МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

Факультет «Информационные технологии и программирование»

Кафедра «Компьютерные технологии»

С. И. Гиндин

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к бакалаврской работе на тему:

«Верификация автоматной модели мобильного банковского приложения»

Научный руководитель – А. А. Шалыто

Санкт-Петербург
2009 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Верификация программ – фундаментальная область исследований в науке «computer science».

В не очень ответственных системах верификация не всегда оправдана, проще исправлять ошибки по мере их обнаружения во время работы системы. Однако существуют такие системы, в которых ошибки нельзя допускать или они могут обойтись слишком дорого. Например, системы управления транспортом (самолеты, поезда), медицинское оборудование, военные программы, финансовые программы и многие другие области, ошибки в которых могут привести к гибели людей или слишком большим убыткам. Тогда верификация необходима, но обычно делается некоторое число допущений, чтобы задача стала решаемой.

Основным методом проверки программы на наличие ошибок является тестирование. На практике оно применяется в большинстве случаев. Однако «тестирование позволяет показать наличие ошибок, но не их отсутствие»[1]. При таком подходе к проверке можно удостовериться в правильности работы программы только при определенном ее поведении или на каком-то конечном числе входных данных. Однако существуют ошибки, которые могут появляться крайне редко, особенно при параллельном исполнении программы. Поэтому, для того чтобы исключить возможность их появления, требуется рассмотреть все возможные варианты поведения системы – провести формальную верификацию.

Формальная верификация – метод проверки того, что программа соответствует спецификации, который основан на строгих математических принципах. При таком подходе используется логический формализм и математическое описание модели. Автоматическая верификация – это формальная верификация, проводимая машиной без вмешательства человека.

[1] Э. Дейкстра, 1970г.

Далее в работе под словом «верификация» будет пониматься автоматическая верификация.

Процесс верификации, как правило, делится на три части:

- моделирование программы – преобразование программы в формальную модель для последующей верификации;
- моделирование спецификации – формальная запись свойств, которые требуется проверить;
- собственно верификация. Все эти части связаны между собой – алгоритмы верификации зависят от способа построения модели и способа записи свойств системы.

В данной работе рассматривается верификация автоматных моделей реактивных программ на языке *MBML*. Прежде всего, ответим на вопрос: что такое реактивная программа? Стандартный шаблон проектирования существующего программного обеспечения таков: программа принимает на вход некоторые данные, выполняет необходимые вычисления и выдает результат. Таким образом, программу можно рассматривать как некую абстрактную функцию из пространства входных данных в пространство выходных. Поведение этой функции состоит в трансформации данных из исходного в конечное состояние.

В противоположность этому реактивная программа не предполагает обязательного завершения. Как подсказывает название, такая система постоянно «реагирует» на окружающие входные воздействия, отвечая на каждое подходящим образом. Примерами таких систем могут служить операционная система, планировщик задач, контроллер и прочее.

Часто реактивные системы являются одновременно сложными распределенными приложениями, поэтому их параллельность также необходимо учитывать. В данной работе не уделяется этому отдельного внимания, в предположении, что исполнение программы в распределенной

системе может быть представимо в виде последовательности взаимно чередующихся действий.

Язык *MBML* позволяет создавать класс программ, реализующих интерфейс меню пользователя с поведением, близким к поведению конечных автоматов. Выбранный математический объект – конечный автомат является достаточно простым вычислителем, а, следовательно, не так сложен при анализе. Есть основания полагать, что грамотным построением конечного автомата можно достаточно хорошо приблизить реальное поведение системы. Новизной данной работы можно считать идею применения автоматов при верификации подобных программ.

ГЛАВА 1. АВТОМАТНОЕ ПРОГРАММИРОВАНИЕ

В данной главе приведен обзор базовых знаний об автоматном программировании и верификации автоматных программ. Более подробные сведения можно найти в работах [1, 2]

1.1. Базовые понятия об автоматном программировании

Автоматное программирование – это парадигма программирования, при использовании которой программа или ее фрагмент осмысливается как система автоматизированных объектов управления. Особенность такого программирования заключается в явном выделении состояний и переходов между ними. По этой причине автоматное программирование также часто называют «программированием с использованием состояний». Процесс исполнения программы заключается в последовательных переходах между состояниями и выполнении определенных действий (зависящей от состояния или перехода). Такая технология программирования была предложена в работе [1] и является удобной при написании определенного класса программ и их последующей верификации.

Данный подход отличается в лучшую сторону от традиционного способа программирования тем, что позволяет явно представить каждое из множества состояний системы. Это позволяет лучше анализировать работу программ, вносить в них изменения, осуществлять отладку и поиск ошибок. Это обусловлено тем, что человеку свойственно мыслить в рамках автоматной модели, где, например, вложенность автоматов представляет собой разбиение логики системы на уровни. Наличие явно выраженных состояний в программах позволяет упростить процесс их верификации.

При таком подходе к созданию программ выделяются три типа объектов: поставщики событий, система управления и объекты управления.

Система управления представляет собой конечный автомат или систему взаимодействующих автоматов. Автомат – это множество состояний и

переходов между ними. Каждый переход характеризуется парой состояний, между которыми осуществляется переход, событием, при котором он может осуществиться, и условием, выполнимость которого требуется для перехода. Поставщики событий генерируют события, а система управления по каждому событию может совершать переход, запрашивая определенные свойства у объектов управления для проверки условий перехода. Заметим, что такое описание очень удачно подходит для реактивных систем, так как в нем естественным образом выделены их компоненты.

Большим достоинством таких программ является также то, что они могут достаточно просто верифицироваться, так как для них не требуется строить другие модели с конечным числом состояний. При верификации программ на языках типа *Java* или *C#*, написанных традиционным путем (без явного выделения состояний), потребовалось бы преобразовать программу к виду, понятному существующим верификаторам. При этом были бы утеряны определенные данные и связи в программе, так как пришлось бы перейти на другой уровень абстракции.

Определенный класс языков, использующихся для описания реактивных систем, в силу своего предназначения близок по своей структуре к конструкциям автоматного подхода. Это позволяет стоить автоматные модели программ на таких языках, которые могут верифицироваться, без существенных изменений или с изменениями, которые не приводят к потере важных данных по сравнению с исходной программой.

1.2. Принципы моделирования

Моделирование является одним из способов изучения объектов, процессов или явлений. Целью изучения, как правило, является получение объяснения этих явлений и/или предсказания дальнейшего поведения рассматриваемых объектов, процессов, явлений. Мотивирующими факторами к

использованию моделирования в качестве способа познания, как правило, являются:

- сложность анализа подлинного объекта изучения;
- дороговизна или невозможность проведения эксперимента на подлинном объекте;
- стремление получить предсказание поведения объекта в будущем.

Моделирование позволяет исключать из рассмотрения несущественные свойства объектов, а, следовательно, существенно упростить анализ. Более того, эксперимент с моделью, а не с настоящим объектом, позволяет существенно сократить затраты.

Для верификации модели, в первую очередь, необходимо определиться, что такое модель. Это может быть представление о программе в определенный момент или значение переменных, состояние взаимодействия с внешними системами и другое. Также при верификации обычной программы, возникает проблема уровня разбиения: строка кода или блок вызова, например.

В настоящей работе рассматривается модель в виде конечного автомата. Далее рассматривается теоретическая часть традиционной верификации автоматной модели, а о практических сторонах ее применения пойдет речь в следующих главах.

1.3. Традиционная верификация на основе метода Model Checking

Традиционный метод верификации автоматных моделей получил название *Model Checking*. В рамках указанного подхода модель программы представляется как множество состояний, причем система в любой момент времени может быть только в одном состоянии. Семантика состояния – глобальный снимок всей системы или мгновенное описание системы. Каждое состояние имеет конечное описание. Имеется выделенное начальное состояние. Применительно к рассматриваемым в данной работе реактивным системам это означает, что система постоянно переходит из состояния в состояние, и в

каждый момент времени может выполняться только один переход. Такой переход означает изменение глобального состояния системы.

Для реагирующих систем характерно бесконечное выполнение – они работают бесконечно долго. Вычисление такой системы – бесконечная последовательность состояний, где каждое следующее состояние получается некоторым переходом из предыдущего. Таким образом, система в общем случае формально может быть представлена в виде графа переходов (S, T, s_0, L, F), который называется моделью Крипке [3]:

- S – конечное множество состояний;
- $T \subseteq S \times S$ – множество переходов;
- s_0 – начальное состояние;
- $L: S \to 2^{AP}$, где AP – множество атомарных высказываний;
- $F \subseteq S$ – множество допускающих состояний.

Тогда путь в этом графе $\pi = s_0, s_1, s_2, \ldots, s_n, \ldots$, для которого выполнено $T(s_{i-1}, s_i)$, будет последовательностью вычислений системы. Путь будет допускающим, если существует состояние из множества F, такое что оно встречается бесконечно часто.

Для модели Крипке существует эквивалентная ей модель – автомат Бюхи. Формально он определяется (S, T, s_0, E, F) следующим образом:

- S – конечное множество состояний;
- E – конечное множество меток переходов;
- $T \subseteq S \times E \times S$ – множество переходов;
- s_0 – начальное состояние;
- $F \subseteq S$ – множество допускающих состояний.

Путь в автомате Бюхи определяется так же, как и в модели Крипке, только переход осуществляется в случае выполнения $T(s_{i-1}, e, s_i)$, где e – метка перехода. Для преобразования модели Крипке (S, T, s_0, L, F) в автомат Бюхи

(S, T, s_0, E, F) достаточно использовать в качества элементов множества меток E множество атомарных высказываний ($E = 2^{AP}$), и добавить другое начальное состояние с переходом в начальное состояние из модели Крипке. После этого переход $T(s_{i-1}, e, s_i)$, выполняется в том и только в том случае, когда состояние s_i в модели Крипке помечено символом e. При этом $e = L(s_i)$.

При представлении модели в качестве автомата Бюхи первое, что можно проверить, это достижимость «хорошего» состояния или недостижимость «плохого». Состояние достижимо, если существует путь из начального состояния в него. Система переходов автомата описывает все состояния и переходы системы. Поэтому можно формулировать условие корректности, как достижимость состояния, а недостижимость – как некорректность. Анализ достижимости проверяется классическими методами обхода в глубину или обхода в ширину.

Анализ достижимости – это простое условие для проверки. На практике также требуется проверять более сложные условия, такие как, например, «устройство готово к работе бесконечно часто», «после отправки события, оно будет получено» или «каждый процесс, который хочет войти в критическую секцию, рано или поздно войдет в нее». Для высказываний такого типа требуется более богатый формализм [4].

