Научно-технический вестник Санкт-Петербургского государственного университета информационных технологий, механики и оптики. Вып. 53. Автоматное программирование. 2008, с. 189–198.

УДК 004.4’242

ПРИМЕНЕНИЕ АВТОМАТНОГО ПОДХОДА ДЛЯ СОЗДАНИЯ КОРРЕКТНЫХ *JAVA CARD*-ПРИЛОЖЕНИЙ

А. Ю. Законов

В статье предлагается подход, повышающий надежность процесса разработки *Java Card*-апплетов. Формулируются особенности проектирования апплетов при применении автоматного подхода, предлагается расширенная утверждениями модель для описания функциональности, генератор кода с заглушками и инструмент для автоматизации тестирования полученных апплетов.

Введение

С каждым днем мобильные устройства занимают все большую роль в повседневной жизни. Их функциональность растет, так же, как и сфера их применения. Примером этого являются смарт-карты. Эти устройства получили широкое применение в различных областях – от систем накопительных скидок до кредитных и дебетовых карт, телефонов стандарта *GSM*, карт доступа и проездных билетов. Смарт-карты используют платформу *Java Card* [1], позволяющую программам, написанным на языке *Java*, исполняться на интеллектуальных картах и других устройствах с ограниченными ресурсами. Со временем растет как сложность поставленных задач, так и, соответственно, решающих их программ. При этом повышаются требования к безопасности кода, так как для банковской сферы и сферы мобильных приложений цена ошибки может быть очень велика и надежность является ключевым вопросом.

Известно, что *Java Card*-апплет функционально эквивалентен конечному автомату как событийный объект [2]. Однако, несмотря на это, на момент написания статьи не существовало работ и инструментов, где проектирование приложения начиналось бы с построения конечного автомата. Идея использовать автоматный подход [3] для описания поведения апплета появилась и была использована в процессе написания *Java Card*-приложения в рамках участия в финале конкурса *SIMagine* [4]. Кроме того, отметим, что существующие средства верификации и тестирования апплетов предназначены для проверки полученного *Java Card*-кода или байт-кода, но никак не охватывают процесс разработки. Создание спецификации, требований и написание кода происходят абсолютно независимо. Таким образом, существует разрыв между спецификацией и реализацией поставленной задачи.

Поэтому в данной работе поставлена следующая задача: разработать подход и набор инструментов для разработки, позволяющих сократить разрыв между описанием функциональности приложения и реализующим его кодом. Подход должен включать в себя все этапы создания *Java Card*-апплета, начиная с проектирования поведения апплета и заканчивая проверкой реализации на предмет соответствия построенной модели.

Предлагаемый подход объединяет в себе преимущества автоматного программирования, автоматической генерации кода [5] и техники автоматической проверки моделей программ *Model Checking* [6]. Для построения автоматной модели используется инструментальное средство *UniMod* [7]. Успешное применение этих технологий позволяет существенно автоматизировать и упростить процесс разработки, а также объединить процессы написания кода и написания документации.

Обзор используемых технологий

Платформа *Java Card* позволяет программам, написанным на языке *Java*, исполняться на интеллектуальных картах и других устройствах с ограниченными ресурсами. В качестве наиболее известных примеров можно привести телефонные *SIM*-карты и банковские *ATM*-карты. Наиболее полное описание особенностей данной платформы можно найти на официальном сайте [1].

Приложения, написанные для платформы *Java Card*, называются апплетами. Помимо языка, платформа *Java Card* поддерживает среду исполнения смарт-карт приложений, позволяя одному и тому же *Java Card*-апплету работать на разных смарт-картах (аналогично тому, как *Java*-апплет может быть запущен на разных платформах). Так же, как и в *Java*, это достигается благодаря использованию виртуальной машины *(Java Card Virtual Machine)* и библиотек исполнения. Важными достоинствами технологии *Java Card* являются безопасность и портируемость.

