Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

А. Ю. Законов

Применение автоматного подхода при создании *JavaCard*-приложений

Бакалаврская работа

Руководитель – А. А. Шалыто

Санкт-Петербург
2008

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

С каждым днем мобильные устройства занимают все большую роль в повседневной жизни. Их функциональность растет, так же, как и сфера их применения. Ярким примером являются смарт-карты – пластиковые карты со встроенной микросхемой (*ICC, integrated circuit card*). На данный момент, эта технология получила широкое применение в различных областях, от систем накопительных скидок до кредитных и дебетовых карт, телефонов стандарта *GSM*, карт доступа и проездных билетов. Смарт-карты представляют собой достаточно сложное устройство, содержащее микропроцессор, операционную систему, контролирующую устройство и доступ к объектам в его памяти. В связи с этим, для решения разнообразных задач используется платформа *Java Card* [1], позволяющая программам, написанным на языке *Java*, исполняться на интеллектуальных картах и других устройствах с ограниченными ресурсами.

С развитием технологий растет как сложность поставленных задач, так и, соответственно, решающих их программ. При этом повышаются требования к безопасности кода, так как, в случае банковской сферы и сферы мобильных технологий, цена ошибки может быть очень велика, и надежность приложения является ключевым вопросом. Однако технологии выявления ошибок и способы проверки программ развиваются медленнее. Один из способов преодолеть эти трудности – уделять должное внимание этапам проектирования и тестирования.

Известный факт, что *JavaCard*-апплет функционально эквивалентен конечному автомату как событийный объект [2]. Однако, несмотря на это, на данный момент не существует 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 *JavaCard*-приложения в рамках участия в финале конкурса *SIMagine* [4]. Также, в процессе изучения предметной области было выяснено, что все существующие на данный момент средства для вери-

фикации и тестирования апплетов предназначены для проверки полученного *JavaCard*-кода или байт-кода, но никак не описывают процесс разработки. Общая структура существующих подходов изображена на рис. 1.1.

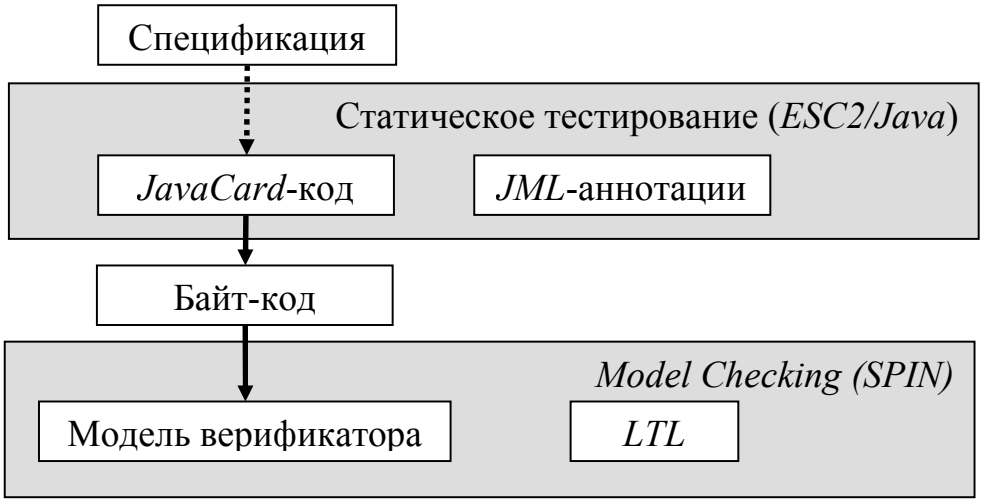

Рис. 1.1. Существующие подходы к разработке *JavaCard*-апплетов

Разработка спецификации, требований и написание кода происходят абсолютно независимо. Как результат спецификация содержит ограниченный набор требований, обычно не позволяющий сделать вывод о корректности кода. Таким образом, существует большой разрыв между спецификацией и реализацией поставленной задачи.

В связи с этим в данной работе поставлена следующая цель: предложить подход и набор инструментов для разработки, позволяющих сократить разрыв между описанием функциональности приложения и реализующим его кодом. Подход должен включать себя все этапы создания *JavaCard*-апплета – начиная с проектирования поведения апплета и заканчивая проверкой реализации на предмет соответствия построенной модели.

Учитывая общепризнанную схожесть апплета с конечным автоматом, в качестве способа проектирования предлагается использовать автоматный подход. Однако, все преимущества, достигнутые на этапе проектирования, окажутся бесполезными, если не будет гарантировано соответствие конечной реализации и автоматной модели. В связи с этим, для сокращения разрыва между моделью и кодом приложения необходимо создать инструмент для генерации

прототипа кода. Сгенерированный код будет отвечать за поведение апплета и обработку поступающих сообщений – таким образом будет гарантировать соответствие поведения разрабатываемого апплета и спроектированного конечного автомата. К сожалению, для целого ряда задач, функциональности полученного таким образом кода будет недостаточно. Некоторый процент кода, во-первых, удобнее, а, во-вторых, иногда просто необходимо, дописывать руками. На этом этапе неизбежно, происходит разрыв между существующей спецификацией и законченным апплетом. Для сокращения этого разрыва предлагается расширить автоматную модель возможностью добавлять утверждения (*assertions*), которые будут накладывать некоторые ограничения и требования на тот код, который будет добавлен вручную. Это позволит, проверив код апплета, сделать вывод, насколько финальный код соответствует спроектированному поведению. Общая идея предлагаемого подхода проиллюстрирована на рис. 1.2. Статическое тестирование *JavaCard*-кода изучено в работе А. Клебанова [5].

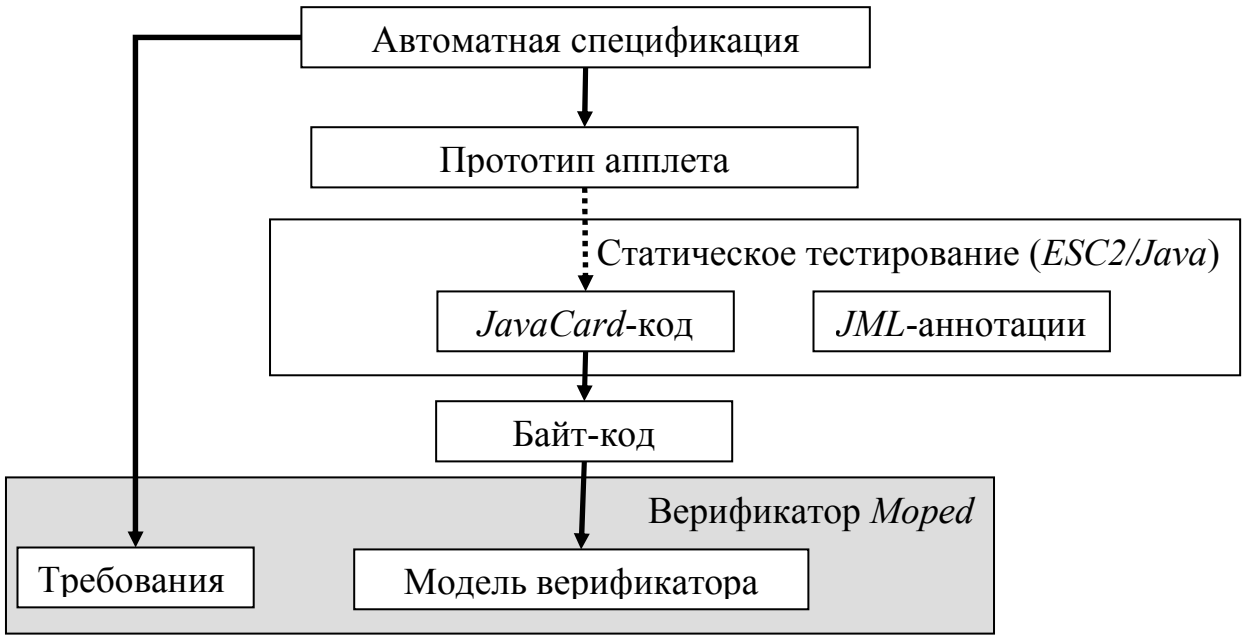

Рис. 1.2. Предложенный подход к разработке *JavaCard*-апплетов

Сформулируем основные задачи, которые следует решить для достижения цели:

- сформулировать особенности проектирования *JavaCard*-апплетов в рамках автоматного подхода;
- предложить расширенное утверждениями описание модели;
- написать генератор прототипа кода;
- создать инструмент для автоматической проверки байт-кода на предмет соответствия автоматному описанию при помощи существующих верификаторов.

Описанный подход будет объединять в себе преимущества автоматного программирования, автоматической генерации кода [6] и техники автоматической проверки моделей программ *Model checking* [7,8]. Для построения автоматной модели предлагается использовать инструментальное средство *UniMod* [9]. Успешное применение всех перечисленных технологий позволит существенно автоматизировать и упростить процесс разработки, а также соединить процессы написания кода и написания документации гармоничным образом.

В главе 1 будут описаны архитектура платформы *Java Card* и ее особенности, ключевые моменты задействованных технологий, их применимость для использования в рамках этой платформы. Также глава содержит обзор существующих исследований, разработок и достижений в этой области.

Глава 2 посвящена детальному описанию предлагаемого подхода. В ней будет рассмотрено использование предложенных в рамках этой работы инструментов и будут сформулированы особенности проектирования *JavaCard*-апплетов с использованием автоматного подхода.

Глава 3 иллюстрирует использование предложенного метода на примере создания двух апплетов, разработанных с применением описанного во второй главе способа разработки. Первое приложение является примером апплета, прилагающимся к набору разработчика *JavaCardDevKit 2.2.2* [10]. С его помощью карта используется в качестве электронного кошелька. Второй апплет был разработан в рамках полуфинала конкурса *SIMagine* – апплет, позволяющий расширить возможности приложении по отправке *SMS*-сообщений.

ГЛАВА 1. ОБЗОР ИСПОЛЬЗУЕМЫХ ТЕХНОЛОГИЙ

Платформа *Java Card* является подмножеством языка *Java*. Благодаря своей ограниченности и простате по сравнению с *Java SE* или *Java ME*, апплеты платформы *Java Card* являются хорошими объектами для разного рода методов валидации и верификации.

В данной главе будут описаны особенности платформы *Java Card*, пояснен тот факт, что апплет функционально эквивалентен конечному автомату как событийный объект, в связи с этим будет обосновано удобство использования автоматного подхода при разработке приложений для этой платформы. Затем будут рассмотрены стандартный подход к тестированию программ и техника автоматической проверки моделей программ *Model checking*.

1.1. Платформа *Java Card*

Java Card позволяет программам, написанным на языке *Java*, исполняться на интеллектуальных картах и других устройствах с ограниченными ресурсами. Как наиболее известные примеры можно привести телефонные *SIM*-карты и банковские *ATM*-карты.

Наиболее полное описание особенностей данной платформы можно найти на официальном сайте [1], платформа *Java Card* включает в себя три части:

1. Виртуальная машина *Java Card Virtual Machine (JCVM)*. Спецификация определяет подмножество языка *Java* и спецификацию *JVM*, подходящую для смарт-карт.
2. Программные интерфейсы *Java Card Application Programming Interface (API)*. Спецификация описывает *Java*-пакеты и классы для программирования смарт-карт.
3. Среда исполнения. *Java Card Runtime Environment (JCRE)*.

Приложения, написанные на основе платформы *Java Card*, называются апплетами. Название было выбрано благодаря схожести модели выполнения со стандартными апплетами, исполняемыми в виртуальной машине *Java (JVM)*

веб-браузера. Помимо языка, платформа *Java Card* поддерживает среду исполнения смарт-карт приложений, позволяя одному и тому же *JavaCard*-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 явно определенных библиотек исполнения, что позволяет абстрагироваться от различий между смарт-картами. Среда исполнения решает вопросы работы с памятью карты, передачи данных, безопасности и выполнения программ. Для *Java Card* эта среда соответствует стандарту *ISO 7816* [11].

Важными преимуществами технологии *Java Card* являются безопасность и портируемость. Безопасность данных апплета гарантируется такими аспектами технологии, как:

1. Приложения – апплеты. Апплет – конечный автомат [2], который получает сигнал на вход, меняет состояние в соответствии с запросом, отвечает отправкой отчета о выполненных действиях.
2. Криптография. Реализована поддержка алгоритмов шифрации, таких как *DES*, *3DES*, *RSA*.
3. Защитная система апплета (*Applet Firewall*). Приложения на карте разделены при помощи защитной системы, которая контролирует и ограничивает доступ к данным апплетов.
4. Инкапсуляция данных. Вся необходимая информация хранится внутри апплета. Выполнение происходит в изолированной от операционной системы и устройства среде (*Java Card Virtual Machine*).

В силу ограничений по доступной памяти и вычислительной мощности, платформа *Java Card* поддерживает только некоторое подмножество языка программирования *Java*. Все языковые конструкции *Java Card* существуют в *Java* и выполняются идентично. На данный момент многие возможности языка *Java* не присутствуют в *Java Card*:

- длинные примитивные типы данных: `long`, `double`, `float` (в общем случае `int`);
- символы и строки;
- многомерные массивы;
- динамическая загрузка классов;
- сборка мусора;
- многопоточность;
- сериализация и клонирование объектов.

Вместе с тем, это по-прежнему объектно-ориентированный язык на основе *Java*:

- короткие примитивные типы данных: `boolean`, `byte`, `short`;
- одномерные массивы;
- пакеты, классы, интерфейсы и исключительные ситуации;
- объектно-ориентированные свойства *Java*: наследование, виртуальные функции, перегрузка методов, динамическое создание объектов, области видимости.

JavaCard-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 *JVM*, но использует другой принцип сжатия, из соображений уменьшения объема байт-кода.

Полноценное *JavaCard*-приложение состоит из нескольких элементов:

1. Клиентская часть.

Обеспечивает сервисы для поддержки апплетов на карте, такие как доступ к необходимой информации и подсистемам.

2. Интерфейсное устройство.

Это может быть устройство для считывания карт, телефон, платежный терминал. Обеспечивает связь апплета с пользователем и с клиентской частью.

3. Апплеты и их среда.

Платформа *Java Card* предусматривает поддержку нескольких апплетов. На карте размещаются апплеты, операционная система карты и среды исполнения *JCRE*.

Все апплеты расширяют базовый класс *Applet* и должны содержать следующие методы:

1. `install` – первый метод апплета, который будет вызван. На момент его вызова объект еще не существует. Инициализирует объект класса *Applet*, вызывая конструктор, регистрирует созданный объект в системе.
2. `select` – любой апплет в системе находится в пассивном состоянии ожидания, пока не будет явно выбран при помощи команды `select`. При выборе нового активного апплета, текущий апплет получит команду `deselect`. В случае если апплет не может быть активизирован по каким-либо причинам, функция возвращает значение `false` или создает исключение (*ISOException*). Если функция `select` успешно завершила свою работу и вернула значение *true*, то происходит вызов метода `process` с параметром `SELECT FILE APDU`.
3. `process` – первоначально все *APDU-команды* [12] принимаются средой *JCRE*, которая, получив команду, делает ее препроцессинг, создает объект класса *APDU* и передает этот объект методу `process` активного на данный момент апплета. Метод `process` может инициировать исключение типа `ISOException` в случае ошибки исполнения.

4. `deselect` – вызывается при получении команды `SELECT FILE APDU`. Даже, если получено указание выбрать уже активный апплет, вначале будет вызвана функция `deselect`, а после нее функция `select`.

5. `uninstall` – стирает апплет с карты.

Взаимодействие апплета и среда проиллюстрировано на рис. 1.3.