Такой формализмом, описывающий последовательность переходов между состояниями реагирующей системы, предоставляют *темпоральные логики*. Они являются языком, на котором удобно формулировать утверждения, использующие понятие времени. Темпоральные логики позволяют формализовать высказывания типа «состояние рано или поздно будет достигнуто», «после состояния s1 автомат перейдет в состояние s2» или «автомат перейдет в состояние s3 бесконечно много раз». Для записи такого вида утверждений кроме обычных булевых операторов используются специальные темпоральные операторы и пропозициональные переменные.

1.4. Линейная темпоральная логика

Логика линейного времени *LTL* (*Linear Time Logic*), является подмножеством более выразительной логики *CTL* (расширение CTL – Computation Tree Logic)*, о которой можно прочесть в работах [3, 4]. Синтаксис логики линейного времени включает в себя множество пропозициональных переменных `Prop` или `P`. $P = \{p, q, p_1, ...\}$, которые могут принимать значения «правда» и «ложь» (им соответствуют константы `T` *è* $\bot = \neg$`T`), булевы связки $(\neg, \wedge, \vee)$ и темпоральные операторы.

Логика линейного времени расширяет классическую логику, добавляя временные операторы, для того чтобы можно было судить о разных моментах времени. В этой логике время линейно и изоморфно натуральным числам.

Для составления утверждений о времени событий применяются следующие темпоральные операторы:

- **X** (ne**X**t) : **X**p – в следующий момент выполнено p;
- **F** (**F**inally) : Fp – в некоторый момент в будущем будет выполнено p;
- **G** (**G**lobally) : Gp – всегда в будущем выполняется p;
- **U** (**U**ntil) : pUq – существует состояние, в котором выполнено q и до него во всех предыдущих выполняется p;
- **R** (**R**elease) : pRq – либо во всех состояниях выполняется q, либо существует состояние, в котором выполняется p, а во всех предыдущих выполнено q.

  Множество *LTL*-формул таково:

  - пропозициональные переменные Prop;
  - True, False (`T` *è* $\bot$)
  - φ и ψ – формулы, то
    - $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$ – формулы;
    - **X**φ, **F**φ, **G**φ, φ**U**ψ, φ**R**ψ – формулы.

Отличие логики *LTL* от логики *CTL** состоит в том, что в последней присутствуют кванторы пути $\forall$ и $\exists$, которые говорят о любом пути и о существовании пути соответственно. Логика линейного времени говорит о всех путях, и квантор $\forall$ опущен. Таким образом, логика *LTL* предполагает, что некоторое утверждение будет выполняться для всех путей. Поэтому можно строить доказательство от противного и проверять существование пути, на котором будет выполняться отрицание данной формулы. Если такой путь не будет найден, то формула выполнима.

В работе [6] предлагается алгоритм, который обеспечивает трансляцию *LTL*-формулы в так называемый *автомат Бюхи*, с помощью которого можно получить доказательство невыполнимости некоторой формулы логики линейного времени построением контрпримера – пути, на котором не выполняется *LTL*-формула. Существуют различные модификации данного метода, позволяющие решать некоторые задачи более эффективно. Подробнее использование автомата Бюхи и его модификаций описано в работах [3, 9], в которых в, частности, приводится ряд преобразований, которые позволяют получить автомат Бюхи меньшего размера.

1.5. Актуальность задач верификации в финансовой сфере

Разнообразие задач, с которыми сталкивается предприятие, рассмотрим на примере процессинга банковских карт (рис. 1).

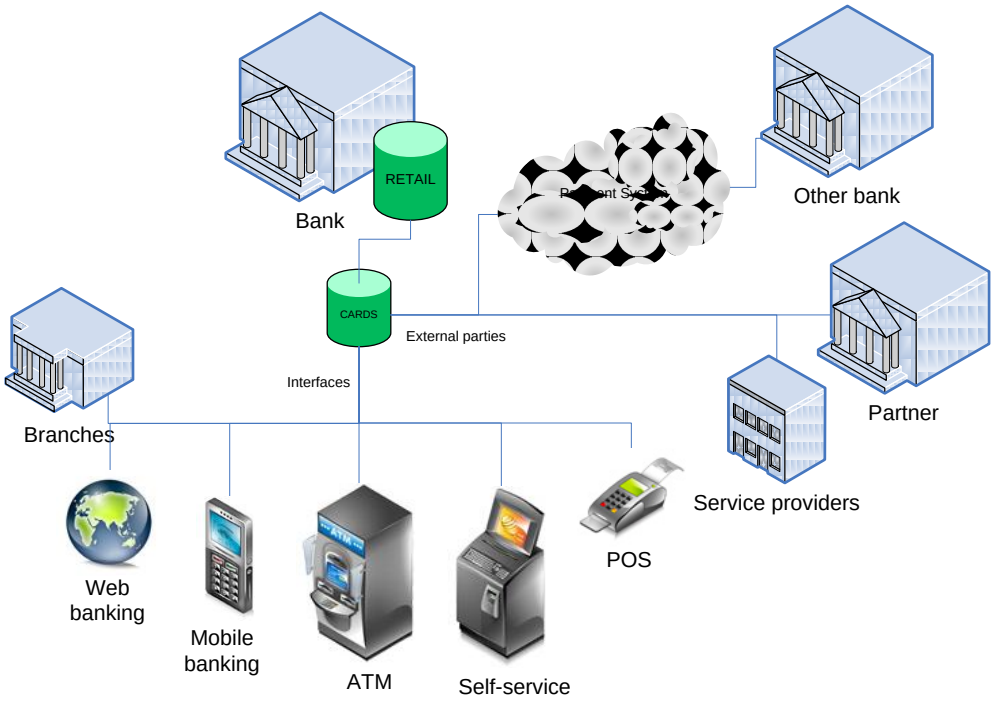

Рис. 1

Обычно банк имеет множество систем, в каждой из которых ведется учет по одному из основных направлений бизнеса. Например, система, в которой ведется учет карт и операций, по ним проводимых, обычно выделяется отдельно от основной банковской системы (называемой *RBS, Retail Bank System*), в которой ведется основной учет. Таким образом, для совокупного учета все операции, проведенные в карточной системе, должны получить отражение в *RBS* – это означает, что карточная система должна тем или иным образом быть интегрирована с *RBS*.

Основа успешного карточного бизнеса – наличие достаточного числа интерфейсов для пользователей карт. Примеры таких интерфейсов – это Интернет-банк (*web banking*), мобильный банк (*mobile banking*), банкоматы (*ATM, Automated Teller Machine*), терминалы самообслуживания (*self service*), торговые терминалы (*POS, Point Of Sale*), интерактивная система самообслуживания по телефону (*IVR, Interactive Voice Response*), автоматизированное место оператора (*APW, Automated Working Place*).

POS-терминалы распространены более всего из-за невысокой стоимости, простоты установки и обслуживания. Очень популярны в сфере розничной

торговли. Они позволяют проводить безналичные покупки по карте прямо в магазине.

Банкоматы – это, пожалуй, самый зрелый интерфейс, появившийся вместе с зарождением карточного бизнеса. Банкоматы используются в основном для выдачи наличных и иногда для оплаты услуг, например, квартплаты.

В последние годы появляется тенденция выделения устройств самообслуживания в отдельный интерфейс. Их отличие в том, что устройства самообслуживания, иначе, информационные киоски, предназначены в основном для проведения безналичных операций и предоставления информационных услуг, они не выдают наличные. Однако они могут быть также оборудованы устройством для приема наличных.

Интернет-банк по функциональности довольно близок к интерфейсу устройств самообслуживания. Поэтому популярность Интернет-банка резко растет. В последние годы с увеличением числа пользователей глобальной сети, увеличивается и разнообразие предоставляемых услуг. В Интернет-банке пользователь, аутентифицированный тем или иным образом, получает доступ к управлению своими картами и счетами, а также имеет возможность совершать платежи и переводы, просматривать историю операций и подсчитывать свою бухгалтерию удаленно, не обращаясь для этого в банк.

Отметим также интерфейсы интерактивной голосовой авторизации и автоматизированного рабочего места оператора. В обоих случаях они используются сотрудниками банка, которые могут выполнять некоторые операции в интерактивном режиме при обращении клиента по телефону (интерактивная голосовая авторизация) или в его присутствии в отделении банка (автоматизированное место оператора).

Мобильный банк – интерфейс, получающий все большую популярность и постепенно оттесняющий более привычные каналы обслуживания, благодаря распространению мобильной телефонии. Здесь число услуг ограничено

возможностями мобильного телефона как устройства, но присутствуют наиболее популярные: всевозможные платежи, контроль состояния счета и просмотр выписки. Наиболее популярен вследствие своей независимости от модели телефона *sms*-интерфейс, при котором пользователь получает необходимую информацию с помощью обмена *sms*-сообщениями с банковской системой для запроса баланса, блокировки карты и т. д., и мгновенных уведомлений об операциях по карте. С развитием и стандартизацией возможностей, предоставляемых современными мобильными телефонами, появилась потребность в приложениях для этой платформы, которые аналогичны по удобству и функциональности другим каналам обслуживания, для которых вся работа проходит «в одном окне», без необходимости постоянных переключений, набора сервисных команд вручную и вынужденного приспосабливания архаичных элементов неразвитого интерфейса. Рынок мидлетов (приложений для мобильных телефонов) очень быстро вырос и по праву стал одним из самых развивающихся в информационном пространстве.

Выводы по первой главе

Автоматное программирование представляет большой интерес, в том числе из-за возможности автоматической верификации автоматных программ на основе метода *Model Checking*. Данный подход имеет хорошие перспективы применимости в финансовой сфере, одной из основных особенностей которой является сильная распределенность и необходимость интеграции между интерфейсами.

ГЛАВА 2. ЯЗЫК ОПИСАНИЯ КОНФИГУРАЦИЙ МЕНЮ *MBML*

Растущая популярность мобильных устройств и технологий отразилась в появлении специального языка *MBML (Mobile Banking Menu Language*), ориентированного на использование в мобильных банковских приложениях, работающих в режиме реального времени.

2.1. Краткая история развития языка

Язык *MBML (Mobile Banking Menu Language)* был разработан компанией *OpenWay* в 2007 году и начал использоваться для решения задач предоставления пользователем мобильного интерфейса к банковским услугам с начала 2008 года. В настоящее время с использованием языка *MBML* созданы типовые решения программы-клиента для доступа к сервисам мобильного банка, а также большое число кастомных решений, ориентированных на конкретных заказчиков.

