Министерство образования и науки Российской Федерации

УДК 004.4’242
ГРНТИ 50.41.25
Инв. №

УТВЕРЖДЕНО:
Исполнитель: **Государственное образовательное учреждение высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики»**
От имени Руководителя организации **_____/_____/** **М.П.**

НАУЧНО-ТЕХНИЧЕСКИЙ ОТЧЕТ

о выполнении 2 этапа Государственного контракта
№ П2373 от 18 ноября 2009 г. и Дополнению от 26 февраля 2010 г. № 1/П2373

Исполнитель: Государственное образовательное учреждение высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики»
Программа (мероприятие): Федеральная целевая программа «Научные и научно-педагогические кадры инновационной России» на 2009-2013 гг., в рамках реализации мероприятия № 1.2.2 Проведение научных исследований научными группами под руководством кандидатов наук.
Проект: Методы повышения качества при разработке автоматных программ с использованием функциональных и объектно-ориентированных языков программирования.
Руководитель проекта: **_____/Шопырин Данил Геннадьевич** (подпись)

Санкт-Петербург
2010 г.

СПИСОК ОСНОВНЫХ ИСПОЛНИТЕЛЕЙ

по Государственному контракту П2373 от 18 ноября 2009 на выполнение поисковых научно-исследовательских работ для государственных нужд

Организация-Исполнитель: Государственное образовательное учреждение высшего профессионального образования "Санкт-Петербургский государственный университет информационных технологий, механики и оптики"

Руководитель темы:

кандидат технических наук, без ученого звания	_____ подпись, дата	Шопырин Д. Г.

Исполнители темы:

кандидат технических наук, без ученого звания	_____ подпись, дата	Гуров В. С.
кандидат технических наук, без ученого звания	_____ подпись, дата	Корнеев Г. А.
без ученой степени, без ученого звания	_____ подпись, дата	Царев Ф. Н.
без ученой степени, без ученого звания	_____ подпись, дата	Степанов О. Г.
без ученой степени, без ученого звания	_____ подпись, дата	Лукин М. А.

без ученой степени, без ученого звания	_____ подпись, дата	Астафуров А. А.
без ученой степени, без ученого звания	_____ подпись, дата	Клебанов А.А.
без ученой степени, без ученого звания	_____ подпись, дата	Яминов Б. Р.
без ученой степени, без ученого звания	_____ подпись, дата	Егоров К. В.
без ученой степени, без ученого звания	_____ подпись, дата	Царев М. Н.
без ученой степени, без ученого звания	_____ подпись, дата	Малаховски Я. М.
без ученой степени, без ученого звания	_____ подпись, дата	Буздалов М. В.
без ученой степени, без ученого звания	_____ подпись, дата	Борисенко А. А.
без ученой степени, без ученого звания	_____ подпись, дата	Федотов П. В.

РЕФЕРАТ

Отчет 98 с., 3 гл., 43 рис., 1 табл., 31 источн., 3 прил,

Ключевые слова: автоматное программирование; валидация; верификация; объектно-ориентированное программирование; качество программ; функциональное программирование.

В настоящем отчете излагаются результаты выполнения второго этапа поисковых научно-исследовательских работ по направлениям «Физика конденсированных сред. Физическое материаловедение», «Коллоидная химия и поверхностные явления», «Информатика», «Общая биология и генетика» по проблеме «Методы повышения качества при разработке автоматных программ с использованием функциональных и объектно-ориентированных языков программирования», выполняемых в рамках государственного контракта, заключенного между Федеральным агентством по образованию и государственным образовательным учреждением высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» в соответствии с решением Единой комиссии (протокол от 29 октября 2009 г. № 3/НК-421П) по конкурсу № НК-421П «Проведение поисковых научно-исследовательских работ по направлениям: «Физика конденсированных сред. Физическое материаловедение», «Коллоидная химия и поверхностные явления», «Информатика», «Общая биология и генетика» в рамках мероприятия 1.2.2 Программы» мероприятия 1.2.2 «Проведение научных исследований научными группами под руководством кандидатов наук» по направлению 1 «Стимулирование закрепления молодежи в сфере науки, образования и высоких технологий» федеральной целевой программы «Научные и научно-педагогические кадры инновационной России» на 2009 - 2013 годы, утвержденной постановлением Правительства Российской Федерации от 28 июля 2008 года № 568 «О федеральной целевой программе «Научные и научно-педагогические кадры инновационной России» на 2009-2013 годы».

Целями настоящего этапа являются:

1. Программная реализация и экспериментальное исследование методов реализации автоматов на функциональных языках программирования.
2. Разработка, программная реализация и экспериментальное исследование методов рефакторинга автоматов при разработке автоматных программ.
3. Разработка, программная реализация и экспериментальное исследование метода динамической верификации автоматных программ.
4. Разработка, программная реализация и экспериментальное исследование методов применения контрактов при разработке объектно-ориентированных автоматных систем.

При выполнении второго этапа работ использовался следующий инструментарий:

1. Работы членов коллектива, опубликованные в рамках НИР по схожей тематике.
2. Персональный компьютер.
3. Язык программирования Haskell.
4. Интегрированная среда разработки Eclipse.
5. Дополнение EclipseFP для интегрированной среды разработки Eclipse.
6. Язык программирования Java.
7. Интегрированная среда разработки MPS.
8. Часть 4 Гражданского кодекса Российской Федерации.
9. Требования к оформлению публикаций.
10. ГОСТ 7.32-2001 «Отчет о научно-исследовательской работе. Структура и правила оформления».

Излагаются результаты выполнения аналитического обзора по следующим направлениям: качество программ, автоматное программирование, функциональное программирование, объектно-ориентированное программирование, автоматное объектно-ориентированное программирование, методы повышения качества объектно-ориентированных программ, методы повышения качества автоматных объектно-ориентированных программ, обзор методов и средств верификации на модели.

Приводится обоснование выбора оптимального варианта направления исследований, а также план проведения теоретических и экспериментальных исследований.

Описываются методы реализации автоматов на функциональных языках программирования. Приводятся примеры применения описанных методов.

Описываются результаты программной реализации и экспериментального исследования методов реализации автоматов на функциональных языках программирования.

Излагаются разработка, программная реализация и экспериментальное исследование методов рефакторинга автоматов при разработке автоматных программ, а также метода динамической верификации автоматных программ.

Приводится разработка, программная реализация и экспериментальное исследование методов применения контрактов при разработке объектно-ориентированных автоматных систем.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящем отчете излагаются результаты выполнения второго этапа поисковых научно-исследовательских работ по направлениям «Физика конденсированных сред. Физическое материаловедение», «Коллоидная химия и поверхностные явления», «Информатика», «Общая биология и генетика» по проблеме «Методы повышения качества при разработке автоматных программ с использованием функциональных и объектно-ориентированных языков программирования», выполняемых в рамках государственного контракта, заключенного между Федеральным агентством по образованию и государственным образовательным учреждением высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» в соответствии с решением Единой комиссии (протокол от 29 октября 2009 г. № 3/НК-421П) по конкурсу № НК-421П «Проведение поисковых научно-исследовательских работ по направлениям: «Физика конденсированных сред. Физическое материаловедение», «Коллоидная химия и поверхностные явления», «Информатика», «Общая биология и генетика» в рамках мероприятия 1.2.2 Программы» мероприятия 1.2.2 «Проведение научных исследований научными группами под руководством кандидатов наук» по направлению 1 «Стимулирование закрепления молодежи в сфере науки, образования и высоких технологий» федеральной целевой программы «Научные и научно-педагогические кадры инновационной России» на 2009 - 2013 годы, утвержденной постановлением Правительства Российской Федерации от 28 июля 2008 года № 568 «О федеральной целевой программе «Научные и научно-педагогические кадры инновационной России» на 2009-2013 годы».

Целями настоящего этапа являются:

1. Программная реализация и экспериментальное исследование методов реализации автоматов на функциональных языках программирования.
2. Разработка, программная реализация и экспериментальное исследование методов рефакторинга автоматов при разработке автоматных программ.
3. Разработка, программная реализация и экспериментальное исследование метода динамической верификации автоматных программ.
4. Разработка, программная реализация и экспериментальное исследование методов применения контрактов при разработке объектно-ориентированных автоматных систем.

Отчет имеет следующую структуру. В первой главе приводятся результаты выполнения аналитического отчета по следующим направлениям:

- программная реализация и экспериментальное исследование методов реализации автоматов на функциональных языках программирования;
- разработка, программная реализация и экспериментальное исследование методов рефакторинга автоматов при разработке автоматных программ;
- разработка, программная реализация и экспериментальное исследование метода динамической верификации автоматных программ;
- разработка, программная реализация и экспериментальное исследование методов применения контрактов при разработке объектно-ориентированных автоматных систем.

Во второй главе приводятся обобщение результатов теоретических и экспериментальных исследований.

В третьей главе приводится публикации результатов НИР.

В заключении дается общая оценка работ по этапу.

Функциональное программирование в последние годы становится весьма популярной областью исследований. В свою очередь, современные средства разработки на функциональных языках предоставляют программные библиотеки, в которых можно найти практически любой

популярный алгоритм, структуру данных или вспомогательные конструкции для реализации различных методов программирования (например, монады), однако в этих библиотеках отсутствуют инструменты для применения подходов автоматного программирования.

Большинство разрабатываемых сегодня программ строится на основе объектно-ориентированного подхода, для которого разработан целый рад методов повышения качества, таких как: тестирование, рефакторинг и паттерны проектирования. В тоже время, активно применяемые в ответственных системах методы объектно-ориентированного автоматного программирования также требуют разработки методов повышения качества для автоматных частей таких программ.

Изложенное позволяет утверждать, что результаты выполнения научно-исследовательской работы будут превышать мировой уровень разработок в рассматриваемой области.

1. АНАЛИТИЧЕСКИЙ ОТЧЕТ О ПРОВЕДЕНИИ ТЕОРЕТИЧЕСКИХ И ЭКСПЕРИМЕНТАЛЬНЫХ ИССЛЕДОВАНИЙ

В настоящем разделе приведен аналитический отчет о проведении теоретических и экспериментальных исследований.

1.1. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ И ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ МЕТОДОВ РЕАЛИЗАЦИИ АВТОМАТОВ НА ФУНКЦИОНАЛЬНЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

В данном разделе приводится программная реализация библиотеки для поддержки методов автоматного программирования на функциональных языках программирования, разработанные в рамках первого этапа настоящего Государственного контракта [1], а также практическая апробация разработанных методов на примере ответственной системы – сервера обмена мгновенными сообщениями.

1.1.1. Программная реализация библиотеки, поддерживающей разработанные методы реализации автоматов на функциональных языках программирования

На основе, разработанных ранее [1] библиотечных функции и методов реализации активных автоматов, была разработана библиотека, приведенная в листинге 1.

Листинг 1. Библиотека поддержки автоматного программирования

```
module Control.FiniteState where

-- Тип функции переходов.
type StateTransition state event output = state -> event ->
        (state, [output])

-- Тип функции переходов, пригодной для использования вместе с
        foldl.
type FoldableTransition state event output = (state, [output])
        -> event -> (state, [output])

-- Преобразователь из StateTransition в FoldableTransition
stt2foldable :: StateTransition state event output ->
        FoldableTransition state event output
stt2foldable transition (pstate, accumulator) event = (nstate,
        accumulator ++ output)
    where (nstate, output) = transition pstate event

-- Модифицированная функция-преобразователь
-- из автомата Мура в автомат Мили.
moore2mealy transition pstate event = (nstate, output)
    where
        (_, trans, outr) = transition pstate
        xstate = trans event
        (inr, _, _) = transition nstate
        (nstate, output) = case xstate of
```

```
            Nothing -> (pstate, [])
            Just x -> (x, outr ++ inr)

-----------------------------------------------------
-- Комбинатор <<просто перейти в состояние state>>.
pure :: state -> (state, [output])
pure state = (state, [])

-- Комбинатор <<перейти в состояние state с одним выходным
        воздействием output>>.
blot :: state -> output -> (state, [output])
blot state output = (state, [output])

-- Комбинатор <<перейти в состояние state со списком выходных
        воздействий output>>.
dark :: state -> [output] -> (state, [output])
dark state output = (state, output)

-----------------------------------------------------
class AutomataState state where
    lastState :: state -> Bool

-- Структура, представляющая отдельный активный автомат.
data IOAutomata source state event output = IOAutomata
    (source -> IO (source, [event])) -- Получение событий из
        внешнего мира.
    (state -> event -> (state, [output])) -- Функция переходов.
    (source -> output -> IO source) -- Функция осуществления
        выходных воздействий.
    (source, state) -- Стартовое состояние системы.

-- Исполняет активный автомат.
runIOAutomata (IOAutomata ep stt og psystem) =
    runIOAutomata' ep stt og psystem

runIOAutomata' ep stt og (psource, pstate) = do
    (xsource, events) <- ep psource
    let (nstate, outputs) = foldl (stt2foldable stt) (pure
        pstate) events
    nsource <- foldlM og xsource outputs
    if lastState nstate
        then return ()
        else runIOAutomata' ep stt og (nsource, nstate)
    where
        foldlM _ s [] = return s
        foldlM ofunc s (o:os) = do
            ns <- ofunc s o
            foldlM ofunc ns os
```

1.1.2. Апробация разработанных методов на примере реализации ответственной системы – сервера обмена мгновенными сообщениями

В настоящем разделе производится апробация разработанных методов реализации конечных автоматов на функциональных языках программирования на примере реализации сервера мгновенных сообщений.

1.1.2.1. Протокол

Для простоты тестирования в качестве средства обмена данными выбран протокол *telnet* (протокол *TCP*, «возврат каретки, перевод строки» в качестве символа конца строки). После подключения клиент должен представиться серверу при помощи команды «*USR name*», где *name* – имя клиента. Такой метод не сложно дополнить запросом пароля, однако серверная часть не имеет какой-либо базы данных, поэтому вся авторизация сводится к сообщению клиентом своего имени серверу. В случае если клиент первым сообщением не представился системе – сокет закрывается сервером.

После авторизации клиент может посылать сообщения серверу, а сервер будет рассылать эти сообщения всем подключенным клиентам.

1.1.2.2. Детали реализации

Серверная часть содержит два типа автоматов – автомат-хаб и автомат-клиент. Автомат-хаб получает сообщения от автоматов-клиентов и организует рассылку сообщений между ними, а автоматы-клиенты обмениваются данными с клиентами сервера по протоколу *TCP* при помощи вспомогательных объектов, описанных ниже.

Диаграмма переходов автомата-хаба представлена на рис. 1, а исходный код функции переходов данного автомата представлен в листинге 2.