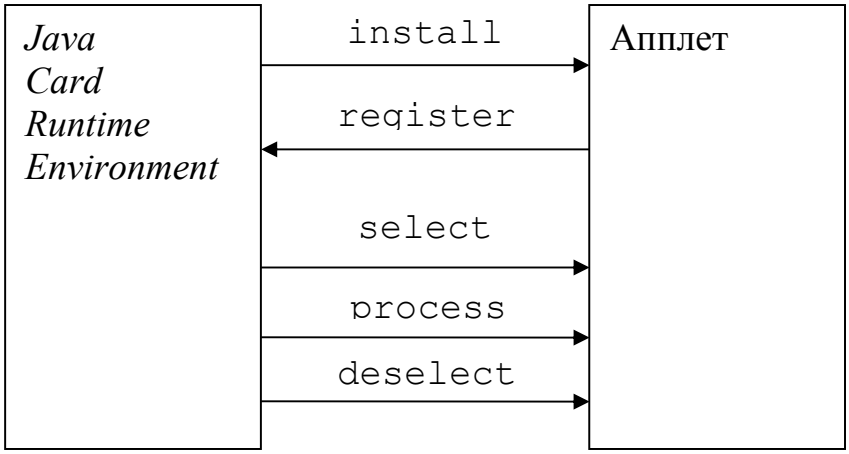

Рис. 1.3. Взаимодействие среды *JCRE* и апплета

Апплет активен только после получения *APDU*-команды. Модель, использующая сообщения, является основным способом коммуникации. Она основана на использовании пакетов *Application Protocol Data Unit (APDU)* – логических пакетов для обмена данными апплета и *Java Card Framework*. Среда получает *APDU*-команду и передает ее выбранному апплету. Апплет выполняет *APD*U-команду и отсылает ответное *APDU*-сообщение.

Используются *APDU* в соответствии со стандартом *ISO/IEC 7816-3* и *7816-4* [11]. Структура *APDU*-команды представлена в табл. 1.1.

Таблица 1.1. Структура *APDU*-команды

APDU-команда						
Заголовок (обязательная часть)				Данные (опциональная часть)		
CLA	INS	P1	P2	Lc	Data Field	Le

APDU-команда определяется по значению первых байт:

1. `CLA` – класс инструкции.
2. `INS` – код инструкции.
3. `P1` – параметр инструкции 1.
4. `P2` – параметр инструкции 2.
5. `Lc` – размер переданной информации в байтах.
6. `Data` – переданная вместе с командой информация.
7. `Le` – максимально разрешенное количество байт в ответе.

При каждом входящем *ADPU*-сообщении для выбранного апплета среда *JCRE* вызывает метод *process* и передает сообщение как аргумент. Апплет должен распознать команду, выполнить запрашиваемое действие, сгенерировать ответное *APDU* и вернуть управление среде исполнения.

1.2. Автоматный подход. Применимость в сфере мобильных устройств

Платформа *Java Card* работает по принципу «клиент-сервер» [13]. Апплет реагирует на получение *APDU*-команды, выполняет действия, затем, в аналогичной форме, отсылает ответ на запрос. Несложно заметить, что такая структура напоминает поведение конечного автомата, который реагирует на событие, и, в зависимости от типа события, текущего состояния и набора 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 трех понятий: состояния, входные воздействия, выходные воздействия. 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. Эти значения представляют уникальное состояние, которое изменяется каким-либо внешним событием. В любой момент времени приложение находится в каком-то определенном состоянии. Таким образом, концепция автоматного программирования хорошо подходит для создания *JavaCard*-апплетов.

Автоматный подход включает в себя много преимуществ – продуманное применение конечных автоматов облегчает организацию и написание, как логики пользовательского интерфейса, так и логики приложения. Обе упомянутые задачи являются критичными в случае мобильных устройств из-за сильно ограниченного количества системных ресурсов и необходимости эффективно использовать пространство экрана в случае, например, мобильных телефонов. Применимость конечных автоматов для программирования мобильных устройств обоснована в работе [14]. Помимо удобств при написании и проектировании, выбор автоматного подхода будет существенным преимуществом при последующем тестировании полученного приложения.

Особенность автоматного подхода заключается в том, что построение программы предлагается начинать с построения схемы связей, содержащей источники информации, систему управления, объекты управления и обратные связи от объектов к системе управления. *SWITCH*-технология предполагает для каждого автомата два типа диаграмм: схему связей и граф переходов. При наличии нескольких автоматов строится схема их взаимодействия. Для создания таких диаграмм предложена соответствующая нотация [15].

Инструментальное средство *UniMod* автоматизирует процесс создания модели системы и позволяет генерировать *Java*-код по модели. При этом автоматически выполняется проверка корректности построенной модели. В случае

обнаружения, ошибок пользователю предлагаются способы их устранения. В дальнейшем производится интеграция части программы, построенной автоматически, с фрагментами кода, написанными вручную. Это обеспечивает построение программы в целом.

В работе [16] процесс создания программных систем подразделяют на следующие этапы:

1. Постановка задачи. Включает в себя сбор требований и создание прототипа программы
2. Технический дизайн. Состоит в создании на основе собранных требований технического задания, описывающего модель системы с помощью диаграмм, псевдокода и поясняющего текста.
3. Создание кода. Состоит в реализации системы для целевой программно-аппаратной платформы в соответствии с техническим дизайном.
4. Тестирование. Завершает цикл разработки и представляет собой отладку созданного приложения и проверку соответствия реализации собранным требованиям.

Процесс создания *JavaCard*-апплета также состоит из указанных этапов. Для этапов постановки задачи и технического дизайна *UniMod* применим, как и в случае с *Java SE*. Этапы создания кода и тестирования с учетом особенностей платформы *Java Card* будут рассмотрены далее. *UniMod* позволяет проектировать автоматную модель. Помимо графического представления модели, можно рассматривать *XML*-представление модели. Данное представление соответствует *DTD (Document Type Definition)* описанию [17]. Каждый тег `StateMachine` описывает конечный автомат. Основные его элементы:

- `state` – состояния конечного автомата;
- `transition` – переходы конечного автомата.

Выходные воздействия описываются при помощи тега `outputAction`, содержащего указание на функцию объекта управления. Условия совершения

переходов хранятся в атрибуте `guard` и соответствуют указанной в работе [9] грамматике.

1.3. Автоматическая генерация кода

В работе [18] этот подход определяется так: код программы не пишется вручную, а создается автоматически программой-генератором на основе другой, более простой программы. При этом часть кода теряет содержательный смысл и становится лишь механическим выполнением правил. Когда эта часть становится значительной, возникает мысль задавать вручную лишь содержательную часть, а остальное добавлять автоматически. Вопрос автоматической генерации кода рассматривается в рамках порождающего программирования (*Generative Programming*) [6].

Автоматическая генерация кода позволяет сократить разрыв между этапом проектирования и этапом написания исполняемого кода. Отсутствие связи между документацией и кодом приложения является одной из важных проблем надежности программ, так как это приводит к тому, что корректная и понятная документация никак не гарантирует корректность написанного кода. В случае, когда основная логика приложения генерируется из спецификации, например, автоматного описания, а вручную дописывается реализация только конкретных функций управления, корректность документации с большей вероятностью гарантирует корректность приложения в целом.

Среди существующих инструментов для генерации кода в рамках автоматного подхода следует отметить, уже упомянутое выше инструментальное средство *UniMod* [19] и *MetaAuto* [20].

1.4. Проверка моделей и тестирование программ

При традиционном подходе, тестирование программ и проверка моделей являются далекими друг от друга методологиями. Тестирование – наиболее распространенное средство проверки надежности программ. Обычно оно за-

ключается в запуске программы с некоторым фиксированным набором параметров и проверке соответствия полученных результатов ожидаемым. Таким способом можно покрыть тестами сравнительно небольшой процент кода, поэтому тестирование направлено на выявление ошибок, а не на доказательство их отсутствия.

Model checking [7, 8] – это метод, при котором алгоритмически доказывается, соответствует ли формальная система спецификации или некоторому свойству. В этом случае обычно рассматриваются системы состояний и переходов, представляющие тестируемую систему. Таким образом, проверяют достижимость определенного состояния или же более сложные свойства системы. Проверяемые свойства могут формулироваться в темпоральной логике, которая позволяет определять не только мгновенное состояние системы, но также и историю развития системы во времени. Метод проверки моделей стремится показать корректность программы, ценой очень долгих (или даже невыполнимых) вычислений. *Model checking* реализуется при помощи построения *модели Крипке*, которая представляет собой полный граф переходов между элементарными состояниями модели.

В работе [7] описывается верификация автоматных программ на основе метода *Model checking*. Основная трудность данного подхода заключается в моделировании – создании модели реальной системы. Модель не всегда полно отражает поведение программы, так как при ее создании разработчик абстрагируется от несущественных свойств. *Model checking* алгоритмы обычно осуществляют полный просмотр пространства состояний модели, для каждого состояния проверяется, соответствует ли оно требованиям. Если модель соответствует заданным требованиям, то верификатор сообщает об этом. В случае обнаружения ошибки, она выводится в виде контрпримера, который показывает, при каких условиях возникло несоответствие.

Подробный обзор существующих в этой области разработок приведен в работе [21]. Среди существующих верификаторов стоит выделить *Spin* [22] и

Bogor [23]. *Spin* производит верификацию модели для языка *Promela*. Этот язык поддерживает дискретные типы данных и функции работы с многопоточностью, использует темпоральную логику (*LTL, Linear Temporal Logic*). Утверждается, что он приспособлен для решения крупномасштабных задач. *Bogor* производит проверку модели для собственного языка *Bir*, поддерживает большинство основных способов работы с многопоточностью.

Отдельно следует выделить подход к *Model checking*, при котором построение модели производится автоматически из кода программы. В этом случае на вход берется исполняемый байт-код и по нему строится модель. Этот подход используется, например, в инструменте *Java Pathfinder* [24], который верифицирует исполняемый *Java* байт-код, транслируя его во входной язык *Promela* верификатора *Spin*. Он может находить пути выполнения, которые ведут к необработанным исключениям, дедлокам и прочим возможным ошибкам. По сведениям на официальном сайте, этот верификатор проверяет программы до 10 000 строк кода и применяется в *NASA Ames Research Center*. Еще одним инструментом, ориентированным на извлечение моделей из исходных кодов, написанных на языке *Java*, является *Bandera Project* [25]. Поддерживает возможность перевода естественных языков в логические утверждения. Последние данные о проекте заканчиваются в 2005 году.

Также этот подход лежит в основе инструмента *jMoped translator* [26] – транслятор исполняемого *Java*-кода в язык *Remolpa*, для последующей проверки при помощи верификатора *Moped* [27]. На данный момент инструмент активно развивается и существует плагин для разработки в среде *Eclipse*.

В последнее время появились инструменты, использующие подход проверки моделей в целях тестирования программ. Такой возможностью обладает *jMoped* – среда тестирования для программ, написанных на языке *Java*. Для заданного метода, *jMoped* может эмулировать его выполнение для всех возможных аргументов в конечном интервале. Инструмент позволяет проверять наличие ошибок в *Java*-коде: стандартные исключения языка *Java*, утверждения (*assertions*), пред- и постусловия. В случае обнаружения ошибки, выводится

контрпример – набор аргументов и последовательность вызовов. *jMoped* поддерживает большинство фундаментальных возможностей языка *Java*. Среди ограничений, на данный момент, можно назвать отрицательные числа, числа с плавающей запятой (тип `float`) и многопоточные программы. Среди аналогичных инструментов можно назвать *BLAST Project* [28], нацеленный на проверку программ на языке *C*.

В работе [29] среди аргументов в пользу такого подхода приводится то, что даже при небольших границах перебора входных переменных, все значения внутри этих интервалов будут проверены, что позволит протестировать много граничных примеров входных данных и проверить устойчивость программы к таким параметрам. *jMoped* преобразовывает программу во входной язык *Remolpa* верификатора *Moped*, строя *граф потока управления (control flow graph)*, но моделируя только конечный объем данных. Размер переменных и размер динамической памяти и задаются пользователем. В стандартном режиме работы при помощи верификатора *Moped* проверяется полученная на вход модель на предмет достижимости выбранного состояния (*reachability analysis*), путем перебора всех возможных значений аргументов в заданных интервалах и, соответственно, всех возможных путей исполнения. Второй режим работы – обратный, для конкретного заданного условия проверяет из каких состояний можно его достичь. Итоги проверки выводятся в виде трассировки вызовов или *JUnit*-теста.

Помимо проверки общих ошибок, таких как исключения, можно расширить проверку корректности, добавив в код программы необходимые утверждения (*assertions*) – пред- и постусловия, для проверки этим инструментом. Верификатор позволит проверить достижимость состояния, сигнализирующего нарушение условий. Первая версия верификатора *Moped* также позволяет проверить, удовлетворяет ли модель свойствам, описанным при помощи темпоральной логики.

Учитывая ограничения платформы *Java Card*, такие как отсутствие поддержки многопоточности, чисел с плавающей запятой, очень небольшой объем данных и сильные ограничения на размер переменных, подробно описанные в разд. 1.1, проверка *JavaCard*-апплетов при помощи *jMoped* выглядит значительно более перспективно, чем полноценных приложений *Java SE*, для которых функциональность *jMoped* покрывает меньший спектр задач.

1.5. Существующие инструменты

Одна из важных задач, поставленных в данной работе – генерация *JavaCard*-кода. Среди существующих проектов и разработок в этой области, попытка решить эту задачу предпринята в рамках *Kestrel Project*, генератором *AutoSmart* [30]. Этот проект предлагает следующий подход к разработке *JavaCard*-апплетов: вся программа пишется на предложенном в рамках этого проекта языке *DSL (domain-specific language)* и языке спецификаций *MetaSlang*, которые являются более высокоуровневыми, чем язык *Java Card*. Затем, код апплета автоматически генерируется из этой спецификации. Для проверки написанной спецификации используется верификатор.

Другой существующий подход к генерации *JavaCard*-апплетов – использование *B*-метода для написания спецификации преложен в работе [31]. Общая идея подхода выглядит аналогично – весь апплет описывается при помощи предложенного формального языка, затем из этой спецификации генерируется код. Существует набор инструментов для проверки корректности спецификации.

Еще одна поставленная задача – верификация и проверка *JavaCard*-кода. Среди работ в этой области следует отметить проект *Verificard* [32], в рамках которого разработан целый набор инструментов для верификации апплетов и самой платформы. Часть проектов использует систему формальных доказательств *Coq*: *CertiCartes* формализует *Java Card* платформу в формате *Coq*, *Jakarta* проверяет, что байт-код апплетов соответствует сформулированным в рамках проекта *CertiCartes* требованиям платформы.

Отдельно следует рассмотреть инструмент *jCave* [33] в рамках проекта *Verificard*. *jCave* – это система для верификации совместной работы апплетов на карте. Из байт-кода апплета генерируется входной язык *Promela* верификатора *Spin*. Инструмент разработан для проверки взаимодействия между двумя и более апплетами. Необходимые правила предлагается задавать вручную при помощи темпоральной логики. Последний опубликованный результат в этой области датирован 2002 годом.

Среди предлагаемых подходов в рамках *Verificard* можно отметить использование *JML*-аннотации для описания методов в *JavaCard*-коде. В этом случае проверка является неполной, но может показывать предупреждения о возможных нарушениях *JML*-спецификации. Предлагается использовать верификатор *ESC/JAVA2* [34] и написанный в рамках проекта инструмент *ChAsE* [35] расширяющий возможности проверки.

Интересным инструментом является *AutoJML* [36] – на данный момент, единственная разработка, учитывающая автоматную природу *JavaCard*-апплета. Он позволяет генерировать *JML*-аннотации из конечных автоматов, описывающих функциональность апплета. На вход *AutoJML* принимает *XML*-описание автомата, в результате своей работы выдает набор *JML*-спецификаций, характеризующих свойства этого конечного автомата и элементы *Java*-кода. Среди достоинств этого подхода можно отметить то, что предлагается использовать *UML*-диаграммы для написания спецификации поведения апплета.

Подход, описанный в данной работе, предлагает строить входную модель верификатора, основываясь на байт-коде приложения. Помимо проекта *jCave*, описанного выше, подобный подход для *JavaCard*-кода предлагается в проекте *Bandera* [25] – предпринимается попытка строить модели, основываясь на *Java*-коде и *JavaCard*-коде. Есть возможность проверять эти модели на предмет соблюдения свойств написанных на предложенном языке спецификации. Последние данные о проекте датируются 2005 годом.

Также существуют проекты, рассчитанные на использование динамической логики для написания требуемых свойств модели. Например, система спецификации и верификации *KIV* [37] позволяет проверять модель на выполнение описанных при помощи динамической логики свойств. Требует построения специальной входной модели апплета и написания спецификации с использованием предложенного языка. Вторым проектом в этой области является *KeY Project* [38]. Помимо свойств, описанных динамической логикой, позволяет проверять соблюдение *JML*-аннотаций.

