалыто А.А. Использование генетических алгоритмов для автоматического построения конечных автоматов в задаче о «Флибах» / Сборник докладов 4-й Всероссийской научной конференции «Управление и информационные технологии» (УИТ-2006). – СПбГЭТУ «ЛЭТИ». 2006. С. 144–149. – Режим доступа: http://is.ifmo.ru/works/flib
6. Царев Ф.Н., Шалыто А.А. Применение генетического программирования для генерации автомата в задаче об «Умном муравье» / Сборник трудов IV-ой Международной научно-практической конференции «Интегрированные модели и мягкие вычисления в искусственном интеллекте». Том 2. – М.: Физматлит. 2007. – С. 590–597. – Режим доступа: http://is.ifmo.ru/genalg/_ant_ga.pdf

4 ВЕРИФИКАЦИЯ АВТОМАТНЫХ ПРОГРАММ

УДК 004.4’242

МЕТОДЫ ВЕРИФИКАЦИИ МОДЕЛЕЙ АВТОМАТНЫХ ПРОГРАММ

С.Э. Вельдер, А.А. Шалыто

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В статье рассматриваются способы преобразования программ, разработанных на основе автоматного подхода, в модели Крипке, предназначенные для проверки свойств, относящихся к поведению системы. Эти свойства задаются формулами темпоральной логики. Предложены несколько методов такого преобразования и способов формулировки свойств.

Ключевые слова: верификация программ, верификация на моделях, автоматное программирование

Введение

В настоящей работе рассматриваются вопросы верификации на моделях (*Model checking*) применительно к автоматным программам. В их контексте исследуется специфика структуры (модели) Крипке для автоматных программ. Причина, по которой этим вопросам уделяется внимание, заключается в преимуществах автоматных программ перед остальными в смысле верификации. Действительно, алгоритмы верификации автоматных программ могут оперировать с моделью Крипке, заданной в явном виде. В работе выделены ключевые этапы верификации автоматных программ: преобразованиие автоматной модели в модель Крипке и построение требований к модели; собственно процесс верификации (отработки алгоритмов на полученных моделях); построение подтверждающих трасс (контрпримеров) в модели Крипке, а также представление контрпримеров, записанных в терминах модели Крипке, в виде путей в исходной автоматной модели. Предполагается, что собственно алгоритм верификации и построения контрпримеров в модели Крипке заранее выбран (такие алгоритмы изложены, например, в работах [1–5]).

Построение модели Крипке по автоматной модели

Рассмотрим несколько методов генерации множества атомарных предложений модели Крипке, соответствующей автоматной программе, и преобразования автомата с булевыми входными переменными в эту модель. Для каждого метода приводится пример записи требований к программе. Требования выражаются на языке темпоральной логики *CTL*.

Моделью Крипке (также *CTL*-моделью) для данного множества атомарных предложений *AP* будем считать тройку $\mathcal{M} = (S, R, Label)$, где:

- S – непустое множество состояний (*позиций*);
- $\rightarrow \subseteq S \times S$ – тотальное отношение на S, называемое отношением переходов. Свойство тотальности можно записать логической формулой $\forall s \in S \; \exists s' \in S \mid (s, s') \in \rightarrow$. Это отношение сопоставляет каждому состоянию *непустое* множество его состояний-последователей;
- $Label \subseteq S \times AP$ – помечающее отношение, которое сопоставляет каждому состоянию $s \in S$ множество атомарных предложений, истинных в s.

Иногда можно потребовать, чтобы в модели Крипке было задано непустое множество начальных состояний $S_0 \subseteq S$ или даже одно начальное состояние $s_0 \in S$.

Кратко опишем особенности методов преобразования автомата в модель Крипке.

Метод атомарных переходов требует «стереть» у автомата все события и переменные и оставить только состояния. Этот метод позволяет проверять не очень много свойств, но для простых свойств он достаточно эффективен. Он не требует преобразования контрпримера из модели Крипке в автомат, так как на переходах нет промежуточных состояний (в отличие от всех остальных методов).

Метод установки состояний на событиях и выходных воздействиях требует стереть у автомата только входные воздействия. Каждый переход разбивается на блоки, соответствующие событиям и выходным воздействиям, которые расположены друг за другом. Данный метод позволяет упоминать события и выходные воздействия в формулах темпоральной логики.

Метод полного графа переходов требует построить для автомата модель, в которой для каждого события будет сформирована полная система переходов. Полученная модель будет иметь большой размер, но в ней можно будет проверять любые темпоральные свойства.

Метод редуцированного графа переходов требует разложить переходы на атомарные блоки из событий и выходных воздействий, а входные воздействия должны быть особым способом в них сохранены. Из подробного описания в соответствующем разделе следует, что *метод установки состояний на событиях и выходных воздействиях* является его упрощением. Метод редуцированного графа переходов строит небольшие модели (линейного размера по отношению к размеру исходного автомата) и позволяет проверять практически все свойства, которые можно сформулировать относительно состояний, событий и воздействий (входных и выходных). Он является лучшим по соотношению эффективность–выразительность.

Для удобства и эффективности в последних трех методах каждое состояние модели Крипке помечено одним из трех «управляющих» атомарных предложений: *InState*, *InEvent*, *InAction* – для состояний модели, построенных, соответственно, из *состояний*, *событий*, *выходных воздействий* исходного автомата. Это сделано для того, чтобы при записи формулы в темпоральной логике можно было различать тип исследуемого состояния.

Во всех методах множество стартовых состояний модели Крипке состоит из одного элемента – стартового состояния исходного автомата. Существуют и альтернативные способы получения модели Крипке, например, методы дублирования состояний [6–9].

Методы конвертации автомата в модель Крипке и формулировки проверяемых свойств демонстрируются на трех примерах.

1. Автомат `ATrig`, эмулирующий работу *R*-триггера (его граф переходов изображен на рис. 1).
2. Автомат `ARemote`, эмулирующий универсальный инфракрасный пульт для бытовой техники [10]. Его схема связей и граф переходов изображены, соответственно, на рис. 2 и 3. Этот автомат изображен в упрощенной форме. Чтобы строго задать его структуру, необходимо ребра, на которых записана логическая дизъюнкция, разбить на несколько ребер, каждое из которых дизъюнкции не содержит (эти ребра будут соответствовать операндам дизъюнкции на исходном ребре).
3. Автомат `AElevator`, управляющий дверьми лифта (рис. 4).

В рамках исследований по государственному контракту [11] разработанные методы демонстрировались на примере системы, эмулирующей работу банкомата [12].

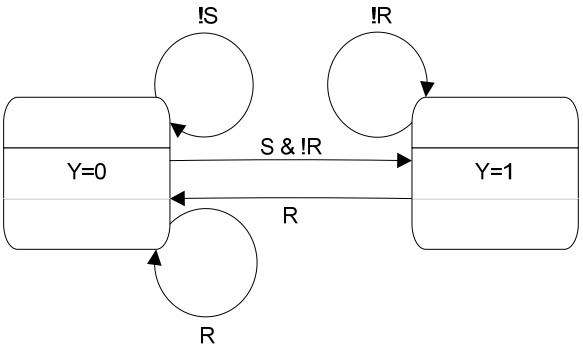


Рис. 1. Граф переходов автомата ATrig

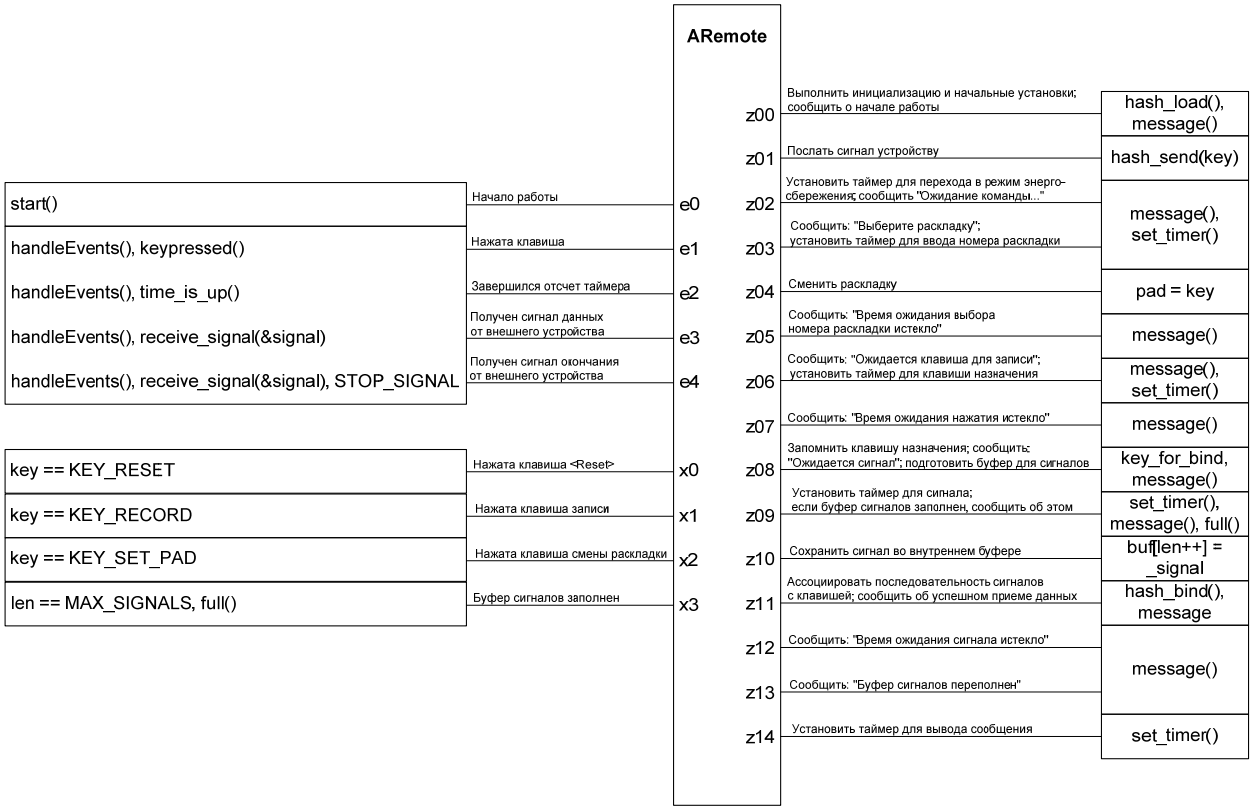


Рис. 2. Схема связей автомата ARemote

Во всех четырех методах вложенные автоматы считаются частями исходного. При построении модели Крипке это означает, что если в состояние *s* автомата *A* вложен автомат *B*, то все ребра модели Крипке, входящие в позицию, соответствующую состоянию *s* модели *A*, следует перенаправить в стартовое состояние модели *B*, а у всех ребер, исходящих из состояния *s* модели *A*, следует изменить начало на терминальную позицию модели *B*. Обратим внимание, что в различные состояния автомата *A* могут быть вложены различные «копии» одного и того же автомата *B*. В этом случае для каждой копии создается своя модель Крипке (все эти модели изоморфны друг другу), и перенаправление ребер выполняется для нее.

После построения модели Крипке для автоматной модели выполним следующие действия. Добавим во множество атомарных предложений *AP* имена всех автоматов системы (множество этих имен обозначим через *Names*). Далее для каждой позиции

полученной модели добавим в отношение *Label* пометку с атомарным предложением (элементом множества *Names*), соответствующим имени автомата, которому это состояние принадлежит. Эти действия предназначены для того, чтобы в формулах темпоральной логики можно было различать, какой именно из автоматов системы выполняется на данном участке пути.

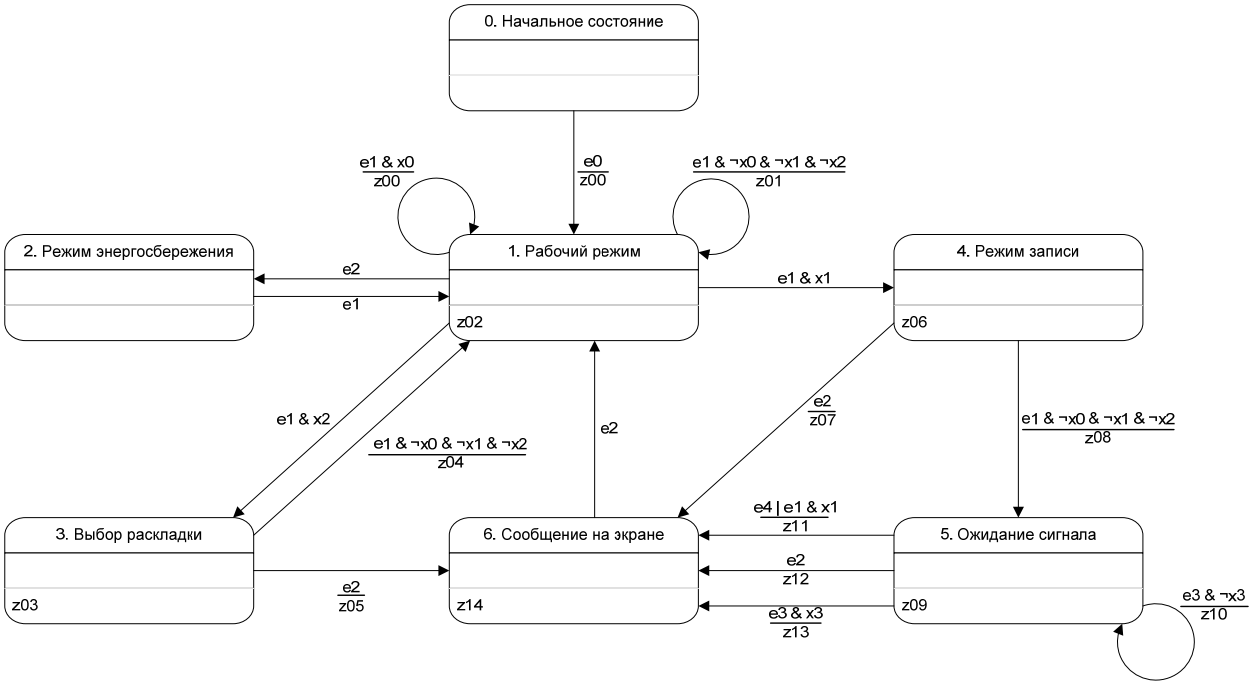


Рис. 3. Граф переходов автомата `ARemote`

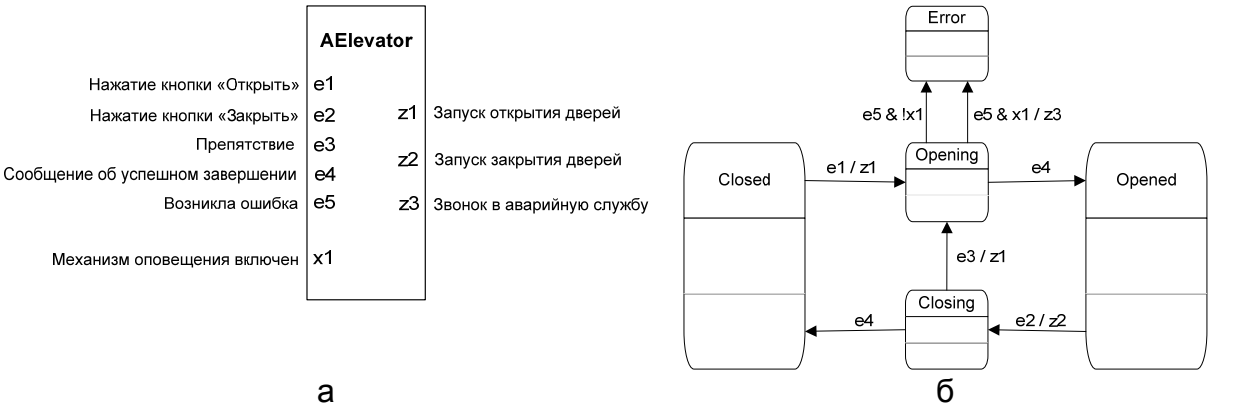


Рис. 4. Схема связей (а) и граф переходов (б) автомата AElevator

Метод атомарных переходов

Множества атомарных предложений *AP* и состояний *S* в модели Крипке, построенной по данному методу, совпадают с множеством состояний исходного автомата, отношение переходов $\rightarrow$ совпадает с отношением переходов исходного автомата, а помечающее отношение *Label* является тождественным (*Label* = *Id*). Это означает, что каждое состояние помечено единственной переменной, которая заведена специаль-

но для него, а все метки на переходах «удаляются» (рис. 5). Общий размер такой модели линеен по отношению к размеру автомата.

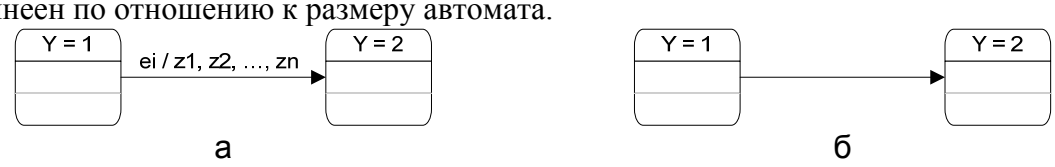


Рис. 5. Переход между состояниями до преобразования (а) и после него (б)

Модели Крипке, полученные по этому методу для автоматов `ARemote` и `AElevator`, изображены, соответственно, на рис. 6 и 7.

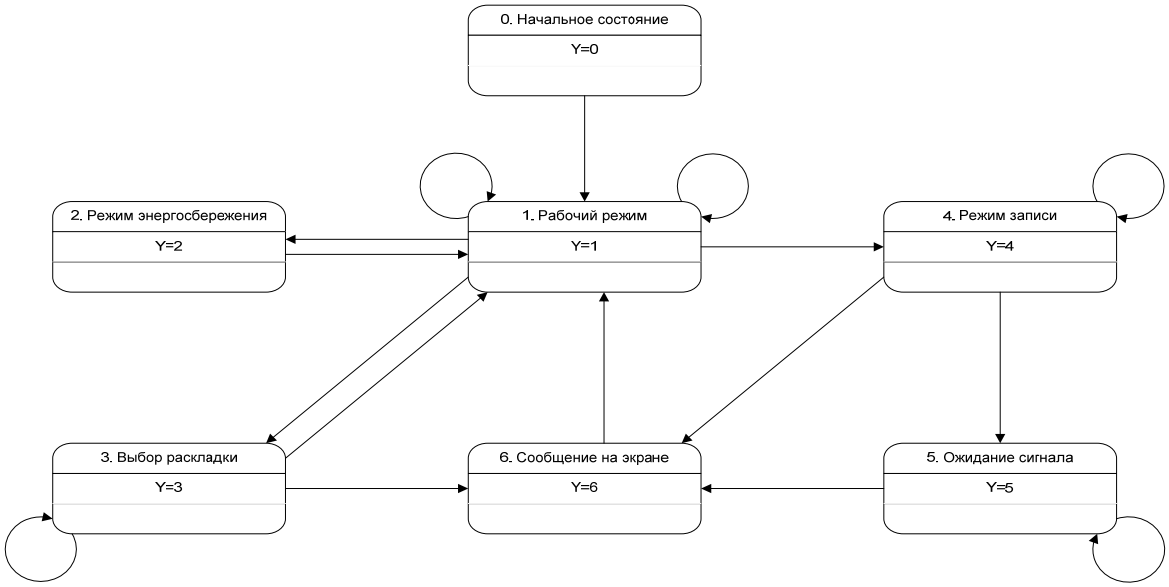


Рис. 6. Сокращенная модель без событий и входных воздействий для автомата `ARemote`

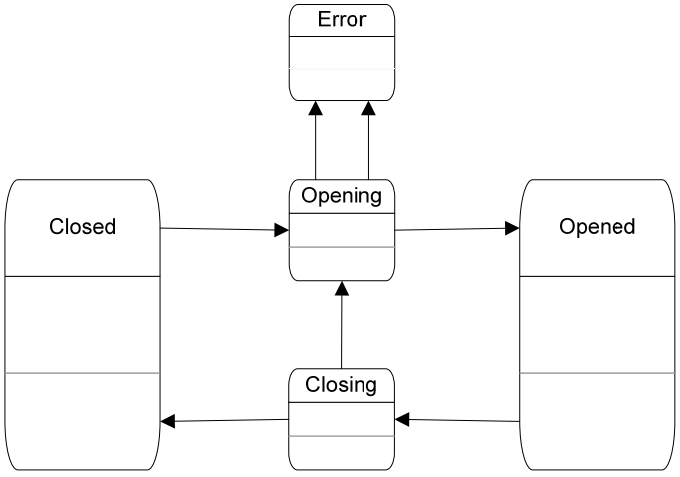


Рис. 7. Сокращенная модель без событий и входных воздействий для автомата `AElevator`

Приведем теперь примеры *CTL*-формул, для которых верификация позволяет проверить выполнимость на модели.

Для автомата ARemote можно взять формулу ¬E[¬(Y=6) U (Y=1)]. Она утверждает, что в состояние 1 нельзя попасть, минуя состояние 6 (нельзя попасть в рабочий режим, минуя сообщение на экране). Эта формула выполняется в состояниях 4, 5 и 6 модели Крипке и исходного автомата и только в них.

Для автомата AElevator проверим формулу AG(Closing → AX Closed), которая означает, что если дверь начала закрываться, то на следующем шаге она обязательно закроется. Эта формула неверна во всех состояниях модели и автомата. Например, последовательность состояний (Closing, Opening) является опровергающей (контрпримером) для этой формулы.

Особенностью данного метода является то, что комбинаторные свойства у модели не отличаются от таковых у исходного автомата. Следовательно, структура Крипке для композиции автоматов не отличается от композиции структур Крипке отдельных автоматов. Сведения о композиции структур Крипке также изложены в работе [13].

Метод установки состояний на событиях и выходных воздействиях

В данном методе множество атомарных предложений *AP* равно {Y1, Y2, …} ⋃ {e1, e2, …} ⋃ {z1, z2, …} ⋃ {*InState*, *InEvent*, *InAction*}. На первом шаге положим множество *S* равным множеству состояний исходного автомата, и для каждого состояния s добавим в отношение *Label* две пометки: (s, s) и (s, *InState*).

После этого для каждого состояния s выполняем следующую операцию. Пусть s содержит выходные воздействия $z_{s[1]}$, …, $z_{s[n]}$, которые выполняются при входе в s. Добавим в модель *n* состояний: {r_1, …, rn} и *n* переходов: $r_1 \to r_2$, …, $rn_{-1} \to rn$, rn → s, в отношение *Label* добавим пометки (r_k, $z_{s[k]}$), (r_k, InAction) для всех *k* от 1 до *n*. Далее, при добавлении ребер в модель на следующих этапах, каждое ребро, идущее в s, будем перенаправлять в r_1. Эту операцию назовем разделением выходных переменных и состояний. Она будет выполняться также в третьем и четвертом методе, описанным в следующих разделах.

Пример такого преобразования проиллюстрирован на рис. 8.

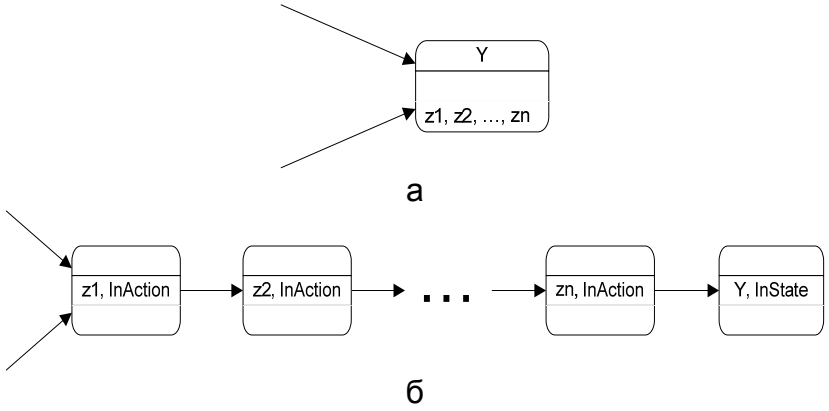


Рис. 8. Состояние с выходными воздействиями до преобразования (а) и после (б)

Далее для каждого ребра r исходного автомата с пометкой (ei & … / zi[1], …, zi[n]), ведущего из состояния p в состояние q, добавим в модель *n* + 1 состояние {re, r1, …, rn}, *n* + 2 перехода: p → re, re → r1, r1 → r2, …, rn-1 → rn, rn → q, а в отношение *Label* добавим пометки (re, ei), (re, *InEvent*), (rk, zi[k]), (rk, *InAction*) для всех *k* от 1 до *n* (рис. 9).

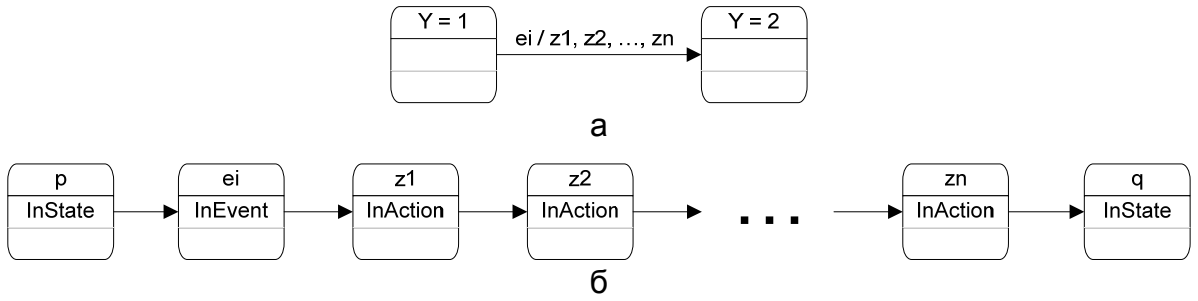


Рис. 9. Переход между состояниями до преобразования по второму методу (а) и после него (б)

Модели Крипке, построенные рассматриваемым методом из автоматов `ARemote` и `AElevator`, изображены на рис. 10 и 11. Размер модели, полученной таким способом, линеен по количеству вхождений переменных в условия на переходах.

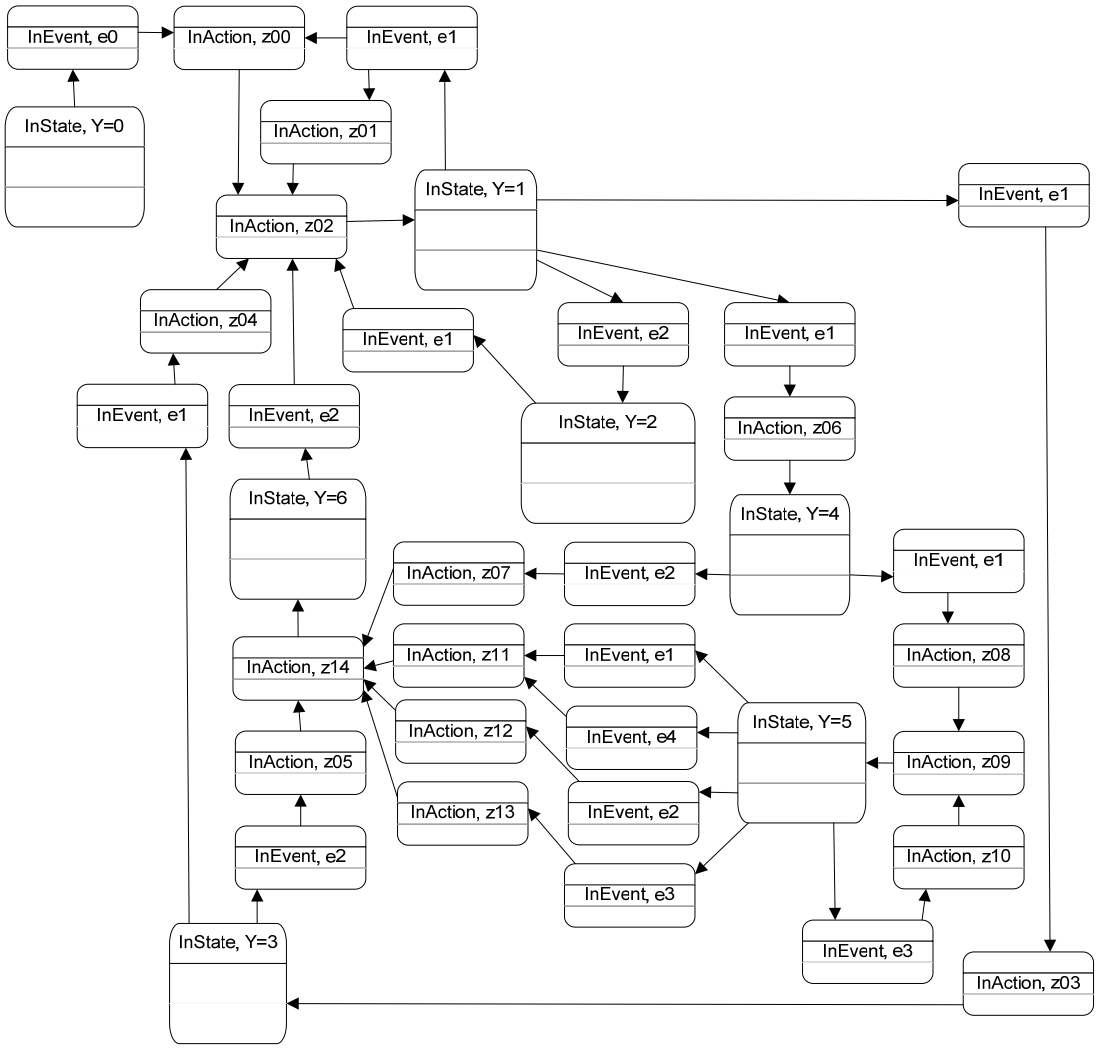


Рис. 10. Модель Крипке, полученная из автомата `ARemote` по второму методу

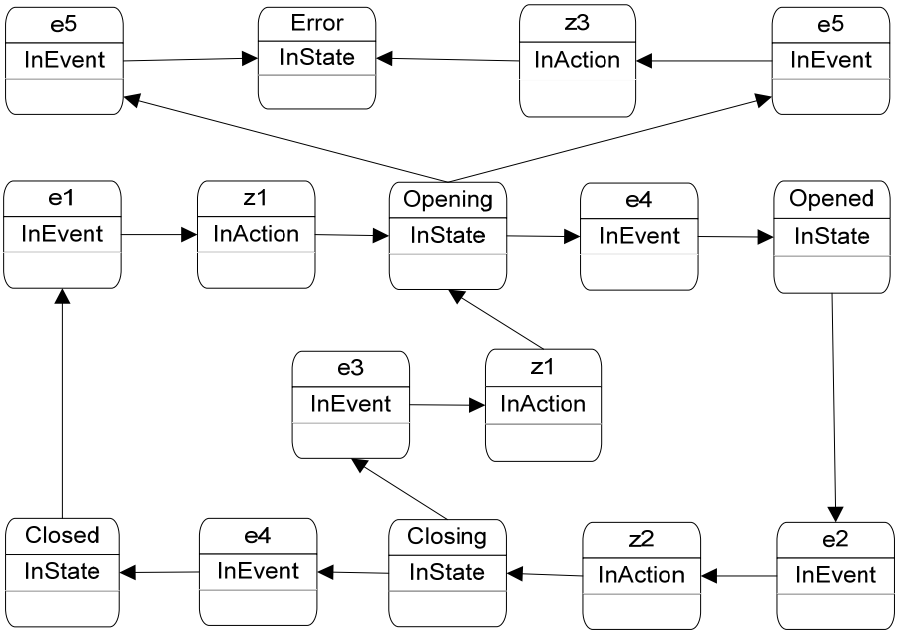


Рис. 11. Модель Крипке, полученная из автомата `AElevator` по второму методу

Приведем пример. Проверим выполнимость формулы AG(*InState* → → (¬*InAction*) W *InEvent*). Во всех состояниях модели Крипке для любого автомата можно убедиться, что в каком бы состоянии автомат ни находился, выходное воздействие не может наступить раньше, чем произойдет некоторое событие.

Метод полного графа переходов

Метод, описанный в этом разделе, является наиболее выразительным в том смысле, что с его помощью можно верифицировать больше всего свойств.

В данном методе множество атомарных предложений *AP* равно {`Y1`, `Y2`, …} ⋃ {`e1`, `e2`, …} ⋃ {`x1`, `x2`, …} ⋃ {`z1`, `z2`, …} ⋃ {*InState*, *InEvent*, *InAction*}.

Как и в предыдущем случае, начиная с множества *S*, равного множеству состояний исходного автомата, добавим в отношение *Label* для каждого элемента `s` ∈ *S* две пометки: (`s`, `s`) и (`s`, *InState*). После этого выполним разделение выходных переменных и состояний (как во втором методе) и приступим к добавлению в модель информации о переходах между состояниями исходного автомата. Для этого рассмотрим набор из всех булевых переменных в исходном автомате (состоящий из событий и выходных воздействий). Для каждого состояния `p` и каждой двоичной последовательности, которую можно присвоить переменным этого набора, введем понятие сценария, который должен произойти в этом состоянии при условии, что значением набора стала данная последовательность. Сценарий этот можно описать на естественном языке следующим образом: в состоянии p произошло событие `ei`, при этом входные воздействия `xj[1]`, …, `xj[t]` (и только они) оказались истинными, после чего были вызваны выходные воздействия `zk[1]`, …, `zk[u]`, и автомат перешел в состояние `q`. Для каждого такого сценария (обозначим его буквой `r`) создадим *u* + 1 дополнительное состояние {`re`, `r1`, …, `ru`} и *u* + 2 перехода: `p` → `re`, `re` → `r1`, `r1` → `r2`, …, `ru-1` → `ru`, `ru` → `q`, а в отношение *Label* добавим пометки (`re`, `ei`), (`re`, `xj[j*]`) (`re`, *InEvent*), (`rk*`, `zk[k*]`), (`rm`, *InAction*) для всех *j** от 1 до *t* и *k** от 1 до *n*.

Таким образом, перебор сценариев будет эквивалентен перебору двоичных наборов, которые могут быть присвоены входным воздействиям, и такой перебор требуется выполнить для каждого состояния. Количество состояний полученной модели ограничено снизу числом

количество_состояний_в_исходном_автомате × (количество_двоичных_наборов + 1),

а число соответствующих двоичных наборов есть $2^{\text{количество\_входных\_переменных}}$.

Пример построения модели Крипке по автомату `ATrig` с помощью данного метода приведен на рис. 12.

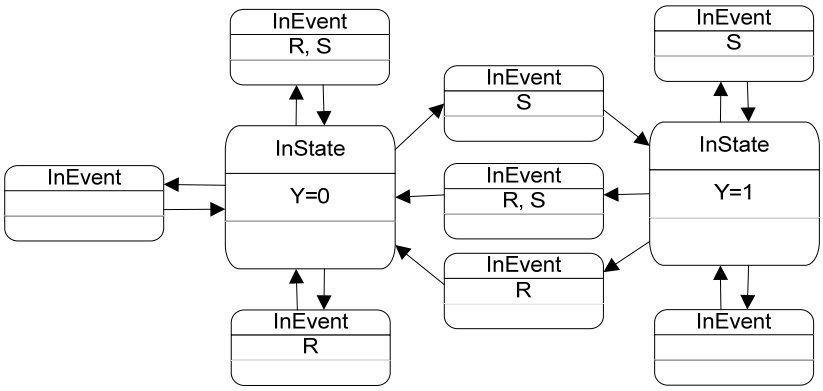


Рис. 12. Полный граф переходов для автомата `ATrig`

Теперь можно проверить, например, свойство $\neg(S \vee R) \rightarrow EX\ EX\ (Y{=}1)$, которое утверждает, что если оба управляющих сигнала отсутствуют, то, преодолев переход, на следующем шаге можно оказаться в состоянии `1`. Это свойство выполняется во всех состояниях, за исключением тех, которые помечены только служебным словом *InEvent*.

Метод редуцированного графа переходов

В данном методе множество *AP* равно $\{Y_1, Y_2, \ldots\} \cup \{e_1, e_2, \ldots\} \cup \{x_1, x_2, \ldots\} \cup \{!x_1, !x_2, \ldots\} \cup \{z_1, z_2, \ldots\} \cup \{\mathtt{InState}, \mathtt{InEvent}, \mathtt{InAction}\}$.

Начинаем, как и в предыдущих методах, с того, что присвоим переменной *S* множество состояний исходного автомата и для каждого состояния $s \in S$ добавим в отношение *Label* две пометки: (`s`, `s`) и (`s`, `InState`). После этого выполним разделение выходных переменных и состояний, аналогичное тому, которое описано во втором методе (установки состояний на событиях и выходных воздействиях).

Рассмотрим множество следующих символов: $\{x_1, !x_1; x_2, !x_2; x_3, !x_3; \ldots\}$. Можно сказать, что это множество всех литералов, составленных из входных переменных. Следует различать смысл знаков $\neg$ и `!`. Первый из них означает выполнение операции логического отрицания, а второй интерпретируется просто как символ (часть строки $!x_i$). Тогда для каждого ребра `r` исходного автомата, ведущего из состояния `p` в состояние `q` с пометкой $e_i\ \&\ h_{j[1]}\ \&\ h_{j[2]}\ \&\ h_{j[3]}\ \&\ \ldots\ h_{j[m]}\ /\ z_{i[1]}, \ldots, z_{i[n]}$, где либо $h_{j[j^*]} = x_{j[j^*]}$, либо $h_{j[j^*]} = !x_{j[j^*]}$ (это значит, что $h_{j[j^*]}$ есть либо входная переменная, либо ее отрицание), добавим в модель $n + 1$ состояние $\{r_e, r_1, \ldots, r_n\}$, $n + 2$ перехода: $p \rightarrow r_e,\ r_e \rightarrow r_1,\ r_1 \rightarrow r_2,\ \ldots,\ r_{n-1} \rightarrow r_n,\ r_n \rightarrow q$, а в *Label* добавим пометки (r_e, e_i), $(r_e, \mathtt{InEvent})$, $(r_k, z_{i[k]})$, $(r_k, \mathtt{InAction})$ для всех k от 1 до n, а также пометки $(r_e, h_{j[1]})$, $(r_e, h_{j[2]})$, …, $(r_e, h_{j[m]})$.

Пример такого преобразования отражен на рис. 13.

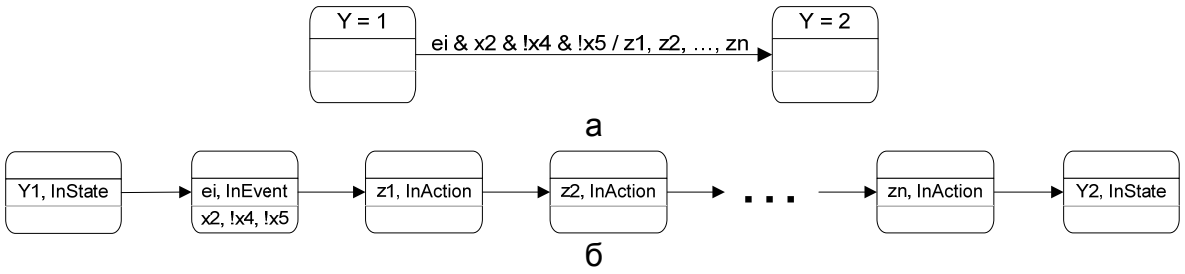


Рис. 13. Переход между состояниями до преобразования по методу редукции (а) и после него (б)

Из рисунка видно, что для тех состояний, которые были построены из событий, во множество атомарных предложений были добавлены входные переменные в том виде, в котором они записаны на переходах автомата (вместе с отрицаниями, если есть).

На рис. 14, 15 и 16 показаны модели Крипке, построенные с помощью редукции графа переходов из автоматов `ARemote`, `AElevator`, `ATrig` соответственно.

Размер получившейся модели линеен, так же, как и при установке состояний на событиях и выходных воздействиях.

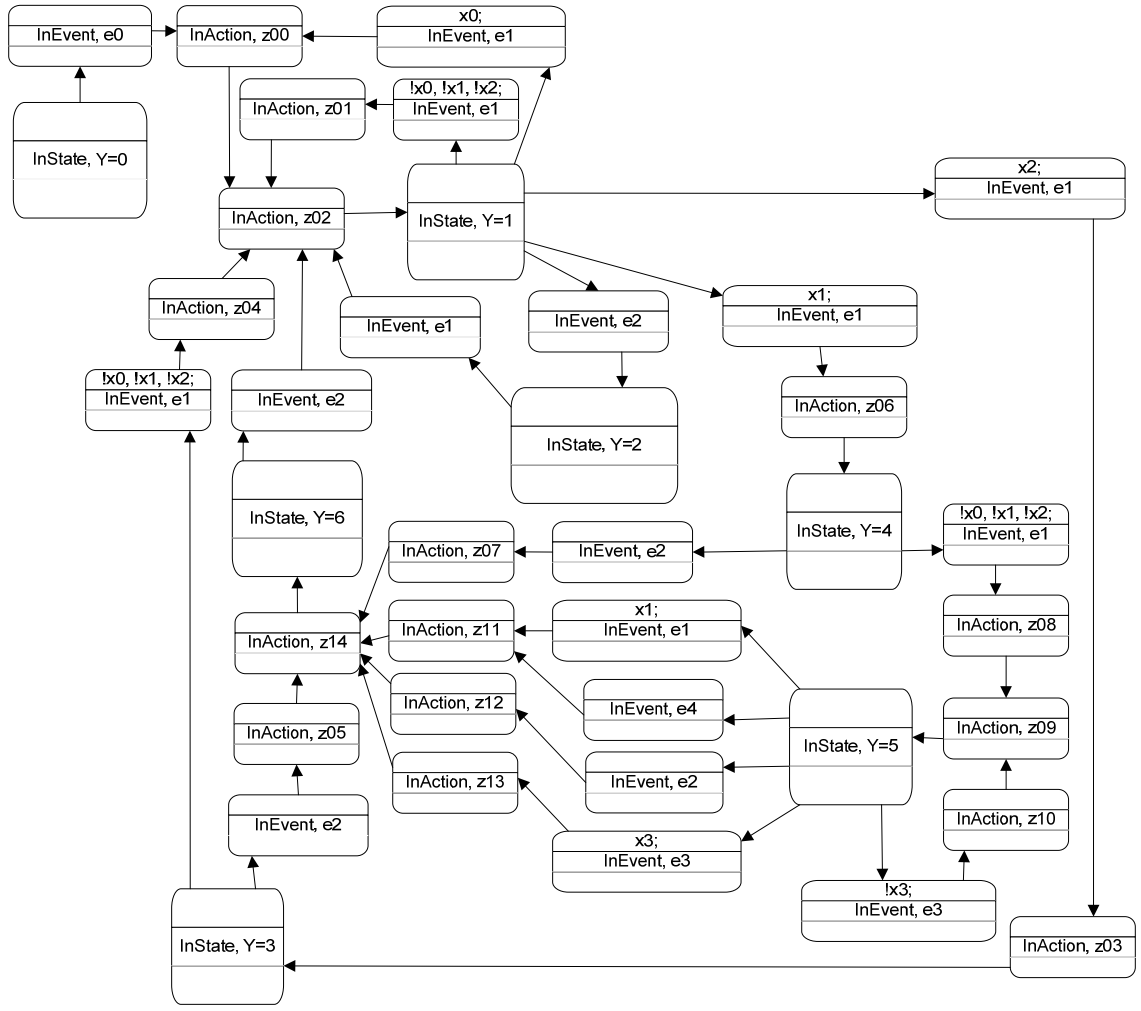


Рис. 14. Редукция графа переходов для автомата `ARemote`

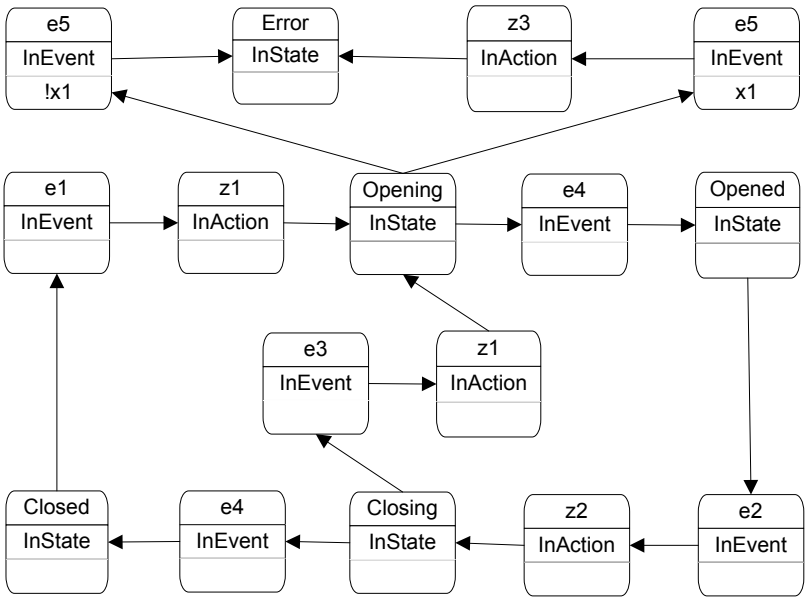


Рис. 15. Редукция графа переходов для автомата `AElevator`

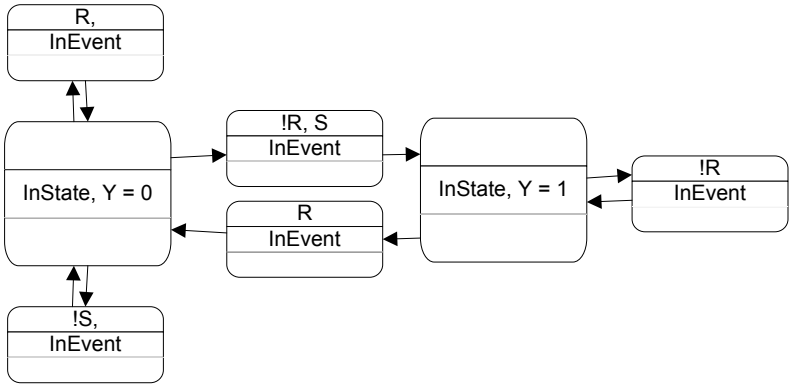


Рис. 16. Редукция графа переходов для автомата `ATrig`

Теперь разберем построение и интерпретацию *CTL*-формул для редуцированных моделей.

CTL-семантика в данном методе будет немного отличаться от общепринятой: перед тем, как выполнять верификацию *CTL*-формулы, ее следует привести к определенному («каноническому») виду. Вначале в ней нужно удалить все парные отрицания (путем замены подформул вида $\neg\neg f$ на f). После этого все входные воздействия, которые присутствуют в формуле без отрицания, необходимо предварить двумя отрицаниями: одно из них синтаксическое, другое логическое (это значит, что нужно заменить литералы вида x_i на формулы $\neg !x_i$). Только после этих модификаций результирующую формулу можно верифицировать методами, предназначенными для языка *CTL*. Причина такого обращения с литералами заключается в следующем: требуется обеспечить, чтобы любая ссылка на несущественную переменную, которая упомянута в *CTL*-формуле, давала истинный результат (несущественными переменными на данном переходе называются те входные переменные исходного автомата, значение которых не проверяется на этом переходе).

Приведем пример для автомата `ARemote`. Пусть требуется проверить свойство: «существует способ попасть в рабочий режим с заполненным буфером сигналов». В терминах языка *CTL* с исходной семантикой данное свойство может быть записано следующим образом: E[`x3` U (`Y=1`)]. Эта формула не выполняется в состоянии `Y=5` (рис. 14). На это, правда, и не стоило рассчитывать. Преобразуем формулу согласно нашему методу: E[¬`!x3` U (`Y=1`)].

В формулу языка *CTL* было внесено отрицание другое атомарного предложения, являющегося «синтаксическим отрицанием» исходного. Преобразованная формула уже верна для состояния `Y=5`.

Таким образом, в методе редукции графа переходов была видоизменена семантика *CTL*. Рассмотренная схема преобразовывала исходную формулу, построенную для новой семантики *CTL*, в новую формулу, для которой применима общепринятая семантика языка *CTL*.

Метод редуцированного графа переходов можно рассматривать как упрощение метода полной системы переходов, которое заключается в том, что двоичные наборы значений несущественных переменных отождествляются. Примеры верификации *CTL*-формул этим методом показывают, что данный подход не снижает существенно способность модели описывать поведение системы по сравнению с предыдущим методом. Использование такой схемы подходит для многих формул.

Преобразование сценария для модели Крипке в сценарий для автомата

После моделирования и верификации требуется обработать результаты проверки модели. Программы, написанные с помощью традиционных методов, имеют достаточно сложные модели, и проводить анализ путей прямо на модели Крипке неэффективно. При интерактивном моделировании совместно с исполнением и визуализацией автомата [14, 15] процесс представления путей в модели Крипке путями в автомате желательно автоматизировать.

После того, как отработала программа-верификатор, необходимо определить выполнимость формул, которые формируют *спецификацию*, на заданных участках автомата. Среди этих участков могут быть состояния, события, выходные воздействия. Сценарий для любой подформулы спецификации представляет собой путь в модели Крипке, иллюстрирующий справедливость или ошибочность данной подформулы. Задача состоит в том, чтобы сценарий, представленный программой для модели Крипке, был представлен в исходном автомате.

Для описанных в настоящей работе методов операция переноса путей из модели Крипке в автомат выполняется однозначно. Действительно, состояния модели, содержащие атомарное предложение `Y=`… или вспомогательное атомарное предложение *InState*, однозначно преобразуются в соответствующие им состояния автомата. Путь между любыми двумя соседними состояниями всегда представляет собой «змейку» из события и выходных воздействий. Любая из этих промежуточных позиций однозначно определяет то главное состояние автомата, из которого эта «змейка» исходит. Из атомарных предложений, которыми помечены состояния «змейки», однозначно восстанавливаются события. Значения существенных входных переменных (тех, которые записаны на переходе) и список несущественных определяется оттуда же (в методе редукции). Последовательным проходом по полученному пути восстанавливается информация о выполнимости литералов, соответствующих выходным воздействиям, очередности этих литералов и о том, как попасть в данное состояние.

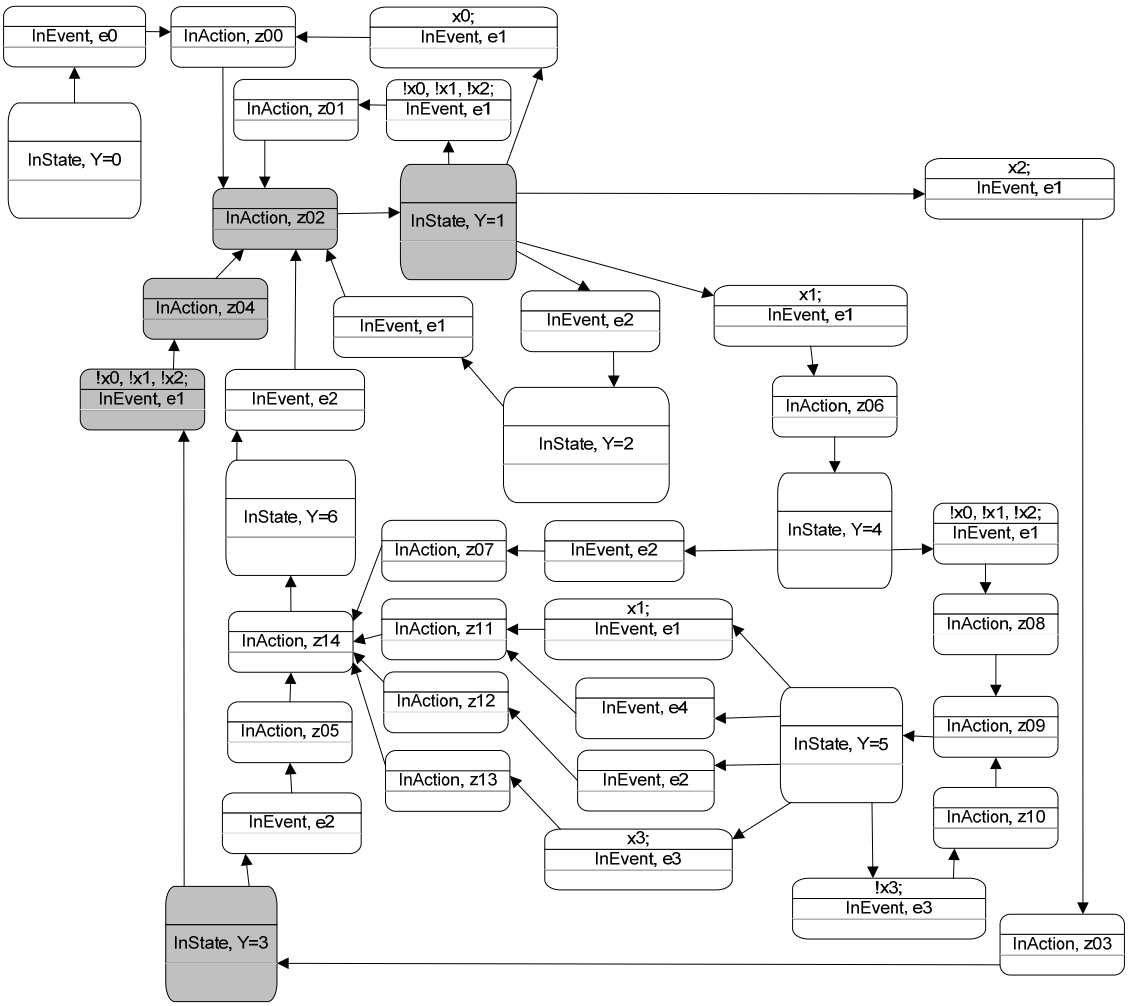


Рис. 17. Путь в модели Крипке

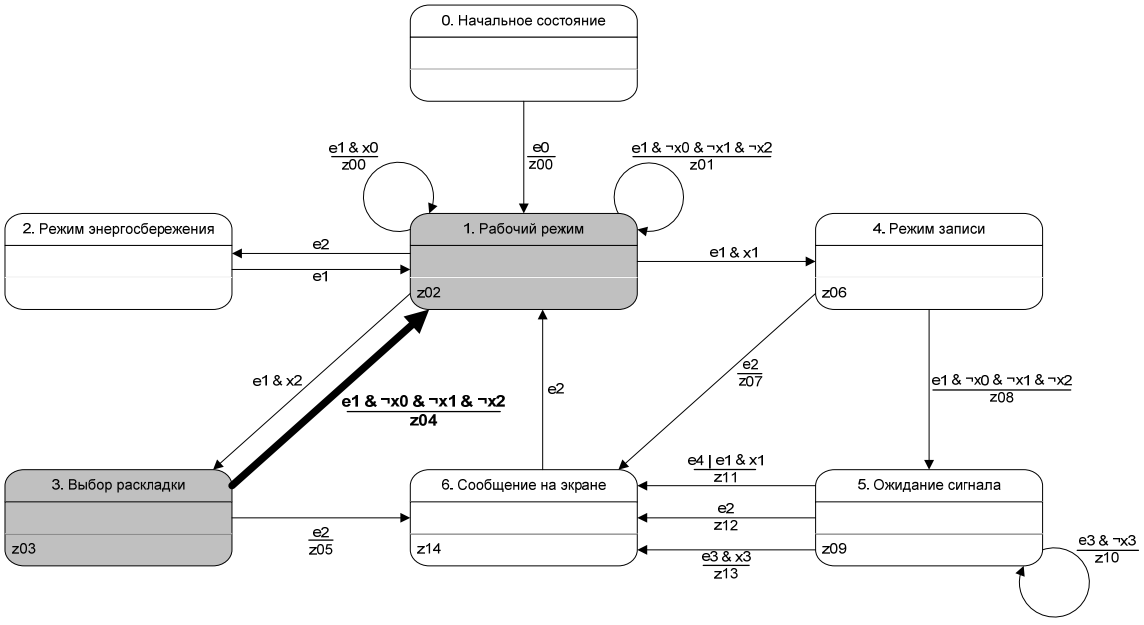


Рис. 18. Путь в исходном автомате ARemote

Рассмотрим пример для автомата `ARemote`. Пусть для состояния `3` выполняется верификация формулы ¬E[¬(`Y=6`) U (`Y=1`)] (в состояние `1` нельзя попасть, минуя состояние `6`). Эта формула в состоянии `3` не выполняется. Верификатор сгенерировал (кратчайший и единственный в данном случае) контрпример, который на рис. 17 выделен серым цветом. Это конечный путь, любое бесконечное продолжение которого удовлетворяет формуле ¬E[¬(`Y=6`) U (`Y=1`)].

Этот же путь, но представленный в исходном автомате, можно увидеть на рис. 18. Промежуточная информация о переходе между состояниями выделена жирностью ребра и его метки.

Заключение

В работе были предложены методы для представления автоматной модели структурами Крипке. В рамках исследований по государственному контракту [11] четвертый из разработанных методов (метод редукции графа переходов) был реализован программно. Также были приведены примеры темпоральных формул, которые можно верифицировать данным методом. Программа позволяет строить трассы, которые подтверждают заданные формулы, начинающиеся с квантора существования пути (или опровергают отрицания этих формул). Рассмотренные примеры показывают, что запрограммированный алгоритм позволяет убеждаться в корректности модели, находить ошибки в случае некорректных формул и находить ошибки в моделях (ряд тестов проводился для намеренно измененной модели, чтобы проверить работу алгоритма в соответствующих ситуациях).

Литература

1. Emerson E. A., Clarke E. M. Using branching time temporal logic to synthesize synchronisation skeletons // Science of Computer Programming 2: 241-266, 1982.
2. Clarke E. M., Emerson E. A. Synthesis of synchronisation skeletons for branching time logic / Logic of Programs, LNCS 131, pp. 52–71, 1981.
3. Лифшиц Ю. Верификация программ и темпоральные логики. Лекция №3 курса «Современные задачи теоретической информатики». – СПбГУ ИТМО, 2005. – Режим доступа: http://logic.pdmi.ras.ru/~yura/modern/03modernnote.pdf
4. Лифшиц Ю. Символьная верификация программ. Лекция №4 курса «Современные задачи теоретической информатики». – СПбГУ ИТМО, 2005. – Режим доступа: http://logic.pdmi.ras.ru/~yura/modern/04modernnote.pdf
5. Вельдер С.Э., Шалыто А.А. О верификации автоматных программ на основе метода Model Checking // Информационно-управляющие системы. – 2007. – № 3. – С. 27–38.
6. Roux C., Encrenaz E. CTL May Be Ambiguous when Model Checking Moore Machines. UPMC – LIP6 – ASIM, CHARME, 2003. – Режим доступа: http://sed.free.fr/cr/charme2003-presentation.pdf
7. Finite state machine. – Режим доступа: http://en.wikipedia.org/wiki/Finite_state_machine
8. Mealy machine. – Режим доступа: http://en.wikipedia.org/wiki/Mealy_machine
9. Moore machine. – Режим доступа: http://en.wikipedia.org/wiki/Moore_machine
10. Вельдер С.Э., Бедный Ю.Д. Универсальный инфракрасный пульт для бытовой техники. Курсовая работа. – СПбГУ ИТМО, 2005. – Режим доступа: http://is.ifmo.ru/projects/irrc/

11. Разработка технологии верификации управляющих программ со сложным поведением, построенным на основе автоматного подхода. НИР, выполняемая по гос. контракту № 02.514.11.4048 от 18.05.2007.
12. Козлов В.А. , Комалева О.А.. Моделирование работы банкомата. – СПбГУ ИТМО, 2006. – Режим доступа: http://is.ifmo.ru/unimod-projects/bankomat/
13. Margaria T. Model Structures. Service Engineering – SS 06. – Режим доступа: https://www.cs.uni-potsdam.de/sse/teaching/ss06/sveg/ps/2-ServEng-Model-Structures.pdf
14. Сайт проекта UniMod. – Режим доступа: http://unimod.sf.net
15. Сайт eVelopers Corporation. – Режим доступа: http://www.evelopers.com

УДК 004.4'242

ВЕРИФИКАЦИЯ ПРОГРАММ, ПОСТРОЕННЫХ НА ОСНОВЕ АВТОМАТНОГО ПОДХОДА С ИСПОЛЬЗОВАНИЕМ ПРОГРАММНОГО СРЕДСТВА *SMV*

Е.А. Курбацкий
(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В работе рассматривается метод верификации программ, построенных на основе автоматного подхода с использованием метода проверки на моделях (*Model Checking*). Для проверки модели используется программное средство *SMV*. При предлагаемом подходе система переходов не строится в явном виде, что позволяет верифицировать программы с большим числом состояний.
Ключевые слова: верификация программ, верификация на моделях, автоматное программирование

Введение

В работе [1] описан метод верификации программных систем, основанный на моделях (*Model Checking*). При использовании этого метода строится система переходов с конечным числом состояний. Свойства модели выражаются на языке темпоральной логики, после чего проверяется соответсвие модели свойствам. Одной из основных проблем методов *Model Checking* является необходимость построения конечной модели для проверки. Эта процедура может быть весьма сложной, а построенная в результате модель может обладать огромным числом состояний, что затрудняет верификацию. При попытке уменьшить число состояний путем абстрагирования от некоторых деталей реализации реальной программы возникает проблема адекватности полученной модели, а, следовательно, и проблема корректности результата верификации.

При использовании автоматного программирования [2] проблема, указанная выше, решается на этапе проектирования программы. В автоматной программе все состояния разделены на два класса [3] – управляющие и вычислительные. При этом управляющие состояния описываются набором конечных автоматов. Набор взаимодействующих автоматов уже является моделью, которая адекватна автоматной программе. Эта модель имеет конечное число состояний, что является необходимым условием для ее верификации.

Данная проблема рассматривалась в ряде работ. В работе [4] описан способ проверки одного автомата. В работе [5] приводится способ проверки системы автоматов. При этом предлагается использовать программное средство *SPIN* [6].

Постановка задачи

Рассматривается задача проверки свойств программы, построенной на основе автоматного подхода [2]. Существует большое число программ, реализующих алгоритмы верификации моделей. В данной работе предлагается использовать программное средство *SMV* (*Symbolic Model Verifier*) [7]. Верификатор *SMV* предназначен для проверки того, что система переходов удовлетворяет требованиям, заданным на языке ветвящейся темпоральной логики C*TL*. В верификаторе *SMV* применяется описанный в работе [1] символьный алгоритм верификации моделей, основанный на упорядоченных двоичных диаграммах решений (*Ordered Binary Decision Diagram – OBDD)*.

Необходимо построить программу, на вход которой поступает система автоматов и свойства на языке темпоральной логики, программа проверяет соответствие модели свойствам и возвращает контрпример, если свойства системы нарушены. В верификаторе *SMV* для описания модели используется одноименный язык. Таким образом, задача сводится к следующим подзадачам:

- преобразовать программу в модель на языке *SMV*;
- преобразовать требования к системе в формулы темпоральной логики;
- запустить программу-верификатор *SMV;*
- преобразовать контрпример к модели в контрпример в автоматной программе.

На рис. 1 изображена схема предлагаемого подхода.

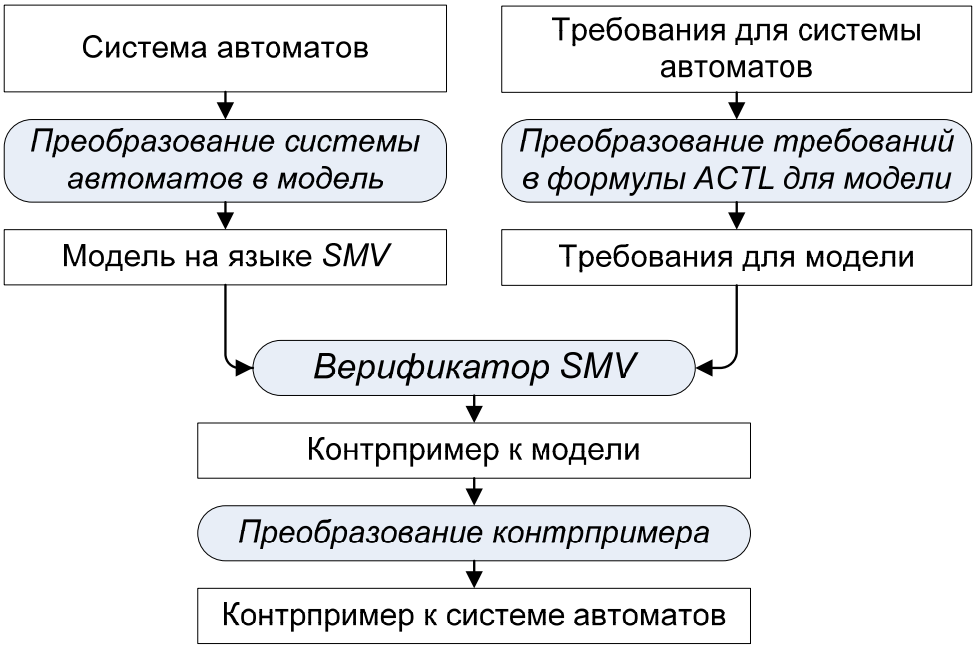


Рис. 1. Предлагаемый подход

Преобразование системы автоматов в модель

Построение модели выполняется в два этапа.

1. Так как система переходов допускает пометки только в вершинах, то для каждого автомата строится модель автомата. Вводятся дополнительные состояния на переходах, которые соответствуют действиям, выполняемым автоматом на переходах.
2. Полученные модели автомата объединяются в одну систему переходов и записываются как переменные и правила переходов между ними на языке SMV.

Рассмотрим задачу построения модели автомата. В автомате выделяются, помимо основных состояний, множество его *промежуточных состояний*, в которых автомат пребывает во время перехода из одного основного состояния в другое. В промежуточных состояниях будет храниться информация о том, какое действие автомат совершает. Промежуточное состояние автомата фиксируется каждый раз, когда автомат совершит одно из следующих действий:

- вызовет выходное воздействие, при этом в состоянии указывается, какое выходное воздействие вызывается;
- вызовет другой автомат, при этом в состоянии указывается, какой автомат и с каким событием вызывается.

Если никаких действий на переходе не выполняется, то выделяется одно дополнительное состояние, для того чтобы идентифицировать данный переход автомата.

Состояниями модели автомата будут состояния исходного автомата и *промежуточные состояния*. Условие и событие, записанные на переходе автомата, записываются на переходе модели из основного состояния в промежуточное. Промежуточные состояния нумеруются так, чтобы начальное состояние автомата имело номер 0. На рис. 2 приведен пример выделения промежуточных состояний для одного перехода.

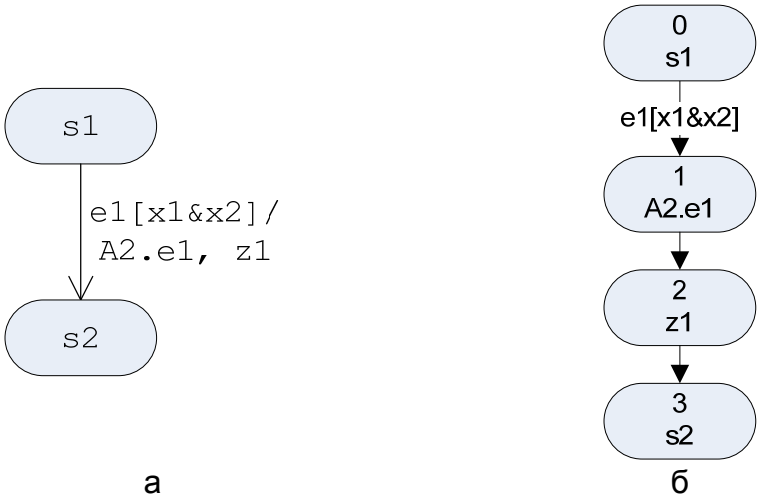


Рис. 2. Выделение промежуточных состояний на переходе.
Исходный переход (а), преобразованный переход (б)

Рассмотрим задачу преобразования нескольких моделей автоматов в систему переходов. Состояние модели будет описваться набором переменных. Каждому автомату A_k сопоставим переменную $State_k$. Для всей системы автоматов в целом введем переменные *Active, Event,* x_k $(0 \le k < m)$, где m – число входных переменных. Введенные переменные имеют следующий смысл:

- переменная *Statek* содержит состояние модели, в котором может находиться каждая из *n* моделей. Каждая переменная может принимать значения от нуля до количества состояний модели;
- переменная *Active* содержит номер автомата, активного в данный момент, или ноль, если никакой автомат не активен;
- переменная *Event* содержит имя события, которое передано автомату в данный момент. Событие может быть передано от внешнего источника или от другого автомата в результате вызова. Если в данный момент никакое событие не передается, то переменная *Event* принимает значение ноль;

- переменные *x1*, *x2*, *xm* ... соответствуют входным воздействиям. Каждая переменная может принимать значения ноль (ложь) или один (истина).

Каждое состояние модели представляет собой объединение переменных $State_k$ $(1 \le k \le n)$, *Active*, *Event,* x_k $(0 \le k < m)$. Так как состояние задается набором переменных, то можно не строить систему переходов в явном виде. Таким образом, можно проверять модели с большим числом состояний.

Чтобы задать систему переходов, требуется выразить отношение между текущими и следующими значениями переменных. Зададим начальное значение всех переменных:

- переменные *Statek* для всех *k* содержат состояния модели, соответствующие начальным состояниям всех автоматов;
- переменная *Active* содержит ноль;
- переменная *Event* содержит ноль;
- переменные *x1*, *x2*, ... *xm* содержат входные воздействия, которые могут принимать значения ноль или единица.

Теперь зададим правила переходов. Для $State_k$ запишем следующие правила переходов:

- если автомат активен *(Active = k)*, то изменение значения переменной *Statek* выполняется в соответствии с переходами модели автомата *k*. Переменная *Statek* может изменить свое значение на любое значение, в которое есть переход из текущего состояния в модели автомата, если условие, записанное на переходе, выполняется;
- если никакой переход невозможен, значение переменной *Statek* не изменяется;
- если автомат не активен, то значение переменной остается прежним.

Опишем изменение значения переменной *Active*. Возможно несколько вариантов:

- если переменная *Active* равна нулю, то никакой автомат в данный момент времени не активен, а, следовательно, требуется выбрать автомат, который будет выполняться. Этот выбор происходит недетерминировано из множества всех номеров автоматов. После того, как автомат для выполнения выбран, его номер записывается в переменную *Active*;
- иначе переменная *Active* равна *k*. Это значит, что активен автомат *k*. Для определения значения переменной *Active* необходимо рассмотреть следующее состояние модели *k*. Это состояние содержится в переменной *Statek*;
- если состояние *Statek* – промежуточное состояние, и в нем вызывается автомат *l*, следовательно, он будет активен на следующем шаге. Таким образом, переменной *Active* необходимо присвоить значение *l*;
- если *Statek* является состоянием исходного автомата, следовательно, автомат *k* уже закончил переход. Проверим, нет ли автомата *l*, который вызвал автомат *k*;
- если автомат *l* был вызван каким-то другим автоматом *m*, то управление требуется вернуть к автомату *m*. Следующее значение переменной *Active* будет равно *m*;
- иначе следующее значение переменной *Active* равно нулю;
- иначе значение переменной *Active* не изменяется.

Переменная *Event* принимает следующие значения:

- если переменная *Active* равна *k*, и следующее значение переменной *Statek* соответствует вызову автомата с событием *e*, то следующее значение переменной *Event* равно *e*;
- иначе следующее значение Event равно нулю.

Преобразование системы автоматов в модель для *SMV*

Опишем преобразование моделей автоматов в модель на *SMV*. Каждая модель автомата размещается в отельном модуле с именем, соответсвующим имени автомата. Каждый модуль будет иметь следующие параметры:

- Active – является ли данный модуль активным;
- Event – входящее событие, переданное автомату;
- Ap1, Ap2,... – состояния экземпляров автоматов, с которыми данный модуль взаимодействует.
- x1, x2,... – входные воздействия.

Определение модуля имеет следующий вид:

MODULE name(Active, Event, Ap1, Ap2, …, x1, x2, … xm)

Каждый модуль будет хранить номер текущего состояния модели в переменной *State*. Она описывается `State: 0..p;`. Здесь *p* – число состояний модели данного автомата.

Далее требуется указать правила, по которым будут изменяться значения переменных. Эти правила описываются в секции `ASSIGN`.

Вначале значение переменной *State* = 0. Это записывается как `init(State) = 0;` Для каждого состояния модели указывается, в какие состояния модель может переходить. Это делается при помощи ключевого слова `TRANS`, после него записывается предикат, истинность которого означает наличие перехода. Чтобы составить такой предикат, запишем условия для каждого перехода, объединив их операцией логического или. Для каждого перехода из состояния s_i в состояние s_j, который происходит по событию e_{ij} при условии c_{ij}, записывается:

```
(Active & State = si & next(State) = sj & Event = eij & cij) |
```

Для каждого состояния модели s_i запишем условие того, что ни один переход из данного состояния не возможен.

```
(Active & State = si & next(State) = si & !(Event = ei1 & ci1) &
          !(Event = ei2 & ci2) & ...) |
```

Если автомат не активен, то состояние не изменяется.

```
(!Active & State = next(State))
```

Далее требуется выразить основные состояния автоматов. Это записывается в секции `DEFINE:`

```
DEFINE
  s1 := State = n1;
  s2 := State = n2;
. . .
```

Далее выразим свойство, что модель находится в состоянии автомата:

`inState := State =` *n1* `| State =` *n2* `| …;`

Здесь `s1`, `s2`, ... – имена состояний автомата, а *n1*, *n2*, ... – соответствующие им номера состояний модели.