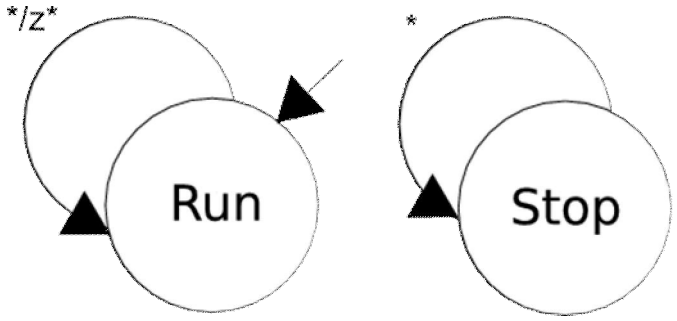


Рис. 1. Диаграмма переходов автомата-хаба. В приводимой реализации состояние «Stop» недостижимо, поскольку клиенты не могут попросить хаб остановится, однако эту возможность при необходимости нетрудно добавить. Подробная логика состояния «Run» не приводится

Листинг 2. Функция переходов автомата-хаба

```
-- Состояния автомата: <<Работает>> (<<Run>> на диаграмме),
-- <<Остановлен>> (<<Stop>> на диаграмме).
data HubState = Running | Stopped deriving Show
-- Входные воздействия: <<Новый клиент>>, <<Клиент
        отключился>>,
-- <<Получено сообщение>>.
data HubEvent = NewClient Int
              | DeadClient Int
              | GotMessage Int String deriving Show
-- Выходные воздействия: <<Послать строку всем клиентам>>,
-- <<Послать строку определенному клиенту>>.
data HubOutput = Send2All String | Send2ID Int String deriving
        Show

switchFunc Stopped _ = pure Stopped
switchFunc Running e = case e of
    NewClient i  -> blot Running $ Send2All $ show e
    DeadClient i -> blot Running $ Send2All $ show e
    GotMessage i s -> blot Running $ Send2All $
        (show i) ++ ": " ++ s
```

Реализация функции переходов этого автомата построена по методу, предложенному в [1], за тем исключением, что вместо оператора *case*, реализующего переход по состоянию используется *pattern matching*. В состоянии *Stopped* переход производится в себя без выходных воздействий. В состоянии *Running* переход также производится в себя, однако выходные воздействия зависят от события: при подключении или отключении клиента всем подключенным клиентам рассылаются уведомления, при получении сообщения всем клиентам рассылается строка с номером клиента-источника сообщения и текстом сообщения.

Диаграмма переходов автомата-клиента представлена на рис. 2, а исходный код функции переходов данного автомата – в листинге 3.

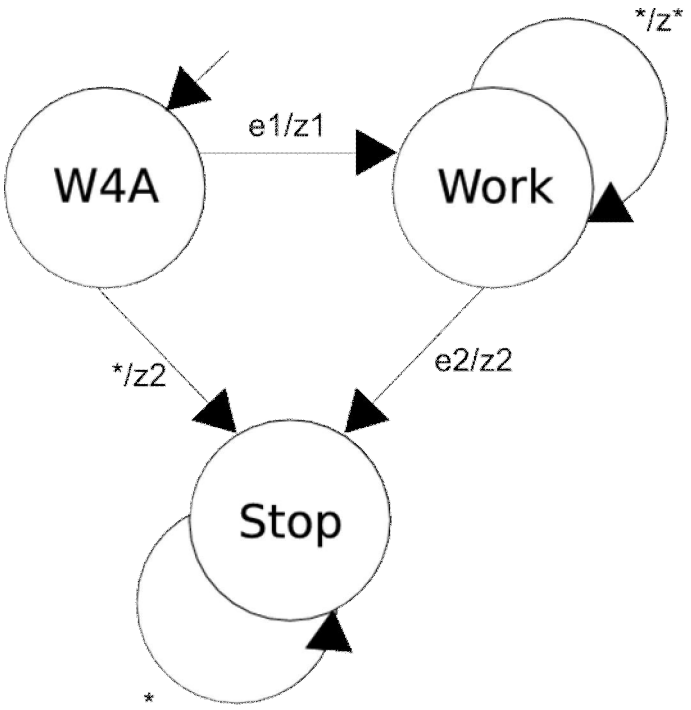


Рис. 2. Диаграмма переходов автомата-клиента. «e1» – сообщение вида «USR name», «e2» – сообщение о завершении (от автомата-хаба, держателя сокета или таймера), «z1» – остановить таймер, «z2» – сообщить автомату-хабу о своем завершении. Подробная логика состояния «Work» не приводится

Листинг 3. Функция переходов автомата-клиента

```
-- Состояния автомата:
-- <<Ожидание авторизации>> (<<W4A>> на диаграмме),
-- <<Рабочее состояние>> (<<Work>> на диаграмме),
-- <<Остановлен>> (<<Stop>> на диаграмме).
data ClientState = WaitForAuth
                 | Working String
                 | Stopped deriving Show

-- Входные воздействия:
-- <<Следует завершится>>,
-- <<Получена строка от клиента>>,
-- <<Следует отправить строку клиенту>>.
data ClientEvent = MustDie
                 | GotLine String
                 | SendLine String deriving Show

-- Выходные воздействия:
-- <<Сообщить автомату-хабу о своей остановке>>,
-- <<Сообщить о новом сообщении>>,
```

```
-- <<Протолкнуть строку в сокет клиента>>,
-- <<Остановить таймер>>.
data ClientOutput = SayDead
                  | SayMessage String
                  | PushMessage String
                  | StopTimer deriving Show

switchFunc s e = case s of
    WaitForAuth -> case e of
        MustDie -> die
        GotLine str -> let (c, p) = splitAt 3 str in
            if c == "USR"
                then blot (Working $ tail p) StopTimer
                else die
        SendLine str -> pure s
    Working name -> case e of
        MustDie -> die
        GotLine str -> blot s $ SayMessage $ "<" ++ name ++ ">
        " ++ str
        SendLine str -> blot s $ PushMessage str
    Stopped -> error "Must not"
    where die = blot Stopped SayDead
```

Функция переходов данного автомата построена в полном соответствии с методом, предложенным в [1]. Из состояния *WaitForAuth* переход производится в состояние *Working* только при получении сообщения авторизации, другие события либо игнорируются, либо завершают работу автомата. В состоянии *Working* производится передача сообщений от автомата-хаба к клиенту и обратно. При получении сообщения от клиента, к нему добавляется префикс имени данного клиента, полученный во время авторизации.

Вспомогательными объектами данной реализации являются: обработчик запросов на подключение, держатель *TCP*-сокета и таймер.

Обработчик запросов при подключении нового клиента создает новый автомат-клиент и запускает его в отдельном потоке. При создании автомата-клиента создается объект-держатель *TCP*-сокета и таймер.

Держатель *TCP*-сокета читает данные из сокета построчно и складывает полученные строки в *STM*-канал (Software Transactional Memory), из другого *STM*-канала держатель выбирает сообщения и проталкивает их в сокет. Аналогичным образом, через *STM*-каналы держатель сообщает о закрытии сокета клиентом сервера и закрывает сокет по требованию автомата-клиента.

Таймер в данной реализации – объект, который засыпает на определенный промежуток времени, а при пробуждении сообщает об этом в *STM*-канал.

Общая схема зависимостей классов объектов сервера обмена мгновенными сообщениями представлена на рис. 3. Ребро между двумя классами объектов на схеме означает, что эти классы как-либо взаимодействуют в работающей программе (например, обмениваются сообщениями). Прямоугольник, вписанный в овал, обозначает реализацию данного класса при помощи явного автомата. В предлагаемой реализации только объект класса автомат-хаб имеет единственный экземпляр (центр передачи сообщений), остальные классы объектов могут иметь несколько одновременно выполняющихся экземпляров.

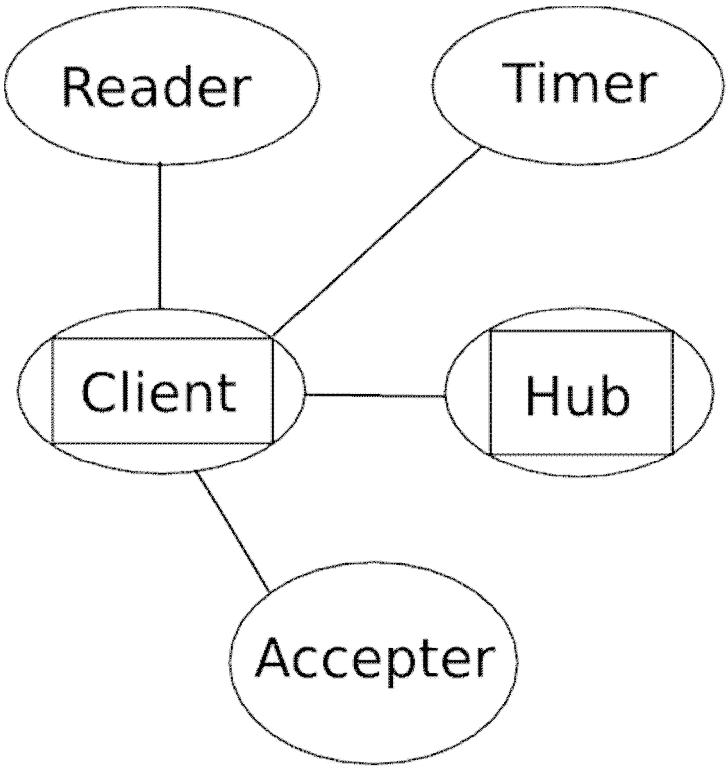


Рис. 3. Схема зависимостей классов объектов сервера обмена мгновенными сообщениями. «Accepter» – обработчик запросов на подключение, «Reader» – держатель TCP-сокета, «Timer» – таймер, «Hub» – автомат-хаб, «Client» – автомат-клиент

Для реализации передачи сообщений между объектами внутри сервера была реализована типобезопасная обертка (модуль *Util.Bus*) над модулем *Control.Concurrent.STM*, позволяющая разделять каналы на две части. Из первой части можно только читать, во вторую – только писать. В практическом применении данная обертка значительно облегчает отладку программ, часто использующих *STM*-каналы.

Библиотечный код из предыдущей главы вынесен в модуль *Control.FiniteState*.

Листинг 4. Main

```
import Util.Bus

import Object.Accepter
import Object.Hub

import Network
import Control.Concurrent

main = do
    (hubin, hubout) <- newPipe
    forkIO $ hubRun hubin
    sock <- listenOn $ PortNumber 4111
    accepterRun sock hubout
```

Листинг 5. Control.FiniteState

```
module Control.FiniteState where

-- Тип функции переходов.
type StateTransition state event output = state -> event ->
        (state, [output])

-- Тип функции переходов, пригодной для использования вместе с
        foldl.
type FoldableTransition state event output = (state, [output])
        -> event -> (state, [output])

-- Преобразователь из StateTransition в FoldableTransition
stt2foldable :: StateTransition state event output ->
        FoldableTransition state event output
stt2foldable transition (pstate, accumulator) event = (nstate,
        accumulator ++ output)
    where (nstate, output) = transition pstate event

-----------------------------------------------------
-- Комбинатор <<просто перейти в состояние state>>.
pure :: state -> (state, [output])
pure state = (state, [])

-- Комбинатор <<перейти в состояние state с одним выходным
        воздействием output>>.
blot :: state -> output -> (state, [output])
blot state output = (state, [output])

-- Комбинатор <<перейти в состояние state со списком выходных
        воздействий output>>.
dark :: state -> [output] -> (state, [output])
dark state output = (state, output)
```

```
-------------------------------------------------
class AutomataState state where
    lastState :: state -> Bool

-- Структура, представляющая отдельный активный автомат.
data IOAutomata source state event output = IOAutomata
    (source -> IO (source, [event])) -- Получение событий из
        внешнего мира.
    (state -> event -> (state, [output])) -- Функция переходов.
    (source -> output -> IO source) -- Функция осуществления
        выходных воздействий.
    (source, state) -- Стартовое состояние системы.

-- Исполняет активный автомат.
runIOAutomata (IOAutomata ep stt og psystem) =
    runIOAutomata' ep stt og psystem

runIOAutomata' ep stt og (psource, pstate) = do
    (xsource, events) <- ep psource
    let (nstate, outputs) = foldl (stt2foldable stt) (pure
        pstate) events
    nsource <- foldlM og xsource outputs
    if lastState nstate
        then return ()
        else runIOAutomata' ep stt og (nsource, nstate)
    where
        foldlM _ s [] = return s
        foldlM ofunc s (o:os) = do
            ns <- ofunc s o
            foldlM ofunc ns os
```

Листинг 6. Util.Bus

```
module Util.Bus
( InPipe(..), OutPipe(..), Bus(..)
, newPipe, newDup, newBus, otherEndOf
, inEndOf, outEndOf
, recvMessage, recvMessageSTM
, sendMessage, sendMessageSTM
, atomically, orElse, retry ) where

import System.IO
import Control.Exception (finally, catch)
import Control.Concurrent
import Control.Concurrent.STM

-------------------------------------------------------------
        ------
```

```
-- Classes
-----------------------------------------------------------------
        ------
class MessageBusRecv bus msg | bus -> msg where
    recvMessageSTM :: bus -> STM msg
    recvMessage :: bus -> IO msg
    recvMessage bus = atomically $ recvMessageSTM bus

class MessageBusSend bus msg | bus -> msg where
    sendMessageSTM :: bus -> msg -> STM ()
    sendMessage :: bus -> msg -> IO ()
    sendMessage bus msg = atomically $ sendMessageSTM bus msg

class MessageBusOtherEnd this that | this -> that where
    otherEndOf :: this -> that

-----------------------------------------------------------------
        ------
-- Types
-----------------------------------------------------------------
        ------
data InPipe  a = InPipe (TChan a)
data OutPipe a = OutPipe (TChan a)
data Bus i o = Bus (InPipe i) (OutPipe o) deriving Show

instance Show (InPipe a) where
    show (InPipe _) = "InPipe"

instance Show (OutPipe a) where
    show (OutPipe _) = "OutPipe"
-----------------------------------------------------------------
        ------
-- Instances
-----------------------------------------------------------------
        ------
instance MessageBusRecv (InPipe a) a where
    recvMessageSTM (InPipe me) = readTChan me

instance MessageBusSend (OutPipe a) a where
    sendMessageSTM (OutPipe me) msg = writeTChan me msg

instance MessageBusOtherEnd (InPipe a) (OutPipe a) where
    otherEndOf (InPipe a) = OutPipe a

instance MessageBusOtherEnd (OutPipe a) (InPipe a) where
    otherEndOf (OutPipe a) = InPipe a

instance MessageBusRecv (Bus i o) i where
```

```
    recvMessageSTM (Bus me _) = recvMessageSTM me

instance MessageBusSend (Bus i o) o where
    sendMessageSTM (Bus _ me) msg = sendMessageSTM me msg

instance MessageBusOtherEnd (Bus i o) (Bus o i) where
    otherEndOf (Bus a b) = Bus (otherEndOf b) (otherEndOf a)

inEndOf  (Bus a b) = a
outEndOf (Bus a b) = b
------------------------------------------------------------------
        ------
-- Functions
------------------------------------------------------------------
        ------
newPipe :: IO (InPipe a, OutPipe a)
newPipe = do
    x <- atomically $ newTChan
    return $ (InPipe x, OutPipe x)

newBus :: IO (Bus a b, Bus b a)
newBus = do
    (xi, xo) <- newPipe
    (yi, yo) <- newPipe
    return $ (Bus xi yo, Bus yi xo)

newDup (OutPipe a) = do
    x <- atomically $ dupTChan a
    return $ InPipe x
```

Листинг 7. Object.Accepter

```
module Object.Accepter where

import Prelude hiding (catch)
import System.IO
import Control.Exception (finally, catch)
import Control.Concurrent
import Network.Socket

import Util.Bus
import Object.Client as C

accepterRun sock chan =
    (sequence_ $ repeat $ do
        accepted <- accept sock
        forkIO $ C.clientRun accepted chan
        return ())
    `catch` (const $ return ())
```

```
    `finally` (do
        sClose sock
        return ())
```

