Опубликовано в Научно-техническом вестнике СПбГУ ИТМО. 2008. Выпуск 53. Автоматное программирование, с. 198-211.

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, кросс-платформенность 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. Однако 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. Спе-

цификации 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.

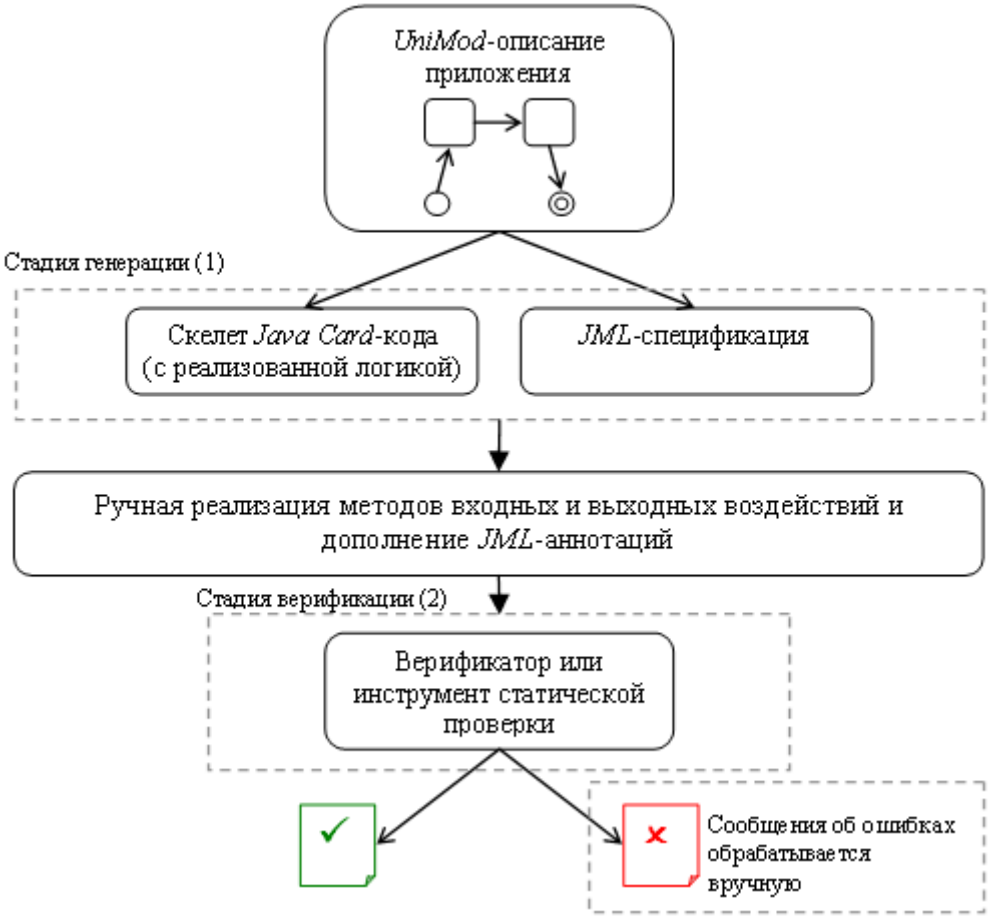


Рис. 1. Схема предлагаемого подхода

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).