При разработке языка учитывались следующие принципы:

- язык должен соответствовать предметной области, для которой он предназначен, и архитектуре (интерпретация мидлетом), в которой он будет использоваться;
- язык должен быть как можно более простым;
- язык должен быть расширяемым (допускать добавление новых шагов)
- меню описывается в отдельном конфигурационном файле, который после предварительного преобразования средствами серверного приложения загружается в мидлет, установленный в мобильный телефон.

2.2. Обзор языка

В основе *MBML* лежит близкая к автоматному программированию мысль о том, что пользовательское меню – это последовательность шагов, связанных друг с другом переходами в зависимости от текущего состояния. Структура программы пользовательского меню представляет собой совокупность шагов, в которых происходит отображение объектов на экране телефона или выполнение различных служебных действий, представленных объектами различных классов, а также выбор следующего шага.

Программа на языке *MBML* называется файлом меню или файлом конфигурации.

Каждая строка файла меню является шагом – описанием объекта определенного класса и его параметров. Подробное описание основных классов языка приведено в Приложение 1. **Основные конструкции языка *MBML***

Символ ";" (точка с запятой) используется в качестве разделителя в строке.

Строки указываются в следующем формате:

```
<ID объекта>;Class=<класс объекта>;[<параметр>=<значение>;]
```

Здесь и далее квадратные скобки указывают на то, что в строке может быть задано произвольное число данных элементов. При этом порядок, в котором указываются эти элементы, не имеет значения.

ID-объекта является произвольно определяемым уникальным строковым значением за исключением ID-объекта для первой строки, значение которого обязательно должно быть указано как `main`.

Регистр в значениях ID-объектов, наименованиях классов и наименованиях параметров не учитывается.

Символ ";" (точка с запятой), указанный в начале строки, означает, что данная строка является закомментированной.

Для всех объектов меню могут использоваться два предопределенных параметра:

- *Exit=<ID объекта>* – определяет ID-объекта (шаг), на который будет передано управление в случае, если возникла ошибка, или было выбрано действие <*Back*> или <*Cancel*> (вызов действия зависит от модели телефона и может быть реализован как путем выбора пункта в графическом меню, так и путем нажатия на соответствующую кнопку на клавиатуре).
- *Next=<ID объекта>* – определяет ID-объекта (шаг), на который будет передано управление при выборе действия типа <*OK*>, <*Next*> и т.п.

2.3. Формальное описание языка

С формальной точки зрения, язык *MBML* описывается следующим набором построений.

Определение 2.1. Переменной называется пара $v = (val, D)$**,** где $D = Type(v)$ – множество значений данной переменной, $val \in D$ – ее текущее значение.

Определение 2.2. Шагом называется атомарная инструкция-объект языка, тройка $step = (Name, Class, F)$**,** где `Name` – уникальное имя объекта, `Class` – класс объекта, `F` – набор значений полей объекта.

Вычислительный процесс определяется тройкой `{p, C, Ch}`, где `p` – программа, `C` – контекст, совокупность данных программы, представляющую собой набор переменных и данных, получаемых от пользователя, `Ch` – набор доступных каналов связи.

Программа `p` – это выражение на языке *MBML*. Множество синтаксически правильных программ на *MBML* будем обозначать `P`. Абстрактный синтаксис *MBML* задается следующей грамматикой [5]:

$$\text{Pr}\,ogram \rightarrow StartStep \quad (Step)^*$$
$$StartStep \rightarrow Step$$
$$Step \rightarrow Switch \mid Quit \mid GeneralStep$$
$$Switch \rightarrow (Condition \quad Step)^+$$
$$Quit \rightarrow \square$$
$$GeneralStep \rightarrow Input \quad Action^* \mid Action^*$$

Программа *(Program)* состоит из начального шага *(StartStep)* и последовательности шагов *(Step)*. При этом подразумевается, что каждый шаг имеет уникальный идентификатор. Шаг может быть одного из нескольких видов. Шаг типа *Выбор (Switch)* состоит из нескольких условий и соответствия им дальнейших шагов. *Выход (Quit)* завершает работу приложения. Остальные *шаги (GeneralStep)* выполняют *действия (Action)* и, возможно, считывают данные *(Input)*.

Condition – это описание функции $C \rightarrow \{0,1\}$, где C – контекст.

Структура шагов в рамках задачи может быть представлена в виде дерева шагов (рис. 2).

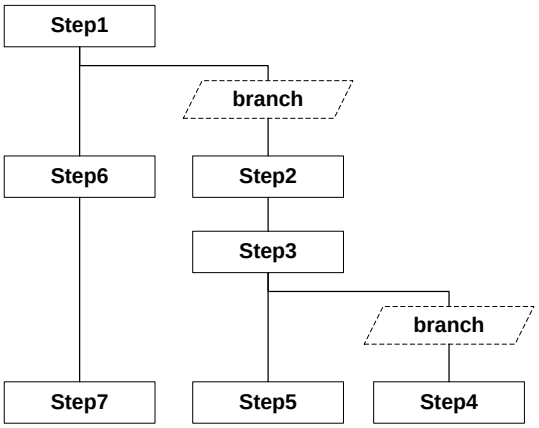

Рис. 2

Определение 2.3. Шаг $s \in p$ называется конечным шагом для программы p, если выполнены следующие условия:

- s является последним шагом в последовательности шагов;

- s для не определено ни одной ветви, содержащей один или более шагов.

Множество конечных шагов программы p обозначается $Fin(p)$.

Из семантики языка следует, что $s \in Fin(p) \Leftrightarrow Class(s) = Quit$ – программа может завершаться только шагом типа *Quit*, поскольку для других шагов определены свойства *Next, Exit*.

Определение 2.4. Пусть τ – некоторый запуск процесса, выполняющийся по программе p. Последовательность (конечная или бесконечная) $trace(\tau) = s_0 s_1 s_2 ...$ называется следом запуска вычислительного процесса τ ($\forall s_i \exists t' \in p : s_i \in t', 0 \le i < |trace(\tau)|$), если вычислительный процесс

- начинается с выполнения шага s_0 ;
- следующим за s_i выполняется s_{i+1}.

Длина следа обозначается $|trace(\tau)|$. В общем случае след не обязан быть конечным:

$$|trace(\tau)| = \begin{cases} n, trace(\tau) = s_0 s_1 ... s_{n-1} \\ \infty, trace(\tau) = s_0 s_1 ... s_{n-1} ... \end{cases}$$

2.4. Постановка задачи

Задачей данной работы является разработка методики построения автоматной модели программ, написанных на языке *MBML*, который описан ранее в данной главе, и ее верификация в соответствии с соображениями, изложенными в первой главе.

Выводы по второй главе

Язык *MBML* является довольно удобным (в силу предметной ориентированности), что позволяет рассматривать вопрос об автоматическом построении автоматных моделей для программ на нем.

ГЛАВА 3. ОПИСАНИЕ МОДЕЛИ И ВЕРИФИКАЦИЯ

В первой главе были приведены некоторые сведения из теории автоматного программирования. В этой главе будет описана автоматная модель программы на языке *MBML*.

3.1. Построение модели

При построении модели существенно использовался тот факт, что шаг является основной выразительной единицей языка *MBML*. Каждому шагу программы соответствует состояние автоматной модели. Переходу $Model(s_1) \to Model(s_2)$ между этими состояниями соответствует возможность в исходной программе выполнить подряд шаги $s_1 s_2$. Ввиду специфики (мобильности приложения) указанный способ построения модели обеспечивает естественный изоморфизм между моделью и исходной программой.

3.2. Верифицируемые свойства

Имея формальное описание *MBML*, опишем формально свойства программ, которые будем проверять.

Синтаксическая корректность записи программы `p` гарантирует, что во множестве шагов выделен начальный, и все ссылки на шаги корректны, иными словами, допускаются ссылки только на присутствующие в программе шаги.

Однако синтаксическая корректность не гарантирует, например, существования корректного завершения вычислений по программе. Не гарантируется существование потенциального конечного шага в программе. Не гарантируется также его достижимость. Приводимое ниже определение вводит свойство *корректности завершения.*

Определение 3.1. Программа p называется *корректно завершаемой* если $\exists s \in p$:

- $s \in Fin(p)$;
- $\exists \tau : trace(\tau) = s_0 ... s$.

Наличие недостижимых инструкций и шагов всегда говорит об избыточности модели и обычно свидетельствует об ошибке, допущенной при моделировании. Эти рассуждения справедливы и применительно к моделям на *MBML*. Ключевым понятием здесь является *достижимость*, которая вводится на основании следующих определений.

Определение 3.2. Вычислительный процесса по программе p называется *имитирующим*, если процесс выполняется в соответствии с последовательностью шагов следующим образом:

- вместо шага типа `GeneralStep` используется заглушка, не выполняющая никаких действий;
- вместо обработки ветви шага типа `Switch`, содержащей выполнимое условие, процесс случайно выбирает одну из ветвей либо переходит к выполнению следующего шага в последовательности.

Определение 3.3. Шаг s *реализуем* в программе p, если существует хотя бы один вычислительный процесс τ по программе p, содержащий в своем следе s. $\exists \tau : s \in trace(\tau)$.

Определение 3.4. Шаг s *(формально) достижим* в программе p, если он выполняется при некотором имитирующем вычислительном процессе.

Из реализуемости шага следует его достижимость, но не наоборот. Условие реализуемости является очень сильным и подразумевает, что в процесс верификации должен быть вовлечен анализ условий для ветвей, а, следовательно, значений переменных контекста. В реальности число переменных контекста может быть велико, также как и множество их допустимых значений. Кроме того, значения переменных контекста могут изменяться пользователем. Достижимость далее понимается в терминах

определения 3.4. В случае, когда в программе все шаги достижимы, то модель обладает свойством *отсутствия избыточности*.

Также важна *корректность обращения к переменным*. Под этим понимается допустимость обращения к их значениям в вычислительном процессе. Жизненный цикл любой переменной бизнес-процесса может быть описан простым конечным автоматом, изображенным на рис. 3.