Переменные, общие для всей системы, записываются в модуле main после ключевого слова `VAR`:

- переменные xk, описывающие входные воздействия, записываются как xk: {0, 1};
- переменные, описывающие экземпляры автоматов записываются как name: name(Active = k, Event, Ap1, Ap2, …, x1, x2, … xm), где *name* – имя автомата *k*, *Ap1*, *Ap2* – имена автоматов, с которыми он взаимодействует, *x1*, *x2*, ..., *xm* – входные воздействия;
- переменная Event описывается следующим образом: `Event: {0, e1, e2, e3, …}.`

Зададим значения переменных. В частности, начальное значение переменной *Active*.

```
init(Active) := 0;
```

Определим следующее значение переменной *Active*:

```
next(Active) := case
```

Если все автоматы не активны, то выберем любой автомат:

```
Active = 0: 1..n;
```

Для всех состояний модели *sk* и всех моделей *Ak*, в которых вызывается автомат:

```
(Active = k & next(Ak.State) = sk): l;
(Active != k & Ak.State = sk & Al.inState): k;
```

Для всех автоматов запишем условие, что он вернет управление, когда закончит переход:

```
(Active = k & next(Ak.inState)): 0;
```

Иначе значение переменной *Active* не изменяется:

```
1: Active;
esac
```

Далее записывается описание переменной *Event*. Начальное значение переменной *Event:*

```
init(Event) := 0;
next(Event) := case
```

Для всех вызовов автоматов с событием *ek* записывается.

```
Active = k & next(Ak.State) = sk: ek;
```

Иначе значение переменной *Event = 0*:

```
1: 0;
esac;
```

Для каждого выходного воздействия *zk* запишем:

```
DEFINE
  Zk = (Active = k1 & A1.State = s1) |
    (Active = k2 & A2.State = s2) | ...
```

Запись требований

Для записи требований используются формулы темпоральной логики *ACTL*. Опишем, как записываются свойства автоматной модели в виде формул *ACTL.*

Введем формулы, которые будут описывать состояния автоматов:

- для записи условия, что автомат *Ak* находится в состоянии sj, достаточно записать Ak.sj;
- условие того, что выполнилось выходное воздействие *z1*, записывается как z1;
- для записи условия, что произошло событие *ei*, достаточно записать Event = ei;

Для записи формул, описывающих состояния автомата, также можно использовать логические операторы. Если `f` и `g` – формулы состояния, то формулами состояния являются:

- f & g – одновременно выполняются f и g;
- f | g – выполняется либо f либо g;
- f xor g – выполняется либо f либо g, но не одновременно;
- !f – не выполняется f;
- f -> g – если выполняется f, то выполняется g;
- f <-> g – тоже что и (f -> g) & (g -> f).

Помимо свойств текущего состояния, в условиях можно использовать темпоральные операторы: `AF`, `AG`, `AU`. Оператор `AX` не используются для записи свойств автоматов, так как в данной модели один переход автомата соответствует неопределенному числу переходов модели. Опишем эти операторы:

- AF f (*Future*) – оператор будущего. Означает, что на всех путях из текущего состояния существует состояние, когда формула f выполнитсяЯ;
- AG f (*Global*) – оператор означает, что данная формула f будет выполняться на каждом пути из текущего состояния в каждом состоянии: f будет выполняться в каждом состоянии, достижимом из текущего состояния;
- A[f U g] (*Until*) – оператор истинен, только если на каждом пути когда-нибудь выполнится формула *g*, а до этого момента всегда будет выполняться формула f.

Преобразование контрпримера

Если опровергаемая формула принадлежит *ACTL*, то при ее невыполнении получим контрпример в системе переходов. Любой контрпример для модели является либо конечным путем, либо путем с конечным началом и циклом.

Каждое состояние является набором переменных *Event,* $State_k$*, Active,* x_k *(*$0 \le k < m$*)*. Предлагается следующий алгоритм для преобразования контрпримера к системе переходов в контрпример к системе автоматов. Контрпример к системе автоматов также представляется в виде последовательности состояний, только вместо значений переменных информация предоставляется в терминах состояний и действий. Для каждого состояния контрпримера выведем следующую информацию о системе автоматов.

Если *Active = 0*, то никакой автомат не активен. Данное состояние не учитывается в контпримере для автомата.

Если *Active != 0*, то выводится название активного автомата. Также выводится действие активного автомата. Выводятся состояния всех автоматов, полученые из переменных $State_k$. Аналогично выводятся значения всех входных воздействий, записанных в переменные x_k, и значение переменной *Event*.

Пример построения модели системы автоматов

Продемонстрируем построение модели на небольшом примере. Рассмотрим два потока, которым необходимо время от времени работать с некоторым ресурсом. При этом необходимо, чтобы в каждый момент времени только один из потоков мог иметь доступ к ресурсу, но не оба сразу. Для моделирования этого примера, потребуются три автомата. Автомат *A0* будет соответствовать ресурсу, а автоматы *W1* и *W2* - потокам. На рис. 3 изображена диаграмма переходов автомата *A0*.

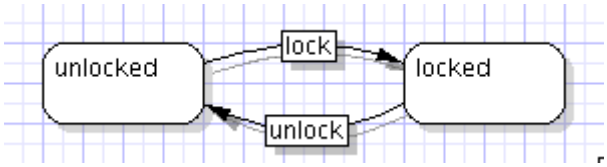


Рис. 3. Автомат A0

Автомат имеет два состояния:

- *unlocked* – начальное состояние означает, что ни один из потоков не использует ресурс. Автомат переходит из этого состояние в состояние *locked* по событию *lock*;
- *locked* – это состояние означает что ресурс в данный момент используется. Автомат переходит из этого состояния по событию *unlock* в состояние *unlocked*.

На рис. 4 приведена диаграмма переходов автомата *W1*. Автомат *W2* устроен аналогично. Изначально оба автомата находятся в состоянии *wait.* После этого один из них

переходит в состояние *work*, в котором он может вызывать выходное воздействие *z1*. По окончанию работы по событию *e1* автомат возвращается в состояние *wait*.

На рис. 5 приведены модели автоматов A0 и W1.

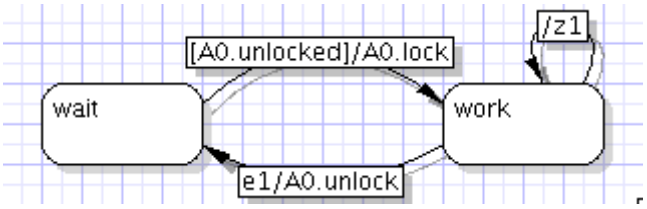


Рис. 4. Граф переходов автомата W1 (автомат W2 устроен аналогично)

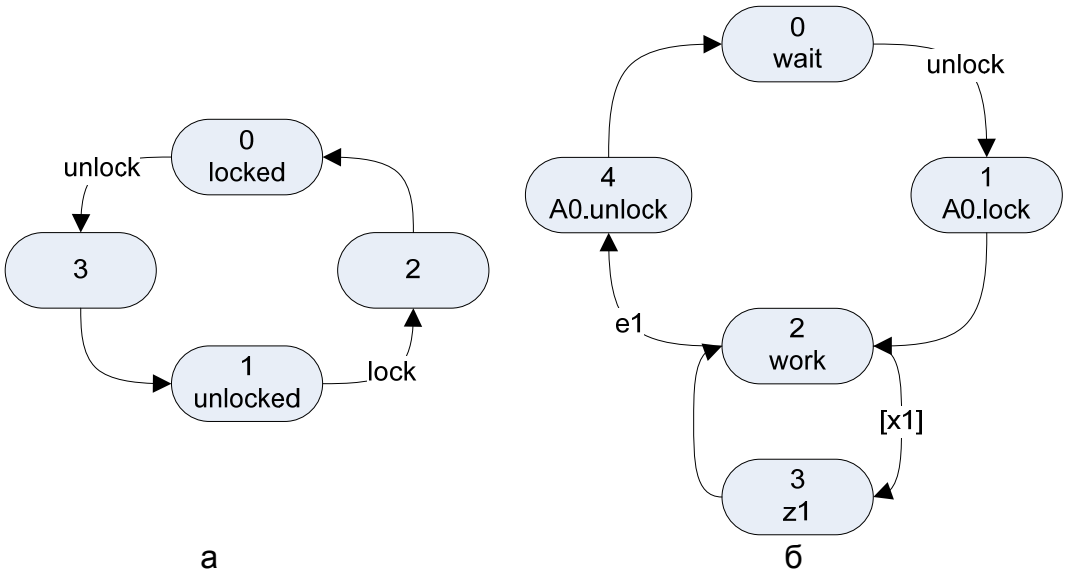


Рис. 5. Модели автоматов A0(а) и W1(б)

Литература

1. Symbolic Model Verifier. – Режим доступа: http://www.cs.cmu.edu/~modelcheck/smv.html
2. SPIN Model checker. – Режим доступа: http://spinroot.com/spin/whatispin.html
3. Шалыто А. А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. – СПб.: Наука. 1998. – Режим доступа: http://is.ifmo.ru/books/switch/1
4. Вельдер С. Э., Шалыто А. А. О верификации простых автоматных программ на основе метода Model Checking //Информационно-управляющие системы. – 2007. – № 3. – С.27–38. – Режим доступа: http://is.ifmo.ru/download/27-38.pdf
5. Кузьмин Е. В. Иерархическая модель автоматных программ // Моделирование и анализ информационных систем. – 2006. – № 1. – С. 27–34.
6. Кларк Э., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. – М.: МЦНМО. 2002.
7. Шалыто А. А., Туккель Н. И. Программирование с явным выделением состояний //Мир ПК. – 2001. – № 8. – С. 116–121, № 9. – С. 132–138.

УДК 004.4’242

ВЕРИФИКАЦИЯ АВТОМАТНЫХ ПРОГРАММ С ИСПОЛЬЗОВАНИЕМ ВЕРИФИКАТОРА *SPIN*

М.А. Лукин, А.А. Шалыто

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

Рассматривается способ использования верификатора *SPIN* для верификации автоматных программ. При использовании этого верификатора модель описывается на языке *Promela*, а требования – на языке *LTL*. В работе предлагается метод автоматизированного преобразования автоматных программ в модели на языке *Promela*. Для преобразования формулы на языке *LTL* в вид, пригодный для использования верификатором, разработано специальное программное средство. Использование предлагаемого метода иллюстрируется на примере верификации модели банкомата.

Ключевые слова: верификация программ, верификация на моделях, автоматное программирование, верификатор SPIN

Введение

В статье описывается методика автоматической верификации визуальных [9] автоматных программ [10]. Для поддержки этой методики разработан инструмент, позволяющий производить верификацию автоматных программ, спроектированных и реализованных на инструментальном средстве *UniMod* [11]. В данной работе верификация производится на основе метода *Model Checking* [12–18]. По данной теме уже проводились исследования [18–20], в которых модель программы строилась вручную. Такой подход имеет следующие недостатки:

- не гарантируется отсутствие ошибок в программе. Гарантируется лишь правильность алгоритма, на основе которого написана программа;
- модель приходится строить вручную, а это трудоемкий процесс.

Отличие предлагаемой методики заключается в том, что верификация программ автоматизирована, что позволяет избежать этих недостатков.

Верификация модели программы — это один из основных методов доказательства правильности программы. Для проведения верификации программы требуется произвести следующие действия.

1. Построить формальную модель, приемлемую для инструментальных средств верификации моделей. Обычно при построении модели абстрагируются от несущественных деталей программы в целях упрощения модели. При этом важно чтобы модель была адекватна программе. В общем случае модель строится вручную. В настоящей работе показано, что для автоматных программ этот этап можно выполнить автоматически.
2. Сформулировать требования к модели – составить спецификацию для модели. Обычно их формулируют на языке темпоральной логики. В настоящей работе требования формулируются на языке *LTL* [15–19, 21]. Эти требования для программ общего вида записать весьма сложно. Для автоматных программ эта задача решается проще.
3. Произвести верификацию модели. Для этого на вход верификатор подаются модель программы и требования к модели. Если модель не соответствует требованиям, пользователь получает трассу ошибки (контрпример), которая может помочь найти ошибку в программе. Иногда ошибка происходит в результате некорректного задания модели или неправильной спецификации (требований). В этом случае трасса ошибки помогает устранить ошибку в моделировании или спецификации. Для программ общего вида переход от контрпримера в терминах модели программы к контрпримеру в терминах самой программы весьма трудоемок. Для автоматных программ он может выполняться автоматически.

Существует большое число верификаторов, многие из которых имеют открытые исходные коды [22]. Одним из наиболее распространенных верификаторов является верификатор *SPIN* [15–17], который будет использоваться в настоящей работе. При его использовании модель описывается на языке *Promela* [15–17], а требования – на языке *LTL*.

Цель настоящей работы – повышение эффективности использования выбранного верификатора применительно к автоматным программам за счет автоматизации построения модели по программе и перехода от контрпримера к программе.

Синтаксис языка Promela

Синтаксис языка *Promela* напоминает синтаксис языка *C*. Модель на языке *Promela* состоит из следующих элементов: объявления типов данных; объявления каналов передачи переменных; объявления переменных; объявления и определения процессов; процесса `init`.

Понятие процесс в данном случае отдаленно можно рассматривать как процедуру, выполняемую в отдельном потоке. Приведем пример объявления и определения процесса:

```
proctype proc(int a; int b) {
byte c; /* локальная переменная */
/* тело процесса */
}
```

Процессы могут иметь параметры и локальные переменные. Процесс может быть запущен в нескольких экземплярах, если для него используется модификатор `active`. Запускаются процессы с помощью модификатора `run`.

Язык *Promela* имеет пять базовых типов данных: `bit`, `bool`, `byte`, `short`, `int`.

Тело процесса является последовательностью операторов. Операторы могут быть *выполнимыми* либо *заблокированными*. *Выполнимый* оператор может быть выполнен *немедленно*. *Заблокированный* оператор – оператор, который не может быть выполнен в данный момент. Такой оператор блокирует выполнение процесса до тех пор, пока он не станет *выполнимым*. Например, оператор

```
x < 7
```

может быть выполнен только когда *x* меньше семи. В противном случае он останавливает выполнение процесса до тех пор, пока условие не выполнится. Некоторые операторы, например, оператор присваивания, *выполнимы* всегда.

Язык *Promela* содержит также операторы ветвления и цикла, синтаксис (табл. 1) которых основан на охраняемых командах Дейкстры [23].

Таблица 1. Операторы ветвления и цикла

`if`
`::guard1 -> S1`
`::guard2 -> S2`
`...`
`:: else -> Sk`
`fi`
	`do`
	`::guard1 -> S1`
	`::guard2 -> S2`
	`...`
	`:: else -> Sk`
	`od`

Поясним работу операторов. Если условие *guardi* истинно, то выполняется действие *Si*. Если одновременно выполняются несколько условий, то происходит недетерминированный выбор одного из них. Если все условия ложны, то выполняется действие *Sk*, соответствующее *else*. Конструкция *else* может отсутствовать. Тогда в случае, если все условия ложны, выполнение процесса блокируется до тех пор, пока хотя бы одно из них не начнет выполняться.

Модель Крипке

Пусть AP — множество атомарных высказываний. Модель *Крипке* [12, 13] над этим множеством – это четверка $M = (S, S0, R, L)$:

- S – конечное множество состояний;
- $S0 \subseteq S$ – множество начальных состояний;
- $R \subseteq S \times S$ – отношение переходов;
- $L : S \to 2AP$ – функция истинности.

Язык LTL

Для записи требований к модели используются языки темпоральной логики, например, *LTL, CTL, CTL**. Как отмечено выше, в настоящей работе используется язык *LTL* (*Linear-Time Logic*). Этот язык состоит из множества атомарных высказываний *p1, p2, ...* $\in AP$, логических операций ($\neg$, $\wedge$, $\vee$, $\to$) и темпоральных операторов. Пусть φ – правильно построенная формула. Тогда темпоральные операторы

- $X\varphi$ (в следующем состоянии φ верно – neXt);
- $G\varphi$ (φ верно всегда – Globally);
- $F\varphi$ (φ когда-нибудь будет верно – Finally);
- $\psi U\varphi$ (ψ будет верно до тех пор, пока не станет верно φ – Until)

– тоже правильно построенные формулы. Темпоральные операторы G и F, нужны для упрощения формул, их можно выразить через U:

- $F\varphi \equiv 1U\varphi$;
- $G\varphi \equiv \neg F\neg\varphi$.

Формулы *LTL* интерпретируются через исполнение системы переходов в модели *Крипке*. Если все пути из начального состояния удовлетворяют формуле φ, будем говорить, что поведение системы удовлетворяет формуле φ. В табл. 2 приведено соответствие стандартного синтаксиса и записи, принятой в верификаторе *SPIN*. Отметим, что *SPIN* не поддерживает оператор X (neXt), так как *SPIN* генерирует вспомогательные и служебные состояния в модели *Крипке*.

Таблица 2. Соответствие стандартного синтаксиса и записи, принятой в верификаторе *r*

[]	G
<>	F
!	¬
U	U
&& или ∧	∧
|| или ∨	∨
->	→
<->	↔

Автомат Бюхи

Пусть AP – множество атомарных высказываний. Автоматом *Бюхи* [23–26] над алфавитом $2AP$ называется четверка $A = (Q, q0, \delta, F)$, где

- Q – конечное множество состояний;
- $q0$ – начальное состояние;
- $\delta \subseteq Q \times 2AP \times Q$ – функция переходов;

- $F \subseteq Q$ – множество допустимых состояний.

Доказано, что для любой *LTL*-формулы можно построить автомат *Бюхи*, который ее выполняет [26]. Более того, он может строиться автоматически. Например, рассмотрим *LTL*-формулу *G(p U q)* (В нотации *SPIN* эта формула записывается `[](p U q)`). Она обозначает, что всегда гарантировано, что *p* остается верным, по крайней мере, до тех пор, пока не станет верным *q*. *SPIN* ее транслирует в конструкцию *never claim:*

```
never { /* [](p U q) */
T0_init:
   if
   :: ((q)) -> goto accept_S9
   :: ((p)) -> goto T0_init
   fi;
accept_S9:
   if
   :: ((((p)) || ((q)))) -> goto T0_init
   fi;
}
```

Эта конструкция соответствует автомату Бюхи, изображенному на рис. 1. Двойная линия обозначает допустимое состояние.

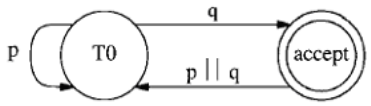


Рис. 1. Автомат Бюхи для формулы *G(p U q)*

С помощью автомата *Бюхи* можно верифицировать модель *Крипке*. С точки зрения верификации автоматных моделей – это наиболее удобный вариант, позволяющий при верификации и спецификации почти полностью ограничиться понятием *конечный автомат*.

Построение модели на языке *Promela*

В данном разделе приводится алгоритм построения модели на языке *Promela* по автоматной программе.

1. Подготовка исходных данных.
   1. Для каждого автомата `Ai` заведем переменную `stateAi`, в которой будет храниться номер текущего состояния. На языке *Promela* это описывается так:
      ```
      int stateAi;
      ```
   2. Заведем одну переменную для событий:
      ```
      int lastEvent;
      ```
   3. Каждому состоянию присвоим уникальный номер, используя сквозную нумерацию для всех автоматов. Переход в новое состояние с номером `k` будет осуществляться присвоением переменной `stateAi` числа `k`:
      ```
      stateAi = k;
      ```
   4. Событие `exx` (`xx` — номер события) на переходе между состояниями описывается в модели следующим кодом:
      ```
      lastEvent = xx;
      ```
2. Построение
   5. Для каждого автомата `Ai` создадим функцию. На языке *Promela* это записывается следующим образом:
      ```
      inline Ai() {
      /* тело функции */
      }
      ```
   6. Для каждого автомата `Ai` в теле созданной функции выполнить шаги 2.3 – 2.9.

7. Определить начальное (стартовое) состояние `s`. Присвоить:

```
stateAi = s;
```

8. Построить цикл:

```
do
::(stateAi == s1) ->
  printf("State Ai 1 : Имя состояния\n");
::(stateAi == s2) ->
  printf("State Ai 2 : Имя состояния\n");
…
::(stateAi == sk) ->
  printf("State Ai k : Имя состояния\n");
od;
```

Здесь `s1`, …, `sk` – номера состояний автомата `Ai`. С помощью инструкции `printf` введена пометка о том, что автомат вошел в данное состояние. Инструкция `printf` аналогична соответствующей инструкции из языка *C*. Пометка используется в дальнейшем для восстановления контрпримера.

9. Если в некоторое состояние `sj` вложен автомат `Am`, то дописать в условие `(stateAi == sj)` пометку о вложенном автомате и вызов функции этого автомата:

```
printf("Calling automaton Am\n");
Am();
```

10. Для каждого состояния `sj` найти все возможные переходы `(sj, sl)` из него. К условию `(stateAi == sj)` дописать конструкцию `if`, а для каждого перехода `(sj, sl)` дописать в конструкции `if` следующее:

```
::stateAi = sl;
 printf("Going to state Ai sl : Имя состояния\n");
```

Это обозначает, что для присваивания `stateAi = sl` необходимое условие выполнено всегда. Также добавляется пометка о переходе в новое состояние.

В результате получится конструкция следующего вида:

```
if
::stateAi = s1
 printf("Going to state Ai s1 : Имя состояния\n");
::stateAi = s2
 printf("Going to state Ai s2 : Имя состояния\n");
…
::stateAi = sk
 printf("Going to state Ai sk : Имя состояния\n");
fi;
```

Эта конструкция обозначает, что присваивание нового номера состояния происходит недетерминировано. Таким образом, верификатор проверит все варианты переходов в новое состояние.

11. Если переход помечен событием `exx`, то дописать выражение

```
lastEvent = xx;
```

и пометку

```
printf("Event = exx\n");
```

Таким образом, в результате получается конструкция вида:

```
if
…
::stateAi = sl;
 printf("Going to state Ai sl : Имя состояния\n");
 lastEvent = xx;
 printf("Event = exx\n");
...
fi;
```

12. Если состояние `st` – конечное, то в условие `(stateAi == st)` дописать инструкцию завершения цикла:

```
break;
```

13. После построения функций для всех автоматов определить стартовый автомат `Ai` и создать процесс, его запускающий. На языке *Promela* это записывается следующим образом:

```
proctype Model() {
  Ai();
}

init {
run Model();
}
```

14. Дописать в модель требования (проверяемые свойства), преобразованные с помощью верификатора *SPIN* с языка *LTL* на язык *Promela*.

Например, рассмотрим автомат из программы [27] (автомат *A3*, рис. 2) и его модель на языке *Promela*, сгенерированную по автомату на основе предлагаемого метода построения модели. В пример не входит построение требований. В данном примере строится модель на языке *Promela* для автомата как для целой программы для того, чтобы продемонстрировать выполнение шага 2.9 алгоритма.

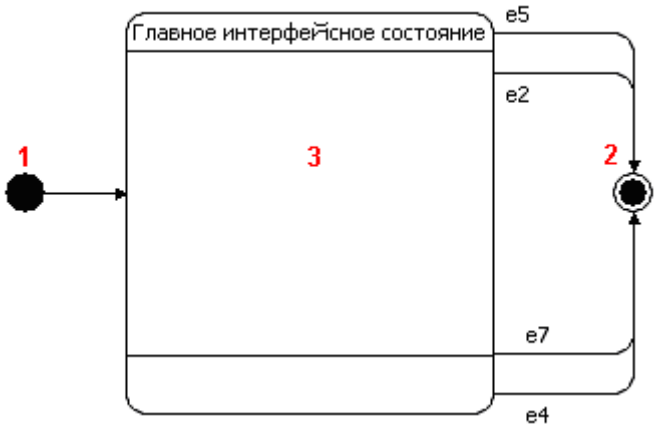


Рис. 2. Автомат *A3*

В автомате *A3* три состояния: `Начальное`, `Конечное` и `Главное интерфейсное состояние`. Перенумеруем их как показано на рис. 2. Модель автомата *A3*, построенная по излагаемой методике, выглядит следующим образом:

```
int stateA3;
inline A3() {
  stateA3 = 1;
  do
  ::(stateA3 == 3) ->
    printf("State A3 3 : Главное интерфейсное состояние\n");
    if
    ::stateA3 = 2;
     printf("Going to state A3 2 : Конечное \n");
     lastEvent = 4;
     printf("Event = e4\n");
    ::stateA3 = 2;
     printf("Going to state A3 2 : Конечное \n");
     lastEvent = 7;
     printf("Event = e7\n");
```

```
      ::stateA3 = 2;
       printf("Going to state A3 2 : Конечное \n");
       lastEvent = 5;
       printf("Event = e5\n");
      ::stateA3 = 2;
       printf("Going to state A3 2 : Конечное \n");
       lastEvent = 2;
       printf("Event = e5\n");
      fi;
    ::(stateA3 == 1) ->
      printf("State A3 1 : Начальное \n");
      if
      ::stateA3 = 3;
       printf("Going to state A3 3 : Главное интерфейсное
           состояние\n");
      fi;
    ::(stateA3 == 2) ->
      printf("State A3 2 : Конечное \n");
      break;
    od;
}
proctype Model() {
   A3();
}
init {
   run Model();
}
```

Преобразование LTL-формул

Каждому элементарному высказыванию присвоим идентификатор *pk*. В модель запишем следующий код:

```
#define pk <элементарное_высказывание>
```

После этого заменим все элементарные высказывания их идентификаторами. В таком виде *LTL*-формула может быть обработана верификатором *SPIN* (при условии, что темпоральные операторы записаны в нотации *SPIN*).

Построение контрпримера

Верификатор *SPIN* автоматически строит контрпример как путь в модели, поданной ему на вход. Так как модель на языке *Promela* эквивалентна исходному автомату, обратное преобразование не требуется.

Общее описание инструментального средства *Converter*

Это инструментальное средство является практической реализацией предложенной методики верификации автоматных программ и позволяет производить полностью автоматическую верификацию автоматных программ. По автоматной программе *Converter* создает модель, в которой отброшены несущественные детали. *LTL*-формула преобразовывается в пригодный для верификатора SPIN вид. Инструментальное средство принимает на вход три параметра: путь к *XML*-описанию автоматной программы, имя файла, в который записывается созданная модель и *LTL*-формулу со спецификацией. На выходе *Converter* выдает файл *report.txt*, в котором записан полный отчет о верификации.

Описание работы инструментального средства

Автоматическая верификация автоматных программ состоит из нескольких этапов (рис. 3).

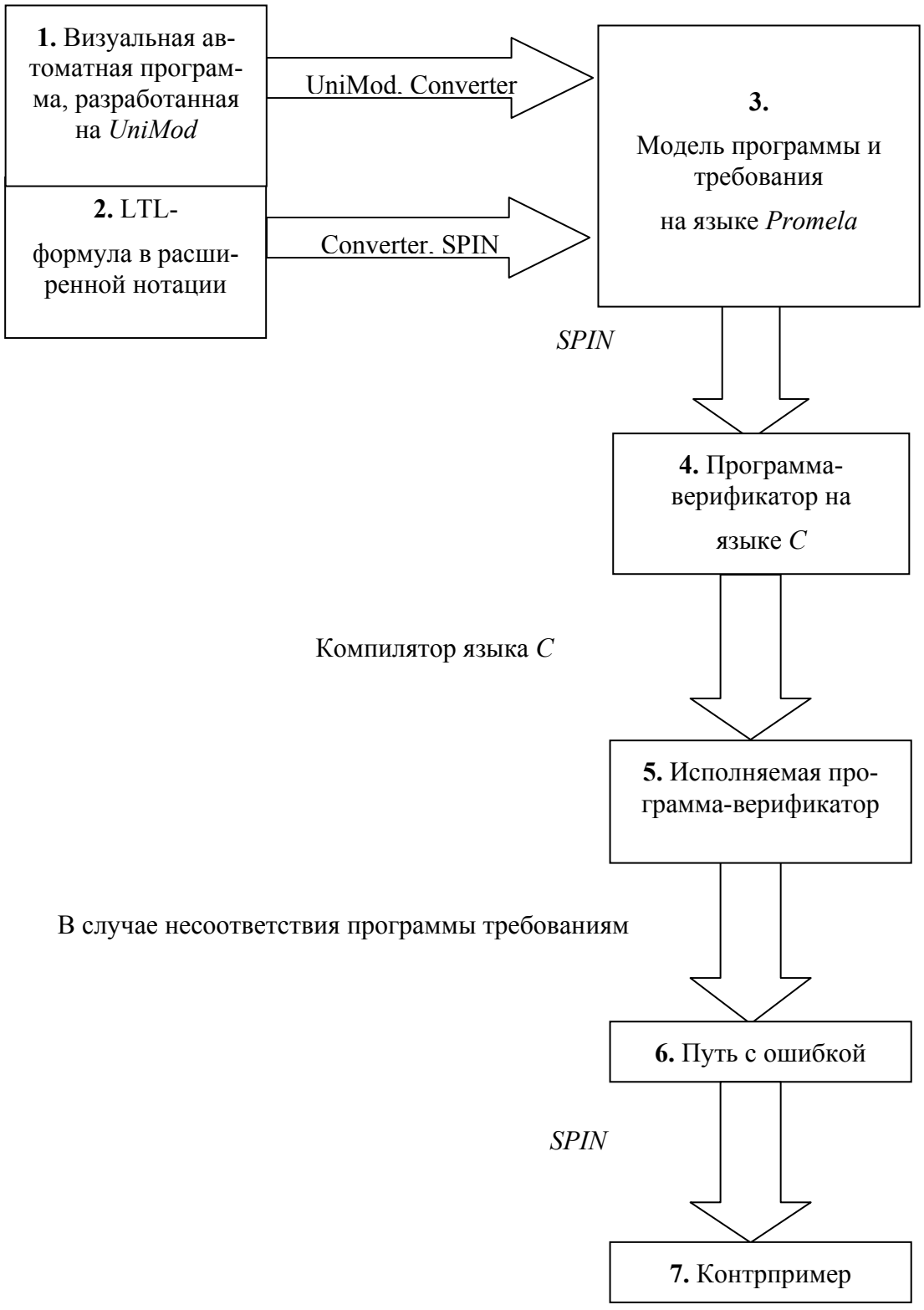


Рис. 3. Общая схема верификации

3. *Converter* принимает на вход визуальную автоматную программу, разработанную при помощи инструментального средства *UniMod* и сохраненную в формате *XML*, и требования к ней, записанные на языке *LTL* в расширенной нотации (рис. 3, переход 1 – 2). *Важно: верификатору на вход подаются не требования, а их отрицание.*
4. При помощи *UniMod* из *XML*-файла получается автоматная модель на языке *Java* (рис. 4) – переход 1-3 на рис. 3.

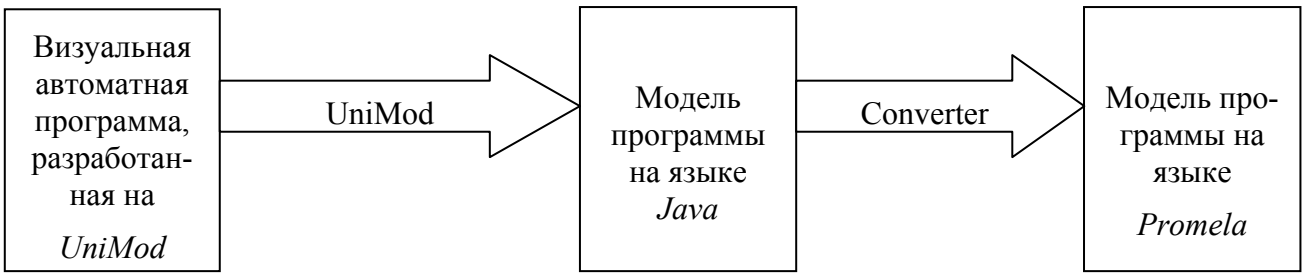


Рис. 4. Взаимодействие с *UniMod*

5. *Converter* транслирует автоматную модель с языка Java на язык *Promela* (рис. 4) – переход 1-3 на рис. 3.
6. *Converter* преобразует *LTL*-формулу в нотацию верификатора *SPIN* (рис. 5). Это преобразование выполняется следующим образом (это также переход 2-3 на рис. 3).

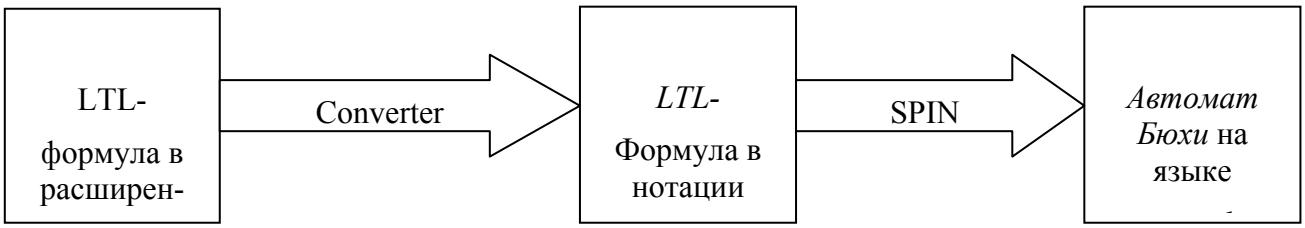


Рис. 5. Преобразование *LTL*-формул

При этом должны выполняться следующие условия.

- Все элементарные высказывания должны быть записаны в фигурных скобках.
- Все элементарные высказывания должны удовлетворять синтаксису языка *Promela*.
- Каждому элементарному высказыванию присваивается идентификатор pk, где *k* — порядковый номер элементарного высказывания в формуле.
- Для каждого элементарного высказывания *Converter* генерирует макрос на языке *Promela*, закрепляющий идентификатор за элементарным высказыванием. Если *Converter* распознает формулу как неправильную, то он выводит в консоль сообщение «Wrong formula».
- Идентификаторы событий exx, где *xx* – номер события, преобразовываются в число *xx*.
- Пример. Формула (элементарное высказывание)

```
{lastEvent == e13}
```

преобразовывается в строку на *Promela*:

```
#define pk (lastEvent == 13),
```

где *k* — номер элементарного высказывания.

7. *Converter* запускает верификатор *SPIN* в режиме генерации конструкции *never claim* (с помощью команды `spin -f <формула>`). Верификатор по *LTL*-формуле генерирует конструкцию n*ever claim*, представляющую собой *автомат Бюхи*, записанный на языке *Promela* (рис. 5). Это также переход 2–3 на рис. 3.
8. После того, как на языке *Promela* описаны модель и требование к ней, *Converter* запускает *SPIN* на верификацию (командой `spin -a <Модель>`). Верификатор *SPIN* генерирует файл `pan.c`, представляющий собой программу-верификатор на языке *C* (переход 3–4 на рис. 3).
9. После компиляции файла `pan.c` получается программа-верификатор для данной конкретной модели с заданным требованием. Программа *pan* выполняет верификацию построенной модели. При обнаружении ошибок программа *pan* выдает *trail*-файл, в котором описана трасса ошибки в формате, понятном верификатору *SPIN* (переходы 4–6 на рис. 3). Кроме того, программа-верификатор *pan* выводит отчет, в котором содержится краткая информация об ошибках (переход 5–6 на рис. 3), использованной памяти, версия *SPIN* и т.д.
10. По команде `spin –t –p <Модель>` верификатор выводит отчет, содержащий контрпример. *Converter* собирает воедино отчет, созданный *SPIN*, и отчет, созданный программой *pan* (переход 6–7 на рис. 3).

Обратное преобразование контрпримера

Верификатор *SPIN* выдает контрпример в виде пути в модели, описанной на языке *Promela*. Благодаря системе пометок контрпример автоматически преобразовывается в удобный для восприятия текст. Отчет, составленный инструментом *Converter*, содержит в себе оба варианта. Строка отчета

```
Calling automaton Ai
```

обозначает вход в автомат *Ai*.

Строка отчета

```
Going to state Ai s : Имя состояния
```

обозначает переход автомата *Ai* в состояние *s*.

Строка отчета

```
Event = exx
```

обозначает, что на данном переходе есть событие *exx*.

Строка отчета

```
State Ai s : Имя состояния
```

обозначает, что автомат *Ai* попал в состояние *s*.

Строка отчета

```
lastEvent = exx,
```

где *exx* – некоторое событие обозначает, что был совершен переход по событию *exx*.

Таким образом строится путь в автоматной модели из файла отчета, выданного инструментом *Converter*.

Пример использования автоматической верификации

Постановка задачи

Проведем верификацию программной модели банкомата [20], разработанной при помощи инструментального средства *UniMod*. На рис. 6, 7 показаны номера, которые *Converter* присвоил состояниям системы автоматов.

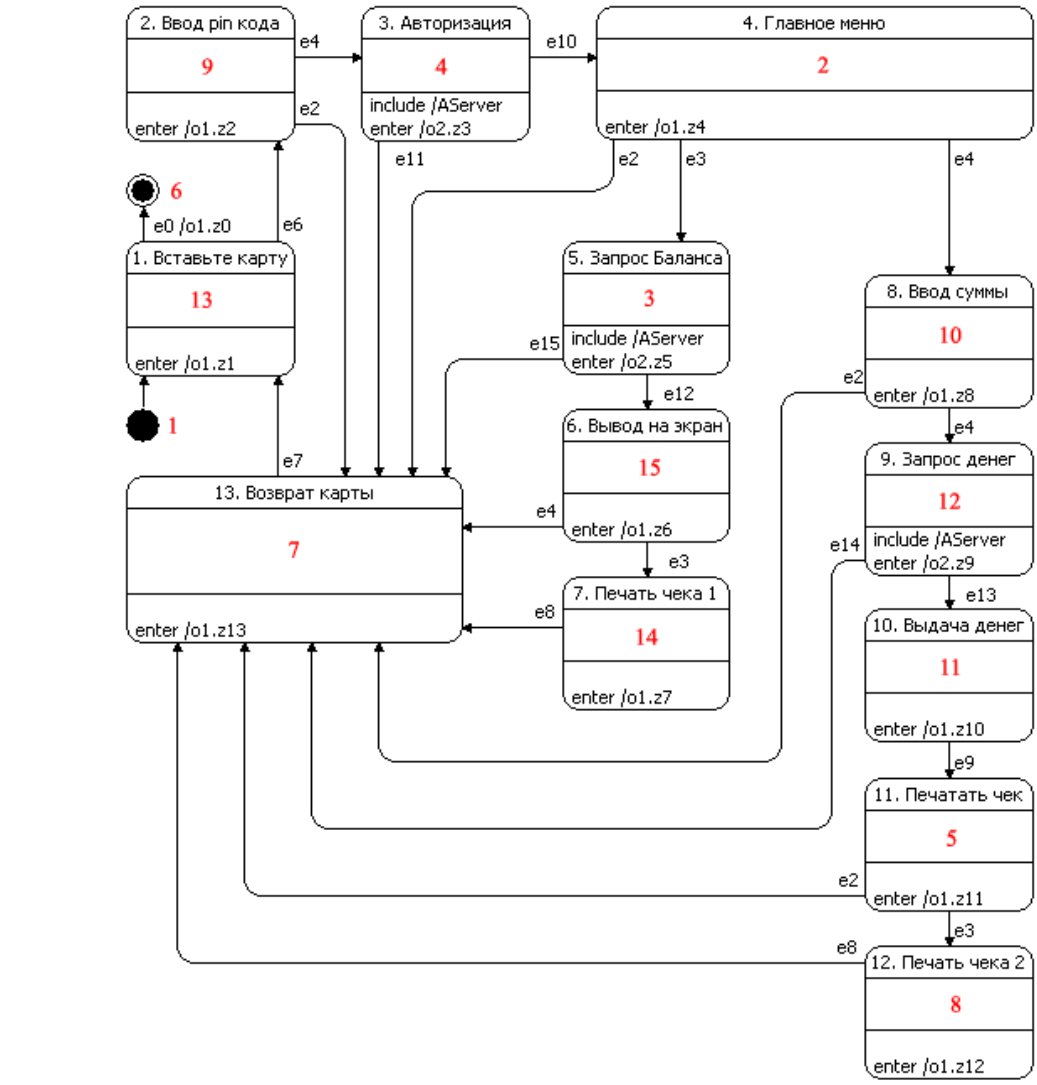


Рис. 6. Автоматная реализация банкомата. Автомат *AClient*

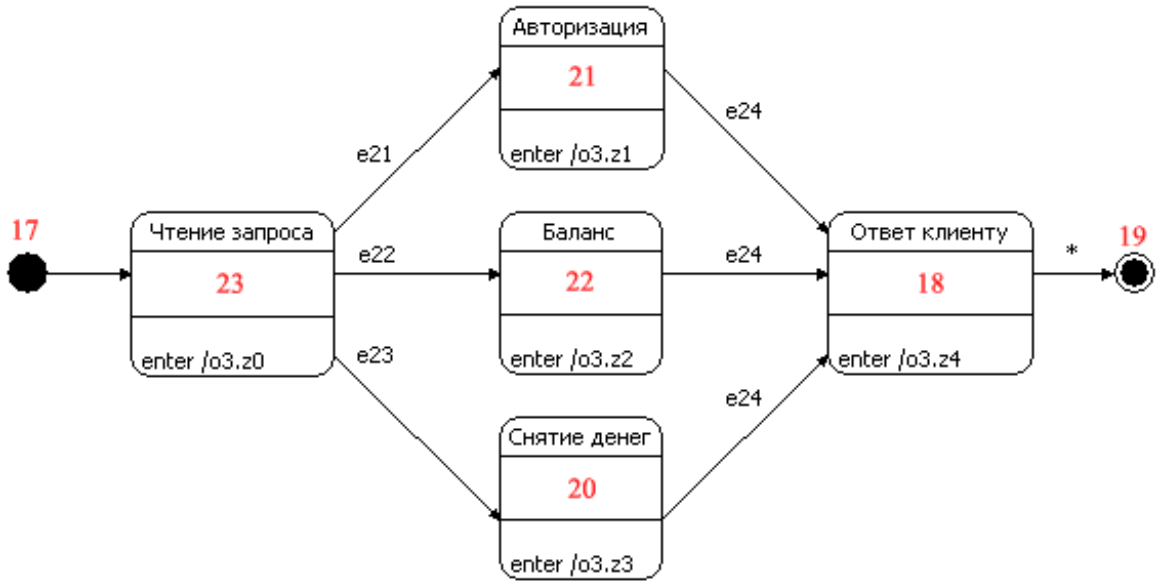


Рис. 7. Автоматная реализация банкомата. Автомат *AServer*

Верификация

Проверим, выдает ли банкомат деньги. Для этого проверим свойство «Банкомат не дает денег». Напомним, что на вход верификатору *SPIN* требуется подавать отрицания проверяемых свойств. Запишем отрицание этого свойства на русском языке: «Когда-нибудь банкомат выдаст деньги». Для построения формулы на языке *LTL*, соответствующей данному свойству, введем атомарное высказывание: `stateAClient == 11`. Оно обозначает, что автомат *AClient* попал в состояние №11 «10. Выдача денег» и выдал деньги. Перефразируем данное свойство (точнее, его отрицание) через введенное элементарное высказывание: «Автомат *AClient* попадет в состояние «10. Выдача денег»». Следовательно, *LTL*-формула для этого свойства будет выглядеть следующим образом: `<>({stateAClient == 11})`.

Ожидаемый результат: в отчете о проведенной верификации должно быть сообщение об ошибке и выведенный контрпример. В этом контрпримере должен быть путь по состояниям автомата *AClient* до состояния «*10. Выдача денег*». Проведем верификацию. Инструментальное средство *Converter* выдаст следующий отчет:

```
Converter v. 0.50
warning: for p.o. reduction to be valid the never claim must be
         stutter-invariant
(never claims generated from LTL formulae are stutter-
         invariant)
pan: claim violated! (at depth 173)
pan: wrote models/aclient.ltl.trail
(Spin Version 4.2.8 -- 6 January 2007)
Warning: Search not completed
  + Partial Order Reduction

Full statespace search for:
  never claim   +
  assertion violations  + (if within scope of claim)
  acceptance cycles  - (not selected)
  invalid end states - (disabled by never claim)

State-vector 28 byte, depth reached 187, errors: 1
 198 states, stored
 10 states, matched
 208 transitions (= stored+matched)
 0 atomic steps
hash conflicts: 0 (resolved)

2.622  memory usage (Mbyte)

Starting :init: with pid 0
Starting :never: with pid 1
Never claim moves to line 267 [(1)]
Starting Model with pid 2
 2:  proc 0 (:init:) line 261 "models/aclient.ltl" (state 1)
            [(run Model())]
 4:  proc 1 (Model) line 23 "models/aclient.ltl" (state 1)
            [stateAClient = 1]
 6:  proc 1 (Model) line 25 "models/aclient.ltl" (state 2)
            [((stateAClient==1))]
 State AClient 1 : s1
 8:  proc 1 (Model) line 26 "models/aclient.ltl" (state 3)
            [printf('State AClient 1 : s1\\n')]
 10: proc 1 (Model) line 28 "models/aclient.ltl" (state 4)
            [stateAClient = 13]
 Going to state AClient 13 : 1. Вставьте карту
```

```
12: proc 1 (Model) line 29 "models/aclient.ltl" (state 5)
          [printf('Going to state AClient 13 : 1. Вставьте
        карту\\n')]
14: proc 1 (Model) line 152 "models/aclient.ltl" (state 306)
          [((stateAClient==13))]
State AClient 13 : 1. Вставьте карту
16: proc 1 (Model) line 153 "models/aclient.ltl" (state 307)
          [printf('State AClient 13 : 1. Вставьте карту\\n')]
18: proc 1 (Model) line 159 "models/aclient.ltl" (state 312)
          [stateAClient = 9]
Going to state AClient 9 : 2. Ввод pin кода
20: proc 1 (Model) line 160 "models/aclient.ltl" (state 313)
          [printf('Going to state AClient 9 : 2. Ввод pin
        кода\\n')]
22: proc 1 (Model) line 161 "models/aclient.ltl" (state 314)
          [lastEvent = 6]
Event = e6
24: proc 1 (Model) line 162 "models/aclient.ltl" (state 315)
          [printf('Event = e6\\n')]
26: proc 1 (Model) line 106 "models/aclient.ltl" (state 201)
          [((stateAClient==9))]
State AClient 9 : 2. Ввод pin кода
28: proc 1 (Model) line 107 "models/aclient.ltl" (state 202)
          [printf('State AClient 9 : 2. Ввод pin кода\\n')]
30: proc 1 (Model) line 109 "models/aclient.ltl" (state 203)
          [stateAClient = 4]
Going to state AClient 4 : 3. Авторизация
32: proc 1 (Model) line 110 "models/aclient.ltl" (state 204)
          [printf('Going to state AClient 4 : 3.
        Авторизация\\n')]
34: proc 1 (Model) line 111 "models/aclient.ltl" (state 205)
          [lastEvent = 4]
Event = e4
36: proc 1 (Model) line 112 "models/aclient.ltl" (state 206)
          [printf('Event = e4\\n')]
38: proc 1 (Model) line 61 "models/aclient.ltl" (state 97)
          [((stateAClient==4))]
State AClient 4 : 3. Авторизация
40: proc 1 (Model) line 62 "models/aclient.ltl" (state 98)
          [printf('State AClient 4 : 3. Авторизация\\n')]
Calling automaton AServer
40: proc 1 (Model) line 63 "models/aclient.ltl" (state 99)
          [printf('Calling automaton AServer\\n')]
42: proc 1 (Model) line 197 "models/aclient.ltl" (state 100)
          [stateAServer = 17]
44: proc 1 (Model) line 199 "models/aclient.ltl" (state 101)
          [((stateAServer==17))]
State AServer 17 : s1
46: proc 1 (Model) line 200 "models/aclient.ltl" (state 102)
          [printf('State AServer 17 : s1\\n')]
48: proc 1 (Model) line 202 "models/aclient.ltl" (state 103)
          [stateAServer = 23]
Going to state AServer 23 : Чтение запроса
50: proc 1 (Model) line 203 "models/aclient.ltl" (state 104)
          [printf('Going to state AServer 23 : Чтение
        запроса\\n')]
52: proc 1 (Model) line 238 "models/aclient.ltl" (state 140)
          [((stateAServer==23))]
State AServer 23 : Чтение запроса
54: proc 1 (Model) line 239 "models/aclient.ltl" (state 141)
          [printf('State AServer 23 : Чтение запроса\\n')]
```

```
 56: proc 1 (Model) line 241 "models/aclient.ltl" (state 142)
           [stateAServer = 20]
 Going to state AServer 20 : Снятие денег
 58: proc 1 (Model) line 242 "models/aclient.ltl" (state 143)
           [printf('Going to state AServer 20 : Снятие
         денег\\n')]
 60: proc 1 (Model) line 243 "models/aclient.ltl" (state 144)
           [lastEvent = 23]
 Event = e23
 62: proc 1 (Model) line 244 "models/aclient.ltl" (state 145)
           [printf('Event = e23\\n')]
 64: proc 1 (Model) line 214 "models/aclient.ltl" (state 116)
           [((stateAServer==20))]
 State AServer 20 : Снятие денег
 66: proc 1 (Model) line 215 "models/aclient.ltl" (state 117)
           [printf('State AServer 20 : Снятие денег\\n')]
 68: proc 1 (Model) line 217 "models/aclient.ltl" (state 118)
           [stateAServer = 18]
 Going to state AServer 18 : Ответ клиенту
 70: proc 1 (Model) line 218 "models/aclient.ltl" (state 119)
           [printf('Going to state AServer 18 : Ответ
         клиенту\\n')]
 72: proc 1 (Model) line 219 "models/aclient.ltl" (state 120)
           [lastEvent = 24]
 Event = e24
 74: proc 1 (Model) line 220 "models/aclient.ltl" (state 121)
           [printf('Event = e24\\n')]
 76: proc 1 (Model) line 205 "models/aclient.ltl" (state 107)
           [((stateAServer==18))]
 State AServer 18 : Ответ клиенту
 78: proc 1 (Model) line 206 "models/aclient.ltl" (state 108)
           [printf('State AServer 18 : Ответ клиенту\\n')]
 80: proc 1 (Model) line 208 "models/aclient.ltl" (state 109)
           [stateAServer = 19]
 Going to state AServer 19 : s2
 82: proc 1 (Model) line 209 "models/aclient.ltl" (state 110)
           [printf('Going to state AServer 19 : s2\\n')]
 84: proc 1 (Model) line 211 "models/aclient.ltl" (state 113)
           [((stateAServer==19))]
 State AServer 19 : s2
 86: proc 1 (Model) line 212 "models/aclient.ltl" (state 114)
           [printf('State AServer 19 : s2\\n')]
 88: proc 1 (Model) line 66 "models/aclient.ltl" (state 160)
           [stateAClient = 2]
 Going to state AClient 2 : 4. Главное меню
 90: proc 1 (Model) line 67 "models/aclient.ltl" (state 161)
           [printf('Going to state AClient 2 : 4. Главное
         меню\\n')]
 92: proc 1 (Model) line 68 "models/aclient.ltl" (state 162)
           [lastEvent = 10]
 Event = e10
 94: proc 1 (Model) line 69 "models/aclient.ltl" (state 163)
           [printf('Event = e10\\n')]
 96: proc 1 (Model) line 31 "models/aclient.ltl" (state 8)
           [((stateAClient==2))]
 State AClient 2 : 4. Главное меню
 98: proc 1 (Model) line 32 "models/aclient.ltl" (state 9)
           [printf('State AClient 2 : 4. Главное меню\\n')]
100: proc 1 (Model) line 42 "models/aclient.ltl" (state 18)
           [stateAClient = 10]
 Going to state AClient 10 : 8. Ввод суммы
```

```
102: proc 1 (Model) line 43 "models/aclient.ltl" (state 19)
           [printf('Going to state AClient 10 : 8. Ввод
         суммы\\n')]
104: proc 1 (Model) line 44 "models/aclient.ltl" (state 20)
           [lastEvent = 4]
 Event = e4
106: proc 1 (Model) line 45 "models/aclient.ltl" (state 21)
           [printf('Event = e4\\n')]
108: proc 1 (Model) line 118 "models/aclient.ltl" (state 213)
           [((stateAClient==10))]
 State AClient 10 : 8. Ввод суммы
110: proc 1 (Model) line 119 "models/aclient.ltl" (state 214)
           [printf('State AClient 10 : 8. Ввод суммы\\n')]
112: proc 1 (Model) line 121 "models/aclient.ltl" (state 215)
           [stateAClient = 12]
 Going to state AClient 12 : 9. Запрос денег
114: proc 1 (Model) line 122 "models/aclient.ltl" (state 216)
           [printf('Going to state AClient 12 : 9. Запрос
         денег\\n')]
116: proc 1 (Model) line 123 "models/aclient.ltl" (state 217)
           [lastEvent = 4]
 Event = e4
118: proc 1 (Model) line 124 "models/aclient.ltl" (state 218)
           [printf('Event = e4\\n')]
120: proc 1 (Model) line 138 "models/aclient.ltl" (state 233)
           [((stateAClient==12))]
 State AClient 12 : 9. Запрос денег
122: proc 1 (Model) line 139 "models/aclient.ltl" (state 234)
           [printf('State AClient 12 : 9. Запрос денег\\n')]
 Calling automaton AServer
122: proc 1 (Model) line 140 "models/aclient.ltl" (state 235)
           [printf('Calling automaton AServer\\n')]
124: proc 1 (Model) line 197 "models/aclient.ltl" (state 236)
           [stateAServer = 17]
126: proc 1 (Model) line 199 "models/aclient.ltl" (state 237)
           [((stateAServer==17))]
 State AServer 17 : s1
128: proc 1 (Model) line 200 "models/aclient.ltl" (state 238)
           [printf('State AServer 17 : s1\\n')]
130: proc 1 (Model) line 202 "models/aclient.ltl" (state 239)
           [stateAServer = 23]
 Going to state AServer 23 : Чтение запроса
132: proc 1 (Model) line 203 "models/aclient.ltl" (state 240)
           [printf('Going to state AServer 23 : Чтение
         запроса\\n')]
134: proc 1 (Model) line 238 "models/aclient.ltl" (state 276)
           [((stateAServer==23))]
 State AServer 23 : Чтение запроса
136: proc 1 (Model) line 239 "models/aclient.ltl" (state 277)
           [printf('State AServer 23 : Чтение запроса\\n')]
138: proc 1 (Model) line 241 "models/aclient.ltl" (state 278)
           [stateAServer = 20]
 Going to state AServer 20 : Снятие денег
140: proc 1 (Model) line 242 "models/aclient.ltl" (state 279)
           [printf('Going to state AServer 20 : Снятие
         денег\\n')]
142: proc 1 (Model) line 243 "models/aclient.ltl" (state 280)
           [lastEvent = 23]
 Event = e23
144: proc 1 (Model) line 244 "models/aclient.ltl" (state 281)
           [printf('Event = e23\\n')]
```

```
146: proc 1 (Model) line 214 "models/aclient.ltl" (state 252)
            [((stateAServer==20))]
 State AServer 20 : Снятие денег
148: proc 1 (Model) line 215 "models/aclient.ltl" (state 253)
            [printf('State AServer 20 : Снятие денег\\n')]
150: proc 1 (Model) line 217 "models/aclient.ltl" (state 254)
            [stateAServer = 18]
 Going to state AServer 18 : Ответ клиенту
152: proc 1 (Model) line 218 "models/aclient.ltl" (state 255)
            [printf('Going to state AServer 18 : Ответ
          клиенту\\n')]
154: proc 1 (Model) line 219 "models/aclient.ltl" (state 256)
            [lastEvent = 24]
 Event = e24
156: proc 1 (Model) line 220 "models/aclient.ltl" (state 257)
            [printf('Event = e24\\n')]
158: proc 1 (Model) line 205 "models/aclient.ltl" (state 243)
            [((stateAServer==18))]
 State AServer 18 : Ответ клиенту
160: proc 1 (Model) line 206 "models/aclient.ltl" (state 244)
            [printf('State AServer 18 : Ответ клиенту\\n')]
162: proc 1 (Model) line 208 "models/aclient.ltl" (state 245)
            [stateAServer = 19]
 Going to state AServer 19 : s2
164: proc 1 (Model) line 209 "models/aclient.ltl" (state 246)
            [printf('Going to state AServer 19 : s2\\n')]
166: proc 1 (Model) line 211 "models/aclient.ltl" (state 249)
            [((stateAServer==19))]
 State AServer 19 : s2
168: proc 1 (Model) line 212 "models/aclient.ltl" (state 250)
            [printf('State AServer 19 : s2\\n')]
170: proc 1 (Model) line 143 "models/aclient.ltl" (state 296)
            [stateAClient = 11]
Never claim moves to line 266 [((stateAClient==11))]
 Going to state AClient 11 : 10. Выдача денег
172: proc 1 (Model) line 144 "models/aclient.ltl" (state 297)
            [printf('Going to state AClient 11 : 10. Выдача
          денег\\n')]
Never claim moves to line 270 [(1)]
spin: trail ends after 174 steps
#processes: 2
     lastEvent = 24
     stateAClient = 11
     stateAServer = 19
174: proc 1 (Model) line 145 "models/aclient.ltl" (state 298)
174: proc 0 (:init:) line 262 "models/aclient.ltl" (state 2)
          <valid end state>
174: proc - (:never:) line 271 "models/aclient.ltl" (state 8)
          <valid end state>
2 processes created
```

Строка отчета

```
State-vector 28 byte, depth reached 133, errors: 1
```

говорит о том, что была найдена ошибка. Выпишем контрпример в явном виде:

Автомат AClient перешел в состояние *s1*.

Автомат *AClient* перешел в состояние *1. Вставьте карту*.

Произошло событие *e6*.

Автомат *AClient* перешел в состояние *2. Ввод pin кода*.

Произошло событие *e4*.

Автомат *AClient* перешел в состояние *3. Авторизация*.

Автомат *AServer* перешел в состояние *s1*.
Автомат *AServer* перешел в состояние *Чтение запроса*.
Произошло событие *e23*.
Автомат *AServer* перешел в состояние *Снятие денег*.
Произошло событие *e24*.
Автомат *AServer* перешел в состояние *Ответ клиенту*.
Автомат *AServer* перешел в состояние *s2*.
Произошло событие *e10*.
Автомат *AClient* перешел в состояние *4. Главное меню*.
Произошло событие *e4*.
Автомат *AClient* перешел в состояние *8. Ввод суммы*.
Произошло событие *e4*.
Автомат *AClient* перешел в состояние *9. Запрос денег*.
Автомат *AServer* перешел в состояние *s1*.
Автомат *AServer* перешел в состояние *Чтение запроса*.
Произошло событие *e23*.
Автомат *AServer* перешел в состояние *Снятие денег*.
Произошло событие *e24*.
Автомат *AServer* перешел в состояние *Ответ клиенту*.
Автомат *AServer* перешел в состояние *s2*.
Произошло событие *e13*.
Автомат *AClient* перешел в состояние *10. Выдача денег*.

Как и ожидалось, контрпример содержит путь по состояниям автомата *AClient* до состояния «*10. Выдача денег*».

Заключение

В предложенном методе верификации автоматных программ используется верификатор *SPIN* – один из наиболее мощных и известных верификаторов. Его используют *NASA* [21] и многие другие организации, где требуется повышенная надежность.

Автоматные программы удобны для проектирования и наглядны. Кроме того, они позволяют автоматически построить модель Крипке и произвести верификацию.

Настоящая работа предлагает метод автоматической верификации автоматных программ, написанных в среде *UniMod* с помощью верификатора *SPIN*.

Литература

1. Новиков Ф.А. Визуальное конструирование программ. – Режим доступа: http://is.ifmo.ru/works/visualcons/
2. Шалыто А.А. Switch-технология. Алгоритмизация и программирование задач логического управления. – СПб: Наука, 1998. – Режим доступа: http://is.ifmo.ru/books/switch/1/
3. UniMod home page. – Режим доступа: http://UniMod.sourceforge.net/
4. Лифшиц Ю. Верификация программ и темпоральные логики. Лекция № 3 курса «Современные задачи теоретической информатики». – Режим доступа: http://download.yandex.ru/class/lifshits/lecture-note03.pdf
5. Лифшиц Ю. Символьная верификация программ. Лекция № 4 курса «Современные задачи теоретической информатики». – Режим доступа: http://download.yandex.ru/class/lifshits/lecture-note04.pdf
6. Кларк Э.М., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. – М.: МЦНМО, 2002.

7. Holzman G. J. Design And Validation Of Computer Protocols. – Prentice Hall, 1991.
8. Holzman G. J. The model checker SPIN //IEEE Trans. on Software Engineering. –1997. – №23. – P. 279–295.
9. SPIN home page. – Режим доступа: http://SPINroot.com
10. Вельдер С.Э., Шалыто А.А. О верификации простых автоматных программ на основе метода Model checking // Информационно-управляющие системы. – 2007. – № 3. – С. 27–38. – Режим доступа: http://is.ifmo.ru/download/27-38.pdf
11. Васильева К.А., Кузьмин Е.В. Верификация автоматных программ с использованием LTL // Моделирование и анализ информационных систем. – 2007. – № 1. – С.3–14. – Режим доступа: http://is.ifmo.ru/verification/_LTL_for_Spin.pdf
12. Виноградов Р.А., Кузьмин Е.В., Соколов В.А. Верификация автоматных программ средствами CPN/Tools / Доклады II-й научно-методической конференции преподавателей матем. ф-та и ф-та ИВТ Ярославского гос. ун-та им. П.Г. Демидова «Преподавание математики и компьютерных наук в классическом университете». – Ярославль: ЯрГУ. 2007. – С. 91–101. – Режим доступа: http://is.ifmo.ru/verification/_ver_prog.pdf
13. Linear temporal logic. – Режим доступа: http://en.wikipedia.org/wiki/Linear_temporal_logic
14. Белешко Д.С. Верификация автоматных моделей программ. Бакалаврская работа. – СПбГУ ИТМО, 2006.
15. Dijkstra E. W. Guarded commands, non-determinacy and formal derivation of programs // CACM. – 1975. – №8.
16. Büchi automaton. – Режим доступа: http://en.wikipedia.org/wiki/Büchi_automaton
17. Кюзара В.Е. Реализация системы проверки моделей раскрашенных сетей Петри с использованием разверток. – Режим доступа: www.iis.nsk.su/preprints/pdf/094.pdf
18. Vardy M., Wolper P. An automata-theoretic approach to automatic program verification // Proc. 1st IEEE Symp. On Logic in Computer Science. – 1986.
19. Яковлев А.В., Лукин М.А., Шалыто А.А. Реализация классической игры «Ним» на основе автоматного подхода. – СПбГУ ИТМО. 2006. – Режим доступа: http://is.ifmo.ru/UniMod-projects/knim/
20. Козлов В.А., Комалева О.А. Моделирование работы банкомата. – СПбГУ ИТМО. 2006. – Режим доступа: http://is.ifmo.ru/UniMod-projects/bankomat/
21. Риган П., Хемилтон С. NASA: миссия надежна // Открытые системы. – 2004. – № 3. – Режим доступа: http://www.osp.ru/os/2004/03/184060/

УДК 004.4’242

ВЕРИФИКАЦИЯ АВТОМАТНЫХ ПРОГРАММ ПРИ ПОМОЩИ ВЕРИФИКАТОРА UNIMOD.VERIFIER

В.С. Гуров, Б.Р. Яминов

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

Описан разработанный авторами верификатор автоматных программ, созданных при помощи инструментального средства для поддержки автоматного программирования *UniMod.* Верификатор работает за счет интеграции инструментального средства *UniMod* и верификатора программ *Bogor*. При использовании разработанного верификатора отсутствует необходимость преобразовывать автоматную программу во входной язык верификатора. Требования к программе записываются на языке темпоральной логики *LTL*.

Ключевые слова: верификация программ, верификация на моделях, автоматное программирования

Введение

В настоящей работе описывается верификация автоматных программ при помощи верификатора *UniMod.verifier*, который разработан на основе интеграции инструментального средства *UniMod* [1, 2], предназначенного для создания и запуска автоматных

программ, и верификатора программ *Bogor* [11, 4]. Сначала описывается метод верификации, который используется в верификаторе *UniMod.verifier,* а далее приводится пример верификации автоматной программы.

Метод верификации

В настоящем разделе описывается предлагаемый метод верификации.

Постановка задачи

Предлагаемый метод предназначен для решения задачи верификации автоматных программ. Автоматная программа [5] может рассматриваться как реактивная система, состоящая из трех компонент:

- источники событий – создают события в системе;
- объекты управления – объекты, выполняющие действия;
- управляющая система – модуль, принимающий события от источников событий, и вызывающий действия объектов управления. В автоматной программе данный модуль реализован, как система иерархически связанных конечных автоматов.

Верификация автоматных программ состоит в проверке того, что управляющая система работает корректно (проверка корректности реализации источников событий и объектов управления должна проводиться отдельно). Корректность автоматной программы определяется выполнением темпоральных утверждений вида «если произошло событие `e1`, то когда-нибудь будет вызвано действие `z1`» или «если `x1` всегда неверно (`x1` всегда `false`), то автомат никогда не попадет в состояние `s2`». Утверждения, которые требуется проверить, называют *требованиями*. Если система автоматов удовлетворяет требованиям (сформулированные утверждения выполняются для каждой истории работы системы), считается, что верификация завершена успешно. Если же утверждения не выполняются и существует последовательность действий, которая приводит систему автоматов к нарушению сформулированных требований, то выводится этот набор действий. Его называют «*контрпример*».

Для решения задачи верификации автоматных программ было выполнено несколько работ, в том числе работа [7], в которой был предложен метод верификации автоматных программ с помощью верификатора *SPIN*. В настоящей работе предлагается оригинальный метод верификации автоматных программ с помощью *эмуляции* (или имитации) ее работы. Этот метод позволяет значительно снизить сложность преобразований исходной автоматной программы, необходимых для верификации, по сравнению с методом, описанным в работе [7].

Верификация программ на основе алгоритма двойного поиска в глубину

Верификация программ может выполняться с использованием алгоритма двойного поиска в глубину [8]. Этот алгоритм используется во многих верификаторах, в том числе в верификаторах *SPIN* и *Bogor*. Верификация с применением алгоритма двойного поиска в глубину выполняется следующим образом.

Сначала для верифицируемой программы строится модель *Крипке* [8] – граф элементарных (атомарных) состояний, в которых может находиться программа, и переходов между ними. Модель *Крипке* является подробной схемой работы программы, в которой в каждом состоянии четко определены элементарные свойства программы.

Требования к программе формулируются в виде формул темпоральной логики. Такие формулы позволяют специфицировать работу программы *во времени*. Темпоральные формулы состоят из *предикатов* – элементарных утверждений о программе,

логических операторов («не», «и», «или») и темпоральных операторов – операторов, описывающих выполнение утверждений во времени.

В настоящей работе в глубину используется темпоральная логика *LTL* (*Linear Temporal Logic*) [8]. В этой логике применяются следующие темпоральные операторы:

- *X* (neXt) – «*X p*» верно тогда, когда в следующий момент времени в программе будет выполняться предикат *p*;
- *G* (Globally) – «*G p*» верно, если во время работы программы всегда выполняется *p*;
- *F* (Future) – «*F p*» верно, если в будущем наступит момент, когда выполнится *p*;
- *U* (Until) – «*p U q*» верно, если в программе в каждый момент времени выполняется *p* до тех пор, пока не выполнится *q*. При этом *q* обязательно должно когда-либо выполниться;
- *R* (Release) – «*q R p*» верно, если *p* выполняется до тех пор, пока не станет выполняться *q* (включая момент, когда выполнится *q*), или всегда, если *q* не выполнится никогда.

Формула *LTL*, описывающая требования к программе, преобразуется в автомат *Бюхи* [8] – конечный автомат над бесконечными словами. Переходы автомата *Бюхи* помечены предикатами из исходной формулы *LTL*. Поскольку задача верификатора – найти контрпример, если он существует, автомат *Бюхи* строится для *отрицания* формулы *LTL*. Автомат *Бюхи* допускает любые последовательности значений предикатов, которые не удовлетворяют требованиям.

Далее модель *Крипке* преобразуется в автомат *Бюхи* для того, чтобы построить пересечение его с автоматом *Бюхи*, построенным по отрицанию формулы *LTL*. Оно также является автоматом *Бюхи*. Для построенного пересечения автоматов запускается алгоритм *двойного поиска в глубину* [8], который находит допускающую последовательность предикатов, если она существует. Если эта последовательность существует, то:

- она допускается автоматом *Бюхи*, построенным по модели *Крипке*. Следовательно, эта последовательность является историей работы исходной программы;
- последовательность допускается автоматом *Бюхи*, построенным из отрицания формулы *LTL*. Следовательно, эта последовательность является историей, нарушающей проверяемые требования.

Таким образом, найденная последовательность предикатов является контрпримером.

В верификаторе *Bogor* явно не строятся модель *Крипке*, автомат *Бюхи*, который строится по модели *Крипке,* и его пересечение с автоматом *Бюхи*. Верификатор получает на вход программу, написанную на входном языке верификатора. Это позволяет верификатору выделять в программе элементарные действия, поскольку в семантике языка верификатора определено, какие действия совершаются атомарно, как откатывать назад эти действия, возвращая систему в предыдущее состояние, как вычислять состояния объектов (переменных) и какие операторы порождают недетерминированность. Таким образом, верификатор может управлять исполнением программы, например, совершать элементарные шаги, «отменять» элементарные шаги (откатываться назад), вычислять глобальное состояние системы. Все это необходимо для работы алгоритма двойного поиска в глубину.

Такие возможности позволяют верификатору избежать явного построения пересечения автомата модели *Крипке* с автоматом *Бюхи*. Для того чтобы так работать, верификатору *Bogor* необходимо выполнять следующие действия.

- Вычислять глобальное состояние программы. Оно должно однозначно определять поведение программы.

- Совершать элементарный шаг работы программы. Такой шаг является переходом программы из одного глобального состояния в другое без посещения иных глобальных состояний.
- Откатывать назад элементарный шаг работы программы. При этом программа возвращается в предыдущее состояние.
- В каждом состоянии определять возможные элементарные шаги.
- Определять значения набора предикатов программы, используемых в требованиях.

Верификация автоматных программ с использованием алгоритма двойного поиска в глубину

В методе эмуляции указанные в предыдущем разделе действия осуществляются следующим образом.

- Глобальное состояние программы складывается из набора текущих состояний каждого автомата.

  Верификатор *UniMod.verifier* использует инструментальное средство *UniMod* для работы с автоматной программой. Это средство хранит текущие состояния каждого автомата, выполняет по команде разработанного верификатора обработку события и т.д. При обработке события информация о сделанных переходах в автоматах, о выполненных действиях и о новом глобальном состоянии передается из *UniMod* в *UniMod.verifier*. Теоретически для построения модели *Крипке* для системы автоматов необходимо было бы построить один автомат как пересечение всех автоматов системы. Однако это делается неявно в самом инструментальном средстве *UniMod*. При этом *UniMod.verifier* сразу получает информацию о глобальном состоянии системы автоматов, как набор состояний каждого автомата.
- Элементарный шаг работы программы – это событий обработка системой автоматов. В результате обработки может смениться набор состояний автоматов.
- Поведение автоматной программы (без учета объектов управления) определяется набором состояний автоматов. Таким образом, для отката назад достаточно вернуть автоматы в те состояния, в которых они находились до выполнения шага вперед.
- Для автоматной программы строго определена схема работы системы автоматов, последовательность передачи управления между автоматами. Кроме того, система работает в одном потоке. Поэтому недетерминированность в работе системы возникает лишь в результате разных последовательностей входных событий, а также в результате различных возможных значений переменных, запрашиваемых у объектов управления. Соответственно, в методе эмуляции возможные элементарные шаги определяются следующим образом.

Перед совершением очередного элементарного шага определяется набор событий, которые система автоматов может обработать в текущем глобальном состоянии, и каждое из этих событий затем используется для создания одной из историй работы программы. Аналогично, при необходимости вычислить условие перехода, в котором участвуют переменные объектов управления, такое условие принимается равным `True` или `False`. Оба варианта используются для создания двух различных историй работы программы.

Для вычисления предикатов при совершении элементарного шага сохраняется следующая информация:

- состояния автоматов до выполнения шага;
- обрабатываемое событие;
- список вычисленных значений условий на переходах;
- список вызванных действий у объектов управления.

Эта информация позволяет вычислять значения предикатов. В методе эмуляции поддерживается следующий набор предикатов:

- wasEvent(e) – возвращает True, если в последнем шаге было выбрано для обработки событие e, и False – в противном случае;
- wasInState(sm, s) – возвращает True, если перед последним шагом автомат sm, находился в состоянии s;
- isInState(sm, s) – возвращает True, если после совершения последнего шага автомат sm находится в состоянии s;
- cameToState(sm, s) – возвращает True, если после совершения последнего шага автомат sm сменил свое состояние на s. Это то же самое, что (isInState(sm, s) && !wasInState(sm, s));
- cameToFinalState() – возвращает True, если после совершения шага корневой автомат модели перешел в конечное состояние. Это означает, что автоматная программа завершила работу.
- wasAction(z) – возвращает True, если в ходе выполнения шага было вызвано выходное воздействие z;
- wasFirstAction(z) – возвращает True, если в ходе выполнения шага первым вызванным действием было z;
- wasLastAction(z) – возвращает True, если в ходе выполнения шага последним вызванным действием было z;
- getActionIndex(z) – возвращает номер действия в списке действий, вызванных в ходе выполнения последнего шага. Этот предикат предназначен для того, чтобы формулировать утверждения, задающие порядок вызова действий объектов управления в автоматной программе;
- wasTrue(g) – возвращает True, если в ходе выполнения последнего шага один из переходов был помечен условием g, и его значение было определено как True. g описывает целое условие, а не значение одной переменной. Например, g = «!o1.x1 && o1.x2»;
- wasFalse(g) – возвращает True, если в ходе выполнения последнего шага на одном из переходов встречается условие `g`, и его значение было определено как `False`.