В силу ограничений на доступную память и вычислительную мощность, платформа *Java Card* поддерживает только некоторое подмножество языка программирования *Java*. Все языковые конструкции *Java Card* существуют в *Java* и выполняются идентично. В спецификации платформы *Java Card 2.2* многие возможности языка *Java* отсутствуют в языке *Java Card*: длинные примитивные типы данных, символы и строки, многомерные массивы, динамическая загрузка классов, сборка мусора, многопоточность, сериализация и клонирование объектов.

Java Card-код компилируется при помощи стандартного *Java*-компилятора. Полученный байт-код обрабатывается инструментами, специфичными для платформы *Java Card*. Байт-код, поддерживаемый *Java Card VM*, также является подмножеством *Java SE* байт-кода, выполняемого на виртуальной *Java*-машине, но из соображений уменьшения объема байт-кода использует другой принцип сжатия.

Платформа *Java Card* работает по принципу «клиент-сервер» [8]. Данные передаются при помощи *Application Protocol Data Unit (APDU)* пакетов. Апплет реагирует на получение *APDU*-команды, выполняет действия, затем отсылает ответ на запрос. Это напоминает поведение конечного автомата, который реагирует на событие, и, в зависимости от типа события, текущего состояния и условий, переходит в новое состояние, выполняя при переходе заданное действие. В работе [3] был предложен метод проектирования программ с явным выделением состояний, названный «автоматное программирование» или «*SWITCH*-технология». В этой технологии базовым является понятие состояния. Автомат рассматривается как совокупность четырех понятий: состояние, переход, входное и выходное воздействие, 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. Таким образом, концепция автоматного программирования хорошо подходит для создания *Java Card*-апплетов.

Важным вопросом является проверка корректности полученного апплета. При традиционном подходе тестирование программ и проверка моделей являются далекими друг от друга методологиями. Тестирование направлено на выявление ошибок, а не на доказательство их отсутствия. *Model Checking* [6] – это метод, при котором алгоритмически доказывается, соответствует ли формальная система спецификации или некоторому свойству. Таким образом, проверяют достижимость определенного состояния или же более сложные свойства системы, что позволяет обеспечивать корректность программ. Подробный обзор существующих в этой области результатов приведен в работе [9].

В рамках *Model Checking* отдельно следует отметить подход, при котором построение модели производится автоматически из кода программы. В этом случае на вход подается исполняемый байт-код и по нему строится модель. Этот подход используется в инструментах *Java Pathfinder* [10] и *jMoped translator* [11].

В последнее время появились инструменты, использующие проверку моделей для тестирования программ. Такой возможностью обладает *jMoped* – среда тестирования программ, написанных на языке *Java*.

Подход, повышающий надежность разработки *Java Card*-апплетов

Для решения поставленной задачи предлагается использовать возможности ряда существующих подходов, и создать инструмент, который позволяет сократить разрыв между спецификацией и реализацией приложения. На этапе проектирования применяется автоматный подход и инструментальное средство *UniMod*. Создание прототипа кода выполняется при помощи написанного генератора. Сгенерированный код отвечает за поведение апплета и обработку поступающих сообщений – таким образом гарантируется соответствие поведения разрабатываемого апплета и спроектированного конечного автомата. Полностью описывать функциональность апплета при помощи автоматной модели невозможно. Поэтому обеспечивается сочетание автоматически сгенерированного по автоматам кода и кода для объектов управления, написанного вручную. Предлагается расширить автоматную модель возможностью добавлять утверждения (*assertions*), которые будут накладывать некоторые ограничения и требования на добавленный код. Для проверки сформулированных требований используется транслятор *Java Card* байт-кода в язык *Remolpa*, являющийся расширением транслятора *jMoped*, и верификатор *Moped Model Checker*. Сформулируем общую последовательность действий при разработке *Java Card*-приложения:

1. Создать автоматную модель приложения в инструментальном средстве *UniMod*.
2. Расширить *XML*-представление модели необходимыми утверждениями, проверяющими те свойства реализации, которые возможно сформулировать на этапе проектирования модели и написания требований к приложению.
3. Используя разработанный генератор кода, создать код апплета с заглушками.
4. Вручную реализовать методы объектов управления.
5. Используя предложенные инструменты, проверить, соответствует ли разработанный апплет набору требований, автоматически сгенерированных из свойств, сформулированных на шаге 2 и автоматному описанию, построенному на шаге 1.
6. В случае нарушения тех или иных свойств, верификатор *Moped* выдает в качестве результата контрпример. Для анализа этого контрпримера предлагается использовать написанное автором настоящей работы средство, позволяющее преобразовать данные контрпримера в понятную пользователю последовательность событий и переходов между состояниями конечного автомата, спроектированного на шаге 1.