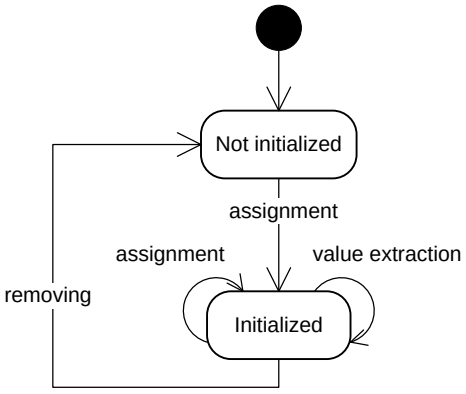

Рис. 3

Переменная контекста может находиться в двух основных состояниях – не инициализированном и инициализированном (после выполнения первого присвоения). Обращение к значению переменной в неинициализированном состоянии может привести к непредсказуемым последствиям.

Определение 3.5. В программе *p обращения к переменным выполняются корректно*, если при любом запуске вычислительного процесса обращение к значению любой переменной выполняется только при условии, что переменная инициализирована.

Проверка корректности обращения к переменным позволяет косвенным образом найти высокоуровневые ошибки в модели. В частности, наличие неинициализированной переменной может свидетельствовать о том, что пропущен вызов одного или более существенных шагов.

Еще одним немаловажным свойством, накладываемым предметной областью, является *соблюдение бизнес-ограничений.* Бизнес-ограничения образуют широкий класс свойств. В то же время эти свойства достаточно

сложно описать формально в общем виде. Природа бизнес-ограничений проистекает из требований, предъявляемых к бизнесу. Например, обязательным условием проведения транзакции перевода денег является проверка *PIN (Personal Identification Number)*. Он может вводиться пользователем сразу после начала работы с приложением и быть сразу же проверенным, или вводиться пользователем каждый раз перед выполнением отдельной операции (платеж, перевод и прочее). В любом случае, требуется выполнение условия: «Перед выполнением операции должен быть проверен *PIN*».

Рассматривая бизнес-ограничения, следует учитывать следующие соображения:

- бизнес-ограничения не являются правилами, которые обязаны всегда выполняться (в отличие от правил корректности операционной модели). Бизнес-ограничения управляются потребностями бизнеса и могут изменяться вместе с ними;
- бизнес-требования могут быть трудно формализуемы в фиксированной модели. Например, если шаг, выполняющий финансовый запрос, не отличим с точки зрения модели от шага, выполняющего запрос нефинансовый, то никакое условие, ссылающееся на факт выполнения финансового запроса, проверено быть не может;
- верификация бизнес-ограничений имеет смысл только тогда, когда вероятность их нарушения велика и требует больших затрат при проверке вручную. Иными словами, задачу необходимо решать, когда она действительно актуальна. Например, если проверка *PIN* всегда осуществляется сразу после запуска приложения, верификация соответствующего свойства приносит мало пользы.

Единицей деятельности в терминах принятой модели является шаг. Поэтому естественно будет сузить множество рассматриваемых бизнес-ограничений в соответствии с определением, приводимом ниже.

Определение 3.6. *Бизнес-ограничениями* являются дополнительные условия, налагаемые на порядок выполнения шагов в рамках вычислительного процесса.

3.3. Методы верификации

Поскольку модель описывается конечным автоматом специального вида, а проверяемые требования представляются в виде формул логического исчисления, позволяющих строить утверждения о поведении автомата в процессе его работы, то для верификации выполнимости формул для данного автомата подходит метод *Model Checking*, описанный в первой главе.

3.4. Инструменты верификации

Как уже было отмечено, для верификации программы при таком подходе на некотором языке достаточно построить для нее автомат *Бюхи*, построить автомат *Бюхи* по отрицанию формулы, реализующей верифицируемое свойство и запустить процедуру поиска принимающих состояний в их пересечении. Большинство инструментов, разработанных для верификации распределенных систем, используют этот метод. Обычно вместе с верификатором предлагается системный язык, программы на котором инструмент отображает в автомат. Также предлагается нотация для записи формул некоторой темпоральной логики, с помощью которых формулируются проверяемые свойства. Для верификации программной системы требуется транслировать код проверяемой системы в код на языке верификатора и сформулировать свойства. Наиболее популярными системами являются верификаторы *SPIN, Bogor, NuSMV и STeP*. Их применение было изучено для верификации автоматных программ в работах [2, 3, 9].

SPIN позволяет верифицировать программы, написанные на высокоуровневом языке *Promela (PROcess MEta LAnguage)*. Требования к программе записываются на языке *LTL*. В работе [2] предложен метод, позволяющий по автоматной программе строить модель на языке *Promela*,

выполнять автоматическое преобразование записанных вручную на языке *Promela* проверяемых требований в автомат *Бюхи*, автоматически выполнять верификацию полученной модели и производить построение контрпримера по модели [7].

Bogor один из сравнительно недавно появившихся инструментов, который, однако, активно используется в практических проектах. Этот верификатор предоставляет возможность проверять модели, написанные на языке *BIR* (*Bogor Input Representation*). Требования к программе также записываются на языке *LTL*. Во входной язык верификатора *Bogor* можно добавлять новые типы и абстракции, а сам верификатор разделен на модули, реализующие различные аспекты верификации (алгоритм обхода, кодирование состояния и т.п.).

SMV и его развитие *NuSMV* реализуют символическую проверку моделей. Символьный верификатор моделей *SMV* (аббревиатура *Symbolic Model Verifier*) – это инструментальное средство, предназначенное для проверки того, что система переходов с конечным числом состояний удовлетворяет спецификации, заданной в логике *CTL*. В нем применяется символьный алгоритм верификации моделей. Для описания сложных систем с конечным числом состояний используется специальный язык *SMV*, поддерживающий модульность, синхронную либо *интерливинговую* (с импользованием чередования) композицию, недетерминированные переходы.

STeP является учебным проектом университета Стэнфорд. Он проверяет программы, записанные на языке *SPL (Simple Program Language)*, и использует *LTL* для формулировки свойств.

Все эти инструменты нацелены на решение одной проблемы и обладают сходной функциональностью. *STeP* имеет ограничения на размер пространства состояний, а *SMV* используется в основном для решения сложных задач синхронизации.

Язык *PROMELA* наиболее удобен для описания процессов и их взаимодействия с помощью обмена сообщениями, что позволяет логично расширить применение описываемого в данной работе метода на систему взаимодействующих агентов. Кроме этого, алгоритмы *SPIN* оптимизированы для работы с большими пространствами состояний, что существенно для программ большого размера.

По этим причинам в качестве верифицирующего инструмента был выбран *SPIN*.

3.5. Верификатор SPIN

Верификатор распределенных систем *SPIN* был разработан Герардом Холцманом (*Gerard Holzmann*) в начале 1990-х годов и был описан им в книге *«Design and validation of Computer protocols»*. Инструмент разрабатывался для проверки корректности сетевых протоколов, но хорошо подходит и для верификации распределенных систем вообще. С точки зрения инструмента, верифицируемая система представляет собой совокупность взаимодействующих процессов. Основными свойствами, которые проверяет инструмент, являются:

- Отсутствие тупиков. Свойство гарантирует, что ситуация, когда несколько взаимодействующих процессов ожидают каких-либо действий друг от друга, невозможна.
- Отсутствие «пустых» циклов. Свойство гарантирует, что процессы защищены от зацикливания.
- Выполнимость ограничений, налагаемых на контекст каждого процесса, выраженных в виде ряда ограничений на значения переменных процесса.
- Выполнимость ограничений, налагаемых на деятельность процессов, выраженных с помощью *LTL*–формул.
- Отсутствие недостижимых инструкций.

Процедура верификации с помощью инструмента *SPIN* представлена на рис. 3.

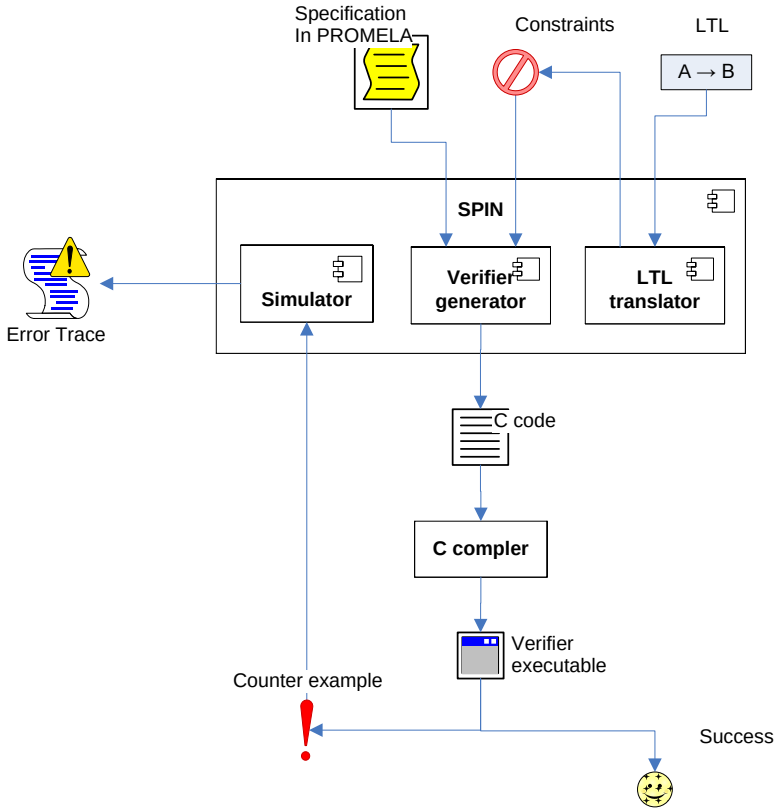

Рис. 3

Спецификация верифицируемой системы поступает на вход инструмента в виде программы на языке *PROMELA* [8]. Вместе с программой на вход инструменту передаются ограничения, также заданные на языке *PROMELA* (ограничения, заданные в виде *LTL*-формул, предварительно транслируются в *PROMELA*). На основании спецификации и заданных ограниченный *SPIN* генерирует код верификатора на языке *C*. Для выполнения верификации необходимо скомпилировать код в выполнимый модуль и запустить его. В случае нарушения какого-либо из ограничений верификатор выдает ошибку, на основании которого инструмент формирует контрпример (след запуска программы, в которой это ограничение нарушается).

Уровень абстракции *PROMELA* делает его годным для описания произвольных взаимодействующих систем. В терминах этого языка любая

система представляется в виде совокупности процессов. Описание процесса задается прототипом процесса (пример 3.1).

Пример 3.1. Описание процесса

```
Proctype A{;
    . . .
}
```

Вычислительный процесс по программе начинается с выполнения `init`-секции, в которой по объявленным прототипам процессы. Поведение процесса моделируется с помощью последовательности инструкций, разделенных «;» и «→», которые являются эквивалентными. Контекст процесса обеспечивается локальными и глобальными переменными. Поддерживаются пять примитивных типов данных (`bit, bool, byte, short, int`) и определяемый пользователем перечислимый тип (`mtype`). Существует возможность определять собственные сложные типы данных. Из структур данных поддерживается массив. Синтаксис объявления переменных проиллюстрирован в примере 3.2.