Сравнение метода эмуляции с известным методом

В работе [7] верификация системы автоматов производится следующим образом.

- Система автоматов преобразуется в модель Крипке, записанную на входном языке верификатора SPIN.
- Требования к системе автоматов переводятся в термины построенной модели.
- Модель верифицируется верификатором SPIN. В случае ошибки выдается контрпример в терминах входного языка SPIN.
- Контрпример переводится в термины исходной системы автоматов.

Эта схема верификации изображена на.рис. 1.

Особенностью метода эмуляции является то, что он не требует дополнительных преобразований автоматной программы или контрпримера, сгенерированного верификатором. Модель *Крипке* не строится явно, а алгоритм верификации работает непосредственно с системой автоматов. В результате не требуются ни преобразование системы во входной язык верификатора, ни преобразование требований, ни преобразование сценария ошибки, поскольку сценарий сразу выдается в терминах состояний и переходов в системе автоматов. Схема верификации по методу эмуляции изображена на рис. 2.

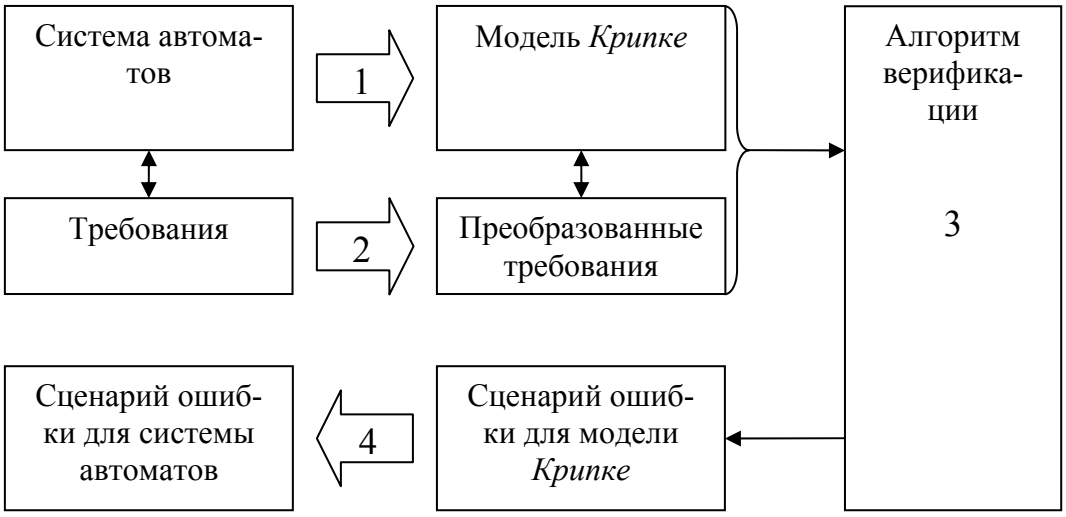


Рис. 1. Схема верификации с явным построением модели Крипке

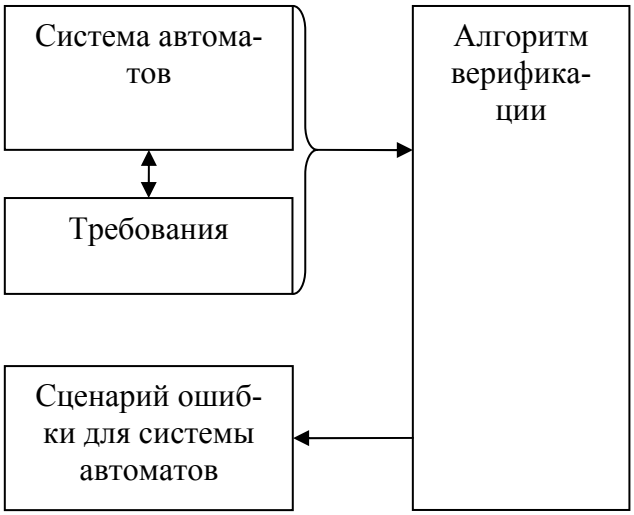


Рис. 2. Схема верификации методом эмуляции

Применение верификатора

Верификатор *UniMod.verifier* был применен для верификации автоматной программы, моделирующей работу банкомата. В следующих разделах описывается автоматная программа банкомата, а затем применение верификатора *UniMod.verifier* для верификации корректности работы банкомата.

Описание банкомата

Банкомат – это устройство, автоматизирующее операции по выдаче и переводу денег, хранящихся в банке, лицу, которому они принадлежат. Идентификация каждого клиента происходит с помощью имеющейся у него кредитной карты банка и соответст-

вующего карте секретного *PIN*-кода. Поэтому человек может вне банка снимать деньги или оплачивать определенные услуги.

Сформулируем основные требования к устройству банкомата. Он должен:

- идентифицировать клиента;
- выполнять операции «Показать доступные средства» и «Снять определенную сумму денег»;
- уметь связываться с банком.

Клиентская программа запускается и предлагает пользователю выполнять различные операции с его личной картой.

Первое, что требуется от пользователя – вставить карту. Далее пользователь вводит свой личный *PIN*-код. Если на сервере не найдена запись о счете с введенным номером и *PIN*-кодом, то работа с этой картой прекращается. Если же *PIN*-код и номер счета были введены правильно, то пользователю предлагается выполнить одну из следующих операций:

- «Забрать карту» – возврат карты. Все текущие операции отменяются, и карта возвращается на руки пользователю;
- «Баланс» – отображает текущий остаток на счете, выводя его на экран и предоставляя возможность распечатать;
- «Снятие денег» – производит операцию снятия денег с карты. Для этого пользователь должен ввести сумму, которую он хочет снять. Клиент пошлет запрос на сервер о текущем балансе и получит ответ. Если на карте достаточно денег, то операция на сервере завершится успешно, и банкомат выдаст требуемую сумму денег. Также при нажатии на кнопку «Печать» будет напечатан чек по данной операции. Если обнаружится, что на карте недостаточно денег, то с карточки ничего не снимется, и клиент выводит соответствующее сообщение на экран.

После возврата карты пользователь может вставить ее снова либо уйти, нажав кнопку «Выход».

Программа банкомата состоит из двух частей – клиентской и серверной. В клиентской части реализован пользовательский интерфейс (*AClient*), а также интерфейс отправки запросов на сервер (*AServer*). Серверная часть производит операции со счетами.

Роль сервера исполняет класс *Server*, написанный и запускающийся отдельно. Поведение клиента моделируется автоматом *AClient* и вложенным в него автоматом *AServer*.

Поставщики событий:

- HardwareEventProvider – системные события, генерируемые оборудованием;
- HumanEventProvider – события, инициируемые пользователем;
- ServerEventProvider – ответы на запросы, поступающие от сервера;
- ClientEventProvider – запросы, поступающие на сервер.

Объекты управления:

- FormPainter – визуализация работы;
- ServerQuery – отправляет запросы на сервер;
- ServerReply – отвечает на клиентские запросы.

На рис. 3 изображена схема связей автоматов *AClient* и *AServer*.

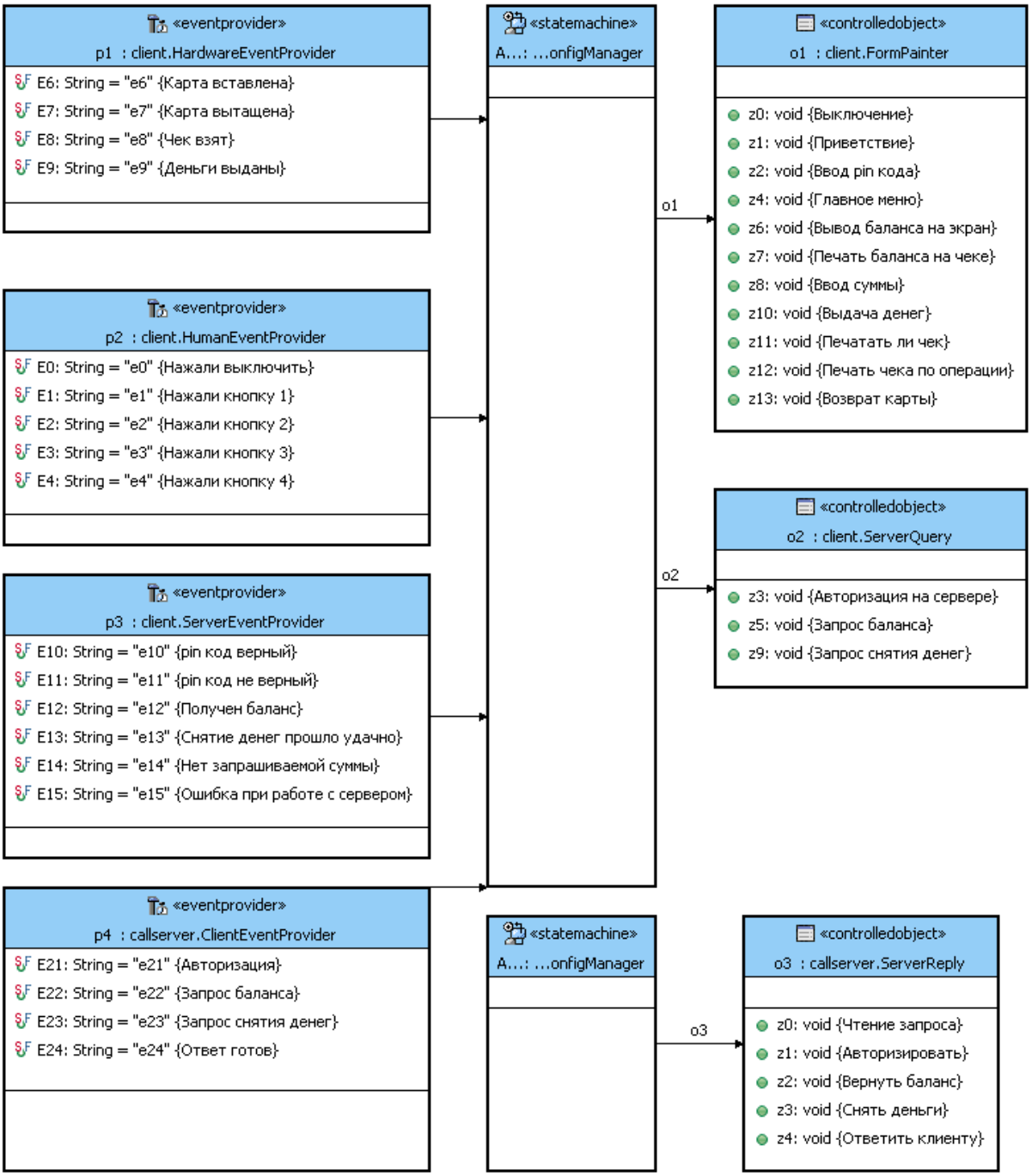


Рис. 3. Схема связей. Несмотря на отсутствие связи между автоматами, AServer вложен в AClient

На рис. 4 приведен граф переходов автомата *AClient*.

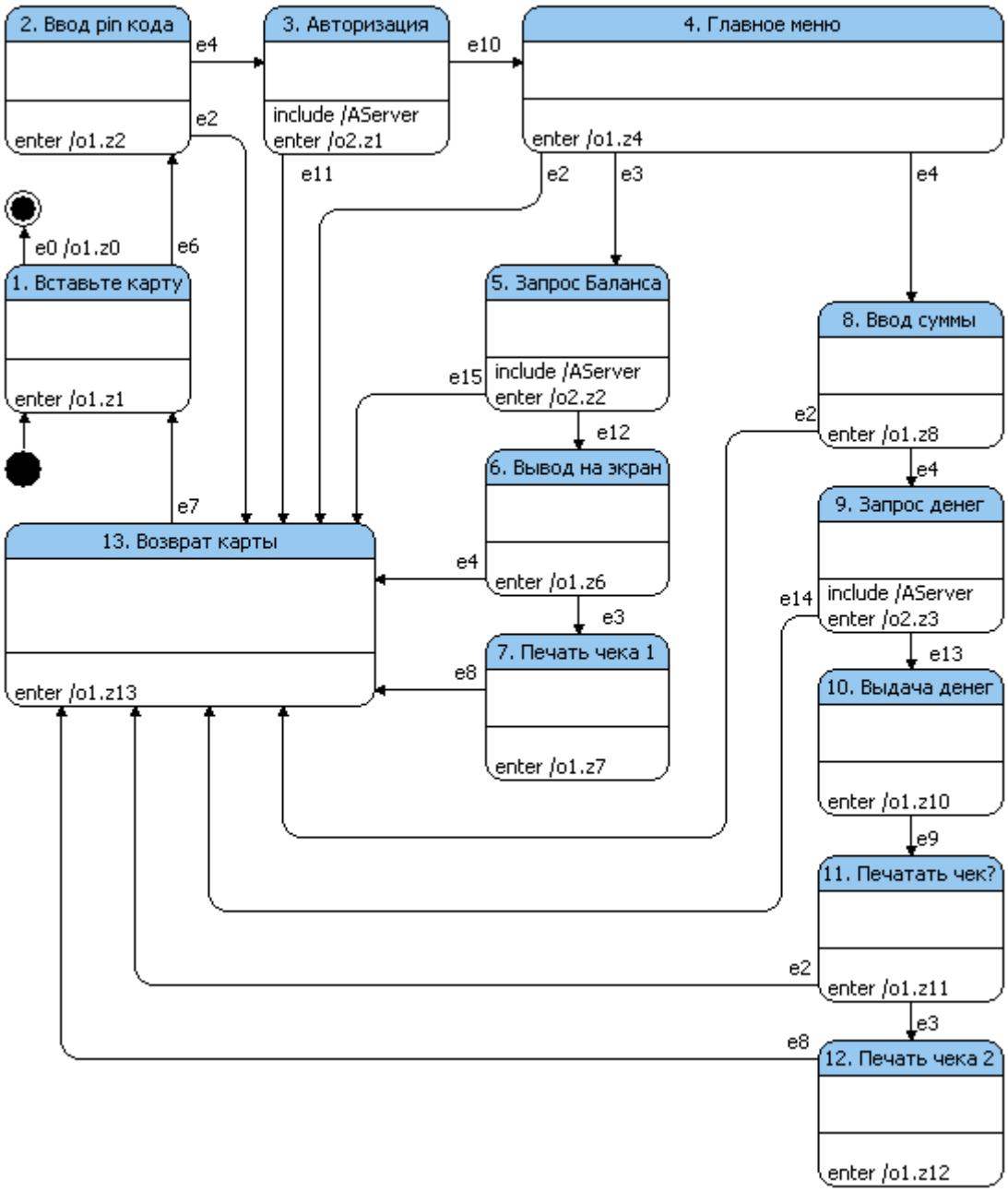


Рис. 4. Автомат *AClient*

На рис. 5 приведен граф переходов автомата *AServer*, посылающего запросы на сервер.

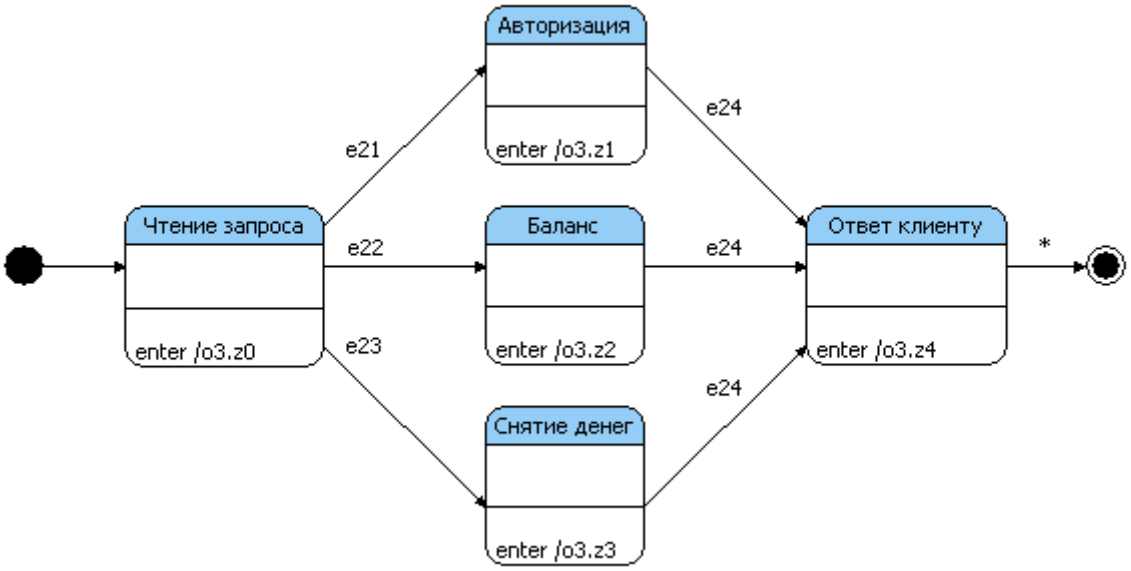


Рис. 5. Автомат *AServer*

Верификация UniMod-банкомата

Используем верификатор *UniMod.verifier* для верификации системы управления банкоматом, описанной выше.

Проверим, что пользователь банкомата не получит денег, пока не введет правильный *PIN*-код. Словесная формулировка требования непосредственно переводится в темпоральную логику *LTL*:

```
[не выдадут деньги] U [введет правильный PIN-код]
```

где `U` – темпоральный оператор `Until` – «пока не». Как показано на рис. 3, в *UniMod*-банкомате выдача денег происходит действием `o1.z10`, а правильно введенный *PIN*-код характеризуется событием `e10`. Таким образом, формула для верификации принимает следующий вид:

```
!o1.z10 U e10
```

где предикат `o1.z10` означает, что было выполнено действие `o1.z10`, а предикат `e10` означает, что произошло событие `e10`. Запишем эту *LTL*-формулу языке BIR — на входном языке верификатора Bogor:

```
LTL.temporalProperty(
   Property.createObservableDictionary(
      Property.createObservableKey("correct_pin",
           AutomataModel.wasEvent(model, "e10")),
      Property.createObservableKey("give_money",
           AutomataModel.wasAction(model, "o1.z10"))
   ),

   LTL.weakUntil(
      LTL.negation(LTL.prop("give_money")),
      LTL.prop("correct_pin")
   )
);
```

Здесь предикат «было выполнено действие `o1.z10`» записан в виде `AutomataModel.wasAction(model, "o1.z10")` и сохранен под ключом «`give_money`». Аналогично, предикат «произошло событие `e10`» записан в виде `Automata-`

`Model.wasEvent(model, "e10")` и сохранен под ключом «`correct_pin`». Эти ключи затем использованы для записи самой темпоральной формулы. Стоит заметить, что вместо темпорального оператора `Until` здесь используется его модификация `weakUntil`. Разница между ними – в том, что `p Until q` требует, чтобы `q` когда-нибудь выполнилось, в то время как `p weakUntil q` этого не требует. Действительно, это оправдано в данном случае, потому что не гарантировано, что пользователь когда-нибудь введет правильный *PIN*-код. При верификации созданной формулы верификатор *UniMod.verifier* выдает следующий результат:

```
Transitions: 1, States: 1, Matched States: 0, Max Depth: 1, Errors found:
0, Used Memory: 2MB
Transitions: 63, States: 41, Matched States: 22, Max Depth: 14, Errors
found: 0, Used Memory: 1MB
Total memory before search: 765a008 bytes (0,73 Mb)
Total memory after search: 1a202a688 bytes (1,15 Mb)
Total search time: 703 ms (0:0:0)
States count: 41
Matched states count: 22
Max depth: 14
Done!
Verification successful!
```

Таким образом, требование, чтобы банкомат не выдавал деньги до введения правильного *PIN*-кода, выполняется в системе автоматов, управляющих банкоматом.

Проверим, что деньги не выдаются, пока не сделан соответствующий запрос. Переведем формулировку в *LTL*-формулу:

```
[не выдаются деньги] U [сделан запрос на выдачу денег]
```

Деньги выдаются с выполнением действия `o1.z10`, а запрос выдачи денег посылается на сервер автоматом *AServer* при событии `e23`, в соответствии со схемой автомата, изображенной на рис. 5. Тогда формула принимает следующий вид:

```
!o1.z10 U e23
```

На языке *BIR* эта формула запишется следующим образом:

```
LTL.temporalProperty (
  Property.createObservableDictionary (
    Property.createObservableKey("money_requested",
         AutomataModel.wasEvent(model, "e23")),
    Property.createObservableKey("give_money",
         AutomataModel.wasAction(model, "o1.z10"))
  ),

  LTL.weakUntil (
    LTL.negation(LTL.prop("give_money")),
    LTL.prop("money_requested")
  )
);
```

Однако, хотя на первый взгляд формула кажется выполняющейся в автомате, верификатор выдает ошибку со следующим сценарием:

```
Model [ step [0] event [null] guards [null] transitions [null] actions
[null] states [null] ] fsaState [T0_init]
Model [ step [0] event [] guards [] transitions [] actions [] states
[(/AClient:9. Запрос денег/AServer) - (Top); (/AClient:5. Запрос
Баланса/AServer) - (Top); (/AClient:3. Авторизация/AServer) - (Top);
(/AClient) - (Top)] ] fsaState [T0_init]
Model [ step [1] event [*] guards [] transitions [s1#1. Вставьте
карту#*#true] actions [o1.z1] states [(/AClient:9. Запрос денег/AServer) -
(Top); (/AClient:5. Запрос Баланса/AServer) - (Top); (/AClient:3.
Авторизация/AServer) - (Top); (/AClient) - (1. Вставьте карту)] ] fsaState
[T0_init]
```

```
Model [ step [2] event [e6] guards [true->true] transitions [1. Вставьте
карту#2. Ввод pin кода#e6#true] actions [o1.z2] states [(/AClient:9. Запрос
денег/AServer) - (Чтение запроса); (/AClient:5. Запрос Баланса/AServer) -
(Авторизация); (/AClient:3. Авторизация/AServer) - (Чтение запроса);
(/AClient) - (2. Ввод pin кода)] ] fsaState [T0_init]
Model [ step [3] event [e4] guards [true->true] transitions [2. Ввод pin
кода#3. Авторизация#e4#true] actions [o2.z3] states [(/AClient:9. Запрос
денег/AServer) - (Чтение запроса); (/AClient:5. Запрос Баланса/AServer) -
(Авторизация); (/AClient:3. Авторизация/AServer) - (Чтение запроса);
(/AClient) - (3. Авторизация)] ] fsaState [T0_init]
Model [ step [4] event [e10] guards [true->true] transitions [3.
Авторизация#4. Главное меню#e10#true] actions [o1.z4] states [(/AClient:9.
Запрос денег/AServer) - (Чтение запроса); (/AClient:5. Запрос
Баланса/AServer) - (Авторизация); (/AClient:3. Авторизация/AServer) -
(Чтение запроса); (/AClient) - (4. Главное меню)] ] fsaState [T0_init]
Model [ step [5] event [e4] guards [true->true] transitions [4. Главное
меню#8. Ввод суммы#e4#true] actions [o1.z8] states [(/AClient:9. Запрос
денег/AServer) - (Чтение запроса); (/AClient:5. Запрос Баланса/AServer) -
(Авторизация); (/AClient:3. Авторизация/AServer) - (Чтение запроса);
(/AClient) - (8. Ввод суммы)] ] fsaState [T0_init]
Model [ step [6] event [e4] guards [true->true] transitions [8. Ввод
суммы#9. Запрос денег#e4#true] actions [o2.z9] states [(/AClient:9. Запрос
денег/AServer) - (Чтение запроса); (/AClient:5. Запрос Баланса/AServer) -
(Авторизация); (/AClient:3. Авторизация/AServer) - (Чтение запроса);
(/AClient) - (9. Запрос денег)] ] fsaState [T0_init]
Model [ step [7] event [e13] guards [true->true] transitions [9. Запрос
денег#10. Выдача денег#e13#true] actions [o1.z10] states [(/AClient:9.
Запрос денег/AServer) - (Чтение запроса); (/AClient:5. Запрос
Баланса/AServer) - (Авторизация); (/AClient:3. Авторизация/AServer) -
(Чтение запроса); (/AClient) - (10. Выдача денег)] ] fsaState
[bad$accept_all]
```

Как видно, на седьмом шаге выполнилось действие `o1.z10`, хотя за всю историю не происходило события `e23`. Посмотрим, что привело к выдаче денег. На шаге 6 автомат *AClient* попал в состояние «9. Запрос денег», в которое вложен автомат *AServer*. При этом выполнилось действие `o2.z3`, которое, как изображено на рис. 3, означает «Запрос снятия денег». Далее произошло событие `e13`, которое означает «Снятие денег прошло удачно». Возникает вопрос, каким образом произошел запрос снятия денег, если не происходило события `e23`. На самом деле, объект управления `o2` (*ServerQuery*) служит для того, чтобы создавать события генератора событий `p4` (*ClientEventProvider*). Объект управления `o2` реализован так, что при вызове действия `o2.z3` («Запрос снятия денег») генерируется событие `e23` («Запрос снятия денег»).

Верификатор *UniMod.verifier* не работает с объектами управления при верификации, поэтому он не может автоматически учитывать их логику при верификации. Таким образом, верификатор «не знает», что если выполнилось событие `o2.z3`, то обязательно произойдет событие `e23`. Для того, чтобы все же верифицировать рассматриваемое свойство банкомата, добавим необходимую логику прямо в верифицируемое свойство, в виде «при условии, что выполняется необходимая логика, верифицируемое свойство тоже выполняется».

Итак, интересующая в данный момент логика объекта управления `o2` заключается в том, что если было выполнено действие `o2.z3`, то будет сгенерировано событие `e23`. Запишем это в логике *LTL*:

```
G (o2.z3 → X e23)
```

где `X` – *LTL*-оператор *Next*. Добавим теперь полученное условие в верифицируемую формулу, используя оператор следования, как было предложено выше:

```
( G (o2.z3 → X e23) ) → ( !o1.z10 U e23 )
```

Теперь запишем эту формулу на языке *BIR*:

```
LTL.temporalProperty (
  Property.createObservableDictionary (
    Property.createObservableKey("server_request_money",
         AutomataModel.wasAction(model, "o2.z3")),
    Property.createObservableKey("money_requested",
         AutomataModel.wasEvent(model, "e23")),
    Property.createObservableKey("give_money",
         AutomataModel.wasAction(model, "o1.z10"))
  ),

  LTL.implication(

    /* Ограничение: o2.z3 генерирует e23 */
    LTL.always (LTL.implication (
      LTL.prop("server_request_money"),
      LTL.next(
        LTL.prop("money_requested")
      )
    )),

    /* Свойство для проверки */
    LTL.weakUntil (
      LTL.negation(LTL.prop("give_money")),
      LTL.prop("money_requested")
    )
  )
);
```

Верификация данной формулы верификатором *UniMod.verifier* оказывается удачной, что означает, что управляющая система банкомата действительно выполняет выдачу денег лишь после того, как запросила наличие денег с сервера.

Проверим, что если произойдет ошибка взаимодействия банкомата с сервером, то карта будет возвращена пользователю. В темпоральной логике такое свойство можно записать следующим образом:

```
G ([произошла ошибка] → F [карта будет возвращена])
```

где `G` – темпоральный оператор *Globally*, а `F` – темпоральный оператор *Future*. Событие «Ошибка при работе с сервером» кодируется в банкомате как `e15`, а возвращение карты – это действие `o1.z13`. Тогда формула принимает следующий вид:

```
G (e15 → F o1.z13)
```

На языке BIR формула выглядит следующим образом:

```
LTL.temporalProperty (
  Property.createObservableDictionary (
    Property.createObservableKey("error",
         AutomataModel.wasEvent(model, "e15")),
    Property.createObservableKey("card_returned",
         AutomataModel.wasAction(model, "o1.z13"))
  ),

  LTL.always (LTL.implication (
    LTL.prop("error"),
    LTL.eventually (LTL.prop ("card_returned"))
  ))
);
```

Верификация данной формулы успешна.

Верифицируем заведомо ложное свойство банкомата, чтобы проверить способность верификатора находить ошибки. Например, проверим, что «пользователь всегда получает деньги». В темпоральной логике оно запишется как

```
G F [пользователь получает деньги]
```

Деньги выдаются действием `o1.z10`, так что формула принимает следующий вид:

```
G F o1.z10
```

На языке *BIR*, соответственно, это записывается как

```
LTL.temporalProperty (
  Property.createObservableDictionary (
    Property.createObservableKey("give_money",
         AutomataModel.wasAction(model, "o1.z10"))
  ),

  LTL.always (LTL.eventually (LTL.prop ("give_money")))
);
```

В результате верификации этой формулы верификатор выдает следующий контрпример:

```
Model [ step [0] event [null] guards [null] transitions [null] actions
[null] states [null] ] fsaState [T0_init]
Model [ step [0] event [] guards [] transitions [] actions [] states
[(/AClient:9. Запрос денег/AServer) - (Top); (/AClient:5. Запрос
Баланса/AServer) - (Top); (/AClient:3. Авторизация/AServer) - (Top);
(/AClient) - (Top)] ] fsaState [T0_init]
Model [ step [1] event [*] guards [] transitions [s1#1. Вставьте
карту#*#true] actions [o1.z1] states [(/AClient:9. Запрос денег/AServer) -
(Top); (/AClient:5. Запрос Баланса/AServer) - (Top); (/AClient:3.
Авторизация/AServer) - (Top); (/AClient) - (1. Вставьте карту)] ] fsaState
[T0_init]
Model [ step [2] event [e0] guards [true->true] transitions [1. Вставьте
карту#s2#e0#true] actions [o1.z0] states [(/AClient:9. Запрос
денег/AServer) - (Top); (/AClient:5. Запрос Баланса/AServer) - (Top);
(/AClient:3. Авторизация/AServer) - (Top); (/AClient) - (s2)] ] fsaState
[T0_init]
Model [ step [2] event [e0] guards [true->true] transitions [1. Вставьте
карту#s2#e0#true] actions [o1.z0] states [(/AClient:9. Запрос
денег/AServer) - (Top); (/AClient:5. Запрос Баланса/AServer) - (Top);
(/AClient:3. Авторизация/AServer) - (Top); (/AClient) - (s2)] ] fsaState
[bad$accept_S2]
Model [ step [2] event [e0] guards [true->true] transitions [1. Вставьте
карту#s2#e0#true] actions [o1.z0] states [(/AClient:9. Запрос
денег/AServer) - (Top); (/AClient:5. Запрос Баланса/AServer) - (Top);
(/AClient:3. Авторизация/AServer) - (Top); (/AClient) - (s2)] ] fsaState
[bad$accept_S2]
```

В этом контрпримере автоматная система совершает лишь два шага: переход из начального состояния в состояние «1. Вставьте карту», и затем переход по нажатию кнопки «Выключить» (событие `e0`) в конечное состояние главного автомата *AClient*. Как и предполагалось, выдачи денег не происходит в этой истории работы банкомата.

Заметим, что в контрпримере строчка с шагом «2» повторяется три раза. Это связано с тем, что после совершения второго шага автоматная система перестает работать, и шаги совершает лишь автомат *Бюхи*: из состояния `T0_init` в состояние `bad$accept_S2`.

Заключение

Рассмотренные примеры показали, что верификатор *UniMod.verifier* адекватно верифицирует предложенные свойства банкомата. Поскольку модель банкомата достаточно простая, видно, какие свойства выполняются, а какие – нет. Это дает возможность проверить, совпадают ли результаты автоматической проверки верификатором с результатами «ручной» или «мысленной» проверки. Примеры показали, что верифика-

тор *UniMod.verifier* успешно верифицирует свойства, которые кажутся выполняющимися, и генерирует контрпримеры для свойств, которые явно не выполняются. Это позволяет говорить о том, что данный верификатор работает корректно.

Кроме того, примеры показали, что анализ генерируемых верификатором контрпримеров прост и прозрачен. Контрпример позволяет достаточно легко выяснить причину невыполнения того или иного свойства.

Несмотря на то, что верификатор не анализирует логику работы объектов управления и генераторов событий, оказалось, что достаточно просто исправить этот недостаток при необходимости. Это было сделано в примере «Деньги не выдаются, пока не сделан соответствующий запрос», где небольшое исправление верифицируемого свойства позволило осуществить его успешную верификацию.

Все это позволяет говорить о том, что верификатор *UniMod.verifier* – надежный, корректный и простой в использовании инструмент.

Литература

1. Гуров В.С., Мазин М.А., Шалыто А.А. UniMod — Инструментальное средство для автоматного программирования // Начуно-технический вестник СПбГУ ИТМО. – 2006. – Вып. 30. Фундаментальные и прикладные исследования информационных систем и технологий. – С. 32–44. – Режим доступа: http://is.ifmo.ru/works/_instrsr.pdf
2. Гуров В., Мазин М., Нарвский А., Шалыто А. UML. SWITCH-технология. Eclipse // Информационно-управляющие системы. – 2004. – № 6. – Режим доступа: http://is.ifmo.ru/works/uml-switch-eclipse/
3. Robby, Dwyer M., Hatcliff J. Bogor: A Flexible Framework for Creating Software Model Checkers. TAIC PART 2006, pp 3–22.
4. Robby, Dwyer M., Hatcliff J. Bogor: An Extensible and Highly-Modular Model Checking Framework, March 2003. // Proceedings of the Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003). Technical Report, SAnToS-TR2003-3.
5. Шалыто А.А. Технология автоматного программирования. / Труды первой Всероссийской научной конференции «Методы и средства обработки информации». – М.: МГУ. 2003. –Режим доступа: http://is.ifmo.ru/works/tech_aut_prog/
6. Deng W., Dwyer M., Hatcliff J., Jung G., Robby, Singh G. Model-checking Middleware-based Event-driven Real-time Embedded Software, March 2003. // Proceedings of the First International Symposium on Formal Methods for Components and Objects (FMCO 2002). Technical Report, SAnToS-TR2003-2.
7. Васильева К.А., Кузьмин Е.В. Верификация автоматных программ с использованием LTL // Моделирование и анализ информационных систем. – Ярославль: ЯрГУ, 2007. – Т. 14. – № 1. – С. 3–14.
8. Кларк Э., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. – М.: МЦНМО, 2002.

УДК 004.4’242

РАЗРАБОТКА ВЕРИФИКАТОРА АВТОМАТНЫХ ПРОГРАММ

К.В. Егоров, А.А. Шалыто

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В статье описан разработанный авторами верификатор автоматных программ, созданных при помощи инструментального средства для поддержки автоматного программирования *UniMod*. При его использовании, в отличие от известных верификаторов, отсутствует необходимость описывать модель на входном языке верификатора. Требования к программе записываются на языке темпоральной логики линейного времени *LTL* (*Linear Time Logic*). При использовании такой логики верификация осуществляется за счет пересечения автомата-произведения модели и автомата *Бюхи*, построенного для отрицания *LTL*-формулы.

Ключевые слова: верификация программ, верификация на моделях, автоматное программирование, автомат Бюхи

Введение

С момента появления первых программ требовалось проверять их правильность, причем не просто удостовериться, что программа работает на конечном числе тестов, а уметь формально доказывать, что ее поведение соответствует спецификации.

Метод проверки того, что программная система соответствует заявленной спецификации *(*обладает необходимыми свойствами или удовлетворяет определенным требованиям (утверждениям)*)*, называется *верификацией.* К сожалению, верифицировать систему обычно намного сложнее, чем ее создать. Поэтому для не очень ответственных систем верификация не всегда оправдана, и в них проще исправлять ошибки по мере обнаружения при тестировании и в процессе работы. Однако существуют такие системы, в которых ошибки нельзя допускать или они могут обойтись слишком дорого [1].

Одним из основных методов проверки программы на наличие ошибок является тестирование. На практике оно применяется в большинстве случаев. Однако «тестирование позволяет показать наличие ошибок, но не их отсутствие» (Э. Дейкстра). При таком подходе к проверке можно удостовериться в правильности работы программы только при определенном ее поведении или каком-то конечном числе входных данных. Однако существуют ошибки, которые могут появляться крайне редко. Поэтому, чтобы исключить возможность их появления, требуется рассмотреть все возможные варианты поведения системы, что при тестировании невозможно.

Наиболее практичным в настоящее время является метод верификации, названный *Model Checking* [2, 3]. При его использовании процесс верификации состоит из трех частей: моделирование программы – преобразование программы в формальную модель с конечным числом состояний для последующей верификации, спецификация – формальная запись утверждений, которые требуется проверить, и собственно верификация. Эти части связаны между собой – алгоритмы верификации зависят от способа построения модели и способа записи требований. При использовании этого метода для программ, написанных традиционно, возникают три проблемы.

1. Как для произвольной программы построить адекватную модель с конечным числом состояний?
2. Как переформулировать требования к программе (системе) в требования к модели?
3. Как при обнаружении ошибки (построении контрпримера) перейти от модели к программе?

Ответ [4, 5] на все эти вопросы может быть найден, если программы являются автоматными [6]. Здесь имеет место та же ситуация, что и при контроле аппаратуры, ко-

торая при сложной логике не может быть проверена, если она не спроектирована специальным образом с учетом контролепригодности.

Цель настоящей работы состоит в разработке верификатора для автоматных программ, созданных при помощи инструментального средства *UniMod* [7].

Особенности этого класса программ позволяют решить первую и третью проблемы верификации, так как каждая автоматная программа уже сама по себе является моделью, которая, в отличие от традиционно написанных программ, пригодна для проверки определенных утверждений о ней либо без ее модификации, как в настоящей работе, либо за счет модификации, которая может быть выполнена автоматически. Вторая проблема в случае автоматных программ решается при проектировании автоматов, устраняя тем самым семантический разрыв между требованиями к программе и модели, который имеет место для традиционно написанных программ.

В настоящей работе требования к программе формулируются в виде формул темпоральной логики линейного времени (*Linear Time Logic, LTL*). Это определяет используемый алгоритм верификации на основе пересечения автоматов *Бюхи* [3].

В ходе работы *создан верификатор*, не использующий уже существующие верификаторы, применение которых связано с преобразованием модели автоматной программы в модель, описываемую на языке верификатора. При применении такого языка после доказательства невыполнимости утверждения об автомате (построении контрпримера) пришлось бы совершать обратное преобразование из модели в автоматную программу.

В связи с ростом числа ядер персональных компьютеров возникает задача *распараллеливания процесса верификации автоматных программ*. Поэтому еще одна задача работы – создание верификатора, который мог бы равномерно распределять нагрузку на ядра и работал бы эффективнее на многоядерном компьютере по сравнению с одноядерным.

Как отмечено выше, цель настоящей работы состоит в разработке верификатора автоматных программ, созданных при помощи *Switch*-технологии [6] в инструментальном средстве *UniMod* [7]. В таких программах выделяются три типа объектов: поставщики событий, система управления и объекты управления.

Система управления представляет собой конечный автомат или систему взаимодействующих автоматов. Автомат – это множество состояний и переходов между ними. Каждый переход помечен событием, при котором он может осуществиться, и условием, выполнимость которого требуется для перехода. Поставщики событий генерируют события, а система управления по каждому событию может совершать переход, считывая значения входных переменных у объектов управления для проверки условия перехода. Такая система называется реагирующей или событийной [8].

UniMod – инструментальное средство, обеспечивающее визуальное проектирование автоматных программ на основе *Switch*-технологии. Это позволяет вынести практически всю логику программы в автоматы, а остальные классы разбить на два типа: поставщики событий и объекты управления. *UniMod* написан на языке программирования *Java* и встраивается в среду разработки *Eclipse* как дополнительный модуль (*plug-in*) [7].

Как отмечалось выше, при верификации программ на языках типа *Java* или *C++,* написанных традиционным путем (без явного выделения состояний), требуется вручную строить по программе модель и описывать ее на языке, понятном используемому верификатору. При этом могут быть утеряны определенные данные и связи в программе, так как приходится переходить на другой уровень абстракции.

Возможны два подхода к использованию автоматной модели для верификации:

- ее формальное преобразование к виду, определяемому выбранным верификатором [5];
- создание верификатора, в котором применяется автоматная модель или некоторое уже существующее ее представление.

В настоящей работе используется второй подход, при котором автоматные программы создаются с помощью инструментального средства *UniMod,* а для верификации применяется *XML*-описание автоматов, являющееся внутренним представлением графов переходов автоматов в указанном средстве.

Автоматная модель, которая строится при создании системы в рамках *Switch*-технологии, может верифицироваться без изменений или с изменениями, которые не приводят к потере данных о ней. Другое достоинство автоматных программ, резко упрощающее их верификацию, – возможность достаточно просто переформулировать требования к системе в высказывания об автоматах, так как в этом случае при проектировании программы строится модель ее поведения, которая может применяться и при верификации.

В настоящей работе верифицируется не вся автоматная программа, а только ее модель, представленная в общем случае системой вложенных автоматов. При этом поставщики событий и объекты управления рассматриваются в качестве «внешней среды», которая ничего не помнит о последовательности переходов рассматриваемого автомата и вызванных действиях. Таким образом, в любой момент времени может быть совершен любой переход из текущего состояния автомата. Такой же подход рассматривается в работе [9].

Для описания требований к автоматным программам будем применять, как отмечено выше, язык *LTL*. В нем время линейно и дискретно. Синтаксис *LTL* включает в себя пропозициональные переменные *Prop*, булевы связки ($\neg$, $\wedge$, $\vee$) и темпоральные операторы. Последние применяются для составления утверждений о событиях в будущем.

Будем использовать следующие темпоральные операторы:

- *X* (ne*X*t) – «*Xp*» – в следующий момент выполнено *p*;
- *F* (in the *F*uture) – «*Fp*» – в некоторый момент в будущем будет выполнено *p*;
- *G* (*G*lobally in the future) – «*Gp*» – всегда в будущем выполняется *p*;
- *U* (*U*ntil) – «*pUq*» – существует состояние, в котором выполнено *q*, и до него во всех предыдущих выполняется *p*;
- *R* (*R*elease) – «*pRq*» – либо во всех состояниях выполняется *q*, либо существует состояние, в котором выполняется *p*, а во всех предыдущих выполнено *q*.

Множество *LTL*-формул таково:

- пропозициональные переменные *Prop*;
- *True, False*
- если φ и ψ – формулы, то
  - $\neg\varphi$, $\varphi\wedge\psi$, $\varphi V\psi$ – формулы;
  - $X\varphi$, $F\varphi$, $G\varphi$, $\varphi U\psi$, $\varphi R\psi$ – формулы.

Оказывается, что как модель автоматной программы, так и *LTL*-формулу можно представить в виде автомата *Бюхи*. Формально он определяется пятеркой (*S, E, T, s0, F*), где

- *S* – конечное множество состояний;
- *E* – множество меток переходов;
- $T \subseteq S \times E \times S$ – множество переходов;
- *s0* – начальное состояние;
- $F \subseteq S$ – множество допускающих состояний.

Тогда путь в этом графе $\pi = s0, s1, s2, \ldots sn, \ldots,$ для которого выполнено $T(si-1, e, si),$ где e – метка перехода, будет последовательностью вычислений системы. Путь является *допускающим*, если существует состояние из множества F, встречающееся бесконечно часто.

Подробно трансляция *LTL*–формулы в автомат *Бюхи* описана в работах [3, 10, 11].

При этом отметим, что в автоматной программе модель поведения может содержать один автомат или являться системой вложенных автоматов. При использовании рассматриваемого подхода по системе автоматов строится автомат-произведение [12]. Автомат или автомат-произведение представляет собой автомат *Бюхи*, в котором метка на переходе – это выполнимость определенного предиката. Под предикатом будем понимать утверждение о текущем переходе, например, вызванные автоматом действия в объектах управления или состояние, в которое перешел автомат.

Для доказательства невыполнимости некоторой *LTL*-формулы на автомате *Бюхи* можно проверить, что пересечение верифицируемого автомата *Бюхи* и автомата *Бюхи*, соответствующего отрицанию *LTL*-формулы, пусто. Для этого требуется доказать, что язык автомата пересечения пуст. Из сказанного следует, что алгоритм верификации может быть следующим: строится автомат *Бюхи* для верифицируемой автоматной программы, по отрицанию *LTL*-формулы строится автомат *Бюхи*, затем строится автомат пересечения, а после этого проверяется, что этот автомат не допускает ни одного слова.

В связи с тем, что рассматриваются бесконечные слова, то, как доказано в работе [3], для пустоты пересечения достаточно доказать, что ни одно допускающее состояние не принадлежит компоненте сильной связности, которая достижима из начального состояния (не существует цикла, проходящего через допускающее состояние). Таким образом, при нахождении цикла, достижимого из начального состояния, будет построен контрпример – путь, на котором не выполняется *LTL*-формула.

При верификации обычно применяют двойной обход в глубину [3], преимущество которого состоит в том, что для реализации этого алгоритма не требуется построение автомата-пересечения целиком – можно строить состояния пересечения автоматов по мере их достижения. Это дает выигрыш на больших моделях.

Общая идея алгоритма такова: обходим в глубину автомат пересечения, при достижении допускающего состояния для проверки достижимости самого себя запускаем второй обход в глубину из данного состояния. Если оказалось, что допускающее состояние достижимо из самого себя, то цикл найден. Следовательно, исходная *LTL*-формула не выполняется на автомате *Бюхи*, представляющем модель программы, и найден контрпример.

Как было отмечено выше, алгоритм верификации *одного автомата* может быть записан следующим образом.

1. Модель программы представляется в виде автомата *Бюхи*.
2. Строится отрицание *LTL*-формулы.
3. По отрицанию *LTL*-формулы строится автомат *Бюхи* с переходами, помеченными специально введенными предикатами.
4. Производится двойной обход в глубину неявного пересечения двух автоматов *Бюхи*. Для построения пересечения выполняются следующие действия:
5. Перебираются все переходы верифицируемого автомата.
6. Перебираются все возможные переходы автомата, построенного по отрицанию *LTL*-формулы, для перехода верифицируемого автомата, полученного на шаге 4.1.

Для верификации *системы вложенных автоматов* применяется такой же алгоритм, только в качестве верифицируемого автомата строится автомат-произведение [12], состояния которого содержат информацию о состояниях всех автоматов иерархической системы. Каждое состояние нового автомата представляет собой дерево, струк-

тура которого совпадает со структурой системы вложенных автоматов. В узлах дерева размещены состояния, в которых находятся соответствующие конечные автоматы. Узел может быть активным, если соответствующий автомат может обрабатывать события, и неактивным, если автомат не может обрабатывать события. Переходы из такого состояния могут совершаться только по переходам одного из активных внутренних состояний. При этом активные и неактивные состояния могут вычисляться следующим образом:

1. Состояние активно, если состояние, в которое вложен автомат, является родителем и активно.
2. Состояние неактивно, если состояние, в которое вложен автомат, не является родителем или неактивно.

При таком подходе, если состояние неактивно, все вложенные в него состояния также неактивны. Это позволяет строить переходы из такого сложного состояния, просматривая не все узлы дерева, а только активные.

Такая структура является неявным произведением автоматов. Однако преимущество этого алгоритма состоит в том, что он позволяет строить произведение системы вложенных автоматов не сразу, а по мере их посещения при обходе в глубину. Это дает возможность обнаружить контрпример до того, как будет построено полное произведение автоматов.

От произведения автоматов нельзя отказаться, так как если требуется делать утверждения о состоянии системы автоматов в целом, то необходимо иметь представление такого глобального состояния. Верификатор, разработанный в настоящей работе, предоставляет возможность проверять утверждения как об отдельном автомате иерархической системы, так и обо всей системе в целом.

Отметим, что проверка утверждения для одного автомата не всегда дает тот же результат, что проверка этого же утверждения для системы автоматов, содержащей этот автомат. Например, утверждение «$G(isInState(A2.s1) \rightarrow X(isInState(A2.s2)))$» (Если автомат *A2* находится в состояние *s1*, то следующим состоянием будет *s2*) вполне может быть истинным для автомата *A2*. Однако если автомат *A2* вложен в *A1*, то это утверждение не будет выполняться для такой системы автоматов, так как если автомат *A2* находится в состоянии *s1*, то следующий переход может совершить автомат *A1* и тогда утверждение не будет выполнено.

Приведем методику использования созданного верификатора. На рис. 1 изображена схема процесса верификации автоматных программ, созданных при помощи инструментального средства *UniMod*.

Разрабатывается спецификация будущей программы. Она описывает поведение программы и требования к ней, которые должны выполняться. Это требуется для того, чтобы в дальнейшем была возможна проверка утверждений о программе.

После создания спецификации возможны два варианта: сначала создать *UniMod*-проект, а затем записать для него словесные требования к системе, проверяемые верификатором, или же сначала сформулировать проверяемые требования, а затем создать программу. Возможно также, что спецификация уже включает в себя четко сформулированные требования об автоматной программе, выполнение которых планируется проверять.

Модель *UniMod*-проекта сохраняется в виде *XML*-файла, который и будет использоваться верификатором для проверки утверждений. *XML*-файл автоматически генерируется инструментальным средством *UniMod* при создании программы. Поэтому можно ожидать, что в нем нет ошибок, свойственных построению вручную.

После словесного описания требований из них выделяются атомарные высказывания (предикаты), соответствующие утверждениям о переходах и состояниях в

UniMod-модели. Например, требование системе «*После возникновения аппаратной ошибки система отменит последнюю операцию*» может быть переформулировано в высказывание об автомате: «*После события p1.e10, рано или поздно будет вызвано действие o1.z10*», где *p1.e10* – событие, посылаемое при аппаратной ошибке, а *o1.z10* – откат последней операции. Такие преобразования над утверждениями позволяют записать требования к модели в виде *LTL*-формул. Если выразительная способность языка *LTL* не позволяет записать требования в виде *LTL*-формул, то они должны быть переформулированы.

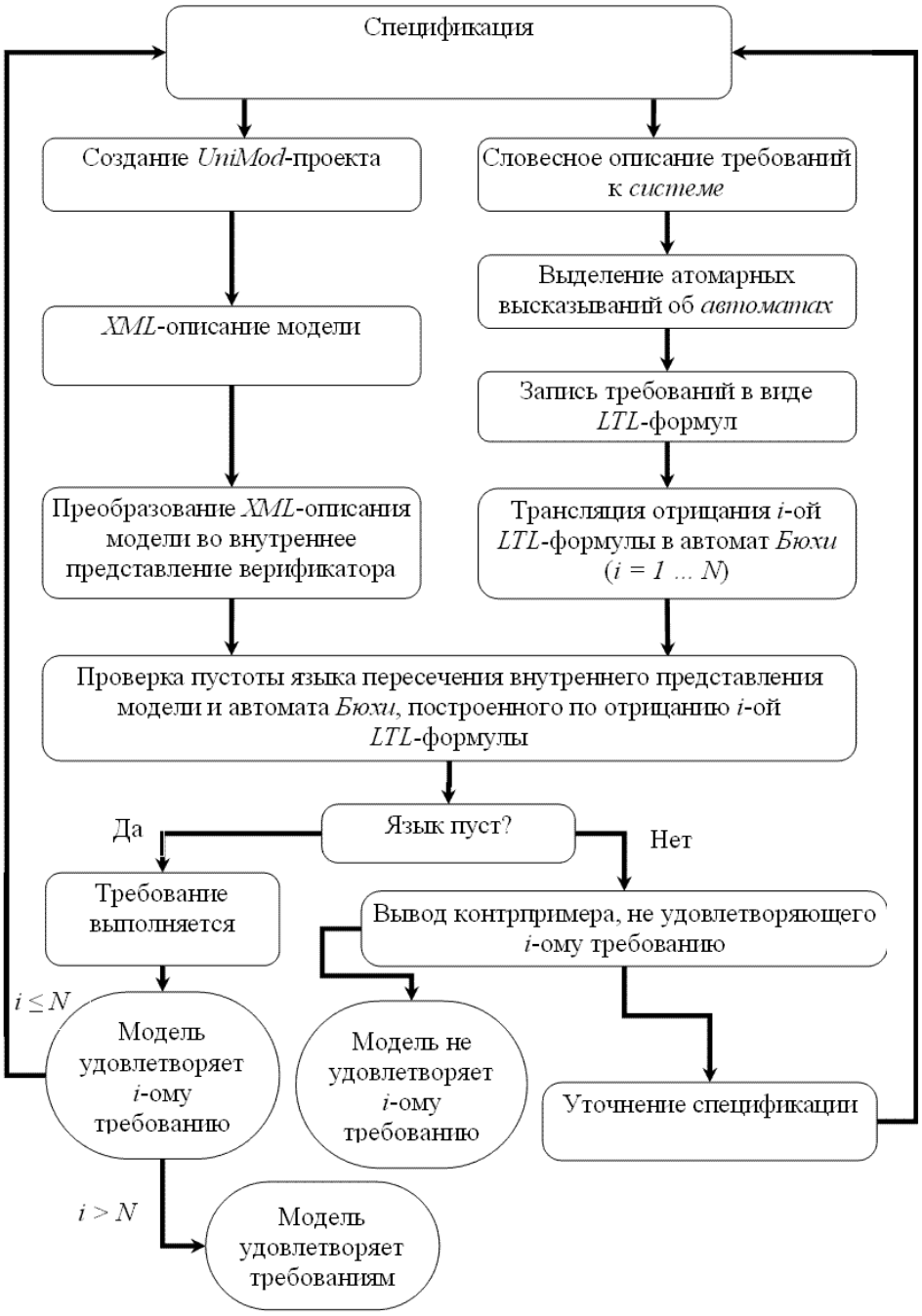


Рис. 1. Методика верификации *UniMod*-моделей

По *XML*-описанию модели и *LTL*-формулам (их отрицаниям) начинается работа созданного верификатора. В ходе работы верификатор подтверждает выполнимость утверждения или выдает контрпример в виде последовательности переходов автомата пересечения автомата модели и автомата *Бюхи*, построенного по инверсии *LTL*-формулы. Пользователю доступен выбор между верификацией одного автомата или иерархической системы автоматов.

Продолжим описание работы верификатора. Верификатор читает *XML*-файл и автоматически строит по нему модель для верификации.

Затем по *LTL*-формуле строится ее отрицание, и оно транслируется в автомат *Бюхи*. Трансляция в автомат *Бюхи* может быть осуществлена как разработанным авторами транслятором, так и при помощи транслятора *LTL2BA* [13].

Затем верификатор совершает двойной обход в глубину по пересечению модели автоматной программы и автомата *Бюхи*, полученного по инверсии *LTL*-формулы. При этом пересечение двух автоматов строится не сразу, а по мере посещения состояний автомата пересечения. Как уже отмечалось выше, при невыполнимости формулы это позволяет обнаружить контрпример, не строя пересечение автоматов в целом.

Двойной обход в глубину может быть реализован в однопоточной и многопоточной (см. последний раздел статьи) формах. Результат работы обеих версий одинаков – контрпример, опровергающий утверждение, или же подтверждение выполнимости формулы.

Если верификатор обнаружил контрпример, то, возможно, найдена ошибка в *UniMod*-модели. Тогда требуется внесение изменения в *UniMod*-модель, ее сохранение в виде *XML*-файла, а затем повторная верификация. Если же контрпример не является ошибкой в *UniMod*-модели из-за неправильной формулировки требований или же из-за неучтенной внутренней реализации поставщиков событий и объектов управления, то необходимо уточнение утверждения, повторная его запись в виде *LTL*-формулы и повторная верификация.

Если верификатор подтвердил выполнимость всех *LTL*-формул, то можно полагать, что модель удовлетворяет заявленным требованиям. Повторная верификация может быть запущена при внесении в модель изменений с целью проверки заявленных свойств.

Для простоты повторной верификации предлагается оформлять каждое проверяемое утверждение в виде отдельного *Unit*-теста. Тогда при внесении изменения в *UniMod*-модель будет иметься возможность повторного запуска всех тестов. При их выполнимости можно утверждать, что модель соответствует заявленным требованиям.

Во время разработки верификатора проводилось тестирование всех его частей на *UniMod*-проектах [14, 15]. На автоматной модели этих проектов были проверенны некоторые свойства. Верификатор доказал верные утверждения и опроверг неверные, тем самым подтвердив возможность своего применения.

Проект [15] реализует банкомат (рис. 2, 3), позволяющий пользователю совершать такие операции, как снятие наличных денег и просмотр доступных средств на счете. Модель банкомата представляет собой двухуровневую структуру автоматов, где автомат *AServer* вложен в автомат *AClient*. Рассмотрим одно из проверенных утверждений про банкомат: «*Пользователь не может запросить снятие наличные или запросить баланс до тех пор, пока не пройдет авторизацию*». При выделении из утверждения предикатов получаем утверждение про автомат: «*Автомат AClient не попадет в состояние «Запрос баланса» или в состояние «Запрос денег» до тех пор, пока не произойдет событие p3.e10*». В утверждении применяется предикат об обработке события *p3.e10*, а не посещение состояния «*Авторизация*», так как это событие означает прохо-

ждение авторизации, а состояние – обращение к серверу для проверки правильности введения *pin*-кода.

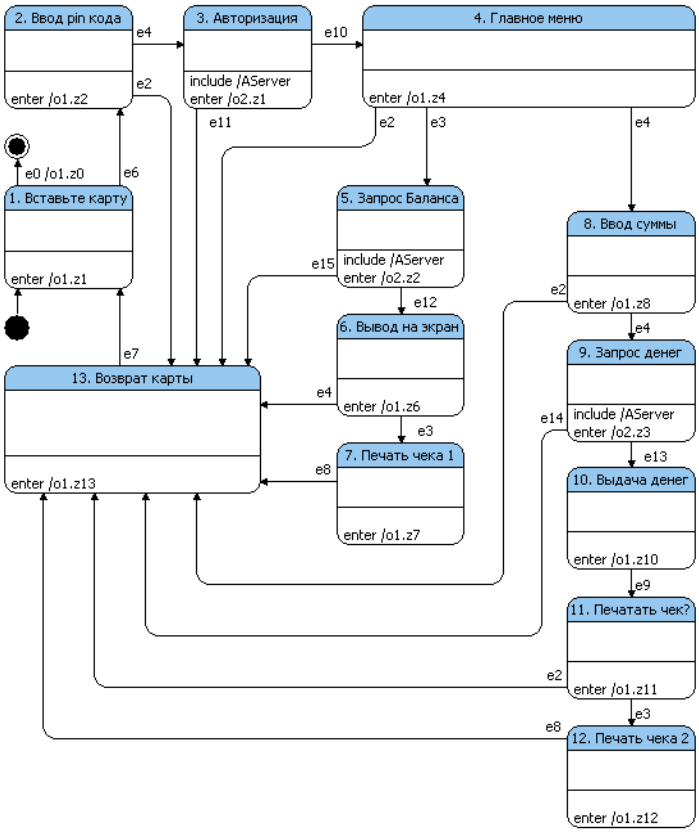


Рис. 2. Модель банкомата (автомат *AClient*)

Верифицируемое утверждение записывается в виде LTL-формулы вида «`wasEvent(p3.e10) R (!isInState(AClient[«Запрос баланса»]) && !isInState(AClient[«Запрос денег»]))`».

Здесь используется оператор R (Release), а не U (Until), так как событие p3.e10 может вообще не произойти из-за недоступности сервера или из-за того, что пользователь забыл свой pin-код. Данное утверждение выполняется для модели банкомата.

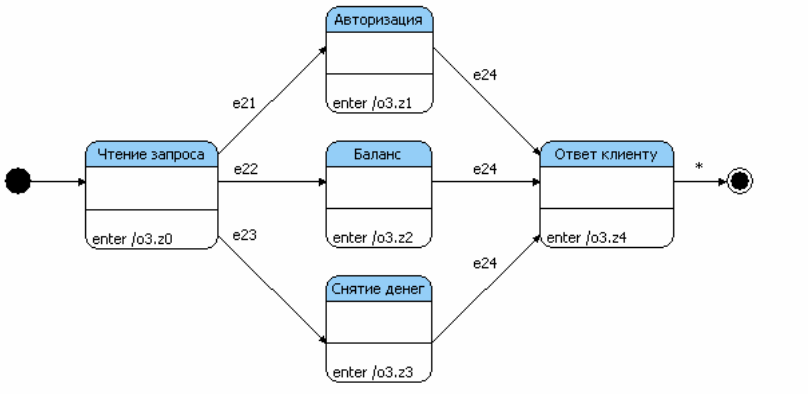


Рис. 3. Модель банкомата (воженный автомат AServer)

Предположим, что кто-то внес в автомат *AClient* еще один переход из состояния «*Возврат карты*» в состояние «*Главное меню*» по событию *e2* (рис. 4). Такое изменение модели нарушает спецификацию банкомата, так как появляется возможность снять наличные или запросить баланс без авторизации. При верификации измененной модели верификатор обнаруживает контрпример, нарушающий спецификацию. На рис. 4 путь, соответствующий контрпримеру, выделен крупными стрелками.

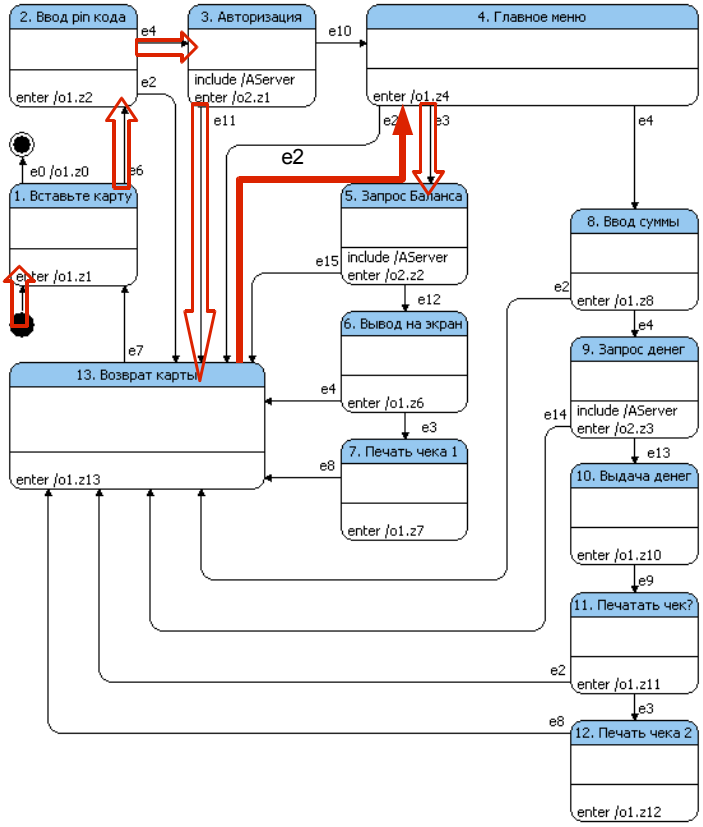


Рис. 4. Измененная модель банкомата. Крупными и жирной стрелками выделена последовательность переходов, нарушающая спецификацию

Этот контрпример верификатор выдает в виде последовательности переходов: [*s1, s1*] → [«*Вставьте карту*», *s1*] → [«*Ввод pin-кода*», *s1*] → [«*Авторизация*», *s1*] → [«*Авторизация*», «*Чтение запроса*»] → [«*Авторизация*», «*Авторизация*»] → [«*Авторизация*», «*Ответ клиенту*»] → [«*Авторизация*», *s2*] → [«*Возврат карты*», *s2*] → [«*Главное меню*», *s2*] → [«*Запрос Баланса*», *s2*]. В квадратных скобках указаны состояния автоматов *AClient* и *AServer*; *s1* – стартовое состояние автомата, а *s2* – завершающее.

Из предложенной последовательности переходов следует, что достичь состояния «*Запрос баланса*» можно, не пройдя авторизацию, так как в ней отсутствует переход по событию *p3.e10*. Достижение данного состояния таким способом свидетельствует о возможности запросить баланс без прохождения авторизации.

Существует реализация верификатора *Spin*, которая работает на нескольких машинах [16, 17], а также работы, предлагающие альтернативные алгоритмы для распределенной верификации [18]. Создание таких верификаторов было обусловлено распределением памяти между несколькими компьютерами. Однако это приводило к значительным сложностям.

В настоящее время проблема с памятью не является существенной, особенно с появлением 64-битных процессоров и винчестеров значительного объема. Благодаря

этому предпочтение отдается параллельным вычислениям, понимая под ними выполнимость на одном компьютере. Поэтому в 2007 г. появилась версия верификатора *Spin*, предназначенная для многоядерной системы [19, 20]. В ней был реализован алгоритм распараллеливания двойного обхода в глубину.

В настоящей работе предлагается другой алгоритм многопоточной версии двойного обхода в глубину. Приведем основную идею этого алгоритма.

Состояния автомата пересечения автомата модели и автомата *Бюхи*, построенного по *LTL*-формуле, обходят в глубину одновременно несколько потоков. У них существует общее множество посещенных состояний. Каждый поток не имеет собственного стека, а формируется общее «дерево стеков». Путь от корня до листа в таком дереве – стек одного из потоков. Как и при обычном обходе в глубину (*Depth-first search, DFS*), обходятся только непосещенные состояния. При посещении всех состояний, достижимых из данного, оно удаляется из дерева в том и только том случае, если у него нет детей. Это означает, что теперь состояние удаляется, только если оно не лежит ни в одном стеке. Удаление состояния из дерева означает удаление его из списка детей его родителя.

Если при обходе в глубину поток t вернулся в состояние s, из которого не ведут переходы в непосещенные состояния, то поток не возвращается в предыдущее состояние, а сначала проверяет, есть ли у данного состояния дети. Наличие ребенка означает, что состояние s находится в стеке другого потока q (возможно, в стеке нескольких потоков). Поэтому поток t не переходит в предыдущее состояние, а сначала пытается помочь потоку q. Для этого он обходит в глубину поддерево с корнем в состояние s, пока не обнаружит такое состояние s', из которого ведут переходы в непосещенные состояния. При обнаружении указанного состояния поток t возобновляет *DFS* по непосещенным состояниям. Если у состояния s нет детей, то первый поток, обнаружившей это, удаляет его. Если s – допускающее состояние, то указанный поток запускает *DFS* для поиска цикла.

Поиск цикла, проходящего через допускающее состояние, может быть запущен не обязательно тем же потоком, который обнаружил данное состояние. Второй *DFS* имеет собственный стек и собственное множество посещенных им состояний. Это исключает коллизии с остальными потоками. Если обнаружен контрпример, то поток заканчивает поиск и информирует все оставшиеся потоки.

Подход, при котором поток не сразу покидает состояние, а сначала помогает другому завершить поиск, позволяет им не простаивать. Например, если в графе переходов существует точка сочленения, то поток, посетивший ее первым, был бы вынужден в одиночку обходить все вершины из второй компоненты связанности, полученной удалением точки сочленения.

В начале работы предложенного алгоритма дерево содержит только стартовую вершину, и все потоки начинают поиск с нее. Поиск заканчивается, если одним из потоков обнаружен контрпример или дерево стало пустым. Таким образом, предложенный метод можно назвать «тройной обход в глубину». Первый обход – поиск по общему дереву такой вершины, из которой существует переход в непосещенное состояние. Второй обход – обход непосещенных состояний и поиск допускающих. Второй обход добавляет вершину в общее дерево при совершении перехода в вершину и может удалять их при покидании, если она не используется другим потоком. Третий обход – поиск цикла, проходящего через допускающее состояние.

Анализ эффективности многопоточной версии верификатора проводился путем сравнения ее с однопоточной версией. Для этого были автоматически созданы модели автоматных программ с различным числом переходов и состояний. Для них проверялась выполнимая формула. При этом при обходе в глубину посещались все состояния (рис. 5).

Из рассмотрения графиков следует, что выигрыш многопоточной версии при проверке выполнимых формул появляется на моделях с большим числом переходов. Однако даже на модели с 30000 переходов время верификации невелико.

Эксперименты также показали, что число состояний и переходов в автомате пересечения увеличивается в несколько раз на сложных *LTL*-формулах. Это может приводить к значительному увеличению времени верификации.

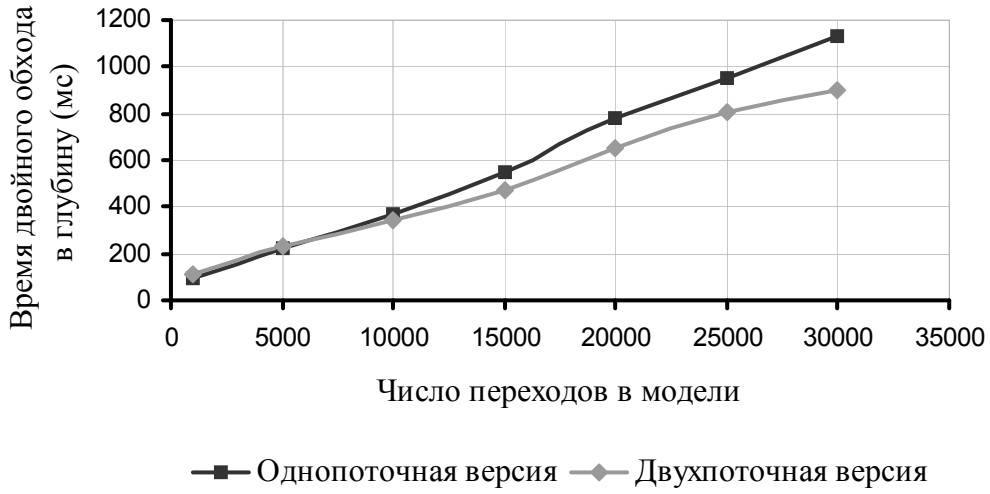


Рис. 5. Зависимость времени работы верификатора от числа переходов автоматной модели

В настоящей работе разработан верификатор автоматных программ, создаваемых при помощи инструментального средства *UniMod*. Он позволяет верифицировать модели программ, которые автоматически строятся верификатором по *XML*-описанию, создаваемому указанным средством по автоматной модели программы. Требования к модели записываются на языке *LTL*. Верификатор предоставляет набор классов и интерфейсов на языке программирования *Java* для проверки выполнимости *LTL*-формул.

При создании верификатора были решены следующие подзадачи:

- трансляция XML-описания модели во внутреннее представление верификатора;
- трансляция LTL-формулы в автомат Бюхи;
- проверка пустоты языка пересечения модели и автомата Бюхи, построенного по отрицанию LTL-формулы.

Для трансляции *LTL*-формулы в автомат *Бюхи* был реализован собственный транслятор и использовался уже существующий транслятор *LTL2BA* [13]. Проверка пустоты языка пересечения двух автоматов осуществляется с помощью двойного обхода в глубину и его многопоточной модификации.

Применение автоматного подхода к написанию программ и созданного верификатора позволяет разрабатывать более надежное программное обеспечение по сравнению с традиционным подходом. Предлагается использовать созданный верификатор для построения и проверки *Unit*-тестов, которые можно запускать на любой стадии жизненного цикла проекта.

Литература

1. Hoffman L. In Search of Dependable Design // Communications of the ACM. – 2008. – V. 51. – № 7. – P. 14–16.

2. Hoffman L. Talking Model-Checking Technology // Communications of the ACM. – 2008. – V. 51. – № 7. – P. 110–112.
3. Кларк Э., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. М.: МЦНМО, 2002. – Режим доступа: http://is.ifmo.ru/verification/_klark_gamberg_pered_verification.djvu
4. Корнеев Г.А., Парфенов В.Г., Шалыто А.А. Верификация автоматных программ / Тезисы докладов Международной научной конференции, посвященной памяти профессора А. М. Богомолова. «Компьютерные науки и технологии». – Саратов: СГУ, 2007. – С. 66–69. – Режим доступа: http://is.ifmo.ru/verification/_KNIT-2007.pdf
5. Васильева К.А., Кузьмин Е.В., Соколов В.А. Верификация автоматных программ с использованием LTL // Моделирование и анализ информационных систем. – 2007. – № 1. – С. 3–14. – Режим доступа: http://is.ifmo.ru/verification/_ltl_aut_ver_1.pdf
6. Шалыто А.А. Switch-технология. Алгоритмизация и программирование задач логического управления. – СПб: Наука, 1998. – Режим доступа: http://is.ifmo.ru/books/switch/1/
7. Гуров В.С., Мазин М.А., Нарвский А.С., Шалыто А.А. UML. SWITCH-технология. Eclipse // Информационно-управляющие системы. – 2004. – № 6. – С. 12–17. – Режим доступа: http://is.ifmo.ru/works/UML-SWITCH-Eclipse.pdf
8. Harel D., et al. Statemate: A Working Environment for the Development of Complex Reactive Systems // IEEE Trans. Software Eng. –1990. – № 4. – P. 403–414.
9. Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Второй этап. СПбГУ ИТМО, 2007. – Режим доступа: http://is.ifmo.ru/verification/_2007_02_report-verification.pdf
10. Gerth R., Peled D., Vardi M. Y.,Wolper P. Simple On-the-fly Automatic Verification of Linear Temporal Logic / Proc. of the 15th Workshopon Protocol Specification, Testing, and Verification. – Warsaw, 1995. – P. 3–18. – Режим доступа: http://citeseer.ist.psu.edu/gerth95simple.html
11. Courcoubetis C., Vardi M., Wolper P., Yannakakis M. Memory-Efficient Algorithms for the Verification of Temporal Properties / Formal Methods in System Design. – 1992. – P. 275–288.
12. Хопкрофт Д., Мотвани Р., Ульман Д. Введение в теорию автоматов, языков и вычислений. – М.: Вильямс, 2002.
13. LTL 2 BA project. – Режим доступа: http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/
14. Егоров К.В., Райков П.М. Игра «Побег». – СПбГУ ИТМО, 2007. – Режим доступа: http://is.ifmo.ru/unimod-projects/la_redada/
15. Козлов В.А., Комалева О.А. Моделирование работы банкомата. – СПбГУ ИТМО, 2006. – Режим доступа: http://is.ifmo.ru/unimod-projects/bankomat/
16. Lerda F., Sisto R. Distributed-Memory Model Checking with SPIN / Proc. of the 5th International SPIN Workshop. – 1999. – P. 3–17.
17. Barnat J., Brim L., Stribrna J. Distributed LTL Model-Checking in SPIN // Lecture Notes in Computer Science. – 2001. – № 2057. – P. 1–17.
18. Lafuente A. Simplified distributed LTL model checking. Technical Report 00176, Institut fur Informatik. University Freiburg. Germany, 2002.
19. Holzmann G.J., Bošnački D. The Design of a Multi-Core Extension of the SPIN Model Checker // IEEE Transactions on Software Engineering. – 2007. – V. 33. – Issue 10. – P. 659–674.
20. Holzmann G.J. A Stack-Slicing Algorithm for Multi-Core Model Checking // Electronic Notes in Theoretical Computer Science (ENTCS). – 2008. – V. 198. – Issue 1. – P. 3–16.