Листинг 8. Object.Client

```
module Object.Client where

import qualified Object.Reader as R
import qualified Object.Timer as T

import Control.Concurrent
import Control.FiniteState
import Util.Bus

data Hub2Client = Die | Send String deriving Show
data Client2Hub = Dead | Alive (OutPipe Int) (OutPipe
        Hub2Client) | Got String deriving Show

data ClientSources = ClientSources
    { myID :: Int
    , hubBus  :: Bus Hub2Client (Int, Client2Hub)
    , readBus :: Bus R.Reader2Client R.Client2Reader
    , timerBus :: Maybe (Bus T.Timer2Client T.Client2Timer)}
        deriving Show

data ClientState = WaitForAuth | Working String | Stoped
        deriving Show
data ClientEvent = MustDie | GotLine String | SendLine String
        deriving Show
data ClientOutput = SayDead | SayMessage String | PushMessage
        String | StopTimer deriving Show

instance AutomataState ClientState where
    lastState Stoped = True
    lastState _ = False

clientRun (socket, info) chan = do
    (anspipeme, anspipeit) <- newPipe
    (idpipeme, idpipeit) <- newPipe
    sendMessage chan $ (0, Alive idpipeit anspipeit)
    cid <- recvMessage idpipeme

    (readerme, readerit) <- newBus
    forkIO $ R.readerRun socket readerit

    (timerme, timerit) <- newBus
    --forkIO $ T.timerRun 60000 timerit
```

```
    runIOAutomata $ IOAutomata
        getEvents
        switchFunc
        doOutputs
        (ClientSources
            { myID = cid
            , hubBus = Bus anspipeme chan
            , readBus = readerme
            , timerBus = Just $ timerme},
        WaitForAuth)

data MessageSource = FromReader R.Reader2Client | FromTimer
        T.Timer2Client | FromHub Hub2Client
getEvents state = do
    msg <- atomically $ foldl orElse retry $
        [ FromReader `fmap` (recvMessageSTM $ readBus state)
        , FromHub    `fmap` (recvMessageSTM $ hubBus state) ]
        ++ (case timerBus state of
            Just timerme -> [FromTimer  `fmap` (recvMessageSTM
        $ timerme)]
            Nothing -> [])
    print $ mapEvent msg
    return $ dark state $ mapEvent msg
    where
        mapEvent m = case m of
            FromReader (R.GotThat str) -> [GotLine str]
            FromHub (Send str) -> [SendLine str]
            FromReader R.Dead -> [MustDie]
            FromHub Die -> [MustDie]
            FromTimer T.Timeout -> [MustDie]
            _ -> []

switchFunc s e = case s of
    WaitForAuth -> case e of
        MustDie -> die
        GotLine str -> let (c, p) = splitAt 3 str in
            if c == "USR"
                then blot (Working $ tail p) StopTimer
                else die
        SendLine str -> pure s
    Working name -> case e of
        MustDie -> die
        GotLine str -> blot s $ SayMessage $ "<" ++ name ++ ">
        " ++ str
        SendLine str -> blot s $ PushMessage str
    Stoped -> error "Must not"
    where die = blot Stoped SayDead
```

```
doOutputs state output = case output of
    SayDead -> do
        toRead R.Die
        toHub Dead
        return state
    StopTimer -> do
        case timerBus state of
            Just timerme -> sendMessage timerme T.Die
            Nothing -> return ()
        return $ state {timerBus = Nothing}
    SayMessage str -> toHub (Got str) >> return state
    PushMessage str -> toRead (R.SendThat str) >> return state
    where
        toHub x = sendMessage (hubBus state) (myID state, x)
        toRead x = sendMessage (readBus state) x
```

Листинг 9. Object.Hub

```
module Object.Hub where

import qualified Object.Client as C

import Control.Concurrent
import Control.FiniteState
import Util.Bus

import qualified Data.Map as M

data HubSources = HubSources
    { channel :: InPipe (Int, C.Client2Hub)
    , clients :: M.Map Int (OutPipe C.Hub2Client)
    , lastID :: Int } deriving Show

data HubState = Running | Stoped deriving Show
data HubEvent = NewClient Int | DeadClient Int | GotMessage Int
        String deriving Show
data HubOutput = Send2All String | Send2ID Int String deriving
        Show

instance AutomataState HubState where
    lastState Stoped = True
    lastState _ = False

hubRun chan = do
     runIOAutomata $ IOAutomata
        getEvents
        switchFunc
        doOutputs
```

```
        (HubSources { channel = chan, clients = M.empty, lastID
        = 0 }, Running)

getEvents s = do
    (id, msg) <- recvMessage $ channel s
    case msg of
        C.Dead -> return $ blot s $ DeadClient id
        C.Alive i p -> do
            let nid = (lastID s) + 1
            sendMessage i $ nid
            return $ blot (s
                { clients = M.insert nid p (clients s)
                , lastID = nid }) $ NewClient nid
        C.Got str -> return $ blot s $ GotMessage id str

switchFunc Stoped _ = pure Stoped
switchFunc Running e = case e of
    NewClient i  -> blot Running $ Send2All $ show e
    DeadClient i -> blot Running $ Send2All $ show e
    GotMessage i s -> blot Running $ Send2All $ (show i) ++ ":
        " ++ s

doOutputs s output = case output of
    Send2All str -> do
        sequence_ $ map (\x -> sendMessage x $ C.Send str) $
        M.elems $ clients s
        return s
    Send2ID id str -> case M.lookup id $ clients s of
        Nothing -> return s
        Just a -> sendMessage a (C.Send str) >> return s
```

Листинг 10. Object.Reader

```
module Object.Reader where

import Prelude hiding (catch)
import System.IO
import Control.Exception (finally, catch)
import Control.Concurrent
import Network.Socket

import Util.Bus

data Client2Reader = Die | SendThat String deriving Show
data Reader2Client = Dead | GotThat String deriving Show

filterl [] = []
filterl ('\n':str) = filterl str
filterl ('\r':str) = filterl str
```

```
filterl (o:str) = o:(filterl str)

readerOutRun handle chan =
    (sequence_ $ repeat $ do
        line <- hGetLine handle
        sendMessage chan $ GotThat $ filterl line)
    `catch` (const $ return ())
    `finally` (do
        hClose handle
        sendMessage chan $ Dead)

readerInRun handle chan =
    (sequence_ $ repeat $ do
        msg <- recvMessage chan
        case msg of
            SendThat line -> do
                hPutStrLn handle (line ++ "\r")
                hFlush handle
            Die -> hClose handle)
    `catch` (const $ return ())

readerRun sock (Bus i o) = do
    handle <- socketToHandle sock ReadWriteMode
    forkIO $ readerOutRun handle o
    readerInRun handle i
```

Листинг 11. Object.Timer

```
module Object.Timer where

import Prelude hiding (catch)
import System.IO
import Control.Exception
import Control.Concurrent
import Data.Typeable

import Util.Bus

data Client2Timer = Die deriving Show
data Timer2Client = Dead | Timeout deriving Show

timerOutRun timeout chan =
    (do
        threadDelay timeout
        sendMessage chan $ Timeout)
    `catch` (const $ return ())
    `finally` (sendMessage chan $ Dead)

timerInRun out chan = do
```

```
    recvMessage chan
    throwTo out Deadlock
    return ()

timerRun timeout (Bus i o) = do
    tid <- forkIO $ timerOutRun timeout o
    timerInRun tid i
```

1.2. Разработка, программная реализация и экспериментальное исследование методов рефакторинга автоматов при разработке автоматных программ

В данном разделе приводятся разработка, программная реализация и экспериментальное исследование методов рефакторинга автоматов при разработке автоматных программ, выполненные в рамках второго этапа настоящей работы.

1.2.1. Внесение изменений в автоматные программы

Какой бы продуманной ни была архитектура программной системы, при изменении требований часто приходится модифицировать ее код. При произвольном изменении программы велик риск внесения ошибок. Одним из распространенных приемов, используемых при модификации кода, является *рефакторинг* – полное или частичное преобразование структуры программы с сохранением ее поведения [2]. Примерами рефакторингов в объектно-ориентированных программах являются переименование класса, выделение интерфейса, перемещение метода и т.д.

Каждый вид рефакторинга реализует определенное изменение структуры программы, состоящее из набора небольших и технически простых шагов, позволяющих сделать процесс внесения изменений контролируемым. Следование правилам выполнения рефакторинга обеспечивает неизменность поведения программы, в чем можно убедиться, проведя процедуру тестирования. В последнее время широкое распространение получили средства автоматического выполнения рефакторингов. Поддержка рефакторинга в каком-либо виде является частью практически любой современной интегрированной среды разработки. Автоматизация рефакторинга позволяет свести вероятность изменения поведения программы при его выполнении практически к нулю.

Использование рефакторинга позволяет организовать процесс изменения поведения программы следующим образом: сначала производится рефакторинг для подготовки к изменению, затем выполняется само изменение поведения. При этом рефакторинг выбирается таким образом, чтобы минимизировать модификации кода, изменяющие поведение программы.

Любая программа, используемая длительное время, подвергается модификации. Это связано с рядом естественных причин: в ходе эксплуатации программы могут выявиться требования, которые не были очевидны изначально, а также могут обнаружиться ошибки в работе программы. Наконец, для проведения описанных изменений может возникнуть потребность изменить структуру программы с целью ее упрощения. Опытные программисты знают, что хорошую структуру удается создать не сразу – она должна развиваться по мере накопления опыта [3]. Поэтому почти любая программная программа рано или поздно подвергается изменениям.

С появлением большого числа программ с явным выделением состояний возникает необходимость в поддержке таких программ. Из сказанного выше следует, что в существующую программу вносят изменения следующих типов:

- изменения в программе в соответствии с изменившимися требованиями к ней;
- исправление ошибок;

- изменения в программе, имеющие целью облегчить понимание ее работы и упростить модификацию, не затрагивая наблюдаемого поведения.

По аналогии с существующим понятием *рефакторинга* [2] объектно-ориентированных программ, будем называть изменения последнего типа, применяемые к программам с явным выделением состояний, *рефакторингом автоматов*.

Внесение любых изменений в работающую программу сопряжено с риском появления ошибок и грозит потерей надежности программы. Вместе с тем, надежность часто является важнейшим требованием, предъявляемым к программе с явным выделением состояний. Поэтому исследования, направленные на разработку подхода к безопасному внесению изменений в такие программы, являются актуальными. Эти исследования позволяют упростить процесс разработки и повысить качество создаваемых программ.

Будем называть автомат *синтаксически корректным*, если он проходит валидацию. Важно отметить, что синтаксически корректный автомат удовлетворяет требованиям полноты и непротиворечивости: множество исходящих переходов для любого состояния полно и непротиворечиво. Это означает, что при обработке любого события выполняется ровно один переход.

Будем называть автомат *семантически корректным*, если его выполнение согласовано со спецификацией. Заметим, что спецификация может быть как неформальной – например, задавать требования словесно, так и формальной – например, задаваться с помощью темпоральной логики. При этом выполнение спецификации, заданной формально, в некоторых случаях можно *верифицировать*.

Синтаксически и семантически корректный автомат будем называть *корректным*. Безопасными изменениями будем называть такие изменения автоматов, которые сохраняют их корректность.

К сожалению, далеко не все изменения являются безопасными. Часто разработчики, столкнувшиеся с необходимостью внесения изменений в программу с явным выделением состояний, вынуждены проектировать ее заново или бессистемно вносить изменения, руководствуясь только собственным пониманием. Такие подходы являются неэффективными и ненадежными, так как даже простейшие изменения влияют на корректность автоматов и могут привести к появлению трудно находимых ошибок. Например, добавление одного перехода между двумя состояниями автомата может нарушить непротиворечивость множества переходов автомата.

1.2.2. Классификация изменений автоматных программ

Часто изменения, вносимые в программу с явным выделением состояний, достаточно сложны, а потому порождают массу проблем, плохо поддающихся анализу. С другой стороны, существует набор *базовых* изменений, которые являются «примитивными» изменениями какой-то одной составляющей графа переходов автомата. Такие изменения достаточно хорошо поддаются описанию. Остальные, более сложные изменения, назовем *составными*. Составные изменения можно представить в виде набора базовых изменений.

В отдельный класс выделим *рефакторинги автоматов*. Как было сказано выше, рефакторинги не меняют поведение программы и применяются для улучшения ее структуры.

1.2.3. Описание базовых изменений автоматов

В данном разделе приводится каталог базовых изменений автоматов. На основе этих изменений строятся более сложные сценарии реконфигурации автомата. Базовые изменения могут быть деструктивными и нарушать полноту и непротиворечивость графа переходов, а также семантические свойства. Для каждого базового изменения приводятся рекомендации по проведению такого изменения.

При описании каждого базового изменения графа переходов автомата будем придерживаться определенного формата, приведенного ниже:

- *название* изменения;
- *неформальное описание* приводимого изменения;
- *рекомендации* по проведению такого изменения.

1.2.3.1. Добавление состояния

Описание. В граф переходов добавляется новое состояние.

Рекомендации.

- Обеспечить достижимость состояния.

1.2.3.2. Удаление состояния

Описание. Из графа переходов удаляется состояние и все связанные с этим состоянием переходы.

Рекомендации.

- Если удаляемое состояние является начальным, необходимо выбрать новое начальное состояние.
- Если удаляемое состояние является конечным, необходимо пересмотреть набор конечных состояний.
- Следует также учитывать описанные ниже рекомендации при удалении перехода.

1.2.3.3. Установка начального состояния

Описание. Состояние объявляется начальным в автомате.

Рекомендации.

- Необходимо проверить, что нет других начальных состояний.

1.2.3.4. Снятие начального состояния

Описание. Начальное состояние объявляется нормальным.

Рекомендации.

- Необходимо объявить начальным другое состояние.

1.2.3.5. Добавление конечного состояния

Описание. Нормальное состояние объявляется конечным в автомате.

Рекомендации.

- Если из этого состояния выходили какие-то переходы, то их необходимо удалить (следуя рекомендациям при удалении перехода).

1.2.3.6. Удаление конечного состояния

Описание. Конечное состояние объявляется нормальным.

Рекомендации.

- Необходимо пересмотреть набор конечных состояний в автомате.

1.2.3.7. Добавление перехода

Описание. Добавление перехода между двумя состояниями. На переходе указывается событие, по которому данный переход осуществляется. Также может указываться условие и выходное воздействие.

Рекомендации.

- Необходимо проверить полноту и непротиворечивость условий на переходах.

1.2.3.8. Изменение события на переходе

Описание. Изменяется событие, по которому активизируется переход.

Рекомендации.

- Проверить полноту и непротиворечивость условий на переходах.

1.2.3.9. Изменение условия на переходе

Описание. Изменяется условие, при котором осуществляется переход.

Рекомендации.

- Проверить полноту и непротиворечивость условий на переходах.

1.2.3.10. Удаление перехода

Описание. В автомате удаляется переход.

Рекомендации.

- Проверить полноту и непротиворечивость условий на переходах.
- Проверить достижимость состояний.

1.2.3.11. Перемещение перехода

Описание. Переход имеет начальное и конечное состояния. При перемещении перехода возможно:

- изменение начального состояния;
- изменение конечного состояния.

Данное изменение формально не является базовым, так как его можно разбить на более простые изменения. Тем не менее, оно рассматривается нами, поскольку перемещение перехода – это регулярно используемая модификация.

Рекомендации.

- Проверить полноту и непротиворечивость условий на переходах.
- Проверить достижимость состояний в автомате.

1.2.4. Рефакторинг автоматных программ

В данном разделе описывается каталог рефакторингов автоматных программ – изменений программ, не меняющих их поведение, но улучшающих их структуру.

Выбор рефакторингов для каталога основан на наборе экспериментов. В качестве основы были взяты несколько автоматов, созданных студентами кафедры «Компьютерные технологии» СПбГУ ИТМО в ходе выполнения курсовых работ по курсу «Применение автоматов в программировании» [4]. Далее производились изменения требований к автомату, основанные на анализе предметной области. В соответствии с изменениями требовалось изменить структуру автомата. При этом часто даже простые изменения спецификации требовали значительной модификации графа переходов, так

как его исходная структура оказывалась неприспособленной к внедряемому изменению. Для обеспечения совместимости структуры автомата и изменения спецификации требовалось произвести рефакторинг: модифицировать структуру автомата таким образом, чтобы изменения, вызванные изменившейся спецификацией, естественно вписались в новую структуру. На основе таких изменений и был разработан нижеприведенный каталог рефакторингов автоматных программ.