Пример 3.2. Описание переменных

```
byte b = 0;
mtype = { one, two, three };
typedef Counter {
  mtype word;
  short num;
};
int arr[5];
```

Принципиальным понятием в языке *PROMELA* является выполнимость инструкций. В момент обработки инструкция может быть выполнимой, либо блокированной в зависимости от текущего контекста. Например, инструкция по логическому выражению `(a == 1)` выполнима, если утверждение верно, и блокирована в противном случае. Процесс не может обработать блокированную

инструкцию и вынужден синхронно ждать, пока инструкция не станет выполнимой, либо перейти к проверке альтернативных инструкций, если они есть.

Инструкции делятся на следующие категории:

- элементарные инструкции;
- условные выражения;
- составные инструкции.

К элементарным инструкциям относятся:

- присваивание `(a = 10)` – выполнимо всегда;
- контекстные ограничения `(assert(a == b))` – выполнимы всегда;
- запуск дочернего процесса `(run myProcess())` – выполним всегда;
- инструкция `skip` – эквивалентна булевой «истине» и выполнима всегда.

К условным выражениям относятся:

- булевские условия над значениями переменных `(a == 1)` – выполнимы, если истинны;
- инструкция `timeout` – системная булевская переменная. Она истинна, если никакая другая инструкция не выполняется (пример 3.4).

К составным инструкциям относятся:

- оператор `if` (пример 3.3) – выполним, если выполнима хотя бы одна из альтернатив (следующая за «`::`»);
- оператор `do` (пример 3.4) – выполним, если выполнима хотя бы одна из альтернатив (следующая за «`::`»);
- инструкции `atomic { … }` и `d_step { … }`, позволяющие определять атомарный набор инструкций (выполняемый

одновременно целиком) и блок детерминированного поведения (не содержащий невыполнимых инструкций) соответственно. Инструкции выполнимы, если выполнимы первые инструкции в их блоках.

Пример 3.3. Использование оператора `if`

```
a = 0;
if
::(a==0)-> a=5;
::(a!=0)-> a=a-1;
Fi
```

Пример 3.4. Использование оператора `do`

```
a = 5;
do
::(a!=0)-> a=a-1;
 ::timeout -> break;
od
```

Алгоритмические конструкции обеспечиваются операторами

- `if` – для организации ветвления;
- `do` – для организации циклов;
- `goto` – для безусловных переходов по метке;
- `break` – для выхода из цикла.

Ограничения на значения переменных задаются с помощью оператора `assert` (<булевское выражение>).

Соглашение об именовании меток позволяет указывать конечные состояния процесса (метки с префиксом `end`) и принимающие состояния (метки с префиксом `accept`). Метки, начинающаяся с «`end`», указывают, что процесс, бесконечно долго блокированный на помеченной инструкции, не

находится в тупике. Например, процесс, описанный в примере 3.5, никогда не завершится, при том, что ситуация не будет расценена верификатором как тупик.

Пример 3.5. Использование меток `end`

```
proctype A() {
end:
  (1 == 2);
}
```

Метки с префиксом «`accept`» позволяют указать, что бесконечный цикл, содержащий инструкцию с такой меткой, не является пустым.

LTL-формулы транслируются инструментом в конструкции, называемые *never-claim*. Каждая такая конструкция определяет процесс специального вида, выполняемый в режиме демона (его порождение выполняется автоматически при запуске программы). Условие, определяемое конструкцией *never-claim,* нарушается, если процесс, определенный данной конструкцией, переходит в конечное или принимающее состояние. Пример 3.6 иллюстрирует *never-claim* для *LTL*-формулы $G(p)$.

Пример 3.6. Конструкция *never-claim*

```
never {
start:
  if
  ::(!p) -> goto end;
  ::else -> goto start;
  fi;
end:
  skip;
}
```

Подробное описание языка *PROMELA*, инструмента *SPIN* и используемых им алгоритмов можно найти в работах [6, 8].

3.6. Основные требования к отображению программ языка MBML

Будем строить отображение индуктивно по структуре *MBML*: определим отображения для шагов, ветвей условий и т.д. Сужение отображения *Trans* на составные элементы программы будем обозначать также *Trans*, понимая при этом, что $Trans: P \to P_{PROMELA}$ определяет $Step \to L_{PROMELA}$, где *Step* – множество синтаксически правильных шагов. Обозначим $P_{PROMELA}$ множество корректных программ на языке *PROMELA*, а $L_{PROMELA}$ – множество синтаксически правильных конструкций языка.

В настоящей работе интересуют такие преобразования $P \to P_{PROMELA}$, которые сохраняют верифицируемые свойства. В первую очередь, разумно потребовать, чтобы выполнение транслированной программы естественным образом моделировало выполнение исходной *MBML*-программы.

Определение 3.1. Пусть $Trans: P \to P_{PROMELA}$, $p \in P, p' = Trans(p)$. Будем говорить, что запуск σ вычислительного процесса по программе p' моделирует запуск τ вычислительного процесса по p, для которого $trace(\tau) = s_1 s_2 ... s_i ...$, если $trace(\sigma) = Trans(s_1)Trans(s_2)...Trans(s_i)...$.

Определение 3.2. Пусть $Trans: P \to P_{PROMELA}$. Назовем отображение (преобразование) *Trans* моделирующим ($Trans(p)$ моделирует p, $\forall p \in P$), если для любого запуска имитирующего вычислительного процесса τ по p существует моделирующий его запуск σ по $Trans(p)$, и наоборот, любому запуску σ соответствует некоторый запуск τ, который она моделирует.

Иными словами,

- s – первый шаг программы $p \Leftrightarrow Trans(s)$ достижим и выполняется при любом запуске ранее $Trans(s_1)$, $\forall s_1 \in p, s_1 \neq s$.
- $\forall s_a, s_b \in p : \exists \tau : trace(\tau) = ...s_a s_b ... \Leftrightarrow$

  $\Leftrightarrow \exists$ запуск $\sigma : trace(\sigma) = ...Trans(s_a)Trans(s_b)....$

Таким образом, в случае моделирующего отображения в процессе верификации все нарушения свойств могут быть выявлены (так как моделируются все возможные запуски) и наоборот, возникновение несуществующих нарушений невозможно (так как моделируются только возможные запуски).

Выводы по третьей главе

Выбранное решение позволяет автоматически задать адекватную автоматную модель, естественно соответствующую исходной программе, а также формально описать необходимые свойства верификации.

ГЛАВА 4. РЕАЛИЗАЦИЯ И ИСПЫТАНИЯ

В данной главе речь пойдет о реализации предложенной автоматной модели и о результатах верификации некоторых программ.

4.1. Трансляция языка MBML в язык Promela

Приступим к построению отображения. Вычислительный процесс по программе $p \in P$ естественным образом моделируется процессом. Будем моделировать программу отдельным прототипом процесса.

Пример 4.1. Моделирование программы прототипом процесса

```
proctype program()
{
    ...
}
```

Шаги можно моделировать

- в рамках одного прототипа процесса программы с помощью ветвления по номеру шага в цикле (пример 4.2);
- в рамках одного прототипа процесса программы с помощью меток и безусловных переходов (пример 4.3);
- отдельными прототипами процессов (пример 4.4).

Пример 4.2. Моделирование структуры шагов внутри единого прототипа

```
short step_id;
proctype program()
{
    do
        :: (step_id == 1)-> <описание шага>;
        :: (step_id == 2)-> <описание шага>;
```

```
        :: (step_id == 3)-> <описание шага>;
        . . .
        :: else -> skip;
    od
}
```

В описании шага переменной `step_id` присваивается значение следующего шага, который должен быть исполнен.

Пример 4.3. Моделирование структуры шагов внутри единого прототипа с помощью меток и безусловных переходов

```
proctype program()
{
   Step1: <описание шага> goto <следующий шаг>
   Step2: <описание шага> goto <следующий шаг>
   Step3: <описание шага> goto <следующий шаг>
   . . .
}
```

Пример 4.4. Моделирование структуры шагов различными прототипами

```
proctype step1()
{
    <описание шага>
}

proctype step2()
{
    <описание шага>
}

proctype step3()
{
```

```
    <описание шага>
}
. . .
```

Ключевым вопросом при выборе решения является механизм перехода между шагами. Поскольку в языке предусмотрен только переход в вызываемый шаг без возврата к вызывающему, это накладывает следующие ограничения на организация трансляцию задач:

1. При переходе по типу к следующему шагу не нужно хранить лишние данные о вызывающем шаге.
2. После выполнения шага типа `quit` выполнение завершается.
3. В случае наличия ветвей после шага вычислительный процесс может пойти по любой из них.

С учетом первого требования последний из вариантов моделирования структуры шагов менее удобен, чем остальные. В нем один процесс вызывает другой, тем самым увеличивается глубина стека вызовов, которая в *SPIN* имеет верхнее ограничение 255 [10]. Было решено отказаться от такой реализации структуры шагов.

Необходимо также поддержать шаги класса `quit`. Второе требование может быть реализовано в каждом из двух оставшихся вариантов (примеры 4.5, 4.6)

Пример 4.5. Моделирование шага `quit` для первого решения

```
proctype program()
{
   do
        :: (step_id == 1)-> <описание шага>;
        :: (step_id == 2)-> <описание шага>;
        :: (step_id == <шаг quit>)->
            <описание шага>; break;
```

```
        . . .
        :: else -> skip;
    Od
    skip;
}
```

Пример 4.6. Моделирование шага `quit` для второго решения

```
proctype program()
{
    Step1: <описание шага> goto <следующий шаг>
    Step2: <описание шага> goto <следующий шаг>
    QuitStep: <описание шага> goto LastStep
    . . .
    LastStep: skip;
}
```

Третье требование в обеих схемах реализуется оператором `if` (пример 4.7). Считается, что вычислительный процесс может пойти по любой ветви. Отметим отдельно, что во многих классах языка ветвление полное, то есть каждой ветви с условием C обязательно соответствует ветвь с условием $\neg C$, поэтому не требуется моделирование условия типа `default`; Однако другие классы, например, `Switch`, предполагают альтернативу, когда ни одно из условий не выполнено. В примере 4.7 приведен общий вид конструкции, моделирующей ветвление.