Перечисленные этапы разработки и применение предложенных инструментов отражены на рис. 1.

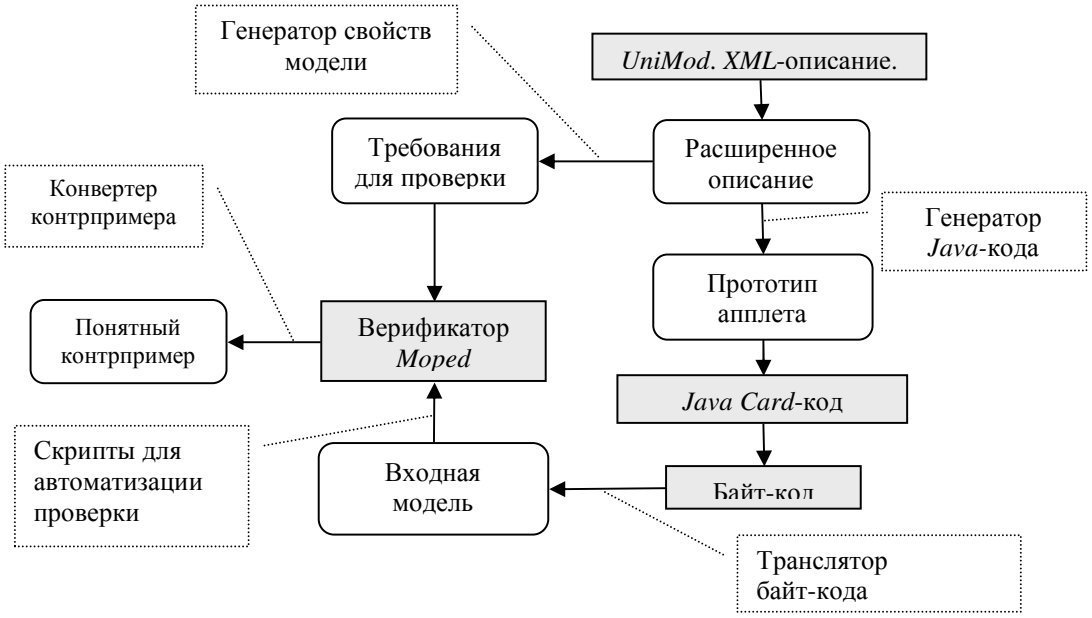

Рис. 1. Предложенный подход

Имея средство проверки достижимости состояний и выполнения утверждений *jMoped* можно проверить программу с реализованными объектами управления на достижимость состояний и последовательность переходов. Получив такую информацию, можно проанализировать корректность добавленного кода и соответствие описанной модели и построенного апплета.

Если на этапе проектирования дополнительно описать необходимые требования к переменным, использованным в функциях объектов управления, то это позволит проверить автоматически, соответствует ли дописанный код этим требованиям.

Расширенная автоматная модель *Java Card*-апплета

Для построения удобной автоматной модели апплета следует определиться с терминами и понятиями, существующими как в *Java Card*, так и в автоматном подходе.

Для *Java Card*-апплета событием является получение *APDU*-сообщения, которое по своей сути является массивом байт. Для удобства использования автоматного подхода, предлагается генерировать код обработки событий, используя информацию поставщиков событий, описываемых в инструменте *UniMod*. Каждому поставщику событий сопоставляется определенное значение *CLA*–байта, а каждому конкретному событию – уникальное значение *INS*–байта [12].

В рамках предложенного подхода возникает вопрос: соответствует ли дописанный руками код условиям модели и выполняет ли он требуемые от него действия? Для автоматически полученного кода на этот вопрос можно ответить, проверяя автоматную модель и корректность процесса генерации. Про написанный вручную код никаких выводов сделать нельзя. Поэтому предлагается расширить существующую модель возможностью добавлять свои условия, выполнимость которых будет проверена на этапе автоматического тестирования. Описанный подход проиллюстрирован на рис. 2.