УДК 004.4’242

ПРИМЕНЕНИЕ АВТОМАТНОГО ПОДХОДА ДЛЯ СОЗДАНИЯ КОРРЕКТНЫХ JAVA CARD-ПРИЛОЖЕНИЙ

А.Ю. Законов

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В статье предлагается подход, повышающий надежность процесса разработки Java Card-апплетов. Формулируются особенности проектирования апплетов при применении автоматного подхода, предлагается расширенная утверждениями модель для описания функциональности, генератор кода с заглушками и инструмент для автоматизации тестирования полученных апплетов.

Ключевые слова: верификация программ, Java Card, автоматное программирование

Введение

С каждым днем мобильные устройства занимают все большее место в повседневной жизни. Их функциональность растет, как и сфера их применения. Примером этого являются смарт-карты. Эти устройства получили широкое применение в различных областях – от систем накопительных скидок до кредитных и дебетовых карт, телефонов стандарта *GSM*, карт доступа и проездных билетов. Смарт-карты используют платформу *Java Card* [1], позволяющую программам, написанным на языке *Java*, исполняться на интеллектуальных картах и других устройствах с ограниченными ресурсами. Со временем растет как сложность поставленных задач, так и, соответственно, сложность решающих их программ. При этом повышаются требования к безопасности кода, так как для банковской сферы и сферы мобильных приложений цена ошибки может быть очень велика, и надежность является ключевым вопросом.

Известно, что *Java Card*-апплет функционально эквивалентен конечному автомату как событийный объект [2]. Однако, несмотря на это, на момент написания статьи не существовало работ и инструментов, где проектирование приложения начиналось бы с построения конечного автомата. Идея использовать автоматный подход [3] для описания поведения апплета появилась и была использована в процессе написания *Java Card*-приложения в рамках участия в финале конкурса *SIMagine* [4]. Кроме того, отметим, что существующие средства верификации и тестирования апплетов предназначены для проверки полученного *Java Card*-кода или байт-кода, но никак не охватывают процесс разработки. Создание спецификации, требований и написание кода происходят абсолютно независимо. Таким образом, существует разрыв между спецификацией и реализацией поставленной задачи.

В данной работе поставлена следующая задача: создать подход и набор инструментов для разработки, позволяющих сократить разрыв между описанием функциональности приложения и реализующим его кодом. Подход должен включать в себя все этапы создания *Java Card*-апплета, начиная с проектирования поведения апплета и заканчивая проверкой реализации на предмет соответствия построенной модели.

Предлагаемый подход объединяет в себе преимущества автоматного программирования, автоматической генерации кода [5] и техники автоматической проверки моделей программ *Model Checking* [6]. Для построения автоматной модели используется инструментальное средство *UniMod* [7]. Успешное применение этих технологий позволяет существенно автоматизировать и упростить процесс разработки, а также объединить процессы написания кода и написания документации.

Обзор используемых технологий

Платформа *Java Card* позволяет программам, написанным на языке *Java*, исполняться на интеллектуальных картах и других устройствах с ограниченными ресурсами. В качестве наиболее известных примеров можно привести телефонные SIM-карты и банковские ATM-карты. Наиболее полное описание особенностей данной платформы можно найти на официальном сайте [1].

Приложения, написанные для платформы *Java Card*, называются апплетами. Помимо языка, платформа *Java Card* поддерживает среду исполнения смарт-карт приложений, позволяя одному и тому же *Java Card*-апплету работать на разных смарт-картах (аналогично тому, как *Java*-апплет может быть запущен на разных платформах). Так же, как и в *Java*, это достигается благодаря использованию виртуальной машины *(Java Card Virtual Machine)* и библиотек исполнения. Важными достоинствами технологии *Java Card* являются безопасность и портируемость.

В силу ограничений на доступную память и вычислительную мощность платформа *Java Card* поддерживает только некоторое подмножество языка программирования *Java*. Все языковые конструкции *Java Card* существуют в *Java* и выполняются идентично. В спецификации платформы *Java Card 2.2* многие возможности языка *Java* отсутствуют: длинные примитивные типы данных, символы и строки, многомерные массивы, динамическая загрузка классов, сборка мусора, многопоточность, сериализация и клонирование объектов.

Java Card-код компилируется при помощи стандартного *Java*-компилятора. Полученный байт-код обрабатывается инструментами, специфичными для платформы *Java Card*. Байт-код, поддерживаемый *Java Card VM*, также является подмножеством *Java SE* байт-кода, выполняемого на виртуальной *Java*-машине, но из соображений уменьшения объема байт-кода использует другой принцип сжатия.

Платформа *Java Card* работает по принципу «клиент–сервер» [8]. Данные передаются при помощи *Application Protocol Data Unit (APDU)* пакетов. Апплет реагирует на получение *APDU*-команды, выполняет действия, затем отсылает ответ на запрос. Это напоминает поведение конечного автомата, который реагирует на событие и, в зависимости от типа события, текущего состояния и условий, переходит в новое состояние, выполняя при переходе заданное действие. В работе [3] был предложен метод проектирования программ с явным выделением состояний, названный «автоматное программирование» или «*SWITCH*-технология». В этой технологии базовым является понятие состояния. Автомат рассматривается как совокупность четырех понятий: состояние, переход, входное и выходное воздействие. В контексте платформы *Java Card* входные воздействия соответствуют *APDU*-инструкциям, выходные воздействия являются действиями апплета при получении команды. Состояние апплета может быть определено как совокупность значений всех его переменных. Таким образом, концепция автоматного программирования хорошо подходит для создания *Java Card*-апплетов.

Важным вопросом является проверка корректности полученного апплета. При традиционном подходе тестирование программ и проверка моделей являются далекими друг от друга методологиями. Тестирование направлено на выявление ошибок, а не на доказательство их отсутствия. *Model Checking* [6] – это метод, при котором алгоритмически доказывается, соответствует ли формальная система спецификации или некоторому свойству. Таким образом, проверяют достижимость определенного состояния или же более сложные свойства системы, что позволяет обеспечивать корректность программ. Подробный обзор существующих в этой области результатов приведен в работе [9].

В рамках *Model Checking* отдельно следует отметить подход, при котором построение модели производится автоматически из кода программы. В этом случае на вход подается исполняемый байт-код, и по нему строится модель. Этот подход используется в инструментах *Java Pathfinder* [10] и *jMoped translator* [11].

В последнее время появились инструменты, использующие проверку моделей для тестирования программ. Такой возможностью обладает *jMoped* – среда тестирования программ, написанных на языке *Java*.

Подход, повышающий надежность разработки *Java Card*-апплетов

Для решения поставленной задачи предлагается использовать возможности ряда существующих подходов и создать инструмент, который позволяет сократить разрыв между спецификацией и реализацией приложения. На этапе проектирования применяется автоматный подход и инструментальное средство *UniMod*. Создание прототипа кода выполняется при помощи написанного генератора. Сгенерированный код отвечает за поведение апплета и обработку поступающих сообщений – тем самым гарантируется соответствие поведения разрабатываемого апплета и спроектированного конечного автомата. Полностью описывать функциональность апплета при помощи автоматной модели невозможно. Поэтому обеспечивается сочетание автоматически сгенерированного по автоматам кода и кода для объектов управления, написанного вручную. Предлагается расширить автоматную модель возможностью добавлять утверждения (*assertions*), которые будут накладывать некоторые ограничения и требования на добавленный код. Для проверки сформулированных требований используется транслятор *Java Card* байт-кода в язык *Remolpa*, являющийся расширением транслятора *jMoped*, и верификатор *Moped Model Checker*. Сформулируем общую последовательность действий при разработке *Java Card*-приложения:

1. Создать автоматную модель приложения в инструментальном средстве *UniMod*.
2. Расширить *XML*-представление модели необходимыми утверждениями, проверяющими те свойства реализации, которые возможно сформулировать на этапе проектирования модели и написания требований к приложению.
3. Используя разработанный генератор кода, создать код апплета с заглушками.
4. Вручную реализовать методы объектов управления.
5. Используя предложенные инструменты, проверить, соответствует ли разработанный апплет набору требований, автоматически сгенерированных из свойств, сформулированных на шаге 2 и автоматному описанию, построенному на шаге 1.
6. В случае нарушения тех или иных свойств, верификатор *Moped* выдает в качестве результата контрпример. Для анализа этого контрпримера предлагается использовать написанное автором настоящей работы средство, позволяющее преобразовать данные контрпримера в понятную пользователю последовательность событий и переходов между состояниями конечного автомата, спроектированного на шаге 1.

Перечисленные этапы разработки и применение предложенных инструментов отражены на рис. 1.

Имея средство проверки достижимости состояний и выполнения утверждений *jMoped*, можно проверить программу с реализованными объектами управления на достижимость состояний и последовательность переходов. Получив такую информацию, можно проанализировать корректность добавленного кода и соответствие описанной модели и построенного апплета.

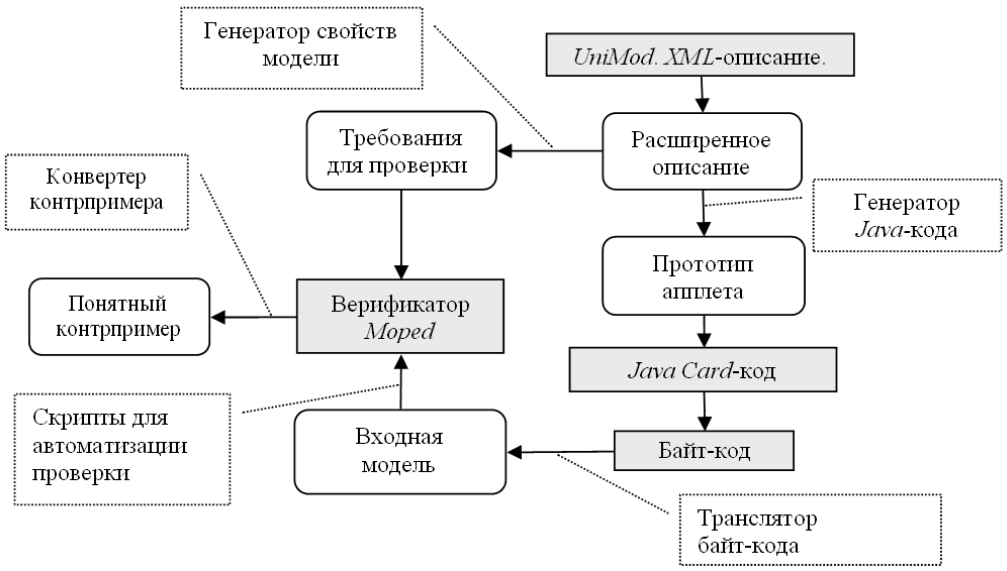


Рис. 1. Предложенный подход

Если на этапе проектирования дополнительно описать необходимые требования к переменным, использованным в функциях объектов управления, то это позволит проверить автоматически, соответствует ли дописанный код этим требованиям.

Расширенная автоматная модель *Java Card*-апплета

Для построения удобной автоматной модели апплета следует определиться с терминами и понятиями, существующими как в *Java Card*, так и в автоматном подходе.

Для *Java Card*-апплета событием является получение *APDU*-сообщения, которое по своей сути является массивом байт. Для удобства использования автоматного подхода, предлагается генерировать код обработки событий, используя информацию поставщиков событий, описываемых в инструменте *UniMod*. Каждому поставщику событий сопоставляется определенное значение `CLA`-байта, а каждому конкретному событию – уникальное значение `INS`-байта [12].

В рамках предложенного подхода возникает вопрос: соответствует ли дописанный руками код условиям модели и выполняет ли он требуемые от него действия? Для автоматически полученного кода на этот вопрос можно ответить, проверяя автоматную модель и корректность процесса генерации. Про написанный вручную код никаких выводов сделать нельзя. Поэтому предлагается расширить существующую модель возможностью добавлять свои условия, выполнимость которых будет проверена на этапе автоматического тестирования. Описанный подход проиллюстрирован на рис. 2.

Еще одним важным аргументом в пользу расширения модели является обработка *APDU*-сообщений, которые присутствуют в большинстве *Java Card*-апплетов и являются проблемными для тестирования. *APDU*-команда содержит две важные составляющие – информацию о произошедшем событии и дополнительные данные, переданные вместе с ним. На поведение и ход исполнения апплета также влияют данные, переданные в качестве параметров. Принимая во внимание, что *Java Card* поддерживает только целочисленные значения переменных, в рамках настоящей работы вводится следующее ограничение: все функции, использующие *APDU*-данные, возвращают целочисленное значение. При этом код этих функций можно исключить из анализа. При правильном распределении функциональности кода формируются два типа функций:

1. Обрабатывающие *APDU*-данные, которые получают необходимую информацию из запроса и возвращают результат исполнения в формате целого числа;
2. Реализующие логику приложения и не использующие *APDU*.

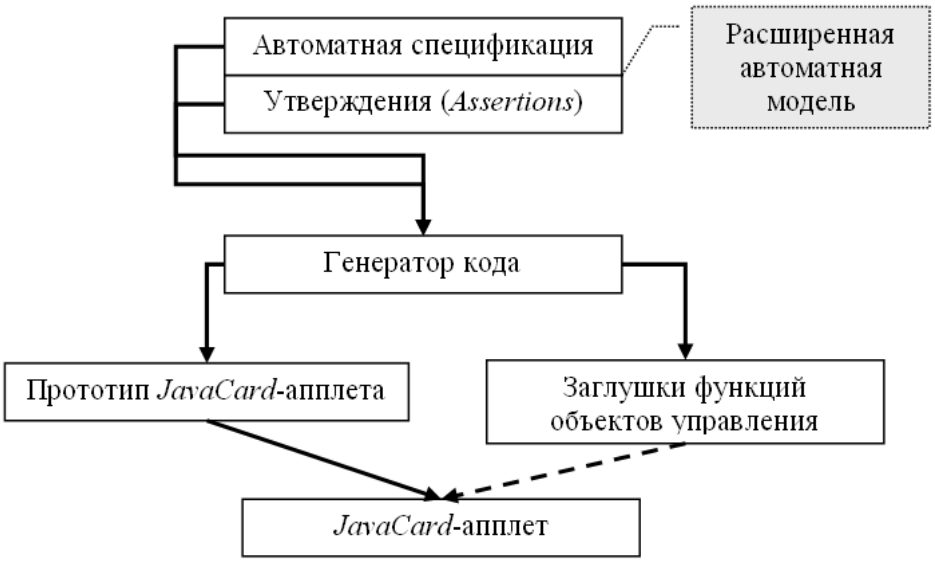


Рис. 2. Подход, использующий расширенную автоматную модель

Исключив из анализа реализацию функций первого типа, можно будет проверять корректность функций второго типа, возможность ошибки в которых значительно более вероятна. Такое решение позволяет абстрагироваться от функций обработки *APDU* и сконцентрироваться на проверке логики выполнения.

Для поддержки предложенной функциональности, расширим существующий способ *XML*-описания модели опциональными тегами и атрибутами. Тег `globalvars` описывает множество глобальных переменных, задействованных в приложении, каждая из которых перечисляется при помощи тега `var`:

```
<!ELEMENT globalvars (var*)>
<!ELEMENT var EMPTY>
 <!ATTLIST var type CDATA #REQUIRED>
 <!ATTLIST var name CDATA #REQUIRED>
 <!ATTLIST var in CDATA #IMPLIED>
 <!ATTLIST var bits CDATA #IMPLIED>
```

Переменные разделяются по смыслу на два типа:

1. Переменные, значения которых будут вычислены приложением;
2. Переменные, значения которых являются входными параметрами апплета, в том числе считываемыми из *APDU*-команды.

Для работы с переменными второго типа используется функциональность транслятора *jMoped*. Интервал значений задается при помощи указания числа бит, в которых будут храниться эти значения. К тегу `outputIndent` добавим возможность задавать пред- и постусловия при помощи тегов `precond` и `postcond`:

```
<!ELEMENT asserts (outputActionCond*)>
<!ELEMENT outputActionCond (precond*, postcond*)>
 <!ATTLIST outputActionCond ident CDATA #REQUIRED>
<!ELEMENT precond EMPTY>
 <!ATTLIST precond value CDATA #REQUIRED>
<!ELEMENT postcond EMPTY>
 <!ATTLIST postcond value CDATA #REQUIRED>
```

Все нововведенные теги являются опциональными. Разработчик может расширить *XML*-описание модели требованиями к переменным на момент вызова функции объекта управления (предусловия) и требованиями к переменным на момент завершения работы функции (постусловия). Соответствующие проверки будут добавлены в код заглушек функций на этапе генерации – это позволит автоматически проверять реализацию заглушек.

Генерация кода и проверка реализации заглушек

Полученный при помощи генератора код является правильным *Java Card*-апплетом, компилируемым, но с ограниченной функциональностью. Поэтому во многих случаях необходимо вручную добавлять код, который выполняет специфические задачи.

Сформулируем свойства реализации, которые хотелось бы проверить и которые позволят проанализировать соответствует ли реализация спроектированной модели:

1. Выполнимость переходов с учетом добавленного кода,
2. Достижимость состояний из начального состояния,
3. Проверка выполнения утверждений в функциях объектов управления.

Следует учитывать, что число путей исполнения апплета растет экспоненциально от числа состояний. Поэтому для сложных систем не все проверки будут выполнимы за разумное время. Однако *Java Card*-апплеты обычно не обладают сложной структурой, и в большинстве случаев проверка будет выполнена достаточно быстро, что позволит выявить потенциальные ошибки реализации.

Пример разработки апплета

Приведем пример применения предлагаемого подхода для разработки небольшого *Java Card*-приложения. Создадим апплет, который превращает смарт-карту в электронную версию кошелька. Существующим решением такой задачи является апплет *Wallet*, код которого включается в набор разработчика (*Java Card Development Kit*) в качестве примера, начиная с версии 2.1.1.

Разработаем аналогичный апплет, используя предложенный подход. Перечислим запросы, которые должен корректно обрабатывать апплет:

- положить деньги на счет;
- снять деньги со счета;
- вернуть текущий баланс;
- проверить *PIN*-код.

Таким образом, можно выделить следующие вопросы, которым стоит уделить внимание в контексте поставленной задачи:

1. получение данных от пользователя,
2. проверка валидности полученного запроса,
3. вызов методов только из допустимых состояний.

Первым этапом разработки является построение автоматной модели. Схема переходов автомата *A1*, который описывает логику приложения, изображена на рис. 3.

Приведем основные элементы *XML*-представления описанного конечного автомата, которое генерируется инструментом *UniMod*:

```
<stateMachine name="A1">
 <state name="s1" type="INITIAL"/>
 ...
 <state name="s5 Check PIN" type="NORMAL"/>
```

```
<transition event="e5" sourceRef="s2 Credit operation"
        targetRef="s1 Stand By"/>
...
<transition event="e5" sourceRef="s5 Check PIN" targetRef="s1
        Stand By"/>
</stateMachine>
```

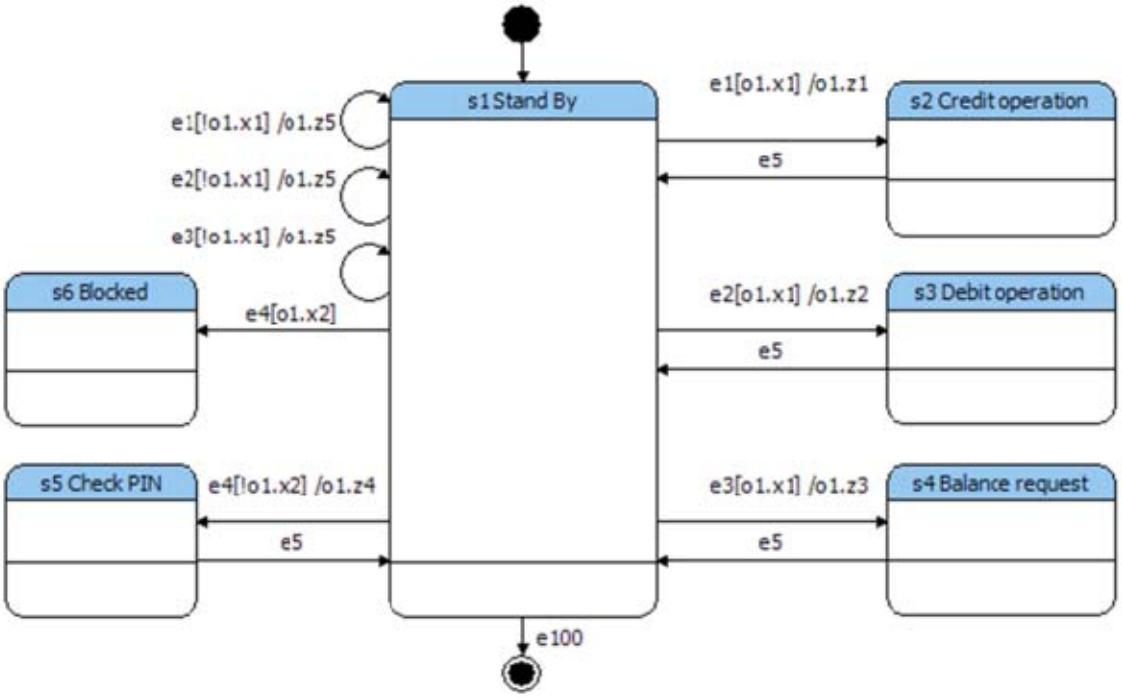


Рис. 3. Граф переходов автомата *A1*

Построение автоматной модели требуемого апплета позволило решить целый ряд поставленных задач.

1. Вызов функции объектов управления производится только из тех состояний, в которых это явно указано и только при получении соответствующих событий и выполнении необходимых условий.
2. Апплет реагирует только на явно перечисленные команды – получение неизвестной команды будет проигнорировано.
3. Перед выполнением любой операции со счетом проверяется, авторизован ли пользователь.

Нерешенными остались задачи, связанные с численными ограничениями – размер транзакции и число попыток авторизации. Следует отметить, что задача проверки числа попыток при авторизации может быть решена при помощи автоматного подхода: путем проектирования небольшого автомата и создания нового поставщика событий. Однако усилия, потраченные на введение этой проверки в автоматную модель, и снижение степени понятности модели делают это решение невыгодным.

Предложенный подход позволяет реализовать другое решение. Проанализируем требования, накладываемые на функции объекта управления, и добавим необходимые проверки в *XML*-описание автомата, учитывающие особенности работы электронного кошелька. Для начала выделим глобальные переменные, которые будут участвовать в логике приложения:

- short balance – сумма денег в кошельке;
- short transaction – размер запрашиваемой транзакции;

- byte pin_counter – допустимое число попыток авторизации.

Добавим перечисленные переменные в *XML*-описание:

```
<globalvars>
 <var type="short" name="balance" />
 <var in=”true” bits=”8” type="short" name="transaction" />
 <var type="byte" name="pin_counter" />
 <var type="short" name="MAX_TRANSACTION_AMOUNT" />
 <var type="short" name="MAX_BALANCE" />
 <var type="short" name="PIN_TRY_LIMIT" />
</globalvars>
```

Рассмотрим ограничения, сформулированные в *Wallet*-апплете из набора разработчика. Максимальное значение баланса ограничено снизу нулем, сверху константой `MAX_BALANCE`. Аналогично ограничен размер одной операции над счетом: снизу нулем, а сверху константой `MAX_TRANSACTION_AMOUNT`. Также накладываются ограничения на авторизацию: `PIN_TRY_LIMIT` – максимальное число попыток авторизации. Переменные `balance` и `transition` задействованы в функциях `o1.z1` (*credit*) и `o1.z2` (*debit*). Можно дописать необходимые условия на значения этих переменных к вызовам функции объектов управления:

```
<outputActionCond ident="o1.z1">
 <precond value="transaction &lt; MAX_TRANSACTION_AMOUNT"/>
 <precond value="transaction &gt; 0"/>
 <postcond value='balance &lt; MAX_BALANCE'/>
</outputActionCond>
<outputActionCond ident="o1.z2">
 <precond value="transaction &lt; MAX_TRANSACTION_AMOUNT"/>
 <precond value="transaction &gt; 0"/>
 <postcond value="balance &gt; 0"/>
</outputActionCond>
<outputActionCond ident="o1.z4">
 <precond value="pin_counter &lt; PIN_TRY_LIMIT"/>
 <precond value="transaction &gt; 0"/>
 <postcond value="pin_counter == 0"/>
</outputActionCond>
```

В результате сгенерированные заглушки функций объектов управления имеют следующий вид:

```
private void o1_z2() {
 /* preconditions */
 assert (transaction < MAX_TRANSACTION_AMOUNT);
 assert (transaction > 0);
 /* stub implementation */
 /* precondition */
 assert (balance > 0);
}
```

Для иллюстрации возможностей предложенного метода проверки реализуем любой из методов заведомо неверным способом и посмотрим, будет ли обнаружена эта ошибка. Пусть метод `o1.z1` содержит цикл с ошибочным условием:

```
private void o1_z1() {
 byte b = 2;
 while (b <= 7) { //b will never be > 7
 b = b+1;
 b = b-1;
 }
}
```

Предложенная реализация метода нормально компилируется. В случае проверки выполнимости переходов будет установлено, что состояние *s2* недостижимо, так как переход невыполним. Следовательно, приложение никогда не будет выполняться корректно, и эта реализация явно не соответствует спецификации. Запустив проверку,

можно выявить несоблюдение ограничений на значение переменной. Если верификатор обнаруживает, что нарушено какое-либо утверждение, то выводятся значения аргументов, которые привели в это состояние:

```
"error" is reachable!
Trace:
s1_Stand_By
e1
transaction s1_Stand_by_to_s2_Credit_Operation

Результат показывает, что не выполнено следующее утверждение в
        функции o1.z1:
assert (transaction < MAX_TRANSACTION_AMOUNT);
```

Предложенный подход позволяет выявить потенциальные ошибки в коде:

- ошибки в условиях циклов в объектах управления, которые приводят к зацикливанию приложения;
- отсутствие явных проверок значений – при реализации не соблюдены ограничения, заданные в спецификации;
- недостижимость состояния, так как условие входа в него никогда не может быть выполнено;
- существование пути исполнения, при котором не выполнена очистка значения переменной или после нескольких итераций значение переменной нарушает требования спецификации – нарушены постусловия функции.

Разработка описанного примера на практике подтвердила преимущества использования автоматного подхода при проектировании приложений для *Java Card*.

Заключение

Перечислим результаты, полученные в данной работе.

1. Предложен метод разработки *Java Card*-апплетов с применением автоматного подхода. Сформулирована связь между понятиями, используемыми в *Java Card* и в автоматном подходе. Показаны преимущества этого подхода.
2. Предложен расширенный вариант *XML*-представления автоматной модели, позволяющий на этапе проектирования задавать условия на реализацию методов объектов управления.
3. Написан генератор кода с заглушками для *Java Card*-апплетов из предложенного *XML*-представления, учитывающий особенности платформы *Java Card* и генерирующий утверждения, соответствующие условиям, добавляемым в модель.
4. Расширена функциональность транслятора *jMoped*. Учтены особенности платформы *Java Card* и использованы преимущества того, что в основе кода лежит автоматная модель.
5. Предложены методы проверки полученной модели, позволяющие находить ошибки в реализации объектов управления и выявлять несоответствия полученного *Java Card*-апплета и спроектированной автоматной модели.
6. Создан инструмент, автоматизирующий предложенный процесс тестирования и конвертирующий контрпримеры верификатора в удобный формат, содержащий элементы автоматной модели.

Литература

1. Технология Java Card. Официальный сайт. – Режим доступа: http://java.sun.com/javacard/
2. Java Card. Wikipedia. The free encyclopedia. – Режим доступа: http://en.wikipedia.org/wiki/Java_Card

3. Шалыто А.А. Switch-технология. Алгоритмизация и программирование задач логического управления – СПб: Наука, 1998. – Режим доступа: http://is.ifmo.ru/books/alg_log
4. Конкурс SIMagine. – Режим доступа: http://www.simagine.gemalto.com
5. Чарнецки К., Айзенекер У. Порождающее программирование: методы, инструменты, применение – СПб.: Питер, 2005.
6. Вельдер С.Э., Шалыто А.А. Введение в верификацию автоматных программ на основе метода Model checking. – Режим доступа: http://is.ifmo.ru/download/modelchecking.pdf
7. Гуров В.С., Мазин М.А. Веб-сайт проекта UniMod. – Режим доступа: http://unimod.sourceforge.net
8. Технология «клиент-сервер». Wikipedia. The free encyclopedia. – Режим доступа: http://en.wikipedia.org/wiki/Master-slave_%28computers%29
9. Шалыто А.А., Красс А.С. Отчет о патентных исследованиях по теме разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. – Режим доступа: http://is.ifmo.ru/verification/_2007_01_patent-verification.pdf
10. Havelund K., Pressburger T. Model Checking Java Programs using Java PathFinder. 1998. – Режим доступа: www.havelund.com/Publications/
11. Среда тестирования jMoped translator. – Режим доступа: www7.in.tum.de/tools/jmoped/
12. APDU. Wikipedia. The free encyclopedia. – Режим доступа: http://en.wikipedia.org/wiki/Protocol_data_unit

УДК 004.4’242

РАЗРАБОТКА КОРРЕКТНЫХ JAVA CARD-ПРОГРАММ НА ОСНОВЕ АВТОМАТНОГО ПОДХОДА

А.А. Клебанов, А.А. Шалыто

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В данной работе описываются результаты исследований, направленных на создание корректного *Java Card*-кода. При этом код генерируется из высокоуровневого описания на основе технологии автоматного программирования. Дополнительным достоинством подобного подхода является возможность генерации формальной спецификации приложения. Соответствие исходного или *byte*-кода спецификации может быть проверено различными верификаторами или средствами динамической или статической проверки.
Ключевые слова: program verification, Java Card, automata-based programming

Введение

Смарт-карта [1] – это пластиковая карта, в которую встроены чип и память, позволяющие хранить и обрабатывать информацию. Фактически смарт-карты представляют собой защищенные от постороннего вмешательства компьютеры размера кредитной карты. Помимо защищенности, эти карты обладают такими достоинствами, как мобильность и простота использования. Они обеспечивают хранение информации, аутентификацию, применяются в платежных системах, мобильной связи и т. д.

Технология *Java Card* [1] адаптирует платформу *Java* для применения на смарт-картах. В дополнение к достоинствам, свойственным технологии *Java* (безопасность, надежность, поддержка принципов объектно-ориентированного программирования, кроссплатформенность и т. д.), при использовании рассматриваемой технологии, во-первых, значительно упрощается процесс разработки (платформа *Java Card* позволяет программисту абстрагироваться от внутренних особенностей карт каждого конкретного производителя), а, во-вторых, вводится механизм запуска нескольких приложений на одной карте.

Java Card API является «надмножеством подмножества» *Java API.* Это означает, что из-за ограниченности ресурсов карты в *Java Card* отсутствует поддержка многопоточности, строк, многомерных массивов, сборки мусора и т.д. Однако *Java Card API* расширен дополнительной функциональностью, необходимой для решения задач, возникающих при применении смарт-карт. Это расширение включает в себя отправку и прием специальных команд (*Application Protocol Data Unit,* (*APDU*)-команд), работу с *PIN*-кодами, криптографическими алгоритмами и т.д. Заметим, что *Java Card*-приложения обычно называются апплетами. Для сохранения логической целостности некоторые аспекты технологии *Java Card* будут более подробно описаны в тех частях работы, где это необходимо для понимания.

Формальные методы – математически строгие методы и инструменты для спецификации, проектирования и верификации программного обеспечения и аппаратных средств [2]. Аналогично автоматному программированию [3], их применение в программной инженерии основывается на популярной идее использования надежных и проверенных технологий из других инженерных дисциплин. Общая цель использования формальных методов состоит в создании надежных программного обеспечения и аппаратных средств.

Существует ряд причин [4], по которым технология *Java Card* является интересной для исследователей в сфере формальных методов. Во-первых, *Java Card*-приложения очень критичны по отношению к вопросам безопасности и надежности, что объясняется спецификой областей применения смарт-карт, в которых цена ошибки весьма велика. Во-вторых, *Java Card* является самой распространенной *Java*-платформой в мире, так как смарт-карты издаются огромными тиражами. Поэтому, в отличие от программного обеспечения для персональных компьютеров, обновлять или исправлять приложения на уже выпущенных картах может быть достаточно трудоемко и экономически невыгодно. И, наконец, ограниченность ресурсов и меньшая функциональность *Java Card* по сравнению с технологией *Java* гарантирует, что программы не будут обладать слишком сложным поведением и большим числом управляющих состояний [3]. *Java Card*-приложения могут стать хорошей «тренировочной площадкой» для различных формальных методов – с одной стороны, программы в этом случае достаточно просты, а с другой – они представляют конкретный практический интерес, а не являются искусственно созданным «игрушечным» кодом. Таким образом, задача применения формальных методов для проектирования и верификации *Java Card*-приложений представляется актуальной и реализуемой.

Цель настоящей работы – расширение технологии автоматного программирования для создания корректных *Java Card*-программ. При этом необходимо решить ряд задач, связанных с реализацией возможности генерации скелета исходного кода *Java Card*-приложений и их последующей *верификацией* на уровне исходного и *byte*-кодов.

Анализ существующих работ

Из изложенного следует, что анализ существующих работ можно проводить в двух направлениях, одним из которых является генерация *Java Card*-кода из высокоуровневых спецификаций, а вторым – верификация автоматных программ.

Генерация *Java Card*-кода из высокоуровневых спецификаций

Существуют два исследовательских проекта, направленных на генерацию *Java Card*-кода из высокоуровневых спецификаций. Первый из них [5] основывается на предметно-ориентированном языке *SmartSlang*, который позволяет описывать программы с помощью высокоуровневых конструкций, характерных для смарт-карт. Од-

нако в нем не используется автоматный подход, и поэтому проект имеет мало общего с настоящей работой.

Второй проект рассмотрен в работах [6, 7]. В этих исследованиях код генерируется на основе конечного автомата, описывающего апплет. В работе [6] используется верификатор совместно с редактором конечных автоматов, а в работе [7] для описания автоматов используется *UML*. Настоящая работа имеет ряд преимуществ по сравнению с этими исследованиями.

Во-первых, применяется технология автоматного программирования, которая предполагает описание поведения иерархической системой автоматов, а не только одним автоматом как в работах [6, 7]. В работе [6] остается открытым вопрос о моделировании самого хост-приложения и его взаимодействия со смарт-картой. Выделение хост-приложения в качестве отдельного поставщика событий в рамках автоматного подхода в достаточной степени решает этот вопрос. Нотация, используемая в технологии автоматного программирования, скрывает низкоуровневые особенности языка *Java Card*, что облегчает процесс разработки программ и повышает их надежность. Действительно, программисту предлагается оперировать не массивами байт, а короткими идентификаторами, для которых всегда доступно текстовое описание. Во-вторых, в предлагаемом подходе генерируется более полная спецификация. В частности, при определенных ограничениях существует возможность генерации предусловий методов, вызываемых при входе в состояние. В работе [6] этот вопрос также упоминается как открытый. Наконец, для инструментального средства *UniMod* [8] разработан целый ряд плагинов, позволяющих осуществить верификацию автоматной модели [9]. Заметим, что в работе [6] верификатор используется исключительно как графический редактор для создания автоматов, а его применение по прямому назначению упоминается лишь в разделе, описывающем дальнейшие исследования. Однако в более поздней работе [7] верификация модели не упоминается вообще. Более того, вопрос генерации кода считается второстепенным, а привлекательной задачей считается извлечение автоматной модели из исходного кода, что противоположно цели настоящей работы.

Подходы к верификации автоматных программ

Подходы к верификации программ можно разделить на статические и динамические.

Наиболее типичным примером динамической верификации является тестирование. Суть этого метода заключается в проверке функциональности программы на некоторых примерах. Несмотря на массовую распространенность такого подхода, он обладает существенным недостатком – зависимостью от конкретного сценария. Общеизвестно (Э. Дейкстра), что тестированием невозможно доказать отсутствие ошибок в программе.

В данной работе вопрос верификации приложений в большей степени рассматривается как вопрос статической проверки, к которой относятся технологии *Model checking* [10] и доказательства теорем [11].

При использовании автоматного подхода ключевую роль играет технология *Model checking* [10, 12, 13]. Она позволяет автоматически проверить выполняется ли заданное требование, выраженное на языке темпоральной логики, на модели поведения системы с конечным числом состояний. Грубо говоря, проверка осуществляется с помощью поиска по всему множеству состояний, а конечность модели гарантирует то, что поиск со временем завершится. Очевидно, что для программ общего вида применение этой технологии является весьма проблематичным. Модель приходится строить эвристически. При этом встает вопрос о том, насколько адекватна полученная модель исходной программе. Однако *Model checking* идеально подходит для верификации автоматных программ.

Действительно, *Model Checking* предполагает формальное построение по программе модели, которая является специальным видом конечного автомата. Поэтому процесс построения модели из автоматной программы может быть автоматизирован, и, как следствие, исчезнут ошибки, вызванные несоответствием модели программе. Запись формальных требований для автоматных программ по сравнению с построенными традиционно также упрощается, так как для этого класса программ семантический разрыв между требованиями к программе и требованиями к модели практически устраняется в ходе построения автоматов. При ошибке выдается контрпример, его трансляция в автоматную программу также может быть выполнена автоматически. Недостаток технологии *Model checking* применительно к автоматному программированию состоит в том, что она охватывает лишь логическую часть программы.

Для настоящей работы важна и другая технология верификации – метод доказательства теорем. В рамках этой технологии и сама система, и требования к ней выражаются с помощью формул математической логики. Логика задается набором аксиом и правилами вывода, а процесс верификации состоит в доказательстве необходимого требования в данной логике. В отличие от технологии *Model checking*, доказательство теорем может выполняться и в случае бесконечномерного пространства состояний. Недостатком технологий доказательства теорем является наличие сообщений о несуществующих ошибках, часто свидетельствующих о проблеме в системе доказательства, а не в самой программе. Технология *Model checking* не обладает подобным недостатком.

Логично поставить вопрос о возможности комбинирования этих двух технологий таким образом, чтобы компенсировать недостатки каждой из них. Варианты возможно решения этого вопроса приводятся в ряде работ.

В работе [14], посвященной текущему состоянию и перспективам формальных методов, есть замечание о том, что для описания и анализа сложной системы, скорее всего, будет недостаточно использовать только один формальный метод. Поэтому становится важным вопрос их комбинирования. Основным направлением в этой области считается сочетание технологий проверки моделей и доказательства теорем. Предлагается два подхода – во-первых, использовать *Model checking* как средство принятия решений при доказательстве теорем, а во-вторых, с помощью систем доказательства теорем получать модели в виде конечных автоматов и уже к ним применять проверку моделей.

В настоящей работе используется другой подход, который будет подробно описан ниже. Его суть в том, что на разных этапах разработки и внедрения приложения предлагается применять различные методы проверки, актуальные для текущей стадии. При этом можно сочетать верификацию модели с верификацией ее реализации, включая верификацию *byte*-кода.

Заметим, что в работе [12] также утверждается, что верификация методом доказательства теорем применима к автоматным программам. Однако, по мнению авторов, этот подход очень трудоемок и бесполезен для проверки логики программы, а может использоваться только для проверки входных и выходных воздействий. В настоящей работе сделана попытка опровергнуть первую часть утверждения (вторая часть полностью подтверждается) – во-первых, описан метод проверки логики программы, а, во-вторых, приведен обзор инструментальных средств (уже существующих и созданных в рамках данной работы), реализующих эту проверку с высокой степенью автоматизации.

Используемые методы и технологии

Перед изложением основной части работы опишем некоторые методы и технологии, на которых она основывается.

Методология проектирования по контракту [15, 16] применяется для разработки надежного программного обеспечения. В ней предлагается рассматривать составные части программного обеспечения как реализацию некоторой формальной спецификации, а не просто как исполняемый код. Основной идеей этой методологии является то, что класс и его клиенты заключают между собой «контракт». Клиент должен гарантировать выполнение некоторых условий перед обращением к классу. За это класс «обязуется» выполнить другие условия после завершения своей работы. Основным достоинством описываемой методологии является исполнимость контрактов – компилятор транслирует их вместе с исходным кодом. Поэтому любое нарушение контракта во время исполнения программы может быть сразу обнаружено.

Java Modelling Language (JML) [17] – язык спецификации *Java*-модулей (интерфейсов и классов), описывающий как поведение, так и сигнатуру. *JML* основывается на идеях проектирования по контракту и спецификациях, основанных на моделях. Ключевыми элементами любой спецификации являются предусловия (`requires`), постусловия (`ensures`) и инварианты (`invariant`). В языке *JML* существует заимствованное из *Eiffel* (языка программирования, разработанного для поддержки технологии проектирования по контракту [15]) выражение `\old(E)`, которое эквивалентно значению выражения `E` в момент входа в тело метода. Изменение значения переменной можно ограничить с помощью ключевого слова `constraint`, выражения `\old()` и логических конструкций, таких как импликация или эквивалентность. Спецификации записываются между символами `/*@ @*/` или после `//@`. Таким образом, стандартный *Java*-компилятор считает их комментариями и просто игнорирует. Они обрабатываются специально разработанными средствами.

Для поддержки *JML* было разработано большое число инструментов, обладающих различной функциональностью [18]. Наиболее простым способом проверки соответствия кода спецификации является *runtime*-проверка – компилятор *JML* (*jmlc*) запускает аннотированный *Java*-код и динамически проверяет соответствие типов и нарушение спецификации. Средства статической проверки (*ESC/Java2*) и формальной верификации (*KeY*, *Loop*, *JACK*) не требуют запуска приложения и способны решать значительно более сложные задачи, но с помощью пользователя и ценой некоторых допущений.

Несмотря на то, что *JML* может применяться для аннотирования произвольных *Java*-приложений, на данный момент основной сферой его применения считается *Java Card* [18].

Постановка задачи и схема предлагаемого решения

Решение проблемы создание корректных *Java Card*-программ в рамках использования автоматного подхода требует решения ряда подзадач:

1. Расширить существующие технологии генерации кода для создания скелета *Java Card*-приложений из их автоматного описания;
2. Разработать метод, позволяющий генерировать формальную спецификацию к коду, покрывающую наибольшее число свойств модели;
3. Исследовать существующие средства для работы с *JML* с целью выявления подходящих для решения задач, связанных с верификацией кода.

Общая схема предлагаемого подхода изображена на рис. 1.

Проблему предлагается решать в несколько этапов. На первом из них из автоматной модели генерируются скелет исходного кода, описывающий логику приложения, с заглушками для методов, реализующих входные и выходные воздействия (решение подзадачи 1). Помимо этого, генерируется формальная спецификация приложения на

языке *JML* (решение подзадачи 2). Далее вручную реализуются методы входных и выходных воздействий и, возможно, дополняется спецификация приложения. Наконец, при помощи специальных средств проверяется соответствие исходного кода или *byte*-кода спецификации (решение подзадачи 3). При этом предупреждения о возможных ошибках выдаются пользователю.

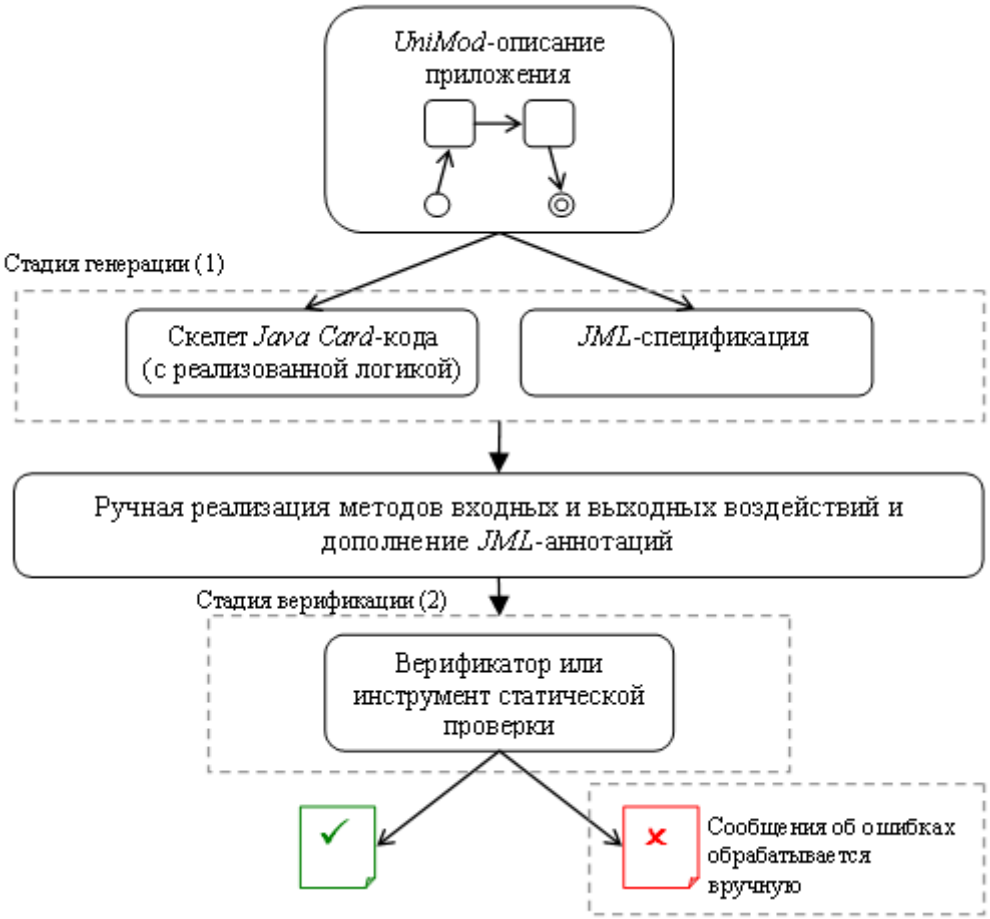


Рис. 1. Схема предлагаемого подхода

Автоматный подход для создания и верификации приложений для смарт-карт

В работе [19] предлагается разделение систем на преобразующие и реактивные. Преобразующая система обрабатывает входную информацию и выдает ответ на ее основе. Реактивная система должна постоянно реагировать на поступающие воздействия. При этом она должна не вычислять какую-либо конкретную функцию, а поддерживать взаимодействие с внешней средой. Реактивная система хранит свое текущее внутреннее состояние, а ее ответ среде и изменение состояния полностью зависят от входного воздействия и текущего состояния.

В настоящей работе для решения описанной выше проблемы используется автоматное программирование, предложенное в работах [3, 20]. Оно является разновидность синхронного программирования [21], которое считается одним из основных подходов при создании приложений для встроенных и реактивных систем.

Согласно парадигме автоматного программирования [22], программа рассматривается как система автоматизированных объектов управления. При использовании этой парадигмы при проектировании программы выделяются поставщики событий, система

управления, состоящая в общем случае из взаимодействующих конечных автоматов, и объектов управления.

Рассмотрим особенности платформы *Java Card* как реактивной системы [19]. Взаимодействие со смарт-картой осуществляется при помощи устройств считывания карт [1]. Это устройство подает питание на карту и создает канал связи между картой и компьютером или терминалом, на котором установлено хост-приложение. Канал связи – полудуплексный, в нем используется модель «главный – подчиненный». Смарт-карта всегда является подчиненным, а хост-приложение – главным. Это означает, что смарт-карта всегда ждет команды от хост-приложения. Последовательность команд поступает от хост-приложения через устройство считывания карт в *Java Card Runtime Environment (JCRE)*. После этого команды передаются выбранному апплету. Апплет обрабатывает команду и посылает ответ в обратном направлении.

Таким образом, можно трактовать взаимодействие между хост-приложением и смарт-картой как событийное. Так как могут быть выделены поставщики событий (хост-приложения) и объект управления (смарт-карта), то автоматное программирование может быть эффективно использовано для создания надежных Java Card-программ (рис. 2).

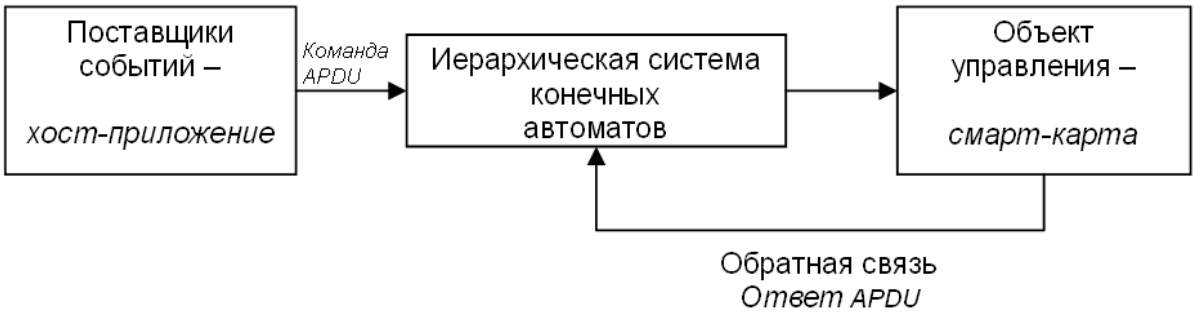


Рис. 2. Интерпретация автоматного подхода для платформы *Java Card*

Применение автоматного подхода не освобождает программиста от написания кода, так как генерируется лишь та часть функциональности, которая реализует логику системы. При этом предполагается ручное создание кода, реализующего входные и выходные воздействия. Проблема заключается в отсутствии средств для проверки соответствия кода его модели и проверки методов, реализующих входные и выходные воздействия. Предлагаемым выходом из данной ситуации является применение контрактов. Контракты также описывают взаимодействие различных частей программы, но делают это на более низком уровне абстракции. Если наряду с генераций исходного кода генерировать и спецификацию, изоморфную модели, то, несмотря на понижение уровня абстракции, сохранится возможность верификации. Следовательно, применение *JML* позволяет устранить разрыв между моделью и конкретной реализацией. Тем самым удается избежать ситуации, когда ограниченность модели является причиной неожиданных ошибок в программе или, наоборот, ошибки в плохо построенной модели будут сигнализировать о несуществующих ошибках в программе. В случае верификации приложений с использованием *JML* нет необходимости строить формальную модель – сам аннотированный исходный код подается на вход верификаторам. Как отмечалось выше, технология *Model checking* ограничивается только проверкой логики приложения, что является ее основным недостатком. Сгенерированная спецификация к коду позволяют решать аналогичные задачи. Однако, если их дополнить вручную, можно будет проверить значительно больший объем свойств программы, таких как, например, поведение

методов, реализующих входные и выходные воздействия, их взаимосвязь, распространенные при программировании ошибки, такие как выхода индекса массива за допустимые границы, обращение к несуществующему месту в памяти и многое другое.

Генерация кода и спецификаций (практическая реализация)

Все *Java Card*-апплеты имеют стандартную структуру – они должны переопределять методы базового класса `Applet`. Для пояснения процесса генерации кода кратко опишем [1] каждый их этих методов в порядке их вызова *JCRE*. Для создания экземпляра апплета вызывается метод `install`. Он аналогичен методу `main` в стандартных *Java*-приложениях. По умолчанию этот метод всегда возвращает значение `true`, что означает, что апплет готов к работе. Когда выбранный апплет получает команду, она передается для обработки в метод `process`, который будет описан в дальнейшем. Для деактивации апплета *JCRE* вызывает метод `deselect`. По умолчанию этот метод является пустым. Используя стандартные варианты реализации перечисленных методов (за исключением метода `process`), их можно определить в шаблоне в качестве статического содержимого, а при необходимости доработать после генерации под требования конкретной задачи.

В предлагаемом подходе вся логика метода `process` описывается иерархической системой конечных автоматов, из которых генерируется скелет кода, реализующий эту логику. Автоматное программирование поддерживается *Switch*-технологией [23]. Поэтому метод `process` состоит из двух вложенных операторов `switch` – внешний из них используется для перебора состояний, а внутренний – для перебора команд. Если полученная команда допустима для текущего состояния, то вызывается соответствующий вспомогательный метод, в противном случае возбуждается исключительная ситуация, которая не изменяет состояния апплета. Для вспомогательных методов генерируются только методы-заглушки.

Согласно технологии автоматного программирования, переходы между состояниями помечаются булевыми выражениями, образованными из входных воздействий. При условии, что эти выражения не имеют побочных эффектов, они могут стать предусловиями для методов, вызываемых при входе в состояние. Специфицировать структуру графа переходов можно, используя ключевое слово `constraint` и выражение `\old()`. Эта спецификация представляет собой дизъюнкцию импликаций, описывающих входящие и исходящие переходы для каждого состояния. Более того, существует возможность гарантировать то, что апплет всегда находится в одном из заданных заранее состояний (при помощи инварианта класса).

Инструментальное средство *UniMod* позволяет сохранять описание автоматной модели в формате *XML*. Для генерации кода используется разработанная при выполнении настоящей работы программа-конвертор, которая использует шаблоны (также разработанные в рамках этой работы) и технологию *Apache Velocity*.

Методика создания корректных *Java Card*-приложений

На основании изложенного можно привести описание методология создания корректных *Java Card*-приложений с применением автоматного подхода.

1. Из спецификации и технического задания к проекту перейти к *UniMod*-описанию приложения. При этом в качестве поставщиков событий используются хост-приложение, а в качестве объекта управления – смарт-карта.
2. Верифицировать получившуюся модель методом *Model checking*, используя такие верификаторы, как, например, *Spin* или *Bogor*.

3. Стандартными средствами *UniMod* экспортировать *UniMod*-описание приложения в *XML*-формат.
4. Подать *XML*-описание приложения и шаблоны, созданные в рамках настоящей работы, на вход программе-конвертору, разработанной авторами для генерации *Java Card*-кода с *JML*-аннотациями.
5. Доработать вручную сгенерированный код и *JML*-спецификацию в следующих направлениях:
   - реализовать методы входных и выходных воздействий (стандартный шаг в рамках автоматного подхода),
   - дополнить *JML*-аннотации (в основном для методов входных и выходных воздействий).
6. Верификация может проводиться по исходному коду (7) или по *byte*-коду (8).
7. В зависимости от решаемой задачи подать аннотированный код на вход одному из существующих инструментов для проверки исходного кода:
   - *Runtime*-проверка – *jmlc*,
   - статическая проверка – *ESC/Java2*,
   - верификация – *KeY*, *LOOP*, *JACK*.
8. Скомпилировать код, транслировать спецификацию в *byte*-код и осуществить его проверку с помощью соответствующих инструментов.
9. Вручную обработать сообщения об ошибках.

Пример

Приведем пример *Java Card*-апплета, иллюстрирующего предложенный подход. Схема связей приложения изображена на рис. 3, а (пункт методики 1). Поставщики событий и объект управления связаны между собой с помощью конечного автомата, граф переходов которого приведен на рис. 3, б. Эти диаграммы построены с помощью инструментального средства *UniMod*.

Апплет функционирует следующим образом. После стадии инициализации (состояние *Applet initialization* и выходное воздействие `o1.z1`) необходимо проверить *PIN*-код (событие `e1`), переходя в соответствующее состояние. Если в этом состоянии код верен (событие `e3`), то апплет переходит в состояние *Do something*, в котором обнуляется счетчик неудачных попыток (входная переменная `x1`) проверки *PIN*-кода (выходное воздействие `o1.z3`) и производится какое-либо действие (выходное воздействие `o1.z4`). Если код неверен (событие `e2`) и число неудачных попыток проверки не превосходит трех (булево выражение `o1.x1<=3`), то состояние автомата не изменяется, но увеличивается счетчик попыток (выходное воздействие `o1.z2`). Наконец, если код неверен (событие `e2`), а число неудачных попыток превосходит три (булево выражение `o1.x1>3`), то автомат переходит в состояние *SIM card is locked*, послав сообщение об этом хост-приложению (выходное воздействие `o1.z5`). В этом примере принят ряд упрощений. Во-первых, проверка *PIN*-кода в действительности осуществляется значительно сложнее, а во-вторых, содержательная часть работы приложения не описана – предполагается, что она выполняется в состоянии *Do something*. Однако этот пример может быть использован в любом приложении, так как проверка *PIN*-кода имеет ключевое значение.

Рассмотрим фрагмент метода `process` (листинг 1) и формальную спецификацию приложения (*Java Card*-код с *JML*-аннотациями), полученную при помощи программы-конвертора (пункт методики 4). Пусть целочисленная переменная `state` соответствует состоянию автомата в текущий момент. Тогда инвариант состояний (листинг 2) позво-

ляет гарантировать то, что апплет всегда будет находиться в одном из предопределенных графом переходов состояний.

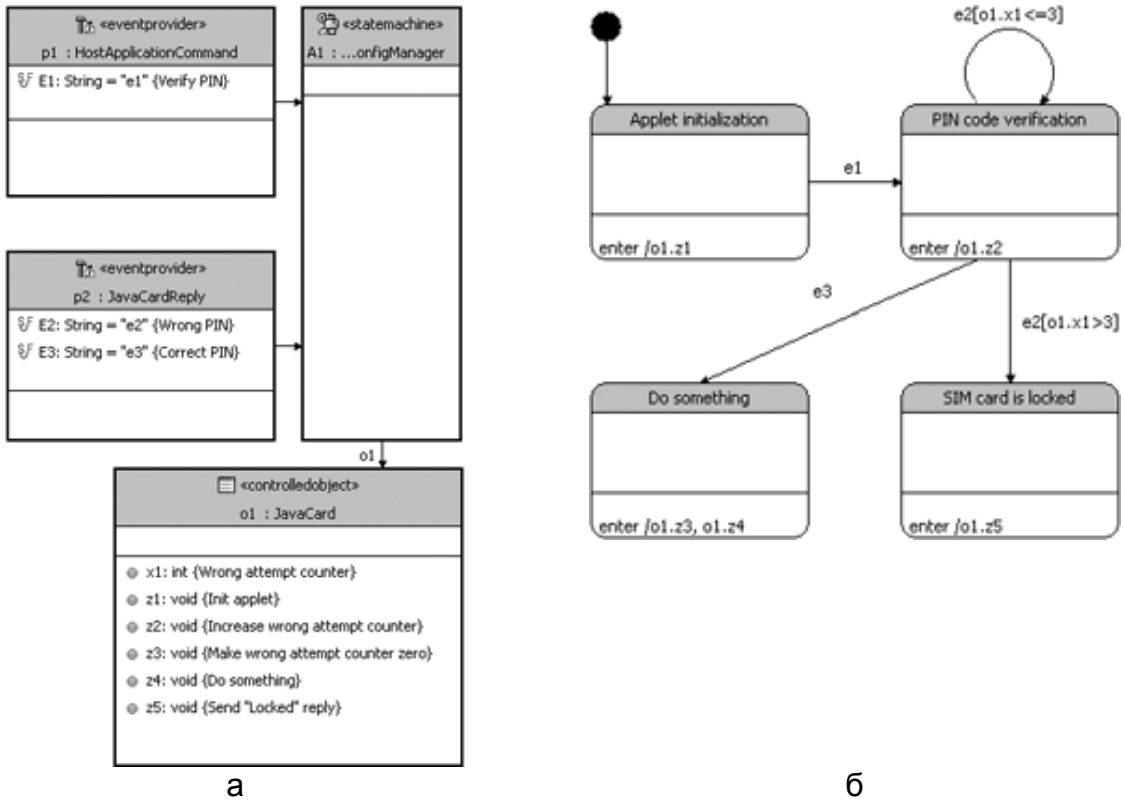


Рис. 3. Пример: (а) схема связей, (б) граф переходов

Листинг 1. Метод `process` (фрагмент)

```
//...

static final byte APPLET_INITIALIZATION = 0;
static final byte PIN_CODE_VERIFICATION = 1;
static final byte DO_SOMETHING = 2;
static final byte SIM_CARD_IS_LOCKED = 3;

//...

public void process(APDU apdu) throws ISOException {
 byte ins = apdu.getBuffer()[ISO7816.OFFSET_INS];
 switch(state) {
 case APPLET_INITIALIZATION:
 switch(ins) {
 //...
 default: ISOException.throwIt
 (ISO7816.SW_CONDITIONS_NOT_SATISFIED)
 } break;
 //...
 default: ISOException.throwIt
 (ISO7816.SW_CONDITIONS_NOT_SATISFIED)
}
```

Листинг 2. Инвариант состояний

```
/*@ invariant
 @ (state == APPLET_INITIALIZATION) ||
 @ (state == PIN_CODE_VERIFICATION) ||
 @ (state == DO_SOMETHING) ||
 @ (state == SIM_CARD_IS_LOCKED);
@*/
```

Описание графа переходов автомата, как ограничения на возможные переходы между состояниями, приводится в листинге 3.

Листинг 3. Ограничения на возможные переходы между состояниями

```
/*@ constraint
((state == APPLET_INITIALIZATION) ==>
(\old(state) == APPLET_INITIALIZATION)) &&

((state == PIN_CODE_VERIFICATION) ==>
((\old(state) == APPLET_INITIALIZATION) ||
(\old(state) == PIN_CODE_VERIFICATION))) &&

((state == DO_SOMETHING) ==>
((\old(state) == PIN_CODE_VERIFICATION) ||
(\old(state) == DO_SOMETHING))) &&

((state == SIM_CARDS_IS_LOCKED) ==>
((\old(state) == PIN_CODE_VERIFICATION) ||
(\old(state) == SIM_CARDS_IS_LOCKED))) &&

((\old(state) == APPLET_INITIALIZATION) ==>
((state == PIN_CODE_VERIFICATION) ||
(state == APPLET_INITIALIZATION))) &&

((\old(state) == PIN_CODE_VERIFICATION) ==>
((state == PIN_CODE_VERIFICATION) ||
(state == DO_SOMETHING) ||
(state == SIM_CARDS_IS_LOCKED))) &&

((\old(state) == DO_SOMETHING) ==>
(state == DO_SOMETHING)) &&

((\old(state) == SIM_CARDS_IS_LOCKED) ==>
(state == SIM_CARDS_IS_LOCKED));
@*/
```

Рассмотрим некоторые вопросы верификации полученного приложения (пункт методики 6). Поскольку сгенерированный код не обладает полной функциональностью, то сообщение об ошибке можно отследить, создав ее искусственно, например, дописав недопустимое значение для переменной `state` в конструкторе апплета (листинг 4).

В данном случае достаточно применить простейший инструмент динамической проверки (пункт методики 7, а). Запустим инструмент *jmlc* (листинг 5) и убедимся, что он находит эту ошибку (пункт методики 9).

Листинг 4. Искусственно созданная ошибка в программе

```
public PINApplet() {
 state = (byte) 5;
}
```

Листинг 5. Сообщение инструмента jmlc (фрагмент)

```
parsing ..\..\..\PinApplet.java
parsing ..\specs\javacard\framework\APDU.jml
parsing ..\specs\javacard\framework\Applet.jml
parsing ..\specs\javacard\framework\ISO7816.jml
parsing ..\specs\javacard\framework\ISOException.jml
...
typechecking ..\..\..\PinApplet.java
...
Exception in thread "main" org.jmlspecs.jmlrac.runtime.JMLInvariantError:
by method PinApplet.PinApplet@post<File "PinApplet.java", line 70,
character 17>
...
Done
```

Заключение

Технология автоматного программирования предоставляет методологию для разработки надежных встроенных и реактивных приложений. В настоящей работе предложено ее расширение для создания *Java Card*-приложений, дополнительной гарантией корректности которых служит применение проектирования по контракту.

В отличие от уже существующих исследований по верификации автоматных программ, в настоящей работе предлагается комбинировать различные подходы – применять верификацию исходного или *byte*-кода, а не только технологию *Model checking*. В данной работе предлагается начать верификацию приложения с верификации модели по технологии *Model checking*. Изоморфно перенеся модель на уровень исходного кода и дополнив спецификацию описанием поведения входных и выходных воздействий, можно гарантировать соответствие кода уже проверифицированной модели и в дополнение – корректность остальных методов, что невозможно было проверить при помощи технологии Model checking. При необходимости подобную проверку можно осуществить и на уровне byte-кода. Следовательно, верификация может проводиться на всех этапах разработки и внедрения приложения.

Литература

1. Чен Ж. Технология Java Card для смарт-карт. Архитектура и руководство программиста. – М.: Техносфера, 2008.
2. NASA LaRC Formal Methods Program: What is Formal Methods? – Режим доступа: http://shemesh.larc.nasa.gov/fm/fm-what.html
3. Шалыто А.А. Алгоритмизация и программирование для систем логического управления и «реактивных» систем // Автоматика и телемеханика. – 2001. – № 1. – С.3–39. – Режим доступа: http://is.ifmo.ru/download/arew.pdf
4. Poll E., van den Berg J., Jacobs B. Specification of the JavaCard API in JML / Proc. Fourth Smart Card Research and Advanced Application Conference. – P. 135–154.
5. Coglio A. An Approach to the Generation of High-Assurance Java Card Applets / Proc. 2nd NSA Conf. on High Confidence Software and Systems (HCSS'02). – 2002. – P. 69–77.
6. Hubbers E., Oostdijk M., Poll E. From finite state machines to provably correct Java Card applets / Proc. 18th IFIP Inform. Security Conf. – 2003. – P. 465–479.
7. Hubbers E., Oostdijk M. Generating JML specifications from UML state diagrams /Proc. Forum on specification and Design Languages (FDL'03). – 2003. – P. 263–273.
8. Гуров В.С., Мазин М.А., Шалыто А.А. UniMod – инструментальное средство для автоматного программирования // Научно-технический вестник СПбГУ ИТМО. – 2006.

– Вып. 30. Фундаментальные и прикладные исследования информационных систем и технологии. – С. 32–44. – Режим доступа: http://is.ifmo.ru/works/_instrsr.pdf
9. Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Промежуточный отчет по II этапу «Теоретические исследования поставленных перед НИР задач». – СПбГУ ИТМО, 2007. – Режим доступа: http://is.ifmo.ru/verification/_2007_02_report-verification.pdf
10. Вельдер С.Э., Шалыто А.А. Введение в верификацию автоматных программ на основе метода Model checking. // Научно-технический вестник СПбГУ ИТМО. – 2007. – Вып. 42. Фундаментальные и прикладные исследования информационных систем и технологий. – С. 33–48. – Режим доступа: http://vestnik.ifmo.ru/ntv/ntv_42.1.5.pdf
11. Непомнящий В.А., Рякин О.М. Прикладные методы верификации программ. – М.: Радио и связь, 1988.
12. Кузьмин Е.В., Соколов В.А. О верификации «автоматных» программ /Актуальные проблемы математики и информатики. / Сборник статей к 20-летию факультета ИВТ ЯрГУ им. П. Г. Демидова. – Ярославль: ЯрГУ, 2006. – С. 27–32. – Режим доступа: http://is.ifmo.ru/verification/_verautpr.pdf
13. Корнеев Г.А., Парфенов В.Г., Шалыто А.А. Верификация автоматных программ / Тезисы докладов Международной научной конференции, посвященной памяти профессора А.М. Богомолова. «Компьютерные науки и технологии». – Саратов: СГУ. 2007. – С. 66–69. – Режим доступа: http://is.ifmo.ru/verification/_KNIT-2007.pdf
14. Clarke E. M., Wing J. M. Formal methods: State of the Art and Future Directions //ACM Computing Surveys. –1996. – V. 4. – P. 626–643.
15. Мейер Б. Объектно-ориентированное конструирование программных систем. – М.: Русская редакция, 2005.
16. Leavens G. T., Cheon Y. Design by Contract with JML. – Режим доступа: http://www.jmlspecs.org/jmldbc.pdf
17. Leavens G.T., Baker A.L., Ruby C. Preliminary design of JML: A behavioral interface specification language for Java. – Iowa State Univ., Dept. of Comput. Sci. Tech. Rep. 98–06u, Apr. 2003.
18. Burdy L., et. al. An overview of JML tools and applications // Int. J. on Software Tools for Technology Transfer (STTT). – 7(3). – Jun. 2005. – P. 212–232.
19. Harel D., Pnueli A. On the Development of Reactive Systems. In Logic and Models of Concurrent Systems / NATO Advanced Study Institute on Logic and Models for Verification and Specification of Concurrent Systems. – 1985. – P. 477–498.
20. Шалыто А.А. Автоматное проектирование программ. Алгоритмизация и программирование задач логического управления // Известия Академии наук. Теория и системы управления. – 2000. – №6. – С. 63–81. – Режим доступа: http://is.ifmo.ru/download/app-aplu.pdf
21. Шопырин Д.Г., Шалыто А.А. Синхронное программирование // Информационно-управляющие системы. – 2004. – № 3. – С. 35–42. – Режим доступа: http://is.ifmo.ru/works/sync_prog_024.pdf
22. Шалыто А.А. Парадигма автоматного программирования / Международная научно-техническая мультиконференция «Проблемы информационных технологий и мехатроники». Материалы международной научно-технической конференции «Многопроцессорные вычисления и управляющие системы». – Таганрог: НИИВМС, 2007. – Т.1. – С. 191–194. – Режим доступа: http://is.ifmo.ru/works/_2007_09_27_shalyto.pdf
23. Шалыто А.А., Туккель Н.И. Switch-технология – автоматный подход к созданию программного обеспечения «реактивных» систем // Программирование. – 2001. – № 5. – С.45–62. – Режим доступа: http://is.ifmo.ru/works/switch/1/

УДК 004.4’242

ВЕРИФИКАЦИИ ВЗАИМОДЕЙСТВИЯ ЧАСТЕЙ РЕАКТИВНОЙ СИСТЕМЫ, РЕАЛИЗОВАННЫХ С ПОМОЩЬЮ АВТОМАТНОГО ПОДХОДА

С.Ю. Канжелев

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В работе рассматриваются вопросы верификации частей реактивной системы, каждая из которых реализована с помощью автоматного подхода. Рассматриваются проблемы, которые могут возникать в таких системах, а также спопобы их обнаружения для различных видов их семантической интерпретации.
Ключевые слова: верификация программ, реактивная система, автоматное программирование

Введение

Метод верификации систем, основанный на моделях (*Model checking*) [1], находит в последнее время все большее число сторонников. Этот метод предполагает построение модели программы и требований к этой модели на языке темпоральной логики. Актуальность этого метода для реактивных систем, реализованных с помощью автоматного подхода, обусловлена тем, что построения модели в этом случае не требуется – набор взаимодействующих автоматов уже является моделью.

Требования к модели, описанные на языке темпоральной логики, как правило, звучат так: «для любой «справедливой» истории выполнены какие-то утверждения». Например, «для любой справедливой истории» действие «положить трубку» всегда происходит после действия «поднять трубку». При этом «справедливой» историей называется такая история, для которой любое ожидаемое событие когда-нибудь произойдет. Однако на практике, наряду с невыполнением требований к модели, большой проблемой являются «непришедшие» события и «несправедливые» истории.

Если событие не приходит, программа может зависнуть в одном состоянии, ожидая это событие. Такая ситуация может возникнуть как из-за плохого проектирования, так и из-за внешних факторов. Но в любом случае необходим анализ диаграмм автоматов, реализующих логику программы, для выявления в ней ситуаций, в которых событие может не прийти.

В настоящей работе проводится классификация возможных причин непредвиденных остановок автоматов. Предлагается способ статической верификации этих ситуаций для случая взаимодействия частей программы, реализованных с помощью автоматов. При этом рассматриваются различные способы семантической интерпретации взаимодействия автоматов.

Причины ожидания событий

Существуют три возможных причин того, что событие, которого ожидает автомат, не приходит. Перечислим их.

1. *Поздняя подписка*: событие пришло раньше времени, когда его еще не ждали.
2. *Задержка*: событие придет, но позже.
3. Событие не придет никогда. Оно может не приходить по двум причинам.
    1. *Голодание*: по внешней причине.
    2. *Блокировка*: в ожидании действий пользователя.

Рассмотрим эти причины ожидания событий на примере автоматов, реализующих логику работы простейших телефона и автоматической телефонной станции (АТС). Диаграмма переходов и схема связей автомата «Телефон» представлены на рис. 1, а для

автомата «АТС» будут рассмотрены позже. Эти диаграммы упрощены и отличаются от реальных меньшим числом действий и переходов.