Описание каждого рефакторинга имеет следующую структуру:

- сначала следует *название* рефакторинга;
- за названием следует *неформальное описание* изменения;
- *мотивация* описывает, почему следует пользоваться этим методом рефакторинга;
- *пример* применения рефакторинга;
- *техника* содержит описание пошагового выполнения рефакторинга;
- *доказательство* корректности рефакторинга.

В конце описания каждого рефакторинга приводится доказательство его корректности: формально доказывается, что выполнение рефакторинга не меняет логику работы автомата. Это означает, что после рефакторинга должен получиться автомат, эквивалентный исходному. Эквивалентными называются такие автоматы, которые на любую последовательность входных воздействий реагируют одинаковыми последовательностями выходных воздействий.

Будем рассматривать последовательность событий e_{i_1}, e_{i_2}, ..., e_{i_k}. При доказательстве эквивалентности автоматов будем придерживаться следующих обозначений: A – изначальный автомат, A' – автомат с внесенными изменениями. Пусть при этом *цепочка элементов* – последовательность состояний, событий и выходных воздействий автомата A имеет вид: s_{j_0}, z_{k_0}, e_{i_1}, z_{l_1}, s_{j_1}, z_{k_1}, e_{i_2}, z_{l_2}, s_{j_2}, z_{k_2}, …, а аналогичная последовательность для автомата A' – s'_{j_0}, z'_{k_0}, e_{i_1}, z'_{l_1}, s'_{j_1}, z'_{k_1}, e_{i_2}, z'_{l_2}, s'_{j_2}, z'_{k_2}, …, где:

- s_{j_t} и s'_{j_t} – состояния,
- z_{k_t} и z'_{k_t} – выходные воздействия, совершаемые при входе в состояния s_{j_t} и s'_{j_t} соответственно,
- z_{l_t} и z'_{l_t} – выходные воздействия, совершаемые на переходах после события e_{i_t}.

Если автоматы A и A' не эквивалентны, то найдется такое $t > 0$, что $z_{k_t} \neq z'_{k_t}$ или $z_{l_t} \neq z'_{l_t}$. При доказательстве эквивалентности автоматов будем опираться на технику выполнения рефакторинга.

1.2.4.1. Группировка состояний

Описание. Несколько простых состояний объединяются в группу состояний. При этом добавляются групповые переходы, заменяющие одинаковые переходы, исходящие из всех группируемых состояний (рис. 4).

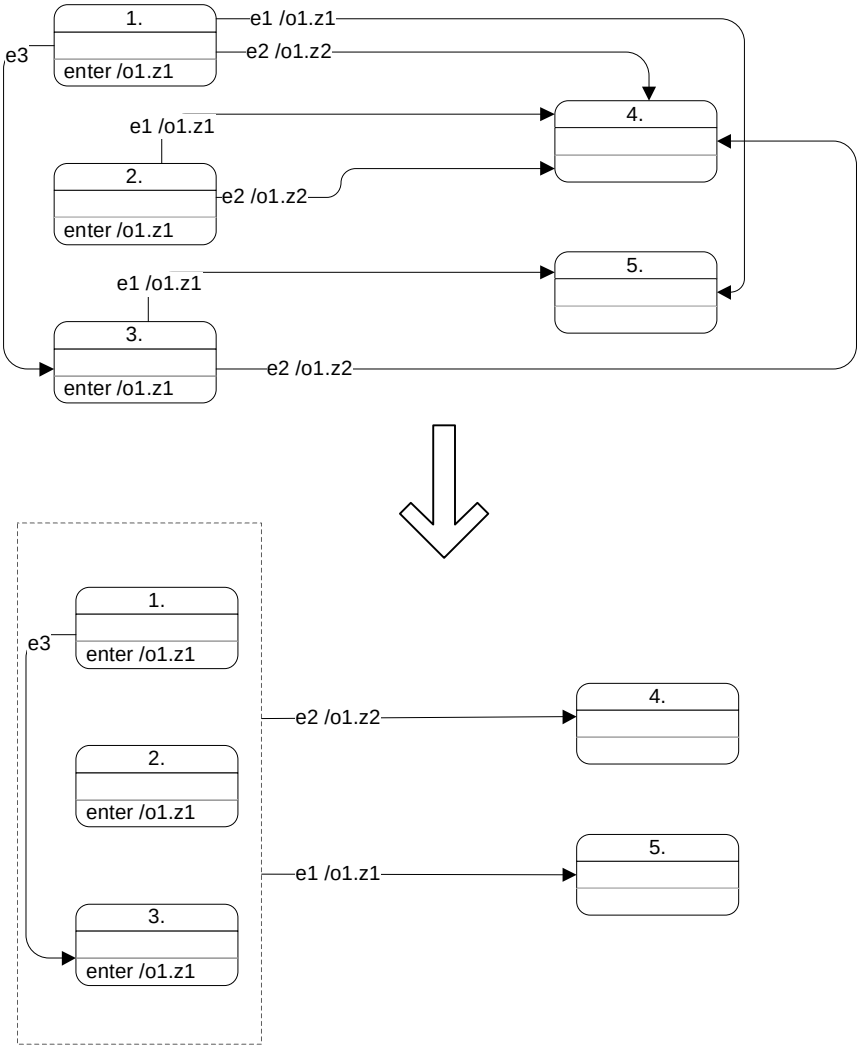


Рис. 4. Пример применения рефакторинга «группировка состояний»

Состояния, объединяемые в группу, могут иметь по несколько одинаковых переходов, в этом случае добавляется несколько групповых переходов.

Мотивация. Часто автомат имеет несколько состояний, которые имеют одинаковые переходы в другие состояния автомата. В этом случае можно выделить группу состояний, упростив структуру автомата.

Пример. Рассмотрим фрагмент графа переходов автомата «Панель в кабине лифта», отвечающий за выключение ламп в кнопках. Автомат имеет следующие пять состояний:

0. Кнопки погашены.
1. Светится «1».
2. Светится «2».
3. Светится «3».
4. Светится «S».

При наступлении события *e* («Выключение лампы в кнопке») в каждом из состояний 1–4 автомат должен перейти в состояние 0. Такое поведение реализуется следующим графом переходов (рис. 5).

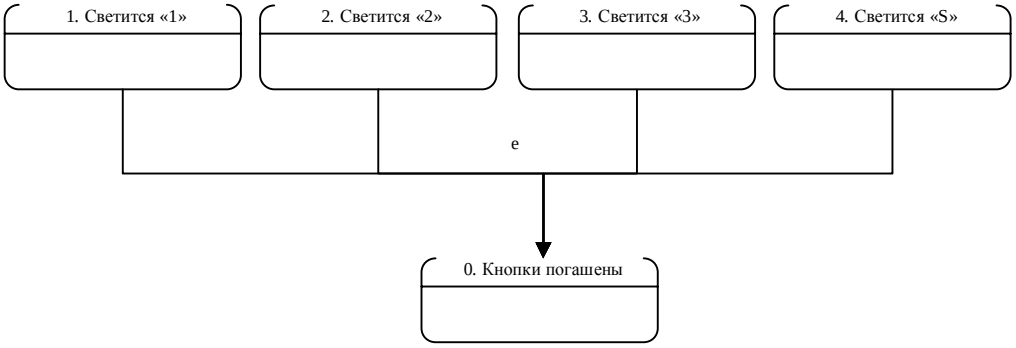


Рис. 5. Граф переходов без группировки состояний

Так как переходы из состояний 1–4 идентичны, их можно сгруппировать и соответствующим образом изменить конфигурацию автомата (рис. 6).

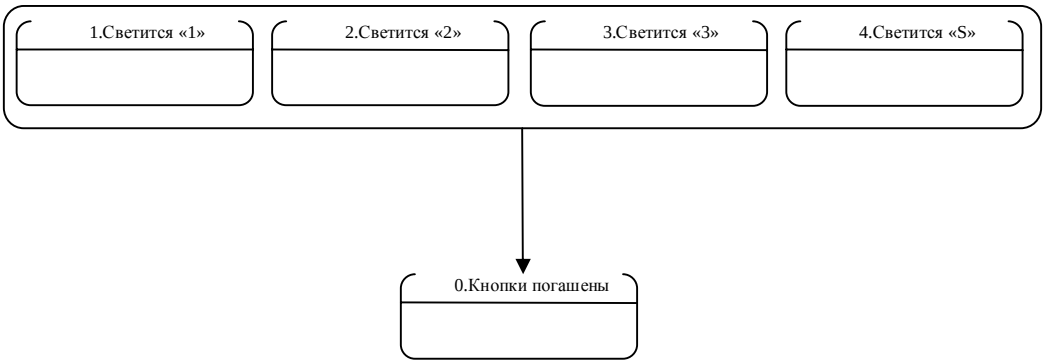


Рис. 6. Граф переходов с группировкой состояний

Техника. Для объединения состояний s_1, s_2, ..., s_k в группу необходимо выполнить следующие действия:

1. Добавить группу *g*, объединяющую состояния s_1, s_2, ..., s_k.
2. Выбрать один переход *t*, исходящий из s_1, который требуется заменить групповым.
3. Убедиться, что каждое из состояний s_2, ..., s_k имеет переход с такими же атрибутами, что и *t* (под атрибутами понимаются конечное состояние, событие, условие и выходные воздействия).
4. Добавить переход, исходящий из группы *g*, и имеющий те же атрибуты, что и *t*.
5. Удалить переходы, отмеченные в пунктах 2, 3.
6. Для оставшихся переходов, которые нужно заменить групповыми, повторить шаги 2–6.

Доказательство корректности. Так как групповой переход идентичен переходу из каждого состояния, измененный автомат *A*' будет иметь такую же цепочку элементов s'_{j_1}, z'_{k_1}, e_{i_2}, z'_{l_2}, s'_{j_2}, z'_{k_2}, …, что и автомат *A*. Поэтому они будут эквивалентны.

1.2.4.2. Удаление группы состояний

Описание. При удалении группы состояний все исходящие из него переходы добавляются в состояния, находящиеся в группе, после чего сгруппированные состояния выносятся из группы.

Мотивация. Такое изменение полезно для последующей модификации автомата, если одно из состояний, входящее в удаляемую группу, изменяет логику поведения.

Техника. Для удаления группы g, объединяющей состояния s_1, s_2, ..., s_k, необходимо выполнить следующие действия:

1. Выбрать переход t, исходящий из группы g.
2. Добавить переходы, исходящие из состояний s_1, s_2, ..., s_k, с атрибутами перехода t.
3. Удалить переход t.
4. Для оставшихся переходов, исходящих из группы g, повторить шаги 1–4.
5. Удалить группу g.

Доказательство корректности. Аналогично предыдущему рефакторингу.

1.2.4.3. Слияние состояний

Описание. Несколько состояний с одинаковыми исходящими переходами сливаются в одно состояние. При этом сохраняются переходы, соединяющие эти состояния с другими состояниями автомата.

Производить слияние состояний можно только в том случае, если одинаковы атрибуты переходов, исходящих из этих состояний, и одинаковы воздействия, вызываемые при входе в сливаемые состояния.

Мотивация. В процессе изменения программы может оказаться, что некоторые состояния дублируют логику поведения программы. В этом случае эти состояния могут быть заменены одним состоянием.

Пример. Рассмотрим пример, иллюстрирующий применение рефакторинга «слияние состояний». Для этого рассмотрим модификацию примера «Панель в кабине лифта», предложенного в разделе 1.2.4.1, с добавлением следующих состояний:

5. Перегорела лампа в кнопке «1»;
6. Перегорела лампа в кнопке «2»;
7. Перегорела лампа в кнопке «3»;
8. Перегорела лампа в кнопке «S»;
9. Неисправность.

Измененный граф переходов представлен на рис. 7:

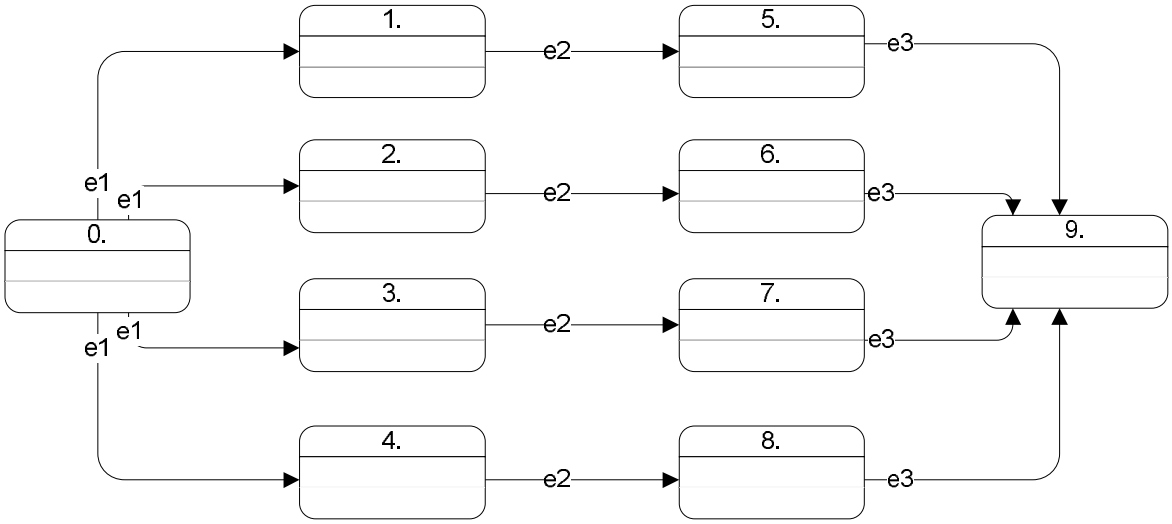


Рис. 7. Граф переходов автомата управления лифтом до слияния состояний 5–8

Граф переходов автомата после слияния состояний 5–8 в одно состояние представлен на рис. 8.

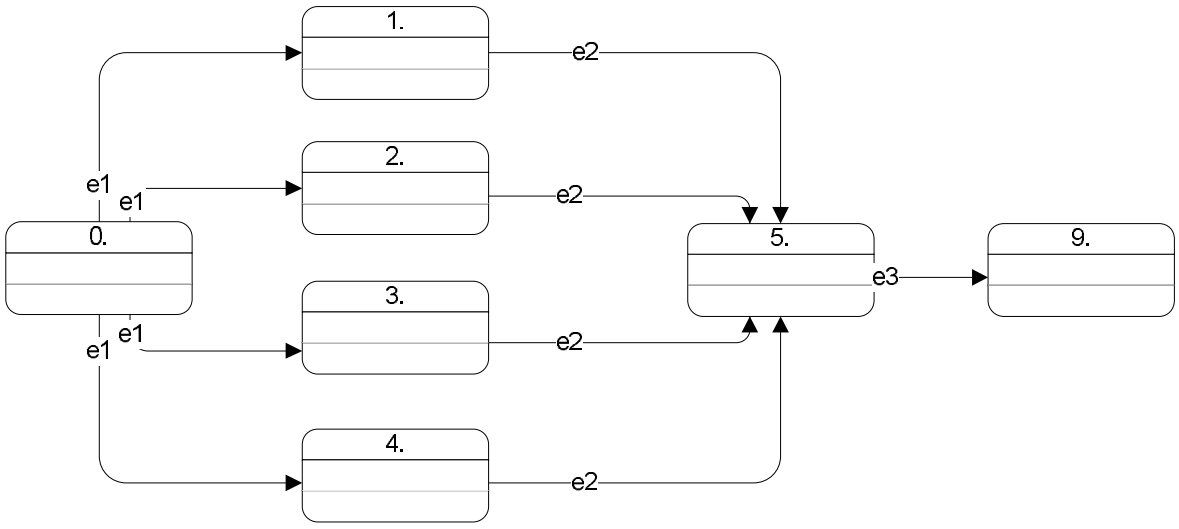


Рис. 8. Граф переходов после слияния состояний

Техника. Для слияния состояний s_1, s_2, ..., s_k необходимо выполнить следующие действия:

1. Убедиться, что состояния s_1, s_2, ..., s_k имеют одинаковые воздействия, вызываемые при входе в состояние.
2. Убедиться, что состояния s_1, s_2, ..., s_k имеют исходящие переходы с одинаковыми атрибутами, и эти переходы заканчиваются в одном и том же состоянии.
3. Изменить конечные состояния переходов, которые ведут в состояния s_2, ..., s_k, на состояние s_1.
4. Удалить состояния s_2, ..., s_k.