1.6. Анализ недостатков

Инструментальное средство *UniMod* позволяет проектировать приложения в терминах автоматного подхода, явно указывать схему переходов, поставщики событий и объекты управления. Также *UniMod* позволяет генерировать прототип кода приложения по составленной модели.

Используя преимущества генерации кода, на выходе пользователь получает прототип программы, который часто не обладает необходимой функциональностью. Это связано с тем, что некоторые требуемые действия невозможно или очень сложно описать на диаграмме классов или при помощи переходов в структуре автомата. В качестве примера можно привести вызов функций библиотек или математические вычисления, которые проще реализовать вручную. Таким образом, в надежный код, сгенерированный автоматически, почти неизбежно, пользователь добавляет свою логику, вызовы функций и вычисления. После этого этапа уверенность в корректности кода пропадает, так как нет гарантии, что добавленные вычисления верны. Поведение приложения, ранее формально описанное при помощи состояний и переходов между ними, теряет свою предсказуемость.

Сформулируем недостатки существующих для платформы *Java Card* инструментов:

1. Все существующие исследования направлены на верификацию уже написанного *JavaCard*-кода или байт-кода, а не на процесс разработки.

Этапам проектирования и написания кода не уделяется достаточного внимания, несмотря на их значимость.

2. Полностью генерировать код апплета удобно, но это требует написания сложной спецификации на предложенном языке, и, в итоге, генерация не намного упрощает процесс разработки. Ошибки в сложной спецификации тяжело обнаружить. Это приводит к ошибкам в коде. Удобнее сочетать автоматически сгенерированный код и написанный вручную. Такой подход к написанию *JavaCard*-апплетов на данный момент не предложен.
3. Несмотря на тот факт, что *JavaCard*-апплет функционально эквивалентен конечному автомату как событийный объект, нет инструментов, позволяющих использовать эту особенность при проектировании, разработке или тестировании.
4. Для использования формальных верификаторов требуется написание спецификации в специальном формате и построение модели, достаточно хорошо описывающей реализацию. Во-первых, это сложный и трудоемкий процесс, во-вторых, такие спецификации не всегда позволяют учесть все требования. Также, модель не всегда достаточно точно описывает реализацию. Например, *jCave* проверяет только поток управления, но никак не учитывает данные и задействованные переменные.
5. Использование *JML* является хорошим примером проверки соответствия реализации и спецификации. Написание *JML*-аннотации к функциям является отдельной задачей, фактически происходящей на этапе написания кода. Значительно удобнее писать требования к коду на этапе проектирования – такой возможности на данный момент нет.

1.7. Постановка задачи

С учетом описанных ранее особенностей предметной области и существующих инструментов можно сформулировать следующую цель: разработать

подход, позволяющий повысить надежность процесса создания апплетов для платформы *Java Card*. Поясним задачу, разбив ее на подзадачи:

1. Предложить способ применения автоматного подхода для удобного создания модели разрабатываемого приложения. Сформулировать связь между понятиями события и состояния, существующими в платформе *Java Card* и рассматривающимися в контексте автоматного подхода и, в частности, инструментального средства *UniMod*.
2. Расширить понятие модели, получаемой в результате проектирования с использованием инструмента *UniMod*, с целью более подробного описания ожидаемой реализации. Предложить способ внесения в полученную модель утверждений (*assertion*), которые позволят формулировать требования к реализации объектов управления.
3. Создать генератор кода *JavaCard*-апплета, принимающий на вход предложенную модель, содержащую требования к реализации объектов управления.
4. Создать инструмент, позволяющий проверять апплет с реализованными объектами управления на предмет соответствия описанному в автоматной модели поведению и добавленным пользователем требованиям.

Выводы по главе 1

В данной главе были описаны особенности процесса разработки и тестирования приложений для платформы *Java Card*. Также была изучена применимость автоматного подхода для проектирования *JavaCard*-апплетов. Произведенный анализ существующих инструментов и работ в этой области выявил ряд недостатков и направлений для дальнейших исследований. В связи с этим была сформулирована цель данной работы и задачи, которые необходимо решить для ее достижения.

ГЛАВА 2. ПОДХОД, ПОВЫШАЮЩИЙ НАДЕЖНОСТЬ РАЗРАБОТКИ *JAVACARD*-АППЛЕТОВ

Для решения поставленной задачи предлагается использовать возможности ряда существующих подходов – создать инструмент, позволяющий в комплексе использовать их функциональность и тем самым сократить разрыв между спецификацией и реализацией приложения. Для проектирования будет использован автоматный подход и инструментальное средство *UniMod*. Создание прототипа кода будет выполнено при помощи написанного генератора. Для проверки сформулированных требований будет использован транслятор *JavaCard* байт-кода в язык *Remolpa*, являющийся расширением транслятора *jMoped*, и верификатор *Moped Model Checker*.

Данная глава посвящена формулированию подхода, описанию решений и инструментов, разработанных для того, чтобы создать по автоматной модели прототип апплета, построить удобную для проверки входную модель верификатора, сформулировать набор интересных для проверки требований и правильно трактовать полученные результаты анализа. Во втором разделе главы будут сделаны выводы о том, насколько данный подход, позволяет повысить надежность и упростить процесс разработки *JavaCard*-приложений – насколько он решает поставленную задачу. Третий раздел содержит анализ преимуществ, которые предложенный подход имеет, по сравнению уже существующими разработками и инструментами в этой области.

2.1. Предлагаемый подход

Сформулируем общую последовательность действий для разработки *JavaCard*-приложения:

1. Создать автоматную модель приложения в инструментальном средстве *UniMod*.
2. Расширить *XML*-представление модели необходимыми утверждениями, проверяющими те свойства реализации, которые возможно сформулиро-

вать 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 написанный инструмент, позволяющий преобразовать данные контрпримера, содержащие метки байт-кода, в понятную пользователю последовательность событий и переходов между состояниями конечного автомата, спроектированного на шаге 1. Используя полученную информацию об ошибке, исправить возможные недостатки реализации и проанализировать несоответствия автоматного описания и конечной реализации.

Перечисленные этапы разработки и применение предложенных инструментов отражено на рис. 2.1.

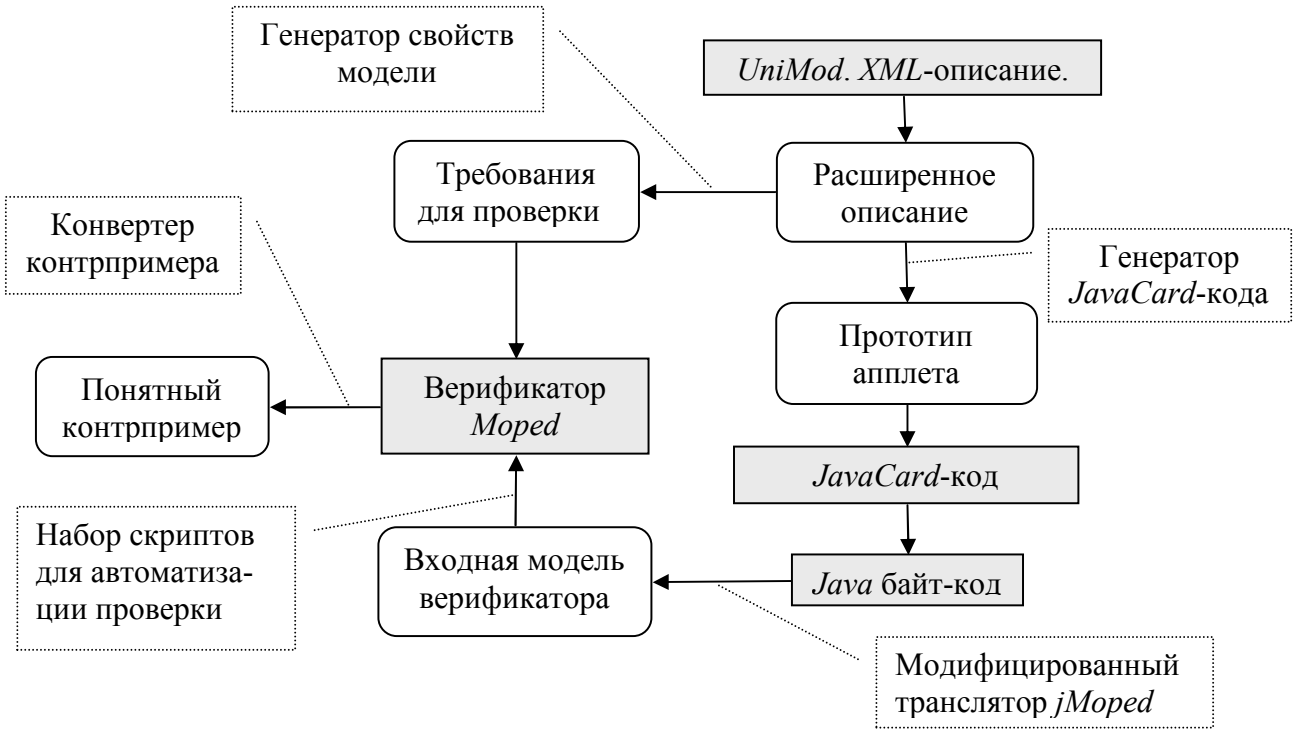

Рис. 2.1. Предложенный подход

Имея средство проверки достижимости состояний и выполнения утверждений *jMoped*, описанное в главе 1, можно проверить уже законченную программу, с реализованными объектами управления, на соблюдение условий следующего типа: достижимость состояния 2 из состояния 1, для заданного интервала значений задействованных переменных. Таким образом, правильное применение этого инструмента позволит проверить, сохранилась ли достижимость описанных в диаграмме состояний и возможную последовательность переходов, с учетом реализации. Получив такую информацию, можно проанализировать корректность добавленного кода и соответствие описанной модели и аплета, полученного в итоге.

С точки зрения тестирования, использование указанного инструмента позволит проверить значительную часть возможных вариантов развития событий. Например, имея конечное количество состояний и набор возможных событий, описанных в *UniMod*-диаграмме, легко проверить, что будет происходить в любом состоянии при любом событии. При неавтоматизированном тестировании, такая простая проверка потребует, либо написания дополнительного кода,

либо, в случае если поставщиком событий является пользовательский интерфейс, работы по генерации всех возможных действий пользователя.

Если же 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.

В данном разделе подробно рассмотрим предложенные решения и поясним каждый из перечисленных этапов разработки.

2.1.1. Проектирование автоматной модели

Для построения удобной автоматной модели апплета следует определиться с терминами и понятиями, существующими как в *Java Card*, так и в автоматном подходе. Наиболее сложный вопрос – провести аналогию и найти связь между понятиями события. Для *JavaCard*-апплета событием является получение *APDU*-сообщения, которое по своей сути является массивом байт. В главе 1 было описано, что апплет различает приходящие события следующим образом:

- первый байт, `CLA` – класс инструкции. В *ISO7816* точно определено только одно значение – ноль, в том случае, если это команда `SELECT APDU`;
- второй байт, `INS` – конкретная инструкция. Ограничений на значение этого байта нет;
- остальные байты содержат параметры инструкции и дополнительную информацию, переданную сервером.

Для удобства использования автоматного подхода предлагается решение, позволяющее абстрагироваться от написания кода обработки событий вручную и генерировать его из поставщика событий, описываемого в инструменте *UniMod*. Каждому поставщику событий будем сопоставлять определенное значение `CLA`-байта:

```
// code of CLA byte for event provider p1
final static byte p1_CLA =(byte)0xB0;

// code of CLA byte for event provider p2
final static byte p2_CLA =(byte)0xC0;
```

Каждому конкретному событию, в рамках одного поставщика событий, уникальное значение INS-байта:

```
// codes of INS byte for event provider p1
final static byte e1 = (byte) 0x20;
final static byte e2 = (byte) 0x30;
final static byte e3 = (byte) 0x40;
final static byte e4 = (byte) 0x50;

// codes of INS byte for event provider p2
final static byte e5 = (byte) 0x20;
final static byte e6 = (byte) 0x30;
```

Использование предложенной методики позволит создавать до 216 событий. Можно считать, что ограничение является более чем достаточным для любой автоматной модели, спроектированной не только в *UniMod*, но и вообще для платформы *Java Card*. Следует понимать, что, хотя клиентские приложения не рассматриваются в контексте данной работы, принятая система разделения событий и представления их в формате *APDU* должна быть учтена не только при разработке сервера, но и клиентского приложения, например *JavaME*-апплета, в случае телефонных *SIM*-карт.

Вторым важным фактором является понятие состояния. Данный подход предлагает использовать в коде апплета именно те состояния, которые описаны в автоматной модели. Обычно код главного метода *process* апплета выглядит следующим образом, как это описано в статье [39]:

```
switch (buffer[ISO7816.OFFSET_INS]) {
      case value1:   method1(apdu);
```

```
                    return;
        case value2:   method2(apdu);
                    return;
        case value3:   method3(apdu);
                    return;
        case value4:   method4(apdu);
                    return;
    }
```

При явном появлении понятия состояния, код метода *process* будет выглядеть следующим образом:

```
switch (fsm_state) {
            case S1:
                switch (event) {
                    case e1:
                    break;
                ...
                }
                break;
            case S2:
            switch (event) {
                    case e3:
                    break;
                ...
                }
        ...
}
```

В коде появилась явно заданная переменная `fsm_state`, содержащая информацию о текущем состоянии, также усложнилась `switch/case` структура. Это почти не влияет на сложность и размер (*footprint*) исполняемого кода.

Этот подход гарантирует, что метод будет вызван только в допустимом состоянии, что является очень важным условием безопасности апплета. Полученная реализация метода *process* гарантирует корректный граф потока управления для апплета в соответствии со спроектированным в рамках автоматной модели.

2.1.2. Расширенная автоматная модель апплета

Несмотря на перечисленные ранее преимущества автоматного подхода, большинство апплетов обладают функциональностью, описывать которую только при помощи автоматной модели неудобно. В связи с этим, разумным решением является сочетание автоматически сгенерированного кода и написанного вручную. Сгенерированный код обеспечит правильность выполнения и корректность вызовов функции, а код, реализующий математические вычисления или работающий с памятью, удобнее дописать руками. При таком подходе, возникает важный 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 дописанный вручную код, никаких выводов сделать нельзя. Возникшую проблему можно частично решить, если на этапе проектирования автоматной модели не только определять, какие методы объектов управления вызывать и в какой момент, но и дополнительно указывать требования к реализации этих методов.

Множество важных требований можно сформулировать уже на этапе разработки модели. Например, если функция объекта управления реализует операцию возведения числа в квадрат, то уже на момент описания этой функции на диаграмме, можно указать требования к задействованной переменной: после выполнения функции значение должно быть неотрицательным. Стоит отметить, что проверку этого требования можно реализовать и при помощи диаграммы переходов. В этом случае, необходимо понимать, что если ключевые особенности функциональности удобно и стоит отображать при помощи

состояний и переходов, то отображение всех таких требований на схеме переходов конечного автомата, сделает эту схему малочитаемой и запутанной.

Предлагается расширить существующую модель возможностью добавлять свои условия, выполнимость которых будет проверена на этапе автоматического тестирования. Описанный подход проиллюстрирован на рис. 2.2.