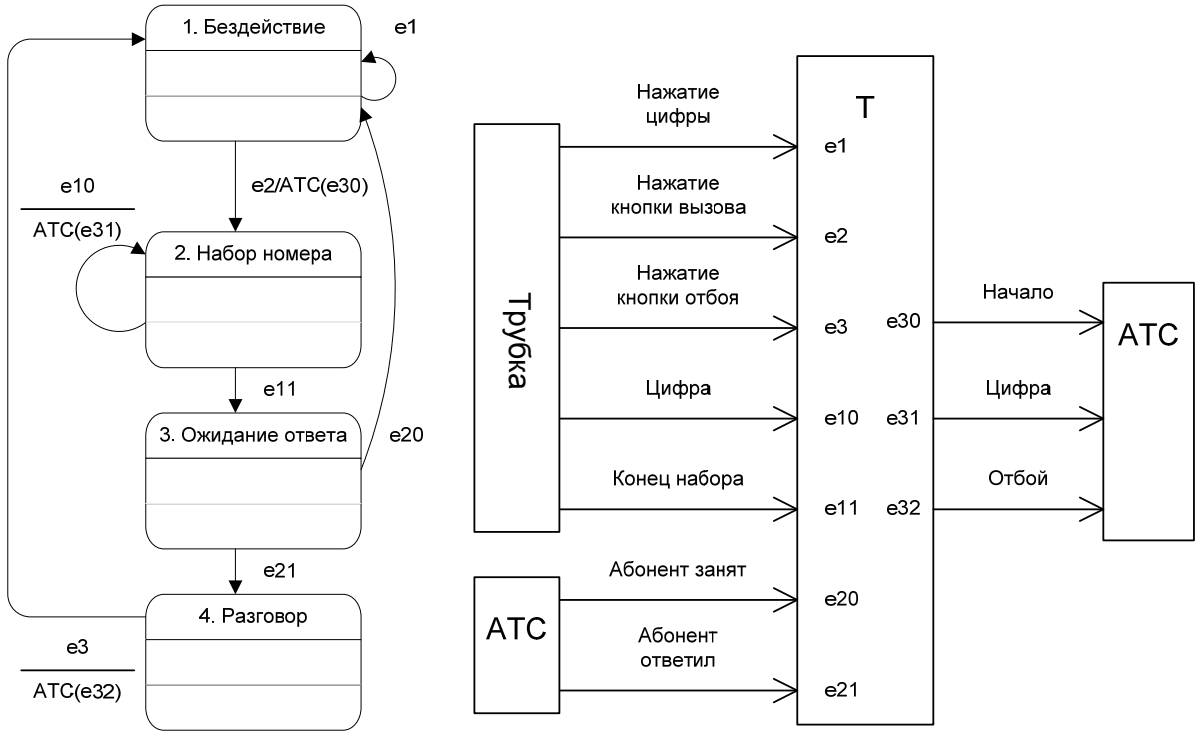


Рис. 1. Диаграмма переходов и схема связей автомата «Телефон»

Представленный автомат обладает ограниченной функциональностью, но в большинстве случаев работает корректно по следующему сценарию:

1. Абонент набирает номер (переходы по событию `e1`).

2. Абонент нажимает кнопку вызова (переход по событию `e2`). Телефон сообщает АТС, что трубка поднята (событие `АТС(e30)`).

3. Телефон начинает обрабатывать цифры, набранные абонентом в пункте 1 (переходы по событию `e10`). О каждой набранной цифре номера телефон сообщает АТС (событие `АТС(e31)`).

4. По окончании набора (событие `e11`) автомат переходит в состояние «Ожидание ответа».

5. В случае если вызываемый абонент занят, автомат возвращается в состояние бездействия (переход по событию `e20`).

6. Если вызываемый абонент ответил (переход по событию `e21`), автомат переходит в состояние «Разговор» и по его завершению (переход по событию `e3`) возвращается в состояние бездействия.

Автомат «Телефон» не во всех случаях работает корректно и может зависать в ожидании событий. Опишем случаи некорректной работы в соответствии с классификацией, данной ранее.

Поздняя подписка. Допустим, что абонент набрал больше цифр, чем необходимо. Если число лишних набранных цифр велико, то, в то время как телефон, находясь в состоянии 2, впустую набирает лишние цифры, вызываемый абонент может успеть ответить и положить трубку. Тогда по событию «Конец набора» телефон, оказавшись в состоянии 3, будет ожидать событие «Абонент ответил», которое уже произошло ранее.

Задержка. Простейший пример задержки – это продолжительное ожидание ответа вызываемого абонента в состоянии 3 «Ожидание ответа».

Голодание. Примером голодания может стать обрыв линий во время набора номера. В этом случае мы не дождемся ответа вызываемого абонента по не зависящей от нас причине.

Блокировка. Блокировка в данном примере возникает, когда набранных цифр недостаточно для вызова абонента. Вместо положенных 7 цифр абонент набрал 6. Набрав короткий номер, автомат окажется в состоянии ожидания ответа, из которого он может выйти, только получив ответ вызываемого абонента или сигнал «занято». В то же время вызываемый абонент не ответит до тех пор, пока не получит от автомата последнюю цифру своего номера.

Необходимо идентифицировать и исправлять перечисленные ошибки проектирования.

Верификация взаимодействия автоматных моделей

Многих перечисленных выше проблем можно избежать на этапе проектирования программы при условии, что взаимодействующие части этих программ реализованы с помощью автоматного подхода.

Одной из эффективных методик анализа взаимодействия автоматов является их перемножение. В книге [2] подробно рассмотрен пример такого анализа, в котором после перемножения трех автоматов – банка, магазина и клиента – автор указал на возможность купить в магазине товар, не заплатив за него деньги. На рис. 2, взятом из книги [2], изображены автоматы «Магазин», «Клиент», «Банк» и их произведение. Состояние `(2, c)` является некорректным, потому что в этом состоянии деньги клиентом не потрачены, а товар получен. Однако это состояние достижимо.

Применим аналогичный подход для выявления некорректных ситуаций зависания автоматов с помощью перемножения автоматов, реализующих логику работы телефона и АТС. На рис. 3 приведены диаграмма переходов и схема связей автомата «АТС». Рассмотрим, как будут взаимодействовать рассматриваемые автоматы.

Построим *автомат-произведение* для этих двух автоматов. Состояниями этого автомата являются пары состояний, первое из которых есть состояние телефона, а второе – состояние АТС. Отметим, что полученный автомат не должен иметь переходов по событиям `e20`, `e21`, `e30`, `e31` и `e32`, так как эти события используются лишь для обозначения взаимодействия автоматов, которое теперь будет закодировано в переходах автомата-произведения.

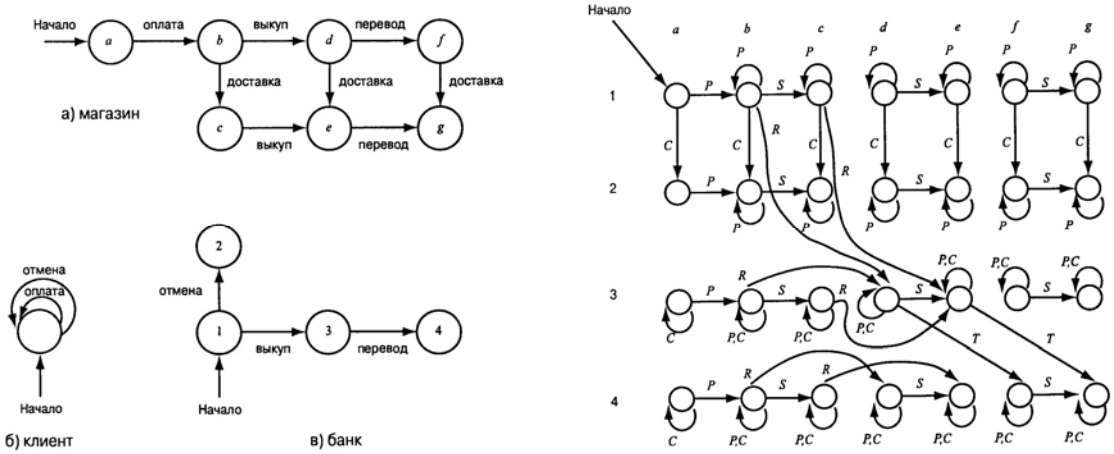


Рис. 2. Автоматы «Магазин», «Клиент» и «Банк» и их произведение [2]

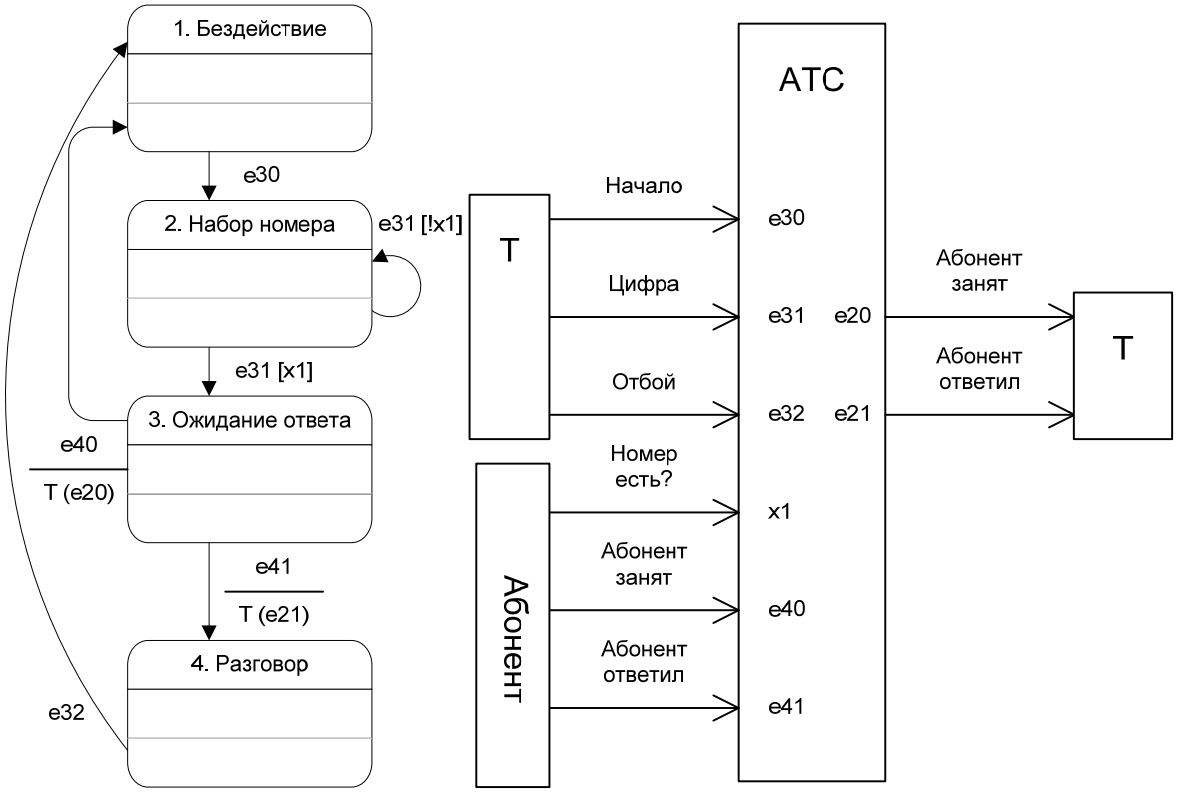


Рис. 3. Диаграмма переходов и схема связей автомата «АТС»

Чтобы правильно построить переходы в автомате-произведении, нужно проследить «параллельную» работу автоматов телефона и АТС. Каждый из двух компонентов автомата-произведения совершает, в зависимости от входных действий, различные переходы. Важно отметить, что если, получив на вход некоторое действие, ни один из этих двух автоматов не может совершить переход по внешнему событию, то автомат-произведение «умирает», поскольку также не может перейти ни в какое состояние.

Правило переходов из одного состояния в другое выглядит следующим образом. Пусть автомат-произведение находится в состоянии `(i, j)`. Это состояние соответствует ситуации, когда телефон находится в состоянии `i`, а АТС – в состоянии `j`. Пусть `e1` означает одно из входных действий. Допустим, автомат телефона или АТС имеет переход из текущего состояния по событию `e1`, и он ведет в состояние `i1` (которое может совпадать с `i`, если автомат, получив на вход `e1`, остается в том же состоянии). Из списка действий, выполняемых на ребре, выбираются действия посылки событий (пусть это будет `e2`) автомату АТС. Затем, если у автомата телефона есть дуга с меткой `e2`, ведущая в некоторое состояние `j1`, рассматриваются действия на переходе с меткой `e2`. Одно из этих действий может быть действием посылки события обратно телефону. Таким образом, получается последовательность `e1/ATC(e2)/T(e3)/…en`, которая в конечном итоге приводит к состояниям `in` и `jn`. Переход из состояния `(i, j)`в состояние `(in, jn)` по событию `e1` добавляется в автомат-произведение.

В процессе построения автомата-произведения могли получиться ошибки двух типов – состояния, из которых нет выхода и события, посылаемые автомату впустую. На рис. 4 приведено произведение автоматов. Вопросительными знаками отмечены получившиеся терминальные состояния и переходы, действия на которых никто не обрабатывает.

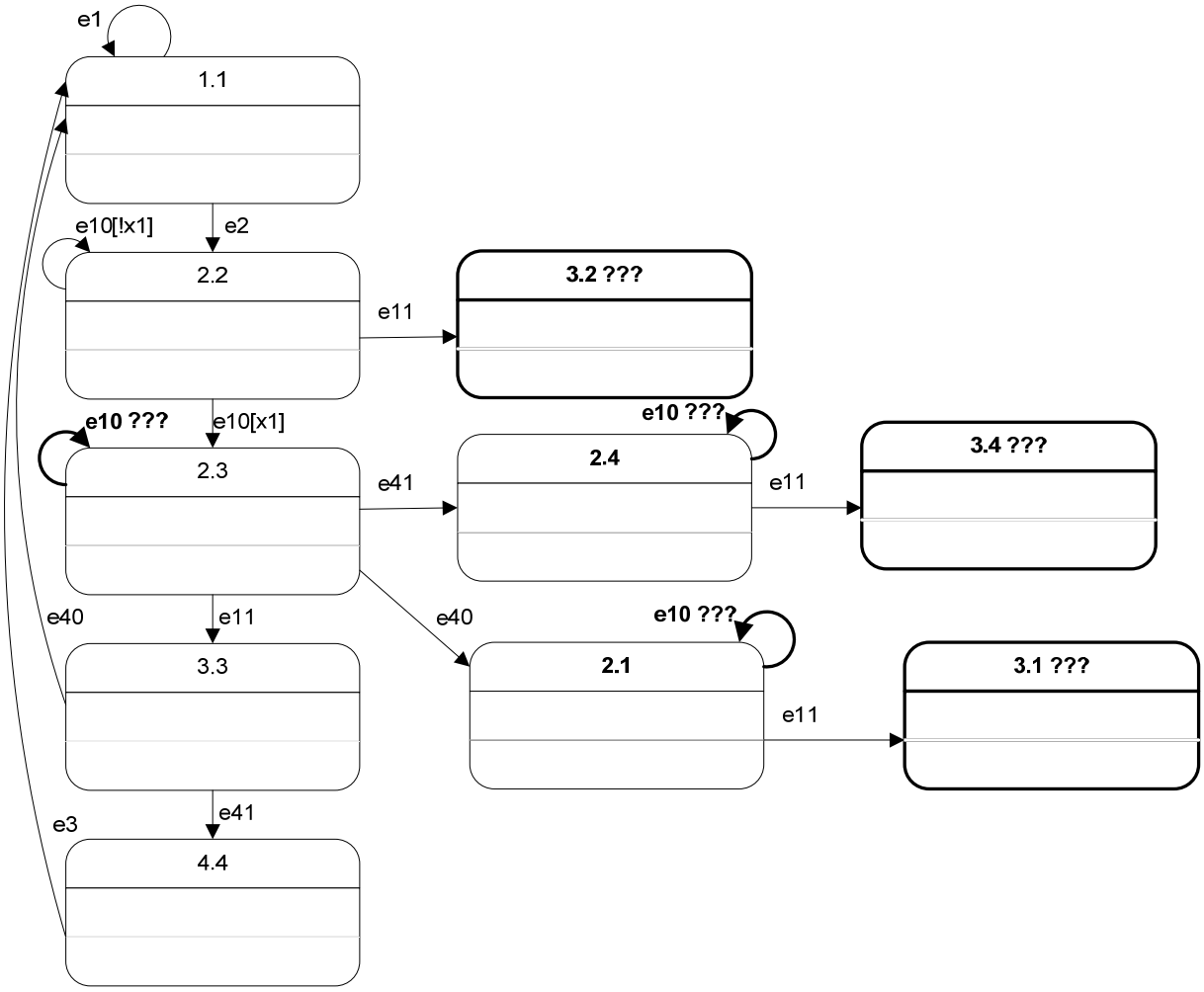


Рис. 4. Произведение автоматов «АТС» и «Телефон»

Полученное произведение автоматов можнет быть использовано как новый автомат, описывающий логику работы сразу двух устройств. У полученного автомата могут быть ошибки двух типов: ребра, посылающие события, которые не могли быть обработаны, и висячие вершины, не являющиеся терминальными состояниями. Посылка событий, которые не могут быть обработаны, говорит о том, что есть вероятность, что произошла поздняя подписка на событие либо событие уже неинтересно. Висячие вершины, не являющиеся терминальными, могут появляться по двум причинам – если произошла поздняя подписка (на рисунке – состояние `(3, 1)`) или блокировка (состояние `(3, 4)`).

Чтобы исправить эти ошибки, можно изменять исходные автоматы «Телефон» и «АТС» и с каждым изменением получать новый автомат-произведение, который необходимо проверять на наличие ошибок. Однако этот способ достаточно трудоемок. Проще и быстрее исправлять ошибки прямо на получившемся автомате-произведении, не рассматривая исходные автоматы. Этот подход описан в книге [3]. Он хорошо подходит для реализации больших систем сложных автоматов. Сначала можно спроектировать работу каждого автомата по отдельности, а затем, вычислив автомат-произведение, работать уже с одним автоматом, описывающим логику всей системы в целом. В случае телефона и АТС произведение автоматов можно рассматривать так, как если бы АТС непосредственно реагировала на нажатие кнопок (в этом случае мож-

но назвать телефон «тонким клиентом») или телефон умел общаться с базой абонентов и без помощи дополнительных систем соединялся бы с конкретным номером («толстый клиент»).

.

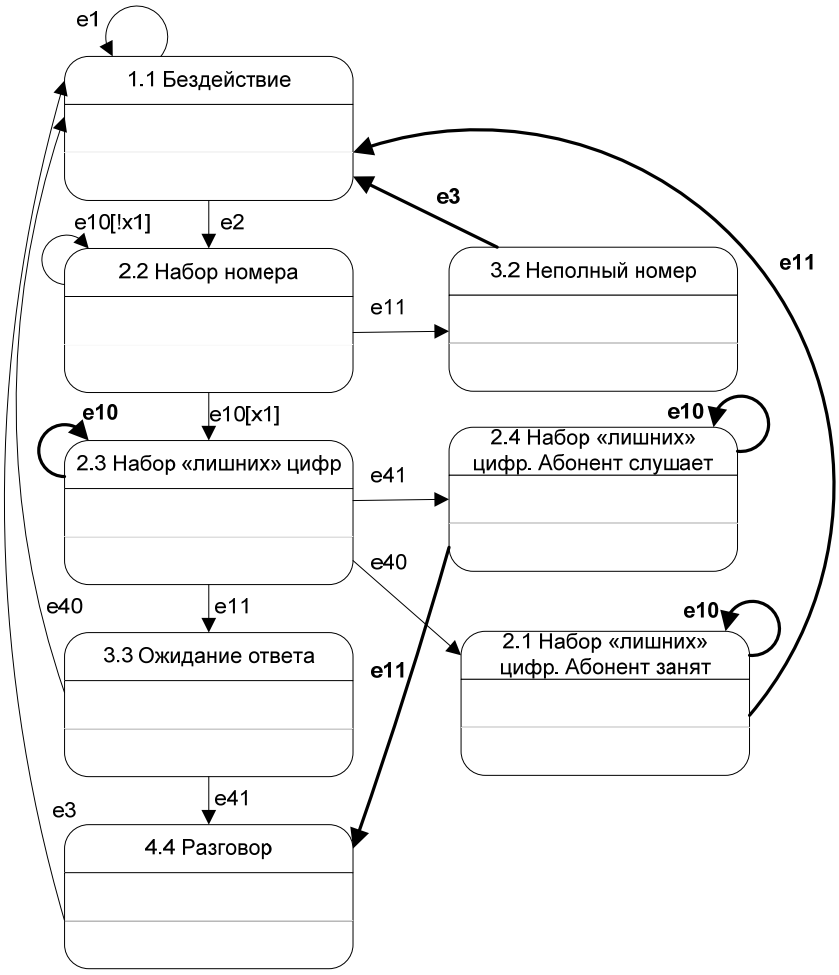


Рис. 5. Исправленное произведение автоматов «АТС» и «Телефон»

Рассмотрим теперь, как можно исправить ошибки в автомате-произведении. Чтобы исключить висячие состояния, не являющиеся терминальными, необходимо все ребра, идущие к этим состояниям, перевести в другие, более подходящие по смыслу состояния. В описанном примере для того, чтобы исключить висячее состояние `(3, 4)`, необходимо ребро с событием `e11` («Трубка поднята») из состояния `(2, 4)` пустить в состояние `(4, 4)`, минуя висячую вершину. А из вершины `(3, 2)` необходимо добавить ребро по событию `e3` («Нажатие кнопки отбоя») в начальное состояние `(1, 1)`.

Ребра, обработчик которых содержит посылку событий и которые не могут быть обработаны, в представленном случае необходимо просто оставить как есть.

В итоге получается исправленный автомат, описывающий логику всей системы в целом. Он представлен на рис. 5. Жирными линиями на рисунке отмечены ребра, перенаправленные в другие состояния. Состояниям были даны имена в соответствии с логикой работы системы

Важным вопросом при рассмотрении автомата-произведения как нового автомата является сложность автомата. Критерием его сложности может быть, например, коли-

чество вершин. Теоретически число вершин равно произведению числда вершин всех автоматов. На практике рассматриваются только достижимые состояния, что часто приводит к сокращению автомата.

В рассмотренном случае перемножения автоматов телефона и АТС получилось 13 вершин вместо 16. Из них действительно имели смысл и рассматриваются в исправленном автомате только 8.

Однако стоит отметить, что при добавлении всего одного ребра, как показано на рис. 6 автомат-произведение увеличивается на 2 вершины, и их количество уже становится близким к максимуму.

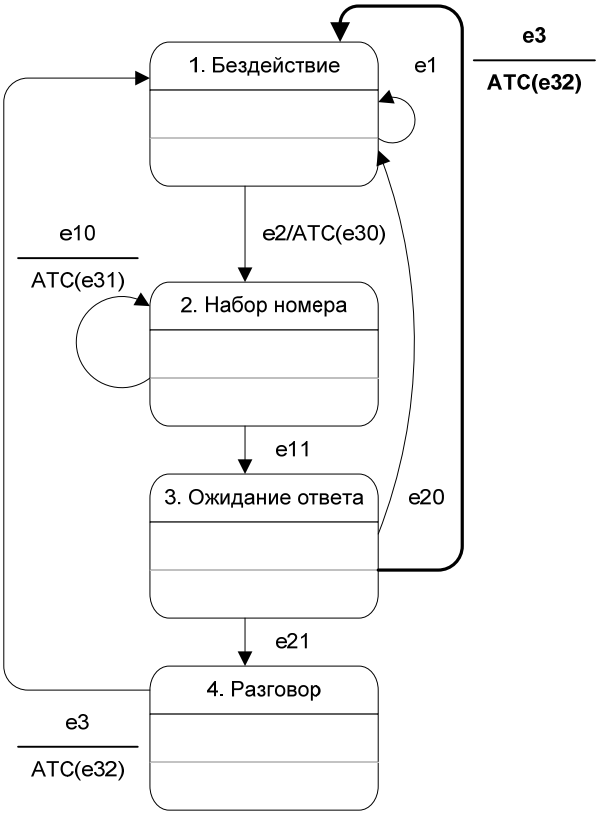


Рис. 6. Измененный граф переходов автомата «Телефон»

Автомат-произведение измененного автомата «Телефон» и автомата «АТС» показан на рис. 7. Отметим, что добавление одного ребра в автомат «Телефон», хоть и уменьшает число висячих нетерминальных состояний, но порождает неопределенность, связанную с тем, набирает ли АТС новый номер в состоянии `(2, 2)` или продолжает набор после перехода из состояния `(1, 2)`.

Проблема многопоточности при верификации автоматов

Построенный в предыдущем разделе автомат-произведение находит ошибки, допущенные на этапе проектирования автоматов. Для подтверждения этого факта рассматриваемые автоматы были реализованы с помощью технологии *Windows Workflow Foundation (WWF)*. Этим автоматам подавались на вход следующие последовательности событий с различными задержками между событиями (звездочка * обозначает произвольное количество событий):

```
24. e1, e2, (e10)*, e11
```

4. `x1=false`
5. `x1=true, e40`
6. `x1=true, e41, e3`

При возникновении исключительных ситуаций, когда событие не ожидается, и при попадании в известные терминальные состояния программа продолжала тестирование сначала.

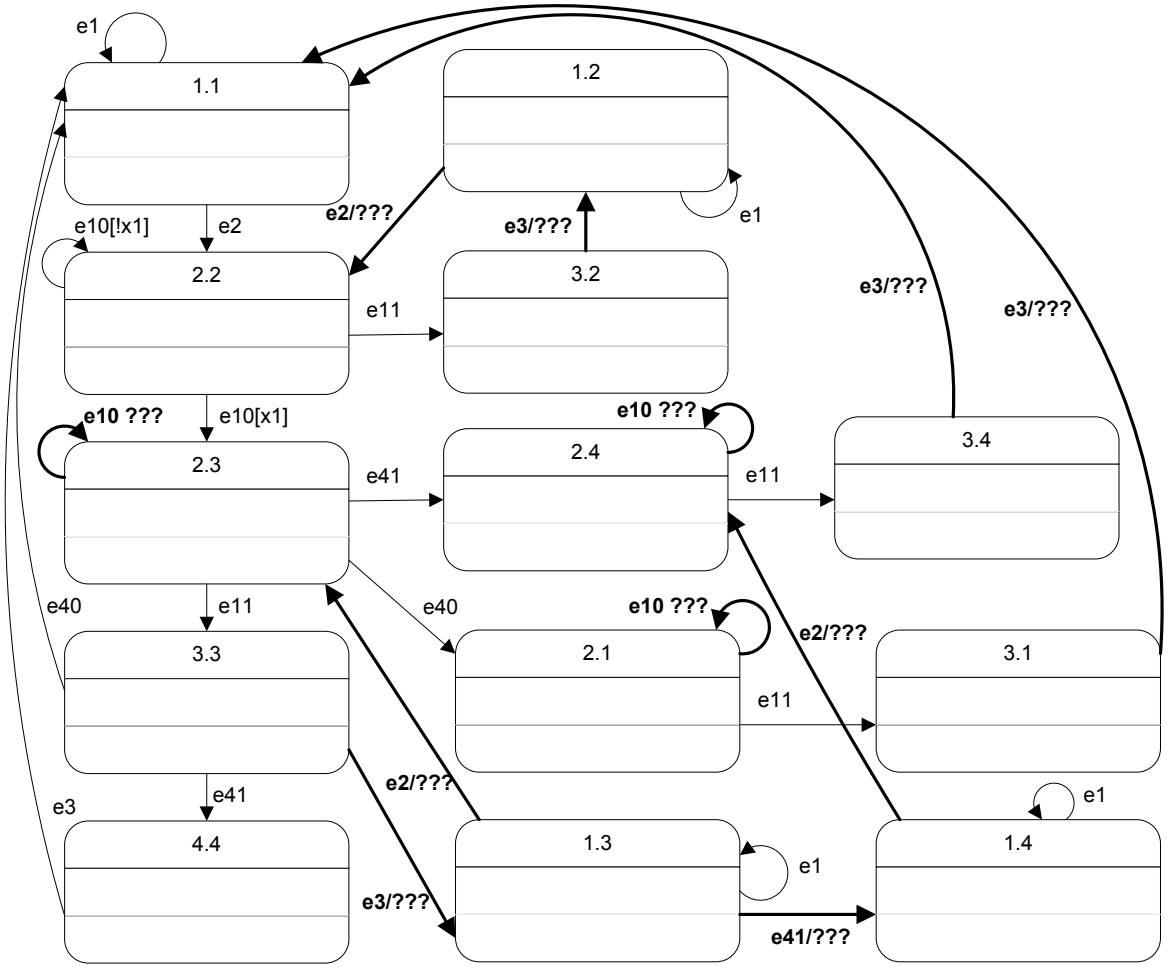


Рис. 7. Произведение измененного автомата «Телефон» и автомата «АТС»

Полученный тест подтвердил, что мы нашли все возможные исключительные состояния. Однако при небольшом изменении автомата, описывающего состояние телефона (рис. 6) и набора подаваемых на вход последовательностей, были найдены новые, не выявленные автоматом-произведением неожидаемые события – например, событие `e30` в состоянии `(1, 2)`. Рассмотрим более подробно причины возникновения таких событий.

При построении автомата-произведения был использован алгоритм, аналогичный описанному в книге [2]. Однако этот алгоритм основывается на предположении, что за один шаг будут обработаны все ребра всех автоматов, переход по которым вызывает событие, инициировавшее этот шаг.

Однако это предположение неверно для реактивных систем, использующих другую семантику взаимодействия частей автоматных программ. Действительно, в нашем случае автомат, реализующий АТС, может реагировать на события очень медленно, например, в силу большой загрузки. В этом случае абонент успеет набрать номер, нажать кнопку вызова и, не дождавшись никакого ответа, нажать кнопку отбоя. В этом случае у АТС «на-

копится» список событий `e30`, `(e31)*`, `e32`, который будет постепенно обрабатываться, и может так случиться, что, когда АТС закончит обработку этого списка событий в состоянии `2`, телефон заново поднимет трубку и пришлет событие `e30`.

Отметим, что в случае корректной реализации телефона и АТС, когда нет непредвиденных событий, такое поведение даже более предпочтительно, чем реализация системы в виде автомата-произведения, представленного на рис. 5. В этом случае при большой загрузке АТС у телефона нет необходимости дожидаться, когда АТС обработает события, посланные ей, тогда как автомат-произведение предполагает именно такое поведение.

Таким образом, при верификации реактивных систем, семантика которых сходна с семантикой WWF, построение автомата-произведения становится затруднительным, и необходимо применять другой способ их верификации.

Семантика взаимодействия реактивных систем, реализованных с помощью автоматов

Выше рассмотрены две семантики взаимодействия реактивных систем, реализованных с помощью конечных автоматов. В первом случае все автоматы обрабатываются в одном потоке, реагирующем на внешние события. Во втором случае каждый автомат представляет собой отдельный поток, и внешние события для него не отличаются от внутренних, необходимых только для обозначения взаимодействия автоматов. Рассмотрим подробнее каждую из этих двух семантик взаимодействия.

В реализации семантики взаимодействия автоматов, когда все автоматы исполняются в одном потоке, исполнение автоматов происходит по следующему алгоритму. При поступлении внешнего события оно ставится в очередь. Начинается его обработка. Сначала происходит поиск автомата, который ждет этого события. Начинается обработка действий на соответствующем ребре. Если требуется обработать посылку внутреннего события, автомат переводится в состояние, в которое ведет ребро, а затем вызывается обработчик внутреннего события.

Отметим ограничения, которые накладываются при такой семантике взаимодействия автоматов для того, чтобы гарантировать однозначность обработки событий.

1. Каждое событие может быть обработано только одним из автоматов.
2. Обработчик любого ребра может содержать только один вызов автомата.

Эта реализация взаимодействия автоматов позволяет использовать автомат-произведение для верификации взаимодействия автоматов.

Другая реализация взаимодействия автоматов заключается в том, что все события – внешние и внутренние обрабатываются в одной очереди. При этом автоматы реализуются как независимые потоки. В этом случае, как было показано выше, нельзя применять способ верификации взаимодействия автоматов, основанный на построении автомата-произведения. Также в этом случае некорректным будет проверка текущего состояния автомата (`y1 == 1`). Вместо таких проверок необходимо использовать события-нотификации о переходе в конкретное состояние.

Вопрос верификации взаимодействия реактивных систем с подобной семантикой взаимодействия рассмотрен в работе [5]. В этой работе предлагается использовать протоколы взаимодействия автоматов – для каждого автомата вычисляются возможные последовательности приходящих и исходящих событий. Затем происходит сравнение протоколов для обнаружения «нестыковок», когда протокол одного автомата не может быть обработан другим автоматом. Подобный подход был применен при тестировании автоматов, описанном выше, когда автоматам на вход подавались всевозможные последовательности событий в произвольном порядке.

Проблема атомарности обработки события в реактивных системах

Как отмечалось в работе [4], диаграмма переходов автомата проще всего реализуются в коде программы с помощью оператора `switch`. Отсюда и происходит одно из названий автоматного подхода – `SWITCH`-технология. Однако многие реализации автоматов не рассчитаны на работу в многопоточной среде. Это также, наряду с плохим проектированием, может привести к проблеме бесконечного ожидания события.

Вернемся к примеру на рис. 1. Допустим, автомат реализован в соответствии с классической нотацией [4]. В этом случае, если мы переходим по ребру из состояния 3 («Ожидание ответа») в состояние 4 («Разговор») и одновременно с этим к нам приходит событие «Кладем трубку», то мы можем начать обрабатывать его, подразумевая, что мы находимся еще в состоянии 3. Такого поведения легко избежать, используя блокировки, гарантирующие атомарность операции перехода из одного состояния в другое. Другой подход описан в работе [5] – для блокировки и корректной работы многопоточной системы использовалась очередь событий для каждого автомата. Также отсутствие такого поведения гарантируется при использовании инструментального средства *UniMod* [6] в режиме обработки событий Queue.

Стоит отметить, что в случае реализации автоматов с помощью библиотеки *Windows Workflow Foundation* [7] следует помнить об отсутствии гарантии такой атомарности. *Windows Workflow Foundation* поддерживают два способа подписки на события:

1. прямая подписка на события через стандартные средства языков *.NET*;
2. подписка через сервис `ExternalDataExchangeService`.

Только при использовании сервиса `ExternalDataExchangeService` гарантируется атомарность перехода по событию и корректная последовательность обрабатываемых событий.

Заключение

В работе приведена классификация возможных причин долгого ожидания событий автоматом. Приведен способ обнаружения такого поведения с помощью автомата-произведения, а также с помощью протоколов событий. Также указаны ограничения, накладываемые на реализацию автомата, при выполнении которых такая верификация возможна.

Литература

1. Вельдер С.Э., Шалыто А.А. Введение в верификацию автоматных программ на основе метода Model Checking. – Режим доступа: http://is.ifmo.ru/verification/modelchecking/
2. Хопкрофт Д., Мотвани Р., Ульман Д. Введение в теорию автоматов, языков и вычислений. – СПб.: Вильямс, 2002, 528 с.
3. Шалыто А.А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. – СПб.: Наука, 1998. – 628 с.
4. Шалыто А.А., Туккель Н.И. SWITCH-технология - автоматный подход к созданию программного обеспечения "реактивных" систем // Программирование. – 2001. – № 5. – Режим доступа: http://is.ifmo.ru/works/switch/1/.
5. Канжелев С.Ю., Шалыто А.А. Моделирование кнопочного телефона с использованием SWITCH-технологии. Вариант 2. – Режим доступа:http://is.ifmo.ru/projects/phone/.

6. Гуров В. С., Мазин М. А., Шалыто А. А. UniMod - программный пакет для разработки объектно-ориентированных приложений на основе автоматного подхода // Труды XI Всероссийской научно-методической конференции "Телематика-2004". – 2004. – Т.1 – Режим доступа:http://tm.ifmo.ru.
7. Эспозито Д. Cutting Edge: Windows Workflow Foundation // MSDN Magazine. – 2006. – № 5.

УДК 004.4’242

МЕТОД АВТОМАТИЧЕСКОЙ ДИНАМИЧЕСКОЙ ВЕРИФИКАЦИИ АВТОМАТНЫХ ПРОГРАММ

О.Г. Степанов

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В настоящей работе предлагается метод автоматической динамической верификации систем автоматов Мили, основанный на обходе альтернирующих автоматов. Описывается структура метода, правила построения протоколов и спецификации, а также выполняется анализ функциональных характеристик метода на основе разработанной реализации.

Ключевые слова: динамическая верификация, автоматное программирование

Введение

Актуальной является задача верификации программ – проверки их соответствия заданным свойствам [1]. Существует два подхода к верификации – статическая и динамическая [1]. Наиболее распространена статическая верификация на основе метода *Model Checking* – проверки свойств программ на их моделях. При использовании этого метода верифицируемую программу требуется представить в специальной форме – в виде модели Крипке, описывающей возможные изменения вычислительных состояний программы [1]. С этой моделью связана и основная проблема метода *Model Checking* – экспоненциальный рост размера модели Крипке при линейном росте размера программы. Эта проблема получила название «экспоненциальный взрыв».

Другой подход – это динамическая верификация, при использовании которой протоколы выполнения программы проверяются на соответствие заданной спецификации. Эта разновидность верификации применяется для проверки поведения программы во время выполнения, например, при отсутствии доступа к коду программы или при исследовании правильности взаимодействия со сторонними компонентами. Хотя верификация протокола конкретного запуска программы не может гарантировать выполнения заданных свойств при любых значениях входных параметров, при правильном подборе тестовых сценариев ее результаты могут оказаться достаточно точными. При этом трудоемкость динамической верификации не зависит от сложности верифицируемой программы, а только лишь от сложности проверяемой спецификации и размера протоколов.

Для верификации автоматных программ в настоящее время наиболее широко используемым является метод *Model checking* [2]. Однако исследования показали, что текущие реализации этого метода позволяют верифицировать лишь программы с небольшим числом автоматов, так как их число определяет размер состояния программы. Следовательно, размер модели Крипке системы автоматов экспоненциально зависит от числа автоматов в системе [2]. Динамическая верификация автоматных программ в настоящее время практически не изучена.

В настоящей работе предлагается метод динамической верификации автоматных программ, основанный на общем методе динамической верификации [3]. Этот метод основан на обходе альтернирующих автоматов, что позволяет верифицировать протоколы программ с трудоемкостью, линейно зависящей от размера протокола и числа подформул в спецификации.

Значительным ограничением динамической верификации программ является ненадежность этого подхода к верификации систем параллельных программ, так как последовательность передачи управления между программами может значительно изменяться от запуска к запуску. В данной работе рассматриваются системы автоматов Мили, в которых события обрабатываются по очереди. Это делает метод динамической верификации применимым для таких систем.

В первом разделе описывается существующий метод динамической верификации программ, а во втором – разработанный автором метод динамической верификации автоматных программ. При этом приводится структура метода, описываются схема построения протокола и выразительные возможности спецификации. В третьем разделе излагаются функциональные особенности и характеристики предложенного метода.

Динамическая верификация программ с использованием альтернирующих автоматов

Для методов как статической, так и динамической верификации в качестве языка спецификации используются языки темпоральной логики. Наиболее распространенными их них являются язык линейной темпоральной логики (*LTL*) и язык логики ветвящихся вычислений (*CTL*). Одним из наиболее распространенных языков темпоральной логики является *LTL*. Формулы этого языка построены на множестве *атомарных высказываний* (*Prop*) и замкнуты через применение булевых операторов, унарного темпорального оператора *N* («на следующем шаге») и бинарного темпорального оператора *U* («до тех пор, как») [4].

Моделью для *LTL*-формул является *вычисление* – функция $\pi : \omega \to 2^{\Pr op}$, которая задает значения истинности высказываний из множества *Prop* в каждый момент времени, задаваемый натуральным числом. Вычисление π в момент времени $i \in \omega$ удовлетворяет *LTL*-формуле φ (обозначается $\pi, i \rhd \varphi$) при выполнении следующих условий:

- $\pi, i \rhd p$ для $p \in Prop \Leftrightarrow p \in \pi(i)$;
- $\pi, i \rhd \xi \wedge \psi \Leftrightarrow (\pi, i \rhd \xi \wedge \pi, i \rhd \psi)$;
- $\pi, i \rhd \xi \vee \psi \Leftrightarrow (\pi, i \rhd \xi \vee \pi, i \rhd \psi)$;
- $\pi, i \rhd \overline{\varphi} \Leftrightarrow \overline{\pi, i \rhd \varphi}$;
- $\pi, i \rhd N\varphi \Leftrightarrow \pi, i+1 \rhd \varphi$;
- $\pi, i \rhd \xi U \psi \Leftrightarrow \exists j > i : \pi, j \rhd \psi, \forall k : i < k < j : \pi, k \rhd \xi$.

Говорят, что π *удовлетворяет* формуле φ (обозначается $\pi \rhd \varphi$), если и только если $\pi, 1 \rhd \varphi$.

Вычисление π является бесконечным словом на алфавите *Prop*. Таким образом, каждой *LTL*-формуле φ соответствует множество бесконечных слов, удовлетворяющих этой формуле. Это множество называется *языком формулы* φ и обозначается $L(\varphi)$.

Отметим, что для верификации спецификаций, заданных в виде *LTL*-формул на протоколах, требуется расширить определение *LTL* на конечные вычисления, в которых π задано на отрезке [1, *n*]. Это сделано в работе [3].

Для данного множества *X* определим *B+(X)* как множество положительных булевых формул над *X* (булевых формул, построенных из элементов *X*, которые соединены операторами $\vee$ и $\wedge$) и формул *истина* и *ложь*.

Альтернирующим автоматом Бюхи называется набор $A = \left(\Sigma, S, s^0, \rho, F\right)$ [5], где Σ – непустой конечный алфавит, S – непустое конечное множество состояний, $s^0 \in S$ – начальное состояние, F – множество *принимающих состояний*, а $\rho : S \times \Sigma \to B^+(S)$ – функция перехода.

Запуском альтернирующего автомата Бюхи на бесконечном слове $\omega = \{a_0, a_1, ...\}$ называется *S*-помеченное дерево *r* такое, что его корень помечен значением *s0* и справедливо следующее: если вершина дерева *x* глубиной *i* помечена значением *s* и $\rho(s, a_i) = \theta$, то эта вершина имеет *k* детей *x1,x2…xk*, где $k \le |S|$, и множество их пометок $\{r(x_1), r(x_2), ...r(x_k)\}$ обращает формулу θ в истину. Запуск называется *принимающим*, если на каждой его ветви принимающие состояния или переход $\rho(s, a_i)$ = *истина* встречаются бесконечно часто. Таким образом, каждому альтернирующему автомату *A* соответствует множество бесконечных слов, для которых существует принимающий запуск. Это множество называется *языком автомата A* и обозначается *L(A)*.

Можно построить аналог альтернирующего автомата Бюхи для конечных слов. Такой автомат будет называться *альтернирующим автоматом*.

В работе [6] показано, что для любой *LTL*-формулы φ можно построить альтернирующий автомат Бюхи *A* такой, что *L(A)=L(φ)*, причем число состояний автомата *A* линейно зависит от размера формулы φ.

Таким образом, для верификации соответствия протокола работы программы спецификации, заданной в виде *LTL*-формулы, достаточно по этой спецификации построить альтернирующий автомат и проверить, является ли данный протокол элементом языка полученного автомата. В работе [3] предложены три алгоритма построения принимающего запуска для данных альтернирующего автомата и протокола.

Первый алгоритм (*обход в глубину)* пытается построить принимающий запуск, обходя альтернирующий автомат рекурсивно в глубину из начального состояния. Этот алгоритм наиболее прост в реализации, но часто требует обхода протокола несколько раз. Например, для спецификации вида *GF*φ *(*где Fφ = *истина U* φ и $G\varphi = \overline{F\overline{\varphi}}$*)* «хвост» протокола будет повторно проанализирован на каждом шаге. Таким образом, для длинных протоколов алгоритм обхода в глубину работает неприемлемо медленно.

Второй алгоритм (*обход в ширину)* обходит автомат в ширину, поддерживая на каждом шаге все возможные комбинации записей протокола, которые могут являться элементами принимающего запуска в данный момент. Алгоритм обхода в глубину анализирует протокол лишь однажды, но вычислительное состояние алгоритма имеет размер, экспоненциально зависящий от размера спецификации. Для небольших формул множества состояний, допустимых в данный момент, невелики, но с ростом сложности формул размер этих множеств может представлять проблему. Экспоненциальный рост вычислительного состояния алгоритма вызван индетерминизмом формулы. Эту проблему можно решить, анализируя протокол от «хвоста к голове» [7].

Третий из предложенных в работе [3] алгоритмов, алгоритм обратного обхода, похож на обход в глубину, но вместо рекурсивных вызовов использует уже вычисленное для «хвоста» протокола состояние. На каждом шаге алгоритм поддерживает набор состояний альтернирующего автомата, допускаемых просмотренным «хвостом» протокола.

В следующей части статьи предлагается метод динамической верификации автоматных программ, основанный на описанном подходе. Также проанализирована эффективность работы описанных трех алгоритмов для динамической верификации автоматных программ.

Метод динамической верификации автоматных программ

В настоящее время для верификации программ, заданных в виде системы автоматов Мили, в основном используется статическая верификация – метод *Model Checking*. Однако существующие реализации этого метода позволяют верифицировать лишь относительно несложные системы автоматов (сотни состояний во всей системе) [2]. В этом разделе предлагается метод динамической верификации систем автоматов Мили. Этот метод включает в себя алгоритм построения протокола выполнения автоматной системы, а также семантику и правила вычисления значений предикатов, используемых при записи спецификаций.

Дадим краткое описание системы автоматов. Имеется набор конечных детерминированных автоматов Мили [8] с несколькими выходными воздействиями на ребрах. Для автомата задается набор событий, с которыми он может вызываться. Для каждого события определяется, может ли это событие поступать от источника событий или только от автомата.

Каждый переход может содержать:

- событие, при котором он происходит;
- условие, при котором он происходит. В условии в качестве атомарных утверждений можно использовать входные переменные *x1,x2,…,xk* и выражения вида *si,j* (оно истинно, если автомат *Ai* находится в состоянии *si,j*);
- последовательность действий. Она может содержать выходные воздействия *zi* и передачу управления другим автоматам *Ai(ej)*.

События обрабатываются последовательно – одно событие за один раз. В качестве входных переменных используются булевы переменные или состояния другого автомата. Переменные других типов необходимо преобразовывать в булевы.

На рис. 1 изображен пример простой системы автоматов Мили, которая может использоваться для управления грузовым лифтом. Автомат *A1* моделирует поведения лифта (*s1* – «Ожидание, двери закрыты», *s2* – «Движение», *s3* – «Ожидание, двери открыты»). Автомат *A2* моделирует лампу в кабине лифта (*s1* – «Свет выключен», *s2*– «Свет включен»). Когда автомат *A1* получает событие *e1* («Вызов»), он переходит в состояние *s2,* в котором он ожидает событие *e2* («Прибытие»). При получении этого события в автомате *A1* осуществляется переход в состояние *s3* и вызывается автомат *A2* с событием *e3,* включая тем самым свет. При получении события *e4* («Двери закрыты») автомат *A1* пересылает это событие автомату *A2,* который в ответ выключает свет. Конечно, это довольно простой пример, но он иллюстрирует взаимодействие основных компонентов автоматной системы.

Опишем метод динамической верификации такой системы автоматов. Он состоит из следующих простых шагов:

- запись отрицания верифицируемой спецификации в виде *LTL*-формул. Это преобразование производится вручную в соответствии с рекомендациями, изложенными в работе [9];
- автоматическое построение по полученным формулам альтернирующего автомата;
- запуск системы автоматов и автоматическое построение протокола ее работы;

- автоматический обход альтернирующего автомата в соответствии с полученным протоколом с помощью одного из алгоритмов, описанных в предыдущем разделе;
- автоматическое построение контрпримера в модели при обнаружении нарушения спецификации.

Заметим, что все шаги метода, кроме первого, выполняются автоматически, а первый шаг в том или ином виде присутствует во всех подходах к верификации автоматов по методу *Model Checking*. Таким образом, предлагаемый метод не увеличивает объем ручной работы по сравнению с другими существующими автоматическими методами.

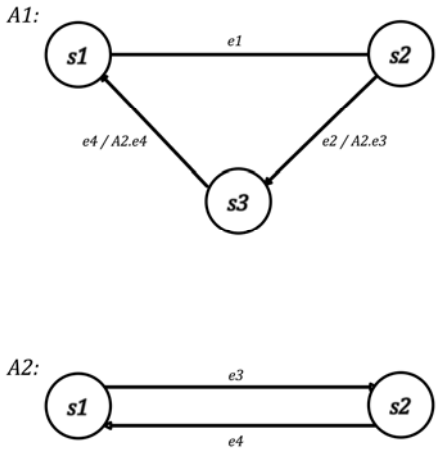


Рис. 1. Диаграммы состояний автоматов системы управления лифтом

В качестве примера приведем верификацию описанной выше автоматной системы управления лифтом. В качестве спецификации выберем условие «через некоторое время после вызова у лифта открываются двери». Для верификации системы автоматов Мили с использованием метода динамической верификации требуется определить:

- набор атомарных предикатов, которые разрешено использовать в *LTL*-формулах спецификации;
- способ построения протокола выполнения автомата;
- правила вычисления значений атомарных предикатов в записях протокола.

Начнем с описания набора предикатов, на которых можно основывать спецификации. Обработка одного события системой автоматов состоит из следующих шагов [2]:

- получение события;
- вычисление условий на переходах из текущего состояния;
- выполнение действий на переходе. При выполнении вызова другого автомата управление передается этому автомату с соответствующим событием, вызванный автомат совершает переход по этому событию и затем возвращает управление вызвавшему автомату;
- переход в новое состояние.

В соответствии с это схемой необходимо предоставить возможность установить факт совершения того или иного этапа обработки события. Таким образом, достаточным представляется следующий набор базовых предикатов:

- *ei,j* – автомат *Ai* обрабатывает событие *ej*;
- *xi* – при обработке текущего события значение входной переменной *xi* истинно;
- *zi* – выполняется выходное воздействие *zi*;
- *si,j* – *Ai* находится в состоянии *si,j*.

Предложенный набор атомарных предикатов покрывает описательные возможности, предложенные в работе [2].

Попробуем перевести спецификацию для системы управления лифтом («через некоторое время после вызова у лифта открываются двери») на язык формул, использующих предложенные предикаты. Она будет записана как $e1,1 \rightarrow \mathrm{F}\ s1,3$.

Далее определим структуру протокола, позволяющую однозначно вычислять значения предложенных атомарных предикатов в каждой записи. Эту структуру будем основывать на разбиении работы системы автоматов на промежуточные состояния, данной в работе [2]. Протоколом является некоторое конечное слово над конечным алфавитом. В предложенном методе алфавитом протокола системы автоматов Мили из n автоматов (по $S_i\Big|_{i=1}^{n}$ состояний в каждом) с m событиями, k входными переменными и l выходными воздействиями является множество со следующими элементами: $e_{1,1},\ldots,e_{n,m}, s_{1,S_1},\ldots,s_{n,S_n}, x_1,\ldots,x_k, \overline{x_1},\ldots,\overline{x_k}, z_1,\ldots,z_t$. В момент начала протоколирования в протокол делаются записи *si,j* о текущих состояниях автоматов (эту последовательность записей мы назовем *заголовком протокола*). Протоколирование может быть начато лишь при условии, что ни один автомат не производит обработку события. При получении входного события *ej* автоматом *Ai* в протокол делается запись *ei,j*. Затем происходит вычисление входных переменных, и для каждой переменной, значение которой *истина*, в протокол делается запись *xi*, а для каждой со значением *ложь* – $\overline{x_i}$. При выполнении выходного воздействия в протокол делается запись *zi*. Наконец, после обработки события автоматом *Ai* в протокол делается запись *si,j*, указывающая новое состояние автомата, даже если произошел переход по петле и состояние автомата при переходе не изменилось.

Остается определить значения описанных предикатов для каждой записи в протоколе. Назовем *секцией обработки события* автоматом A_i последовательность записей *ei,j,…,sk*. *Заголовком секции* назовем начальную подпоследовательность записей длины *1 + k*: *ei,j,…,xk*. Секции могут быть вложены. При этом вложенные друг в друга секции обязательно соответствуют разным автоматам [2]. Тогда предикат *ei,j* имеет значение *истина* во всех записях секции обработки события, начинающейся с записи *ei,j*. Предикат x_i имеет значение *истина* для всех записей секции, если в заголовке секции встречается запись *xk* и *ложь*, если встречается запись $\overline{x_k}$. Отметим, что в каждом заголовке для каждой входной переменной обязана присутствовать ровно одна из этих записей. При вызове вложенного автомата значения *xi*, построенные на основании заголовка секции обработки вложенного события, перекрывают предыдущие значения предикатов. Предикат *zi* имеет значение *истина* только в записи *zi*. Предикат *si,k* принимает значение *истина* в записи *si,k* и сохраняет это значение во всех записях протокола до следующей записи вида *si,k'*, не включая саму эту запись. Во всех остальных случаях предикаты имеют значение *ложь*.Например, при одиночном вызове лифта и прибытии лифта на этаж рассматриваемая система управления лифтом произведет действия, описываемые следующим протоколом (в квадратные скобки взяты секции обработки событий): *s1,1s2,1[e1,1s2][e1,2[e2,3s2,3]s1,3]*.

Отметим, что в результате разделения записей заголовка протокола (*ei,j,…,xk*) некоторые моменты времени, на которых производится обход альтернирующего автомата, могут быть представлены несколькими записями протокола. Так как всему заголовку каждой секции соответствует один момент времени, то для определения значений предикатов в этот момент времени требуется наличие информации обо всех записях заголовка. Эту особенность необходимо учитывать при реализации предложенного ме-

тода на практике, но, так как заголовок всегда имеет фиксированную длину, то при получении последней записи заголовка можно немедленно выяснить значения всех предикатов, описывающих состояние системы, в момент начала обработки события.

При обнаружении несоответствия записей протокола заданной спецификации требуется построить контрпример – путь в системе автоматов, который приводит к нарушению условий спецификации. При использовании предложенного метода для построения контрпримера достаточно взять заголовки секций протокола от начала до момента обнаружения нарушения спецификации. Эта последовательность записей в совокупности с заголовком протокола и даст путь в автоматной системе, являющийся контрпримером.

Функциональные особенности и характеристики метода

Для проверки возможности практического использования предложенного метода динамической верификации была осуществлена простейшая реализация алгоритмов построения и обхода альтернирующих автоматов. Реализация выполнена на языке *C#* и исполнялась в среде *Microsoft CLR 2.0 SP1* под операционной системой *Microsoft Windows Vista 64*. Код написан без применения специальных оптимизаций, применение которых может дополнительно ускорить работу верификатора.

Для тестирования в качестве примера использовалась две спецификациии. Одна из них – простая спецификация вида $Gx1 \rightarrow Fz1$: после получения события, при котором значение входной переменной *x1 – истина*, всегда выполняется выходное воздействие *z1*. Вторая – более сложная: $Gx1 \vee (Fz1 \wedge Fz2)$ соответствует утверждению «при получении любого события либо значение переменной *x1 – истина,* либо в некоторый момент времени после него будут выполнены выходные воздействия *z1* и *z2*.

Так как алгоритм анализа протокола не зависит от структуры автомата, то для тестирования были искусственно построены протоколы различной длины, удовлетворяющие и не удовлетворяющие спецификации.

Для спецификации φ и протокола π алгоритм обратного обхода имеет трудоемкость $O(|\varphi| * |\pi|)$ и требует $O(|\varphi|)$ единиц памяти. Это делает его наиболее эффективным алгоритмом при наличии всего протокола в целом. Тестовая реализация алгоритма проводила верификацию протокола относительно спецификации вида $Gx1 \rightarrow Fz1$ с производительностью около 200 000 записей в секунду.

При отсутствии протокола в целом выбор алгоритма ограничен обходом автомата в глубину и в ширину. Алгоритм обхода в глубину в худшем случае имеет трудоемкость $O\left(|\varphi| * |\pi|^2\right)$ и требует $O(|\pi|)$ единиц памяти, что в ряде случаев (например, при верификации спецификации вида $GF\varphi$) делает его не лучшим выбором. На спецификации $Gx1 \rightarrow Fz1$ скорость работы алгоритма составила около 100 000 записей в секунду.

Наконец, обход в ширину позволяет верифицировать протоколы с трудоемкостью $O(|\varphi| * |\pi|)$, не требуя наличия всего протокола целиком. Однако оценка потребления памяти этим алгоритмом в худшем случае составляет $O\left(2^{|\varphi|}\right)$.

В целом полученные результаты подтверждают рекомендации по использованию указанных алгоритмов, приведенные в работе [3]: при возможности рекомендуется использовать алгоритм обратного обхода. При невозможности его применения следует выбирать между обходом в ширину и глубину в зависимости от вида формулы.

Для изучения характеристик метода на практике был построен набор протоколов различной длины – от 10 до 10^6 записей, в которых записи *xi, z1* и *z2* распределены равномерно. На каждом протоколе был несколько раз запущен алгоритм обратного обхода для каждой из двух рассматриваемых спецификаций. В результате было вычислено среднее время верификации протокола заданной длины. Результаты измерений приведены на рис. 2.

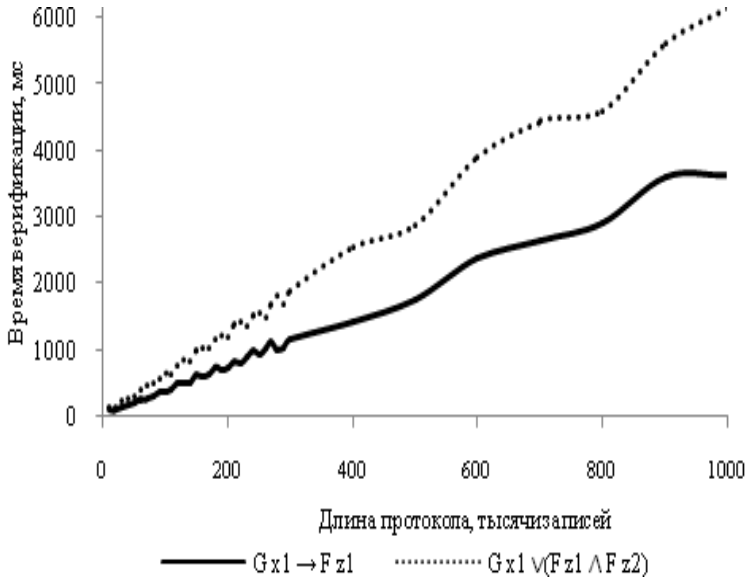


Рис. 2. Скорость работы алгоритма обратного обхода на протоколах различной длины

Важной особенностью динамической верификации программ является зависимость от точки прерывания протокола. Например, если оборвать протокол, используемый в описанном тесте, на записи *x1*, то спецификация окажется невыполненной, так как соответстующая этой записи запись *z1* о выполнении выходного воздействия просто не успела попасть в протокол. В связи с этой особенностью исследователей могут интересовать не столько факт нарушения спецификации, сколько статистика выполнения или не выполнения спецификации. В работе [3] приведены модифицированные алгоритмы обхода, позволяющие собирать такого рода статистику. Предложенный в данной работе метод можно использовать также с этими алгоритмами: последний шаг (построение и анализ контрпримера) следует заменить на анализ полученной статистики.

Заключение

В работе предложен метод автоматической динамической верификации программ. Предложенный метод позволяет верифицировать гораздо более сложные системы автоматов, нежели традиционный метод *Model Checking*. Предложенный метод основан на известном подходе к верификации программ, в котором используются альтернирующие автоматы. В работе проанализирована применимость этого подхода к верификации автоматных программ; в частности, исследованы особенности использования различных алгоритмов обхода альтернирующего автомата.

Автором произведено сравнение характеристик предложенного метода с рядом методов, описанных в [2], на основании чего выработано руководство к выбору того или иного метода верификации в зависимости от поставленной задачи.

Литература

1. Кларк Э., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. – М.: МЦНМО, 2002.
2. Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Теоретические исследования поставленных перед НИР задач. – СПбГУ ИТМО. 2007. – Режим доступа: http://is.ifmo.ru/verification/_2007_02_report-verification.pdf
3. Finkbeiner B., Sipma H. Checking Finite Traces Using Alternating Automata // Form. Methods Syst. Des. – 2004. – 24. – 2.
4. Emerson E. A. Temporal and modal logic / In «Handbook of theoretical Computer Science (Vol. B): Formal Models and Semantics». – MA: MIT Press, 1990. – P. 995–1072.
5. Vardi M. Y. Alternating Automata and Program Verification / Computer Science Today. Recent Trends and Developments. – Vol. 1000 of LNCS. – Springer–Verlag. 1995.
6. Vardi M. Y. Alternating Automata: Checking Truth and Validity for Temporal Logics / Proc. 14th International Conference on Automated Deduction. – Vol. 1249 of LNCS. – Springer–Verlag. 1997.
7. Havelund K., Roşu G. Testing Linear Temporal Logic Formulae on Finite Execution Traces. Technical Report TR 01–08, RIACS. 2001.
8. Mealy G. A Method to Synthesizing Sequential Circuits // Bell System Technical J. – 1955. – 34. – P. 1045–1079.
9. Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Выбор направления исследований и базовых компонентов. – СПбГУ ИТМО. 2007. – Режим доступа: http://is.ifmo.ru/verification/_2007_01_report-verification.pdf

5 МЕТОДЫ РАЗРАБОТКИ АВТОМАТНЫХ ПРОГРАММ

УДК 004.4’232

ДЕКЛАРАТИВНЫЙ ПОДХОД К ВЛОЖЕНИЮ И НАСЛЕДОВАНИЮ АВТОМАТНЫХ КЛАССОВ ПРИ ИСПОЛЬЗОВАНИИ ИМПЕРАТИВНЫХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

А.А. Астафуров

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В работе предлагается декларативный подход к реализации автоматных объектов при использовании объектно-ориентированных императивных языков программирования со статической проверкой типов. Отличительной особенностью предлагаемого подхода является возможность применения наследования и вложения макросостояний.

Ключевые слова: декларативный подход, автоматное программирование, макросостояние

Введение

При реализации автоматов с использованием объектно-ориентированных языков применяются различные подходы. Их можно разделить на императивные и декларативные. Наиболее распространенным императивным подходом является использование паттерна проектирования *State* [1, 2], основными достоинствами применения которого являются распределение специфического поведения между состояниями и выполнение переходов в явном виде непосредственно в коде. Недостатком применения этого паттерна является необходимость создания сложной иерархии при наличии большого числа состояний у автомата, которая может быть решена при использовании шаблона *Decorator* [2, 3].

Основным недостатком императивного подхода в целом при реализации автоматных программ является то, что в коде автоматного объекта присутствуют такие лишние детали, как явное делегирование вызова из контекста конкретному состоянию или вызов вложенного автомата.

В данной работе предлагается подход, являющийся компромиссом между декларативным и императивным. От декларативного подхода используется возможность описания структуры автомата, а от императивного – описание логики переходов, так как она является частью автоматного кода и императивна по своей природе. Рассматриваемый подход к реализации автоматных классов является декларативным, но ориентирован на использование объектно-ориентированных императивных языков со статической проверкой типов. Подход иллюстрируется примерами программ на языке C#.

Описание подхода

При задании поведения автоматов будем использовать модифицированную нотацию диаграмм *Statecharts* [4]. Основное отличие между применением диаграмм *Statecharts* и *SWITCH*-технологией [5] состоит в том, что в *SWITCH*-технологии явно вводится понятие автомата. Семантика используемых в *SWITCH*-технологии графов переходов близка к семантике диаграмм *Statecharts*, но не эквивалентна ей. В *SWITCH*-технологии вводится понятия *«автомат»* и *«вложение автоматов»*, но отсутствуют понятия *«вложенное»* и *«ортогональное»* состояния. Вложение и ортогонализация состояний в *SWITCH*-технологии реализуется при помощи вложения автоматов и введе-

ния понятия *«система автоматов»*. *SWTICH*-технологию удобнее применять в документации: при использовании вложения автоматов не возникает проблем с нотацией, как это происходит при использовании вложенных ортогональных состояний в диаграммах *Statecharts*. Так, например, при использовании диаграмм *Statecharts* возникают трудности с расположением названий макросостояний, содержащих вложенные ортогональные состояния [4].

В предлагаемом подходе автомат с вложенными в него состояниями рассматриваются как набор *макросостояний*, так как у состояния нет никакой специфики по отношению к автомату. Макросостояние может включать в себя другие макросостояния с их состояниями. При реализации библиотеки для поддержки автоматного программирования между состоянием и автоматом также нет никакой разницы. В рамках библиотеки, разрабатываемой в настоящей работе, в системе будет создан класс, определяющий состояния, который, помимо стандартных свойств, таких как текущее состояние, вложенные макросостояния и т.д., реализует интерфейс, описывающий допустимые входные воздействия. При этом отметим, что в диаграмме *State Machine* в языке *UML 2* (диаграмме *State Chart* в языке *UML*) [6] понятие автомата также отсутствует: существуют только состояния, которые могут включать другие состояния. Таким образом, решается вопрос вложения автоматов.

Вложение автоматных объектов

Благодаря отождествлению понятий состояния и автомата и введения термина «макросостояние» определим вложение автоматных объектов следующим образом: при получении события состоянием оно сначала передается всем его вложенным состояниям, а затем выполняется действие, связанное с этим событием. Таким образом, в данной работе будем считать, что событие сначала передается всем вложенным состояниям, а затем выполняется действие внутри состояния.

Наследование автоматных объектов

Рассмотрим вопрос о наследовании автоматов. Разрабатываемый подход должен позволить наследовать автоматы, переопределяя состояния и правила перехода между ними. Наследование автоматных объектов основано на переопределении состояний базового автоматного объекта: производный объект должен переопределить поведение базового объекта как минимум в одном из его состояний. В производный объект могут быть добавлены новые состояния и переходы между ними [7].

Декларативный подход при использовании императивных языков

Декларативный подход и императивный язык. Несмотря на императивность многих современных языков программирования, при их использовании тоже можно применять декларативный подход. Для этого необходимо инкапсулировать все недекларативные детали внутри библиотеки или модели, которая будет управляться декларативной мета-информацией, позволяя таким образом использовать декларативные конструкции. В зависимости от языка и платформы это может быть достигнуто разными способами. Например, в языке *Java* и платформе *Microsoft .NET* существует механизм отражения (*reflection*), позволяющий получать доступ к компонентам класса во время исполнения. Это дает возможность декларативно описывать поведение классов, а затем, во время исполнения, добавлять императивное поведение.

Декларативный подход на основе атрибутов. Для более детального рассмотрения реализации декларативного подхода при применении императивных языков в качестве

примера используем язык *C#* и платформу *Microsoft .NET*. Наличие такой конструкции, как *атрибут* (*attribute*), позволяет применить декларативный подход для этого языка. Атрибуты в языке *C#* – это конструкции, позволяющие добавлять мета-информацию как к членам класса, так и к самим классам. В дальнейшем информация об атрибутах может быть получена во время исполнения библиотекой, инкапсулирующей недекларативные составляющие программы при помощи механизма отражения.

Рассмотрим подходы к выполнению декларативной программы в императивной среде.

Механизм Context Bound Objects. Context Bound Object (объект, привязанный к контексту) – это специальный объект, позволяющий контролировать все вызовы, поступающие к нему из контекста. *Контекст* (в терминах *Microsoft CLR*) – это окружение, устанавливающее правила, которым будут следовать объекты, находящиеся в нем [8]. Таким образом, если объект привязан к контексту, он также привязан и к его правилам. Контекст создается при активации объекта. Общение объекта с внешним миром происходит при помощи сообщений, что не имеет отношения к автоматам. Среда *Microsoft CLR* предоставляет механизмы, позволяющие встраиваться в цепочку обработки сообщений. Это необходимо для их перехвата, а также для генерации или модификации сообщений, проходящих через границу контекста.

Отметим, что использование механизма *Context Bound Object* оправдано в тех случаях, когда производительность системы не критична. Этот механизм выгодно использовать при создании прототипа будущей декларативной системы.

Инструментализация сборок. Данный подход эффективен для продуктов, критичных к скорости, так как все вызовы выполняются напрямую, без дополнительных затрат на управление очередью сообщений, как это происходит в *Context Bound Object*. Явным недостатком рассматриваемого подхода являются трудности при отладке приложения: из-за того, что байт-код сборки модифицируется, и код, полученный в результате этой модификации, перестает соответствовать отладочной информации, сгенерированной на этапе компиляции.

Реализация на платформе *.NET*

Описание библиотеки DOME. В рамках данной работы для реализации декларативного подхода при использовании императивных языков *C#* и *Visual Basic.NET* была разработана библиотека классов *DOME* (*Declarative Object Machines Extension*). На рис. 1 приведена диаграмма основных классов этой библиотеки. Здесь `State` – базовый класс для состояний, а свойство `Container` – экземпляр класса *State*, которому принадлежит рассматриваемое состояние. Это свойство необходимо для доступа к объемлющему состоянию при вложении состояний. Свойство `CurrentState` – текущее вложенное состояние. Оно используется при вложении состояний, а также непосредственно в коде автоматного объекта. Метод `SetState(Type state)` изменяет текущее состояние. В качестве параметра ему необходимо передать тип (класс) состояния, в которое требуется перейти. `StateAttribute` – атрибут, применяемый к классам-наследникам `State` для описания возможных состояний для данного макросостояния. `Type` – *CLR*-тип (класс) состояния. `Name` – имя состояния. *Конструктор* – конструктор с одним параметром, принимает тип (класс) состояния. Перегруженный конструктор с двумя параметрами предоставляет механизм переопределения состояний. Для этого необходимо в качестве значения параметра `Type` указать тип (класс) нового состояния, а в качестве значения параметра *Overrides* – тип (класс) состояния, которое будет переопределено.

В этом случае в ходе создания экземпляра атрибута будет также произведена проверка того, что переопределяемое состояние присутствует выше в иерархии.

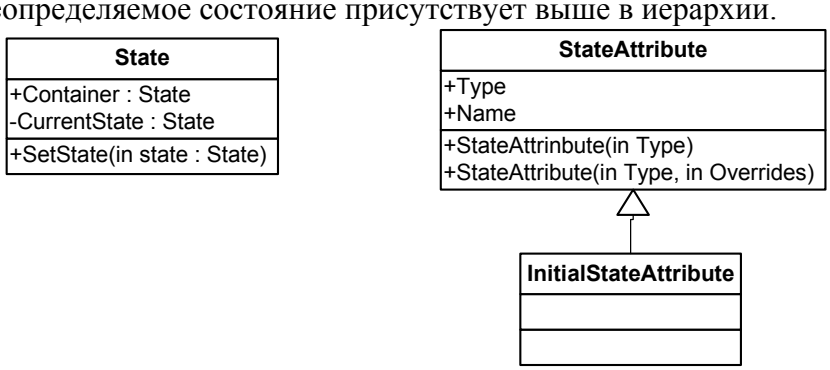


Рис. 1. Основные классы библиотеки

`InitialStateAttribute` имеет ту же семантику, что и атрибут `StateAttribute`, и применяется для обозначения начального состояния автомата или начального вложенного состояния.

Реализация автоматных объектов

В качестве первого примера рассмотрим простейший автомат, состоящий из двух состояний: *ON* и *OFF* (рис.2).

Определим три интерфейса:`IOn` – интерфейс для состояния ON, содержащий только один метод `E0` – единственное событие, которое обрабатывается в этом состоянии; `IOff` – интерфейс для состояния OFF, содержащий метод `E1` – событие, обрабатываемое в этом состоянии; `ISwitch` – интерфейс, описывающий автомат в целом и реализующий интерфейсы `IOn` и `IOff`. Эти интерфейсы объявляются следующим образом:

```
public interface IOn
{
 void E0();
}

public interface IOff
{
 void E1();
}

public interface ISwitch : IOn,IOff
{

}
```

После этого необходимо реализовать классы состояний ON и OFF:

```
class On : State, IOn
{
 public void E1()
 {
 Container.SetState(typeof(Off));
 }
}
class Off : State, IOff
{
 public void E1()
 {
 Container.SetState(typeof(On));
```

```
 }
}
```

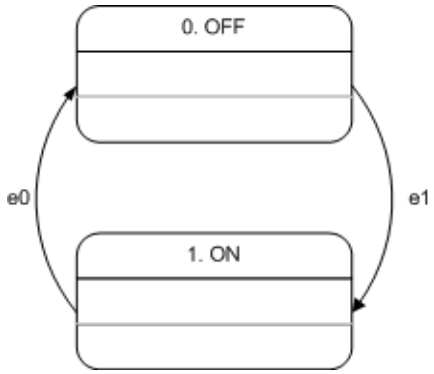


Рис. 2. Автомат выключателя

Поясним приведенный фрагмент кода. Поле `Container` является ссылкой на автомат, которому принадлежит данное состояние. Метод `SetState(Type)` изменяет текущее состояние автомата.

После этого реализуем автомат, который, как отмечено выше, рассматривается как набор макросостояний и поэтому наследуется от класса `State`, реализуя при этом интерфейс `ISwitch`:

```
[State(typeof(On)), State(typeof(Off))]
class Switch : State, ISwitch
{
 public void E0() { }
 public void E1() { }
}
```

Обратим внимание на атрибуты класса `Switch`. Они указывают на то, что автомат, описанный этим классом, будет содержать два состояния `On` и `Off`, описанные, как показано выше.

Во время выполнения программы все вызовы методов класса `Switch` будут перехватываться и передаваться методу текущего состояния с набором параметров, идентичным вызываемому методу. После этого выполняется метод автомата. В случае отсутствия у текущего состояния метода с набором параметров, идентичным вызываемому методу, вызов перейдет непосредственно к методу автомата.