Доказательство корректности. Между состояниями автоматов *A* и *A*' строится соответствие $f : S \to S'$. При этом все сливаемые состояния переходят в одно состояние s_1': $f(s_1) = s_1'$, $f(s_2) = s_1'$, ..., $f(s_k) = s_1'$.

Начальное состояние автомата *A* соответствует начальному состоянию автомата *A*'. Тогда в цепочках элементов s_{j_0}, z_{k_0}, e_{i_1}, z_{l_1}, s_{j_1}, z_{k_1}, e_{i_2}, z_{l_2}, s_{j_2}, z_{k_2}, … и s'_{j_0}, z'_{k_0}, e_{i_1}, z'_{l_1}, s'_{j_1}, z'_{k_1}, e_{i_2}, z'_{l_2}, s'_{j_2}, z'_{k_2}, … первые состояния s_{j_0} и s'_{j_0} соответствуют друг другу: $f(s_{j_0}) = s'_{j_0}$ и $z_{k_0} = z'_{k_0}$.

Докажем, что если в цепочках совпадают первые *t* (*t* > 0) элементов, то (*t*+1)-е элементы также совпадают. Другими словами, если выполняется: $f(s_{j_x}) = s'_{j_x}$, $z_{k_x} = z'_{k_x}$, $z_{l_x} = z'_{l_x}$ для всех $x \le t$, то также верно, что $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$. Состояние s_{j_t} может входить или не входить в множество сливаемых состояний s_1, s_2, ..., s_k. Рассмотрим оба случая:

1) s_{j_t} *не входит* в множество сливаемых состояний:
   a) $s_{j_{t+1}}$ входит в множество сливаемых состояний s_1, s_2, ..., s_k. Тогда $s_{j_{t+1}} = s_i$ для некоторого *i*. Получаем $f(s_{j_{t+1}}) = f(s_i) = s_1' = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$ (см. пункт 3 техники), $z_{l_{t+1}} = z'_{l_{t+1}}$ (см. п. 1 техники).
   b) $s_{j_{t+1}}$ не входит в множество сливаемых состояний s_1, s_2, ..., s_k. Тогда $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$, так как соответствующий переход и состояние не претерпевали никаких изменений.
2) s_{j_t} *входит* в множество сливаемых состояний:
   a) $s_{j_{t+1}}$ также входит в множество сливаемых состояний s_1, s_2, ..., s_k. Тогда $s_{j_{t+1}} = s_i$ для некоторого *i*. Получаем $f(s_{j_{t+1}}) = f(s_i) = s_1' = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$ (см. п. 3 техники выполнения рассматриваемого рефакторинга), $z_{l_{t+1}} = z'_{l_{t+1}}$ (см. п. 1 техники).
   b) $s_{j_{t+1}}$ не входит в множество сливаемых состояний s_1, s_2, ..., s_k. Но так как s_1, s_2, ..., s_k имеют исходящие переходы с одинаковыми атрибутами, в том числе с одинаковыми конечными состояниями (см. п. 2 техники), то после события $e_{i_{t+1}}$ автомат *A*' перейдет в состояние $s'_{j_{t+1}} = f(s_{j_{t+1}})$. Тогда $z'_{k_{t+1}} = z_{k_{t+1}}$ (выходное воздействие в состоянии $s_{j_{t+1}}$ осталось неизменным) и $z'_{l_{t+1}} = z_{l_{t+1}}$ (согласно п. 2 техники).

Корректность рассмотренного рефакторинга доказана.

1.2.4.4. Выделение автомата

Описание. Часть логики программы переносится в отдельный вызываемый автомат (рис. 9).

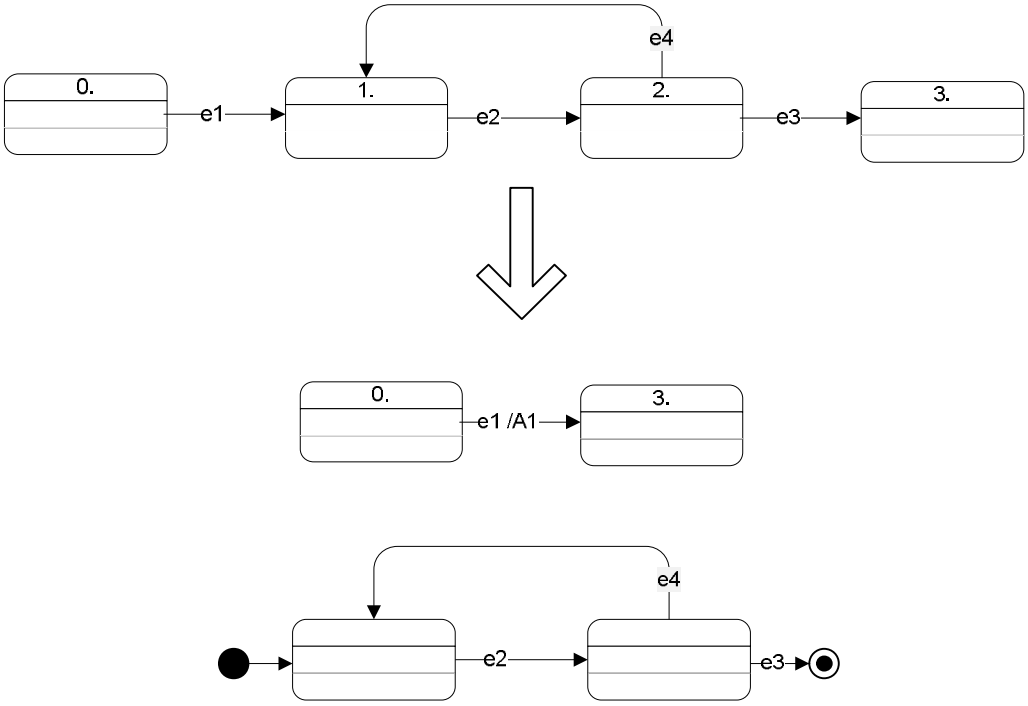


Рис. 9. Выделение автомата

Мотивация. Большие автоматы с множеством состояний, реализующие сразу несколько функций программы сложны в поддержке. При внесении изменений в работу одной из функций необходимо учитывать реализацию других функций. Понимание работы такого автомата усложнено тем, что протоколы взаимодействия частей автомата, отвечающих за реализацию различных функций, не определены четко. Соответственно, поддержка сложных автоматов может привести к постоянному возникновению ошибок, обнаружить которые и исправить гораздо сложнее, чем в программах, в которых различные функции реализованы отдельными автоматами.

Для решения этой проблемы сложный автомат разделяют на несколько более простых, для каждого из которых четко определены обязанности и протоколы взаимодействия с другими автоматами и остальной программой. Такое разделение называют *автоматной декомпозицией* [5]. Автоматная декомпозиция обычно производится на этапе анализа требований, когда происходит построение изначального набора автоматов, которые требуется реализовать в программе. Однако часто в ходе эволюции программной программы один из автоматов накапливает реализацию чрезмерно большой функциональности. В этом случае требуется выполнить автоматную декомпозицию в уже существующей программе. Существует несколько критериев автоматной декомпозиции, следуя которым в большинстве случаев можно получить логичную архитектуру программы.

Декомпозиция *по режимам* уместна тогда, когда в поведении программы можно выделить несколько качественно различных режимов (каждый из которых, при необходимости, можно конкретизировать, выделив режимы более низкого уровня абстракции). В этом случае логично сопоставить автомат каждому из режимов, в которых поведение программы является сложным, или, иными словами, сопоставить отдельный автомат каждому абстрактному действию.

Декомпозиция *по объектам управления* применима в том случае, когда в программе присутствует несколько объектов управления. В этом случае логично поручить управление каждым из объектов отдельному автомату или поддереву в иерархии автоматов.

Следование такому критерию декомпозиции приводит к выделению в архитектуре программы пар автомат – объект управления (или группа автоматов – объект управления) и значительно приближает ее к «идеалу» парадигмы автоматного программирования – взаимодействующих автоматизированных объектов управления.

Следует отметить, что техника реализации большинства рефакторингов не зависит от характера взаимодействия автоматов в программе. В некоторых случаях различные автоматы программы могут взаимодействовать непосредственно (через автоматные интерфейсы). В других, как, например, при следовании парадигме *автоматизированные объекты управления как классы*, автоматы взаимодействуют друг с другом так же, как с объектами управления – через входные и выходные переменные. Однако, техника выполнения рефакторингов по выделению и встраиванию автомата определенно зависит от того каким образом взаимодействуют автоматы в программе. Для решения этой проблемы техника описанных рефакторингов приводится в двух вариантах: для программ, в которых автоматы взаимодействуют через вызовы соседних автоматов и для программ, в которых автоматы взаимодействуют друг с другом так же, как и с объектами управления.

Техника. Для выделения состояний $s_1, s_2, ..., s_k$ и исходящих из них переходов $t_1, t_2, ..., t_l$ в вызываемый автомат необходимо выполнить следующие действия:

1. Убедиться, что среди состояний $s_1, s_2, ..., s_k$ есть ровно одно такое состояние s_i, в которое ведет хотя бы один переход из состояния, не входящего в множество $s_1, s_2, ..., s_k$. Обозначим отмеченный переход как t', а его начальное состояние – s'.
2. Убедиться, что из отмеченного состояния s_i достижимы все остальные состояния выбранного подмножества.
3. Убедиться, что переходы $t_1, t_2, ..., t_l$ имеют в качестве конечных состояний только состояния из множества $s_1, s_2, ..., s_k$ и ровно одно состояние s'', не входящее в это множество.
4. Создать новый автомат.
5. Создать в новом автомате состояния $s_1', s_2', ..., s_k'$, соответствующие состояниям $s_1, s_2, ..., s_k$, с сохранением выходных воздействий.
6. Добавить переходы между состояниями $s_1', s_2', ..., s_k'$ с такими же атрибутами, что и переходы между состояниями $s_1, s_2, ..., s_k$.
7. Объявить состояние s_i' начальным в созданном автомате.
8. Выбрать среди состояний $s_1, s_2, ..., s_k$ такие, из которых есть переходы в s''. Пусть в созданном автомате им соответствуют состояния $s_{p_1}', s_{p_2}', \ldots, s_{p_m}'$. Добавить из состояний $s_{p_1}', s_{p_2}', \ldots, s_{p_m}'$ переходы, ведущие в конечное состояние созданного автомата. Присвоить этим переходам такие же атрибуты, как у соответствующих переходов изначального автомата.
9. Добавить переход между состояниями s' и s'' с атрибутами перехода t' и вызовом созданного автомата.
10. Удалить состояния $s_1, s_2, ..., s_k$.

Доказательство корректности. Между состояниями автомата *A* и системы автоматов *A'* естественным образом строится взаимно однозначное соответствие $f: S \to S'$. При этом начальное состояние автомата *A* соответствует начальному состоянию автомата *A'*. Тогда в цепочках элементов

s_{j_0}, z_{k_0}, e_{i_1}, z_{l_1}, s_{j_1}, z_{k_1}, e_{i_2}, z_{l_2}, s_{j_2}, z_{k_2}, … и s'_{j_0}, z'_{k_0}, e_{i_1}, z'_{l_1}, s'_{j_1}, z'_{k_1}, e_{i_2}, z'_{l_2}, s'_{j_2}, z'_{k_2}, … первые состояния s_{j_0} и s'_{j_0} соответствуют друг другу: $f(s_{j_0}) = s'_{j_0}$ и $z_{k_0} = z'_{k_0}$.

Докажем, что если в цепочках совпадают первые t ($t > 0$) элементов, то (t+1)-е элементы также совпадают. Другими словами, если выполняется: $f(s_{j_x}) = s'_{j_x}$, $z_{k_x} = z'_{k_x}$, $z_{l_x} = z'_{l_x}$ для всех $x \le t$, то также верно, что $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$.

Рассмотрим несколько случаев:

1) s_{j_t} не входит в множество выделяемых в отдельный автомат состояний s_1, s_2, ..., s_k:
   a) $s_{j_t} = s'$. Тогда $s_{j_{t+1}} = s_i$ для некоторого i (см. п. 1 техники) или $s_{j_{t+1}}$ не входит в множество выделяемых в отдельный автомат состояний s_1, s_2, ..., s_k. В первом случае $f(s_{j_{t+1}}) = f(s_i) = s'_i = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$ (см. пункт 5 техники), $z_{l_{t+1}} = z'_{l_{t+1}}$ (см. п. 9 техники). Во втором же случае сразу получаем $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$, так как соответствующий переход и состояние не претерпевали никаких изменений.
   b) $s_{j_t} \ne s'$. Тогда $s_{j_{t+1}}$ не входит в множество выделяемых в отдельный автомат состояний s_1, s_2, ..., s_k, и переходы, ведущие из s_{j_t}, остались неизменными. Следовательно, $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$.
2) s_{j_t} входит в множество выделяемых в отдельный автомат состояний s_1, s_2, ..., s_k. Тогда либо $s_{j_{t+1}}$ также входит в множество состояний, выделяемых в отдельный автомат, либо $s_{j_{t+1}} = s''$ (см. пп. 3, 6, 8 техники):
   a) $s_{j_{t+1}}$ входит в множество выделяемых в отдельный автомат состояний s_1, s_2, ..., s_k. Тогда согласно пп. 5, 6 техники $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$.
   b) $s_{j_{t+1}} = s''$. Тогда из п. 8 техники получаем, что $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$.

Корректность рефакторинга доказана.

Пример. Выделение автомата подробно рассматривается в разд. 1.2.6.

1.2.4.5. Встраивание вызываемого автомата

Описание. Вызываемый автомат встраивается в места своего вызова на переходах (рис. 10). Рефакторинг является обратным к предыдущему.