Пример 4.7. Моделирование ветвления

```
if
     ::skip -> ... /* шаги ветви 1 */
     ::skip -> ... /* шаги ветви 2 */
     ...
     ::else -> ... /* шаги альтернативной ветви */
```

```
fi
```

Проанализируем достоинства и недостатки обоих вариантов.

В случае первого варианта для каждого состояния, генерируемого верификатором, требуется хранить глобально идентификатор шага.

Во втором случае дополнительные данные не хранятся, однако необходимо тщательно следить, чтобы описание шага заканчивалось инструкцией `goto`, иначе вычислительный процесс перейдет в некорректную область. В первом же случае эта ситуация обернется тем, что процесс будет постоянно находиться на одном и том же шаге, такое поведение более приемлемо.

При пренебрежимо меньшем объеме требуемой памяти для хранения состояния, генерируемого верификатором, реализация второго варианта накладывает дополнительную ответственность (оператор `goto`) на каждый шаг, тело и блок обработки ветвей шага, что выливается в увеличение объема генерируемого кода и снижению его читаемости и надежности.

Вывод: первый вариант является более предпочтительным.

4.2. Доказательство корректности построенного отображения

Докажем, что отображение, построенное по первому варианту, является моделирующим. Будем вести доказательство в соответствии с определением 3.2.

Проверим первое свойство. Пример 4.8 демонстрирует общую структуру транслированной программы. Поскольку при запуске `step_id` указывает на первый исполняемый шаг программы (по построению), то в первую очередь будет выполнен код, соответствующий первому шагу. Аналогично доказательство в обратную сторону: в первую очередь выполняется код, соответствующий номеру шага, указанному в переменной `step_id`. Этот номер по построению соответствует первому шагу исходной программы *MBML*. Первый пункт доказан.

Пример 4.8. Общая структура транслированной программы

```
short step_id = 1;
proctype program()
{
    do
       :: (step_id == 1)-> <описание шага>;
       :: (step_id == 2)-> <описание шага>;
       :: (step_id == 3)-> <описание шага>;
       . . .
       :: (step_id == <N>)-> <описание шага>;
    od
}
```

Теперь покажем, что $\forall s_a, s_b \in p: \quad \exists \tau : trace(\tau) = ...s_a s_b ... \Leftrightarrow \exists$ симуляция $\sigma : trace(\sigma) = ...Trans(s_a)Trans(s_b)...$

Если в следе *MBML*-программы p существует последовательность шагов $s_a s_b$, то это соответствует одному из следующих случаев:

1. s_b является одним из вариантов выбора, либо $s_b = Exit(s_a)$ для шага s_a типа `Menu`;
2. s_b является одним из вариантов ветвления (в том числе, возможно, вариантом «если ни одно из условий не выполнено») для шага s_a типа `Switch`;
3. $s_b = Next(s_a)$ либо $s_b = Exit(s_a)$ для шагов s_a остальных типов.

Рассмотрим первый случай. К шагам типа `Menu` применяется трансляция, показанная в примере 4.9. Любая из ветвей транслированной программы может быть выбрана, а, следовательно, существует и такой вычислительный процесс, который выберет ID-шага, соответствующего s_b.

Пример 4.9. Трансляция шага типа `Menu`

```
...
/* код шага типа Menu */
    if
       :: skip -> step_id = <ID шага пункта 1 меню>;
       :: skip -> step_id = <ID шага пункта 2 меню>;
       . . .
       :: skip -> step_id = <ID шага пункта N меню>;
       :: skip -> <ID шага Exit-пункта меню>
    fi
}
```

Отметим, что аналогичные рассуждения применимы и к шагам, где выбранное значение запоминается в переменной и затем используется в программе. В примере 4.10 показана трансляция, применяемая к шагам типа `List`. Каждому возможному значению соответствует своя ветвь. Любая из ветвей транслированной программы может быть выбрана.

Пример 4.10. Трансляция шага типа `List`

```
...
/* код шага типа List */
    if
       :: skip ->step_id =<ID шага Exit-пункта меню>
       :: skip ->
            <флаг инициализации переменной>
            if
             ::skip -> <Переменная> = <Значение 1>;
             ::skip -> <Переменная> = <Значение 2>;
             . . .
             ::skip -> <Переменная> = <Значение N>;
```

```
            fi
    fi
}
```

Рассмотрим второй случай. К шагам типа `Switch` применяется трансляция, соответствующая примеру 4.11. Здесь также любая из ветвей транслированной программы может быть выбрана, а, следовательно, существует и такой вычислительный процесс, который выберет ID-шага, соответствующего s_b.

Пример 4.11. Трансляция шага типа `Switch`

```
...
/* код шага типа Switch */
    if
       :: skip -> step_id = <ID шага условия 1>;
       :: skip -> step_id = <ID шага условия 2>;
       . . .
       :: skip -> step_id = <ID шага условия N>;
       :: skip -> <ID шага, если ни одно условие не
                   удовлетворено>
    fi
}
```

В последнем случае доказательство строится аналогично. К шагам общего вида применяется трансляция в соответствии с примером 4.12. Любая из ветвей транслированной программы может быть выбрана, а, следовательно, существует и такой вычислительный процесс, который выберет ID-шага, соответствующего s_b.

Пример 4.12. Трансляция шага общего вида

```
...
/* код шага общего вида*/
```

```
    ... /* выполняемые действия */
    if
       :: skip -> step_id = <ID шага Exit-свойства>;
       :: skip -> step_id = <ID шага Next-свойства>;
    fi
}
```

Необходимость второго пункта доказана во всех случаях. Достаточность требует, чтобы любая генерируемая в процессе преобразования инструкция была частью некоторого $Trans(s)$. Это следует из построения отображения. Таким образом, второй пункт доказан, и, следовательно, отображение моделирующее, общая корректность доказана.

4.3. Проверка верифицируемых свойств

Рассмотрим, каким образом свойства, перечисленные в предыдущей главе, могут быть проверены при запуске верификатора *SPIN* по транслированной программе.

Требование *корректности завершения* удовлетворяется следующей *LTL*-формулой: $!F(main\_menu \wedge G(!close\_session))$. Она означает, что пользователь после начала сеанса (`main_menu`) должен иметь возможность завершить сеанс (`close_session`).

Для верификации свойства отсутствия избыточности необходимо, чтобы присутствие недостижимых шагов в исходной программе было равносильно наличию недостижимых инструкций в транслированной программе. В свою очередь, чтобы это условие выполнялось, необходимо, чтобы для любой конструкции *MBML* (шаг, ветвь и условие), достижимой в *MBML*-программе, все соответствующие ей инструкции в транслированной программе были достижимы. В общем случае это условие не выполняется из-за использования конструкции `if` в различных случаях (кроме моделирования ветвей). В данном случае, поскольку оператор `if` используется только для выбора одной из

ветвей шага, любое из ветвлений оператора `if` выполнимо (существует соответствующий запуск). Таким образом, отсутствие избыточности проверяемо.

Для того чтобы проверить свойство корректности обращения к переменным контекста, достаточно промоделировать работу автомата, определяющего допустимость обращения к переменной, приведенного на рис. 3. Для каждой переменной дополнительно хранится бит состояния. Нулевое значение означает состояние «не инициализирована», а единица – «инициализирована». Соответственно, во всех точках инициализации переменной будем устанавливать бит в единицу, а во всех точках обращения к ее значению – проверяется, что бит установлен, как в примере 4.13.

Пример 4.13. Проверка бита четности переменной

```
bit a = 0;
proctype program()
{
  ...
  a = 1; /*для любой операции присваивания*/
  ...
  assert(a == 1); /*для любого обращения к значению
                    переменной*/
  ...
 }
```

Таким образом, это свойство также проверяемо, если добавить соответствующие действия и проверки во все шаги, работающие с переменными. Пример 4.14 демонстрирует трансляцию шага ввода данных `Input`, в котором устанавливается бит четности для запоминаемой переменной.

Пример 4.14. Трансляция шага типа `Input`

```
...
/* код шага типа Input */
    if
       :: skip ->step_id =<ID шага Exit-пункта меню>
       :: skip ->
            /* установка бита четности*/
            _check_<имя_переменной> = 1;
            <Переменная> =  <Значение>;
            Step_id = <ID шага Next-пункта меню>;
    fi
}
```

В дальнейшем бит четности проверяется в шагах, где требуется обращение к переменной. Пример 4.15 показывает трансляцию шага отправки запроса на сервер `Submit`, в котором проверяется бит четности для каждой переменной запроса.

Пример 4.15. Трансляция шага типа `Submit`

```
...
/* код шага типа Submit */
    if
       assert (_check_<имя_переменной_1> == 1);
       assert (_check_<имя_переменной_1> == 2);
       ...
       assert (_check_<имя_переменной_N> == 1);
       :: skip ->step_id =<ID шага Exit-пункта меню>
       :: skip ->step_id =<ID шага Next-пункта меню>
    fi
}
```

Бизнес-ограничения представимы в виде *LTL*-формул. Например, требование «*PIN* всегда должен проверяться перед операцией» можно записать в виде следующей формулы:

$$G(main\_menu \rightarrow ((G(!oper)) \vee (F(pin\_entry)U(oper)))).$$

Она означает, что пользователь после начала сеанса (`main_menu`) сеанса имеет две возможности: никогда не выполнить ни одной операции или в какой-то момент ввести *PIN* перед выполнением операции. Эта формула в терминах шагов записывается также в виде: $G((step\_id == n_1) \rightarrow ((G(!(step\_id == n_3))) \vee (F(step\_id == n_2)U(step\_id == n_3))))$

Вывод: построенное отображение обеспечивает возможность для верификации всех требуемых свойств.

4.4. Реализация транслятора программа MBML в PROMELA

Одним из основных результатов этой работы стала реализация транслятора, выполняющего описанное выше отображение множества программ на *MBML* во множество программ на языке *PROMELA*.

При реализации транслятора учитывались следующие соображения: транслятор должен легко вызываться из другого приложения; верификатор (представляющий комплекс, в который входят транслятор, *SPIN*, интерпретатор результатов верификации и другие сопутствующие программы) должен стать частью интегрированной среды разработки программ на языке *MBML*.

Транслятор был реализован на языке *Java*. Среда разработки программ на *MBML* написана на *Java*, поэтому его использование мотивировано, кроме прочих соображений, требованием интегрируемости с этим инструментом.

Транслятор управляет процессом перевода программы, а также отвечает за перевод каждого типа базовых шагов (таких, как *Menu, Vars, Submit* и т.д.).