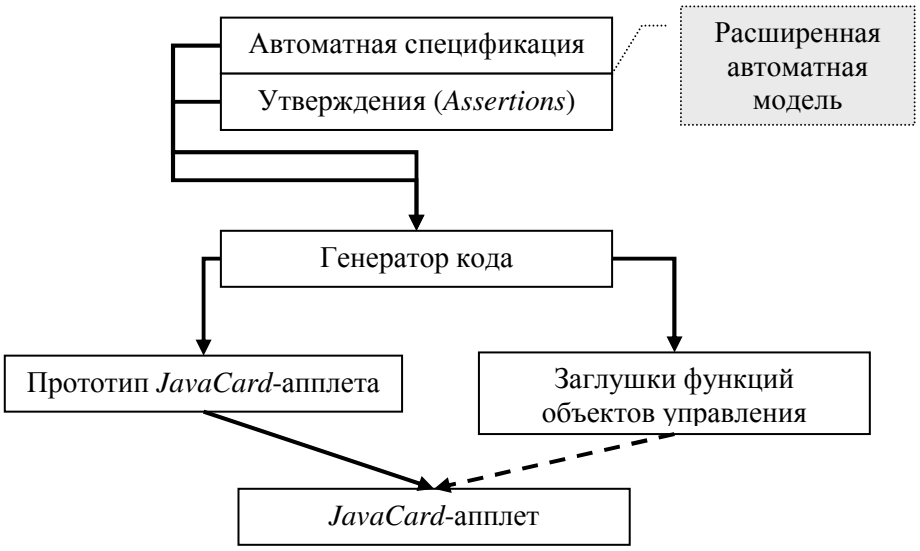

Рис. 2. Подход, использующий расширенную автоматную модель

Еще одним важным аргументом в пользу расширения модели является обработка *APDU*-сообщений, которые присутствуют в большинстве *Java Card*-апплетов и являются проблемными для тестирования. *APDU*-команда содержит две важные составляющие – информацию о произошедшем событии и дополнительные данные, переданные вместе с ним. На поведение и ход исполнения апплета также влияют данные, переданные в качестве параметров. Принимая во внимание, что *Java Card* поддерживает только целочисленные значения переменных, в рамках данной работы вводится следующее ограничение: все функции, использующие *APDU*-данные, возвращают целочисленное значение. При этом код этих функций можно исключить из анализа. При правильном распределении функциональности кода формируются два типа функций:

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

Исключив из анализа реализацию функций первого типа, можно будет проверять корректность функций второго типа, возможность ошибки в которых значительно более вероятна. Такое решение позволяет абстрагироваться от функций обработки *APDU* и сконцентрироваться на проверке логики выполнения.

Для поддержки предложенной функциональности, расширим существующий способ *XML*-описания модели опциональными тегами и атрибутами. Тег `globalvars` описывает множество глобальных переменных, задействованных в приложении, каждая из которых перечисляется при помощи тега `var`:

```
<!ELEMENT globalvars (var*)>
<!ELEMENT var EMPTY>
   <!ATTLIST var type CDATA #REQUIRED>
   <!ATTLIST var name CDATA #REQUIRED>
   <!ATTLIST var in CDATA #IMPLIED>
   <!ATTLIST var bits CDATA #IMPLIED>
```

Переменные разделяются по смыслу на два типа:

1. Переменные, значения которых будут вычислены приложением.

2. Переменные, значения которых являются входными параметрами апплета, в том числе считываемыми из *APDU*-команды.

Для работы с переменными второго типа используется функциональность транслятора *jMoped*. Интервал значений задается при помощи указания числа бит, в которых будут храниться эти значения. К тегу `outputIndent` добавим возможность задавать пред- и постусловия при помощи тегов `precond` и `postcond`:

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

Все нововведенные теги являются опциональными. Разработчик может расширить *XML*-описание модели требованиями к переменным на момент вызова функции объекта управления (предусловия) и требованиями к переменным на момент завершения работы функции (постусловия). Соответствующие проверки будут добавлены в код заглушек функций на этапе генерации – это позволит автоматически проверять реализацию заглушек.