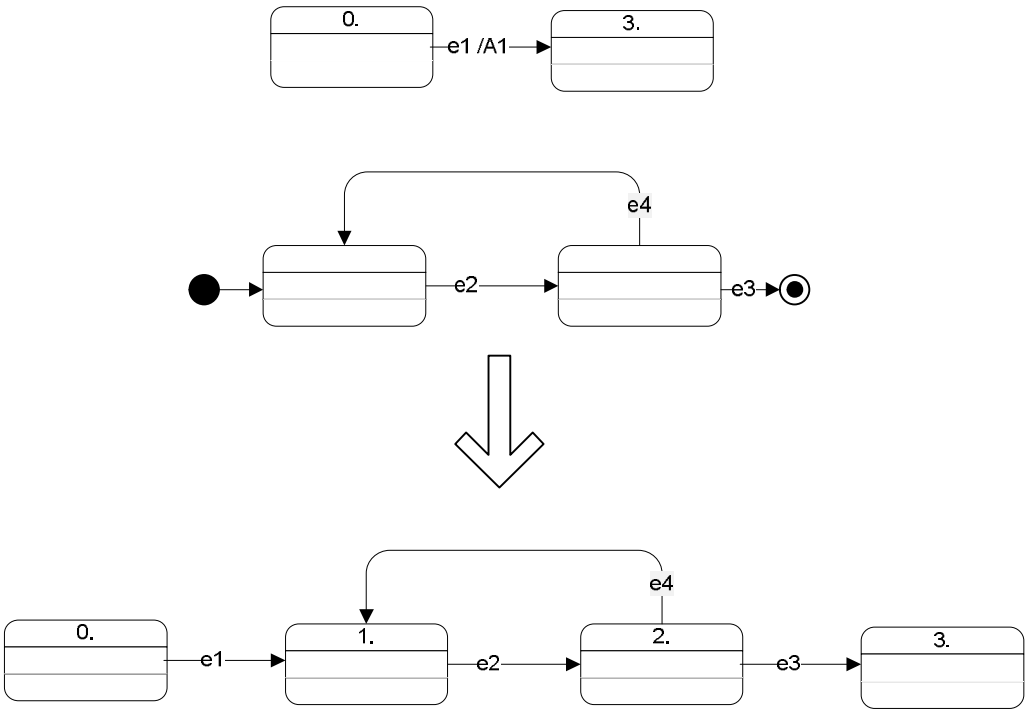


Рис. 10. Встраивание вызываемого автомата

Мотивация. Размещением всей логики поведения программы в одном графе переходов можно добиться большей наглядности, так как такой граф можно охватить «одним взглядом».

Тем не менее, следует данный рефакторинг применять с осторожностью, потому как излишне «разросшийся» граф переходов ничуть не понятнее программы более простых. Более того, встраивание автомата нарушает автоматную декомпозицию, и поэтому потенциально усложняет последующую модификацию программы.

Техника. Пусть встраиваемый автомат A_1 содержит состояния s_1, s_2, ..., s_k (s_1 – начальное), а его вызов совершается на переходе *t*, соединяющем состояния s' и s'' автомата *A*. Для встраивания автомата необходимо выполнить следующие действия:

1. Добавить в автомат *A* состояния s_1', s_2', ..., s_k', соответствующие состояниям s_1, s_2, ..., s_k с сохранением выходных воздействий.
2. Для каждого перехода t_l, соединяющего состояния s_i и s_j встраиваемого автомата добавить соответствующий переход с теми же атрибутами между парой состояний s_i' и s_j'.
3. Изменить конечное состояние перехода *t* на состояние s_1'.
4. Для каждого перехода, ведущего из некоторого состояния s_i в конечное состояние встраиваемого автомата, добавить переход с такими же атрибутами, ведущий из состояния s_i' в состояние s''.
5. Если вызовов автомата *A* больше нет, то удалить состояния s_1, s_2, ..., s_k.

Доказательство корректности. По аналогии с предыдущим доказательством рассмотрим взаимно однозначное соответствие между состояниями изначального и измененного автоматов $f : S \to S'$. При этом начальное состояние автомата *A* соответствует начальному состоянию автомата

A'. Тогда в цепочках элементов s_{j_0}, z_{k_0}, e_{i_1}, z_{l_1}, s_{j_1}, z_{k_1}, e_{i_2}, z_{l_2}, s_{j_2}, z_{k_2}, … и s'_{j_0}, z'_{k_0}, e_{i_1}, z'_{l_1}, s'_{j_1}, z'_{k_1}, e_{i_2}, z'_{l_2}, s'_{j_2}, z'_{k_2}, … первые состояния s_{j_0} и s'_{j_0} соответствуют друг другу: $f(s_{j_0}) = s'_{j_0}$ и $z_{k_0} = z'_{k_0}$.

Докажем, что если в цепочках совпадают первые t ($t > 0$) элементов, то (t+1)-ые элементы также совпадают. Другими словами, если выполняется: $f(s_{j_x}) = s'_{j_x}$, $z_{k_x} = z'_{k_x}$, $z_{l_x} = z'_{l_x}$ для всех $x \le t$, то также верно, что $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$. Рассмотрим несколько случаев:

1) s_{j_t} не принадлежит автомату A_1:
   a) $s_{j_t} = s'$. Тогда $s_{j_{t+1}} = s_1$ или $s_{j_{t+1}}$ также не принадлежит автомату A_1. В первом случае $f(s_{j_{t+1}}) = f(s_1) = s'_1 = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$ и $z_{l_{t+1}} = z'_{l_{t+1}}$ (см. п. 3 техники). Во втором же случае сразу получаем $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$, так как соответствующий переход и состояние не претерпевали никаких изменений.
   b) $s_{j_t} \neq s'$. Тогда $s_{j_{t+1}}$ не принадлежит автомату A_1, и переходы, ведущие из s_{j_t}, остались неизменными. Следовательно, $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$.
2) s_{j_t} принадлежит автомату A_1. Тогда либо $s_{j_{t+1}}$ также принадлежит A_1, либо $s_{j_{t+1}} = s''$ (см. пп. 2, 4 техники):
   a) $s_{j_{t+1}}$ принадлежит A_1, то есть входит в множество состояний s_1, s_2, ..., s_k. Тогда согласно пункту 2 техники $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$.
   b) $s_{j_{t+1}} = s''$. Тогда из пункта 4 техники получаем, что $f(s_{j_{t+1}}) = s'_{j_{t+1}}$, $z_{k_{t+1}} = z'_{k_{t+1}}$, $z_{l_{t+1}} = z'_{l_{t+1}}$.

Корректность рефакторинга доказана.

1.2.4.6. Переименование состояния

Описание. Изменение имени состояния.

Мотивация. Описанные до сих пор рефакторинги изменяли структуру автомата, адаптировали ее для лучшей реализации текущей или изменившейся спецификации. Между тем, часто большей прозрачности и понятности структуры автомата можно добиться простой сменой имени одного или нескольких состояний. Часто при анализе автоматов и их рефакторинге требуется изменить имена состояний так, чтобы они лучше отражали их семантику. Четкое и полное выражение смысла состояния одним-двумя словами часто может оказаться нетривиальной задачей, которая не поддается решению с первого раза. Со временем, когда программист работает с задачей уже несколько дней или недель, в голову часто приходят более удачные метафоры.

Наименования являются важной частью информационной составляющей описания автомата и в значительной степени определяют скорость восприятия его семантики. В некоторых случаях стоит выбрать более короткое имя для состояния, так как короткие имена делают граф переходов более компактным.

Переименование состояния является очень простым рефакторингом, но не стоит им пренебрегать: оно может существенно упростить восприятие логики поведения программы.

Техника. Для изменения имени состояния необходимо выполнить следующие действия:

1. Убедиться, что новое имя является уникальным для графа переходов.
2. Изменить имя состояния.

Доказательство корректности. Так как поведение автомата не зависит от имен состояний, переименование состояния не повлияет на работу автомата.

1.2.4.7. Перемещение воздействия из состояния в переходы

Описание. Вызов выходного воздействия, совершаемого при входе в состояние, перемещается в переходы, входящие в рассматриваемое состояние (рис. 11).

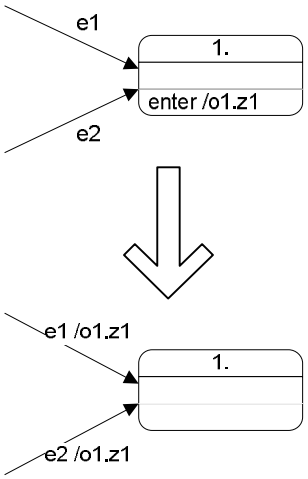


Рис. 11. Перемещение воздействия из состояния в переходы

Если при входе в состояние совершается несколько воздействий, то перемещается только первое. При необходимости рефакторинг можно повторить для остальных воздействий.

Техника. Для переноса воздействия из состояния в переходы необходимо выполнить следующие действия:

1. На каждом переходе, входящем в состояние, добавить вызов воздействия. Если на переходах уже были выходные воздействия, то добавляемое воздействие становится последним.
2. Вызов воздействия удалить из состояния.

Доказательство корректности. Пусть для некоторого t : $z_{k_t} \neq z'_{k_t}$ или $z_{l_t} \neq z'_{l_t}$. Рассмотрим среди всех таких t минимальное значение.

Тогда $s_{j_t} = s$. После внесения изменения цепочка элементов элементов s_{j_0}, z_{k_0}, e_{i_1}, z_{l_1}, s_{j_1}, z_{k_1}, e_{i_2}, z_{l_2}, s_{j_2}, z_{k_2}, …, e_{i_t}, z_{l_t}, s_{j_t}, z_{k_t}, … будет иметь вид s_{j_0}, z_{k_0}, e_{i_1}, z_{l_1}, s_{j_1}, z_{k_1}, e_{i_2}, z_{l_2}, s_{j_2}, z_{k_2}, …, e_{i_t}, z_{l_t}, z_{k_t}, s_{j_t}, … ($z = z_{k_t}$). Легко видеть, что при этом общая последовательность выходных воздействий не изменяется. Аналогично для остальных t получаем, что последовательность выходных воздействий не изменится.

1.2.4.8. Перемещение воздействия из переходов в состояние

Описание. Вызовы одинаковых выходных воздействий, совершаемых на переходах, входящих в одно состояние, заменяются одним воздействием, выполняемым при входе в это состояние (рис. 12).

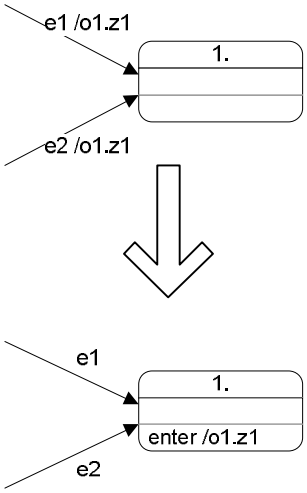


Рис. 12. Перемещение воздействия из переходов в состояние

Описываемое изменение будет являться рефакторингом, если одинаковое выходное воздействие имеют все переходы, входящие в состояние.

Пример. Перемещение воздействия из переходов в состояние подробно рассматривается в разд. 1.2.6

Техника. Для перемещения воздействия *z* внутрь состояния *s* необходимо выполнить следующие действия:

1. Убедитесь, что воздействие *z* имеется на всех переходах, входящих в состояние *s*. Если на одном из переходов вызываются несколько воздействий, проверьте, что воздействие *z* вызывается последним.
2. Добавьте вызов воздействия *z* в начало списка воздействий, выполняемых при входе в состояние *s*.
3. Удалите вызовы воздействия *z* из всех переходов, входящих в состояние *s*.

Доказательство корректности. Пусть для некоторого t : $z_{k_t} \neq z'_{k_t}$ или $z_{l_t} \neq z'_{l_t}$. Рассмотрим среди всех таких t минимальное значение.

Тогда $s_{j_t} = s$. После внесения изменения цепочка элементов s_{j_0}, z_{k_0}, e_{i_1}, z_{l_1}, s_{j_1}, z_{k_1}, e_{i_2}, z_{l_2}, s_{j_2}, z_{k_2}, …, e_{i_t}, z_{l_t}, s_{j_t}, z_{k_t}, … будет иметь вид s_{j_0}, z_{k_0}, e_{i_1}, z_{l_1}, s_{j_1}, z_{k_1}, e_{i_2}, z_{l_2}, s_{j_2}, z_{k_2}, …, e_{i_t}, s_{j_t}, z_{l_t}, z_{k_t}, … ($z = z_{l_t}$). Легко видеть, что при этом общая последовательность выходных воздействий не меняется. Аналогично для остальных t получаем, что последовательность выходных воздействий не изменится.

1.2.5. Метод внесения изменений в автоматные программы

Как было показано в разд. 1.2.4, только рефакторинги могут обеспечить безопасность изменения – сохранение корректности автомата. Соответственно, изменения, совершаемые исключительно в целях изменения структуры автомата без изменения его поведения могут быть произведены совершенно безопасно, если они реализованы путем комбинирования нескольких рефакторингов.

Однако, согласно классификации изменений, введенной в разд. 1.2.1, два из трех типов изменений призваны модифицировать логику работы автомата для исправления ошибок или

адаптации автомата к изменившимся требованиям. При выполнении таких изменений невозможно добиться гарантированной корректности измененного автомата, не учитывая новой или старой спецификации.

Для проверки корректности изменения потребуется верифицировать получившийся автомат на соответствие оригинальной (в случае исправления ошибки) или новой (в случае адаптации автомата к изменившимся требованиям) спецификации. В [1] указано, что ошибки, найденные при верификации, представлены в виде сценария, прохождение которого автоматом не соответствует спецификации. Результатом анализа такого сценария является коррекция либо автомата, либо спецификации. В нашем случае предполагается, что спецификация верна, и в коррекции нуждается автомат. Точнее, в коррекции нуждается набор действий по изменению автомата, после которого он перестал (или не начал) удовлетворять спецификации.

Из рассмотрения заведомо можно исключить рефакторинги: их безопасность формально доказана в разд. 1.2.4. Таким образом, анализу должен подвергнуться набор базовых изменений, не являющихся частью рефакторингов. К сожалению, полностью автоматизировать такой анализ не удается и его потребуется, как минимум частично, выполнять вручную. Так как ручной анализ изменений – достаточно трудоемкий процесс, чем меньший набор изменений требуется анализировать, тем лучше.

Вышесказанное приводит к следующему методу внесения изменений в графы переходов автоматных программ.

Основа метода заключается в разделении любого сложного изменения графа переходов на две фазы:

1. Рефакторинг автомата.
2. Набор модификаций, приводящих к изменению поведения автомата.

В ходе первой фазы автомат «подготавливают» к изменениям, модифицируя его структуру, не затрагивая при этом поведение (корректность этой фазы можно проверить автоматически, если спецификация исходного автомата была формализована). Целью рефакторинга является минимизация числа модификаций, выполняемых во второй фазе.

Модификации второй фазы, изменяющие поведение, делают максимально простыми, чтобы облегчить их анализ. Дополнительную помощь оказывают указанные в разд. 1.2.3 для каждого базового изменения списки потенциальных проблем, которые могут возникнуть при внесении соответствующего изменения. Эти списки помогут выполнять дополнительные проверки после каждого такого изменения. При этом валидация автомата после основных изменений может быть проведена автоматически, так как она не связана с семантической корректностью.

Предложенный метод позволяет избежать внесения в программу сложных изменений, с трудом поддающихся анализу. Наиболее сложные изменения доказуемо безопасны, потенциально опасные изменения просты.

Заключительным шагом должна стать верификация получившегося автомата на соответствие измененной формальной спецификации. В случае обнаружения несоответствия спецификации, использование предложенного метода значительно упрощает процедуру поиска ошибок, так как набор изменений, направленный на отражение новых требований, минимален.

1.2.6. Экспериментальное применение подхода для безопасного внесения изменений на примере системы автоматов, отвечающей за работу банкомата

Рассмотрим возможное применение описанного подхода на примере внесения изменений в граф переходов автомата, отвечающего за работу банкомата. Моделирование работы банкомата с применением автоматного подхода осуществлено в работах [6, 7]. Приведем неформальное описание работы с банкоматом.

Модель общения банкомата с сервером банка построена на основе транзакций – в ходе взаимодействия с пользователем устройство банкомата накапливает вводимую информацию в специальном внутреннем списке, отправляя серверу по требованию совершения операции все описание транзакции. Так, в начале работы пользователь вводит номер карты. После этого банкомат запрашивает у него *PIN*-код. Как номер карты, так и введенный код запоминаются во внутреннем списке. Банкомат запрашивает у сервера авторизацию для карты. В случае неверного ввода *PIN*-кода, карта возвращается.