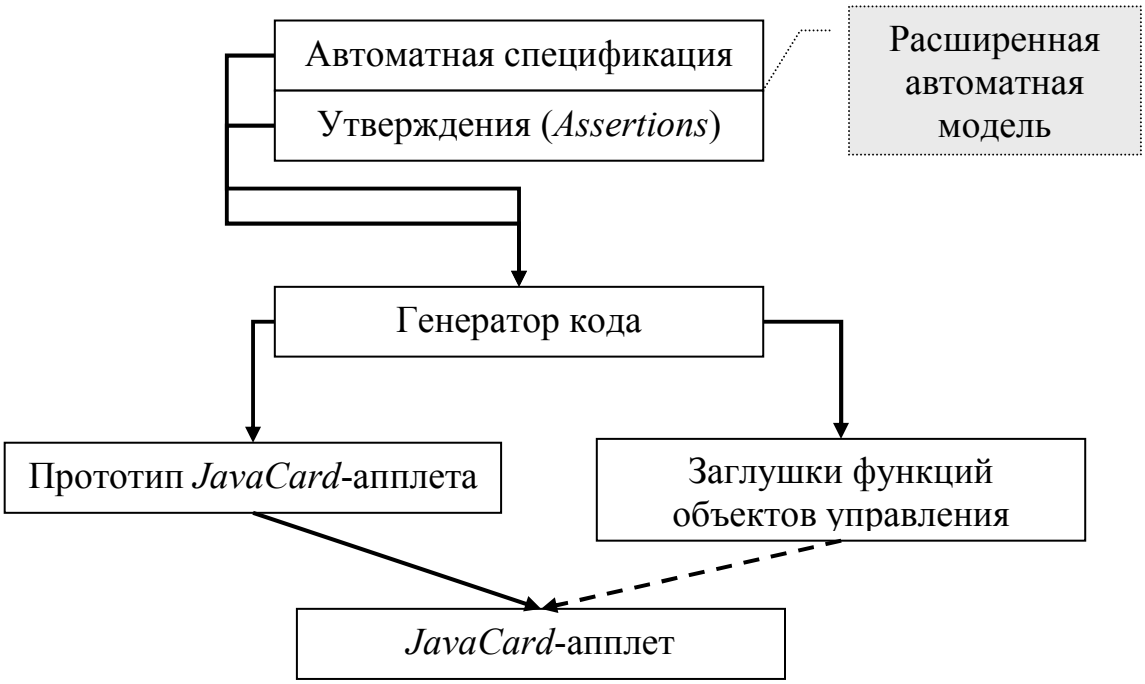

Рис. 2.2. Подход, использующий расширенную автоматную модель

Еще одним важным аргументом в пользу расширения модели является обработка *APDU*-сообщений, которые присутствуют в большинстве *JavaCard*-апплетов и являются проблемными для тестирования. Как было сказано в главе 1, *APDU*-команда состоит из двух важных составляющих – информация о произошедшем событии и дополнительные данные, переданные вместе с ним. Произошедшее событие является важным фактором в рамках автоматной модели. Помимо самого события, на поведение и ход исполнения апплета влияют данные, переданные в качестве параметров. Поэтому, желательно учесть эти данные еще на этапе проектирования. Если полностью исключить их из анализа, то многие функции станут непригодными для проверки.

Принимая во внимание, что *Java Card* поддерживает только целочисленные значения переменных, в рамках данной работы вводятся следующие ограничения – пусть все функции, использующие *APDU*-данные, возвращают це-

лочисленное значение. Тогда, код этих функций, работающих с *APDU*, можно исключить из анализа. При правильном распределении функциональности кода, получится два типа функций:

1. Обрабатывающие *APDU*-данные, получающие необходимую информацию из запроса и возвращающие результат своего исполнения в формате целого числа.
2. Реализующие логику приложения и не использующие *APDU*.

Исключив из анализа реализацию функций первого типа, можно будет проверять корректность функций второго типа, возможность ошибки в которых значительно более вероятна. Проанализируем предлагаемое ограничение. Для этого ответим на вопрос, какая информация может передаваться в *APDU*-запросе. С учетом ограничений *Java Card*, это, скорее всего, число. Строки, символы, многомерные массивы, динамическая загрузка классов не поддерживаются самой платформой. За исключением числа, это может быть только одномерный массив. Таким образом, замена всех функции работы с *APDU* на интервалы возможных возвращаемых значений не на много ограничивает тестируемую модель.

Спектр задач, решаемых приложениями, написанными для платформы *Java Card*, обычно не включает в себя алгебраические вычисления или работу с данными массивов. Поэтому, значения элементов массива, в большинстве случаев, не являются важными при исполнении логики приложения. Например, массив может использоваться, как способ хранения строки, где каждый элемент типа байт хранит код соответствующего символа – так сделано в *SIM Toolkit* [40]. Полученные из памяти значения передаются в качестве *SMS*-сообщения пользователю. В этом случае конкретные значения данных массива никак не влияют на ход исполнения апплета, и, исключение их из модели, никак не испортит результаты анализа. Поэтому функции, которые получали из *APDU* какую-либо информацию, отличную от целого числа, будем рассматривать следующим образом – они будут возвращать единицу, в том случае, если информация была получена, или ноль, если получить информацию не уда-

лось, например, *APDU*-команда не соответствует ожидаемому формату. Такое решение позволяет абстрагироваться от функций обработки *APDU* и сконцентрироваться на проверке логики выполнения.

Для поддержки предложенной функциональности, расширим существующий способ *XML*-описания модели, описанный в главе 1, 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 считываются из *APDU*-команды.

В первом случае атрибуты `in` и `bits` являются ненужными, или атрибут `in` должен содержать значение `false`. Для работы с переменными второго типа используется функциональность транслятора *jMoped* – при тестировании будут проверены все возможные значения в заданном интервале. Интервал значений задается при помощи количества бит, в которых будет храниться значение. Для этого введен атрибут `bits`, а атрибут `in="true"` указывает на то, что переменная второго типа.

К тегу `outputIndent` добавим возможность задавать пред- и постусловия при помощи тегов `precond` и `postcond`, значения которых должны быть заданы по тем же правилам, что и в атрибуте `guard`, описанном выше:

```
<!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 вручную ввести пред- и постусловия вызовов функции объектов управления. При генерации кода заданные условия будут вставлены в заглушки указанных функции в виде проверок – это позволит автоматически проверять реализацию заглушек. В итоге, на вход генератор принимает *XML*-файлы следующего формата:

```
<!ELEMENT stateMachine (initcode*, globalvars*,
asserts*, state+, transition*)>
   <!ATTLIST stateMachine name CDATA #IMPLIED>

<!ELEMENT globalvars (var*)>

<!ELEMENT var EMPTY>
   <!ATTLIST var type CDATA #REQUIRED>
   <!ATTLIST var name CDATA #REQUIRED>

<!ELEMENT asserts (outputActionCond*)>

<!ELEMENT outputActionCond (precond*, postcond*)>
   <!ATTLIST outputActionCond ident CDATA #REQUIRED>

<!ELEMENT name (#PCDATA)>

<!ELEMENT initcode (#PCDATA)>
   <!ATTLIST initcode language CDATA #IMPLIED>
<!ELEMENT state (state*, stateMachineRef*,
outputAction*)>
   <!ATTLIST state name CDATA #REQUIRED>
   <!ATTLIST state type (INITIAL|FINAL|NORMAL)
#REQUIRED>

<!ELEMENT init EMPTY>
   <!ATTLIST init ref IDREF #REQUIRED>
```

```
<!ELEMENT transition (outputAction*)>
   <!ATTLIST transition name CDATA #IMPLIED>
   <!ATTLIST transition sourceRef CDATA #REQUIRED>
   <!ATTLIST transition targetRef CDATA #REQUIRED>
   <!ATTLIST transition event CDATA #IMPLIED>
   <!ATTLIST transition guard CDATA #IMPLIED>

<!ELEMENT outputAction EMPTY>
   <!ATTLIST outputAction ident CDATA #REQUIRED>

<!ELEMENT precond EMPTY>
  <!ATTLIST precond value CDATA #REQUIRED>

<!ELEMENT postcond EMPTY>
  <!ATTLIST postcond value CDATA #REQUIRED>
```

2.1.3. Генерация кода

Генерация кода позволяет избежать большого количества ошибок, связанных с человеческим фактором, неправильным пониманием хода работы и несоответствий со спецификацией.

Апплеты *Java Card* имеют стандартную структуру. Все они реализуют метод *process*, который обрабатывает поступающие пакеты инструкции *APDU*, отсылаемые на карту терминалом. В *APDU* имеется заголовок, который содержит инструкцию для выполнения. Обычно метод *process* анализирует заголовок пакета для получения нужной инструкции, а затем вызывает конкретный метод для ее выполнения. В зависимости от текущего состояния апплета, инструкция может быть валидной или нет. При получении недопустимой инструкции, апплет кидает исключение.

Генерация кода приложения придерживается следующего принципа – любые переходы между состояниями реализуются в рамках главной функции. При этом используется инкапсуляция – при появлении любого события, происходит вызов соответствующей функции, и все необходимые действия происходят в рамках ее выполнения, а не в функции обработки событий. Для определения заданного варианта изменения состояния, учитывая при этом текущее со-

стояние, удобно использовать блок операторов `switch/case`. Каждый оператор `case` соответствует одному из вариантов изменения состояния и содержит вызов функции, выполняющей заданное действие.

Написанное таким способом приложение легко масштабируется при изменении функциональности. Основные изменения могут быть включены в главную функцию централизованным образом. Подробнее такой подход и его преимущества описаны в работе [41]. Полученный код является правильным *JavaCard*-апплетом, компилируемым, но с ограниченной функциональностью. Поэтому, во многих случаях, необходимо будет добавить код, выполняющий специфические задачи, вручную.

Как сказано в главе 1, в случае платформы *Java Card* инструментальное средство *UniMod* можно применить на этапе проектирования автоматной модели. Процесс создания кода отличается от создания *JavaSE*-кода, который поддерживает *UniMod*. Таким образом, стоит задача генерации *JavaCard*-кода из модели *UniMod*, представленной в виде *XML*-описания.

Процесс генерации кода из расширенного *XML*-описания конечного автомата подразделен на следующие этапы:

1. Построение внутреннего представления *XML*-файла.
2. Выделение множества состояний автомата, переходов и глобальных переменных.
3. Разбор входных и выходных воздействий, заданных вручную утверждений.
4. Генерация главной функции, осуществляющей управление и функций обработки *APDU*.
5. Генерация заглушек методов объектов управления с необходимыми утверждениями.

В табл. 2.1 описаны основные классы, реализующие заданную функциональность, и особенности решения поставленной задачи.

Таблица 2.1. Классы генератора *JavaCard*-кода

Классы и их назначение	Особенности реализации

`FSM`
`FSMXMLParser`
Построение внутреннего представления *XML*-файла. Производится парсинг заданного файла в *DOM*-дерево.
	Для внутреннего представления *XML*-данных выбрана «объектная модель документов» (*Document Object Model*). В статье [42] определена модель *DOM*. Модель не накладывает ограничений на структуру документа – любой документ известной структуры при помощи *DOM* может быть представлен в виде дерева узлов, каждый их которых содержит элемент, атрибут, текстовый, графический или любой другой объект. Узлы связаны между собой отношениями родитель-потомок.

`State`
`Transition`
`VarList`
`OutputAction`
`Condition`
Классы для удобного внутреннего представления автоматной модели.
	Внутреннее представление автомата состоит из следующих элементов:
	1. Состояния (*State*).
	2. Переходы (*Transition*).
	3. События (*UnimodEvent*).
	4. Выходные воздействия (*OutputAction*).
	5. Условия на переходах (*Condition*).

Продолжение табл. 2.1

`FSMXMLParser`
`UnimodParser`
Обработка DOM-представления входного файла.
	Выделяются элементы модели и преобразовываются в удобный для хранения вид.

`Applet`
`Generator`
Генерация главной функции управления и обработки поступающих событий.
	Для генерации кода из автоматной спецификации в полученный код добавляется специальная переменная `fsm_state` для кодирования текущего состояния конечного автомата. Переходы в структуре конечного автомата соответствуют возможным инструкциям, валидным в данном состоянии. Метод `process` генерируется в виде сложной `switch/case` структуры, где внешний уровень отвечает за выбор текущего состояния автомата, а внутренний предназначен для выбора нужной инструкции. Сгенерированный код, при получении инструкции, проверяет выполненность всех условий (*guard*) и выполняет все выходные воздействия на переходах.

`ControlledObject`
`Generator`
Генерация заглушек методов объектов управления с необходимыми утверждениями.
	Данный класс предназначен для генерации прототипов классов объектов управления. В том случае, если *XML*-описание было расширено требованиями, то соответствующие утверждения будут добавлены в код функции в виде команды `assert`, поддержка которой появилась в *Java*-компиляторе, начиная с версии 1.5.

Опишем результаты работы генератора, приведем общий вид получаемого кода и подробнее рассмотрим некоторые его элементы. Генератор создает каркас файла апплета, соответствующий требованиям *Java Card*-платформы и объявляющий все необходимые для последующего использования переменные:

```
import javacard.framework.*;
/**
 * Generated java card applet, implementing A1 state
machine
 */
public class A1Applet extends Applet implements
ISO7816 {

/* FSM states */
   short fsm_state;
   static final short S1 = (short)0;
   static final short S2 = (short)1;
    ...
/* FSM events */
   static final byte e1 = (byte)0;
   static final byte e2 = (byte)1;
   static final byte e3 = (byte)2;
    ...
/* User defined variables */
    short var1;
   ...
    static final short const1 = 123;
   ...
}
```

Наиболее важными являются функции обработки поступающих сообщений и управления апплетом. Обычно эти задачи решает метод process,

обрабатывающий *APDU*-команду и передающий управление заданной функции. В случае автоматного подхода к проектированию, удобнее выделить логику, связанную с вызовом методов объектов управления, в отдельную функцию. В связи с этим, генерируются функции `process(APDU apdu)` и `processAutomataEvent(short event)`:

```
    public void process(APDU apdu) throws ISOException {
        byte[] buffer = apdu.getBuffer();
        byte cla = buffer[OFFSET_CLA];
        byte ins = buffer[OFFSET_INS];
          if (selectingApplet() || cla == CLA_ISO7816
          &&      ins == INS_SELECT) {
            return;
        processAutomataEvent(parseAPDUCommand(apdu));
    }
       private void processAutomataEvent(short event)
throws ISOException {
        switch (fsm_state) {
        case S1:
                switch (event) {
                    case e4:
                        if (/*transition condition*/) {
                            /*output action*/
                           fsm_state = /*new state*/;
                        }
                    break;
                case e2:
                        ...
                case S2:
                    ....
            }
```

Подробнее рассмотрим генерируемую функцию `parseAPDUCommand`, целью которой является извлечь из входящей *APDU*-команды все необходимые данные – понять какое сообщение произошло в термнах автоматной модели, сохранить данные, переданные в *APDU*. Работа с данными не описывается в рамках автоматной модели, поэтому возможно сгенерировать только шаблон этой функции, в который необходимо будет подставить значения байт `CLA` и `INS`, смысл которых был описан в главе 1, и вручную реализовать получение данных из *APDU*. С учетом этого, получается следующая структура:

```
private short parseAPDUCommand(APDU apdu) {
        byte[] buffer = apdu.getBuffer();
        byte cla = buffer[OFFSET_CLA];
        byte ins = buffer[OFFSET_INS];
        switch (cla) {
            case /*event provider 1*/:
                switch (ins) {
                    case /* event 1 */:
                        res = e1;
                        setVarsForE1(apdu);
                        break;
                    ...
                }
            ...
        }
    }
```

Последним важным элементом сгенерированного файла являются заглушки функций объектов управления, которые выглядят следующим образом:

```
private void o#_z#() {
        /* preconditions */
        assert (...);
```

```
        /* stub implementation */

        /* precondition */
        assert (...);
}
```

Символ '#' обозначает любое число. Помимо описания, в тело функции добаляются утверждения, указанные в модели и накладывающие требования на результаты выполнения. В последствии, когда тело функции будет реализовано, это позволит проверить, что добавленный код соблюдает эти условия. Аналогично, если функция требует некоторых свойств от переменных к началу своей работы, и это было указано при построении модели, соответствующая проверка будет добавлена в начало заглушки функции.

Помимо прототипа кода, важным результатом генерации является набор свойств автоматной модели, на предмет соблюдения которых будет проверяться конечная реализация. Данную задачу выполняет класс *PropertiesGenerator*. Результатом его работы является текстовый фаил, содержащий информацию о спроектированном конечном автомате: список состояний автомата, список всех возможных переходов между состояниями, все утверждения, добавленные в модель. Впоследствии этот файл будет подан на вход скрипту, автоматизирующему проверку и тестирование апплета.

2.1.5. Проверка реализации заглушек

Сформулируем свойства реализации, которые хотелось бы проверить и которые можно определить, исходя из существующей автоматной модели приложения:

1. Выполнимость переходов, с учетом добавленного кода. Схема переходов состояний конечного автомата, спроектированная при помощи *UniMod*, описывает ожидаемую от приложения последовательность действий. На этапе проектирования, разработчик никак не учиты-

вает реализацию функций объектов управления, которые будут вызваны при переходах между состояниями. Например, возможна такая ситуация, когда функция проверки всегда будет возвращать только отрицательный результат, или выходное действие будет содержать в себе ошибочный код – например, зацикливающийся или порождающий исключение. В этом случае, на практике, такой переход не будет выполнимым. Обнаружить такие ошибки можно следующим образом – для каждого возможного состояния переберем все возможные переходы и проверим их выполнимость для указанного интервала входных значений.

2. Достижимость состояний из начального состояния. Нельзя гарантировать выполнимость этой проверки, так как в каждом новом состоянии возможное число вариантов развития событий увеличивается в n раз, где n – число возможных переходов из этого состояния. Это связано с тем, что на этапе проверки нельзя предугадать какое событие произойдет, и, следовательно, необходимо проверять все возможные варианты. Число вариантов для проверки растет экспоненциально в зависимости от числа состояний. На практике необходимо учитывать, что *JavaCard*-апплеты обычно не обладают такой сложной структурой, как приложения, спроектированные для полноценных компьютеров, поэтому, во многих случаях, такая проверка окажется выполнимой и более того сможет выявить потенциальные ошибки реализации.
3. Проверка выполнения утверждений в функциях объектов управления. Разработчик может расширить *XML*-описание, сгенерированное при помощи инструментального средства *UniMod*, 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. В итоге, добавленная разработчи-

ком реализация объектов управления, будет проверена на их соблюдение.

Имея результаты проверки перечисленных свойств, разработчик сможет сделать вывод, соответствует ли полученная в итоге реализация спроектированной на первом шаге автоматной модели. Для того чтобы сделать проверку этих свойств выполнимой задачей и получать контрпример верификатора в удобном формате необходимо, чтобы та входная модель удовлетворяла следующим требованиям:

1. Каждому переходу должна соответствовать своя метка, достижимость которой будет означать выполнимость перехода.
2. Каждому состоянию должна соответствовать метка, достижимость которой будет означать достижимость состояния.
3. Все функции работы с *APDU* должны быть исключены из анализа, а их возвращаемое значение заменено переменной.

Соответствующие модификации были сделаны в трансляторе *jMoped*, описанном в главе 1. Проиллюстрируем эти изменения на примерах. Стандартный *Java*-код смены состояния выглядит так:

```
case S5:
      fsm_state = S2;
      break;
```

Существующая версия *jMoped* сопоставит этим строкам следующий код на языке *Remolpa*:

```
I_I28:
s[sptr] = 0, sptr = sptr + 1;
I_I29:
lv1 = s[sptr - 1], sptr = sptr - 1;
I_I30:
if
:: (true) -> goto I_I39;
fi
```

Такая модель неудобна для проверки, так как нет метки, достижимость которой явно означала бы осуществленный переход или попадание в состояние. Удобный для проверки *Java*-код выглядел бы так:

```
case S2_STAND_BY:

transition_from_S2_STAND_BY_to_S5_CHECK_PIN();
            set_state_to_S5_CHECK_PIN();
            break;
```

Соответственно, модель выглядит следующим образом:

```
I_I33: s[sptr] = lv0, sptr = sptr + 1;
I_I34: if
    :: (s[sptr - 1] == 0) -> I_I34_NPE: goto npe;
    :: (s[sptr - 1] != 0 && (true)) ->
transition_from_S2_to_S5__V0(s[sptr - 1]),
sptr = sptr - 1;
 fi;
 I_I37: s[sptr] = lv0, sptr = sptr + 1;
 I_I38: if
    :: (s[sptr - 1] == 0) -> I_I38_NPE: goto npe;
    :: (s[sptr - 1] != 0 && (true)) ->
set_state_S2__V0(s[sptr - 1]), sptr = sptr - 1;
 fi;
```

В данном случае явно выделена метка `transition_from_S2_to_S5__V0`, достижимость которой означает выполнимость перехода. Метка `set_state_S2__V0` соответствует тому, что приложение перешло в состояние *S2*.

Второй тип изменений, необходимый для построения корректной входной модели верификатора – исключение вызовов функций работы с *APDU*. Рассмотрим функцию `apdu_var_x()`, которая считывает значения из *APDU*-сообщения. На этапе проектирования, для анализа возможных значений, разработчик задал переменную `x_in_var`. Стандартный *Java*-код, в такой ситуации выглядит так:

```
x = apdu_var_x();
```

Java-коду соответствует такой код *Remolp*а-модели:

```
     I_I2: if
        :: (s[sptr - 1] == 0) -> I_I2_NPE: goto npe;
        :: (s[sptr - 1] != 0 && (true)) ->
apdu_var_x__I0(s[sptr - 1]), sptr = sptr - 1;
     fi;
     s[sptr] = ret, sptr = sptr + 1;
```

В предлагаемом подходе, эту функцию необходимо исключить из анализа. Поэтому код выглядит следующим образом:

```
     x = x_in_var;
```

Значит, модифицированный транслятор должен генерировать такой *Remolpa*-код:

```
     I_I2: if
        :: (s[sptr - 1] == 0) -> I_I2_NPE: goto npe;
        :: (s[sptr - 1] != 0 && (true)) -> s[sptr - 1] =
    heap[s[sptr - 1] + 1];
     fi;
     I_I5: if
        :: (s[sptr - 2] == 0) -> I_I5_NPE: goto npe;
        :: (s[sptr - 2] != 0 && (true)) ->
heap[s[sptr - 2] +   2] = s[sptr - 1], sptr = sptr - 2;
     fi;
```

Полученная таким образом модель с достаточной точностью описывает логику приложения и позволяет выполнить проверку важных свойств.

В табл. 2.2 приведен список написанных в рамках данной работы скриптов, позволяющих автоматизировать проверку реализации заглушек.

Таблица 2.2. Скрипты, автоматизирующие тестирование

`Check transitions`	Проверка выполнимости всех переходов для заданных интервалов входных параметров. Для созданного генератором набор меток проверяется их достижимость. В этом случае использована функциональность выбранного верификатора, позволяющая перебирать значения переменных: входное состояние на входе в функцию *process*, появившееся событие и значения задействованных параметров. Скрипт выдаст результат анализа в

Продолжение табл. 2.2

	следующем формате:
	`Reachable transitions:`
	`...`
	`Not reachable transitions:`
	`...`

`Check states`	Аналогично, проверка достижимости состояний из начального состояния:
	`Reachable states:`
	`...`
	`Not reachable states:`
	`...`

`Check assertions`	Поиск контрпримера, для которого будут нарушены добавленные разработчиком утверждения.
	`Error is reachable at <label>`
	`Error trace:`
	`...`

2.2. Анализ предложенного подхода

Описанный в данной работе подход учитывает все этапы разработки *JavaCard*-апплета. Большое внимание уделяется этапам проектирования и написания требований к коду – значимые шаги процесса разработки, выполнение которых позволит разработчику не только сформулировать и описать работу апплета, но и повысить надежность кода в целом. Для построения автоматной модели предлагается использовать инструментальное средство *UniMod*. Для написания требований к коду была предложена расширенная утверждениями автоматная модель, проверка которой позволит оценить надежность кода, написанного вручную.

Код, выполняющий обработку поступающих сообщений и содержащий вызовы соответствующих функций, генерируется автоматически. Это не только сильно упрощает процесс разработки апплета и помогает избежать появления случайных ошибок, но также гарантирует, что код удовлетворяет всем свойствам, описанным на схеме переходов – соответствует построенной модели поведения. Уверенность в том, что апплет меняет свои состояния в правильной последовательности и вызов функций объектов управления происходит только в допустимых состояниях, – важное свойство безопасности приложения.

Тестирование апплета при помощи верификатора *Moped* позволяет оценить, соответствует ли конечная реализация свойствам и требованиям, описанным на этапе проектирования. Результат проверки верификатором является неполным – найденные ошибки могут на самом деле отсутствовать в коде, а отсутствие выявленных ошибок не позволяет уверенно гарантировать их отсутствие вообще. Несмотря на это, информация, полученная благодаря этой проверке, позволяет существенно сократить разрыв между спецификацией и реализацией – проанализировать, насколько корректен дописанный вручную код и решает ли он поставленную задачу.

Для использования сформулированного подхода, в рамках данной работы были реализованы следующие инструменты: генератор *JavaCard*-кода из расширенного *XML*-описания автоматной модели, генератор свойств модели, которые можно проверять впоследствии, и набор скриптов, автоматизирующих тестирование. Также был модифицирован транслятор байт-кода для поддержки *JavaCard*-элементов и генерации входной модели верификатора, учитывающей автоматную структуру приложения.

Таким образом, предложенный подход и созданный для его реализации набор инструментов позволяют не только упростить процесс разработки *JavaCard*-апплетов, но также существенно повысить надежность кода. Несмотря на неполноту анализа и на тот факт, что корректность полученного приложения не может быть гарантирована, применение подхода при разработке по-

зволит избежать множества ошибок, что является критичным фактором в случае платформы *Java Card*.

2.3. Сравнение с существующими инструментами

Предложенный подход учитывает реальные условия разработки *JavaCard*-апплетов, когда основная часть кода генерируется автоматически, а какая-то часть кода пишется вручную. Из существующих на данный момент работ можно выделить два предлагаемых решения: анализ уже написанного кода и генерация *JavaCard*-кода из специально созданного языка-спецификации. Например, *AutoSmart* генератор, описанный в главе 1. Написание подробной спецификации, из которой впоследствии будет получен код – процесс достаточно трудоемкий, более того требует изучения синтаксиса предлагаемого языка. Допустить ошибку, описывая функциональность программы при помощи такой спецификации так же легко, как и при написании кода. Это приведет к тому, что сгенерированный код будет содержать эту ошибку. В случае предложенного автоматного подхода к проектированию, вероятность появления ошибки меньше, так как конечный автомат является удобным и понятным способом визуализации поведения апплета. Среди преимуществ предложенного подхода можно также отметить соотношение числа строк кода, сгенерированного автоматически, и кода, написанного руками. Как признают авторы *Kestrel Project* [30], для их инструмента это соотношение примерно три к одному. При применении рассматриваемого подхода, это соотношение будет значительно больше для апплетов среднего размера.

В работах, посвященных анализу написанного кода, тоже можно выделить ряд недостатков, решенных в данной работе. Правильное проектирование и написание требований к приложению – важнейшие элементы разработки, и в предложенном подходе этим этапам уделяется ключевое внимание. Правильно построив модель и применив автоматическую генерацию кода, можно избежать множества ошибок, как простых, так и незаметных на первый взгляд, связанных с логикой приложения. Таким образом, анализ уже написанного кода явля-

ется интересным вопросом для исследования, но это не лучшее решение для повышения надежности апплетов. Лучше повышать надежность самого процесса разработки, а не только искать ошибки в готовом коде. Необходимо понимать, что существующие методы проверок можно использовать в рамках предложенного подхода – итоговый *JavaCard*-код проверять описанными в работах методами. Это позволит учесть преимущества существующих разработок и гарантировать соблюдение ряда свойств.

Интересным подходом является генерация *JML*-аннотаций из конечного автомата. Этой функциональностью обладает инструмент *AutoJML*, описанный в главе 1, и подход, предложенный в работе А. Клебанова [5]. *AutoJML* позволяет из автоматного описания генерировать *JML*-аннотации, используя которые, можно проанализировать *Java*-код методами статической проверки, например, при помощи *ESC/JAVA2*. Такие аннотации позволяют проверить следующие свойства: инварианты, определяющие корректную последовательность переходов между состояниями, вызов функции объектов управления из корректных состояний, выполнение указанных на переходах условий. В данной работе предлагается использовать автоматный подход. Поэтому *JML*-аннотации, полученные путем анализа схемы переходов конечного автомата, не представляют интереса, так как все эти свойства гарантируются при помощи автоматической генерации главной функции управления и обработки событий. Правильно сгенерированный код будет осуществлять управление в точности с требованиями автоматной модели. Поэтому *JML*-аннотации такого рода можно использовать для проверки автоматной модели. Для тестирования кода, написанного вручную, они неприменимы.

Можно добавить *JML*-аннотации к методам объектов управления. В этом случае, постановка требований будет происходить этапе написания кода приложения. Предложенный в данный работе метод добавления утверждений к функциям отличается тем, что их формулировка выполняется на этапе проектирования. Такой подход является более разумным, так как ключевые моменты реализации следует прояснять именно на этапе проектирования, а не в процессе

написания кода. Информация о соответствии кода с *JML*-спецификациями, безусловно, полезна, но никак не позволяет проанализировать, насколько реализация соответствует своему автоматному описанию. Таким образом, это никак не решает вопрос разрыва спецификации и кода, поставленный в данной работе. Эту технологию можно использовать в качестве дополнительного этапа проверки в рамках предложенного подхода. Использование только этой технологии не позволяет учесть преимущества автоматного подхода и делает этап построения модели менее значимым. На рис. 2.3 изображена возможность совместного использования предлагаемого подхода и инструментов работы А. Клебанова [5].

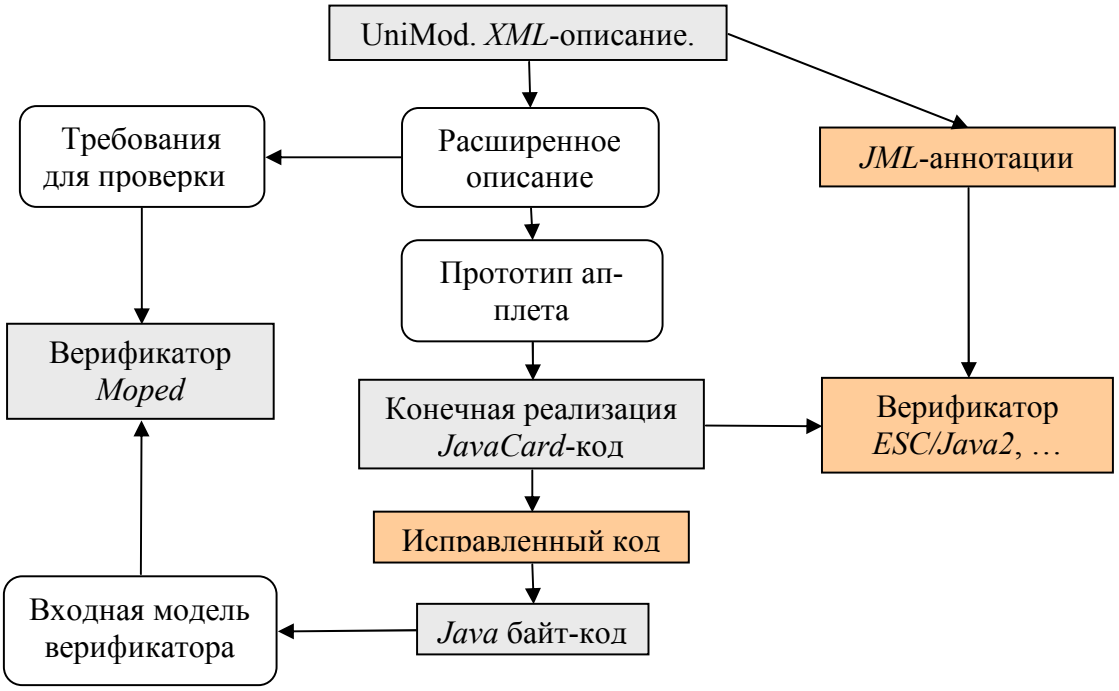

Рис. 2.3. Возможность интеграции с существующими методами

Выводы по главе 2

Предложенный подход позволяет повысить надежность процесса разработки и обладает рядом преимуществ по сравнению с существующими инструментами:

1. Несмотря на то, что *JavaCard*-апплет функционально эквивалентен конечному автомату как событийный объект, существующие исследования не учитывают этот факт ни при проектировании программы, ни при ее реализации. Автоматный подход – хорошее решение, учитывающее эту особенность.
2. Удобное сочетание автоматически сгенерированного кода и написанного вручную. Для генерации всего кода потребуется писать сложную спецификацию. В некоторых случаях это может быть неприменимо. Генерация основных элементов апплета по автоматной спецификации позволяет путем затраты малых усилий на проектирование предотвратить появление разного рода ошибок и тем самым повысить надежность кода.
3. Предложен не только способ проверки тех или иных свойств написанного приложения, но подход, описывающий весь процесс разработки – начиная с проектирования автоматной модели и формулирования требований к переменным, заканчивая проверкой соответствия автоматного описания и конечной реализации.
4. Не рассматривается вопрос о гарантиях корректности приложений. Такие подходы в любом случае требуют больших усилий по созданию модели для верификации и написания четких требований. В тех случаях, когда такие затраты времени неоправданны, существующие верификаторы неприменимы. Цель предложенного подхода – повысить надежность, найти возможные ошибки в реализации и оценить, насколько реализация удовлетворяет автоматному описанию.

ГЛАВА 3. ПРИМЕР ИСПОЛЬЗОВАНИЯ

Среди открытых результатов в области разработки *JavaCard*-приложений отсутствуют на данный момент описания полного цикла разработки апплета, начиная с построения модели или написания спецификации и заканчивая его тестированием или верификацией. Доступные работы посвящены либо написанию, и, в ряде случаев, генерации *JavaCard*-программ, либо исследованию уже написанного кода с целью выявить в нем ошибки – например, при помощи ручного или автоматического добавления *JML*-аннотаций.

В данной главе будут рассмотрены два примера разработки небольших *JavaCard*-апплетов с применением предложенного в данной работе подхода. Для полученных приложений будет произведен анализ ошибок и сделан вывод о полезности такого анализа.

3.1. Разработка электронного кошелька

Апробируем предложенный в работе подход для разработки небольшого *JavaCard*-приложения. В качестве задачи поставим написание апплета, 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 раздел примеров, начиная с версии 2.1.1.

3.1.1. Описание задачи

Как кошелек, апплет может хранить некоторое количество электронных денег, позволяет добавлять деньги в кошелек и забирать их оттуда. Естественно, работа с кошельком должна быть доступна только его владельцу, поэтому *Wallet*-апплет использует стандартный для *Java Card* механизм безопасности, требующий авторизации с использованием Персонального Идентификационного Номера (*Personal Identification Number – PIN*). Только успешно авторизованный пользователь получает возможность производить работу с хранящимися деньгами.

Разработаем аналогичный апплет, но, используя предложенный подход, что позволит в итоге оценить его преимущества и недостатки. Для начала необходимо более четко определить требуемую функциональность и ограничения.

Перечислим запросы, которые должен корректно обрабатывать разрабатываемый апплет:

- положить деньги на счет;
- снять деньги со счета;
- вернуть текущий баланс;
- проверить *PIN*-код.

Ограничения, заданные в спецификации:

- численные, к состоянию счета и размеру транзакции, вызванные как ограничениями платформы, так и внешними ограничениями – максимальный размер транзакции часто ограничен в реальных приложениях;
- на проверку *PIN*-кода. Обычно число попыток авторизации ограничено.

Таким образом, можно выделить следующие вопросы, которым стоит уделить внимание в контексте поставленной задачи.

1. Получение данных от пользователя.

   Следует уточнить, что эти данные будут получены приложением в виде *APDU*-команды, а не привычным, в случае *Java ME* или *Java SE* обращением к интерфейсу приложения, который мог бы осуществлять какой-либо контроль валидности введенных данных. Поэтому, необходимо учитывать, что при получении запрашиваемого размера транзакции из *APDU*-команды, может быть получено любое число, которое умещается в доступных байтах памяти.

2. Проверка валидности полученного запроса.

   Во-первых, перед выполнением любой операции со счетом, необходимо проверять, что пользователь авторизован. Во-вторых, ограничения, наложенные на состояние счета, должны выполняться как до начала операции, так и после. В-третьих, любой запрос, отличный от перечисленных выше,

должен считаться некорректным и не должен приводить к каким-либо действиям.

3. Вызов методов только из допустимых состояний.

Для каждой команды существует состояние, из которого она корректно может быть вызвана. В описываемом примере, любая операция со счетом может начинаться только в тот момент, когда предыдущая закончена – если пришел запрос на снятие денег до того, как был завершен предыдущий запрос, апплет не должен обрабатывать его.

3.1.2. Расширенная автоматная модель апплета

Рассмотрим спроектированную при помощи *UniMod* схему связей на рис. 3.1.

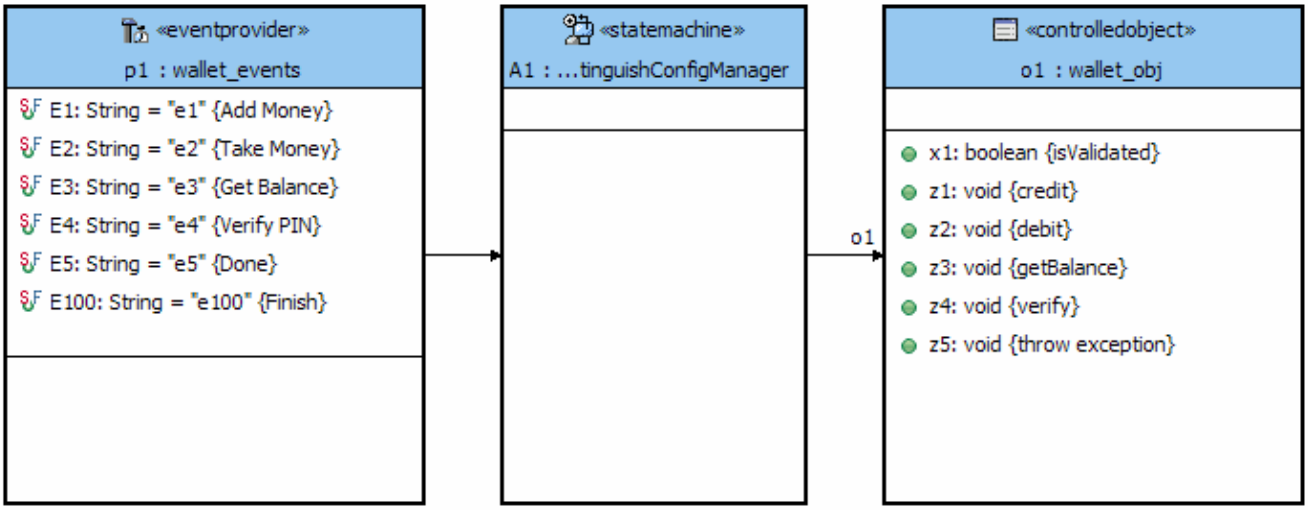

Рис. 3.1. Схема связей

Использованная нотация и значения объектов представленных на схеме связей подробно описаны в работе [3]. Основным поставщиком событий в рассматриваемой задаче является сервер, отсылающий запросы, содержащие выбор команды и дополнительную информацию при необходимости. Помимо этого, описано одно событие, поставщиком которого является апплет – событие, означающее, что полученная операция со счетом произведена. В контексте данного примера не будем подробно рассматривать возможные ответы апплета, так как целью является спроектировать логику апплета, а не сервера. В соответ-

ствии с заданной функциональностью электронного кошелька существуют следующие варианты возможного события, описанные в поставщике событий *WalletEvents*:

1. Событие *e1* (*Add Money*) – требуется пополнить счет некоторой суммой.
2. Событие *e2* (*Take Money*) – необходимо снять некоторую сумму со счета.
3. Событие *e3* (*Get Balance*) – запрос текущего остатка, необходимо передать серверу эту информацию.
4. Событие *e4* (*Verify PIN*) – следует проверить корректность введенного *PIN*-кода.
5. Событие *e5* (*Done*) – апплет сообщает о завершении выполнения команды. Если события, поступающие от сервера, важны и влияют на поведение приложения, то ответы апплета, направленные серверу, выходят за рамки данного примера. Поэтому ограничимся одним событием, соответствующим тому, что апплет выполнил полученную команду.
6. Событие *e100* (*Finish*) – жизненный цикл *JavaCard*-апплета отличается от привычного жизненного цикла приложения, так как отсутствует четкое понятие начала выполнения, и четкое понятие завершения работы. Как было описано в главе 1, активизация апплета происходит при помощи вызова метода `select()`, а деактивизация – при помощи вызова метода `deselect()`. Включать эти методы в проектирование логики не стоит, так как они не предназначены для выполнения какой-либо работы связанной с логикой приложения. По существующим правилам *UniMod*, автомат должен включать конечное и начальное состояние. Поэтому, для корректности схемы переходов введем состояние *e100*, которое будет аналогично заглушке (*stub*) – не будет влиять на логику. Если же разработчик хочет добавить проверки на выполнение метода `select()`, то, это возможно сделать, добавив необходимые предусло-

вия в начальное состояние, в которое будет переведен апплет при получении команды `select()`, в соответствии правилами генерации, использованными в данной работе.

Для каждого запроса должна быть функция, выполняющая требуемые действия. В связи с этим объект управления `o1` содержит четыре функции, описывающие выходные воздействия. Учтем, что для выполнения операции со счетом, необходимо удостоверится в том, что пользователь апплета был авторизован. В связи с этим, добавим функцию `o1.x1` проверки этого свойства к объекту управления, которая позволит не учитывать это требование при построении схемы связей. Функция `o1.z5` будет создавать исключение, сигнализирующее о том, что авторизация не была выполнена.

На данном этапе можно построить схему переходов автомата *A1*, изображенную на рис. 3.2, описывающую логику приложения.

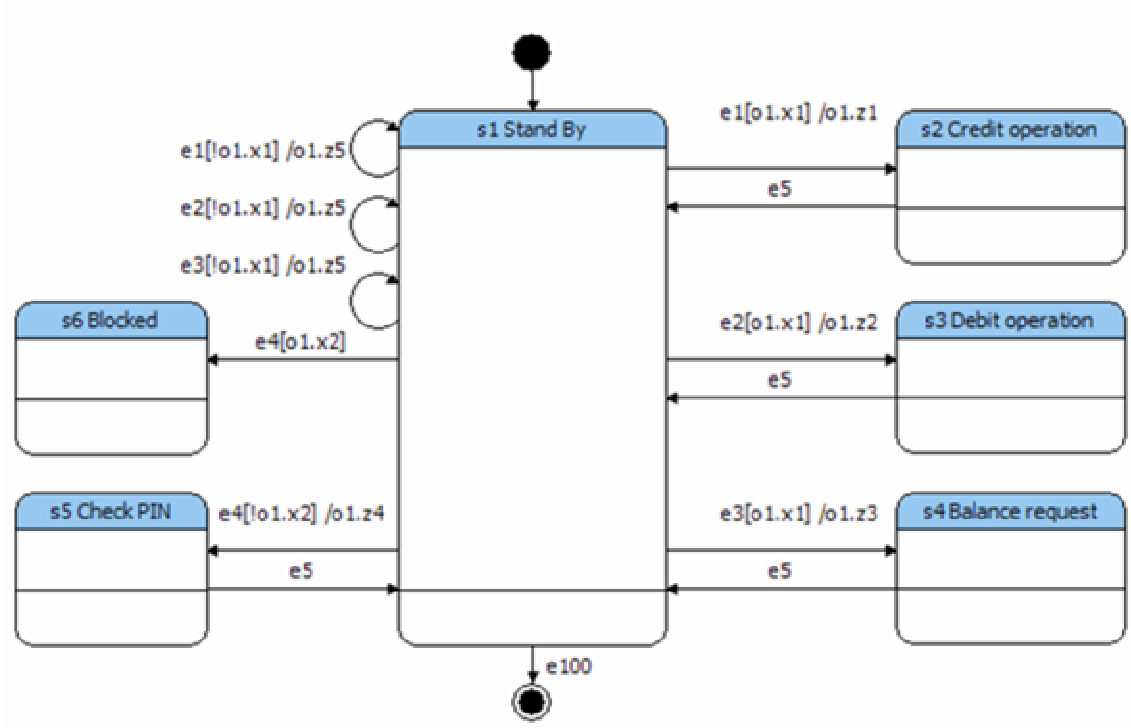

Рис. 3.2. Граф переходов автомата *A1*

Рассмотрим возможные состояния автомата и переходы между ними:

1. *S1 Stand By* (Ожидание команды) – апплет ожидает команду для выполнения какого-либо действия. Это состояние можно рассматривать как начальное для апплета. При получении соответствующей команды апплет либо пытается авторизовать клиента, либо проверяет наличие авторизации посредством вызова `o1.x1` и выполняет переход к заданному действию.
2. *S2 Credit Operation* (Положить деньги) – при условии авторизации, кошелек пополняется суммой денег, переданной в запросе. Завершение операции сигнализируется событием *e5*, которое возвращает апплет в состояние готовности к работе. Аналогичный способ возврата использован и в остальных состояниях, описанных далее.
3. *S3 Debit Operation* (Снять деньги) – при условии авторизации, из кошелька изымается сумма денег, переданная в запросе.
4. *S4 Balance Request* (Запрос остатка) – при условии авторизации, апплет отсылает текущее состояние счета серверу.
5. *S5 Check PIN* (Авторизация) – апплет проверяет валидность полученного в запросе *PIN*-кода и, возможно, выставляет флаг, что авторизация прошла успешно.

Описанный конечный автомат имеет следующее *XML*-представление, сгенерированное инструментальным средством *UniMod*:

```
<stateMachine name="A1">
    <state name="Top" type="NORMAL">
        <state name="s2 Credit operation"
        type="NORMAL"/>
      <state name="s1" type="INITIAL"/>
      <state name="s1 Stand By" type="NORMAL"/>
        <state name="s3 Debit operation"
        type="NORMAL"/>
      <state name="s4 Balance request"
      type="NORMAL"/>
      <state name="s5 Check PIN" type="NORMAL"/>
      <state name="s2" type="FINAL"/>
    </state>
```

```
<transition event="e5" sourceRef="s2 Credit
operation" targetRef="s1 Stand By"/>
 <transition sourceRef="s1" targetRef="s1 Stand
 By"/>
 <transition event="e1" guard="o1.x1"
 sourceRef="s1 Stand By" targetRef="s2 Credit
 operation">
   <outputAction ident="o1.z1"/>
 </transition>
<transition event="*" guard="!o1.x1"
sourceRef="s1 Stand By" targetRef="s1 Stand By">
   <outputAction ident="o1.z5"/>
 </transition>
     <transition event="e2" guard="o1.x1"
     sourceRef="s1 Stand By" targetRef="s3 Debit
     operation">
   <outputAction ident="o1.z2"/>
 </transition>
     <transition event="e3" guard="o1.x1"
     sourceRef="s1 Stand By" targetRef="s4
     Balance request">
   <outputAction ident="o1.z3"/>
 </transition>
     <transition event="e4" guard="o1.x1"
     sourceRef="s1 Stand By" targetRef="s5 Check
     PIN">
   <outputAction ident="o1.z4"/>
 </transition>
<transition event="e100" sourceRef="s1 Stand By"
targetRef="s2"/>
 <transition event="e5" sourceRef="s3 Debit
operation" targetRef="s1 Stand By"/>
 <transition event="e5" sourceRef="s4 Balance
request" targetRef="s1 Stand By"/>
 <transition event="e5" sourceRef="s5 Check PIN"
targetRef="s1 Stand By"/>
</stateMachine>
```

Построение автоматной модели требуемого апплета позволило решить целый ряд поставленных в разд. 3.1.1 задач:

1. Вызов функции объектов управления производится только из тех состояний, где 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 на состояние счета: размер транзакции и количество попыток авторизации. Следует отметить, что задача проверки количества попыток при авторизации, может быть решена при помощи автоматного подхода – путем проектирования небольшого автомата и создания нового поставщика событий.

Рассмотрим эту возможность подробнее. Для добавления логики, связанной с проверкой *PIN*-кода понадобится ввести события, соответствующие правильному *PIN*-коду, неправильному *PIN*-коду, набор функций управления, подсчитывающих количество попыток и некоторые другие. Если включать в имеющийся автомат состояния и события, связанные с этой проверкой, то пропадет одно из важнейших преимуществ построенной модели – понятность и логичность в рамках *Java Card*-платформы. На данный момент, все значимые события являются вариантами *APDU*-сообщений. И переходы между состояниями символизируют реакцию апплета на полученное событие. В смешанном автомате эти свойства автоматной модели пропадут. Также, необходимо помнить, что на уровне реализации, для соблюдения этого свойства требуется лишь завести переменную и добавить одно `if` условие в код функции авторизации. Количество усилий потраченных на введение этой проверки в автоматную модель и ухудшение понятности модели в итоге делают это решение не выгодным.

Разработанный в данной работе подход предлагает альтернативное 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. В реальном апплете размер транзакции передается в *APDU*-команде, и поэтому, на этапах проектирования, написания кода и даже тестирования, это число не известно. На данном этапе уже можно сказать, в каком интервале оно будет находиться. Например, если оно считается из одного байта *APDU*, можно утверждать, что оно лежит в пределах от нуля до 255. Так что заведем соответствующую этому переменную, значение которой на практике будет выставляться функциями обработки *APDU* – функциями, чья логика не влияет на логику апплета, а интересно только возвращаемое значение. На этапе проверки кода будем обращаться к этой переменной, указав интервал ее возможных значений;
- `byte pin_counter` – допустимое число попыток авторизации.

В разрабатываемом примере электронный кошелек должен соблюдать некоторые ограничения на параметры. 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, сверху константой `MAX_TRANSACTION_AMOUNT`. Также накладываются ограничения на авторизацию: `PIN_TRY_LIMIT` – максимальное число попыток авторизации.

Добавим все 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>
```