Трансляция программы на язык *MBML* состоит из двух основных этапов – предобработки программы и непосредственно ее перевода. На этапе предобработки задачи программы просматриваются, определяется множество

переводимых шагов, выделяется набор свойств и прочее. На втором этапе выполняется непосредственно перевод программы.

В процессе перевода автоматически определяется множество используемых переменных и множество свойств, аннотирующих программу.

После завершения перевода формируется окончательная программа, состоящая из объявлений необходимых переменных, определенных в процессе перевода, самого перевода в виде прототипа процесса и init-секции.

Одной из интересных задач, решенной в процессе разработки транслятора, стало определение по шагам *Vars* последовательности операций присвоения и обращения к переменным контекста.

Процедура верификации программ на MBML приведена на рис. 4.

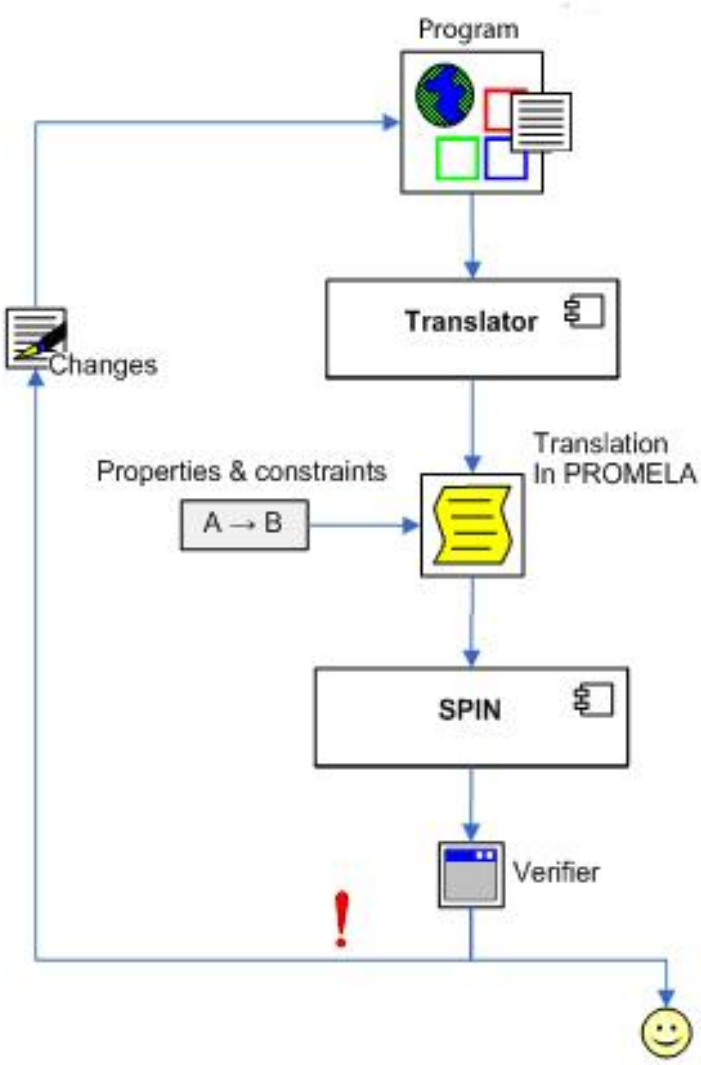

Рис. 4

Программа поступает на вход транслятора, порождающего транслированную программу на языке *PROMELA*. Верификатор *SPIN* генерирует на основе этой программы верификатор. Запуск верификатора завершается либо успешно, либо порождением сообщения об ошибке. В последнем случае в программу вносятся поправки, и процесс повторяется.

Хорошей иллюстрацией применения языка *MBML* является осуществление платежей в пользу операторов услуг через интерфейс мобильного банкинга.

Программа позволяет выполнить следующие функции:

- выбор получателя платежа из доступного списка;
- ввод реквизитов платежа: суммы, номера личного счета и прочих деталей.

В начале работы программа отображает меню доступных действий. Пользователь в интерактивном режиме выбирает один из возможных вариантов сценария проведения платежа. Далее пользователь указывает необходимые реквизиты, которые сохраняются для последующей отправки на сервер. Важно, что в каждый момент допустимы действия отдельных типов, определяемых классом текущего шага. В конце сценария для выполнения финансовых запросов используются шаги, выполняющие запрос на сервер банковской системы.

В примере 4.16 приводится листинг программы на *MBML*:

Пример 4.16. Исходная *MBML*-программа платежей мобильного-банкинга имеет вид:

```
; Entry point
main;Class=Init;Ver=1;Lang=RU;Next=menu;
; Main menu
Menu;Class=Menu;TITLE=Главное
меню;Платежи=payments;Выход=quit;Exit=quit;
; \-------------------------------------------\
```

```
payments;Class=Menu;TITLE=Платежи;За телефон=mbilling;Utility
Payments=utility;Exit=menu;
; Платеж за телефон
mbilling;Class=List;TITLE=Выберите
оператора;FIELD=CODE;Мегафон=MEGAFON;МТС=MTS;Билайн=Beeline;K
-Mobile=KMOBILE;Next=phone_type;Exit=menu;
phone_type;Class=Input;Mode=Numeric;FIELD=DEST;TITLE=Номер
телефона;MAX=11;MIN=11;Exit=mbilling;Next=phone_amnt;
phone_amnt;Class=Amount;TITLE=Сумма
платежа;FIELD=AMNT;Next=send_payment;Exit=phone_type;
send_payment;Class=Submit;RQ=PAY1;Fields=CODE,AMNT,DEST;Next=
Menu;Exit=Error;
;Utility-payments
utility;Class=List;TITLE=Select
payment;FIELD=CODE;GAS=GAS;PHONE=PHONE;Next=util_sel;Exit=Men
u;
util_sel;Class=Switch;FIELD=CODE;GAS=util_receipt;PHONE=mbill
ing;Next=Menu;
util_receipt;Class=Input;FIELD=DESC;TITLE=Enter       receipt
number;TEXT=Receipt;MODE=numeric;Next=util_amnt;Exit=utility;
util_amnt;Class=Amount;TITLE=Enter
amount;FIELD=AMNT;Next=send_u_payment;Exit=util_receipt;
send_u_payment;Class=Submit;RQ=UTL;Fields=CODE,AMNT,DESC;Next
=Menu;Exit=Error;
; \-----------------------------------------------\
; Выход из мидлета
Quit;Class=quit;
; Errors:
error;Class=Alert;Type=Alert;Title=Ошибка;Text=Произошла
ошибка. Попробуйте еще раз.;Next=Menu;Exit=Quit;
```

Соответствующая преобразованная разработанным транслятором программа приведена в Приложение 2. *MBML-программа* платежей мобильного-банкинга после транслирующего преобразования

В процессе верификации программы были обнаружены недостижимые шаги. Также было обнаружено недопустимое обращение к переменной при отправке запроса коммунального платежа.

После исправления указанных ошибок и повторной верификации новых ошибок выявлено не было.

Время трансляции программы, генерации верификатора и его выполнения составляет не более одной-двух секунд, что позволяет использовать его в интегрированной среде разработки приложений для мгновенной проверки редактируемых конфигураций.

Выводы по четвертой главе

Показано практическое применение автоматного подхода к построению моделей и их верификации, позволяющего добиться повышения надежности кода и следующего из этого экономического эффекта в финансовых приложениях.

ЗАКЛЮЧЕНИЕ

1. Предложена модель для программ на языке *MBML*, в которой поведение описывается на основе конечных автоматов. Приведено обоснование того, что конечный автомат естественным образом моделирует работу программы.

2. Приведены аспекты теории верификации, которые были использованы для верификации построенной автоматной модели.

3. Разработано транслирующее отображение, реализующее предложенную модель.

4. Поставлен ряд экспериментов с моделью, выявивших ошибки в исходной программе.

5. Выявлены ограничения использования модели.

ИСТОЧНИКИ

1. *Шалыто А. А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1/
2. *Разработка* технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Отчет по контракту о верификации автоматных программ. Второй этап. СПбГУ ИТМО, 2007. http://is.ifmo.ru/verification/_2007_02_report-verification.pdf
3. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. МЦНМО, 2002. http://is.ifmo.ru/verification/_klark_gamberg_pered_verification.djvu
4. *Егоров К. В.* Разработка верификатора автоматных программ http://is.ifmo.ru/papers/automataverificator

5. *Миронов А. М.* Математическая теория программных систем. http://intsys.msu.ru/staff/mironov/mthprogsys.pdf
6. *Madhavan Mukund.* Linear-Time Temporal Logic and Bűchi Automata.
7. http://www.spinroot.com – официальный сайт инструмента *SPIN*.
8. http://www.spinroot.com/spin/Man/grammar.html – грамматика языка *PROMELA*.
9. *Bjorner N.* Verifying Temporal Properties of Reactive Systems: A STeP Tutorial. Kluwer Academic Publishers, 1999. http://www-step.stanford.edu/papers/fmsd00.pdf
10. *Интернет-ресурс* http://en.wikipedia.org.

Приложение 1. Основные конструкции языка *MBML*

Язык *MBML* имеет набор выделенных классов, реализующих базовые операции, которые, как правило, необходимы при разработке мобильного банковского клиента, рассмотрим наиболее важные из них:

Класс `Init`

Назначение: инициализация контекста меню.

Параметры:

- *SMC* – указываемый по умолчанию номер телефона для отправки резервных шифрованных *SMS*-сообщений;
- *SMS_PREFIX* – фиксированный префикс для резервных шифрованных *SMS*-сообщений.

Класс `Menu`

Назначение: отображение пунктов меню.

Параметры:

- *TITLE* – заголовок экрана;
- Перечень пунктов меню с привязкой к шагам в формате:
  `[<Display Menu Item Name> = <Step ID>;]`

Класс `List`

Назначение: выбор значения переменной из списка.

Параметры:

- *TITLE* – заголовок экрана;
- Перечень элементов списка и их значений в формате:
  `[<Display Item Name> = <Data Value>;]`
- *FIELD=<Variable Name>* – имя переменной, куда помещаются данные выбранного элемента.

Класс `Switch`

Назначение: ветвление по значению указанной переменной.

Параметры:

- *FIELD* – имя переменной, откуда выбирается значение для ветвления;
- *Next* – шаг, на который осуществляется переход в случае, если ни одно из условий не выполнено;
- Перечень значений и соответствующих им ID-шагов в формате:
  `[<Data Value> = <Step ID>;]`

Класс `Input`

Назначение: поле для ввода данных.