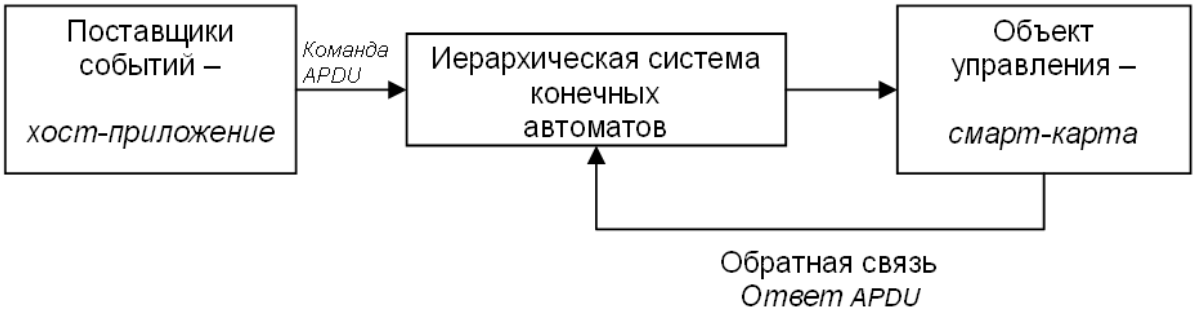


Рис. 2. Интерпретация автоматного подхода для платформы *Java Card*

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.

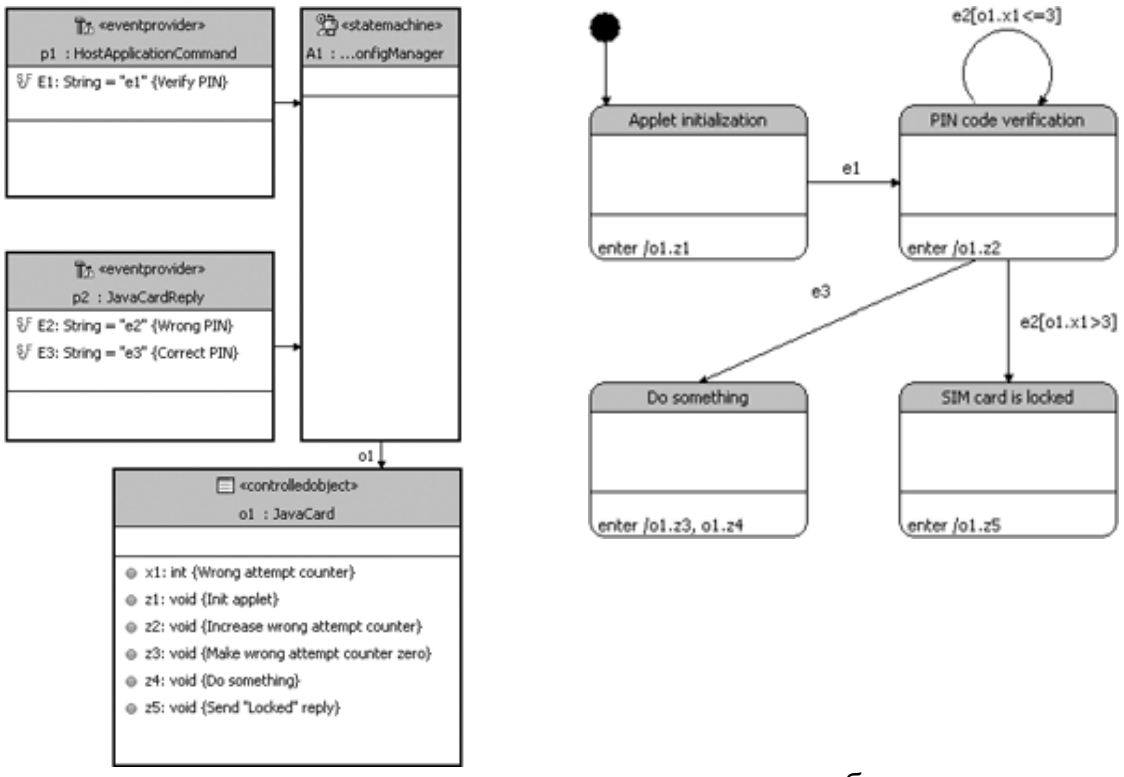
5. Доработать вручную сгенерированный код и *JML*-спецификацию в следующих направлениях:
   - реализовать методы входных и выходных воздействий (стандартный шаг в рамках автоматного подхода),
   - дополнить *JML*-аннотации (в основном для методов входных и выходных воздействий).
6. Верификация может 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) позволяет гарантировать то, что апплет всегда будет находиться в одном из предопределенных графом переходов состояний.



а б

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/