Теперь можно дописать необходимые условия на значения этих переменных к вызовам функции объектов управления. Переменные `balance` и

`transition` задействованы в функциях `o1.z1` (*credit*) и `o1.z2` (*debit*). Таким образом, добавим к ним требования, накладываемые на значения этих переменных правилами кошелька:

```
<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>
```

Также учтем ограничения, накладываемые на авторизацию:

```
<outputActionCond ident="o1.z4">
    <precond value="pin_counter &lt;
PIN_TRY_LIMIT"/>
    <precond value="transaction &gt; 0"/>
    <postcond value="pin_counter == 0"/>
</outputActionCond>
```

3.1.3. Генерация кода апплета

Основой полученного кода является главная функция управления состояниями и обработки событий `processAutomataEvent`:

```
switch (fsm_state) {
    case S2_STAND_BY:
        if (!o1_x1()) {
            o1_z5();
            fsm_state = S2_STAND_BY;
        }
        switch (event) {
            case e4:
             if (o1_x1()) {
                o1_z4();
                fsm_state = S5_CHECK_PIN;
             }
             break;
```

```
            case e2:
                    if (o1_x1()) {
                        o1_z2();
                    fsm_state = S3_DEBIT_OPERATION;
                    }
                    break;
                case e7:
                    fsm_state = S2;
                    break;
                case e1:
                    if (o1_x1()) {
                        o1_z1(tr, bal);
                    fsm_state = S3_CREDIT_OPERATION;
                    }
                    break;
                case e3:
                    if (o1_x1()) {
                        o1_z3();
                     fsm_state = S4_BALANCE_REQUEST;
                    }
                    break;
            }
            break;
        case S5_CHECK_PIN:
            switch (event) {
                case e5:
                    fsm_state = S2_STAND_BY;
                    break;
            }
            break;
        case S4_BALANCE_REQUEST:
            switch (event) {
                case e5:
                    fsm_state = S2_STAND_BY;
                    break;
            }
            break;
        case S2_CREDIT_OPERATION:
            switch (event) {
                case e5:
                    fsm_state = S2_STAND_BY;
                    break;
            }
            break;
        case S3_DEBIT_OPERATION:
```

```
                switch (event) {
                    case e5:
                        fsm_state = S2_STAND_BY;
                        break;
                }
                break;
        }
```

Наличие этой функции гарантирует правильный путь потока управления апплета, так как все события и переходы с ними были описаны в *UniMod*-модели. Помимо этого, генерируются прототипы функции управления с набором автоматически добавленных к ним утверждений, в соответствии с требованиями, описанными на этапе проектирования. Расширить логику и функциональность этих функции разработчик должен в большинстве случаев вручную.

Следует отметить одно из преимуществ добавления требований на переменные на этапе проектирования – при написании логики функции, наложенные на нее требования невозможно забыть или проигнорировать, так как они уже встроены в код заглушки. В качестве примера приведем сгенерированную заглушку функции, выполняющей снятие денег со счета:

```
private void o1_z2() {
  /* preconditions */
  assert (transaction < MAX_TRANSACTION_AMOUNT);
  assert (transaction > 0);

  /* stub implementation */

  /* precondition */
  assert (balance > 0);
}
```

3.1.5. Реализация заглушек и тестирование

Реализуем все сгенерированные функции самым простейшим способом, решающим поставленную задачу с финансовыми операциями. Как было описано раннее, вся работа с *APDU*-данными вынесена в функцию, которая должна быть реализована отдельно. В зависимости от типа события, эта функция получает нужные данные из *APDU*-команды и сохраняет их в заданные глобальные

переменные. Это позволяет абстрагироваться от этих вопросов на этапе реализации логики объекта управления. Таким образом, к моменту вызова функции работы со счетом, можно считать, что размер транзакции, содержащийся в запросе, считан, и переменная `transaction` содержит соответствующее значение.

Функция `o1.z1` реализует операцию пополнения счета:

```
private void o1_z2() {
    /* preconditions */
    assert (transaction <
MAX_TRANSACTION_AMOUNT);
    assert (transaction >= 0);
    //credit
    balance = (short)(balance + transaction);

    /* postconditions */
    assert (balance >= 0);
     assert (balance < MAX_BALANCE);
}
```

Функция `o1.z2` реализует операцию снятия денег со счета:

```
private void o1_z2() {
    /* preconditions */
    assert (transaction <
    MAX_TRANSACTION_AMOUNT);
    assert (transaction >= 0);
    //debit
    balance = (short)(balance - transaction);

    /* postconditions */
    assert (balance >= 0);
   assert (balance < MAX_BALANCE);
}
```

Функция `o1.z3` возвращает текущий баланс, при помощи вызова функции работы с *APDU*:

```
private void o1_z3() {
    makeAPDUResponse(balance);
}
```

Функция `o1.z4` выполняет авторизацию:

```
private void o1_z4() {
        /* preconditions */
        assert (pin_counter <= 3);
        //authorization attempt
        pin_counter = (byte)(1+pin_counter);
        checkPIN();
        /* postconditions */
        assert (pin_counter == 0);
    }
```

От функции `checkPIN` ожидается следующее поведение – она реализует проверку полученного *PIN*-кода, и, в случае несовпадения, создает исключение, а, в случае успеха, выставляет значение флага, проверяемое в функции `o1.x1`:

```
    private boolean o1_x1() {
        if(pin_flag) {
            return true;
        } else {
            return false;
        }
    }
```

В итоге, приведенная простая реализация функций должна выполнять все необходимые операции. Выводы о корректности этой реализации будут сделаны далее, с учетом результатов проверки.

Следующим шагом, при помощи транслятора создается входная модель верификатора *Moped*. Используя файл свойств, созданный на этапе генерации кода, формируются два списка меток, проверка достижимости которых автоматизирует процесс тестирования и позволит оценить реализацию:

- список меток, соответствующих успешно выполненным переходам между состояниями:

```
A1Applet_transition_from_S2_STAND_BY_to_S4_BALANCE_RE
QUEST__V0
A1Applet_transition_from_S2_STAND_BY_to_S5_CHECK_PIN_
_V0
A1Applet_transition_from_S2_STAND_BY_to_S2__V0
A1Applet_transition_from_S2_STAND_BY_to_S2_STAND_BY__
V0
A1Applet_transition_from_S5_CHECK_PIN_to_S2_STAND_BY_
_V0
A1Applet_transition_from_S2_STAND_BY_to_S3_DEBIT_OPER
ATION__V0
A1Applet_transition_from_S3_DEBIT_OPERATION_to_S2_STA
ND_BY__V0
A1Applet_transition_from_S4_BALANCE_REQUEST_to_S2_STA
ND_BY__V0
A1Applet_transition_from_S3_CREDIT_OPERATION_to_S2_ST
AND_BY__V0
A1Applet_transition_from_S2_STAND_BY_to_S3_CREDIT_OPE
RATION__V0
```

- список меток, соответствующих переходу в каждое, из выделенных в автоматной модели, состояние:

```
A1Applet_set_state_to_S4_BALANCE_REQUEST__V0
A1Applet_set_state_to_S5_CHECK_PIN__V0
A1Applet_set_state_to_S2_STAND_BY__V0
A1Applet_set_state_to_S2__V0
A1Applet_set_state_to_S3_DEBIT_OPERATION__V0
A1Applet_set_state_to_S3_CREDIT_OPERATION__V
```

Далее, при помощи автоматизированного скрипта выполняют предложенные в главе 2 проверки с учетом заданного интервала входной переменной программы – размера транзакции `transaction`:

1. Проверка всех возможных состояний и всех возможных событий для каждого из них.

    a. В первую очередь, производится проверка достижимости состояния «error», сигнализирующего о нарушении какого-либо утверждения, выводящая набор аргументов, приведший к этому нарушению. Выдается следующий результат:

    ```
    “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);
    ```

    при значении fsm_state=S1_STAND_BY, event=e1 и значении transaction=200. Ошибку следует исправить, добавив в код следующие строки:

    ```
    if ( transaction > MAX_TRANSACTION_AMOUNT ) {
       ISOException.throwIt(SW_INVALID_TRANSACTION_
    AMOUNT);
    }
    ```

    Аналогичная ошибка будет обнаружена в функции o1.z2 и должна быть исправлена.

    b. Затем запускается проверка выполнимости переходов, которая выдает следующий результат:

    ```
    transaction s1_Stanb_By_to_s1_Stand_by not
    reachable
    e3
    ```

Метка является недостижимой – переход невыполним. При внимательной проверке реализации задействованных вызовов можно заметить ошибку в функции `o1.z5` – зацикливание. Исправив условие, повторим проверку и убедимся, что переход признан выполнимым.

2. Проверка достижимости состояний из начального.

    a. Также как в предыдущем варианте, вначале запустим проверку достижимости ошибочного состояния:

```
"error" is reachable!
Trace:
s1_Stand_By
e1
transaction s1_Stand_by_to_s2_Credit_Operation
e5
s1_Stand_By
e1
transaction s1_Stand_by_to_s2_Credit_Operation
e5
s1_Stand_By
e1
transaction s1_Stand_by_to_s2_Credit_Operation
```

    Из полученного пути к ошибке можно сделать следующий вывод – при получении трех запросов на увеличение счета, суммарный баланс превышает константу `MAX_BALANCE`.

    b. Проверка достижимости состояний для данного примера показывает, что все состояния достижимы, кроме одного: `s6_Blocked not reachable`

    Предложенная проверка не может гарантировать, что состояние не достижимо не практике, в силу сложности этой задачи. Это явля-

ется поводом для более тщательной инспекции написанного кода. В итоге, можно понять, что переменная *pin_counter* не увеличивается, и поэтому состояние блокировки и вправду не достижимо.

3.2. Апплет для отправки сообщений

В рамках полуфинала конкурса *SIMagine* была поставлена задача реализовать *JavaCard*-апплет, расширяющий возможности приложений отправки *SMS*-сообщений с телефона. В связи с этим, рассмотрим процесс разработки небольшого приложения, реализующего отправку сообщения с необходимой информацией из приложения во внешний мир. Наглядный пример такого рода задачи – отправка *SMS*-сообщения с *GSM*-телефона, *SIM*-карта которого поддерживает платформу *Java Card*. Телефон играет роль клиента, способного посылать *APDU*-запросы на свою *SIM*-карту. Запущенные на *SIM*-карте приложения являются серверами, ожидающими запроса.

3.2.1. Описание задачи

Для иллюстрации преимуществ предложенного подхода абстрагируемся от технических деталей, связанных с использованием библиотеки *SIM Toolkit* [40], реализующей функции отправки *SMS*-сообщений и прочих особенностях применения *Java Card* на *SIM*-картах. В рамках интересующей нас задачи разработки логики подобных приложений, достаточно знания того, что *SMS*-сообщение является специально сформированной *APDU*-командой, отсылаемой апплетом.

Таким образом, можно выделить следующие вопросы, которым стоит уделить внимание в контексте поставленной задачи:

1. Получение данных сообщения, таких как текст и идентификатор получателя, от пользователя.

Следует уточнить, что эти данные будут получены приложением в виде *APDU*-команды, а не привычным, в случае *Java ME* или *Java SE*, обращением к интерфейсу приложения.

2. Проверка валидности полученной информации.

Существуют определенные ограничения, как на размер текста сообщения, так и на номер получателя.

3. Вызов методов только из допустимых состояний.

Отправка некорректно сформированной *APDU*-команды, содержащей текст *SMS*-сообщения, может привести к нарушению безопасности данных или некорректному поведению телефона.

3.2.2. Расширенная автоматная модель апплета

Рассмотрим спроектированную при помощи *UniMod* схему связей на рис. 3.3 для поставленной задачи.

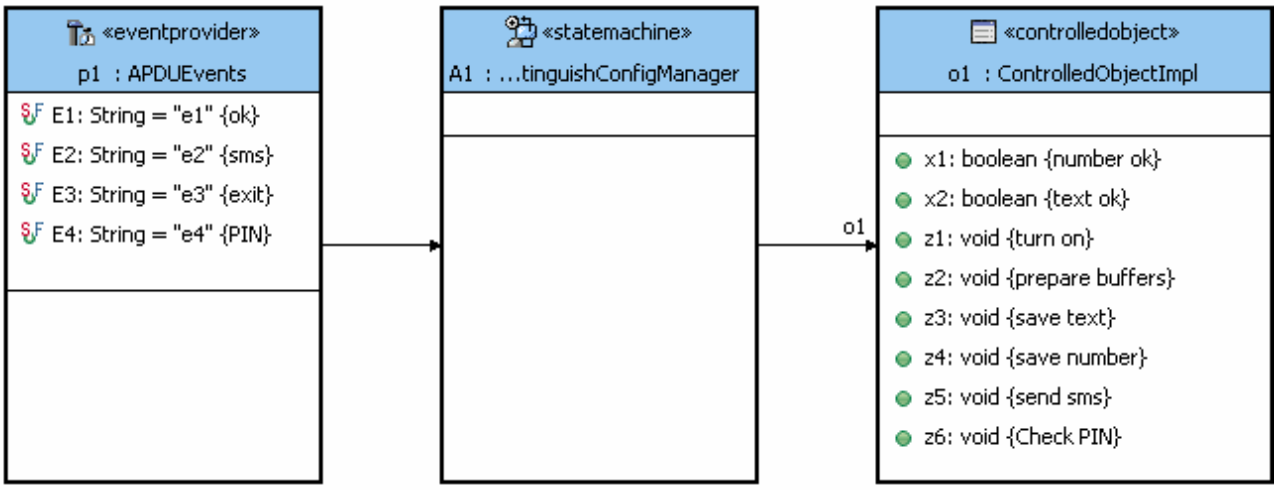

Рис. 3.3. Схема связей

Поставщиком событий в рассматриваемой задаче является телефон, отсылающий запросы, содержащие выбор пользователя и введенную информацию. В примере описаны три события:

1. Событие *e1* – в зависимости от текущего состояния апплета, сигнализирующее о том, что пользователь закончил ввод текста или ознакомился с информацией переданной приложением.
2. Событие *e2* – пользователь выбрал режим отправки сообщения.
3. Событие *e3* – следует завершить работу.

Возможно, для наглядности схемы следовало бы сопоставить отдельные события для получения текста сообщения, номера сообщения, согласия на отправку и прочих возможных действий пользователя. Как уже было сказано, взаимодействие с пользователем программ, рассчитанных на мобильные устройства, отличается по своей сути от традиционного взаимодействия, рассчитанного на полноценные приложения. Ярким подтверждением тому являются существующие приложения оправки *SMS*-сообщений на мобильных телефонах – при их использовании очень редко задействована более чем одна кнопка управления. Благодаря четко описанной последовательности возможных действий, ее нажатие всегда несет в себе понятный приложению смысл.

В описываемом примере таким событием является *e1*. Рассматриваемый пример иллюстрирует удобство применения автоматного подхода для разработки подобных интерфейсов, упомянутое ранее – при стандартном подходе к написанию программ, любое событие должно содержать в себе все необходимые сведения для его обработки. В случае автоматного подхода, решение о необходимых действиях принимается на основе не одного параметра, а двух: произошедшее событие и текущее состояние. Таким образом, зная свое состояние в момент получения события *e1*, апплет явно может определить, какому действию пользователя оно соответствует – завершению ввода текста или команде отправить сообщение. Благодаря этому можно создать удобный интерфейс при помощи всего нескольких кнопок, и, более того, упростить обмен данными между апплетом и устройством, так как не требуется передавать никакую пояснительную информацию. Поясним этот факт при помощи графа переходов автомата *A1*, изображенного на рис. 3.4, который описывает поведение приложения.