Реализация вложенных состояний

Для описания вложенных состояний рассмотрим более сложный пример (рис. 3). Пусть имеется диаграмма состояний героя-бойца (*Fighter*) из компьютерной игры, в котором детализировано состояние «Прыжок». Рассмотрим, как реализуются вложенные состояния в исходном коде:

```
 [InitialState(typeof(Jumping.Rising)),
         State(typeof(Jumping.Falling)),
State(typeof(Jumping.Hovering)),
State(typeof(Jumping.Finished))]
public class Jumping : State, IFighter
{
 public void Tick()
 {
 if (CurrentState is Finished)
 Container.SetState(
 typeof(Fighter.Main));
```

```
 }

 public void ButtonPressed(Keys key)
 {}
 public bool InAir()
 {
 return true;
 }

 public class Rising : State, ITickable
 {
 public void Tick()
 {
 Console.WriteLine("rising");
Container.SetState(typeof(Hovering));
 }
 }

 public class Hovering : State, ITickable
 {
 public virtual void Tick()
 {
 Console.WriteLine("hovering");
 Container.SetState(
 typeof(Falling));
 }
}

public class Falling : State, ITickable
{
 public void Tick()
 {
 Console.WriteLine("falling");
 Container.SetState(
 typeof(Finished));
 }
}

public class Finished : State, ITickable
 {
 public void Tick()
 {
 }
 }
}
```

Рассмотрим состояния, вложенные в состояние `Jumping`. В этом случае при реализации вложенных состояний `Raising`, `Hovering`, `Falling`, `Finished` событие в состоянии `Jumping` сначала будет передано текущему вложенному состоянию, а затем обработано соответствующим методом состояния `Jumping`. Как и в предыдущем примере, вызов передается при помощи перехвата сообщения с использованием механизма *Context Bound Object*, описанного в разделе с тем же названием.

Реализация наследования

При использовании декларативного подхода достаточно просто и интуитивно понятно удается описывать наследование автоматов. Для этого достаточно создать класс-

наследник базового класса «автомат». При этом для переопределения состояний необходимо использовать атрибут `State`. Таким же образом можно наследовать и базовые состояния, переопределяя их вложенные состояния или добавляя новые.

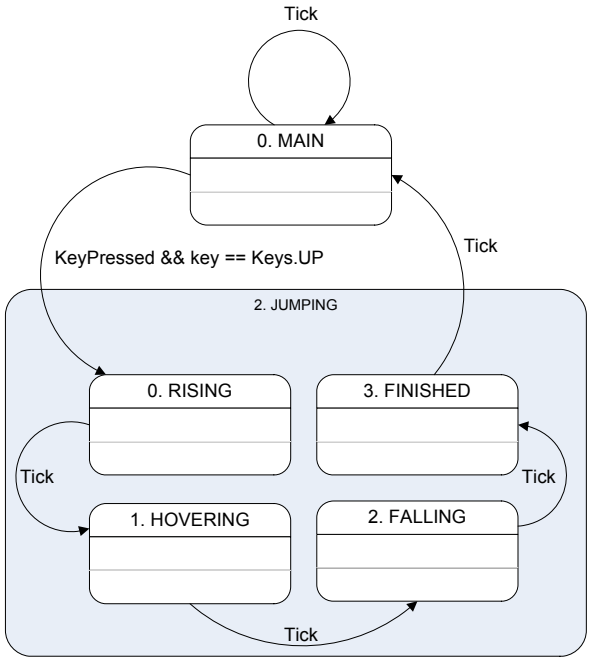


Рис. 3. Диаграмма состояний бойца с детализацией состояния «Прыжок»

В качестве примера рассмотрим задачу переопределения одного из состояний в автомате `Fighter` из предыдущего примера. Пусть необходимо переопределить состояние `Hovering`, вложенное в состояние `Jumping`. Для этого создадим класс `EasternFighter`, являющийся наследником класса `Fighter`, и с помощью атрибутов опишем переопределение состояния `Fighter.Jumping`. Для этого заменим это состояние на класс `Fighter.EasternJumping`, который будет содержать переопределенное вложенное состояние `Hovering`. Программно это выражается следующим образом:

```
[State(typeof(EasternJumping), typeof(Fighter.Jumping))]
public class EasternFighter : Fighter
{
}
```

Теперь необходимо создать наследника класса-состояния `Jumping`, чтобы переопределить его вложенное состояние `Hovering` новым состоянием `EasternHovering`:

```
[State(typeof(EasternHovering),
         typeof(Fighter.Jumping.Hovering))]
public class EasternJumping : Fighter.Jumping
{
}
```

Наконец, необходимо реализовать само состояние `EasternHovering`, создав наследника от класса `Jumping.Hovering`:

```
public class EasternHovering : Fighter.Jumping.Hovering
{
 public override void Tick()
 {
 Console.WriteLine("Eastern Hovering");
 Container.SetState(
 typeof(Fighter.Jumping.Falling));
 }
}
```

Таким образом, когда новый автомат будет вызван, то при переходе в состояние `Jumping` будет использован новый класс `EasternJumping`, который, в свою очередь, использует переопределенное состояние `EasternHovering`, о чем он сообщает при протоколировании на консоль.

Заключение

В данной работе предложен декларативный подход к реализации автоматных объектов. Описаны способы реализации такого подхода при использовании императивных языков, а также предложена библиотека *DOME* (*Declarative Object Machines Extension*), позволяющая реализовать этот подход на платформе *Microsoft .NET*. Использование библиотеки проиллюстрировано примерами реализации наследования и вложения. Эта библиотека будет опубликована на сайте http://is.ifmo.ru/..

Данный подход к реализации автоматов позволяет совместно использовать декларативный и императивный стили программирования, позволяя применять декларативный объектно-ориентированный подход к реализации автоматов с помощью императивных языков. Благодаря тому, что данная концепция описана на уровне парадигмы декларативного программирования без привязки к конкретному языку программирования, ее можно применять и для других императивных языков, например, для языка *Java*.

В дальнейшем планируется более детально рассмотреть вопрос создания библиотеки подобной библиотеке *DOME* для языка *Java*, а также изучить вопрос о виртуальных и не виртуальных состояниях и их переопределении.

Литература

1. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб: Питер, 2001.
2. Adamczyk P. The Anthology of the Finite State Machine Design Patterns / The 10th Conference on Pattern Languages of Programs, 2003.
3. Odrowski J., Sogaard P. Pattern Integration – Variations of State / Proceedings of PLoP96.
4. Harel D. Statecharts: A visual formalism for complex systems // Sci.Comput. Program. – 1987. – Vol.8. – P. 231–274.
5. Шалыто А.А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. – СПб.: Наука, 1998. – Режим доступа: http://is.ifmo.ru/books/switch/1
6. Шопырин Д.Г. Методы объектно-ориентированного проектирования и реализации программного обеспечения реактивных систем. Диссертация … канд. техн. наук / СПбГУ ИТМО. – СПб, 2005. – Режим доступа: http://is.ifmo.ru/disser/shopyrin_disser.pdf
7. Буч Г., Рамбо Д., Джекобсон А. UML. Руководство пользователя. – М.: ДМК, 2000.
8. Box D., Sells C. Essential .NET Vol. 1, The Common Language Runtime. NJ: Addison-Wesley, 2002.

УДК 004.4’232

НАСЛЕДОВАНИЕ АВТОМАТНЫХ КЛАССОВ С ИСПОЛЬЗОВАНИЕМ ДИНАМИЧЕСКИХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ НА ПРИМЕРЕ *RUBY*

К.И. Тимофеев, А.А. Астафуров

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

Объектно-ориентированное программирование с явным выделением состояний совмещает в себе основные преимущества объектно-ориентированной и автоматной парадигм программирования. Основными достоинствами такого подхода являются расширяемость, гибкость и наличие механизма описания сложного поведения, реализованного на конечных автоматах. В данной работе рассматриваются объектно-ориентированный и динамический подходы к наследованию и вложению автоматных классов с использованием динамических языков программирования.

Ключевые слова: динамический язык программирования, автоматное программирование

Введение

При создании систем со сложным поведением целесообразно применять автоматное программирование, называемое также «программирование от состояний» или «программирование с явным выделением состояний». Метод разработки программного обеспечения, основанный на расширенной модели конечных автоматов, ориентирован на создание широкого класса приложений [1]. При совместном использовании объектной и автоматной парадигм программирования этот подход был назван в работе [1] «объектно-ориентированное программирование с явным выделением состояний», которое в дальнейшем будем называть «объектно-автоматное программирование». Однако при использовании автоматного программирования возникают проблемы, связанные с поддержкой автоматного кода и документации, с внесением изменений в систему, а также наглядностью и понятностью автоматного кода.

Объектно-автоматное программирование основано на объектной и автоматной парадигмах программирования. Оно совмещает в себе их основные преимущества, такие как гибкость, расширяемость и наличие мощного механизма описания сложного поведения, основанного на конечных автоматах.

Базовым понятием объектно-автоматного программирования является автоматный класс. Автоматным называется такой класс, поведение объектов которого зависит от текущего управляющего состояния. Эти классы реализуются на основе конечных автоматов [2, 3]. Наследование и вложение состояний, применяемое в объектно-автоматном программировании, позволяет значительно сократить требуемое число переходов для описания сложного автомата [4].

Однако в описываемых в работе [4] объектно-автоматных подходах не всегда удается удобно выделять условия переходов, а также писать код, который наглядно отражает переходы между состояниями, так как логика переходов обычно скрывается в методах-обработчиках входных воздействий.

Для устранения недостатков объектно-автоматного подхода, а также сохранения всех его достоинств в данной работе предлагается рассмотреть применение динамических языков программирования для построения автоматных программ. В качестве такого языка будет использован язык *Ruby*.

В данной работе будут описаны два подхода к разработке автоматных программ на языке *Ruby*, а также выполнено их сравнение с уже существующими решениями. В качестве примера применения подходов рассмотрено создание автоматных классов для чтения и записи в файл с использованием вложения и наследования.

Обзор подходов и решений реализации автоматов

Существует несколько подходов к реализации автоматов, каждый из которых имеет свои достоинства и недостатки. Рассмотрим их.

1. Полностью ручное программирование. Существует различные методы реализации этого подхода, например, использование одного или нескольких операторов `switch`, определяющих действия программы в зависимости от текущего состояния [5], и использование шаблона проектирования *State* [6]. Несмотря на высокую производительность и полный контроль над получаемым кодом, эти методы обладают недостатками: низкая читаемость кода из-за применения множества вложенных операторов `switch` (для первого похода) и большая трудоемкость его написания (для второго подхода).
2. Автоматическая генерация кода по диаграмме переходов. Обычно при таком подходе генерируется код, аналогичный тому, который можно получить с использованием ручного программирования. Эта группа методов обладает следующими достоинствами: высокая скорость создания программного кода и возможность его создания экспертами предметной области, которые не имеют опыта использования языков программирования высокого уровня. Недостатками этого подхода являются: низкая читаемость, связанная с тем, что в качестве целевого языка используется императивный язык, например, *Java*, который не способен полностью отразить декларативную сущность диаграммы переходов автомата [7], потеря информации, специфичной для логики диаграмм переходов (вершины диаграммы и ее переходы заменяются их реализациями на целевом языке – классами и кодом выполнения переходов), недостаточная степень контроля над получаемым кодом и невозможность его ручного изменения.
3. Ручное написание кода с применением специальной библиотеки. В этом случае происходит перенос диаграммы переходов в вызовы указанной библиотеки, которая по этим инструкциям строит внутреннее представление рассматриваемой диаграммы. Затем по этому представлению происходит реализация автомата. Основным достоинством этого подхода является то, что вызовы библиотеки отражают семантику диаграммы переходов (каждый вызов может, например, соответствовать объявлению состояния или перехода). Это позволяет создавать читаемый код, который легко поддерживать [8].

В приведенных ниже работах используется третий подход к реализации автоматных классов, однако каждое из указанных решений наряду с достоинствами, обладает и недостатками.

1. В работе [8] используется динамический язык программирования *Ruby*, с помощью которого была разработана библиотека `STROBE`, что позволило удобно преобразовать автомат из графической нотации в программный код. Однако в работе [8] не было реализовано наследование автоматных классов.
2. В работе [7] применяется декларативный подход к реализации автоматных объектов при помощи объектно-ориентированных императивных языков программирования со статический проверкой типов. Используя язык программирования *C#*, было реализовано наследование и вложение автоматов. В то же время код программы обладает синтаксической избыточностью, что не позволяет легко модифицировать логику переходов.
3. Плагин *Acts as State Machine* [9] для *Ruby on Rails* позволяет описывать логику веб-приложения с использованием автоматного программирования. *Ruby on Rails* – это фреймворк для разработки веб-приложений. Плагин – это компонент, который можно добавить в приложение *Ruby on Rails*, расширив его функциональность [10]. Этот плагин обладает простым синтаксисом. Однако в нем не было реализовано наследование и вложение автоматных классов.

В данной работе устранены такие недостатки приведенных выше решений, как синтаксическая избыточность, невозможность наследования или вложения автоматов. Для этого реализованы два подхода к преобразованию диаграмм переходов в программный код. Первый подход будет использовать объектно-ориентированные свойства языка *Ruby*, второй – динамические.

В качестве примера будут реализованы автоматные классы для чтения из файла и записи в файл, поведение которых может быть обобщено и структурировано с помощью наследования. Эти классы образуют иерархию, показанную на рис. 1.

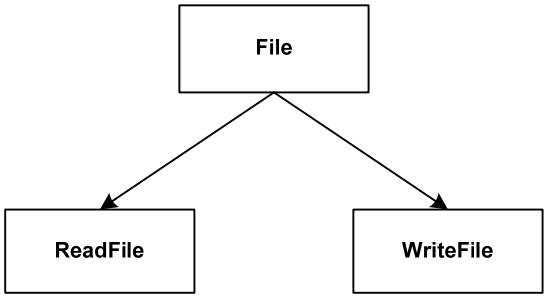


Рис. 1. Иерархия классов доступа к файлу

Динамические языки программирования

Языки этого класса характеризуется динамической типизаций и возможностью выполнения кода, созданного в процессе работы программы. Динамические языки программирования позволяют:

- расширять программу, классы или объекты с помощью добавления нового кода во время выполнения программы;
- выполнять любые инструкции в виде текста (метод `eval`);
- использовать:
  - лямбда-функции, которые обозначают безымянные функции;
  - замыкания – способность лямбда-функции запоминать свой контекст выполнения;
  - функции высшего порядка – возможность передать функцию в качестве аргумента и возможность вернуть функцию в качестве результата работы;
  - отражения – анализ типов и метаданных, представление полного текста программы как данных;
  - макросы – правило или шаблон для преобразования входной последовательности в соответствии с объявленной процедурой. Результатом выполнения макросов является программа исходного языка – макросы расширяют исходный язык.

Стоит отметить, что неотъемлемой частью функционального программирования являются функции высшего порядка и замыкания, что позволяет в некоторой степени объединить функциональный и динамические подходы.

Функциональные языки программирования

Близость к математической формализации и функциональная ориентированность дают следующие преимущества этому классу языков программирования:

- сложные программы строятся на основе агрегирования функций;
- полиморфизм (отличается по сути от аналогичного термина в объектно-ориентированном программировании) – обеспечение возможности обработки разнородных данных [11];

- простота тестирования и верификации на основе строгого математического доказательства корректности программ;
- унификация представления программы и данных;

Таким образом, при создании программ на функциональных языках программист основное внимание обращает на предметную область и в меньшей степени заботится о рутинных операциях («сборка мусора», оптимизация и т.д.).Элементы функционального программирования используются и в императивных языках, например, в языке C# 3.0 существуют лямбда-функции, функции высшего порядка и замыкания.

Особенности языка *Ruby*

Ruby – кросс-платформенный, объектно-ориентированный динамический язык программирования с элементами функционального стиля [12]. Он имеет все черты объектно-ориентированных языков, такие как инкапсуляция, наследование и полиморфизм. Множественное наследование реализуется с помощью *технологии примесей*. Примесь – это механизм расширения одного класса методами другого. Язык *Ruby* содержит в себе все элементы динамических языков программирования, которые были указаны выше, и частично поддерживает функциональное программирование за счет лямбда-функций, замыканий и функций высшего порядка.

Динамические и функциональные свойства языка *Ruby* позволяют:

1. Получить автомат, код которого удобен для чтения и дальнейшего сопровождения, благодаря возможности разработки предметно-ориентированного языка на макросах рассматриваемого языка.

Предметно-ориентированный язык (*Domain Specific Language*, *DSL*) – это язык программирования, который разработан специально для решения определенного круга задач [13]. Преимуществами применения *DSL* являются:

   - возможность создания решения в терминах предметной области. В результате эксперты данной предметной области могут понимать, проверять и модифицировать написанный код;
   - самодокументированный код;
   - повышение качества, надежности и сопровождаемости программного обеспечения.

2. Легко верифицировать автомат. Для этого потребуется решить две задачи:
   - доказать корректность построения структуры автомата с помощью DSL. Так как свойством функциональных языков программирования является отсутствие побочных эффектов (побочным эффектом считается изменение функцией состояния своего окружения), то программа будет состоять из набора несвязанных друг с другом функций, что связано с тем, что результат выполнения одной функции не зависит от результата выполнения другой. Таким образом, для верификации DSL будет достаточно убедиться в корректной работе каждой отдельной функции;
   - верифицировать автомат, применяя темпоральную логику.
3. Решить вопрос изоморфного переноса графической нотации в программный код [14] – при реализации группового перехода не требуется дублировать код перехода для вложенных состояний.

Использование графической нотации для описания логики автоматных классов

При проектировании автоматных классов предлагается использовать *диаграммы поведения*, являющиеся расширенной версией графов переходов. Отличительной особенностью диаграмм поведения, представленных в работе [4], является возможность описания декомпозиции и структурирования логики автоматных классов с помощью наследования.

При наследовании производный автоматный класс может расширять и модифицировать поведение базовых автоматных классов. Такая модификация базовых классов основана на переопределении их состояний. В производный класс могут быть добавлены новые состояния и переходы между ними [4].

Диаграмма поведения обладает следующими свойствами.

1. Диаграмма изображает одно или несколько состояний системы и переходы между ними.
2. Состояния на диаграмме могут быть объединены в группы, которые могут быть вложены друг в друга.
3. Переходы могут начинаться и заканчиваться в состоянии или в группе состояний (переходы, начинающиеся в группе состояний, называются групповыми переходами).
4. Переходы могут начинаться и заканчиваться в одном и том же состоянии. При этом они называются петлями.

Основные элементы графической нотации [4] представлены на рис. 2.

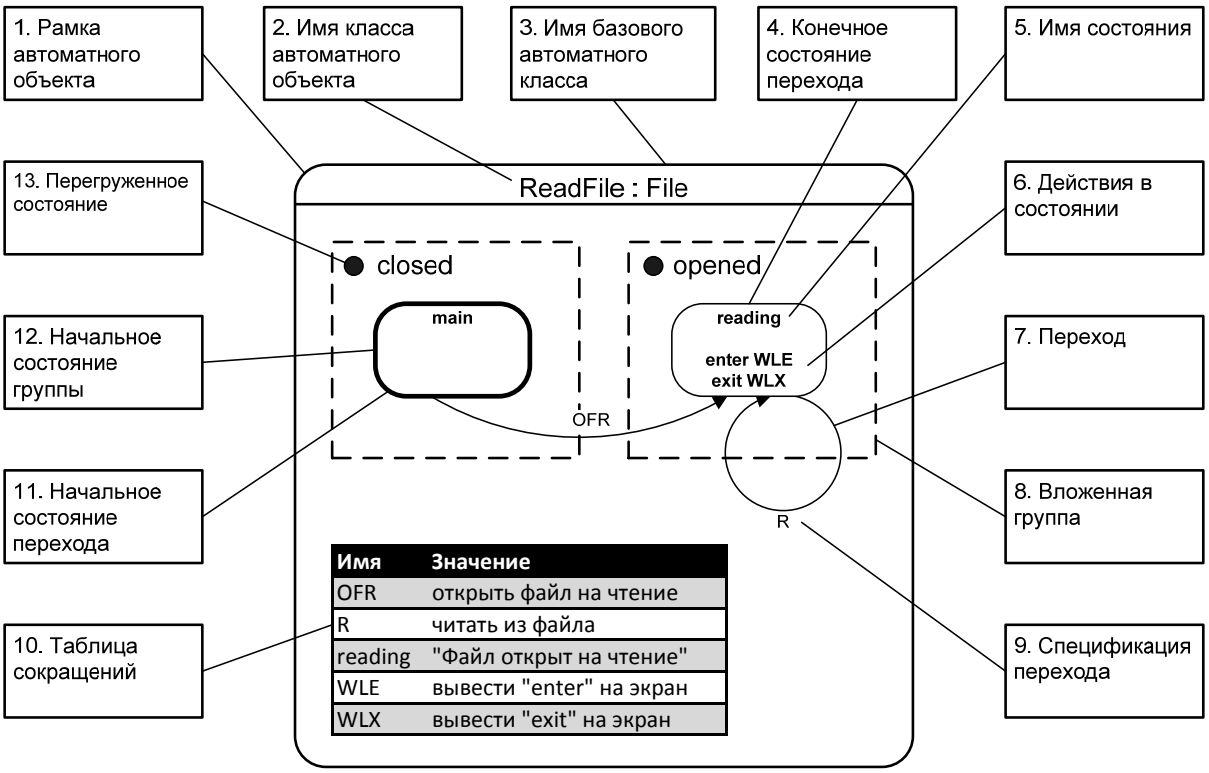


Рис 2. Основные элементы графической нотации

Графическая нотация используется в данной работе для описания автоматных классов.

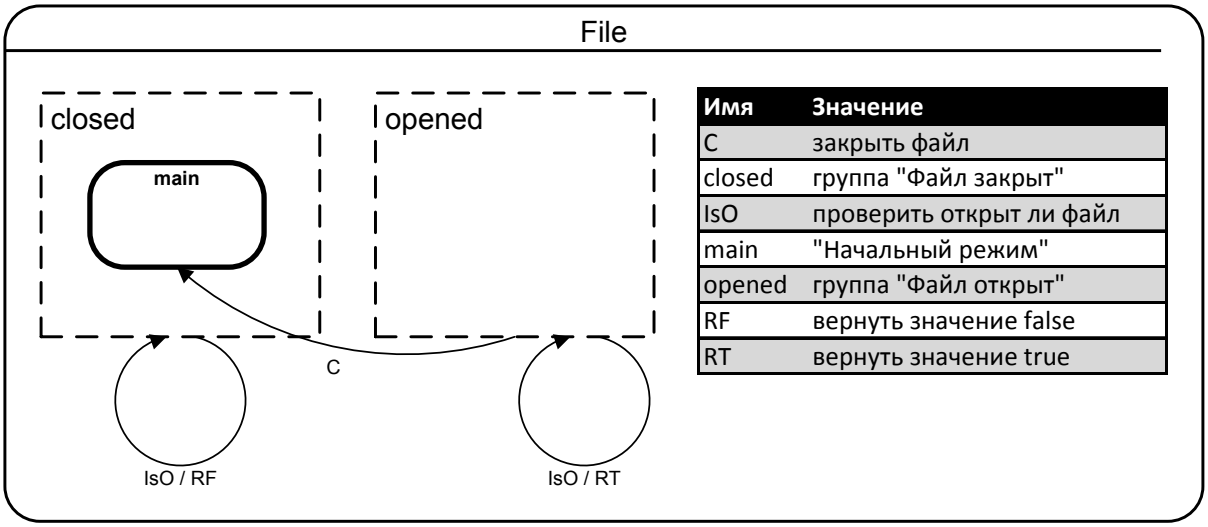
Наследование и вложение автоматов с использованием объектно-ориентированных особенностей языка *Ruby*

На рис 3. представлен абстрактный автоматный класс для работы с файлом (`File`), который состоит из двух вложенных групп `closed` и `opened`. Вложенная группа `closed` включает состояние `main`. Вложенная группа `opened` не содержит ни одно-

го состояния и будет использована при создании наследуемых автоматов `ReadFile` и `WriteFile` (рис. 4).

Определены общие для всех файлов переходы:

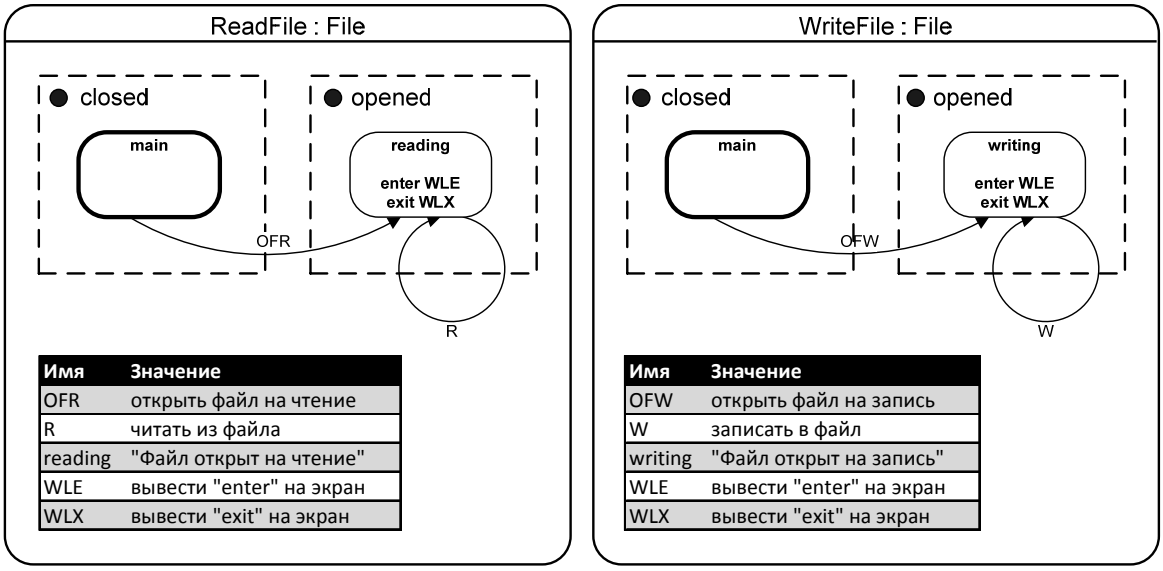
- `IsO` – позволяет определить, открыт ли в данный момент файл;
- `C` – закрывает файл.



Имя	Значение
C	закрыть файл
closed	группа "Файл закрыт"
IsO	проверить открыт ли файл
main	"Начальный режим"
opened	группа "Файл открыт"
RF	вернуть значение false
RT	вернуть значение true

Рис 3. Абстрактный класс работы с файлом

Диаграмма поведения классов `ReadFile` и `WriteFile`, построенная с использованием наследования, приведена на рис. 4. Корневым элементом предлагаемой иерархии является абстрактный класс `File`, обобщающий некоторые аспекты доступа к файлу.



Имя	Значение
OFR	открыть файл на чтение
R	читать из файла
reading	"Файл открыт на чтение"
WLE	вывести "enter" на экран
WLX	вывести "exit" на экран

Имя	Значение
OFW	открыть файл на запись
W	записать в файл
writing	"Файл открыт на запись"
WLE	вывести "enter" на экран
WLX	вывести "exit" на экран

Рис 4. Реализуемые классы ReadFile и WriteFile

Автомат `ReadFile` наследует от автомата `File` группы `closed` (содержит состояние `main`) и `opened` (не содержит ни одного состояния). В группу `opened` добавляется новое состояние `Reading` с действием `WLE` при входе в состояние и действием

WLX при выходе из состояния. Добавляются новые переходы OFR (из состояния Main в состояние Reading) и R (петля в состоянии Reading).

Автомат WriteFile наследует от автомата File группы closed (содержит состояние main) и opened (не содержит ни одного состояния). В группу opened добавляется новое состояние Writing с действием WLE при входе в состояние и действием WLX при выходе из состояния. Добавляются новые переходы OFW (из состояния Main в состояние Writing) и W (петля в состоянии Writing).

Реализуем приведенные выше автоматные классы, используя объектно-ориентированные особенности языка *Ruby*. При этом каждый автоматный класс и каждое состояние являются отдельным классом.

Разработаем абстрактный автоматный класс File. Для этого предварительно реализуем все его состояния и группы состояний. Приведенный ниже класс отвечает за создание состояния main, которое наследуется от класса абстрактного состояния State.

Созданное состояние имеет имя main:

```
class Main < State
 def initialize container
 super :Main, container
 end
end
```

Разработаем группу opened. Она имеет переход C в состояние main с действием на этом переходе RT (рис. 3). Переходы задаются публичными методами (например, переход C), тогда как действия задаются приватными методами (например, действие RT):

```
class Opened < Automaton
 def initialize name, container=nil
 super name, container
 end

 def C
 @container.state :Closed
 RT()
 end

 private
 def RT; true; end
end
```

Аналогичным образом создадим вложенную группу closed:

```
class Closed < Automaton
 def initialize name, container=nil
 super name, container
 automaton Main.new(self)
 initial :Main
 end

 def IsO
 @container.state :Main
 RF()
 end

 private
 def RF; false; end
end
```

После того, как были созданы все состояния и вложенные группы, можно создать автомат File, который будет включать в себя две вложенные группы opened и closed.

```
class FileAbstract < Automaton
 def initialize name, container=nil
 super name, container
 automaton Closed.new(:Closed, self), Opened.new(:Opened, self)
 initial :Closed
 end
end
```

После того, как был описан абстрактный автоматный класс для работы с файлами (`FileAbstract`), создадим наследуемый от него автомат `ReadFile` (рис. 4). Для этого создадим новое состояние `Reading`:

```
class Reading < State
 def initialize container
 super :Reading, container
 end

 def R; @container.state :Reading; end
end
```

Группа `ReadOpened` наследована от группы `opened`. Следовательно, она будет иметь все переходы и состояния, которые были объявлены в группе `opened`. Однако в эту группу требуется добавить состояние `Reading`, что достигается вызовом метода `automaton`. Этот метод используется для создания вложенной группы с заданными состояниями. Начальным состоянием вложенной группы `ReadOpened` является состояние `Reading`:

```
class ReadOpened < Opened
 def initialize name, container=nil
 super name, container
 automaton Reading.new(self)
 initial :Reading
 end
end
```

Так же создадим оставшиеся состояния `ReadMain`, `ReadFile` и группу `ReadClosed`:

```
class ReadMain < Main
 def OFR; @container.state :Reading; end
end

class ReadClosed < Closed
 def initialize name, container=nil
 super name, container
 automaton ReadMain.new(self)
 end
end

class ReadFile < FileAbstract
 def initialize name, container=nil
 super name, container
 automaton ReadOpened.new(:Opened, self),
          ReadClosed.new(:Closed, self)
 end
end
```

В результате создан программный код, отображающий автоматы, представленные на рис. 3, 4.

Отметим следующее достоинство использования объектно-ориентированного подхода для реализации наследования и вложения автоматов [7]: каждое состояние и каждый автомат является отдельным классом, что позволяет сохранить иерархию автомата при переносе его в объектно-ориентированный код. Недостаток этого подхода –

синтаксическая избыточность, связанная с необходимостью использования синтаксических конструкций языка, что не позволяет легко модифицировать логику переходов и расширять код.

В работе [4] графическая нотация может быть преобразована в программный код лишь одним способом: с помощью объектно-ориентированного программирования, тогда как динамические языки программирования позволяют применить еще и другой способ, основанный на метапрограммировании, который будет рассмотрено ниже.

Наследование и вложение автоматов с помощью динамических свойств языка *Ruby*

Этот подход устраняет синтаксическую избыточность, которая была отмечена в качестве недостатка предыдущего решения.

В результате применения динамического языка *Ruby* был разработан специальный предметно-ориентированный язык, который состоит из двух основных конструкций:

- `current` – начальное состояние автомата;
- `state` – создает новое состояние, которое может иметь несколько *ключевых параметров* (ключевые параметры позволяют передавать аргументы функции по имени параметра) :
  - `:enter` – действие при входе в состояние;
  - `:exit` – действие при выходе из состояния.
- `transition` – переход из одного состояния в другое. Имеет несколько ключевых параметров:
  - `:from` – состояние, из которого происходит переход;
  - `:to` – состояние, в которое происходит переход;
  - `:guard` – условие, при котором может произойти переход;
  - `:event` – действие на переходе.

Ключевые параметры `:guard`, `:exit` и `:enter` являются лямбда-функциями. Условие, при котором выполняется переход, является лямбда-предикатом – лямбда-функцией, которая возвращает значение «истина» или «ложь».

Приведем код, который реализует автоматы, изображенные на рис. 3, 4:

```
class AbstractFile < Automaton::Base
 class << self
 def RT; lambda { true } end
 def RF; lambda { false } end
 def IsO; lambda { @file.is_opened? } end
 end

 current :main

 state :closed, :enter => lambda { { :current_state => :main }
         }
 state :opened
 state :main

 transition :from => :opened, :to => :main,
 :event => :C

 transition :from => :closed, :to => :closed,
 :event => :IsO,
 :proc => RF()

 transition :from => :opened, :to => :opened,
 :event => :IsO,
```

```
 :proc => RT()
end

class ReadFile < AbstractFile
 class << self
 def WLE; lambda { puts "enter" } end
 def WLX; lambda { puts "exit" } end
 def R; lambda { @file.read } end
 def OFR; lambda { |name| @file.open(name, "r") } end
 end

 state :reading,
 :enter => WLE(),
 :exit => WLX()

 state :opened, :enter => lambda { { :current_state => :reading
          } }

 transition :from => :reading, :to => :reading,
 :event => :R

 transition :from => :main, :to => :reading,
 :event => :OFR
end

class WriteFile < AbstractFile
 class << self
 def WLE; lambda { puts "enter" } end
 def WLX; lambda { puts "exit" } end
 def R; lambda { @file.write } end
 def OFW; lambda { |name| @file.open(name, "w") } end
 end

 state :writing,
 :enter => WLE(),
 :exit => WLX()

 state :opened, :enter => lambda { { :current_state => :writing
          } }

 transition :from => :writing, :to => :writing,
 :event => :W

 transition :from => :main, :to => :writing,
 :event => :OFW
end
```

Рассмотрим конструкции предметно-ориентированного языка, использованные в этой программе:

- установить начальное состояние автомата в `main`;

```
current :main
```

- создать новое состояние с именем `writing`. При входе в это состояние выполняется функция `WLE()`, а при выходе из него – функция `WLX()`.

```
state :writing,
 :enter => WLE(),
 :exit => WLX()
```

- создать переход из состояния `closed` в состояние `closed`, что образует петлю при входном воздействии `IsO()`. В случае перехода будет выполнена функция `RF()`:

```
transition :from => :closed, :to => :closed,
 :event => :IsO,
```

```
 :proc => RF()
```

Рассмотрим наследование автомата `ReadFile`. Для этого необходимо добавить новое состояние `Reading` и два новых перехода:

```
class ReadFile < AbstractFile
 state :reading,
 :enter => WLE(),
 :exit => WLX()

 transition :from => :reading, :to => :reading,
 :event => :R

 transition :from => :main, :to => :reading,
 :event => :OFR
end

Дополнительно потребуется определить методы R, WLE, WLX и OFR:
class ReadFile < AbstractFile
 class << self
 def WLE; lambda { puts "enter" } end
 def WLX; lambda { puts "exit" } end
 def R; lambda { @file.read } end
 def OFR; lambda { |name| @file.open(name, "r") } end
 end
end
```

Таким образом, преимуществом данного подхода является отсутствие синтаксической избыточности, которая позволяет использовать такие достоинства *DSL*, как самодокументированный код, простота его понимания и расширяемость. В то же время существенным недостатком подхода является потеря иерархии родительского автомата: состояние автомата не является отдельным классом, как было при использовании объектно-ориентированного подхода.

Протоколирование

Благодаря такому свойству динамических языков программирования, как макросы, можно разработать модуль, который будет автоматически отслеживать переходы, и выводить об этом соответствующую информацию. При этом не важно, был ли использован объектно-ориентированный или динамический подход к реализации автоматов.

Рассмотрим пример для автомата `ReadFile`:

```
class ReadFile < AbstractFile
 def initialize name, container=nil
 super name, container
 automaton ReadOpened.new(:Opened, self),
         ReadClosed.new(:Closed, self)
 end

 tracer
end
```

Этот класс отличается от использованного выше лишь одной строкой `tracer`, которая является макровызовом и создает для каждого перехода и состояния дополнительный метод, который выводит отладочную информацию. Она состоит из названия метода, который был вызван, и переданных ему аргументов. Дополнительно можно вывести информацию о времени выполнения того или иного метода:

```
ReadMain::OFR begin
In the OFR
ReadMain::OFR end
```

Заключение

В настоящей работе были разработаны два подхода, которые позволяют изоморфно переносить диаграммы состояний в диаграммы поведения, а после этого изоморфно преобразовывать полученный результат в программный код. При этом были рассмотрены объектно-ориентированные и динамические языки свойства языка программирования *Ruby*.

Достоинством использования объектно-ориентированного подхода к реализации наследования и вложения автоматов [4] является то, что каждое состояние и каждый автомат – это отдельный класс, что позволяет сохранить иерархию автомата при переносе его в объектно-ориентированный код. Однако данный подход обладает таким недостатком, как синтаксическая избыточность, что связано с необходимостью использования синтаксических конструкций языка и не позволяет легко модифицировать и расширять код.

Показано, что для динамических языков программирования может быть разработан предметно-ориентированный язык (*DSL*), который позволяет:

- создать решение в терминах предметной области. В результате эксперты в этой области могут понимать, проверять и модифицировать написанный код;
- обеспечить самодокументированный код.

Недостатком подхода является потеря иерархии родительского автомата, так как состояние автомата не является отдельным классом, как было обеспечено в объектно-ориентированном подходе.

Таким образом, оптимальным решением будет являться объединение объектно-ориентированного и динамического подходов: предметно-ориентированный язык, разработанный с использованием динамического языка программирования *Ruby*, будет использован для создания классов при применении объектно-ориентированного подхода.

В данной статье не был рассмотрен чистый функциональный подход к реализации конечных автоматов. Этот подход позволит более эффективно верифицировать автоматы. Кроме этого, в дальнейшем планируется расширить наследование автоматных классов до множественного наследования.

Литература

1. Поликарпова Н.И., Шалыто А.А. Автоматное программирование. Учебно-методическое пособие. – СПбГУ ИТМО, 2007. – Режим доступа: http://is.ifmo.ru/books/_umk.pdf.
2. Шалыто А.А., Туккель Н. И. SWITCH-технология – автоматный подход к созданию программного обеспечения «реактивных» систем. // Программирование. – 2001. – № 5. – С.45–62. – Режим доступа: http://is.ifmo.ru/works/switch/1/.
3. Шалыто А.А., Туккель Н.И. От тьюрингова программирования к автоматному // Мир ПК. – 2002. – № 2. – С. 144–149. – Режим доступа: http://is.ifmo.ru/works/turing/.
4. Шопырин Д.Г., Шалыто А.А. Графическая нотация наследования автоматных классов // Программирование. – 2007. – № 5. – С. 62–74. – Режим доступа: http://is.ifmo.ru/works/_12_12_2007_shopyrin.pdf.
5. Шалыто А.А., Туккель Н.И. Реализация автоматов при программировании событийных систем // Программист. – 2002. – № 4. – С.74–80. – Режим доступа: http://is.ifmo.ru/works/evsys/.
6. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб.: Питер, 2007.

7. Астафуров А.А., Шалыто А.А. Декларативный подход к вложению и наследованию автоматных классов при использовании императивных языков программирования. // Software Conference (Russia). – М.: ТЕКАМА. 2007. – С. 230–238. – Режим доступа: http://is.ifmo.ru/works/_astafurov_secr_word_2003.pdf.
8. Степанов О.Г., Шалыто А.А. Предметно-ориентированный язык автоматного программирования на базе динамического языка Ruby // Информационно-управляющие системы. – 2007. – № 4. – С. 22–27. – Режим доступа: http://is.ifmo.ru/works/_2007_10_05_aut_lang.pdf.
9. Scott B. Acts As State Machine. – Режим доступа: http://agilewebdevelopment.com/plugins/acts_as_state_machine.
10. Obie F. The Rails Way. – NJ: Addison-Wesley, 2007
11. Field A., Harrison P. Functional Programming. – Wokingham: Addison-Wesley, 1993
12. Thomas D., Fowler C., Hunt A. Programming Ruby. – Texas: Pragmatic Bookshelf, 2004.
13. Parr T. The Definitive Antlr Reference: Building Domain-Specific Languages. – Texas: Pragmatic Bookshelf, 2007.
14. Заякин Е.А., Шалыто А.А. Метод устранения повторных фрагментов кода при реализации конечных автоматов. – СПбГУ ИТМО, 2007. – Режим доступа: http://is.ifmo.ru/projects/life_app/

6 ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА АВТОМАТНОГО ПРОГРАММИРОВАНИЯ

УДК 004.4.’232

ИНСТРУМЕНТАЛЬНОЕ СРЕДСТВО ДЛЯ ПОДДЕРЖКИ АВТОМАТНОГО ПРОГРАММИРОВАНИЯ *UNIMOD 2*. ПРОЕКТИРОВАНИЕ. ВАЛИДАЦИЯ. ВЕРИФИКАЦИЯ. РЕАЛИЗАЦИЯ

Д.Ю. Кочелаев, И.А. Лагунов, Б.З. Хасянзянов, Б.Р. Яминов
(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В данной статье дается краткий обзор инструментального средства *UniMod 2* для поддержки автоматного программирования. Данное инструментальное средство предназначено для разработки автоматных программ и предоставляет средства для визуального проектирования, отладки, валидации и верификации автоматных программ.
Ключевые слова: автоматное программирование, инструментальное средство, валидация, верификация

Введение

В 2005 г. было разработано инструментальное средство *UniMod* [1], позволяющее визуально конструировать автоматные диаграммы и исполнять программы, созданные на их основе. Это средство базируется на следующих основных концепциях:

- *UML* [2, 3];
- автоматное программирование [4];
- платформа *Eclipse* [5];
- язык программирования *Java;*
- открытый программный код (исходные коды доступны для скачивания на сайте проекта [1]).

Инструментальное средство *UniMod* является одной из реализаций «исполняемого *UML*» [6]. При использовании автоматного подхода программа в целом проектируется с помощью двух типов *UML*-диаграмм: диаграмм классов, которые представляются в виде схем связей автоматизированных объектов [7], и диаграмм состояний, которые реализуют автоматы, указанные на диаграмме классов. Эти диаграммы исполняются автоматически, а фрагменты программ, соответствующие входным и выходным воздействиям, пишутся вручную. Таким образом, это инструментальное средство позволяет при создании программ совмещать разные уровни абстракции (диаграммы и текст на языке *Java*) и разные стили программирования (графический и текстовый).

Перечислим достоинства инструментального средства *UniMod*.

1. Возможность визуального построения диаграмм классов и состояний. Средство обеспечивает тесную интеграцию между визуальным представлением и реализацией представленных объектов в коде.
2. Возможность не только интерпретировать и компилировать построенную программу, но и осуществлять визуальную отладку, добавляя точки останова на состояния и переходы. Данная возможность совместно с автоматным представлением программы существенно облегчает поиск ошибок.
3. Возможность валидации построенных автоматных моделей. Это дает дополнительную информацию об ошибках, которые были допущены при построении диаграмм состояний.

Однако инструментальное средство *UniMod* имеет и некоторые недостатки:

- валидация осуществляется с применением неоптимального алгоритма, и ее реализация «зашита» в код инструментального средства;
- отсутствует возможность верификации программ;
- отсутствует возможность текстового ввода автоматных программ.

Разработка инструментального средства *UniMod 2*

В 2007 г. была начата работа над второй версией инструментального средства *UniMod*, названной *UniMod 2*. Цель этого проекта – устранить имеющиеся в средстве *UniMod* недостатки, усовершенствовать некоторые из его компонент, а также ввести новую функциональность, такую как верификация программ и обеспечение текстового ввода автоматных программ.

Модель данных

Одно из основных отличий инструментального средства *UniMod 2* от прототипа состоит в широком использовании ресурсов, предоставляемых платформой *Eclipse*. Это позволяет улучшить все компоненты системы. В частности, для описания автоматной метамодели применен фреймворк *EMF* (*Eclipse Modeling Framework*) [8]. Этот фреймворк предназначен для построения приложений, основанных на модельных структурах данных. Он позволяет генерировать исходный код на языке программирования *Java* для составляющих модели, а также адаптеров для просмотра и редактирования модели. Использование этого фреймворка позволяет избежать возможных при реализации функциональность вручную ошибок. Редактирование модели (например, внесение новых атрибутов у составляющих модели) выполняется визуально и не требует ручной модификации кода, так как он будет сгенерирован в дальнейшем автоматически.

Использование фреймворка *EMF* позволяет использовать уже существующие фреймворки и библиотеки для реализации других частей приложения, таких как визуализация или валидация. При этом число строк кода, который написан вручную, значительно сокращается, что ведет к снижению вероятности появления ошибки в конечном программном продукте.

Отметим также, что использование *EMF* не ограничивает в реализации модельных классов, так как сгенерированный код может быть доработан вручную. При этом изменения, внесенные в реализацию вручную, будут сохранены и при автоматической перегенерации классов из измененной модели.

Валидация

В инструментальном средстве *UniMod 2* используется новый алгоритм валидации автоматов [9]. Этот алгоритм (проверка непротиворечивости построенной модели и автоматной метамодели) является более быстродействующим, чем алгоритм, используемый в первой версии инструментального средства. Повышение производительности достигается за счет использования предварительных вычислений, а также контекстнозависимой проверки правил – правила проверяются только в контексте измененного элемента. Такой подход существенно сокращает объем вычислений, которые необходимо выполнить для проверки непротиворечивости модели, а, следовательно, увеличивает скорость этой проверки.

Рассмотрим подробнее два этапа алгоритма проверки непротиворечивости модели.

1. *Этап предварительных вычислений.* Перед началом работы с моделью собираются данные, необходимые для быстрого определения правил, которые могли нарушиться в результате изменения модели. Таким образом, для каждого правила определяется

набор атрибутов, изменение которых может повлечь за собой изменение состояния правила – определяется, выполняется правило или нет.

2. *Этап динамической проверки.* При каждом изменении модели, на основе изменного атрибута и данных, которые были получены на этапе предварительных вычислений, составляются множества правил, которые необходимо перепроверить, и выполняется их проверка. Отметим, что проверка выполняется в контексте того элемента, атрибут которого был изменен, что существенно сокращает число проверяемых правил, по сравнению с оригинальным алгоритмом.

Для записи правил, описывающих ограничения на модель, используется *Object Constraint Language* (*OCL*) [10]. Этот язык разработан компанией *IBM* специально для записи ограничений на *UML*-модели [2, 3]. Он используется при решении задачи валидации. Для иллюстрации такого способа записи ограничений рассмотрим ограничение на отсутствие переходов в начальное состояние, записанное на языке *OCL*:

```
context coremodel::State inv:
     self.substates->select(x | x.initial).
          incomingTransitions->size() = 0
```

Любое правило начинается с задания контекста, в котором необходимо осущствлять проверку. Далее следует условие, которое необходимо проверить. В данном случае у автомата выбираются все вложенные в него состояния, а для каждого начального состояния осуществляется проверка на отсутствие переходов в это состояние. Такая запись позволяет легко определить контекст, в котором необходимо проверять правило, а также атрибуты, изменение которых может повлечь за собой изменение состояния правила.

Для оценки производительности применяемого алгоритма выполнено теоретическое и практическое сравнение этого алгоритма и алгоритма, который был использован в инструментальном средстве *UniMod*. В результате показано, что предложенный алгоритм обладает существенно более высокой производительностью [9].

Верификация

Нововведением в инструментальном средстве *UniMod 2* стала возможность верификации программ [11]. Для этого указанное средство было интегрировано с верификатором *Bogor* [12] за счет создания ряда дополнительных модулей. В результате появилась возможность верифицировать автоматные программы на основе метода *Model Checking* [13].

В известных подходах для верификации автоматной программы с помощью метода *Model Checking* необходимо построить специальную формальную модель, которая будет представлять программу. Утверждения, которые необходимо верифицировать, также следует перевести из терминов программы в термины построенной модели. Поэтому существовало два схожих способа верификации программы. Первый [14] – преобразовать автоматную программу в модель *Крипке* [13], а затем верифицировать ее вручную с помощью метода *Model Checking*. Второй способ [15] – преобразовать автоматную программу во входной язык верификатора, а затем запустить верификатор для проверки корректности модели. Если автоматная программа некорректна, то верификатор возвращает отчет об ошибке. Однако отчет об ошибке возвращается в терминах преобразованной формальной модели, а не исходной автоматной программы. Таким образом, пользователю приходится преобразовывать отчет об ошибке обратно в термины исходной автоматной программы, что в некоторых случаях может быть сложной задачей.

Рис. 1 описывает известные методы верификации автоматных программ.

Преимущество метода верификации, который реализован в инструментальном средстве *Unimod 2*, состоит в том, что верификатор *Bogor* работает непосредственно с автоматной программой, как если бы она уже была формальной моделью, готовой к ве-

рификации. В результате отсутствует необходимость в сложном преобразовании автоматной программы в формальную модель и обратно. Поэтому процесс верификации становится значительно проще, быстрее и надежнее. Схема этого метода верификации приведена на рис. 2.

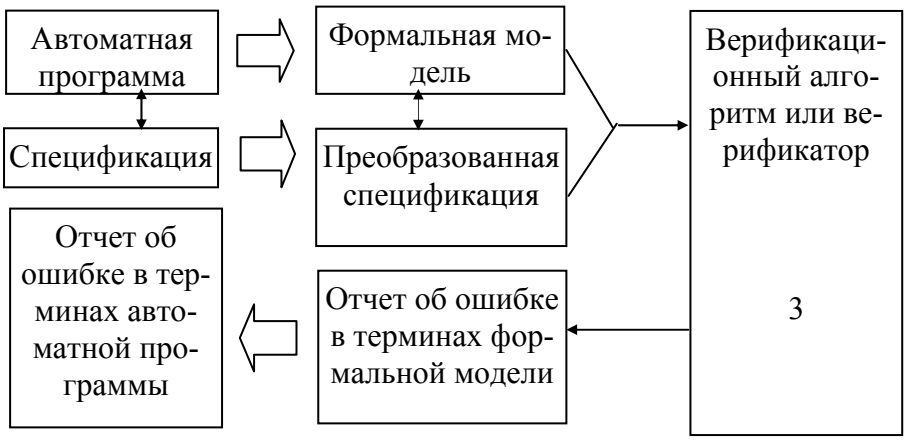


Рис. 1. Существующие методы верификации. Необходимы четыре шага: (1) преобразовать программу в формальную модель, (2) преобразовать спецификацию в спецификацию в терминах формальной модели, (3) верифицировать модель и (4) преобразовать отчет об ошибке обратно в термины автоматной программы

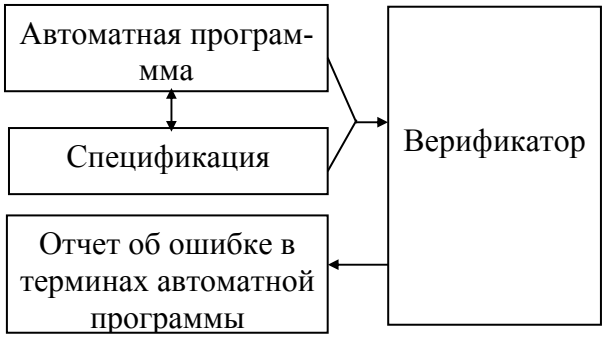


Рис. 2. Новый метод верификации — без преобразования программы

Таким образом, *UniMod 2* – это первое инструментальное средство, которое позволяет создавать автоматные программы, запускать и *автоматически* верифицировать их. Для проверки корректности программы пользователю достаточно ввести список утверждений, которые должны быть верны для программы, и запустить процесс верификации. По окончании процесса верификации пользователю сообщается, какие утверждения верны, а какие нет. Более того, если какое-то из утверждений нарушается, пользователь получает детальную историю шагов, выполненных программой, которые привели к нарушению утверждения. Это позволяет пользователю легко найти причину ошибки. Выделяются два основных преимущества верификации программ перед тестированием.

1. Верификация позволяет выявить причину ошибки, а не только саму ошибку.
2. При верификации проверяются все варианты работы программы, в то время как при тестировании проверяются только некоторые из них.

Отладка

В инструментальном средстве *UniMod 2* построение отладчиков для предметно-ориентированных языков программирования выполняется на основе технологии *EMF*

[16]. Предлагаемый метод позволяет реализовать универсальную систему отладки для произвольного предметно-ориентированного языка, которая реализует стандартные средства отладки, предусмотренные средой разработки *Eclipse*. Она включает:

- точки останова (*breakpoints*), позволяющие приостановить выполнение приложения в указанном месте;
- пошаговое выполнение;
- просмотр значений переменных в конкретный момент исполнения приложения (*context variables*).

Визуализация

Для построения редактора моделей, используемого для отображения и управления процессом отладки, была использована технология *Graphical Modeling Framework* (*GMF*) [17]. Эта технология позволяет по описанию метамодели и дополнительных конфигураций построить полнофункциональный графический редактор, встраивающийся как расширение в среду разработки *Eclipse*. Конфигурации задают список элементов метамодели, которые будут использованы на диаграмме, а также их графическое представление и наборы связанных с ними элементов управления редактора.

Также предусмотрена возможность выделения некоторых элементов модели в редакторе [16]. Эта возможность используется для отображения нарушенных ограничений, которые были выявлены в процессе валидации [9]. Этот же механизм будет применяться и для визуализации результатов верификации. Таким образом, разработан и внедрен единый механизм визуализации действий над моделью (отладка), изменений модели и выделения ее элементов (валидация).

Текстовый язык автоматного программирования

Визуальный подход к разработке автоматных программ обладает существенным недостатком – процесс визуального построения диаграмм более трудоемок по сравнению с использованием традиционных текстовых языков программирования. Для решения этой проблемы был разработан текстовый язык автоматного программирования *FSML* (*Finite State Machine Language*) [18], а также реализован редактор этого языка для инструментального средства *UniMod*.

Выделим особенности языка *FSML* и определяемые им особенности инструментального средства *UniMod*.

- Текстово-визуальный подход к разработке автоматных программ позволяет совместно использовать текстовый редактор языка *FSML* и графический редактор диаграмм *UniMod* для создания и модификации автоматных моделей.
- Явное задание графа переходов гарантирует его изоморфность программе. Это позволяет программисту избежать многих ошибок на этапе кодирования.
- Интеграция кода на языке *FSML* с объектно-ориентированным кодом обеспечивается автоматической генерацией автоматной модели по *FSML*-программе и использованием объектов управления. При этом отсутствует необходимость в автоматически сгенерированном коде на языке общего назначения.
- Переиспользование компонентов кода возможно благодаря поддержке вложенных автоматов и интеграции с объектно-ориентированным кодом через объекты управления. Вложенные автоматы позволяют использовать повторно элементы логики, а объекты управления – элементы поведения.
- Краткость и понятность синтаксиса языка обусловливают эффективность его использования в целом.

Редактор языка *FSML* связывает этот язык с инструментальным средством *UniMod*. При этом редактор является эффективным средством разработки автоматных программ. Рассмотрим реализованные в нем функции.

- Построение автоматной модели по коду программы является основной функцией редактора. Для этого используются лексический и синтаксический анализаторы, выполняющие разбор программы. Поскольку эти средства естественно представляются в виде событийной системы, они были реализованы на основе автоматного подхода с использованием инструментального средства *UniMod*. При этом лексический анализатор является поставщиком событий-лексем, а синтаксический анализатор – автоматом, управляющим процессом разбора *fsml*-программы.
- Валидация программы проверяет автоматную программу на наличие ошибок синтаксиса и подсвечивает их.
- Автоматическое завершение ввода пользователя позволяет значительно ускорить процесс разработки автоматных программ, что дает дополнительное преимущество текстовому подходу по сравнению с визуальным [19].

Заключение

В результате разработки инструментального средства *UniMod 2* были усовершенствованы или добавлены следующие компоненты системы:

- изменено представление автоматной модели для обеспечения удобства работы с ней;
- усовершенствован алгоритм валидации программ;
- обеспечена возможность верификации автоматных программ;
- перепроектирован редактор модели для интеграции в него удобных инструментов редактирования, предоставляемых технологией *GMF*;
- введен отладчик программ, написанных на предметно-ориентированных языках программирования, адаптированный для отладки автоматных моделей;
- создана единая система визуализация отладки и валидации;
- разработан текстовый язык автоматного программирования *FSML* и создан редактор для него, интегрируемый с инструментальным средством *UniMod* [18].

Литература

1. Гуров В.С., Мазин М.А.. Нарвский А.С., Шалыто А.А. Инструментальное средство для поддержки автоматного программирования // Программирование. – 2007. – № 6. – С. 65–80. – Режим доступа: http://is.ifmo.ru/works/_2008_01_27_gurov.pdf
2. Гома Х. UML. Проектирование систем реального времени, распределенных и параллельных приложений. – М.: ДМК, 2002.
3. Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя. – М.: ДМК, 2000.
4. Шалыто А.А. Switch-технология. Алгоритмизация и программирование задач логического управления. – СПб: Наука, 1998. – Режим доступа: http://is.ifmo.ru/books/switch/6
5. Веб-сайт проекта Eclipse. – Режим доступа: http://www.eclipse.org/
6. Mellor S., Balcer M. Executable UML: A Foundation for Model-Driven Architecture. – Addison-Wesley, 2002.
7. Поликарпова Н.И., Шалыто А.А. Учебно-методическое пособие по дисциплине «Автоматное программирование». – СПбГУ ИТМО, 2007. – Режим доступа: http://is.ifmo.ru/books/_umk.pdf

8. Веб-сайт проекта Eclipse Modeling Framework. – Режим доступа: http://www.eclipse.org/modeling/emf/
9. Кочелаев Д.Ю. Методы динамической проверки правил непротиворечивости автоматной модели. Бакалаврская работа. – СПбГУ ИТМО, 2007. – Режим доступа: http://is.ifmo.ru/papers/_kochelaev-bachelor.pdf
10. Warmer J., Kleppe A. The Object Constraint Language: Getting Your Models Ready for MDA. – Addison-Wesley, 2003.
11. Гуров В.С., Шалыто А.А., Яминов Б.Р. Технология верификации автоматных моделей программ без их трансляции во входной язык верификатора / Материалы международной научно-технической конференции «Многопроцессорные вычислительные и управляющие системы». – Т. 1. – 2007. – С. 198–203.
12. Гуров В.С., Шалыто А.А., Яминов Б.Р. Технология верификации автоматных программ без их трансляции во входной язык верификатора. Материалы международной научно-технической конференции “Многопроцессорные вычислительные и управляющие системы ” (МВУС-2007). Т. 1, с. 198–203. – Режим доступа: http://is.ifmo.ru/verification/_jaminov.pdf
13. Robby, Dwyer M., Hatcliff J. Bogor: A Flexible Framework for Creating Software Model Checkers /IEEE Conf. of the Testing: Academic & Industrial Conference – Practice and Research Techniques (TAIC PART) 2006, pp. 3–22. – Режим доступа: http://ieeexplore.ieee.org/iel5/11139/35654/01691665.pdf?arnumber=1691665
14. Кларк Э., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. – М.: МЦНМО, 2002. – Режим доступа: http://is.ifmo.ru/verification/_klark_gamberg_pered_verification.djvu
15. Вельдер С.Э., Шалыто А.А. Верификация простых автоматных программ на основе метода Model Checking / Материалы XV научно-методической конференции «Высокие интеллектуальные технологии и инновации в образовании и науке». – СПбГПУ, 2008. – С. 285–288. – Режим доступа: http://is.ifmo.ru/download/2008-02-25_politech_verification.pdf
16. Лукин М.А., Шалыто А.А. Автоматизация верификации визуальных автоматных программ / Материалы XV научно-методической конференции «Высокие интеллектуальные технологии и инновации в образовании и науке». – Режим доступа: СПбГПУ, 2008. – С. 296, 297. – Режим доступа: http://is.ifmo.ru/download/2008-02-25_politech_tezis.pdf
17. Хасянзянов Б. С. Метод создания отладчиков для доменно-ориентированных языков программирования на основе технологии Eclipse Modeling. Бакалаврская работа. – СПбГУИТМО, 2007. – Режим доступа: http://is.ifmo.ru/papers/domainlanguagedebugmethod/
18. Веб-сайт проекта Eclipse GMF. – Режим доступа: http://www.eclipse.org/modeling/gmf/
19. Лагунов И.А. Разработка текстового языка автоматного программирования и его реализация для инструментального средства UniMod на основе автоматного подхода. Бакалаврская работа. – СПбГУ ИТМО, 2008. – Режим доступа: http://is.ifmo.ru/papers/fsml/fsmleditor.pdf
20. Гуров В.С., Мазин М.А., Шалыто А.А. Автоматическое завершение ввода условий на диаграмме состояний // Информационно-управляющие системы. – 2008. – №1. – С. 24–33.

7 ТЕКСТОВЫЕ ЯЗЫКИ АВТОМАТНОГО ПРОГРАММИРОВАНИЯ

УДК 004.432.4

ТЕКСТОВЫЙ ЯЗЫК АВТОМАТНОГО ПРОГРАММИРОВАНИЯ

В.С. Гуров, М.А. Мазин, А.А. Шалыто

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В настоящей работе описывается текстовый язык автоматного программирования, построенный на основе системы метапрограммирования JetBrains MetaProgramming System (MPS). Этот язык лишен таких недостатков графического языка автоматного программирования, как низкая скорость ввода диаграмм, и более привычен для программиста.

Ключевые слова: автоматное программирование, текстовый язык

В рамках проекта *UniMod* [1] предложены метод и средство для моделирования и реализации объектно-ориентированных программ со сложным поведением на основе автоматного подхода. Модель системы в рамках указанного проекта предлагается строить с помощью двух типов *UML*-диаграмм: диаграммы классов и диаграммы состояний [2]. При этом на диаграмме классов, которая представляется в виде схемы связей и взаимодействия автоматов, изображаются источники событий, автоматы и объекты управления, которые реализуют функции входных и выходных воздействий. Код для источников событий и объектов управления пишется на языке *Java* [3]. Поведение автоматов описывается с помощью диаграмм состояний.

При помощи инструментального средства *UniMod* выполнен ряд проектов, которые доступны по адресу http://is.ifmo.ru/unimod-projects/. Данные проекты показали эффективность применения автоматного программирования и средства *UniMod* при реализации систем со сложным поведением, но также выявили и ряд недостатков:

- ввод диаграмм состояний с помощью графического редактора трудоемок;
- многие программисты предпочитают работать с текстовым представлением программы, несмотря на то, что диаграммы позволяют представлять информацию более компактно и обозримо;
- невозможно в одном *Java*-классе совместить автомат и объект управления, что не позволяет прозрачно использовать автоматное пограммирование совместно с объектно-ориентированным, так как в настоящее время код, генерируемый из автоматной модели, не является в полной мере объектно-ориентированным.

В настоящий момент существует несколько текстовых языков, поддерживающих программирование с применением автоматов, как в рамках парадигмы автоматного программирования (UniMod FSML [4], State Machine [5]), так и вне ее (SMC [6], TABP [7], SCXML [8], Ragel [9]). Каждый из этих языков программирования обладает некоторыми недостатками и ограничениями.

Язык State Machine представляет собой расширение языка Java, основанное на паттерне проектирования State Machine [10]. Этот паттерн позволяет в объектно-ориентированном стиле описывать структуру автомата. Однако такое описание оказывается громоздким, что препятствует использованию языка State Machine в реальной разработке.

Язык SMC – универсальный компилятор автоматов. Этот язык позволяет описывать в автоматы в едином формате и генерировать целевой код на различных языках общего назначения. В то же время интеграция этого языка с целевыми языками требует от разработчиков дополнительных усилий – необходимо существенно модифицировать код класса для того, чтобы связать его с автоматом, описывающим его поведение.

В универсальном языке программирования TABP предусмотрены встроенные конструкции, облегчающие написание автоматов. Однако синтаксис этого языка вынуждает разработчика оперировать понятиями более низкоуровневыми, чем «состояние». Кроме того, универсальность языка TABP доставляет дополнительные трудности разработчику, связанные с необходимостью изучать конструкции нового языка даже для тех задач, для которых подходят более распространенные универсальные языки.

Язык Ragel предназначен для описания конечных автоматы с помощью регулярных выражений. Такой подход ограничивает область применимости этого языка задачами лексического анализа и спецификации протоколов,

Стандарт языка SCXML – спецификация XML-представления автоматов Харела [11]. Использовать этот язык непосредственно для написания кода автоматов на нем крайне затруднительно, скорее он подходит для обмена автоматами Харела между приложениями.

Для устранения перечисленных недостатков авторы предлагают новый подход к разработке автоматных программ и применению автоматов в объектно-ориентированных системах. В рамках этого подхода предлагается использовать систему метапрограммирования JetBrains Meta Programming System (MPS) [12, 13], которая позволяет создавать проблемно-ориентированные языки (Domain Specific Language – DSL) [13, 14].

Для задания языка в системе MPS требуется разработать:

- структуру абстрактного синтаксического дерева (АСД) [15] для разрабатываемого языка. Узлам АСД могут соответствовать такие понятия, как «объявление класса», «вызов метода», «операция сложения» и т.п.;
- модель текстового редактора для каждого типа узла АСД. Задание редактора для узла АСД равноценно заданию конкретного синтаксиса для этого узла. При этом если для традиционных текстовых языков программирования создание удобного редактора – отдельная сложная задача, то для языков, созданных с помощью средства MPS, редакторы являются частью языка. Эти редакторы поддерживают автоматическое завершение ввода текста и проверку корректности программы;
- модель ограничений на экземпляры АСД;
- модель системы типов [16] для языка;
- модель трансформации программы на задаваемом языке в исполняемый код.

Система *MPS* позволяет как создавать новые языки, так и расширять языки, уже созданные с помощью этой системы.

В отличие от традиционных языков, языки, созданные с помощью системы *MPS*, не являются текстовыми в общепринятом смысле, так как при программировании на них пользователь пишет не текст программы, а вводит ее в виде АСД с помощью специальных редакторов. Структура и внешний вид этих редакторов таковы, что работа с моделью программы для пользователя выглядит, как традиционная работа с текстом программы. Отказ от традиционного текстового ввода программ значительно упрощает создание новых языков [17] – исчезает необходимость в разработке лексических и синтаксических анализаторов, и, как следствие, перестают действовать ограничения на класс грамматик языков. Недостатком такого подхода является зависимость языков от системы *MPS* – невозможно разрабатывать программы без этой системы. Однако подобное ограничение присуще и традиционным, чисто текстовым языкам, которые зависят от компиляторов. Впрочем, после трансляции программы, написанной на языке, созданном в системе *MPS*, исполняемый код перестает зависеть от этой системы.

Ядро среды *MPS* написано на кросс-платформенном языке *Java*. В связи с этим в среде *MPS* существуют развитые средства для взаимодействия с *Java*-платформой. Для написания *Java*-кода в среде *MPS* разработан язык *baseLanguage*. Этот язык является

почти полной реализацией спецификации *Java* 5 [18]. В нем определены такие конструкции, как «класс», «интерфейс», «метод», «предложение», «выражение» и т.д.

Язык для автоматного программирования в среде *MPS* получил название *stateMachine*. Он представляет собой автоматное расширение языка *baseLanguage*. Основной целью, которая преследовалась при разработке языка *stateMachine*, было создание средства, позволяющего описывать поведение классов в виде автоматов, не накладывая на сами классы никаких дополнительных ограничений. Более того, автоматное описание поведения класса должно быть инкапсулировано. Это означает, что код, использующий класс, не знает о том, каким образом задано поведение класса. Этот подход отличается от предложенного в работе [1] тем, что позволяет использовать автоматы не только в программах, написанных в соответствии со SWITCH-технологией, но и в традиционных объектно-ориентированных программах.

В качестве примера использования языка *stateMachine* рассмотрим систему управления лифтом [19]. Используемая реализация системы является упрощенным решением задачи управления лифтом, описанной в работе [20]. Логика управления всеми подсистемами лифта реализована в одном автомате.

Каждый автомат в языке *stateMachine* связан с некоторым классом и описывает его поведение. Чтобы задать поведение класса с помощью автомата, необходимо в этом классе определить события, на которые будет реагировать автомат. Особенностью языка *stateMachine* является то, что события в нем – это методы специального вида. В языке *Java* декларации методов различаются по способу реализации:

- обычные методы, реализация которых следует сразу за объявлением метода;
- нативные методы, реализованные на платформо-зависимых языках программирования;
- абстрактные методы, вообще не имеющие реализации.

```
public class Elevator extends <none>  implements DoorsEngineListener    {
                                                  ElevatorEngineListener
                                                  LoadingTimerListener
  <<initializer>>
  <<static fields>>
  public int currentFloor = 1;
  public TaskList tasks = new TaskList();
  public ElevatorEngine elevatorEngine;
  public DoorsEngine doorsEngine;
  public LoadingTimer loadingTimer;
  <<properties>>

  public Elevator(ElevatorEngine elevatorEngine, DoorsEngine doorsEngine,
    this.elevatorEngine = elevatorEngine;
    this.doorsEngine = doorsEngine;
    this.loadingTimer = loadingTimer;
    this.elevatorEngine.addElevatorEngineListener(this );
    this.doorsEngine.addDoorsEngineListener(this );
    this.loadingTimer.addLoadingTimerListener(this );
  }


  public event doorsOpened( );
  public event doorsClosed( );
  public event floorReached(int floor);
  public event call(int floor, CallType callType);
  public event openDoors( );
  public event loadingTimeout( );
  public event fire( );
```

Рисунок. События автомата, управляющего лифтом

Событие в языке *stateMachine* – это метод, реализация которого находится в автомате и зависит от текущего состояния автомата. Например, в классе *Elevator* объявлены следующие события (см. рисунок):

- «*doorsOpened*», «*doorsClosed*» – события, которые посылает объект управления *DoorsEngine,* когда двери оказываются в максимально открытом и полностью закрытом положении соответственно;
- «*floorReached*» – событие, которое посылает объект управления *ElevatorEngine,* когда лифт достигает очередного этажа;
- «*call*» – событие, которое получает автомат, когда пассажир нажимает кнопку вызова лифта на этаже, или в кабине лифта;
- «*openDoors*» – событие, которое получает автомат, когда пассажир нажимает кнопку экстренного открывания дверей в кабине лифта;
- «*loadingTimeout*» – событие, которое посылает объект управления *LoadingTimer*, когда истекает время ожидания погрузки пассажиров.

Так как события являются обычными методами, их можно использовать в качестве реализации абстрактных методов. Это позволяет уменьшить количество зависимостей в программе. Например, объект управления *ElevatorEngine* не имеет непосредственных ссылок на класс *Elevator*. Вместо этого в программе управления лифтом объявлен интерфейс *ElevatorEngineListener*, и объект управления *ElevatorEngine* извещает о событиях экземпляры этого интерфейса. Класс *Elevator* реализует интерфейс *ElevatorEngineListener* и поэтому может обрабатывать события объекта управления *ElevatorEngine*. Таким образом, применяется классический паттерн проектирования «Обозреватель» [21].

Реакция автомата на событие зависит от состояния, в котором автомат находится. Состояния автомата бывают двух типов – обычные и конечные. При этом конечные состояния не имеют исходящих переходов. Таким образом, когда автомат оказывается в конечном состоянии, он перестает обрабатывать события.

Первое по порядку состояние, объявленное в автомате, считается начальным. При создании экземпляра класса, для которого определен автомат, после выполнения конструктора осуществляется переход в начальное состояние.

Обычные состояния могут быть вложены друг в друга. При этом если состояние содержит другие состояния, то оно называется составным. При переходе в составное состояние выполняется переход в первое по порядку вложенное в него состояние. Поэтому автомат после обработки события не может оказаться в составном состоянии. Каждый исходящий из составного состояния переход работает так, как если бы такой переход был добавлен к каждому вложенному состоянию.

Каждый автомат в языке *stateMachine* связан с некоторым классом, имеет доступ к его полям и методам. Поэтому в языке *stateMachine* нет необходимости в специальных конструкциях для взаимодействия между автоматами, так как вместо вложения одного автомата в другой можно использовать агрегацию одного класса с автоматом, другим классом с автоматом, а посылка события из одного автомата в другой есть не что иное, как вызов метода. После написания программы на языке *stateMachine* она сначала транслируется в *Java*-код, а затем компилируется стандартным *Java*-компилятором. Преимуществом этого языка является простота его использования в объектно-ориентированных приложениях, написанных на языке *Java*. При применении этого языка проверка корректности программы осуществляется на стадии ее написания, а не в процессе компиляции.

Исходя из изложенного, можно утверждать, что в настоящей работе предложен подход к построению программ, который качественно упрощает применение автоматов в объектно-ориентированных программах.