После успешной проверки *PIN*-кода пользователю становится доступно основное меню, в котором он может выбрать одно из следующих действий:

- снять деньги со счета;
- посмотреть остаток счета;
- забрать карту.

При выборе последнего пункта пользователю возвращается карта, и работа с банкоматом завершается. При выборе любого другого пункта пользователь должен еще дополнительно выбрать один из возможных вариантов или ввести число с клавиатуры, после чего автомат отправляет серверу накопленные данные. При этом клиенту отображается информация о результате операции. После этого банкомат вновь отображает главное меню, или пользователь забирает карту.

Управление осуществляется системой взаимодействующих автоматов. Формальное описание, содержащее в себе объекты управления, источники событий и систему автоматов, приведено ниже.

Схема связей автоматов с поставщиками событий и объектами управления (рис. 13).

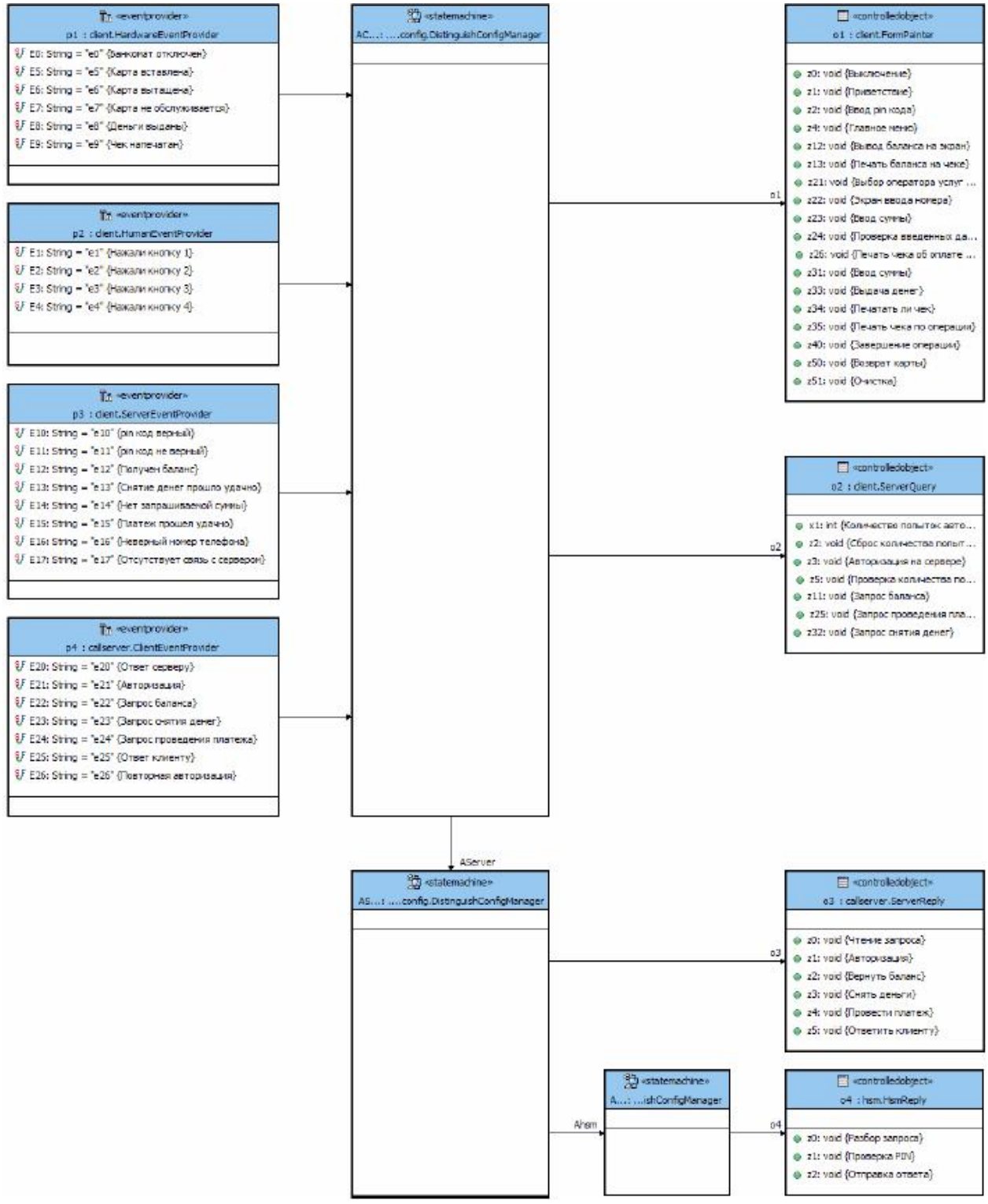


Рис. 13. Схема связей автоматов с поставщиками событий и объектами управления

Граф переходов автомата, отвечающего за работу банкомата, представлен на рис. 14:

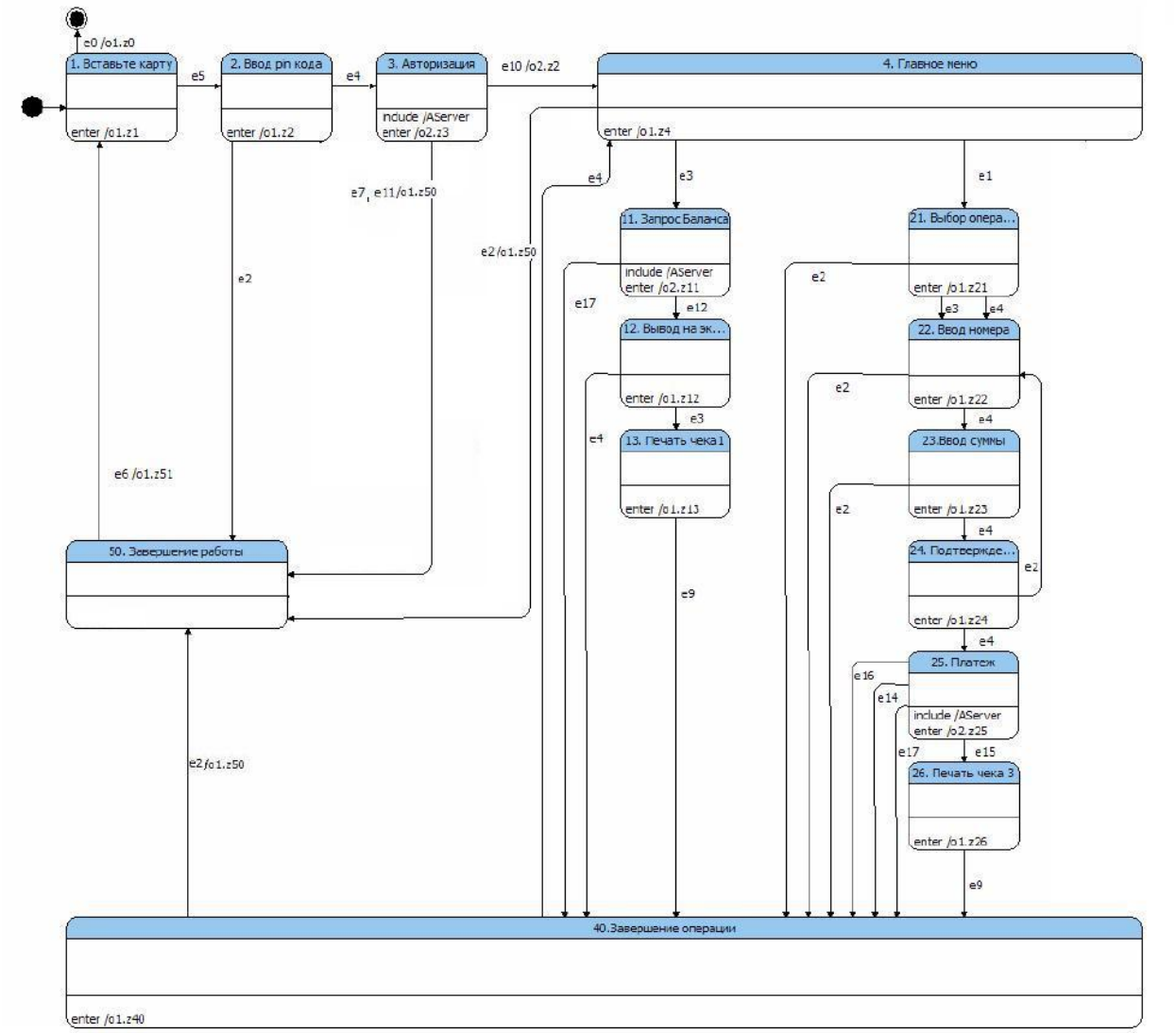


Рис. 14. Граф переходов автомата, отвечающего за работу банкомата

Рассмотрим ситуацию, когда к созданной системе изменяются требования. Пусть сценарий работы с банкоматом изменился следующим образом: «В случае неверного ввода *PIN*-кода он запрашивается повторно. В случае неверного ввода *PIN*-кода три раза подряд работа с банкоматом принудительно завершается».

Граф переходов автомата, отвечающего за работу банкомата, достаточно громоздок. Упростим его, выделив два вызываемых автомата: один – для режима просмотра остатка счета, и еще один – для режима снятия денег со счета. Граф переходов автомата после применения таких рефакторингов представлен на рис. 15.

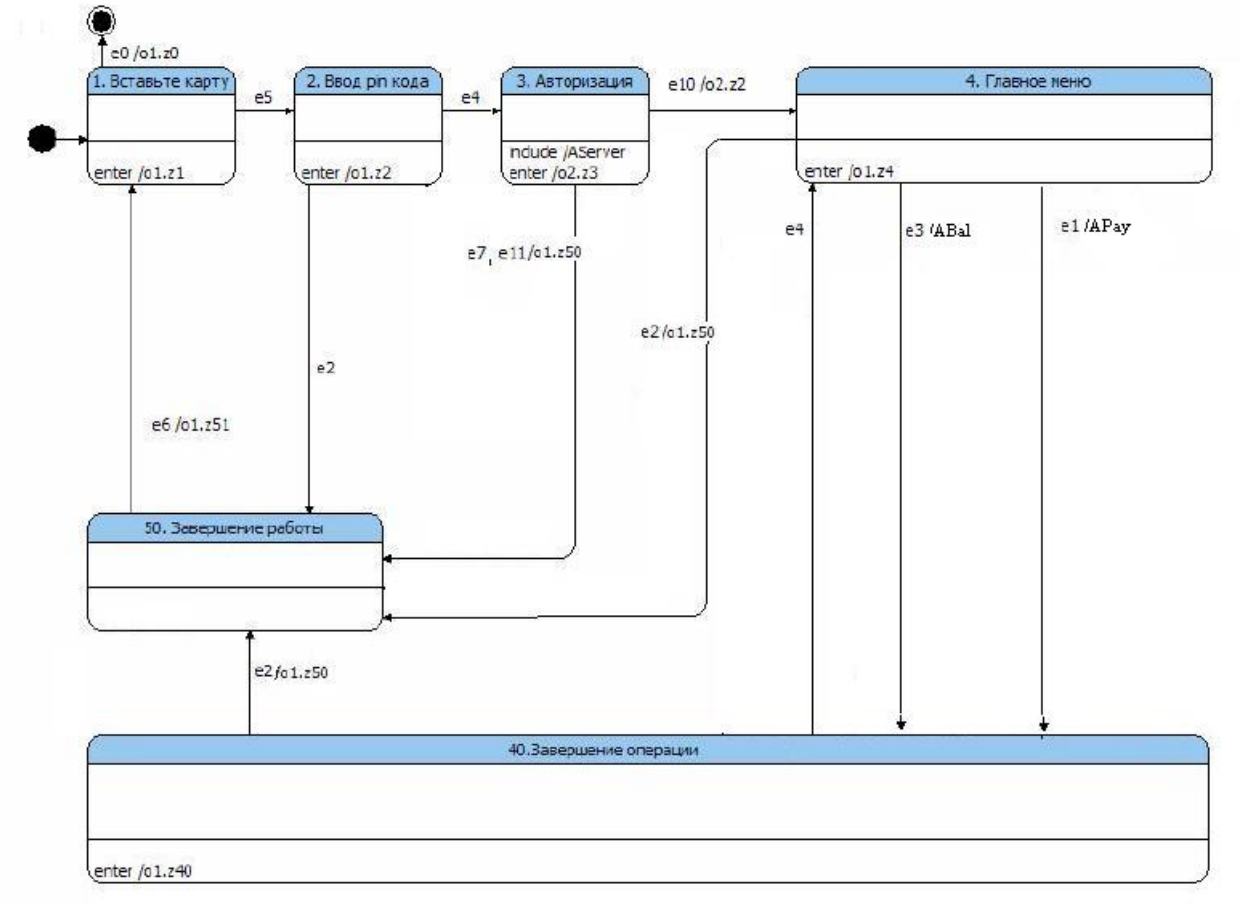


Рис. 15. Граф переходов автомата, отвечающего за работу банкомата, после выделения вызываемых автоматов

Граф переходов стал существенно легче. Теперь стало достаточно просто найти место, которое нуждается в изменении: из состояния *3* («Авторизация») по событию *e11* («pin код неверный») совершается действие *o1.z50* («Возврат карты») и переход в состояние *50* («Завершение работы»). Необходимо, чтобы по событию *e11* карта не возвращалась, а производился выбор действия в зависимости от числа неверных попыток ввода *PIN*-кода. Получаем, что надо некоторым образом разделить событие *e11* и выходное воздействие *o1.z50*. Самый простой способ сделать это – осуществить перенос воздействий с переходов, ведущих в состояние *50*, внутрь этого состояния. Граф переходов автомата, получающийся после применения такого рефакторинга (рис. 16).