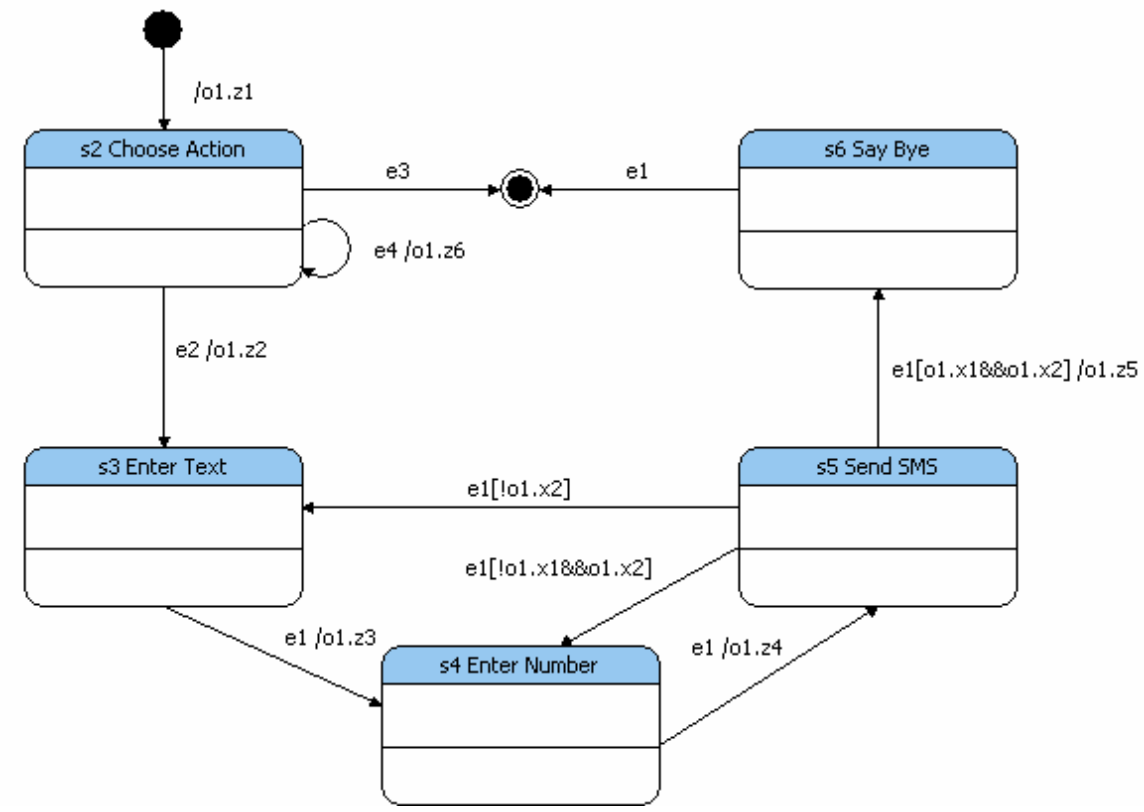

Рис. 3.4. Граф переходов автомата *A1*

Рассмотрим возможные состояния автомата и переходы между ними.

1. *Choose Action* (Выбор действия) – апплет ожидает от пользователя выбор действия. В рассматриваемом примере, ограничимся двумя вариантами: завершить работу (событие *e3*) – приложение заканчивает работу, отправить сообщение (событие *e2*) – начать составление сообщения, автомат переходит в состояние 2. При этом вызывается функция объекта управления *z2*, которая инициализирует необходимые переменные и готовит апплет к приему информации.
2. *Enter Text* (Ввод текста) – приложение ожидает ввода текста сообщения. Событие *e1* в данном состоянии означает, что текст введен и передан в *APDU*-команде. Поэтому апплет переходит в состояние 3, сделав все необходимые преобразования над текстом и сохранив его при помощи функции `o1.z3`.
3. *Enter Number* (Ввод номера) – аналогично состоянию 2, апплет ожидает событие *e1*, но на этот раз при его получении из *APDU*-команды будет

извлечен номер получателя, сохранен в соответствующем формате при помощи функции `o1.z4` и апплет переходит в состояние 4.

4. *Send SMS* (Отправка сообщения) – в этом состоянии получение события *e1* будет рассматриваться как согласие пользователя на отправку сообщения. Никакая дополнительная информация из *APDU*-команды извлекаться не будет. При его получении будет произведен анализ текущего состояния и сделан один из следующих переходов:
   - если текст и номер корректны (функции проверки `o1.x1` и `o1.x2` соответственно), то переход в состояние 5 и отсылка сообщения функцией `o1.z5`;
   - если номер или текст не корректны, то переходы в состояние 2 или 3 соответственно.
5. *Say Bye* (Завершение работы) – событие *e1* означает, что приложение должно закончить работу, освободив при этом все задействованные ресурсы и вызвав функцию `o1.z5`.

3.2.3. Реализация заглушек и тестирование

Основой полученного кода является главная функция управления состояниями и обработки событий `processAutomataEvent`.

Помимо этого, генерируются заглушки функции управления, реализовать которые разработчик должен вручную:

```
private void o#_z#() {
        /* preconditions */

        /* stub implementation */

        /* postconditions */
}
```

Рассмотрим переход из состояния 4 (*Send SMS*) в состояние 5 (*Say Bye*). При генерации свойств этому переходу была сопоставлена метка `transition_5_to_6`. Выполнение этого перехода происходит следующим

образом – сначала проверяются условия `o1.x1` и `o2.x2`, затем вызывается функция `o1.z5`, реализующая выходное воздействие.

Проверка выполнимости этого перехода показывает, что он выполним при использовании сгенерированных автоматически заглушек объекта управления `o1`:

```
private void o1_z2() {
        /* stub implementation */
}
private void o1_x1() {
    /* stub implementation */
    return true;
}
private void o1_x2() {
    /* stub implementation */
    return true;
}
```

Реализуем любой из этих методов заведомо неверным способом и проверим будет ли обнаружена эта ошибка. Пусть метод `o1.z1` содержит цикл с ошибочным условием:

```
private void o1_z2() {
   byte b = 2;
   while (b <= 7) { //b will never be > 7
       b = b+1;
       b = b-1;
    }
}
```

Предложенная реализация метода будет нормально компилироваться. В случае проверки выполнимости переходов будет установлено, что состояние 5 (*Say Bye*) недостижимо, так как переход невыполним.

Следовательно, приложение никогда не будет выполняться корректно и эта реализация явно не соответствует спецификации. Аналогично, можно обнаружить такую ошибку:

```
private boolean o1_x1() {
    if(1<2) {
        return false;
    } else {
        return true; //never occures
    }
}
```

Проиллюстрируем на этом примере преимущества ручного добавления утверждений в *XML*-описание. Предположим апплету необходимо изменить свое окружение для отправки сообщения. Например, записать в память карты значение. После выполнения апплета для корректной дальнейшей работы карты требуется вернуть этому сегменту памяти начальное значение. Для наглядности предположим, что это просто глобальная переменная `a`, значение которой должно быть равно единице во время работы апплета и нулю в остальные моменты. Добавим блок описания глобальных переменных:

```
<globalvars>
        <var type="byte" name="a" />
</globalvars>
```

Апплет завершает свою работу, попадая в конечное состояние по переходу из состояния 5 (*Say Bye*). При попадании в это состояние вызывается метод `o1.z7`. В расширенной модели добавим следующее утверждение, проверяющее требуемое спецификацией свойство:

```
<outputActionCond ident="o1.z7">
    <postcond value="a == 0"/>
</outputActionCond>
```

После генерации заглушек функции, метод `o1.z5` будет выглядеть так:

```
private void o1_z7() {
        /* postconditions */
        assert(a==0);
}
```

Добавим в `o1.z1` код, изменяющий значение переменной `a` на единицу, а в коде функции `o1.z6` будем выставлять значение `a` на ноль. Схема переходов автомата показывает, что при правильной последовательности действий, до отправки будет вызвана функция `o1.z1`, а после отправки функция `o1.z6`. Если же запустить проверку на нарушение условий, то окажется, что существует развитие событий, при котором утверждение не выполнено.

Анализ контрпримера показывает, что условие может быть нарушено при попадании в конечное состояние из состояния 1 – в том случае, когда пользователь отказывается отправлять сообщение.

Выводы по главе 3

На примере разработки простых апплетов предложенные способы проверки позволили выявить потенциальные ошибки в коде:

- ошибка в условии цикла в функции управления, которая приводила к тому, что апплет зациклился бы во время ее выполнения;
- отсутствие явной проверки значений – реализация не соблюдала ограничения, заданные в спецификации;
- недостижимость состояния, так как условие для 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.

Также, разработка описанных примеров на практике подтвердила:

1. Преимущества автоматного подхода при проектировании – приложения для *Java Card* функционально эквивалентны конечному автомату как событийному объекту, поэтому автоматная модель является удобным

способом описания поведения апплета. Также это удобное решение в случае проектирования пользовательских интерфейсов для *JavaCard*-апплетов, рассчитанных для использования на мобильных телефонах.

2. Возможность проверки реализации объектов управления – в случае явных ошибок логики приложения, состояние, соответствующее успешной отправке сообщения, не достижимо. Это позволяет сделать вывод, что реализация некорректна и не соответствует спроектированной модели.
3. Преимущества расширенной модели – формулирование требований на этапе проектирования дает возможность проверять реализацию и находить множество ошибок, возможно незаметных при ручной проверке кода.

ЗАКЛЮЧЕНИЕ

Перечислим полученные в рамках данной работы результаты:

1. Предложен способ разработки *JavaCard*-апплетов с применением автоматного подхода – сформулирована связь между понятиями присутствующими в контексте платформы *Java Card* и в контексте автоматного подхода. Показаны преимущества выбора именно этого подхода.
2. Предложен расширенный вариант *XML*-представления автоматной модели, позволяющий на этапе проектирования задать ряд условий на реализацию функций объектов управления.
3. Написан генератор кода с заглушками для *JavaCard*-апплетов из предложенного *XML*-представления, учитывающий особенности платформы *Java Card* и генерирующий утверждения, соответствующие условиям, добавленным в модель.
4. Расширена функциональность транслятора *jMoped*, создающего по байт-коду входную модель верификатора *Moped*. Учтены отличия 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 *JavaCard*-апплета и спроектированной автоматной модели.
6. Написан инструмент, автоматизирующий предложенный процесс тестирования и конвертирующий контрпримеры верификатора в удобный формат, содержащий элементы автоматной модели.

Пути развития работы.

1. Удобный формат вывода результатов, таких как процент покрытого тестированием кода и полученная в ходе проверок информация.
2. Интеграция с инструментальным средством *UniMod* – создание интерфейса для добавления утверждений к функциям объектов управления, отображения полученной информации о достижимости состояний с учетом реализации объектов управления на схеме переходов автомата.

ИСТОЧНИКИ

1. *Технология Java Card*. Официальный сайт. http://java.sun.com/javacard/
2. *Java Card*. Wikipedia. The free encyclopedia. http://en.wikipedia.org/wiki/Java_Card
3. *Шалыто А. А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/alg_log
4. *Конкурс SIMagine*. http://www.simagine.gemalto.com
5. *Клебанов А. А.* Генерация исходного кода доказуемо-корректных *JavaCard*-программ с использованием автоматного подхода. СПбГУ ИТМО. Бакалаврская работа. 2008. http://is.ifmo.ru (раздел «Работы»).
6. *Чарнецки К., Айзенекер У.* Порождающее программирование: методы, инструменты, применение. СПб.: Питер, 2005.
7. *Вельдер С. Э., Шалыто А. А.* Введение в верификацию автоматных программ на основе метода *Model checking*. http://is.ifmo.ru/download/modelchecking.pdf
8. *Clarke E. M., Grumberg O., Peled D. A.* Model checking. The MIT Press. 2000.
9. *Гуров В. С., Мазин М. А.* Веб-сайт проекта *UniMod*. http://unimod.sourceforge.net/
10. *Набор разработчика Java Card*. http://java.sun.com/javacard/devkit/
11. *Стандарт ISO 7816*. http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816.aspx
12. *APDU*. Wikipedia. The free encyclopedia. http://en.wikipedia.org/wiki/Protocol_data_unit

13. *Технология «клиент-сервер»*. Wikipedia. The free encyclopedia. http://en.wikipedia.org/wiki/Master-slave_%28computers%29
14. *Салмре И.* Программирование мобильных устройств на платформе *.Net Compact Framework*. М.: Вильямс. 2006. http://is.ifmo.ru/progeny/mobdev/
15. *Шалыто А. А.* Разработка технологии создания программного обеспечения систем управления на основе автоматного подхода. http://is.ifmo.ru/science/1/
16. *Шалыто А. А., Гуров В. С., Мазин М. А.* Мобильные системы. http://is.ifmo.ru/science/MD-Mobile.pdf
17. *DTD-формат XML-представления модели инструмента UniMod.* http://unimod.sourceforge.net/statemachine.dtdx.html
18. *Automatic programming*. Wikipedia. The free encyclopedia. http://en.wikipedia.org/wiki/Automatic_programming
19. *Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.* UML. SWITCH-технология. Eclipse //Информационно-управляющие системы. 2004. № 6. http://is.ifmo.ru/works/uml-switch-eclipse/
20. *Канжелев С. Ю., Шалыто А. А.* Преобразование графов переходов, представленных в формате *MS Visio*, в исходные коды программ для различных языков программирования (инструментальное средство MetaAuto). СПбГУ ИТМО. 2006. http://is.ifmo.ru/projects/metaauto/
21. *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. http://is.ifmo.ru/verification/_2007_01_patent-verification.pdf
22. *Holzmann G. J.* The *SPIN Model Checker*: Primer and Reference Manual. Addison-Wesley. 2004.
23. *Верификатор Bogor*. Официальный сайт. http://bogor.projects.cis.ksu.edu/

24. *Havelund K., Pressburger T. Model Checking Java Programs* using Java PathFinder. 1998. www.havelund.com/Publications/
25. *Bandera Project*. http://bandera.projects.cis.ksu.edu/
26. *Среда тестирования jMoped translator*. www7.in.tum.de/tools/jmoped/
27. *Верификатор Moped Model Checker*. www.fmi.uni-stuttgart.de/szs/tools/moped/
28. *Beyer D., Henzinger T., Jhala R., Majumdar R.* The Software Model Checker Blast: Applications to Software Engineering. Int. Journal on Software Tools for Technology Transfer. 2007.
29. *Suwimonteerabuth D., Berger F., Schwoon S., Esparza J. jMoped*: A Test Environment for Java Programs. http://www.fmi.uni-stuttgart.de/szs/publications/
30. *Генератор AutoSmart* http://www.kestrel.edu/home/projects/jcapplets/
31. *Deharbe D., Gomes B., Moreira A.* Automation of Java Card component development using the *B* method. http://download.gna.org/brillant/docs/B-Bibliography/Bmethod-online.html
32. *Проект Verificard*. http://www-sop.inria.fr/lemme/verificard/
33. *Верификатор связей между апплетами jCave*. http://www.sics.se/fdt/projects/vericode/jcave.html
34. *Инструмент для статической проверки Java-кода ESC/JAVA2* en.wikipedia.org/wiki/ESC/Java
35. *Инструмент ChASe.* http://www-sop.inria.fr/lemme/verificard/modifSpec/
36. *Poll E.* From Finite State Machines to Provably Correct Java Card Applets. www.cs.ru.nl/E.Poll/papers/
37. *Stenzel K.* Verification of *JAVA CARD*-programs. Technical Report 2001–5. Institut f¨ur Informatik. Universit¨at Augsburg. Germany. 2001.
38. *Верификатор KeY.* http://key-project.org/

39. *Ort E.* Writing a Java Card Applet. http://developers.sun.com/mobility/javacard/articles/intro/

40. *SIM Toolkit*. Wikipedia. The free encyclopedia. http://en.wikipedia.org/wiki/SIM_Application_Toolkit/

41. *Шалыто А. А., Туккель Н. И.* Программирование с явным выделением состояний. //Мир ПК. 2001. № 8. http://is.ifmo.ru/works/mirk/

42. *Модель DOM*. Wikipedia. The free encyclopedia. http://ru.wikipedia.org/wiki/DOM