Литература

1. Гуров В.С., Мазин М.А., Нарвский А.С., Шалыто А.А. UML. SWITCH-Технология. Eclipse // Информационно-управляющие системы. – 2005. – №6(13). – С. 12–17. – Режим доступа: http://is.ifmo.ru/works/uml-switch-eclipse/
2. Гуров В.С., Мазин М.А., Шалыто А.А. Операционная семантика UML-диаграмм состояний в программном пакете UniMod / Труды XII Всероссийской научно-методической конференции "Телематика-2005". – СПб: СПбГУ ИТМО, 2005. – Т.1. – С.74–76. – Режим доступа: http://is.ifmo.ru/works/uml-switch-eclipse/
3. Гуров В., Мазин М., Нарвский А., Шалыто А. UML. SWITCH-технология. Eclipse// «Информационно-управляющие системы», 2004, № 6, с.12-17. – Режим доступа: http://is.ifmo.ru/works/uml-switch-eclipse/
4. Гуров В.С., Мазин М.А., Шалыто А.А. Операционная семантика UML-диаграмм состояний в программном пакете UniMOD. – Режим доступа: http://tm.ifmo.ru/tm2005/db/doc/get_thes.php?id=224
5. Eckel B. Thinking in Java. – NJ: Prentice Hall, 2006.
6. Лагунов И.А. Разработка текстового языка автоматного программирования и его реализация для инструментального средства UniMod на основе автоматного подхода. – 2008.
7. Шамгунов Н.Н., Корнеев Г.А., Шалыто А.А. State Machine — расширение языка Java для эффективной реализации автоматов // Информационно-управляющие системы. – 2005. – № 1. – С. 16–24.
8. SMC – State Machine Compiler. – Режим доступа: http://smc.sourceforge.net/.
9. Цымбалюк Е. А. Текстовый язык автоматного программирования ТАВР. – 2008.
10. State Chart XML (SCXML): State Machine Notation for Control Abstraction. W3C Working Draft 16 May 2008. – Режим доступа: http://www.w3.org/TR/2008/WD-scxml-20080516/
11. Thurston A. Parsing Computer Languages with an Automaton Compiled from a Single Regular Expression. / 11th International Conference on Implementation and Application of Automata (CIAA 2006), Lecture Notes in Computer Science, volume 4094, pp. 285—286. – Taipei, Taiwan, August 2006.
12. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Mass., 1995.
13. Harel D. Statecharts: A visual formalism for complex systems //Science of Computer Programming. – 1987. - № 8, pp. 231–274.
14. Дмитриев С. Языково-ориентированное программирование: следующая парадигма //RSDN Magazine. – 2005. – № 5.
15. Фаулер М. Языковой инструментарий: новая жизнь языков предметной области – Режим доступа: http://www.maxkir.com/sd/languageWorkbenches.html
16. Ward M. Language Oriented Programming //Software — Concepts and Tools, 15, 1994.
17. Ахо А., Сети Р., Ульман Дж. Компиляторы. Принципы, технологии, инструменты. – М.: Вильямс, 2003.
18. Luo Z. Computation and Reasoning: A Type Theory for Computer Science. – Oxford University Press, 1994.
19. Simonyi C. The Death of Computer Languages, the Birth of Intentional Programming // The Future of Software, Univ. of Newcastle upon Tyne, England, Dept. of Computing Science, 1995.
20. J. Gosling, B. Joy, G. Steele, G. Bracha. The Java Language Specification, Third Edition – Addison Wesley, 2005.
21. Кнут Д. Искусство программирования. Т. 1: Основные алгоритмы. – М.: Вильямс, 2001. – 712 с.

22. Наумов Л. А., Шалыто А. А. Искусство программирования лифта. Объектно-ориентированное программирование с явным выделением состояний // Информационно-управляющие системы. – 2003. – № 6. – С.38–49; Мир ПК – Диск. – 2004. – № 2. – С 18. – Режим доступа: http://is.ifmo.ru.
23. E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. – Addison Wesley, Reading, Mass., 1995.

УДК 004.432.4

ТЕКСТОВЫЙ ЯЗЫК АВТОМАТНОГО ПРОГРАММИРОВАНИЯ *FSML* ДЛЯ ИНСТРУМЕНТАЛЬНОГО СРЕДСТВА *UNIMOD*

И.А. Лагунов

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

Инструментальное средство *UniMod* позволяет проектировать и исполнять автоматные программы. Основное его ограничение – возможность только визуального описания конечных автоматов. В статье рассматриваются текстовый язык автоматного программирования *FSML* (*Finite State Machine Language*) и его редактор, позволяющие описывать автоматы в указанном средстве с помощью текстового языка автоматного программирования.

Ключевые слова: автоматное программирование, UniMod, текстовый язык

Введение

В настоящее время в сфере разработки программного обеспечения широко используются объектно-ориентированные языки программирования. Однако для широкого класса задач управления традиционный объектно-ориентированный подход является не самым оптимальным. В этом случае *SWITCH*-технология, предложенная в работах [1, 2] для поддержки автоматного программирования, является, пожалуй, наиболее естественным решением.

К настоящему моменту было выполнено несколько попыток облегчить применение этой сравнительно молодой парадигмы программирования [3]. Создаются графические среды, позволяющие создавать и редактировать графы переходов автоматов [4, 5]. Одну из таких сред предоставляет проект *UniMod* [4]. Это надстройка для интегрированной среды разработки *Eclipse* [6], позволяющая строить графы переходов автоматов, интерпретировать или компилировать автоматные программы на одном из объектно-ориентированных языков, проверять корректность построения графов переходов, а также отлаживать созданную программу непосредственно в среде разработки. Однако используемый в этом проекте способ описания конечных автоматов с помощью графического редактора весьма трудоемок.

Для решения этой проблемы автором был создан текстовый язык автоматного программирования *FSML* и реализован его редактор (*Editor*), который используется в инструментальном средстве *UniMod 2*. При этом синтаксический анализатор языка и некоторые возможности редактора *FSMLEditor* реализованы на основе *SWITCH*-технологии и инструментального средства *UniMod*.

Описание языка

Язык *FSML* предназначен для текстового представления конечных автоматов и их связей. Его возможности позволяют с использованием синтаксиса, близкого к синтак-

сису языка программирования *Java*, описать события, состояния, вложенные автоматы, переходы и другие элементы автоматной модели.

Рассмотрим пример реализации на предложенном языке модели пешеходного светофора с таймером (листинг 1). Полное описание синтаксиса языка приведено в [7].

Листинг 1. Код примера на языке *FSML*

```
1 uses trafficlight.provider.TrafficSignalProvider;
2
3 statemachine TrafficLightWithTimer {
4
5 trafficlight.object.RedLamp red;
6 trafficlight.object.GreenLamp green;
7 trafficlight.object.Timer timer;
8
9 initial Init {
10 transitto Inactive;
11 }
12 Active {
13 on switch transitto Inactive;
14
15 initial InitActive {
16 execute red.turnOn, timer.reset
17 transitto Red;
18 }
19 Red {
20 on tick if timer.value == 0
21 execute red.turnOff, green.turnOn, timer.reset
22 transitto Green;
23 on tick else
24 execute timer.decrement;
25 }
26 Green {
27 on tick if timer.value < 5
28 execute green.turnBlinking, timer.decrement
29 transitto GreenBlinking;
30 on tick else
31 execute timer.decrement;
32 }
33 GreenBlinking {
34 on tick if timer.value == 0
35 execute green.turnOff, red.turnOn, timer.reset
36 transitto Red;
37 on tick else
38 execute timer.decrement;
39 }
40 }
41 Inactive {
42 on enter execute red.turnOff, green.turnOff, timer.turnOff;
43 on switch transitto Active;
44 on stop transitto Final;
45 }
46 final Final {
47 }
48 }
```

На листинге 2 приведена вырезка из кода на языке программирования *Java* для поставщика событий `TrafficSignalProvider`, в которой описываются события. На листингах 3 и 4 приведены вырезки из кода для объектов управления `RedLamp` и `Timer`

соответственно, в которых описываются действия. Код объекта управления `GreenLamp` аналогичен объекту управления `RedLamp`.

Этой программе соответствуют диаграмма связей и диаграмма состояний, представленные на рис. 1 и 2 соответственно. Средства *FSML* и *UniMod* позволяют сгенерировать эти диаграммы по коду программы и автоматически расположить их на плоскости. Приведенные диаграммы оптимизированы вручную для большей наглядности.

Листинг 2. Объявление событий в поставщике событий TrafficSignalProvider

```
public class TrafficSignalProvider implements EventProvider {
 /**
 * @unimod.event.descr Full stop signal
 */
 public static final String STOP = "stop";

 /**
 * @unimod.event.descr Switch on/off the traffic-light
 */
 public static final String SWITCH = "switch";

 /**
 * @unimod.event.descr Next tick of system timer happened
 */
 public static final String TICK = "tick";
```

Листинг 3. Объявление действий в объекте управления RedLamp

```
public class RedLamp implements ControlledObject {
 /**
 * @unimod.action.descr switches the lamp on
 */
 public void turnOn(StateMachineContext context) {
 /* Обработка действия */
 }

 /**
 * @unimod.action.descr switches the lamp off
 */
 public void turnOff(StateMachineContext context) {
 /* Обработка действия */
 }

 /**
 * @unimod.action.descr switches the lamp to blinking mode
 */
 public void turnBlinking(StateMachineContext context) {
 /* Обработка действия */
 }
}
```

Листинг 4. Объявление действий в объекте управления Timer

```
public class Timer implements ControlledObject {
 /**
 * @unimod.action.descr current value of timer
 */
 public int value(StateMachineContext context) {
 /* Обработка действия */
 }

 /**
 * @unimod.action.descr decrement timer value
 */
 public void decrement(StateMachineContext context) {
 /* Обработка действия */
 }

 /**
 * @unimod.action.descr reset timer
 */
 public void reset(StateMachineContext context) {
 /* Обработка действия */
 }

 /**
 * @unimod.action.descr turn off
 */
 public void turnOff(StateMachineContext context) {
 /* Обработка действия */
 }
}
```

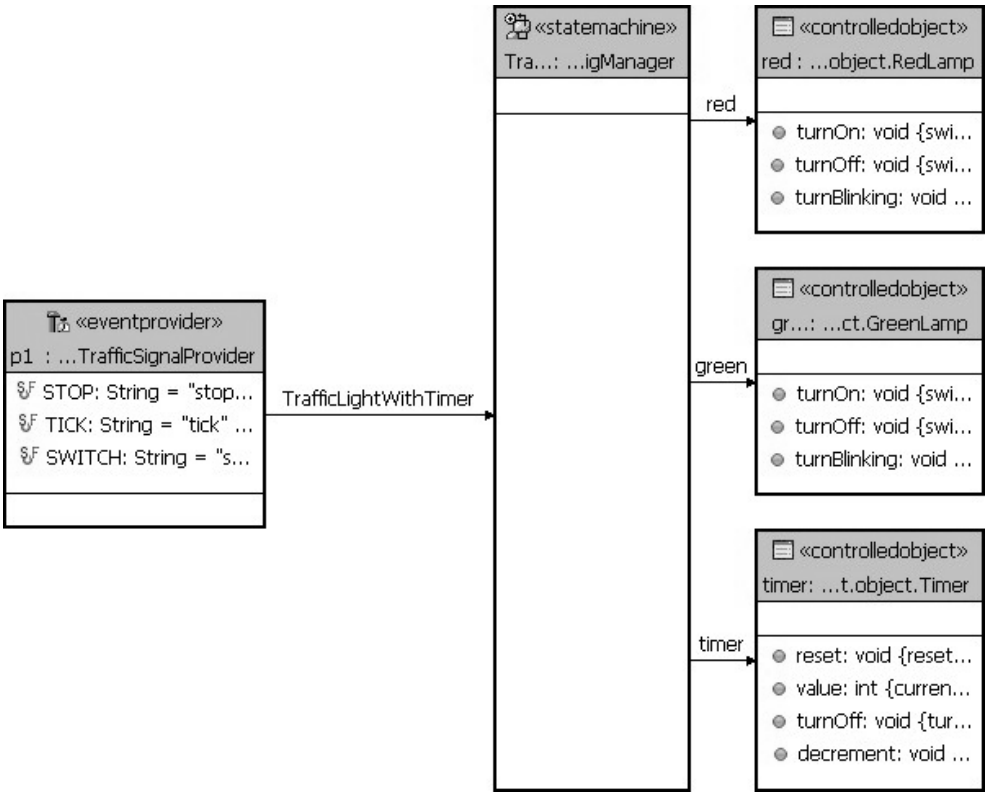


Рис. 1. Диаграмма связей для светофора

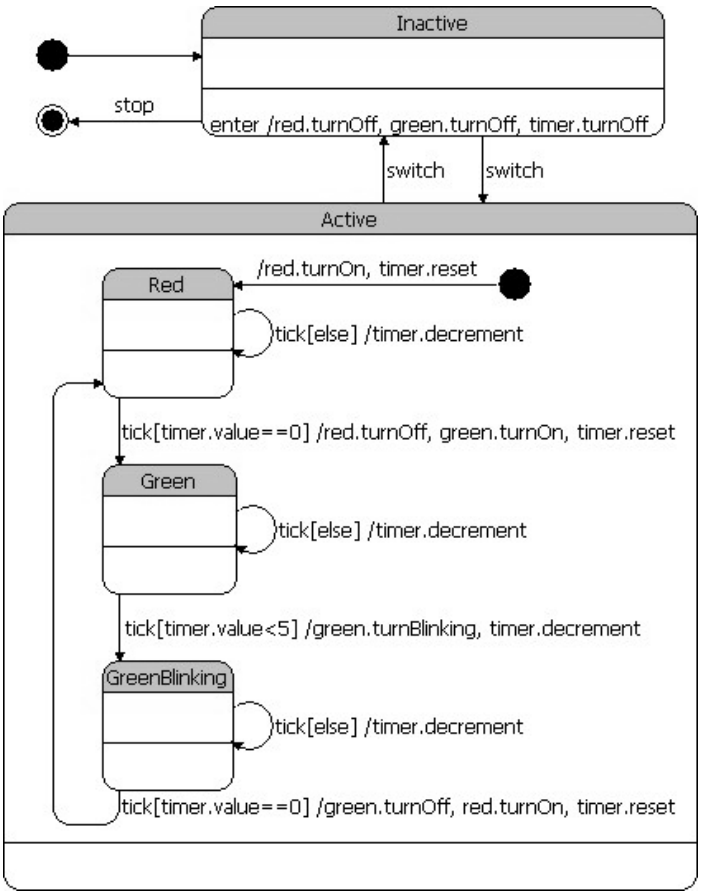


Рис. 2. Диаграмма состояний для светофора

Рассмотрим код программы на языке *FSML* более подробно. Сначала объявляется поставщик событий данной системы:

- `trafficlight.provider.TrafficSignalProvider` – он предоставляет следующие события:
  - `stop` – сигнал завершения работы системы;
  - `switch` – сигнал включения/выключения светофора;
  - `tick` – сигнал от системного таймера.

После этого объявляется автомат `TrafficLightWithTimer` и его объекты управления. Каждой из физических частей светофора соответствует свой объект управления:

- `trafficlight.object.RedLamp red` – красная лампа. Этот объект управления имеет следующий интерфейс:
  - `turnOn - включить лампу;`
  - `turnOff - выключить лампу;`
  - `turnBlinking - включить лампу в режиме мигания;`
- `trafficlight.object.GreenLamp green` – зеленая лампа. Этот объект управления аналогичен предыдущему;
- `trafficlight.object.Timer timer` – табло таймера. Этот объект управления имеет следующий интерфейс:
  - `value - получить текущее значение таймера;`
  - `decrement - уменьшить значение таймера` на единицу;
  - `reset - сбросить таймер к начальному значению;`
  - `turnOff - выключить табло таймера.`

Оставшуюся часть программы занимают описания состояний и переходов между ними. В рассмотренном примере, помимо начального (`initial Init`) и конечного (`final Final`) состояний автомата, присутствуют следующие состояния:

- `Active` – светофор включен. Это сложное состояние, содержащее основной цикл работы светофора, состоящий из четырех состояний:
  - `InitActive – начальное состояние после включения светофора;`
  - `Red – включен красный сигнал светофора;`
  - `Green – включен зеленый сигнал светофора;`
  - `GreenBlinking – включен зеленый мигающий` сигнал светофора;
- `Inactive` – светофор выключен.

Заметим, что это далеко не единственная и не лучшая реализация автомата, управляющего светофором. Можно произвести декомпозицию и выделить содержимое состояния `Active` в отдельный автомат, сделав его вложенным в это состояние. Это позволит упростить как программу на языке *FSML*, так и соответствующую диаграмму состояний.

Особенности языка

Выделим особенности данного языка автоматного программирования. Напомним, что он ориентирован на использование в инструментальном средстве *UniMod 2*.

Текстово-визуальный подход к разработке автоматных программ не реализован полноценно ни в одном из существующих инструментальных средств.

Система *MPS* [8] предлагает возможность просмотра диаграммы состояний создаваемого автомата. Однако на данный момент в ней невозможно создание графических редакторов. Это исключает возможность редактирования диаграммы состояний в виде графа. В то же время эта возможность может быть полезна для внесения быстрых изменений в структуру автомата, а текстовый ввод удобен для быстрого первоначального описания автомата. Инструментальное средство *UniMod* [4], наоборот, значительно теряет в эффективности, не позволяя редактировать автомат в текстовом представлении. Именно этот недостаток исправляет язык *FSML*.

Явное задание графа переходов позволяет гарантировать его изоморфность программе, что избавляет программиста от многих ошибок уже на этапе описания автомата. Кроме того, этого значительно облегчает проверку валидности графа переходов.

Интеграция с объектно-ориентированным кодом необходима для использования языка автоматного программирования на практике, поскольку, за исключением языка *TABP* [9], эти языки [10–14] не являются универсальными. Однако большая часть существующих автоматных языков реализует интеграцию с помощью трансляции кода программы в код на одном из языков общего назначения. При этом происходит разбор текста программы и преобразование ее в абстрактное синтаксическое дерево (АСД), по которому генерируется код на языке общего назначения (рис. 3).

Рис. 3. Стандартная схема интеграции автоматного языка с объектно-ориентированным кодом

Такая «непрозрачная» связь затрудняет разработку на этих языках, так как сгенерированный код намного сложнее читать, чем исходный код или диаграмму состояний. Подходом к выполнению этого требования выгодно отличается система *MPS*, на основе которой созданы два языка автоматного программирования. Она позволяет редактировать абстрактное представление программы, а, следовательно, модель конечного автомата в памяти. Затем эта модель может быть транслирована в любой из языков общего назначения или в графическое изображение диаграммы состояний.

Язык *FSML* совместно с инструментальным средством *UniMod* выгодно отличаются от других средств автоматного программирования. Можно выделить два главных отличия схемы на рис. 4 от схемы на рис. 3, соответствующей другим текстовым языкам автоматного программирования:

- с помощью парсера *FSMLParser* сразу строится конкретная автоматная *UniMod*-модель вместо абстрактного синтаксического дерева автоматной программы;
- существует и активно используется обратная связь с объектно-ориентированным кодом на языке *Java* – через поставщики событий и объекты управления автомата.

Эти два отличия в совокупности позволяют работать непосредственно с моделью автомата, запуская его в режиме интерпретации. В этом случае отпадает необходимость трансляции автоматного кода в код на языке общего назначения. Однако такая возможность имеется при необходимости применения компилятивного подхода.

Автоматная модель может модифицироваться через графический редактор в виде диаграмм *UniMod* и через текстовый редактор, использующий парсер языка *FSMLParser* и генератор *fsml*-программ *FSMLGenerator*.

Переиспользование компонентов кода позволяет решить проблему дублирования кода и является необходимым требованием при создании сложных систем. Однако в случае языка автоматного программирования это достаточно сильное требование, полноценная реализация которого может свести почти на нет преимущества такого языка. Такая реализация использована в языке *State Machine* [14]. В результате он является очень громоздким, требуя для каждого состояния создание отдельного класса, объявление и инициализацию дополнительных переменных. Поэтому требуется найти компромисс между удобством проектирования и удобством кодирования.

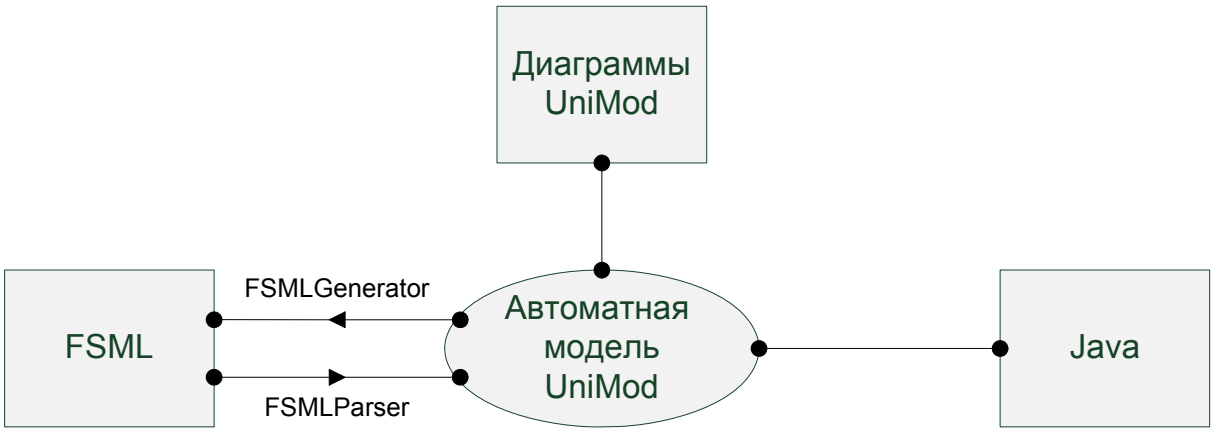


Рис. 4. Схема интеграции языка *FSML* с объектно-ориентированным кодом

В языке *FSML* существует два варианта повторного использования кода:

- вложенные автоматы позволяют повторно использовать компоненты логики;

- поставщики событий и объекты управления на языке *Java* позволяют использовать стандартные техники объектно-ориентированного программирования для повторного использования кода.

Краткость и понятность синтаксиса языка является простым, но немаловажным фактором. В целом, от него зависит эффективность использования языка автоматного программирования.

Описание редактора языка

Редактор состоит из лексического и синтаксического анализаторов, генератора объектных автоматных моделей, систем валидации и автоматического завершения ввода. В перспективе планируется добавить отладчик кода программы и генератор *fsml*-программ из объектной автоматной модели.

Рассмотрим более подробно каждый элемент. *Лексический анализатор* (*FSMLLexer*) осуществляет чтение входной цепочки символов и их группировку в элементарные конструкции, называемые лексемами. *Синтаксический анализатор* (автомат *FSML*) выполняет разбор исходной программы, используя поступающие лексемы, а также семантический анализ программы. *Генератор объектных автоматных моделей* (*FSMLToModel*) строит конечное представление автомата, сохраненного в *fsml*-программе. *Система валидации* проверяет код *fsml*-программы на наличие синтаксических и семантических ошибок. *Система автодополнения* (автоматического завершения ввода) предоставляет пользователю список строк, при добавлении которых редактируемая программа будет синтаксически верна. *Отладчик* предоставляет средства для интерактивного поиска нетривиальных семантических ошибок. *Генератор fsml*-программ осуществляет обратную связь, преобразуя отображаемые визуально объектные модели автоматов в программы на языке *FSML*. Это позволит пользователю полноценно редактировать конечные автоматы как в текстовом представлении, так и в визуальном.

Теперь рассмотрим функции редактора языка *FSML* и соответствующие им проектные решения. Структура редактора изображена на рис. 5.

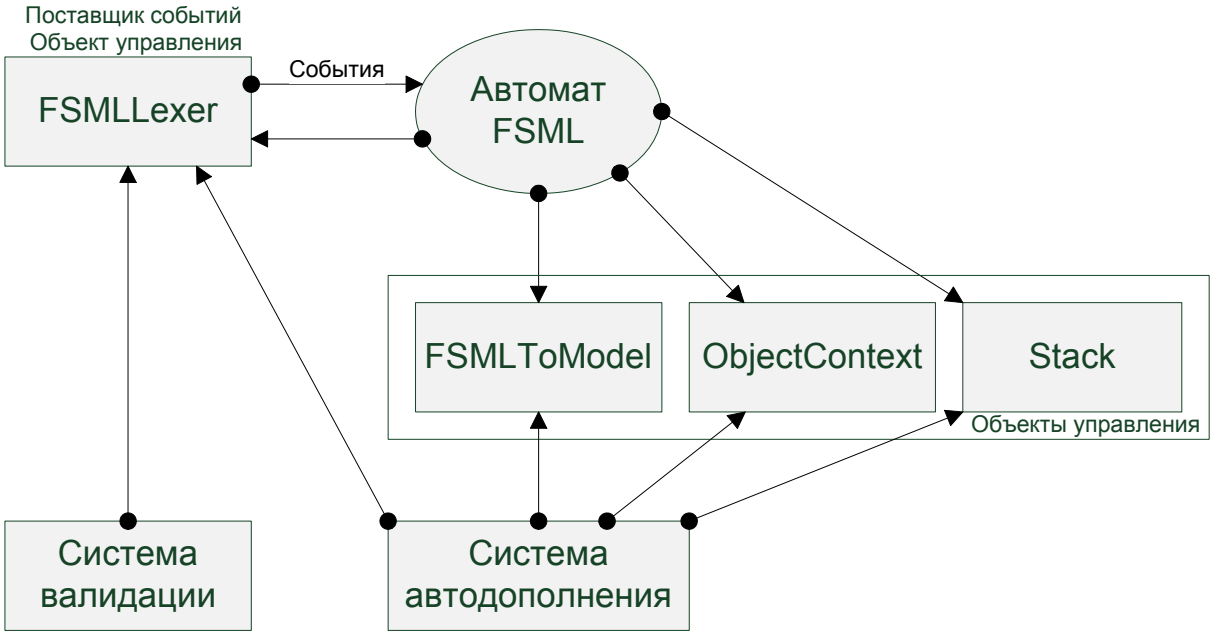


Рис. 5. Структура редактора языка *FSML*

При построении редактора языка автоматного программирования использовались автоматы.

1. Основной функцией данного редактора является *построение автоматной модели по программе на языке FSML*. Автоматная модель – это внутреннее представление диаграмм состояний автоматов и схемы их связей с поставщиками событий и объектами управления. Для реализации этой функции редактор языка *FSML* включает в себя синтаксический анализатор (автомат *FSML*) и генератор объектной автоматной модели (*FSMLToModel*). Эти средства естественно представляются в виде событийной системы. Поэтому было решено реализовать их на основе автоматного подхода с использованием пакета *UniMod* [4]. Синтаксический анализатор – это система, которая получает события в виде поступающих лексем и управляет генератором объектной модели. Таким образом, синтаксический анализатор реализован в виде конечного автомата *FSML*, лексический анализатор – в виде поставщика событий-лексем *FSMLLexer*, генератор объектной автоматной модели – в виде объекта управления *FSMLToModel* для этого автомата. Для выполнения рассматриваемой функции автомату также требуются вспомогательные данные, реализованные в виде объектов управления *ObjectContext* и *Stack*.
2. Дополнительная функция редактора – *валидация программы*. Она позволяет автоматически находить ошибки в программе. Эта система использует данные, генерируемые лексическим и синтаксическим анализаторами, поэтому для нее достаточно единственной зависимости – от объекта *FSMLLexer*.
3. Система автодополнения обеспечивает *автоматическое завершение ввода пользователя*. Автоматный подход значительно упрощает реализацию этой функции. Для этого используется построенный автомат, который позволяет получить набор ожидаемых лексем в каждом состоянии. Поэтому система автодополнения имеет зависимости от объектов управления автомата, хранящих необходимые данные. Кроме того, для обработки ошибок автомат дополняется всеми недостающими переходами с помощью алгоритма, предложенного в работе [15].

Заключение

Итак, создан текстовый язык *FSML* для представления конечных автоматов и реализован редактор этого языка *FSMLEditor*, предназначенный для использования совместно с инструментальным средством *UniMod*. В совокупности эти две компоненты имеют существенные преимущества перед существующими средствами автоматного программирования:

- текстово-визуальный подход к описанию конечных автоматов;
- естественное сочетание автоматного и объектно-ориентированного подходов при разработке программ;
- современные средства для работы с кодом, такие как валидация ошибок и автоматическое завершение ввода.

Отметим, что текстовый редактор *FSMLEditor* реализован с использованием инструментального средства *UniMod*. Таким образом, на основе автоматного подхода создан эффективный инструмент автоматного программирования.

Литература

1. Шалыто А.А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. – СПб: Наука, 1998. – Режим доступа: http://is.ifmo.ru/books/switch/1

2. Шалыто А.А., Туккель Н.И. SWITCH-технология – автоматный подход к созданию программного обеспечения «реактивных» систем // Программирование. – 2001. – № 5. – С. 45–62. – Режим доступа: http://is.ifmo.ru/works/switch
3. Шалыто А.А. Парадигма автоматного программирования / Международная научно-техническая мультиконференция «Проблемы информационно-компьютерных технологий и мехатроники». Материалы международной научно-технической конференции «Многопроцессорные вычислительные и управляющие системы». Т. 1. – Таганрог: НИИМВС, 2007. – С. 191–194.
4. Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А. UML. SWITCH-технология. Eclipse // Информационно-управляющие системы. – 2004. – № 6. – С. 12–17. – Режим доступа: http://is.ifmo.ru/works/UML-SWITCH-Eclipse.pdf
5. Решетников Е. О. Инструментальное средство для визуального проектирования автоматных программ на основе Microsoft Domain-Specific Language Tools. Бакалаврская работа / СПбГУ ИТМО. – 2007.
6. Решетников Е.О. Инструментальное средство для визуального проектирования автоматных программ на основе Microsoft Domain-Specific Language Tools. Бакалаврская работа. СПбГУ ИТМО. 2007. – Режим доступа: http://is.ifmo.ru/download/StateMachineDesigner2/doc/StateMachineDesigner2.pdf
7. Среда разработки Eclipse. – Режим доступа: http://www.eclipse.org
8. Язык программирования FSML. – Режим доступа: http://unimod.sourceforge.net/wiki/index.php/FSML
9. Дмитриев С. Языково-ориентированное программирование: следующая парадигма // RSDN Magazine. – 2005. – № 5.
10. Дмитриев С. Языково-ориентирвоанное программирование: следующая парадигма. – Режим доступа: http://www.rsdn.ru/article/philosophy/LOP.xml
11. Цымбалюк Е.А. Текстовый язык автоматного программирования TABP. Магистерская диссертация / СПбГУ ИТМО. 2008.
12. Язык AsmL. – Режим доступа: http://research.microsoft.com/fse/asml
13. Язык SMC. – Режим доступа: http://smc.sf.net
14. Степанов О.Г., Шалыто А.А., Шопырин Д.Г. Предметно-ориентированный язык автоматного программирования на базе динамического языка Ruby // Информационно-управляющие системы. – 2007. – № 4. – С. 22–27. – Режим доступа: http://is.ifmo.ru/works/_2007_10_05_aut_lang.pdf
15. 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. – Режим доступа: http://is.ifmo.ru/works/_2007_10_05_mps_textual_language.pdf
16. Шамгунов Н.Н. Разработка методов проектирования и реализации поведения программных систем на основе автоматного подхода. Диссертация … канд. техн. наук / СПбГУ ИТМО. – 2004. – Режим доступа: http://is.ifmo.ru/disser/shamg_disser.pdf
17. Гуров В.С., Мазин М.А. Создание системы автоматического завершения ввода с использованием пакета UniMod // Вестник II Межвузовской конференции молодых ученых. Т.1. – СПбГУ ИТМО, 2005. – С. 73–87.

8 ТЕСТИРОВАНИЕ ПРИЛОЖЕНИЙ И КОНЕЧНЫЕ АВТОМАТЫ

УДК 004.4’233

МЕТОД РАЗРАБОТКИ ТЕСТОВ ДЛЯ ПРОГРАММНЫХ ИНТЕРФЕЙСОВ ПРИЛОЖЕНИЙ НА ОСНОВЕ КОНЕЧНО-АВТОМАТНОЙ МОДЕЛИ ТЕСТИРОВАНИЯ

К.В. Рубинов, В.В. Веденеев, В.Г. Парфенов

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В работе описывается предложенный авторами метод разработки тестов для программных интерфейсов приложений на основе конечно-автоматной модели тестирования. Предлагаемый метод предполагает построение автоматной модели тестирования в графическом виде. После этого по этой модели с помощью обхода графа переходов строится набор тестовых сценариев. Такой подход к тестированию позволяет упростить и формализовать построение тестов не только для отдельных функций, но и для их совокупностей.

Ключевые слова: автоматное программирование, тестирование

Введение

Программные интерфейсы приложений (*Application Programming Interfaces – API*) проектируются на основе стандартов, которые поддерживают производители программного обеспечения. Соответствие стандартам контролируется и базируется на тестировании интерфейсов в соответствии со спецификациями. Все чаще для тестирования программных компонентов и приложений применяется тестирование на основе моделей (*Model-based testing*) [1, 2]. При этом используются различные модели, на базе которых разработаны разнообразные средства генерации тестов [3]. Анализ литературы показал, что модельный подход практически не применяется для тестирования *API*. К известным работам можно отнести, например, работу [4] о применении марковских моделей. Таким образом, рассмотрение других моделей в качестве базы для создания тестов программных интерфейсов приложений является весьма актуальной задачей.

В работе описывается метод разработки тестов для *API* на основе моделей. Учитывая достоинства и недостатки существующих моделей, в предложенном авторами методе для построения модели тестирования выбраны конечные автоматы, которые позволяют воспользоваться преимуществами теории автоматов для новой задачи.

Отличительной особенностью метода является то, что предлагается строить не модель поведения системы, а модель тестирования использования программных интерфейсов. При этом в качестве состояний выделяются тестируемые функции (методы), а в качестве событий (условий переходов) рассматриваются результаты выполнения функций. Для автоматизации задач, возникающих при разработке тестов для *API*, в работе используются существующие подходы преобразования моделей конечных автоматов в *XML*-формат с последующей генерацией из полученного представления тестовых сценариев и исходного кода тестового приложения. Предложенный метод разработки тестов позволяет решить задачу их создания для *API*, а также автоматизировать некоторые из этапов решения этой задачи. Применение метода демонстрируется на примере.

Постановка задачи

В работе рассматривается процесс разработки тестов для функционального тестирования программных интерфейсов приложений (далее интерфейсов). Эти интерфейсы

предназначены для предоставления другим программам функциональности программной системы, в которой они реализованы.

Для проверки соответствия интерфейса спецификации необходимо выполнить следующие виды тестирования.

1. Синтаксическое тестирование отдельных функций интерфейса. Проверка отдельных функций на широком спектре входных данных.
2. Проверка корректности предоставления функциональности отдельных функций интерфейса.
3. Проверка корректности предоставления функциональности совокупностью функций интерфейса.
4. Проверка некорректного применения отдельных функций интерфейса.
5. Проверка некорректного использования совокупности функций интерфейса.
6. Проверка взаимодействия при интеграции функций интерфейса с другими интерфейсами или программными компонентами.

Особый интерес из перечисленных выше видов тестирования представляет проверка корректности предоставления функциональности совокупностью функций интерфейса, так как на данный момент для него не существует формализованного метода разработки тестов.

В работе описывается метод разработки тестов для совокупностей функций интерфейсов. Предлагается конечно-автоматная модель тестирования, а также рассматривается вопрос о тестовом покрытии на основании предложенной модели.

Результатом применения описываемого метода является приложение, которое тестирует функции программного интерфейса в соответствии со спецификацией.

Спецификация API

Процесс тестирования *API* начинается с изучения и анализа заданной спецификации тестируемого интерфейса. Обычно спецификации к интерфейсам описывают функциональное назначение отдельных функций (методов) интерфейса, а также ограничения на используемые через интерфейс данные.

Для процесса тестирования, базирующегося на спецификации, критерием тестового покрытия является разработка такого числа тестов, чтобы был создан как минимум один тест для каждого требования из спецификации.

При успешном прохождении определенного числа тестов, которое удовлетворяет заранее заданному значению, можно утверждать, что система прошла аттестационное/сертификационное тестирование (*conformance/certification testing*).

Рассмотрим процесс тестирования *API* на примере.

Пример спецификации API

В качестве примера спецификации *API* рассмотрим часть программного интерфейса операционной системы, предназначенного для работы с файлами [5]. Для тестирования выделим минимальный набор операций, необходимых для чтения/записи файла: открытие файла с необходимым уровнем доступа, закрытие файла, позиционирование по файлу, чтение и запись последовательности байт с определенного места в открытом файле.

Приведем функции, отвечающие в спецификации за эти операции, в упрощенной нотации и кратко опишем их функциональность.

Функция

```
HANDLE CreateFile(…)
```

позволяет открывать или создавать файл и возвращает указатель на него. Этот указатель необходимо передавать во все функции, которые работают с открытым файлом. Если при его открытии произошла ошибка, то вместо корректного указателя на файл возвращается зарезервированный код ошибки. В качестве параметров функция принимает название файла, желаемый уровень доступа и т.д.

Функция

```
BOOL CloseHandle(HANDLE)
```

позволяет закрывать файл. В случае успеха возвращается значение `TRUE`, иначе – `FALSE`. Единственным параметром является указатель на открытый файл.

Функция

```
DWORD SetFilePointer(HANDLE, …)
```

позволяет установить указатель на место начала чтения или записи в открытом файле. Если операция позиционирования прошла успешно, то возвращается указатель на место в файле, иначе возвращается отрицательное число. В качестве параметров функция принимает указатель на открытый файл и параметры, определяющие новое место в файле.

Функция

```
BOOL ReadFile(HANDLE, …)
```

позволяет читать данные из открытого файла. В случае успеха возвращается значение `TRUE`, иначе – `FALSE`. В качестве параметров функция принимает указатель на открытый файл, параметры, определяющие адрес и размер буфера для записи прочтенных данных, и число байт для чтения.

Функция

```
BOOL WriteFile(HANDLE, …)
```

позволяет записывать данные в открытый файл. В случае успеха возвращается значение `TRUE`, иначе – `FALSE`. В качестве параметров функция принимает указатель на открытый файл, параметры, определяющие адрес и размер буфера для чтения записываемых данных, и число байт для записи.

Анализ спецификации API

После изучения спецификации перейдем к следующему этапу – инженер по тестированию проводит анализ спецификации и предметной области с целью выделения возможных вариантов совместного использования совокупности функций интерфейса. При этом формируются последовательности вызовов функций. Из этих последовательностей в дальнейшем будут формироваться тестовые сценарии. На основании анализа зависимостей в спецификации функции программного интерфейса можно разбить на группы по типу выполняемой функциональности.

Проведем неформальный анализ описанного выше примера спецификации программного интерфейса. Рассмотрим операции открытия и закрытия файла.

Для тестирования функции `CreateFile` в наиболее простом случае необходимо выполнить два теста:

- успешно открыть файл;
- получить ошибку при открытии файла.

Учитывая известные методики тестирования [6], число этих тестов необходимо расширить. Например, можно выделить следующие ситуации, требующие проверки:

- создать файл с названием, содержащим недопустимые символы;
- создать файл с названием в однобайтной или двухбайтной кодировке;
- создать файл по несуществующему пути;
- создать существующий файл;
- открыть уже открытый файл и т. д.

Для тестирования функции `CloseHandle` необходимо провести следующие тесты:

- успешно закрыть файл;
- получить ошибку при закрытии файла.

Составим тестовый сценарий для успешного закрытия файла. Для этого в функцию `CloseHandle` необходимо передать указатель на открытый файл. Однако этот указатель можно получить только с помощью еще непротестированной функции `CreateFile`.

Отложим написание тестового сценария для функции `CloseHandle` и приступим к проверке функции `CreateFile`. При тестировании функции `CreateFile` необходимо иметь в виду, что при завершении теста следует закрыть открытые файлы с помощью функции `CloseHandle` для освобождения ресурсов. При этом функция `CloseHandle` еще не протестирована.

Тестовые сценарии для функций программного интерфейса неявно опираются на то, что остальные его функции работают корректно и в пределах каждого теста происходит дублирование тестов для других функций.

В результате анализа спецификации выявлены возможные варианты совместного использования совокупности функций интерфейса. Далее на основе полученных вариантов осуществляется построение модели тестирования *API*.

Модель тестирования API

Для разработки тестовых сценариев строится модель, описывающая использование функций тестируемого интерфейса. Модель необходима для формализации процесса разработки тестовых сценариев и тестовых приложений. Применение модели позволяет автоматизировать некоторые этапы процесса разработки тестов, а также в дальнейшем, при внесении изменений в спецификацию, поддерживать соответствие между спецификацией тестируемого интерфейса и тестовым приложением.

Построение модели тестирования API

В работе для построения модели выбраны конечные автоматы. Такой выбор обусловлен несколькими факторами:

- наглядное визуальное представление модели в форме конечного автомата;
- удобство отображения связей между возможными вызовами функций интерфейса в конечном автомате;
- существование широкого спектра алгоритмов, методов и средств работы с моделями в форме конечных автоматов.

Конечно-автоматная модель тестирования API

Модель системы, как отмечено выше, представляется в виде конечного автомата, в котором состояниям соответствуют вызовы функций, а условиями перехода являются результаты выполнения функций. Вызовы функций осуществляются при входе в состояния. Наличие действий в выходных воздействиях таких автоматов не является обязательным, но они могут быть. Действия могут вводиться искусственно для вызова промежуточных функций, которые не являются тестируемыми, но необходимы для поддержания тестового приложения в необходимом состоянии (например, вызов функции обновления экрана). Инициализация перед запуском отдельных функций программного интерфейса представляется в виде состояний автомата.

Поясним это на примере (рис. 1).

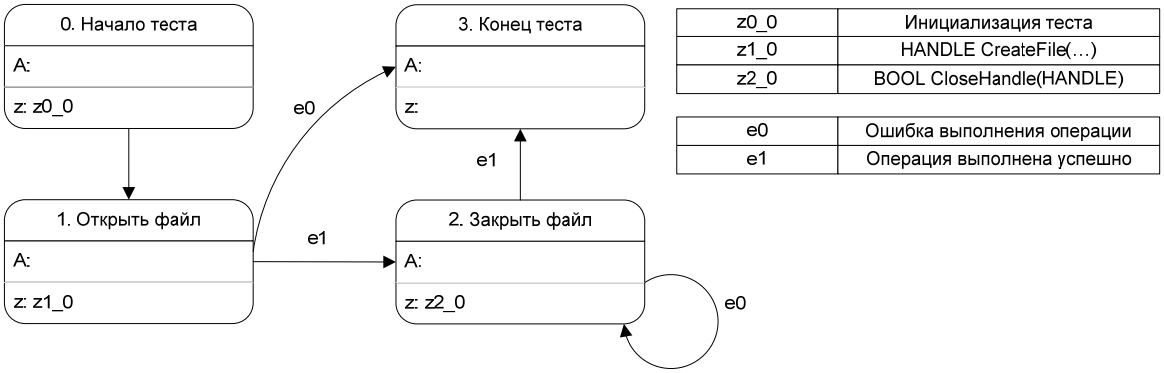


Рис. 1. Модель тестирования функций открытия и закрытия файла

Конечный автомат имеет четыре состояния. Состояния 0 и 3 соответствуют началу и концу теста. В состояниях 1 и 2 вызываются функции `CreateFile` и `CloseHandle` соответственно.

Из начального состояния 0 после проведения необходимой инициализации автомат переходит в состояние 1. В случае неуспешного открытия файла в состоянии 1 происходит окончание теста – переход в состояние 3. При успешном открытии файла автомат переходит в состояние 2. Если файл успешно закрывается в состоянии 2, то происходит окончание теста в состоянии 3, иначе автомат остается в состоянии 2.

Представив возможные зависимости между функциями программного интерфейса в виде конечного автомата, удается решить проблемы, описанные выше. Из рассмотрения графа на рис. 1 следует, что обе функции (`CreateFile` и `CloseHandle`) можно протестировать только в совокупности.

На основании построенного графа переходов можно получить необходимые тестовые сценарии: любой путь из начального состояния в конечное является тестом. Например, путь 0 – 1 – e0 – 3 является описанным выше тестом «Получить ошибку при открытии файла». Это верно и для других путей в графе.

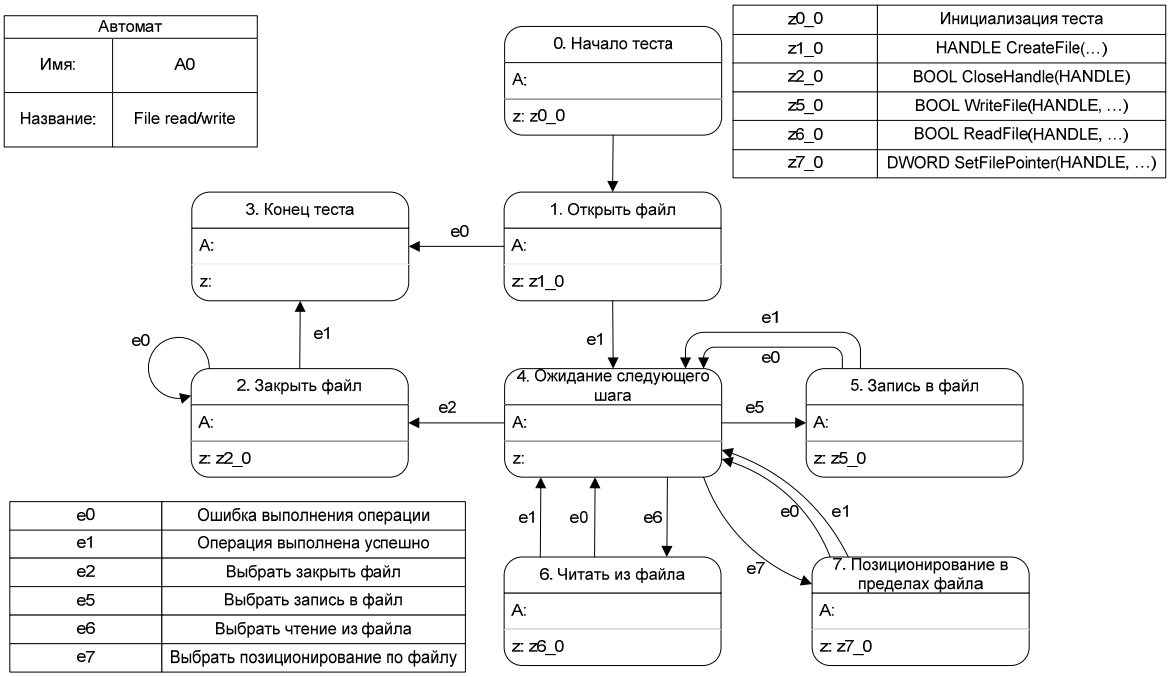


Рис. 2. Моделирование конечным автоматом использования программного интерфейса

Отметим известный факт, что число возможных путей в графе с циклами бесконечно. Поэтому необходимо использовать методы ограничения числа путей для получения приемлемого числа тестов по составленной модели.

Продолжая усложнять созданную модель, включим в нее все описанные в примере функции программного интерфейса (рис. 2).

В данном случае отметим состояние 4. Это «искусственное» состояние, введенное для упрощения модели. В этом состоянии не производится непосредственный вызов тестируемой функции, и его можно рассматривать как ветвление для создания той или иной тестовой последовательности. Из этого состояния под действием искусственных событий e2, e5, e6 и e7 автомат переходит в состояния 2, 5, 6 и 7 соответственно.

Введение искусственного состояния позволило упростить модель, увеличивая при этом число возможных тестовых сценариев.

Преобразование автоматной модели в XML-формат

Граф переходов разработанной конечно-автоматной модели предлагается хранить в формате *XML* [7]. Этот формат является одним из наиболее распространенных форматов хранения динамических данных. Следующие преимущества явились определяющими при выборе этого формата хранения данных:

- простота понимания и доступность;
- простота изменения и дополнения хранимых данных;
- простота преобразования и конвертирования хранимых данных;
- расширяемость.

Для хранения графа переходов предлагается следующая структура *XML*-документа:

- таблица состояний (*States*) – хранит описания всех состояний приложения;
- таблица событий (*Events*) – хранит описания всех возможных событий;
- таблица переходов (*Switches*) – связывает таблицу состояний и таблицу событий.

В табл. 1–3 приведены описания используемых полей.

Таблица 1. Описания полей таблицы *Events*

Поле	Описание
StateId	Уникальный идентификатор
Description	Описание сущности

Таблица 2. Описания полей таблицы *States*

Поле	Описание
StateId	Уникальный идентификатор
Description	Описание сущности

Таблица 3. Описания полей таблицы *Switches*

Поле	Описание
StateFromId	Из какого состояния осуществляется переход
StateToId	В какое состояние осуществляется переход
EventId	Под действием чего осуществляется переход

Предложенный формат является базовым. По мере необходимости он может легко расширяться и дополняться. Например, в таблицу событий можно добавить поля, отвечающие за специфические события.

Модель в виде *XML*-файла можно получить из обычного визуального представления. В качестве системы визуального проектирования воспользуемся, например, программой *Microsoft Visio 2003*. Эта программа, как и все продукты *Microsoft Office 2003*, позволяет сохранять разработанные документы в *XML*-формате. Для представления автоматов воспользуемся инструментальным средством *Visio2Switch* [8].

С помощью *XSL*-трансформации [9] документ, сохраненный программой *Visio*, преобразуется в требуемый формат.

Далее будет показано, как на основе разработанной модели можно автоматизировать последующие этапы процесса разработки тестов для *API.*

Тестовое приложение

На данном этапе происходит формирование тестовых сценариев и создание каркаса тестового приложения. На основе разработанной модели тестирования автоматизируется процесс создания приложения для тестирования *API*.

Формирование тестовых сценариев

Как было отмечено выше, каждый тестовый сценарий является некоторым путем в графе переходов конечного автомата, представляющего модель использования программного интерфейса. Начальная вершина пути – это состояние приложения перед началом теста. Путь по вершинам графа – это последовательность вызовов функций во время проведения теста. Конечная вершина пути – это ожидаемый результат проделанных действий. Таким образом, можно рассматривать модель как первичную сущность, а элементарный тест – как вторичную. В этом случае изменение спецификации программного интерфейса отразится только на изменении модели, а тестовые сценарии будут обновлены по модели.

Набор тестовых сценариев можно создать автоматически на основе разработанной модели – достаточно произвести обход графа модели. Задача поиска путей в графе описана в ряде источников (например, в работе [10]). При этом для создания тестовых сценариев и преобразования графа переходов, представленного в *XML*-формате, повторно выполняется *XSL*-трансформация. Так как число возможных сценариев в общем случае бесконечно, у разработанной трансформации есть два ограничивающих параметра: максимальное число проходов по циклу и максимальная длина сценария.

Помимо этого, в рамках настоящего метода можно опитимизировать создание тестовых сценариев – задача поиска минимального числа тестовых сценариев, наиболее полно покрывающих функциональность, является задачей, которая может решаться отдельно. Решения этой задачи описаны, например, в работе [6]. Тем самым можно говорить о том, что предлагаемый метод не ограничивает использование вновь появляющихся технологий и алгоритмов.

Каркас тестового приложения

Используя описанную выше модель и тестовые сценарии, создается тестирующая программа. Так же, как и на предыдущих этапах, это выполняется с помощью *XSL*-трансформации. Для этого разрабатываются конфигурационные файлы, с помощью которых осуществляется переход от *XML*-представления тестовых сценариев к представлению на используемом языке программирования – создается каркас тестового приложения.

XSL-трансформацию можно легко модифицировать для использования необходимого языка программирования и конкретной среды для запуска тестов. Как и другие этапы автоматизации, этот этап является независимым.

Созданный каркас приложения содержит заглушки функций, которые необходимо реализовать вручную инженеру по тестированию. Каждая функция проверяет генера-

цию конкретного события в определенном состоянии. Эти функции могут быть реализованы один раз и впоследствии использоваться при изменении модели.

Последним шагом подготовки тестового приложения является насыщение его каркаса данными для тестирования. Они выбираются по спецификации в соответствии с используемыми техниками тестирования. Задачи выбора данных для тестирования изложены во многих работах (например, в работе [6]).

Тестовые сценарии запускаются из меню приложения в автоматическом режиме. Результатом выполнения приложения является вердикт: тест прошел – положительный результат (`PASSED`) или тест не прошел – отрицательный результат (`FAILED`).

Заключение

Предложенный метод позволяет разрабатывать тесты для программных интерфейсов приложений на основе спецификаций. При этом увеличивается вероятность обнаружения неисправностей, не только связанных со свойствами отдельных функций, но и с общей функциональностью системы, предоставляемой через интерфейс.

Использование конечно-автоматной модели обеспечивает прозрачность структуры и поведения разрабатываемых тестовых приложений. Кроме того, применение модели позволяет автоматизировать определенные этапы создания тестов.

В рамках настоящего метода имеются возможности для улучшений и будущих исследований. Наиболее интересными представляются следующие направления работ:

- извлечение паттернов использования *API* из существующих приложений и репозиториев кода [11] и применение их в методе создания тестов;
- оптимизация обхода модели тестирования с использованием различных алгоритмов.

Литература

1. El-Far I.K., Whittaker J. Model-Based Software Testing. Encyclopedia on Software Engineering (edited by Marciniak J.). – Wiley. 2001.
2. DACS (Data and Analysis Center for Software) Gold Practices Website. Model-based testing. – Режим доступа: https://www.goldpractices.com/practices/mbt/
3. Hartman A. AGEDIS model based test generation tools. – Режим доступа: http://www.agedis.de/documents/ModelBasedTestGenerationTools.pdf
4. Jorgensen A., Whittaker J. An API Testing Method /Proceedings of the International Conference on Software Testing Analysis & Review (STAREAST 2000). Software Quality Engineering. Orlando. 2000.
5. File Management Functions, MSDN. – Режим доступа: http://msdn2.microsoft.com/en-us/library/aa364232.aspx
6. Бейзер Б. Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем. – СПб.: Питер. 2004.
7. XML Developer Center. – Режим доступа: http://msdn.microsoft.com/xml
8. Головешин А. Конвертор Visio2Switch. – Режим доступа: http://www.geocities.com/goloveshin/v2s_rus.htm
9. XSL Transformations. – Режим доступа: http://www.w3.org/TR/xslt
10. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы. Построение и анализ. – М.: МЦНМО. 2001.
11. Acharya M., Xie T., Pei J., Xu J. Mining API patterns as partial orders from source code: from usage scenarios to specifications / In Proc. ESEC/FSE 2007. – Режим доступа: http://people.engr.ncsu.edu/txie/publications/esecfse07.pdf

9 АВТОМАТНОЕ ПРОГРАММИРОВАНИЕ МОБИЛЬНЫХ РОБОТОВ

УДК 004.896

ПОСТРОЕНИЕ СИСТЕМЫ АВТОМАТИЧЕСКОГО УПРАВЛЕНИЯ МОБИЛЬНЫМ РОБОТОМ НА ОСНОВЕ АВТОМАТНОГО ПОДХОДА

В.О. Клебан, В.Г. Парфенов, А.А. Шалыто

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В настоящей работе описывается подход к созданию системы автоматического управления мобильным роботом на основе автоматного подхода. Автоматное программирование применяется для написания программного обеспечения робота КВАРК-М на всех трех уровнях: на верхнем уровне применяется три автомата, на среднем – пять, а на нижнем – одиннадцать. Такой подход позволяет существенно повысить качество программного обеспечения роботов.

Ключевые слова: автоматное программирование, мобильный робот, автоматическое управление

Для построения надежного программного обеспечения (ПО) целесообразно использовать технологию автоматного программирования [1], в которой, в частности, предлагается строить программу как систему автоматов, взаимодействующих между собой за счет *вложенности и вызываемости.* Использование автоматного подхода при создании ПО обладает рядом достоинств: документируемость [2], возможность верификации [3], упрощение внесения изменений и т.д.

Рассмотрим в качестве примера применения автоматного программирования проектирование модуля свободного движения мобильного робота *КВАРК-М* (рис. 1)[4].

Модуль свободного движения должен осуществлять случайные перемещения мобильного робота, при этом контролируя наличие препятствий на пути робота.

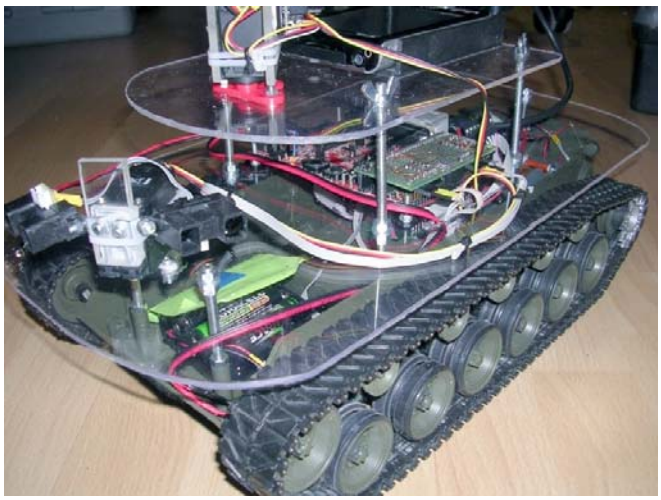


Рис. 1. Мобильный робот *КВАРК-М*

Разделим модуль свободного движения на три составляющих: модуль датчиков, модуль двигателей, расчетный модуль.

Рассмотрим *модуль датчиков*. В качестве сенсоров для определения препятствий выступает пара индикаторов дальности смонтированных на поворотной платформе. Поворот платформы регулируется с некоторой дискретностью.

Задача рассматриваемого модуля – обеспечить робота как можно более точными данными о наличии препятствий на маршруте. При решении этой задачи стало ясно, что поворотный стол должен вращаться не с постоянной скоростью, а изменять ее в зависимости от наличия препятствий. Так, например, при наличии препятствия справа необходимо усилить контроль правой части обзора. При этом для поворотного стола можно выделить четыре режима (состояния) вращения: влево, вправо, влево точно, вправо точно. Автомат управления этим объектом будет иметь одноименные состояния.

Пусть переменные `x1` и `x2` обозначают соответственно помеху слева или справа, а переменная `x3` будет устанавливаться в ноль, когда платформа повернута вправо, и в единицу, когда она повернута влево. Выделим также событие `e0`, соответствующее достижению платформой крайнего положения. Это воздействие используется как событие, так как это воздействие импульсное. Для указанных состояний, переменных и события построим управляющий автомат *A*, граф переходов которого приведен на рис. 2.

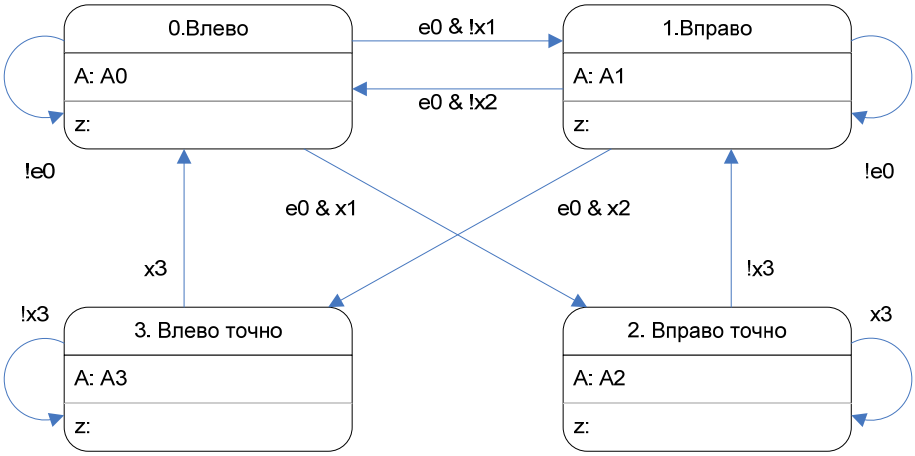


Рис. 2. Граф переходов автомата управления поворотным столом

В *i*-ое состояние этого автомата вложим автомат *Ai* (*i* = 0,…,3), реализующий действия, необходимые для обеспечения работы объекта управления в этом состоянии.

Результат работы модуля датчиков приведен на рис. 3.

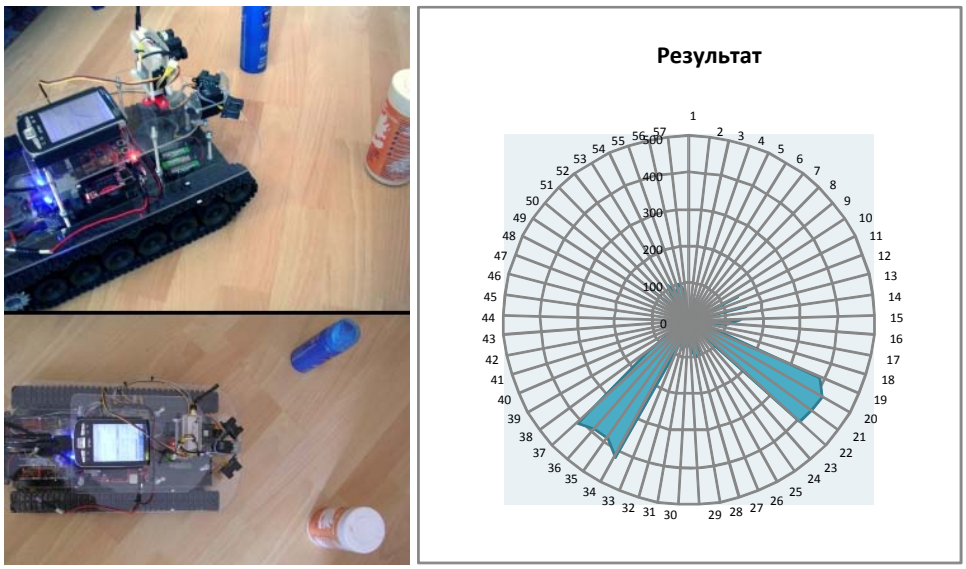


Рис. 3. Результат работы модуля датчиков

Рассмотрим *модуль обеспечения передвижения*. Робот приводится в движение с помощью пары коллекторных двигателей, которые оборудованы импульсными энкодерами. Задачей данного модуля является отслеживание пройденного роботом пути и скорости его передвижения.

Рассмотрим автомат *A1,* реализующий функцию отслеживания пройденного пути (рис. 4). Задачей данного автомата является измерения пройденного пути роботом и остановка по завершению задания.

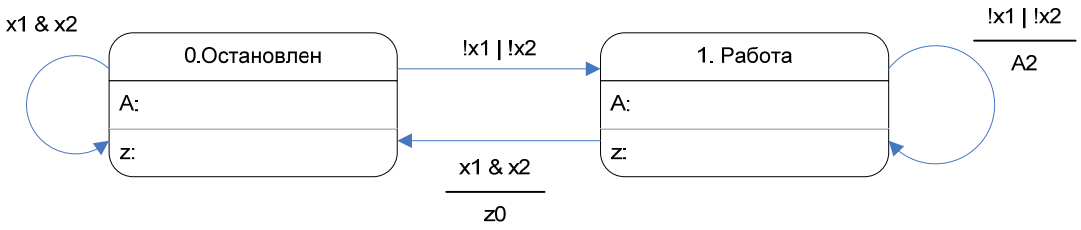


Рис. 4. Автомат *A1* контроля пройденного пути

В данном автомате использованы следующие переменные: `x1`, `x2` обозначают, достигнут ли задатчик пройденного пути для левого и правого борта робота соответственно. Выходное воздействие `z0` – останов двигателей. В состоянии «Работа» при переходе по петле (`!x1 | !x2`) происходит вызов автомата *A2*.

Построим автомат *A2,* регулирующий скорость вращения вала двигателя (рис. 5). Входными воздействиями в данном случае являются: *x0* – текущая скорость двигателя, *x1* – задатчик необходимой скорости двигателя. Выходными воздействиями *z0.z3* являются непрерывные законы управления применимые для каждого из состояний автомата. Переключение законов управления осуществляется автоматом, образуя, так называемый, гибридный автомат (гибридную систему) [5].

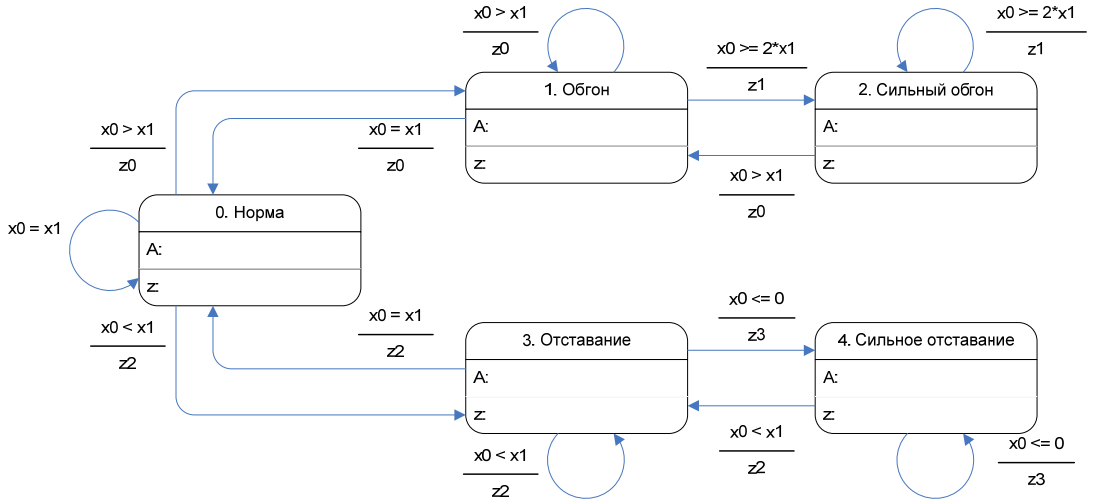


Рис. 5. Автомат стабилизации скорости вращения вала двигателя

Рассмотрим *расчетный модуль,* задачей которого является выдача команд на передвижение мобильного робота. Для обеспечения случайных, но «осмысленных» движений робота построим вероятностный конечный автомат (рис. 6). Данный автомат в случае отсутствия препятствий с вероятностью 0.6 будет выдавать роботу команду на

движение вперед, с вероятностями 0.2 влево либо вправо. Входными воздействиями данного автомата являются: `x0` – присутствует препятствие, `x1` – операция перемещения завершена.

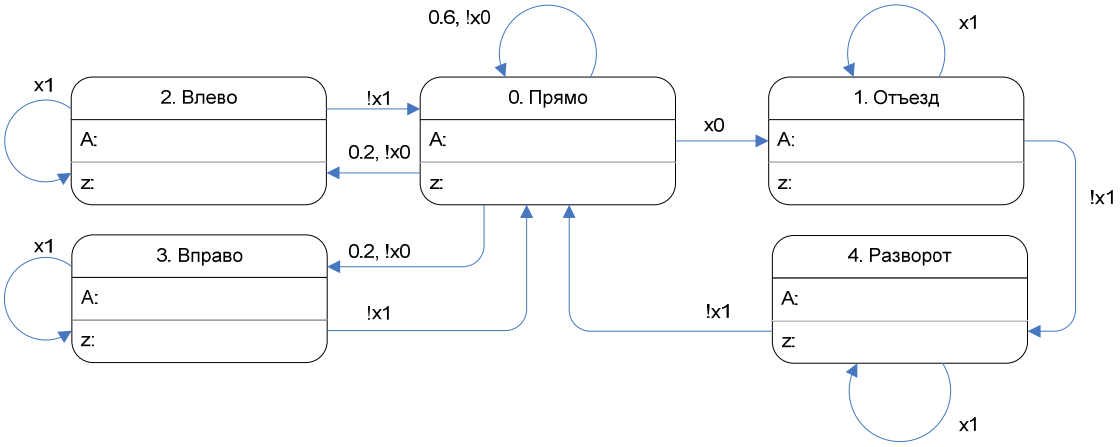


Рис. 6. Вероятностный автомат расчетного модуля

Полученные автоматы объединяются в иерархическую систему взаимодействующих автоматов.

В настоящее время программный комплекс управления мобильным роботом *КВАРК-М* является трехуровневой системой. Верхний (первый) уровень (автоматизированное рабочее место оператора телеуправления) реализован на переносном персональном компьютере с использованием беспроводной связи *Wi-Fi*. Средний (второй) уровень (бортовой компьютер) обеспечивает автономное управление, например, при отключении телеуправления. Он реализован на основе карманного персонального компьютера. Нижний (третий) уровень (периферийные модули), реализованный на контроллерах, например, *AT91SAM7P256*. Эти модули управляют различными узлами робота.

Программирование всех уровней системы выполнялось на основе автоматного подхода. При этом на верхнем уровне применялось три автомата, на среднем – пять, а на нижнем – одиннадцать (два из них – гибридные, а четыре – однотипные). При помощи гибридных автоматов осуществляется контроль и управление скоростью вращения гусениц (используется пять законов непрерывного управления).

Дальнейшее развитие методов автоматного программирования применительно к проектированию ПО мобильных роботов связано с применением автоматизированных сервисов для обеспечения повторного использования модулей ПО, которые также реализуются с помощью конечных автоматов [6].

В заключение работы отметим, что применение автоматов в мобильных роботах, как и в работе [7], позволяет резко повысить качество ПО.

Литература

1. Шалыто А. А. Switch-технология. Алгоритмизация и программирование задач логического управления. – СПб: Наука, 1998. – Режим доступа: http://is.ifmo.ru/books/switch/1
2. Сайт по автоматному программированию. Разделы «Проекты» и «UniMod-проекты». – Режим доступа: http://is.ifmo.ru

3. Сайт по автоматному программированию. Раздел «Верификация». – Режим доступа: http://is.ifmo.ru
4. Клебан В.О. Мобильный робот КВАРК-М. – Режим доступа: http://quark-bot.blogspot.com
5. Сениченков Ю.Б., Колесов Ю.Б. Моделирование систем. Динамические и гибридные системы. – СПб: БХВ-Петербург, 2005.
6. Клебан В.О., Шалыто А.А. Автоматизированные сервисы и мобильные роботы / Настоящий сборник.
7. Brooks R.A. A Robust Layered Control System for a Mobile Robot // IEEE Journal of Robotics and Automation. – 1986. – 2. – P.14–23.

10 ПРИМЕНЕНИЕ КОНЕЧНЫХ АВТОМАТОВ В ДОКУМЕНТООБОРОТЕ

УДК 004.4’233

ПРИМЕНЕНИЕ КОНЕЧНЫХ АВТОМАТОВ В ДОКУМЕНТООБОРОТЕ

В.О. Клебан, Ф.А. Новиков
(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В работе рассматривается задача построения системы автоматизации документооборота. Для построения системы автоматизации документооборота применяются конечные автоматы, так как такой подход имеет ряд преимуществ по сравнению с традиционным. В качестве примера применения предлагаемого подхода рассматривается построение инструментального программного средства на базе *Microsoft Office*, которое предназначено для разработки и внедрения автоматизированной системы менеджмента качества в компаниях с *проектным* типом организации производства.
Ключевые слова: конечный автомат, документооборот

Введение

Проблема автоматизации бизнес-процессов на сегодняшний день достаточно остро стоит перед предприятиями. Многие крупные предприятия уже обзавелись системами электронного документооборота, тогда как малые и средние – практически нет и продолжают использовать разрозненные документы *Microsoft Office* (в редких случаях с использованием распределенного хранилища *Microsoft Groove*).

Традиционный способ автоматизации бизнес-процессов – разработка прикладного программного обеспечения – постепенно вынужден отходить на второй план ввиду того, что внесение даже небольших изменений в схему бизнес-процесса означает необходимость перепрограммирования и большие затраты времени и средств. В результате прикладные программы не успевают обновляться в том темпе, который диктуют изменяющиеся условия бизнеса и потребности самого предприятия.

Активно развивающийся бизнес, связанный с автоматизацией предприятий, требует большого количества обученных кадров из-за больших объемов работ. При этом количество квалифицированных специалистов в области автоматизации растет недостаточно быстро.

Таким образом, стоит задача создания простого в освоении, надежного средства автоматизации, имеющее в своем арсенале не только средства описания документооборота, но и его исполнения. Возможность исполнения является ключевым моментом, ведь чисто описательное средство интересно только с точки зрения анализа бизнес-процессов и может быть применено при реализации конкретной модели документооборота лишь как часть технического задания. Во всех известных авторам случаях «описательная» часть системы автоматизации документооборота оторвана от «исполнительной» части, что делает невозможным быстрый переход от спроектированной схемы к реализации.

Существующие технологии автоматизации бизнес-процессов

Business Process Execution Language (*BPEL*) – язык на основе *XML* для формального описания бизнес-процессов и протоколов их взаимодействия между собой [1]. *BPEL* расширяет модель взаимодействия веб-сервисов и включает в эту модель под-

держку транзакций. Язык *BPEL* определяет два вида процессов – абстрактный и исполняемый. Абстрактный процесс определяет протокол обмена сообщениями между различными участниками, не раскрывая алгоритмы их внутреннего поведения. В отличие от абстрактного, исполняемый процесс содержит в себе алгоритмы, определяющие порядок выполнения *деятельностей* (activities), назначение исполнителей, обмен сообщениями, правила обработки исключений и т.д. Главной проблемой данного языка является организация человеко-машинного взаимодействия, так как данный язык ориентирован на полностью независимые от пользователя бизнес-процессы.

IDEF0 – методология и графическая нотация, предназначенная для формализации и описания бизнес-процессов. Отличительной особенностью IDEF0 является ее акцент на соподчиненность объектов. В IDEF0 рассматривается логические отношения между работами, а не их временная последовательность. Также на диаграмме отображаются все сигналы управления. Данная модель является одной из самых заслуженных моделей и используется при организации бизнес-проектов и проектов, основанных на моделировании всех процессов – как административных, так и организационных [2]. Основной проблемой данной методики является отсутствие операционной семантики, т.е. невозможность исполнить нарисованные диаграммы.

Нотация Architecture of Integrated Information Systems (ARIS) – методология и программный продукт компании IDS Scheer для моделирования бизнес-процессов [3]. Методология ARIS рассматривается с четырех точек зрения: организационная структура, функциональная структура, структура данных и структура процессов. Для описания бизнес-процессов предлагается использовать большое количество (около 120) типов моделей, каждая из которых описывает тот или иной аспект, кроме того, пользователь может сам определять новые типы моделей. Все это делает данную методологию гибкой, но трудно воспринимаемой, а большое количество типов моделей осложняет обучение новых специалистов.