Параметры:

- *FIELD* – имя переменной, которой будет присвоено введенное значение;
- *TITLE* – заголовок экрана;
- *TEXT* – название поля (надпись);
- *MIN* – минимальная длина поля;
- *MAX* – максимальная длина поля.
- *Mode=<флаг1>,..,<флагN>* – список флагов, определяющих характеристики поля ввода. Флаги определяют тип вводимых данных. Должно быть указано одно из перечисленных ниже значений. В случае если указаны несколько значений из списка, тип поля будет назначен в соответствии с приоритетом поля. Значения перечислены в порядке убывания приоритета:
  - "EMAILADDR" – поле для ввода адреса электронной почты;
  - "NUMERIC" – поле для ввода целых числовых данных (число ограничивается максимальным значением типа `Integer`, определенным в *Java*);

- "DECIMAL" – поле для ввода чисел с дробной частью;
- "PHONENUMBER" – поле для ввода номера телефона (некоторые модели телефонов позволяют вставлять номера телефонов из адресной книги);
- "URL" – поле для ввода *URL*-адреса;
- "PASSWORD" – поле для ввода пароля – вводимые данные маскируются;
- "SENSITIVE" – поле для ввода секретных величин – вводимые данные маскируются.

Класс `Alert`

Назначение: отображение диалогового окна с сообщением указанной категории.

Параметры:

- *Type* – тип сообщения:
  - "ALARM" – сообщение типа "Alarm";
  - "CONFIRM" – сообщение типа "Confirmation";
  - "WARNING" – сообщение типа "Warning";
  - "ERROR" – сообщение типа "Error";
  - "INFO" или любое другое значение – сообщение типа "Info";
- *Title* – заголовок окна.

Класс `Quit`

Назначение: выход из приложения.

Класс `Amount`

Назначение: ввод суммы и выбора валюты.

Параметры:

- *Title* – заголовок окна;

- *Text* – текст подсказки;
- *FIELD* – список переменных для сохранения значения введенной суммы и валюты. Имена указываются через запятую; если имя переменной, содержащей валюту, не указано, то по умолчанию используется `CURR`.

Класс *Submit*

Назначение: отправка запроса на сервер.

Параметры:

- *Fields=<n1>, <n2>...;* – список переменных, которые требуется передать в запросе;
- *RQ=<код запроса1>, ..., <код запросаN>;* – соответствие кода и выполняемой операции.

  Например:

  - "ACL" – получение списка счетов клиента
  - "PIN" – отправка на сервер нового *PIN*-кода мидлета;
  - "BLOCK" – блокировка карты;
  - "MINI" – запрос минивыписки;

  Допустимо указывать через запятую сразу же несколько кодов запросов для выполнения их за один проход, например, `"RQ=BLOCK,ACL;".`
- *OK=<ID шага>;* – необязательный параметр, позволяющий переопределить содержимое демонстрируемого по умолчанию окна "Successfully Completed";
- *SMC* – номер телефона для отправки *SMS*. Если параметр не указан, для отправки *SMS*-сообщения используется номер, указанный в одноименном параметре шага класса `Init`.

Класс `Vars`

Назначение: выполнение операций над переменными.

Операции над переменными выполняются последовательно. Каждая операция возвращает значение "true" (истина), если выполнена успешно (или выражение истинно). Далее выполняется следующая операция. Если операций больше нет, то происходит переход к шагу, определенному в параметре *Next*. Если операция возвращает значение "False" (ложь), то выполнение цепочки прерывается, и происходит переход к шагу, определенному в параметре *Exit*.

Формат:

```
[<операция>=<операнды через запятую>;]
```

Пример операции:

- "EQ" – сравнение двух или более переменных. Например:

  ```
  EQ=PIN,PIN2;
  ```

- "EQE" – сравнение двух или более строковых выражений. Например, сравнить без учета регистра значение переменной PASSWORD со словом "SECRET":

  ```
  EQE={PASSWORD,UPPER},SECRET;
  ```

- "Set" – присвоить переменной строковое значение.

  ```
  Set=<имя переменной>:<значение>;
  ```

  С помощью этой операции можно также присвоить значение одной переменной другой переменной (скопировать):

  ```
  Set=TO:{FROM}
  ```

- "Del" – удалить переменную из памяти приложения. Например:

  ```
  Del=AMOUNT;
  ```

- "SetPIN" – выполнить установку нового *PIN*-кода мидлета.

Например:

```
SetPIN=PIN;
```

- "MapList" – операция служит для создания ассоциативного массива (`Map`) из списка переменных и их значений (`List`). Определяется в следующем формате:

  `MapList = <Имя массива>:<Имя списка>:[Переменные через запятую];`

  Как правило, требуется долговременное хранение данных массивов, поэтому рекомендуется использовать модификатор "$".

Класс `SMS`

Назначение: отправка *SMS*-сообщения.

Параметры:

- *SMC* – номер телефона для отправки *SMS*. Если параметр не указан, то для отправки *SMS*-сообщения используется номер, указанный в одноименном параметре шага класса `Init`.
- *SMS* – текст *SMS*-сообщения.

Переменные

Объявление переменных и присвоение им значений выполняется в описании объектов меню (шагов) как параметр с уникальным наименованием. Данные переменные носят название контекстных переменных. Область видимости этих переменных – вся конфигурация меню. Время их жизни – шаги, выполняемые между шагами класса `Init`.

Обычно шаг класса `Init` выполняется при запуске/перезапуске мидлета. Для сохранения значений переменных между перезапусками мидлета используются, так называемые, постоянные (*persistent*) переменные, для которых в качестве префикса указывается модификатор "$". Данные переменные хранятся в долговременной области памяти *RecordStore* мобильного телефона.

При этом контекстные переменные перегружают *persistent*-переменные: если программе не удается найти контекстную переменную (например, *TEXT1*), она пытается найти *persistent*-переменную с таким же именем (*$TEXT1*), однако при записи значений запись происходит в контекстную переменную (выполняется ее объявление).

Метасимволы

- $<имя переменной> – создание persistent-переменной – сохранение значения в *RecordStore* мобильного телефона. При этом значение контекстной переменной не изменяется.
- {<имя переменной>} – для вставки значения переменной (используется для текстовых полей меню);
- {<имя переменной>, преобразование, преобразование, ...} – преобразование строки.

  Функции преобразования применяются к операнду последовательно.

  Используются следующие функции:
  - "SUBSTR: <позиция>, <кол-во>" – возвращает подстроку. Если позиция < 0, то отсчет ведется от конца строки;
  - "TRIM" – удаление лишних пробелов;
  - "UPPER" – приведение к верхнему регистру;
  - "LOWER" – приведение к нижнему регистру.

  Например, запись "{TEXT,trim,substr:1:1,upper}" означает следующее: взять значение переменной "TEXT", удалить пробелы, взять первую букву, привести в верхний регистр.
- {#(xx)}, где (xx) – *HEX*-последовательность UTF-8 байтов.

  Используется для замены *escape*-последовательностей, например, {#0A} - принудительный перевод строки в диалоговых окнах.

Приложение 2. *MBML*-программа платежей мобильного-банкинга после транслирующего преобразования

```
short step_id =1;
bit _check_ver;
short ver;
bit _check_lang;
short lang;
bit _check_code;
short code;
bit _check_dest;
short dest;
bit _check_amnt;
short amnt;
bit _check_desc;
short desc;


proctype program()
{
do
   /* id = main; class = Init */
   :: (step_id == 1)->
     _check_ver = 1;
     ver = 1;
     _check_lang = 1;
     lang = 1;
     step_id = 2;

   /* id = menu; class = Menu */
   :: (step_id == 2)->
     if
        :: skip -> step_id = 3;
        :: skip -> step_id = 4;
        :: skip -> step_id = 4;
```

```
  fi

/* id = payments; class = Menu */
:: (step_id == 3)->
  if
     :: skip -> step_id = 5;
     :: skip -> step_id = 6;
     :: skip -> step_id = 2;
  fi

/* id = mbilling; class = List */
:: (step_id == 5)->
  if
     :: skip -> step_id = 2;
     :: skip ->
       _check_code = 1;
       if
          :: skip -> code = 1;
          :: skip -> code = 2;
          :: skip -> code = 3;
          :: skip -> code = 4;
       fi;
       step_id = 7;
  fi

/* id = phone_type; class = Input */
:: (step_id == 7)->
  if
     :: skip -> step_id = 5;
     :: skip ->
       _check_dest = 1;
       dest = 1;
       step_id = 8;
  fi
```

```
/* id = phone_amnt; class = Amount */
:: (step_id == 8)->
  if
     :: skip -> step_id = 7;
     :: skip ->
       _check_amnt = 1;
       amnt = 1;
       step_id = 9;
  fi

/* id = send_payment; class = Submit */
:: (step_id == 9)->
  assert (_check_code == 1);
  assert (_check_amnt == 1);
  assert (_check_dest == 1);
  if
     :: skip -> step_id = 10;
     :: skip ->  step_id = 2;
  fi

/* id = utility; class = List */
:: (step_id == 6)->
  if
     :: skip -> step_id = 2;
     :: skip ->
       _check_code = 1;
       if
          :: skip -> code = 1;
          :: skip -> code = 2;
       fi;
       step_id = 11;
  fi
```

```
/* id = util_sel; class = Switch */
:: (step_id == 11)->
  assert (_check_code == 1);
  if
     :: skip -> step_id = 12;
     :: skip -> step_id = 5;
     :: skip -> step_id = 2;
  fi

/* id = util_receipt; class = Input */
:: (step_id == 12)->
  if
     :: skip -> step_id = 6;
     :: skip ->
       _check_desc = 1;
       desc = 1;
       step_id = 13;
  fi

/* id = util_amnt; class = Amount */
:: (step_id == 13)->
  if
     :: skip -> step_id = 12;
     :: skip ->
       _check_amnt = 1;
       amnt = 1;
       step_id = 14;
  fi

/* id = send_u_payment; class = Submit */
:: (step_id == 14)->
  assert (_check_code == 1);
  assert (_check_amnt == 1);
```

```
      assert (_check_desc == 1);
      if
         :: skip -> step_id = 10;
         :: skip ->  step_id = 2;
      fi


   /* id = quit; class = Quit */
   :: (step_id == 4)-> break;

   /* id = error; class = Alert */
   :: (step_id == 10)->
      if
         :: skip -> step_id = 2;
         :: skip ->  step_id = 4;
      fi

   :: else -> assert (false);
od
}

init {
run program();
}
```