Генерация кода и проверка реализации заглушек

Полученный при помощи генератора код является правильным *Java Card*-апплетом, компилируемым, но, с ограниченной функциональностью. Поэтому, во многих случаях необходимо вручную добавлять код, который выполняет специфические задачи.

Сформулируем свойства реализации, которые хотелось бы проверить и которые позволят проанализировать соответствует ли реализация спроектированной модели:

1. Выполнимость переходов с учетом добавленного кода.
2. Достижимость состояний из начального состояния.
3. Проверка выполнения утверждений в функциях объектов управления.

Следует учитывать, что число путей исполнения апплета растет экспоненциально от числа состояний. Поэтому для сложных систем не все проверки будут выполнимы за разумное время. Однако *Java Card*-апплеты обычно не обладают сложной структурой, и в большинстве случаев проверки будут выполнимы, что позволит выявить потенциальные ошибки реализации.

Пример разработки апплета

Апробируем предложенный в работе подход при разработке небольшого *Java Card*-приложения. Создадим апплет, который превращает смарт-карту в электронную версию кошелька. Существующим решением такой задачи является апплет *Wallet*, код которого включается в набор разработчика (*Java Card Development Kit*) в качестве примера, начиная с версии 2.1.1.

Разработаем аналогичный апплет, используя предложенный подход. Перечислим запросы, которые должен корректно обрабатывать апплет:

- положить деньги на счет;
- снять деньги со счета;

- вернуть текущий баланс;
- проверить *PIN*-код.

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

1. Получение данных от пользователя.
2. Проверка валидности полученного запроса.
3. Вызов методов только из допустимых состояний.

Первым этапом разработки является построение автоматной модели. Схема переходов автомата *A1*, который описывает логику приложения, изображена на рис. 3.