Использование конечных автоматов

Наряду с описанием процессов в виде блок-схем (схем алгоритмов), существует другой подход – автоматный [4]. Данный подход заключается в представлении процесса в виде системы взаимодействующих автоматов. Автоматы могут взаимодействовать по вложенности (один автомат вложен в одно или несколько состояний другого автомата), по вызываемости (один автомат вызывается с определенным событием из выходного воздействия, формируемого при переходе другого автомата), по обмену сообщениями (один автомат получает сообщения от другого) и по номерам состояний (один автомат проверяет, в каком состоянии находится другой автомат). Вложенность может рассматриваться как вызываемость с любым событием. Ни число автоматов, вложенных в состояние, ни глубина вложенности не ограничены. Такое представление позволяет более компактно описывать поведение программы, модуля, а в нашем случае – жизненный цикл документа либо бизнес-процесса. Компактное представление, в свою очередь, улучшает наглядность.

Конечный автомат может быть представлен в виде простейшей языковой конструкции, состоящей из одного или нескольких операторов `SWITCH`. Такой подход делает возможным формальное и изоморфное автоматическое преобразование автомата в код программы. Повышается наблюдаемость программы за счет сокращения количества наблюдаемых переменных.

Автоматные программы могут быть эффективно верифицированы методом проверки на моделях (*Model Checking*) [5], так как в таких программах управляющие со-

стояния явно выделены, а их количество обозримо. Это позволяет строить компактные модели Крипке даже для программ большой размерности.

Автоматные программы показали свою эффективность при построении «реактивных систем»[6]. Документооборот, по сути, является такой системой. Документы реагируют на действия пользователей, а бизнес-процессы на изменения в документах, то есть *система управляется событиями*.

В качестве примера использования данной технологии рассмотрим применение предлагаемого подхода к построению инструментального программного средства на базе *Microsoft Office*, которое предназначено для разработки и внедрения автоматизированной системы менеджмента качества (АСМК) в компаниях с *проектным* типом организации производства. В разработке также используется продукт *Microsoft Groove*, который позволяет организовать распределенное хранилище документов. При проектном типе организации производства *реализации* одного и того же бизнес-процесса могут отличаться в разных проектах в рамках одной организации. Автоматизированная система менеджмента качества – это частично или полностью программно реализованная система менеджмента качества определенная в стандарте ИСО 9001: 2000.

Принципы работы и средства описания

Система представляет собой инструментальное программное средство (конструктор), предназначенный для разработки и внедрения АСМК.

Определим следующие роли пользователей системы:

- разработчик – изготовитель инструментария;
- инженер – пользователь инструментария, осуществляющий автоматизацию предприятия;
- пользователь – сотрудник предприятия, использующий систему для выполнения повседневных обязанностей.

Инструментарий состоит из следующих основных средств, которые обеспечивают базовую функциональность:

- система времени выполнения (клиент и сервер);
- транслятор моделей жизненного цикла бизнес-процессов;
- транслятор моделей жизненного цикла документов.

Используя средства инструментария, инженер определяет в графическом виде модели бизнес-процессов и модели жизненных циклов документов, а также производит их стыковку с системой времени выполнения.

Моделью жизненного цикла *активного* объекта (документа или бизнес-процесса) является конечный автомат, в котором определены состояния объекта, условия перехода из одного состояния в другое и выходные воздействия (эффекты). Условие перехода является булевой формулой, в которой участвуют *входные воздействия* и *события.* Входное воздействие представляет собой функцию, как правило, оперирующую с записями о качестве (в терминах ИСО 9001) и возвращающую булево значение. Событие – это именованное сообщение, инициирующее переход автомата и поступающее на вход системы в произвольный момент времени. Событие может быть сгенерировано автоматом жизненного цикла документа, либо автоматом жизненного цикла бизнес-процесса, либо другим «поставщиком событий» в зависимости от задачи.

Рассмотрим в качестве примера процесс исправления ошибки в программном продукте, упрощенная модель которого представлена на рис. 1. На данной модели выделено два состояния: «Обнаружена ошибка» и «Ошибка исправлена». Используются два события, на которые реагирует модель: «Создан отчет об ошибке» и «Отчет об

ошибке обновлен» и условия «Ошибка исправлена», «Ошибка не исправлена». Также используется выходное воздействие «Сообщить об исправлении». Стрелками показано направление перехода.

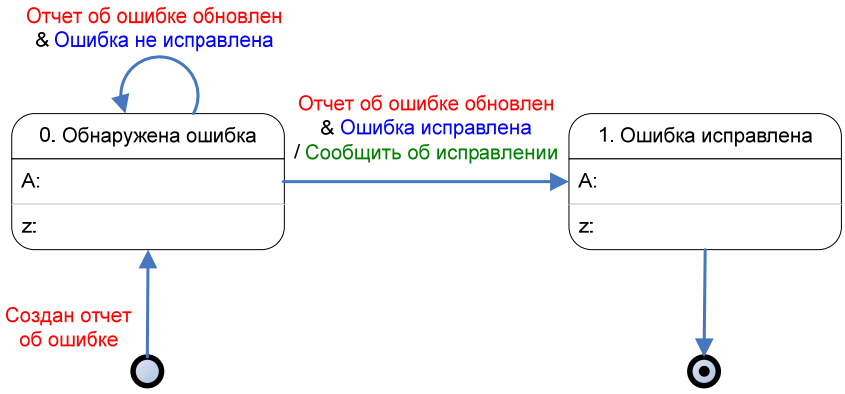


Рис. 1. Процесс исправления ошибки

Переход из состояния «Обнаружена ошибка» в состояние «Ошибка исправлена» следует читать так: при наступлении события «Отчет об ошибке обновлен» и истинности условия «Ошибка исправлена» совершить действие «Сообщить об исправлении» и перейти в состояние «Ошибка исправлена».

Как было указано ранее, активный объект системы способен реагировать на события от других объектов, а также сам порождать события (в эффектах). Например, документ при переходе из одного состояния в другое порождает событие, которое передается бизнес-процессу. Важно, что жизненный цикл документа исполняется на стороне клиента, а жизненный цикл бизнес-процесса на стороне сервера. Процесс передачи события проиллюстрирован на рис. 2.

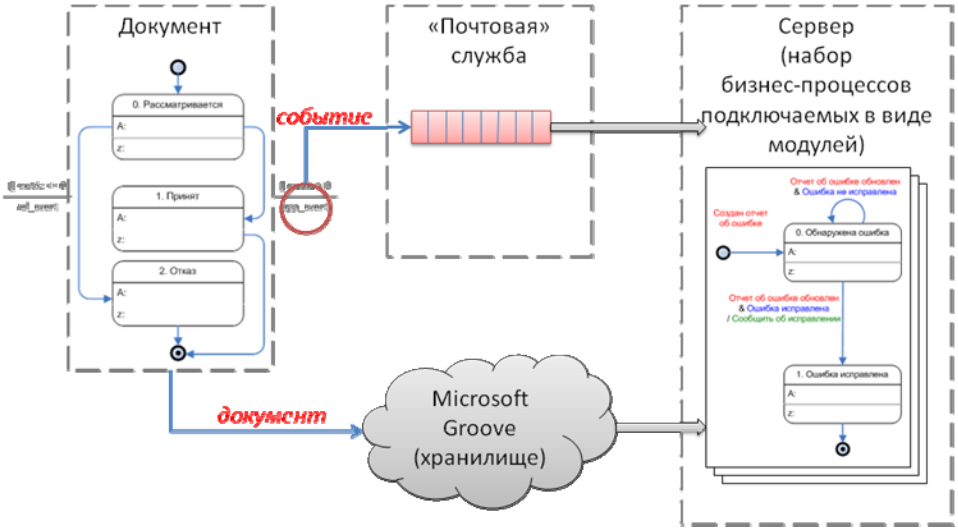


Рис. 2. Передача события из документа в бизнес-процесс

Документ при переходе из состояния «Рассматривается» в состояние «Принят» генерирует событие, которое попадает в «Почтовую службу». Она является временным хранилищем событий на стороне пользователя и сохраняет события (и порядок их следования), в том случае, если передача их на сервер затруднена или невозможна. Одно-

временно с отсылкой события на сервер происходит сохранение документа в распределенное хранилище *Microsoft Groove*. Сервер, после получения события и соответствующего ему документа, действует в соответствии с заложенными в него моделями жизненных циклов бизнес-процессов, например, извлекает из хранилища необходимый документ и анализирует, изменяет его содержимое.

Заметим, что для обеспечения функционирования АСМК в рамках организации необходимо соблюдение следующих *условий функционирования*:

- бизнес-процессы организации, подпадающие под действие АСМК, определены и документированы;
- управление бизнес-процессами отражено в электронных документах;
- выполнение бизнес-процессов основано на обработке событий;
- обрабатываемыми событиями являются события документов: создание, удаление, изменение;
- записи о качестве (метрики) хранятся в специальных полях документов.

Некоторые из данных условий справедливы не только в рамках внедрения АСМК, но и для документооборота в целом.

Структура документа

Документ делится на две составляющие: «модель» и «представление», объединенные в один файл в соответствии со стандартом *OpenXML*, который реализован в *Microsoft Office 2007*. К документу *Microsoft Office* прикреплено *XML*-хранилище, в котором хранятся служебные данные и значения записей (метрик). Подчеркнем, что данные (записи о качестве) хранятся непосредственно в документе.

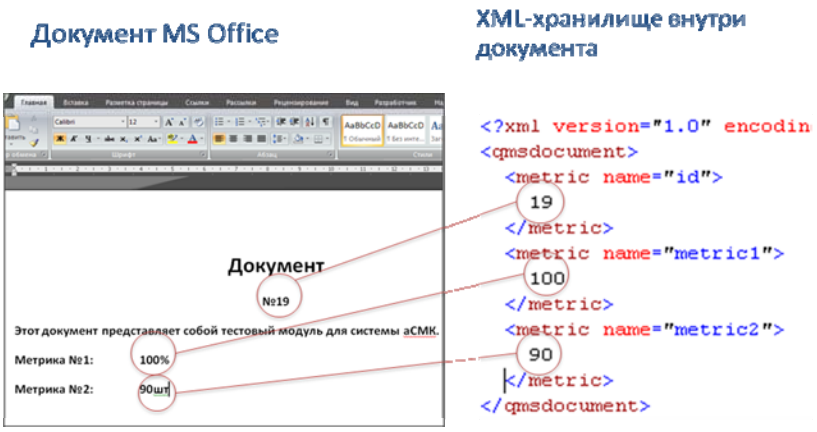


Рис. 3. Структура документа

Внутри документа также хранится его жизненный цикл, записанный в виде *XML*-представления (рис. 4).

Для возможности «отката» документа либо всей системы в предыдущее состояние в документе хранится также и история его изменения:

```
<!--История фиксирования состояний документа-->
<history>
  <state name="Review" date="22.02.2008 19:59:48" user="User">
    <docid>78ac5ada3142f3997306911d05c38dc2</docid>
    <processed>0</processed>
    <fromstate>Approved</fromstate>
    <metric name="metric1">01</metric>
    <metric name="metric2">02</metric>
    <metric name="metric3">01</metric>
  </state>
</history>
```

Также для активных объектов могут быть автоматически сгенерированы обратные автоматы при условии обратимости производимых операций [7].

Рис. 4. Модель жизненного цикла и его представление

Структура модуля бизнес-процесса

Модуль бизнес-процесса представляет собой динамическую библиотеку (plug-in), которая подключается сервером АСМК.

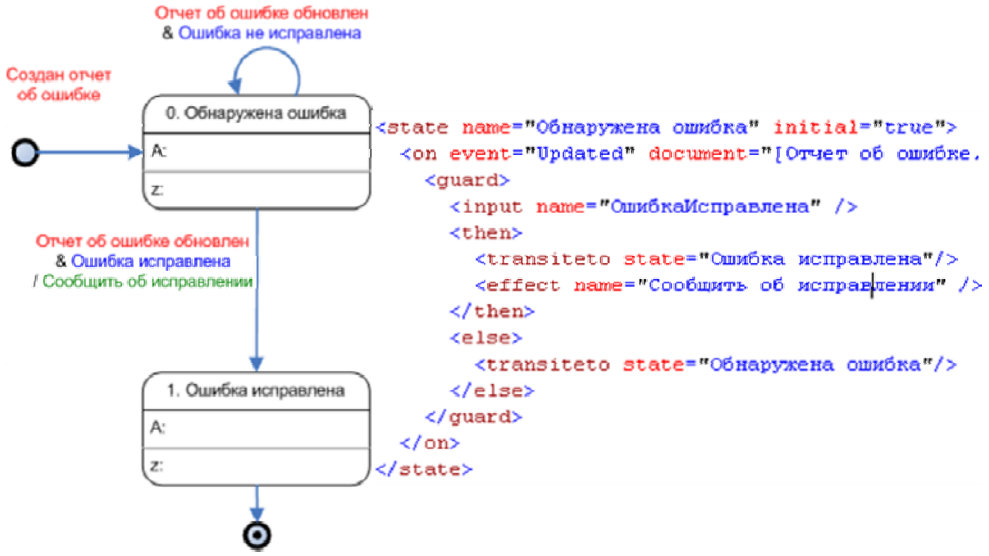


Рис. 5. Модель процесса исправления ошибки в графическом и XML представлениях

Устройство такой библиотеки аналогично устройству документа, с той разницей, что в документе модель жизненного цикла интерпретируется и хранится в виде XML-модели, а в модуле бизнес-процесса модель жизненного цикла скомпилирована. Модель жизненного цикла бизнес-процесса описывается аналогичным образом, в качестве примера приведем упрощенную модель процесса исправления ошибки и соответствующее *XML*-представление.

Описанные выше *XML*-модели, а в дальнейшем и исполняемый код, генерируются автоматически из графического представления конечных автоматов, созданных в редакторе *Microsoft Visio*, и текстовых описаний входных и выходных воздействий.

Полученная промежуточная *XML*-модель преобразуется в исходный код модуля на языке *C#* и компилируется соответствующим компилятором. Подробнее входные и выходные данные алгоритма генерации представлены на рис. 6.

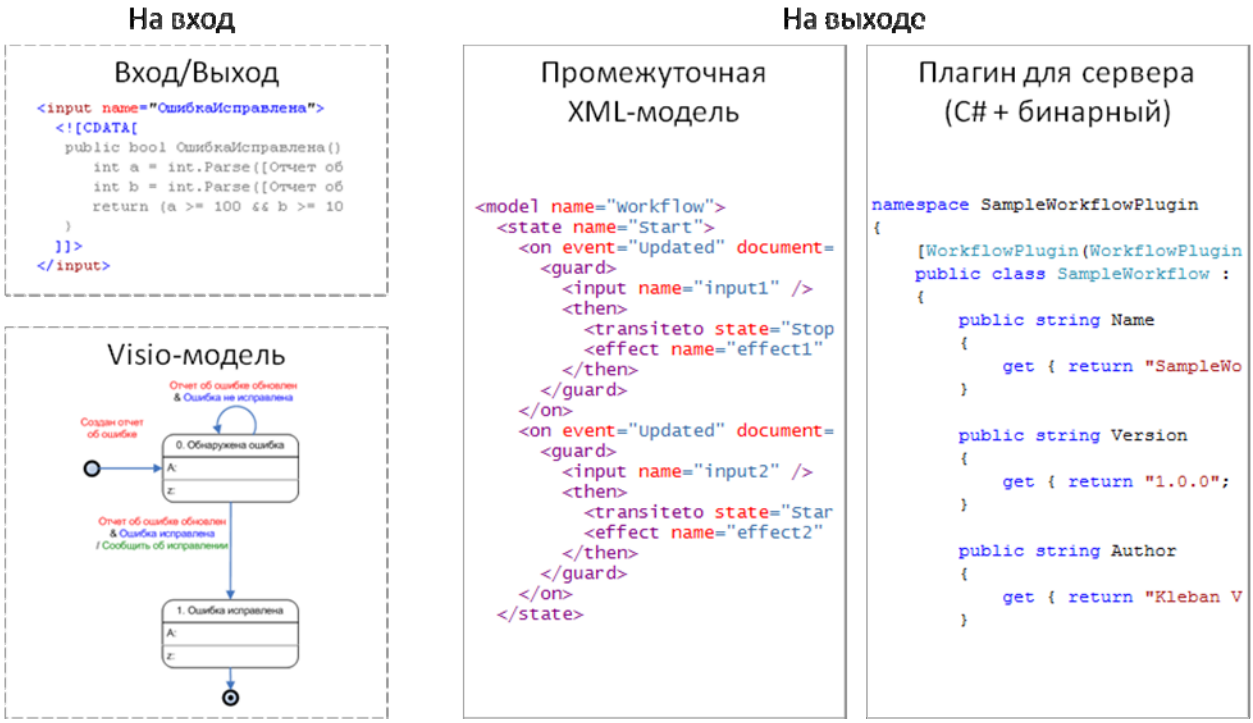


Рис. 6. Входные данные и результат работы

Язык представления модели

Модель жизненного цикла документа изначально представляется в графическом виде, но в документе хранится в виде *XML*-представления, формат которого приведен в таблице.

Таблица. Формат XML-представления

XML-тег	Комментарий
`<state name="Рассматривается">`	Имя состояния
`<to name="Принят" user="[.*]">`	Дуга из текущего состояния в состояние «Принят». Переход может быть инициирован любым пользователем. В случае указания квадратных скобок, имя пользователя сравнивается с регулярным выражением, в отсутствие скобок происходит прямое сравнение.

<guard> <input name="input" />	Переход из текущего состояния в состояние «принят» будет осуществлен, только если будет соблюдено условие с именем input
<then> <raise name="app_event"> <param name="param" value="0" /> </raise> </then>	Если условие соблюдено, то сгенерировать событие с именем app_event и параметром param=0
<else> <message text="Невозможно сменить состояние" /> </else> </guard> </to>	Если условие не соблюдено – вывести диалоговое окно, с текстом «Невозможно сменить состояние»
<to name="Refused" user="[.*]"> <raise name="MyExternalEvent"> <param name="MyParam" value="123" /> </raise> <message text="Document refused" /> </to> </state>	Пример перехода без условия (отсутствует guard)

Входные и выходные воздействия описываются с помощью языка C# в следующем виде:

```
<input name="ОшибкаИсправлена">
  <![CDATA[
   public bool ОшибкаИсправлена(){
      int a = int.Parse([Отчет об ошибке.docx].metric1);
      int b = int.Parse([Отчет об ошибке.docx].metric2);
      return (a >= 100 && b >= 100);
   }
  ]]>
</input>
```

Для удобства инженера в язык введена операция доступа к документам `[]`. Чтобы считать запись с именем «`metric1`» из документа «`Отчет об ошибке.docx`», необходимо использовать следующую операцию:

```
[Отчет об ошибке.docx].metric1
```

Заключение

Применение конечных автоматов в документообороте уже находит поддержку: например, в *Microsoft Workflow,* помимо представления документооборота в традиционной форме (в виде блок-схем), введено представление в виде конечных автоматов.

Использование конечных автоматов при автоматизации документооборота позволяет решить несколько важных задач:

- отделение в автомате логики работы от входных и выходных действий позволит в ряде случаев возложить обязанности по описанию бизнес-процессов не на инженера, а на руководящий состав;
- простота автоматной модели существенно снижает требования к разработчикам, которые занимаются автоматизацией предприятий, что очень важно при наблюдающемся дефиците специалистов;
- изменение модели жизненного цикла активного объекта (документа или процесса) не приводит к необходимости перепрограммирования, достаточно лишь изменить

схемы, так как большая часть программного кода генерируется автоматически по моделям;

- наблюдаемость автомата и его обратимость позволяют совершать «откат» системы в предыдущие состояния, не прибегая к сложным алгоритмам;
- возможность верификации автоматной модели позволяет строить системы правил, которые будут отвечать за ключевые моменты документооборота, что, в свою очередь, позволит проверять построенные модели еще до внедрения;
- классический способ наблюдения за показателями бизнеса – по метрикам (количество продукции и т.д.) – в случае использования автоматной модели может быть дополнен показателями, основанными на поведении моделей, а это означает, что повышается вероятность обнаружения «патологии» в работе предприятия.

Литература

1. Business Process Execution Language for Web Services version 1.1. – Режим доступа: http://www.ibm.com/developerworks/library/specification/ws-bpel/
2. Шеер А.-В. Моделирование бизнес-процессов. – М.: Весть-Метатехнология, 2000
3. Марка Д., МакГоуэн К. Методология структурного анализа и проектирования. – Режим доступа: http://www.marathon.ru/~fedor/doc/IDEF/ooad.asf.ru/standarts/idef/sadt/index.shtml
4. Шалыто А.А. Switch-технология. Алгоритмизация и программирование задач логического управления. –СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1
5. Автоматное программирование. Раздел «Верификация». http://is.ifmo.ru/verification/
6. Шалыто А.А., Туккель Н.И. SWITCH-технология — автоматный подход к созданию программного обеспечения «реактивных» систем // Промышленные АСУ и контроллеры. – 2000. – № 10. – Режим доступа: http://is.ifmo.ru/works/switch/1
7. Корнеев Г. А. Автоматизация построения визуализаторов алгоритмов дискретной математики на основе автоматного подхода. – Диссертация … канд. техн. наук /СПбГУ ИТМО. – 2006. – Режим доступа: http://is.ifmo.ru/disser/korn_disser.pdf

11 ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ И КОНЕЧНЫЕ АВТОМАТЫ

УДК 004.4’242

МЕТОД ОБУЧЕНИЯ СЛОЖНЫХ СИСТЕМ С БОЛЬШИМ ЧИСЛОМ ВХОДНЫХ ДАННЫХ И ВЫХОДНЫХ ВОЗДЕЙСТВИЙ

А.Л. Красс

(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

В работе предлагается метод, упрощающий процесс обучения класса сложных систем с большим числом входных данных и выходных воздействий. Он позволяет учитывать ограничения на поведение системы, что ведет к уменьшению размерности пространства поиска решений, удобным образом организовать разбиение задач системы на подзадачи, выполнению каждой из которых можно обучать систему отдельно от других и на основе различных методов машинного обучения. После этого полученные решения могут быть объединены с обеспечением достаточно сложные условия перехода между выполнением подзадач, в том числе вероятностные.

Ключевые слова: конечный автомат, машинное обучение

Введение

В области искусственного интеллекта существует множество задач обучения, в которых требуется учитывать большое число входных данных. Обучение таких систем – сложная задача. Оно обычно требует больших вычислительных затрат, а также тщательного и трудоемкого подбора настраиваемых параметров алгоритма обучения [1]. Наиболее существенны эти трудности для обучения без учителя [2].

В данной работе рассматриваются задачи, для которых применяется обучение без учителя. Для некоторых из них существуют небольшие подзадачи, для которых оптимальное или близкое к нему решение известно еще до обучения системы. В таких случаях можно сократить пространство поиска решений, зафиксировав решение известной подзадачи до обучения. Для распространенных методов это затруднительно, а если таких подзадач несколько, то могут возникнуть серьезные технические трудности. В данной работе предлагается метод, который позволяется избавиться от этих трудностей.

В работе вводится понятие автомата ограничений. Он представляет собой вероятностный автомат Мура-Мили [3] и организует совместную работу систем искусственного интеллекта, каждая из которых отвечает за выполнение своей подзадачи. С помощью этой структуры производится переключение между выполнением подзадач системы и контроль соблюдения ограничений на ее поведение. Примером ограничения может служить попытка робота пройти сквозь стену. Действия, которые непосредственно ведут к этой попытке, не имеют смысла и должны блокироваться автоматом ограничений.

Использование такой структуры дает возможность задавать каркас поведения системы, т.е. позволяет еще до обучения ограничить число допустимых стратегий системы, а также гарантировать, что нежелательные стратегии поведения не будут использованы, даже если они достаточно близки к оптимальной.

Автомат ограничений позволяет использовать решения подзадач, для которых существует оптимальное или достаточно близкое к нему решение еще до начала обучения системы. Во многих случаях это позволяет сократить количество входных данных, требующихся системе искусственного интеллекта для принятия решений, полностью перенеся ответственность за совершение некоторых действий в автомат ограничений. Это может существенно сократить пространство поиска решений и тем самым как

уменьшить вычислительные затраты на процесс обучения системы, так и упростить подбор настраиваемых параметров алгоритма обучения.

Использование автомата ограничений может быть полезно в процессе применения генетических алгоритмов для генерации детерминированных конечных автоматов [4–7]. В этом случае при нарушении некоторого ограничения, к переходу, нарушившему его, имеет смысл применить оператор мутации. Таким образом, получается аналог условно-рефлекторной схемы [8] для обучения детерминированных конечных автоматов.

Методы обучения систем искусственного интеллекта

Наиболее часто системы искусственного интеллекта строят с помощью искусственных нейронных сетей или детерминированных конечных автоматов. В случае обучения без учителя единственно возможными вариантами обучения таких систем являются генетические алгоритмы [4, 9–13] и подкрепляющее обучение [14–18].

Для многих сложных систем выполняемую задачу часто можно разбить на некоторый набор подзадач. Для обучения таких систем имеет смысл использовать иерархическое подкрепляющее обучение [19–21]. Его основная идея состоит в том, чтобы сначала обучить систему выполнять наиболее простые подзадачи, а уже на их основе строить решения других подзадач и всей задачи в целом. Однако использование подкрепляющего обучения часто оказывается достаточно сложным или невозможным из-за того, что непонятно, как поощрять или наказывать за действия, которые привели к ситуации, в которой нужно поощрить или наказать систему. Очень часто непонятно, за что наказывать, если цель не достигнута по истечении времени, отведенного на выполнение задачи. В таких случаях иногда используют комбинацию генетических алгоритмов и подкрепляющего обучения [22–27]. Часто это позволяет добиться более высокой скорости обучения, чем при раздельном использовании этих методов.

Постановка задачи

Пусть дан робот, получающий информацию от некоторого числа сенсоров, возможно, обладающий памятью о своих прошлых действиях или о состоянии среды, способный совершать определенное число действий.

Перед роботом стоит задача, состоящая из набора подзадач, которые он должен выполнить максимально эффективно в автономном режиме. Для оценки эффективности выполнения каждой подзадачи имеется оценочная функция. Существует некоторое число подзадач, для которых решение может быть однозначно построено еще до обучения системы. Для остальных подзадач стратегия поведения не определена, и поэтому требуется применять методы обучения без учителя (в том числе подкрепляющее обучение). Требуется построить и обучить систему, управляющую роботом, так, чтобы она обеспечивала выполнение поставленных перед роботом задач. Заметим, что постановку задачи можно было бы дать и в более общих терминах систем искусственного интеллекта, но для простоты она была сужена на системы, применяемые для управления автономными роботами.

Рассмотрим достоинства и недостатки существующих методов применительно к данной задаче, сгруппировав их по управляющей структуре.

Искусственная нейронная сеть

При использовании в качестве управляющей структуры искусственной нейронной сети для ее обучения следует применять генетические алгоритмы и частично подкрепляющее обучение, так как по постановке задачи доступно только обучение без учителя.

Существует проблема задания каркаса поведения системы. Например, если обучаемая система управляет роботом-собакой, то этот робот должен вести себя именно

как собака, а не как кошка. Искусственная нейронная сеть пытается найти оптимальное решение. Поэтому за такие ограничения должна отвечать оценочная функция нейронной сети, которая используется генетическим алгоритмом. Такая модификация может существенно замедлить вычисление этой функции, а также замедлить процесс обучения в целом. При этом ее построение может быть нетривиальной задачей.

Если требуется учитывать память о прошлом состоянии системы или состояниях окружающей среды, то для их учета требуется ввести новые входы нейронной сети, что также ведет к усложнению процесса обучения системы.

При использовании искусственных нейронных сетей вероятностное поведение системы может быть организовано достаточно естественным образом, если каждому действию системы сопоставить выход нейронной сети и считать, что чем больше значение на соответствующем выходе, тем больше вероятность совершения данного действия.

Детерминированный конечный автомат

Автомат – естественная структура для учета информации о прошлом состоянии системы или состоянии окружающей среды.

Детерминированные конечные автоматы обучают с помощью генетических алгоритмов [5–7]. На переходах автомата, применяемого для управления роботом, обычно размещают условия, сформированные из полученных от сенсоров данных. Так как сенсоров обычно много, то их данные группируют в одно значение или производят кластеризацию. Для кластеризации обычно применяют классифицирующие нейронные сети [1]. При большом числе сенсоров достаточно существенна проблема выбора правильного разбиения на кластеры. Эта проблема должна решаться каждый раз эвристически до основного процесса обучения или же можно объединить классифицирующую структуру, например, нейронную сеть, и автомат [28], а потом применять генетический алгоритм к ним обоим, что крайне сильно увеличивает пространство поиска решений. При этом детерминированный конечный автомат достаточно естественно может задать каркас системы, если того требует задача. Однако вероятностное поведение с помощью такого автомата задать нельзя.

Вероятностный автомат

На вероятностный автомат [3] переносятся особенности обучения, изложенные в предыдущем разделе. Единственное существенное различие состоит в вероятностном поведении. Оно достигается за счет усложнения автомата, а это ведет к увеличению размерности пространства поиска решений, что может быть неприемлемо во многих случаях.

Выводы

На основании изложенного можно сделать вывод, что на данный момент не существует удовлетворительного метода, который объединял бы в себе возможность задания каркаса поведения системы, обучения подзадач различными методами на основе различных структур (искусственные нейронные сети, детерминированные автоматы Мура-Мили, вероятностные автоматы), а также имел возможность учитывать подзадачи, решение которых известно еще до процесса обучения, ограничения на поведение системы и обеспечивать вероятностное поведения системы.

Метод обучения сложных систем с большим числом входных данных и выходных воздействий

В настоящей работе предлагается метод обучения систем с большим числом входных и выходных воздействий. Он основывается на применении автомата ограничений. Это модификация вероятностного автомата Мура-Мили [3], который задает ос-

нову поведения системы, указывает, какие действия можно совершать из данного состояния системы и при каких условиях, отвечает за переключением между выполнением подзадач. К каждому состоянию этого автомата можно «прикрепить» систему искусственного интеллекта (например, искусственную нейронную сеть или автомат Мура-Мили [29]). Поведение системы определяется системой искусственного интеллекта, соответствующей текущему состоянию автомата ограничений. Если к текущему состоянию не прикреплена система искусственного интеллекта, то поведение определяется только автоматом ограничений. При этом состояниям, относящимся к одной подзадаче, обычно ставится в соответствие одна система искусственного интеллекта.

Рассмотрим общую *схему предлагаемого метода*. Для некоторых задач отдельные пункты могут быть опущены.

1. Определение входных данных.
2. Определение выходных воздействий.
3. Выделение подзадач.
4. Построение автомата ограничений.
5. Закрепление за состояниями автомата ограничений систем искусственного интеллекта.
6. Выбор оценочных функций для каждой подзадачи.
7. Раздельное обучение систем искусственного интеллекта, отвечающих независимым подзадачам.
8. Объединение решений.
9. Выбор глобальной оценочной функции и дообучение системы в целом (при необходимости).

Опишем каждый этап метода на примере имитации поведения муравья во время поиска и сбора пищи. Части метода, выходящие за рамки данного примера будут опущены и могут быть найдены в работе [30].

Постановка задачи

По ограниченному квадратному полю размером 30×30 клеток передвигается муравей (рис. 1). На поле в случайно выбранных клетках находятся 100 единиц еды. Муравей может передвигаться в любую из восьми соседних клеток, брать еду из клетки, на которой стоит. При смене позиции муравей оставляет в покинутой клетке некоторое количество феромона. Чем больше ходов сделал муравей с последнего момента, когда он брал еду, тем меньше феромона он оставляет. Если муравей кладет феромон в клетку, в которой уже находится феромон, то их количества суммируются.

Муравей обладает информацией о том, что находится в соседних восьми клетках и в клетке, на которой он стоит (рис. 2). Если муравей стоит на границе поля, то к нему поступает информация, что клетки за границей поля недостижимы.

Таким образом, муравей имеет 35 сенсоров: девять сенсоров передают информацию о наличии еды в текущей и восьми соседних клетках поля, девять сенсоров передают информацию о количестве феромонов в текущей и восьми соседних клетках поля, девять сенсоров передают информацию о наличии муравейника в текущей и восьми соседних клетках поля, восемь сенсоров сообщают о проходимости восьми соседних клеток поля (находятся ли они за его границей).

После того как муравей взял еду, он должен с ней вернуться в специально выделенную часть поля размером 2×2 клетки, отстоящую на одну клетку от верхнего левого угла, и выполнить на одной из клеток этой части поля действие – положить еду. После этого он может приступить к поиску новой еды.

В любой момент времени муравей может нести не более одной единицы еды.

Муравей может за один ход либо взять еду из текущей клетки поля, либо положить еду в муравейник, либо переместиться в соседнюю клетку поля.

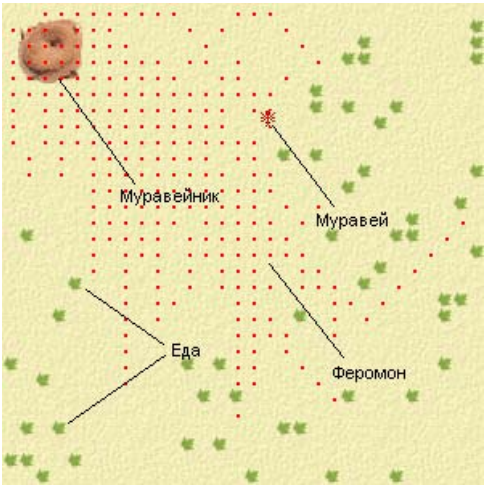


Рис. 1. Муравей, передвигающийся по полю

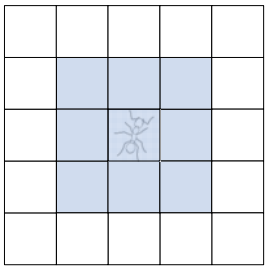


Рис. 2. Область обзора муравья

Муравей делает 1000 ходов. После этого количество еды, перенесенное в муравейник, фиксируется. Задача состоит в том, чтобы обучить муравья переносить как можно больше еды за 1000 ходов.

Определение входных данных

Входные данные генерируются 35 сенсорами:

- 9 сенсоров передают информацию о наличии еды в текущей и восьми соседних клетках поля;
- 9 сенсоров передают информацию о количестве феромона текущей и восьми соседних клетках поля;
- 9 сенсоров передают информацию о наличии муравейника в текущей и восьми соседних клетках поля;
- 8 сенсоров формируют информацию о проходимости восьми соседних клеток поля (находятся ли они за границей поля).

Определение выходных воздействий

Выходные воздействия – это действия муравья:

- взять еду из текущей клетки поля;
- положить еду в муравейник;
- переместиться на соседнюю клетку поля.

Выделение подзадач

На этом этапе требуется выбрать разбиение задачи системы на подзадачи. Для каждой подзадачи имеет смысл определить, какие входные данные ей потребуются и какие выходные воздействия система может генерировать.

Можно выделить две подзадачи: найти и взять еду, вернуться в муравейник и положить еду. Первую будем назвать задачей *A*, вторую – задачей *B*.Для выполнения каждой из этих задач муравей должен знать о наличии еды в текущей и восьми соседних клетках поля, о количестве феромонов в текущей и восьми соседних клетках поля, о проходимости восьми соседних клеток поля (находятся ли они за границей поля). Муравей должен иметь возможность перемещаться в любую из восьми соседних клеток поля, брать еду во время выполнения задачи *A* и класть еду в муравейник во время выполнения задачи *B*.

Построение автомата ограничений

Автомат ограничений строят в виде модификации вероятностного автомата Мура-Мили. Модификация заключается в том, что к каждому его состоянию может быть прикреплена система искусственного интеллекта, и переходы автомата могут содержать дополнительную информацию, говорящую насколько значим и желателен соответствующий переход. Например, имеет смысл помечать переходы, которые происходят по достижению системой какой-либо локальной цели. Эта информация может быть использована во время обучения подзадач и системы в целом.

Автомат ограничений должен отражать то, какие действия и при каких условиях система может совершать, находясь в определенных состояниях, и управлять переключением между выполнением подзадач (рис. 3). Если по одному событию возможны несколько переходов с условиями, которые могут быть верны одновременно, то этим переходам должны быть присвоены вероятности, которые в дальнейшем можно будет подобрать оптимальным образом в процессе обучения системы.

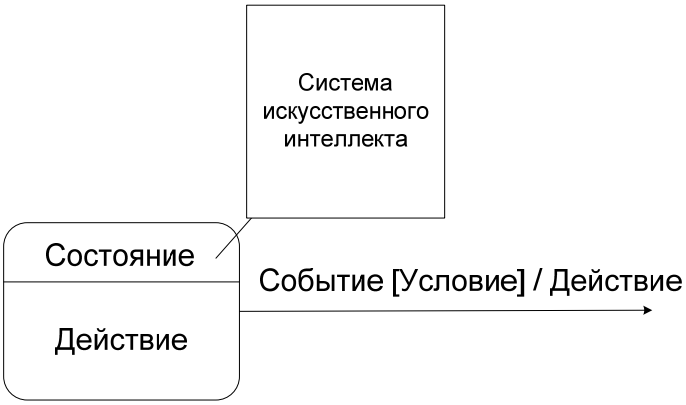


Рис. 3. Состояние автомата ограничений

Когда система хочет сделать ход (рис. 4), она получает из текущего состояния автомата ограничений соответствующую систему искусственного интеллекта, которая должна выбрать выходное воздействие системы или выдать вектор вероятностей выходных воздействий, на основе данных от сенсоров и памяти. Выходное воздействие является событием для автомата ограничений. Если при этом если не существует ребра из текущего состояния автомата ограничений по данному событию с выполняющимся условием перехода, то выходное воздействие не допускается. Если выбранное системой искусственного интеллекта выходное воздействие (или ни одно выходное воздействие

из вектора вероятностей выходных воздействий с ненулевой вероятностью) не допускается автоматом, то происходит перебор всех переходов из данного состояния автомата, пока не будет найден допустимый. Он и будет выполнен.

Рис. 4. Основной цикл работы системы под управлением автомата ограничений

Состоянию автомата ограничений может не ставиться в соответствие система искусственного интеллекта. В этом случае переход выбирается только на основании соблюдений условий перехода. Таким образом, автомат ограничений можно использовать как основную структуру управления системой, включая разного рода внутренние функции, например, диагностику неисправностей аппаратных компонентов системы. Это позволяет упростить использование автомата ограничений для задания многофункциональных систем.

Автомат ограничений строится на основе выбранного разбиения на подзадачи, а его переходы задают ограничения на поведение системы. Автомат ограничений строится так, чтобы каждое его состояние соответствовало некоторому состоянию системы. К каждому состоянию автомата ограничений добавляются переходы, которые возможны из этого состояния, тем самым будут заданы ограничения на поведение системы.

Построим автомат ограничений для рассматриваемой задачи. Автомат ограничений будем строить в виде автомата Мили [29], так как в данной задаче не требуется организовывать вероятностные переходы между выполнением подзадач.

Достаточно естественно выделить два состояния: *A* и *B*. В состоянии *A* муравей не несет еды, в состоянии *B* – несет. Опишем, каким условиям должны удовлетворять переходы из состояния *A:*

- если муравей в состоянии *A* стоит на еде, то он должен ее взять (это несколько спорный вопрос, но для простоты было выбрано именно это решение) и перейти в состояние *B;*
- если муравей в состоянии *A* видит еду, но не стоит на ней, то он должен идти в клетку, ее содержащую (если он видит несколько клеток с едой, то выбирается произвольная);
- если муравей не видит еды, то он может идти в любую сторону, если соответствующая клетка проходима;
- муравей не может класть еду в муравейник (так как у него не может быть еды в этом состоянии автомата).

Опишем, каким условиям должны удовлетворять переходы из состояния *B:*

- если муравей в состоянии *B* стоит на клетке муравейника, то он должен положить еду и перейти в состояние *A*;
- если муравей в состоянии *B* видит клетку муравейника, то он должен идти в нее (если он видит несколько клеток муравейника, то выбирает произвольную);
- если муравей не видит клетку муравейника, то может идти в любую сторону, при условии, что соответствующая клетка проходима;
- муравей не может взять еду (так как он уже несет еду).

Вышеописанный автомат приведен на рис. 5, 6.

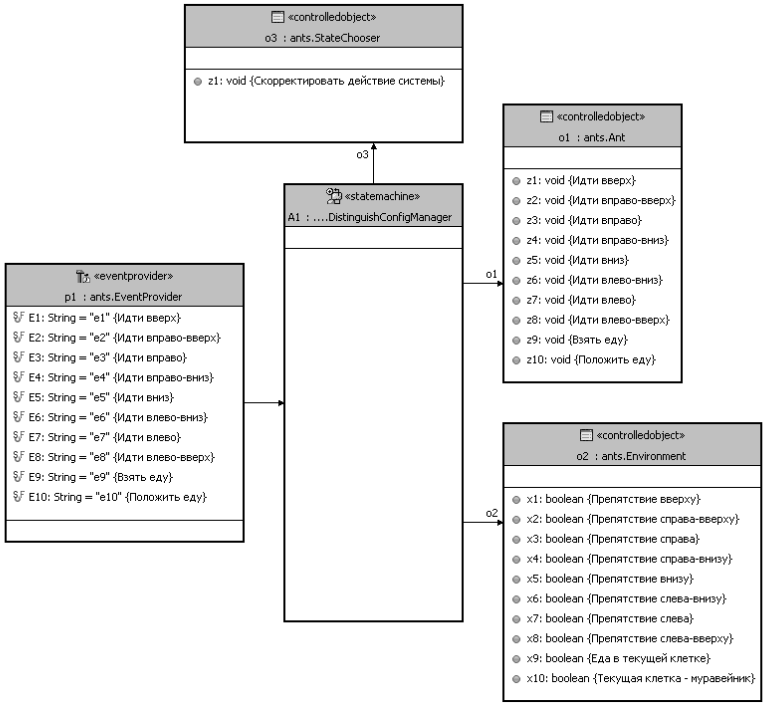


Рис. 5. Диаграмма классов для автомата, задающего ограничения на поведение муравья

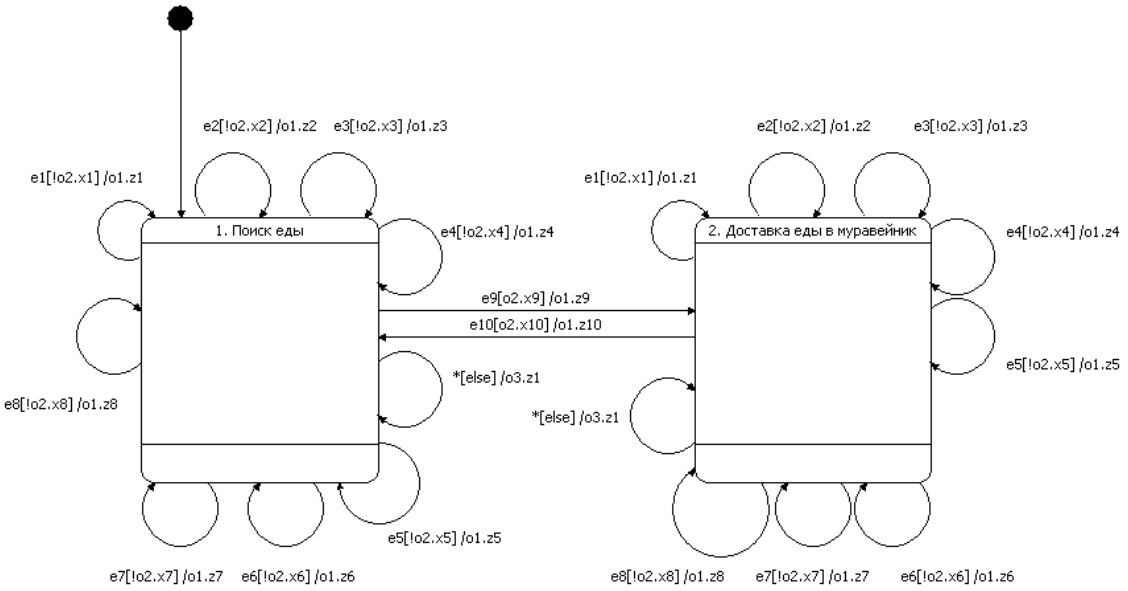


Рис. 6. Автомат ограничений для муравья

Закрепление за состояниями автомата ограничений систем искусственного интеллекта

«Прикрепим» к каждому состоянию автомата однослойную нейронную сеть. Такие нейронные сети просты в обучении и позволяют естественным образом обеспечивать вероятностное поведение системы. Для этого будем интерпретировать выходы нейронной сети как вектор вероятностей действий системы, где каждому действию соответствует один выход.

Как следует из предыдущего раздела, если в зону видимости муравья попадает еда, то его поведение полностью управляется автоматом ограничений, поэтому в нейронную сеть, прикрепленную к состоянию *A*, добавлять входы, отвечающие за информацию о еде не имеет никакого смысла. То же касается и выхода отвечающего за действие взять еду. Аналогичным образом автомат не позволит муравью выйти за границы поля. Поэтому можно исключить из нейронной сети и входы, отвечающие за проходимость клеток. Полученная нейронная сеть приведена на рис. 7.

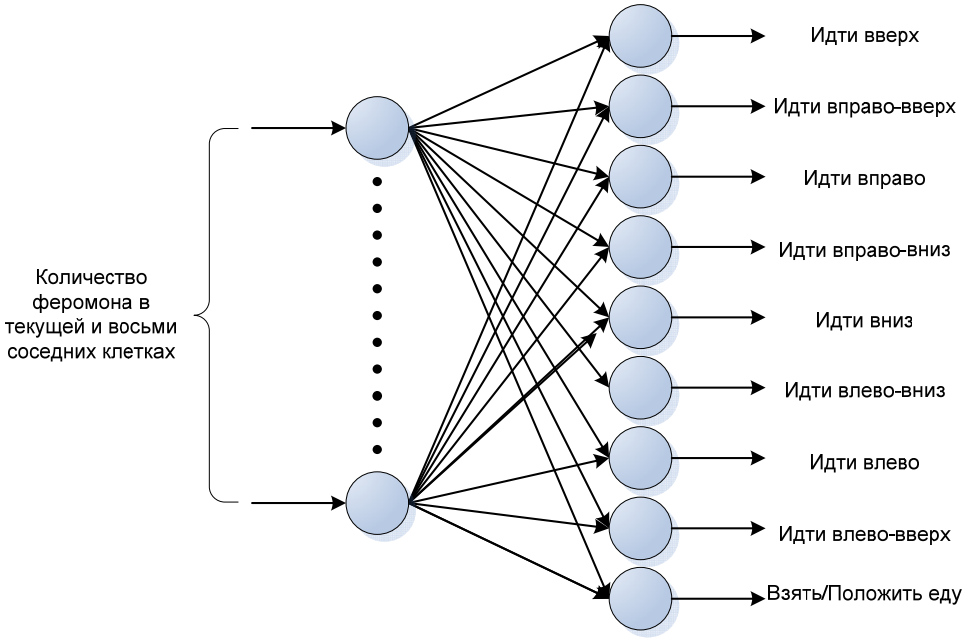


Рис. 7. Нейронная сеть, используемая для получения вероятностей совершения действий муравьем при использовании автомата ограничений

Отметим, что знание о наличии муравейника в текущей и восьми соседних клетках поля может быть полезным, так как муравейник находится практически в левом верхнем углу поля. Поэтому муравей может определить свою позицию на поле, если видит одну из клеток муравейника. В данном случае эта информация не используется, чтобы сохранить чистоту эксперимента при сравнении решения, полученного с помощью предлагаемого метода, с решением данной задачи классическими методами, где при использовании этой информации существенно бы увеличилось пространство поиска решений, что еще больше усложнило бы нахождение решения.

Для выполнения задачи *B* муравей должен знать о наличии муравейника в текущей и восьми соседних клетках поля может быть полезным и о количестве феромонов в текущей и восьми соседних клетках поля. Так же он должен обладать информацией о проходимости восьми соседних клеток поля (находятся ли они за границей поля). Муравей должен иметь возможность перемещаться в любую из восьми соседних клеток поля и класть еду в муравейник.

Как можно видеть из предыдущего раздела, если в зону видимости муравья попадает муравейник, то его поведение полностью обеспечивается автоматом ограничений. Поэтому в нейронную сеть, прикрепленную к состоянию *A*, добавлять входы, отвечающие за информацию о муравейнике, не имеет никакого смысла. Это же относится и к выходу, отвечающему за действие «Положить еду». Аналогичным образом автомат не позволит муравью выйти за границы поля. Поэтому можно исключить из нейронной сети и входы, отвечающие за проходимость клеток. Полученная нейронная сеть приведена на рис. 7. Она имеет точно такую же структуру, что и нейронная сеть, использующаяся в состоянии *A*.

Таким образом, за рассмотренными состояниями закреплены нейронные сети одинаковой структуры. Они имеют девять входов, отвечающих за передачу информации о количестве феромона в текущей и соседних клетках, и девять выходов (рис. 7). Эти нейронные сети отвечают только за передвижение муравья, основываясь на феромонах, которые лежат в зоне видимости муравья. Остальные функции будет выполнять автомат ограничений.

Выбор оценочных функций для каждой отдельной задачи

Для задачи сбора еды оценочная функция будет выглядеть как усредненное количество еды, собранное за некоторое число запусков на случайно сгенерированных полях. Для задачи возвращения в муравейник оценочная функция – усредненное количество еды, положенное в муравейник за некоторое число запусков на полях, полученных в процессе выполнения первой подзадачи.

Раздельное обучение систем искусственного интеллекта, отвечающих независимым подзадачам

Теперь можно обучить систему выполнять каждую подзадачу отдельно. Так как было принято решение построить конечное решение только для подзадачи поиска и сбора пищи, то рассмотрим этот и последующие разделы в рамках этой задачи.

Данной подзадаче соответствует только одно состояние автомата ограничений. Чтобы иметь возможность обучить систему выполнять эту задачу, необходимо все переходы, которые ведут в состояния, не относящиеся к поставленной задаче, перевести в соответствующее состояние внутри группы. При этом во многих случаях требуется указать на этом ребре действие, которое бы приводило состояние системы в соответствие с поставленной задачей.

В данном случае есть только один переход, который ведет вовне группы. Это переход по действию «Взять еду». Переведем его в то же состояние, а на переходе к действию «Взять еду» добавим перемещение в левый верхний угол муравейника и действие «Положить еду». Тем самым будет выполнена имитация решения второй подзачи.

Для обучения нейронной сети используются генетические алгоритмы. Достаточная скорость и качество обучения достигаются при размере популяции в 100 особей, двухточечным кроссовером и отбором с применением элитизма. Подробнее процесс обучения рассмотрен в работе [30].

Анализ полученных результатов

После 100 итераций обучение перестало давать сколь-нибудь существенное увеличение функции приспособленности. Муравей был обучен собирать в среднем 48.8 единиц еды за 1000 ходов.

Данная подзадача была решена с помощью обучения искусственной нейронной сети генетическим алгоритмом без применения предложенного метода. Потребовалось

использовать однослойную нейронную сеть из 26 входов и 9 выходов. Лучший результат был показан при схожих параметрах генетического алгоритма. За 500 итераций для популяции в 300 особей удалось научить муравья собирать столько же еды, что и при применении предлагаемого в данной работе метода. Но, как можно видеть, это потребовало в 15 раз больше вычислительных затрат.

Учитывая простоту задачи, можно сделать вывод, что для более сложных задач выигрыш при применении данного метода будет еще более существенен.

Заключение

В области искусственного интеллекта много задач обучения, в которых требуется учитывать большое число входных и выходных данных. Обучение таких систем – сложная задача. Оно обычно требует больших вычислительных затрат, а также тщательного и трудоемкого подбора настраиваемых параметров алгоритма обучения.

Если доступно только обучение без учителя, то распространенные методы [1, 2, 4–7, 9–28] из-за большой размерности пространства поиска решений могут требовать недопустимо высоких вычислительных затрат для обучения системы. Однако если из задач, которые должна выполнять система, можно выделить подзадачи, для которых оптимальное или близкое к нему решение известно еще до обучения системы, то во многих случаях можно сократить пространство поиска решений, применив предлагаемый в данной работе метод.

В результате сравнения результатов обучения искусственной нейронной сети генетическим алгоритмом с применением предложенного методом и без него можно сделать вывод, что данный подход может быть полезен для решения задач описанного класса, так как из-за уменьшения размерности пространства поиска решений возрастает скорость обучения. По этой же причине метод требует менее тщательного подбора настраиваемых параметров обучения.

Данный метод дает возможность задавать каркас поведения системы, т.е. еще до обучения ограничить число допустимых стратегий системы, а также гарантировать, что нежелательные стратегии поведения не будут использованы, даже если они достаточно близки к оптимальной.

Литература

1. De Nardi R., Togelius J., Holland O., Hollland L., Simon M. Evolution of Neural Networks for Helicopter Control: Why Modularity Matters / Proceedings of the IEEE Congress on Evolutionary Computation, 2006.
2. Барский А.Б. Нейронные сети: распознавание, управление, принятие решений. – М.: Финансы и статистика, 2004.
3. Bukharaev R. G. Probabilistic automata // Journal of Mathematical Sciences. – Springer New York. – 1980. – V. 13. – № 3.
4. Chambers L. D. Practical Handbook of Genetic Algorithms, Volume 3, Chapter 26, 6 – Algorithms to Improve the Convergence of a Genetic Algorithm with a Finite State Machine Genome. CRC Press, 1999.
5. Царев Ф.Н., Шалыто А.А. О построении автоматов с минимальным числом состояний для задачи об «умном муравье» / Сборник докладов X международной конференции по мягким вычислениям и измерениям. – СПбГЭТУ "ЛЭТИ". – Т. 2. – 2007. – С. 88–91.
6. Petrovic P. Evolving automatons for distributed behavior arbitration. Technical Report. Norwegian University of Science and Technology. 2005.

7. Petrovic P. Simulated evolution of distributed FSA behaviour-based arbitration / The Eighth Scandinavian Conference on Artificial Intelligence (SCAI'03). – 2003.
8. Цетлин М.Л. Исследования по теории автоматов и моделированию биологических систем. – М.: Наука, 1969.
9. Hussain T. Methods of Combining Neural Networks and Genetic Algorithms. http://citeseer.ist.psu.edu/hussain97methods.html
10. Filippidis A., Jain L. C., Martin N. M. Using genetic algorithms and neural networks for surface land mine detection //IEEE Transactions on Signal Processing. – 1999. – 47(1) – 176–186.
11. Koehn P. Combining genetic algorithms and neural networks: The encoding problem. Master's thesis. University of Erlangen and The University of Tennessee. Knoxville. 1994. – Режим доступа: http://citeseer.ist.psu.edu/article/koehn94combining.html
12. Mitchell M. An Introduction to Genetic Algorithms. First MIT Press, 1998.
13. Miller B., Goldberg M. Genetic algorithms, tournament selection, and the effects of noise // Complex Systems. – 1995. – V. 9. – I. 3. – P. 193–212.
14. Sutton R. S., Barto A. G. Reinforcement Learning: An Introduction. – The MIT Press, Cambridge. MA, 1998.
15. Kaelbling L. P., Littman M. L., Moore A. W. Reinforcement learning: A Survey // Journal of Artificial Intelligence Research. – 1996. – 4. – P. 237–285.
16. Stanley K., Miikkulainen R. Efficient Reinforcement Learning Through Evolving Neural Network Topologies / GECCO 2002, pp. 569–577.
17. Armstrong W. W., Coghlan B., Gorodnichy D. O. Reinforcement learning for robot navigation / International Joint Conference on Neural Networks (IJCNN'99). Proceedings. – Washington DC. – 1999.
18. Coulom R. Reinforcement Learning Using Neural Networks, with Applications to Motor Control. Institut National Polytechnique de Grenoble, 2002. – Режим доступа: http://citeseer.ist.psu.edu/coulom02reinforcement.html
19. Kaiser M., Dillmann R. Hierarchical learning of efficient skill application for autonomous robots / International Symposium on Intelligent Robotics Systems. – Pisa. Italy. – 1995.
20. Soni V., Singh S. Reinforcement Learning of Hierarchical Skills on the Sony Aibo Robot /In the International Conference on Developmental Learning (ICDL '06). – Bloomington. USA, 2006.
21. Diuk C., Strehl A. L., Littman M. L. A hierarchical approach to efficient reinforcement learning in deterministic domains / Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems. – Hakodate. Japan. – 2006.
22. Teller A., Veloso M. Internal Reinforcement in a Connectionist Genetic Programming Approach // Artificial Intelligence. – 2000. – V. 120. – № 2. – P. 165–198.
23. Whitley D., Dominic S., Das R. Genetic Reinforcement Learning with Multilayer Neural Networks / Proceedings of the Fourth International Conference on Genetic Algorithms. – P. 562–569. – Morgan Kaufmann.
24. Smith J. E. Coevolving Memetic Algorithms: A Review and Progress Report. // IEEE Transactions on Systems Man and Cybernetics – Part B. – 2007. – 37. – P. 6–17.
25. Ong Y. S., Keane A. J. Meta-Lamarckian learning in memetic algorithms // IEEE Transactions on Evolutionary Computation. – 2004. – 8. – P. 99–110.
26. Ong Y. S., Lim M. H., Zhu N., Wong K. W. Classification of Adaptive Memetic Algorithms: A Comparative Study" // IEEE Transactions on Systems Man and Cybernetics – Part B. – 2006. – 36. – P. 141–152.
27. Царев Ф.Н., Шалыто А.А. Применение генетического программирования для построения мультиагентной системы одного класса / Международная научно-техническая мультиконференция «Проблемы информационно-компьютерных техно-

логий и мехатроники». Материалы международной научно-технической конференции «Многопроцессорные вычислительные и управляющие системы» (МВУС`2007). – Таганрог: НИИМВС, 2007. – Т.2. – С. 46–51.
28. Брауэр В. Введение в теорию конечных автоматов. – М.: Радио и связь, 1987.
29. Красс А. Метод обучения сложных систем с большим числом сенсоров и актуаторов. Бакалаврская работа / СПбГУ ИТМО, факультет «Информационные технологии и программирование», кафедра «Компьютерные технологии». – 2008.

SUMMARY

AUTOMATA-BASED PROGRAMMING PARADIGM

Anatoly Shalyto (shalyto@mail.ifmo.ru)

Keywords: programming paradigm, automata-based programming, programming with explicit state emphasis

Automata-based programming and its advantages for software development are described in this paper. Tools for automata-based programming and examples of its successful application are also described. Procedure-based programming and object-oriented programming with explicit state emphasis are described. Automata-based programs can be efficiently verified, for their construction genetic algorithms can be applied.

APPLICATION OF GENETIC PROGRAMMING FOR GENERATION OF AUTOMATA HAVING A LOT OF INPUT VARIABLE

Nadezhda Polikarpova (policorn@rain.ifmo.ru), Vladimir Tochilin (tochilin@rain.ifmo.ru), Anatoly Shalyto (shalyto@mail.ifmo.ru)

Keywords: genetic algorithm, automata-based programming

Efficiency of the known methods of automatic generation of finite automata based on genetic programming exponentially decreases when number of input variables increases. In this paper presented a method that does not have these deficiencies. Preference for the application of the proposed method with a large number of input variables substantiated theoretically. The method was implemented in a tool to automate the development of control system of aircraft at a high level of abstraction.

APPLICATION OF GENETIC PROGRAMMING, FINITE STATE MACHINES AND NEURAL NETS FOR CONSTRUCTION OF CONTROL SYSTEM FOR UNNMANED AIRCRAFT

Fedor Tsarev (tsarev@rain.ifmo.ru)

Keywords: genetic programming, neural net, finite state machine, unmanned aircraft, automata-based programming

Application of genetic algorithms for unmanned aircraft control system construction is considered in this paper. Control system contains a finite state machine and an artificial neural net. Neural net converts real input variables into finite state machine boolean input variables. Finite state machine generates input actions. Genetic algorithms are used for optimization of such model.

APPLICATION OF GENETIC PROGRAMMING AND OF REDUCED TRANSITION TABLES METHOD AND OF DECISION TREES METHOD FOR CONSTRUCTING A CONTROL SYSTEM FOR A MODEL OF AN UNMANNED AIRCRAFT

Andrey Davydov (davydov@rain.ifmo.ru), Dmitri Sokolov (sokolov@rain.ifmo.ru), Fedor Tsarev (tsarev@rain.ifmo.ru)

Keywords: genetic programming, finite state machine, unmanned aircraft, automata-based programming

Application of genetic algorithms for construction of control systems for systems with complex behavior is considered in this paper. Two methods (reduced tables method and decision trees method) of automata representation are used. Application of these methods is illustrated on the problem of unmanned aircraft control.

CONSTRUCTION OF A CONTROL SYSTEM FOR A HELICOPTER MODEL USING GENETIC ALGORITHMS

Pavel Lobanov, Sergey Sytnik, Anatoly Shalyto (shalyto@mail.ifmo.ru)

Keywords: genetic algorithm, finite state machine, automata-based programming

The genetic algorithm for constructing the autopilot for a simplified model of a helicopter is proposed. Goal for helicopter is to pass predefined set of targets in predefined order. Implementation of genetic algorithm was made with *Java* programming language. Computational experiments to demonstrate the effectiveness of the algorithm are implemented.

CONSTRUCTION OF A CONTROL SYSTEM FOR ROBOCODE TANK USING GENETIC ALGORITHMS

Yury Bedny (bedny@rain.ifmo.ru), Anatoly Shalyto (shalyto@mail.ifmo.ru)

Keywords: genetic algorithm, finite state machine, Robocode, automata-based programming

The paper proposes a method of developing of the control system of the tank for the game Robocode, based on genetic algorithms that can automate the process of building control systems. In addition, the way to improve generated control systems by the use of more sophisticated algorithms realized with finite automaton.

DEVELOPMENT OF A TOOL FOR FINITE STATE MACHINE GENERATION WITH GENETIC ALGORTIHMS

Evgeny Mandrikov (mandrikov@rain.ifmo.ru), Vladimir Kulev (kulev@rain.ifmo.ru)

Keywords: genetic algorithm, tool, finite state machine, automata-based programming

As part of the research the tool is created for generating automata based on genetic programming, which allows you to increase the automation of development of automata-based programs. This tool name is GAAP (Genetic Algorithms for Automata-based Programming).

A METHOD OF REPRESENTING AUTOMATA WITH DECISION TREES FOR GENETIC PROGRAMMING

Vladimir Danilov (vdaniloff@rain.ifmo.ru)

Keywords: genetic programming, decision tree, finite state machine, automata-based programming

The paper proposes a new method for representation of automaton as an individual in an evolutionary algorithm based on a decision trees. Application of this method greatly reduces amount of memory needed for running evolutionary algorithm. The method of genetic programming based on the proposed representation is created. Efficiency of this method is illustrated on the example of solving one of the variants of “Artificial ant” problem.

APPLICATION OF GENETIC ALGORTIHMS FOR MOORE AUTOMATA AND SYSTEMS OF MEALY AUTOMATA CONSTRUCTION IN “ARTIFICIAL ANT” PROBLEM

Andery Davydov (davydov@rain.ifmo.ru), Dmitry Sokolov (sokolov@rain.ifmo.ru), Fedor Tsarev (tsarev@rain.ifmo.ru)

Keywords: genetic algorithm, Moore finite state machine, Mealy finite state machine, automata-based programming

A genetic algorithm for constructing Moore finite state machines is proposed in this paper. The same algorithm can be used for constructing systems of interaction Mealy finite state machines. For these types of automata genetic operations of mutation and cross-over were developed. Genetic algorithms are implemented with *Java* programming language. Application of these algorithms is illustrated on the example of “Artificial Ant” problem

.

METHODS FOR OPTIMIZATION OF GENETIC ALGORITHMS CONSTRUCTING FINITE STATE MACHINES

Pavel Lobanov (pavel.lobanov@gmail.com)

Keywords: genetic algorithm, automata-based programming

Three ways is proposed to increase the rate of convergence of standard genetic algorithms for the class of problems in which the solution is a finite automaton. First of them allows to greatly reduce number of automaton states thanks to usage of flags. Method of recovering of links between states allows reducing number of unused states. Method of sorting states in usage order allows reducing number of states in automata.

METHODS OF AUTOMATA-BASED PROGRAMS VERIFICATION

Sergey Velder (velder@rain.ifmo.ru), Anatoly Shalyto (shalyto@mail.ifmo.ru)

Keywords: program verification, model checking, automata-based programming

This article discusses ways of converting automata-based programs to Kripke models, designed to check the properties related to the behavior of the system. These properties are described with temporal logic formulas. Several methods of such a transformation are proposed.

VERIFICATION OF AUTOMATA-BASED PROGRAMS WITH *SMV* TOOL

Evgeny Kurbatsky (kurbatsky@rain.ifmo.ru)

Keywords: program verification, model checking, automata-based programming, SMV verifier

We consider the method of verification of automata-based programs using Model Checking method. To verify the model the tool SMV is used. In the proposed approach, transition system is not built in explicit form. This allows the program to verify systems with large number of states.

VERIFICATION OF AUTOMATA-BASED PROGRAMS WITH SPIN VERIFIER

Michail Lukin (lukin@rain.ifmo.ru), Anatoly Shalyto (shalyto@mail.ifmo.ru)

Keywords: program verification, model checking, automata-based programming, SPIN verifier

A way to use SPIN verifier for the verification of automata-based programs is proposed. When using this verifier model is described with *Promela* language and requirements are specified with *LTL* language. A method of automated automata-based program to *Promela* model transformation is proposed. For LTL-formulas transformation a special program was developed. Usage of this methods is illustrated on the example of cash-machine model verification.

VERIFICATION OF AUTOMATA-BASED PROGRAMS WITH UNIMOD.VERIFIER TOOL

Vadim Gurov (vadim.gurov@gmail.com), Bulat Jaminov (jaminov@rain.ifmo.ru)

Keywords: program verification, model checking, automata-based programming

This article describes the verifier of automata-based programs created with the tool to support automata-based programming UniMod. Verifier works by integrating tool UniMod and verifier Bogor. Using developed verifier there is no need to convert automata-based program to the input language of the verifier. Requirements for the program are written in the language of temporal logic LTL.

DEVELOPMENT OF A VERIFIER FOR AUTOMATA-BASED PROGRAMS

Kiril Egorov (egorov@rain.ifmo.ru), Anatoly Shalyto (shalyto@mail.ifmo.ru)

Keywords: program verification, model checking, automata-based programming, Byuhi automaton

This article describes verifier of automata-based programs created with the tool to support automatic programming UniMod. When it is used, in contrast to the well-known verifiers, there is no need to describe the model in the input language of the verifier. Requirements for the program are written in the language of temporal logic of linear time LTL (Linear Time Logic). Using this logic verification carried out by intersection of automaton-product of the model and Byuhi automaton built to deny LTL formula.

APPLICATION OF AUTOMATA-BASED APPROACH FOR JAVA CARD-APPLICATIONS DEVELOPMENT

Andrey Zakonov (zakonov@rain.ifmo.ru)

Keywords: program verification, Java Card, automata-based programming

The article proposes an approach that increases the reliability of the development of Java Card applets. Features of applet developing using automata-based approach are formulated. Enhanced representation model to describe the functionality, code generator with plugs and tools for automatic testing of applets are provided.

DEVELOPMENT OF CORECT JAVA CARD-APPLICATION WITH AUTOMATA-BASED APPROACH

Andrey Klebanov (klebanov@rain.ifmo.ru), Anatoly Shalyto (shalyto@mail.ifmo.ru)

Keywords: program verification, Java Card, automata-based programming

In this paper describes the results of studies aimed at establishing the correct Java Card-code. In doing so, the code is generated from the high level description of automata-based programming. An additional advantage of this approach is the ability to generate a formal specification of the application. Compliance with the source or byte-code specifications can be verified by various verifiers or tools for dynamic and static testing.

VERIFICATION OF INTERACTION OF PARTS OF A REACTIVE SYSTEM IMPLEMENTED USING AUTOMATA-BASED APPROACH

Sergey Kanzhelev (kanzhelev@rain.ifmo.ru)

Keywords: program verification, reactive system, automata-based programming

Verification of interaction of parts of a reactive system implemented with automata-based approach is discussed in this paper. Problems which can appear in such systems and methods of their detection for several types of semantical interpretation are considered.

AUTOMATIC DYNAMIC VERIFICATION OF AUTOMATA-BASED PROGRAMS

Oleg Stepanov (oleg.stepanov@gmail.com)

Keywords: dynamic verification, automata-based programming

Method of automatic dynamic verification of Mealy finite state machines system based on alternating automat traversal. Structure of method, rules of building protocols and specification are described. Functional characteristics of this method are analysed on the base of its implementation.

DECLARATIVE APPROACH TO EMBEDDING AND INHERITANCE OF AUTOMATA CLASSES WITH IMPERATIVE PROGRAMMING LANGUAGES

Artem Astafurov (Artyom.Astafurov@gmail.com)

Keywords: declarative approach, automata-based programming, macrostate

Declarative approach for automata-based objects in object-oriented imperative programming languages with static type checking is discussed in this paper. Main feature of this method is the ability of application of inheritance and embedding of macrostates.

INHERITANCE OF AUTOMATA CLASSED WITH DYNAMIC PROGRAMMING LANGUAGES ON THE EXAMPLE OF *RUBY*

Kirill Timofeev (timofeev@rain.ifmo.ru), Artem Astafurov (Artyom.Astafurov@gmail.com)

Keywords: dynamic programming language, automata-based programming

Object-oriented programming with explicit state emphasis has advantages of object-oriented and automta-based paradigms. Main advantages of such an approach are scalability, flexibility and presence of mechanism for complex behaviour description on the base of finite state machines. Object-oriented and dynamic approaches for inheritance and embedding of automata classes in dynamic programming languages.

A TOOL SUPPORTING AUTOMATA-BASED PROGRAMMING – UNIMOD 2. DESIGN. VALIDATION. VERIFICATION. IMPLEMENTATION

Dmitry Kochelev (kochelaev@rain.ifmo.ru), Ivan Lagunov (lagunov@rain.ifmo.ru), Bulat Khasanzyanov, Bulat Jaminov (jaminov@rain.ifmo.ru)

Keywords: automata-based programming, tool, validation, verification

This article gives a short overview of instrumental tool with automata based programming *UniMod 2*. This tool was designed for development of automata based programs and provides means for visual designing, debugging, validation and verification of automata based programs.

TEXTUAL LANGUAGE FOR AUTOMATA-BASED PROGRAMMING

Vadim Gurov (vadim.gurov@gmail.com), Maxim Mazin (maxim.mazin@gmail.com), Analoty Shalyto (shalyto@mail.ifmo.ru)

Keywords: automata-based programming, textual language

Textual automata-based programming language is described in this paper. This language is built on the base of JetBrains MetaProgramming System (MPS). This language does not have such drawbacks of graphical language as low speed of entering diagrams and is more habitual to programmer.

TEXTUAL LANGUAGE FOR AUTOMATA-BASED PROGRAMMING *FSML* FOR *UNIMOD* TOOL

Ivan Lagunov (lagunov@rain.ifmo.ru)

Keywords: automata-based programming, UniMod tool, textual language

UniMod tool allows designing and running automata-based programs. Its main constraint is graphical language for description of finite state machines. Textual automata-based programming language *FSML* (*Finite State Machine Language*) is described in this paper. An editor for this language was also created. It allows to describe finite state machines in *UniMod* with textual language.

A METHOD OF TESTS FOR API GENERATION ON THE BASE OF FINITE-STATE MODEL OF TESTING

Konstantin Rubinov, Vladimir Vedeneev (vedeneev@rain.ifmo.ru), Vladimir Parfenov (parefenov@mail.ifmo.ru)

Keywords: automata-based programming, testing

An automata-based model approach for testing API is described in this paper. Method proposed is based on constructing graphical automata-based model for testing. After it test-cases are built with transitions graph traversal. Such an approach allows simplifying and formalizing test-case generation not only for functions, but for aggregation of functions.

APPLICATION OF AUTOMATA-BASED APPROACH FOR A MOBILE ROBOT CONTROL SYSTEM

Vitaliy Kleban (vkle@mail.ru), Vladimir Parfenov (parfenov@mail.ifmo.ru), Anatoly Shalyto (shalyto@mail.ifmo.ru)

Keywords: automata-based programming, mobile robot, automatic control

An automata-based approach for design of control system for mobile robot is described in this paper. Automata-based programming is used for creating software for KVARK-M robot on three levels: on the top level three automata are used, on the middle there are five, on the bottom – eleven. Such an approach allows to greatly heighten the quality of software for robots.

APPLICATION OF FINITE STATE MACHINES FOR AUTOMATIZATION OF DOCUMENTS CIRCULATION

Vitaly Kleban (vkle@mail.ru), Fedor Novikov (fedornovikov@rambler.ru)

Keywords: finite state machine, documents circulation

Problem of automatization of documents circulation is considered in this paper. Finite state machines are used for building such a system. Such an approach has some advantages in comparison with traditional approach. Example of this approach – development of a *Microsoft Office*-based tool for automated quality management system for project based production – is described in this paper.

A MACHINE LEARINIGN METHOD FOR COMPLX SYSTEMS WITH A LOT OF INPUT DATA AND OUTPUT ACTIONS

Alexander Krass (krass@rain.ifmo.ru)

Keywords: finite state machine, documents circulation

A machine learning method for complex systems with many input variables and many output actions is described in this paper. This method allows taking into consideration behaviour restrictions. It greatly reduces size of a search space. This method also allows splitting system into subsystems on the base of subproblems. Solutions for subproblems can be united with rather complex conditions on transitions between them.

СОДЕРЖАНИЕ