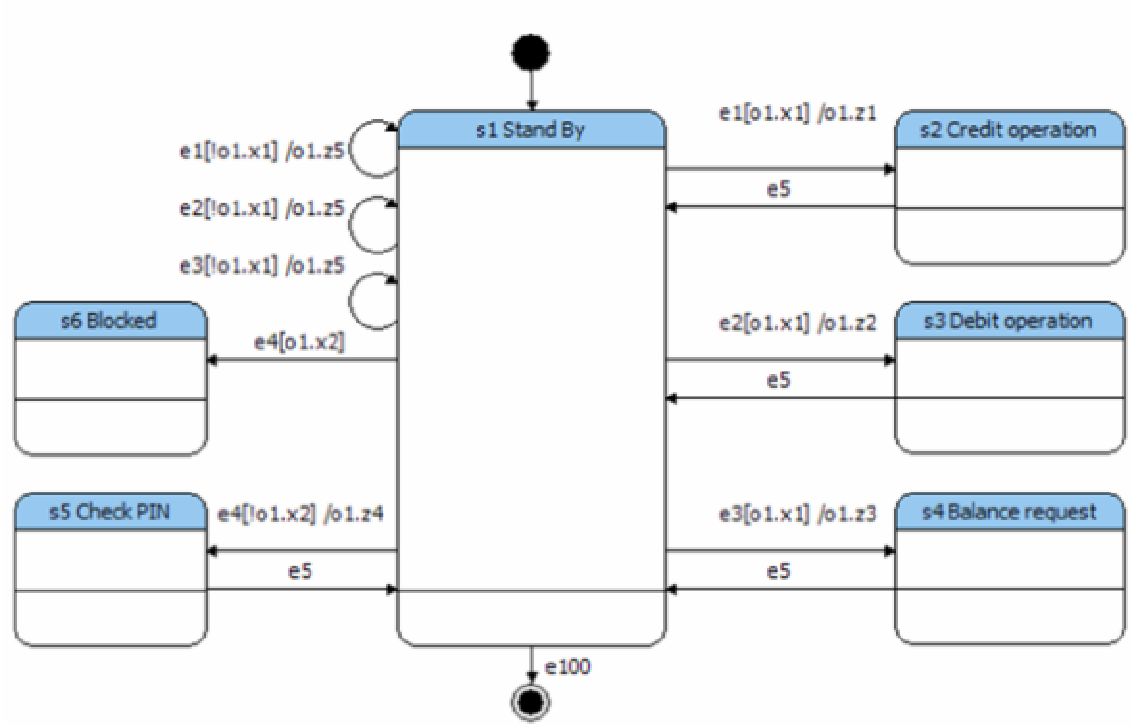

Рис. 3. Граф переходов автомата *A1*

Приведем основные элементы *XML*-представления описанного конечного автомата, которое генерируется инструментом *UniMod*:

```
<stateMachine name="A1">
  <state name="s1" type="INITIAL"/>
  ...
  <state name="s5 Check PIN" type="NORMAL"/>
       <transition event="e5" sourceRef="s2 Credit
 operation"      targetRef="s1 Stand By"/>
       ...
       <transition event="e5" sourceRef="s5 Check PIN"
 targetRef="s1 Stand By"/>
  </stateMachine>
```

Построение автоматной модели требуемого апплета позволило решить целый ряд поставленных задач:

1. Вызов функции объектов управления производится только из тех состояний, в которых это явно указано и только при получении соответствующих событий и выполнении необходимых условий.

2. Апплет реагирует только на явно перечисленные команды – получение неизвестной команды будет проигнорировано.
3. Перед выполнением любой операции со счетом проверяется авторизован ли пользователь.

Нерешенными остались задачи, связанные с численными ограничениями: размер транзакции и число попыток авторизации. Следует отметить, что задача проверки числа попыток при авторизации может быть решена при помощи автоматного подхода: путем проектирования небольшого автомата и создания нового поставщика событий. Однако усилия, потраченные на введение этой проверки в автоматную модель, и снижение понятности модели делают это решение невыгодным.

Предложенный подход позволяет реализовать другое решение. Проанализируем требования, накладываемые на функции объекта управления, и добавим необходимые проверки в *XML*-описание автомата, учитывающие особенности работы электронного кошелька. Для начала, выделим глобальные переменные, которые будут участвовать в логике приложения:

- `short balance` – сумма денег в кошельке;
- `short transaction` – размер запрашиваемой транзакции.
- `byte pin_counter` – допустимое число попыток авторизации.

Добавим перечисленные переменные в *XML*-описание:

```
<globalvars>
 <var type="short" name="balance" />
<var in=”true” bits=”8” type="short" name="transaction" />
      <var type="byte" name="pin_counter" />
      <var type="short" name="MAX_TRANSACTION_AMOUNT" />
      <var type="short" name="MAX_BALANCE" />
      <var type="short" name="PIN_TRY_LIMIT" />
</globalvars>
```

Рассмотрим ограничения, сформулированные в *Wallet*-апплете из набора разработчика. Максимальное значение баланса ограничено снизу нулем, сверху константой `MAX_BALANCE` . Аналогично ограничен размер одной операции над счетом: снизу нулем, а сверху константой `MAX_TRANSACTION_AMOUNT`. Также накладываются ограничения на авторизацию: `PIN_TRY_LIMIT` – максимальное число попыток авторизации. Переменные `balance` и `transition` задействованы в функциях `o1.z1` (*credit*) и `o1.z2` (*debit*). Можно дописать необходимые условия на значения этих переменных к вызовам функции объектов управления:

```
<outputActionCond ident="o1.z1">
     <precond value="transaction &lt;
     MAX_TRANSACTION_AMOUNT"/>
     <precond value="transaction &gt; 0"/>
     <postcond value='balance &lt; MAX_BALANCE'/>
</outputActionCond>
<outputActionCond ident="o1.z2">
     <precond value="transaction &lt;
     MAX_TRANSACTION_AMOUNT"/>
     <precond value="transaction &gt; 0"/>
     <postcond value="balance &gt; 0"/>
</outputActionCond>
```

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

В результате сгенерированные заглушки функций объектов управления имеют следующий вид:

```
private void o1_z2() {
  /* preconditions */
  assert (transaction < MAX_TRANSACTION_AMOUNT);
  assert (transaction > 0);
   /* stub implementation */
  /* precondition */
  assert (balance > 0);
}
```

Для иллюстрации возможностей предложенного метода проверки реализуем любой из методов заведомо неверным способом и посмотрим, будет ли обнаружена эта ошибка. Пусть метод `o1.z1` содержит цикл с ошибочным условием:

```
private void o1_z1() {
 byte b = 2;
 while (b <= 7) { //b will never be > 7
      b = b+1;
      b = b-1;
    }
}
```

Предложенная реализация метода нормально компилируется. В случае проверки выполнимости переходов будет установлено, что состояние *s2* недостижимо, так как переход невыполним. Следовательно, приложение никогда не будет выполняться корректно и эта реализация явно не соответствует спецификации. Запустив проверку, можно выявить несоблюдение ограничений на значение переменной. Если верификатор обнаруживает, что нарушено какое-либо утверждение, то выводятся значения аргументов, которые привели в это состояние:

```
“error” is reachable!
Trace:
s1_Stand_By
e1
transaction s1_Stand_by_to_s2_Credit_Operation
```

Результат показывает, что не выполнено следующее утверждение в функции `o1.z1`:

```
assert (transaction < MAX_TRANSACTION_AMOUNT);
```

Предложенный подход позволяет выявить потенциальные ошибки в коде:

- ошибки в условиях циклов в объектах управления, которые приводят к зацикливанию приложения;

- отсутствие явных проверок значений – при реализации не соблюдены ограничения, заданные в спецификации;
- недостижимость состояния, так как условие 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 – нарушены постусловия функции.

Разработка описанного примера на практике подтвердила преимущества использования автоматного подхода при проектировании приложений для *Java Card*.

Заключение

Перечислим результаты, полученные в данной работе:

1. Предложен метод разработки *Java Card*-апплетов с применением автоматного подхода. Сформулирована связь между понятиями, используемыми в *Java Card* и в автоматном подходе. Показаны преимущества выбора именно этого подхода.
2. Предложен расширенный вариант *XML*-представления автоматной модели, позволяющий на этапе проектирования задавать условия на реализацию методов объектов управления.
3. Написан генератор кода с заглушками для *Java Card*-апплетов из предложенного *XML*-представления, учитывающий особенности платформы *Java Card* и генерирующий утверждения, соответствующие условиям, добавляемым в модель.
4. Расширена функциональность транслятора *jMoped*. Учтены особенности платформы *Java Card* и использованы преимущества того, что в основе кода лежит автоматная модель.
5. Предложены методы проверки полученной модели, позволяющие находить ошибки в реализации объектов управления и выявлять несоответствия полученного *Java Card*-апплета и спроектированной автоматной модели.
6. Создан инструмент, автоматизирующий предложенный процесс тестирования и конвертирующий контрпримеры верификатора в удобный формат, содержащий элементы автоматной модели.

Литература

1. Технология Java Card. Официальный сайт. 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. Чарнецки К., Айзенекер У. Порождающее программирование: методы, инструменты, применение // СПб.: Питер, 2005.
6. Вельдер С. Э., Шалыто А. А. Введение в верификацию автоматных программ на основе метода Model checking. http://is.ifmo.ru/download/modelchecking.pdf
7. Гуров В. С., Мазин М. А. Веб-сайт проекта UniMod. http://unimod.sourceforge.net
8. Технология «клиент-сервер». Wikipedia. The free encyclopedia. http://en.wikipedia.org/wiki/Master-slave_%28computers%29
9. Шалыто А. А., Красс А. С. Отчет о патентных исследованиях по теме разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. http://is.ifmo.ru/verification/_2007_01_patent-verification.pdf
10. Havelund K., Pressburger T. Model Checking Java Programs using Java PathFinder. 1998. www.havelund.com/Publications/
11. Среда тестирования jMoped translator. www7.in.tum.de/tools/jmoped/
12. APDU. Wikipedia. The free encyclopedia. http://en.wikipedia.org/wiki/Protocol_data_unit