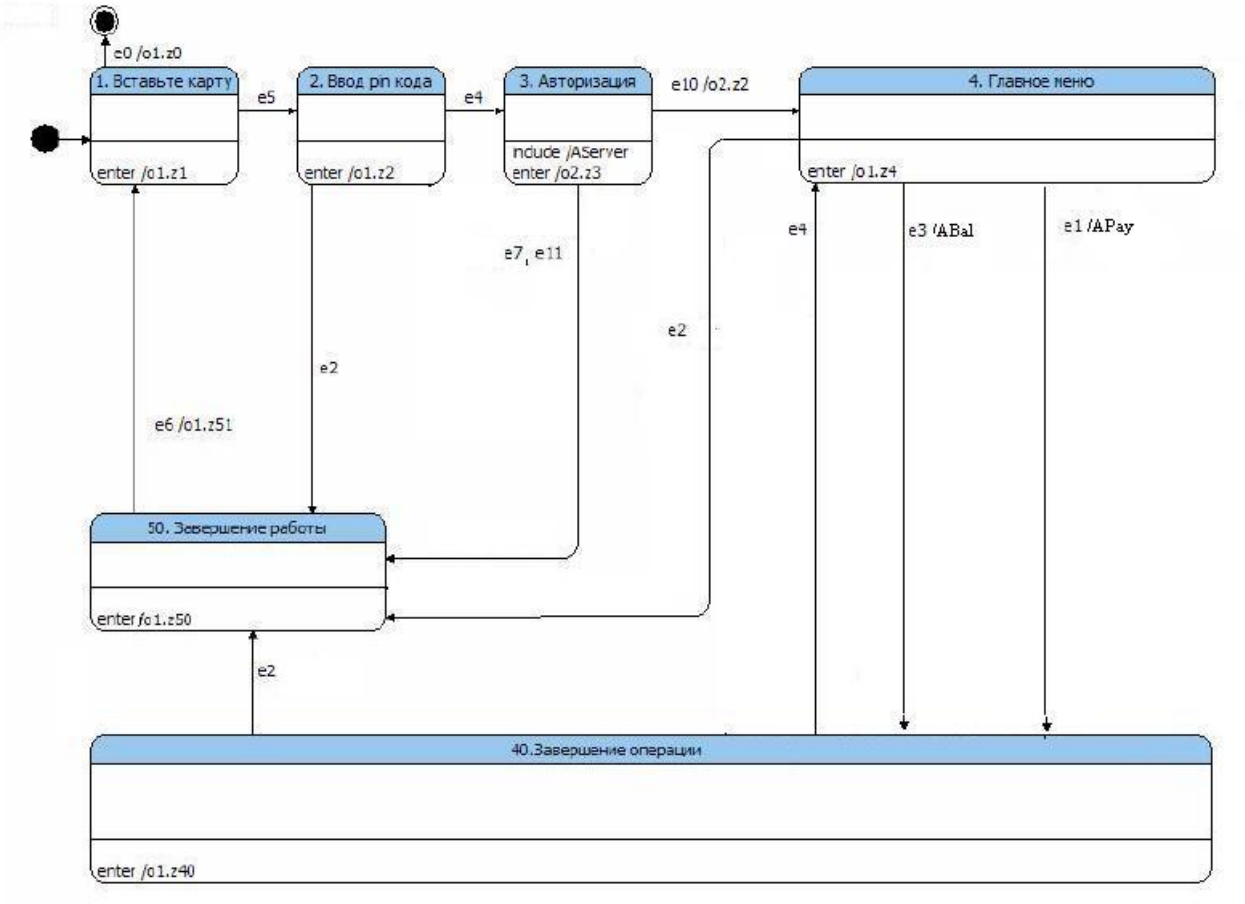


Рис. 16. Граф переходов автомата, отвечающего за работу банкомата, после переноса воздействия внутрь состояния

По событию *e11* должно осуществляться ветвление в зависимости от числа попыток. Поэтому добавим в граф переходов состояние *5* («Неверный pin»), в котором будет это ветвление осуществляться. Теперь установим конечным состоянием перехода по событию *e11* добавленное состояние. Граф переходов представлен на рис. 17.

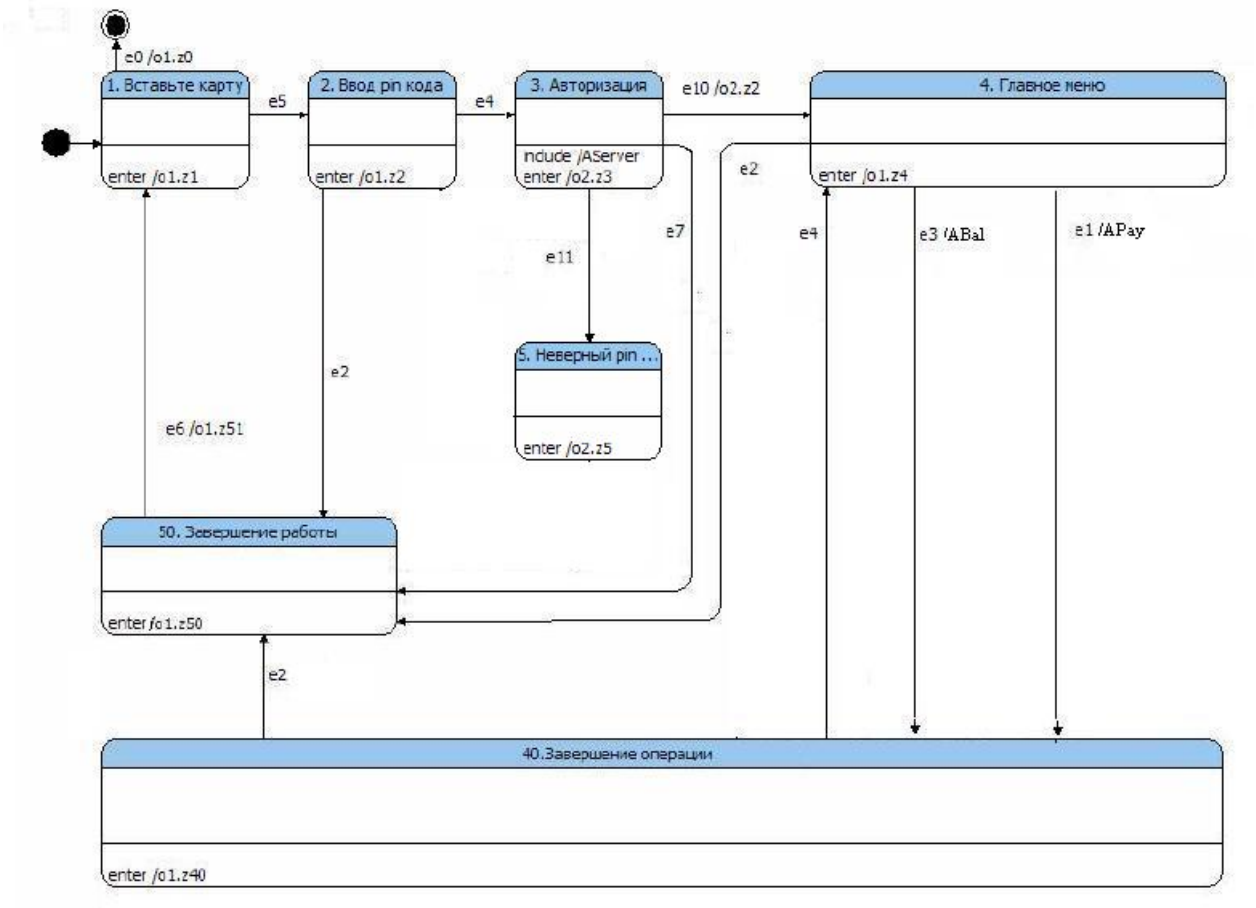


Рис. 17. Граф переходов автомата, отвечающего за работу банкомата, после добавления нового состояния

При входе в состояние *5* выполняется выходное воздействие *o2.z5* («Проверка количества попыток»). Воздействие *o2.z5* активизирует одно из двух событий: *e26* («Повторная авторизация») или *e7* («Карта заблокирована»). В первом случае, автомат должен перейти в состояние *2*, во втором – в состояние *50*. Соответствующие изменения графа переходов легко осуществить (рис. 18).

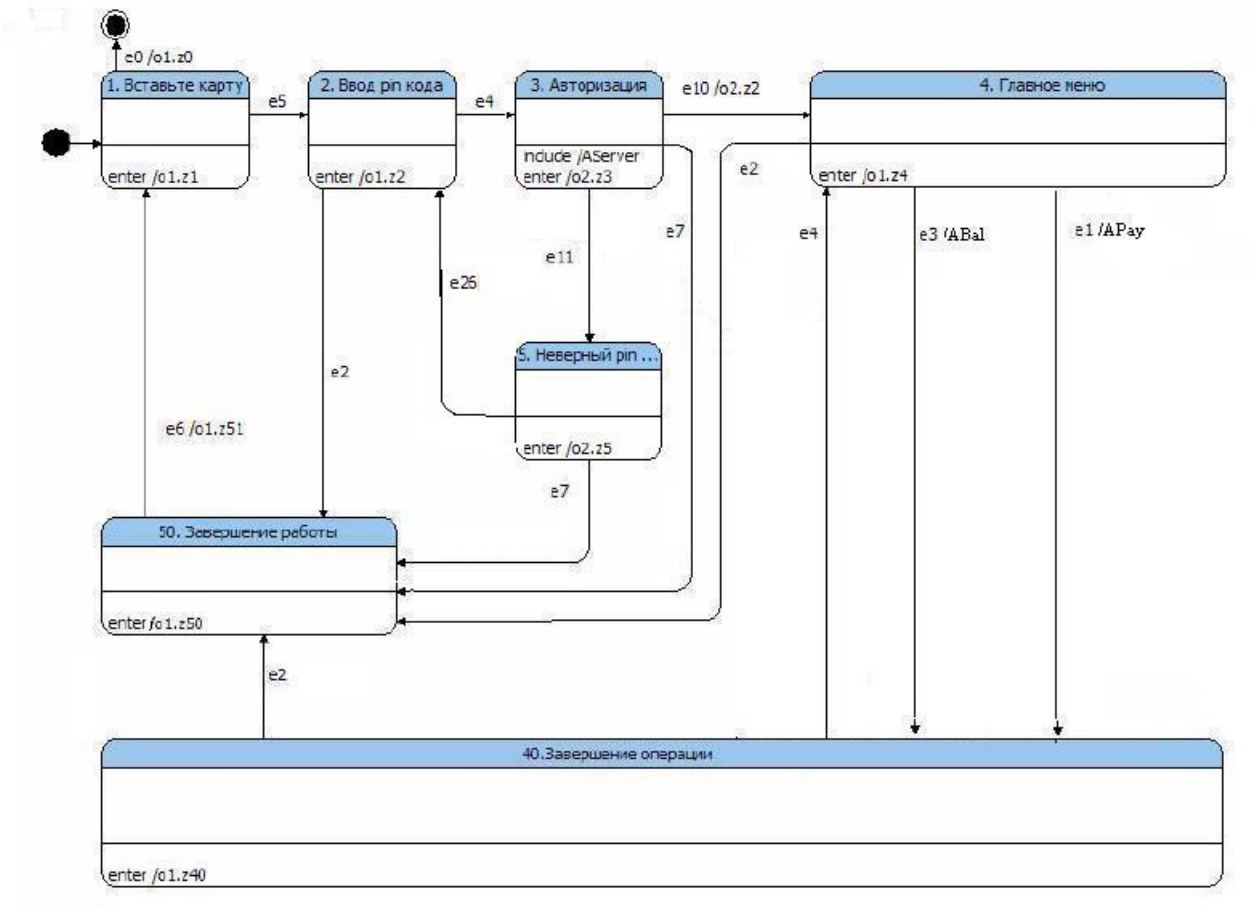


Рис. 18. Граф переходов автомата, отвечающего за работу банкомата

Полученный граф переходов удовлетворяет всем поставленным требованиям. Стоит отметить, что при модификации системы применялись только рефакторинги и базовые изменения, что позволило избежать анализа графа переходов после каждого проведенного изменения.

1.3. Разработка, программная реализация и экспериментальное исследование метода динамической верификации автоматных программ

В данном разделе приводятся разработка, программная реализация и экспериментальное исследование метода динамической верификации автоматных программ, выполненные в рамках второго этапа настоящей работы.

1.3.1. Динамическая верификация

Актуальной является задача верификации программ – проверки их соответствия заданным свойствам [8]. Существует два подхода к верификации – статическая и динамическая [8]. Наиболее распространена статическая верификация на основе метода Model Checking – проверки свойств программ на их моделях. При использовании этого метода верифицируемую программу требуется представить в специальной форме – в виде модели Крипке, описывающей возможные изменения вычислительных состояний программы [8]. С этой моделью связана и основная проблема метода

Model Checking – экспоненциальный рост размера модели Крипке при линейном росте размера программы. Эта проблема получила название «экспоненциальный взрыв».

Другой подход – это динамическая верификация, при использовании которой протоколы выполнения программы проверяются на соответствие заданной спецификации. Эта разновидность верификации применяется для проверки поведения программы во время выполнения, например, при отсутствии доступа к коду программы или при исследовании правильности взаимодействия со сторонними компонентами. Хотя верификация протокола конкретного запуска программы не может гарантировать выполнения заданных свойств при любых значениях входных параметров, при правильном подборе тестовых сценариев ее результаты могут оказаться достаточно точными. При этом трудоемкость динамической верификации не зависит от сложности верифицируемой программы, а только лишь от сложности проверяемой спецификации и размера протоколов.

Для верификации автоматных программ в настоящее время наиболее широко используемым является метод Model checking [9]. Однако исследования показали, что текущие реализации этого метода позволяют верифицировать лишь программы с небольшим числом автоматов, так как их число определяет размер состояния программы. Следовательно, размер модели Крипке системы автоматов экспоненциально зависит от числа автоматов в системе [9]. Динамическая верификация автоматных программ в настоящее время практически не изучена.

В настоящей работе предлагается метод динамической верификации автоматных программ, основанный на общем методе динамической верификации [10]. Этот метод основан на обходе альтернирующих автоматов, что позволяет верифицировать протоколы программ с трудоемкостью, линейно зависящей от размера протокола и числа подформул в спецификации.

Значительным ограничением динамической верификации программ является ненадежность этого подхода к верификации систем параллельных программ, так как последовательность передачи управления между программами может значительно изменяться от запуска к запуску. В данной работе рассматриваются системы автоматов Мили, в которых события обрабатываются по очереди. Это делает метод динамической верификации применимым для таких систем.

В первом разделе описывается существующий метод динамической верификации программ, а во втором – разработанный автором метод динамической верификации автоматных программ. При этом приводится структура метода, описываются схема построения протокола и выразительные возможности спецификации. В третьем разделе излагаются функциональные особенности и характеристики предложенного метода.

1.3.2. Альтернирующие автоматы

Для методов как статической, так и динамической верификации, в качестве языка спецификации используются языки темпоральной логики. Наиболее распространенными их них являются: язык линейной темпоральной логики (*LTL*) и язык логики ветвящихся вычислений (*CTL*). Одним из наиболее распространенных языков темпоральной логики является *LTL*. Формулы этого языка построены на множестве *атомарных высказываний* (*Prop*) и замкнуты через применение булевых операторов, унарного темпорального оператора **N** («на следующем шаге») и бинарного темпорального оператора **U** («до тех пор, как») [11].

Моделью для *LTL*-формул является *вычисление* — функция $\pi : \omega \to 2^{\Pr op}$, которая задает значения истинности высказываний из множества *Prop* в каждый момент времени, задаваемый натуральным числом. Вычисление π в момент времени $i \in \omega$ удовлетворяет *LTL*-формуле φ (обозначается $\pi, i \triangleright \varphi$) при выполнении следующих условий:

- $\pi, i \triangleright p$ для $p \in Pr op \Leftrightarrow p \in \pi(i)$;
- $\pi, i \triangleright \xi \wedge \psi \Leftrightarrow (\pi, i \triangleright \xi \wedge \pi, i \triangleright \psi)$;

- $\pi,i \triangleright \xi \vee \psi \Leftrightarrow (\pi,i \triangleright \xi \vee \pi,i \triangleright \psi)$;
- $\pi,i \triangleright \overline{\varphi} \Leftrightarrow \overline{\pi,i \triangleright \varphi}$;
- $\pi,i \triangleright N\varphi \Leftrightarrow \pi,i+1 \triangleright \varphi$;
- $\pi,i \triangleright \xi U\psi \Leftrightarrow \exists j > i : \pi, j \triangleright \psi, \forall k : i < k < j : \pi,k \triangleright \xi$.

Говорят, что π *удовлетворяет* формуле φ (обозначается $\pi \triangleright \varphi$), если и только если $\pi,1 \triangleright \varphi$.

Вычисление π является бесконечным словом на алфавите *Prop*. Таким образом, каждой *LTL*-формуле φ соответствует множество бесконечных слов, удовлетворяющих этой формуле. Это множество называется *языком формулы* φ и обозначается *L(*φ*)*.

Отметим, что для верификации спецификаций, заданных в виде *LTL*-формул на протоколах, требуется расширить определение *LTL* на конечные вычисления, в которых π задано на отрезке [1, *n*]. Это сделано в работе [10].

Для данного множества *X* определим $B^+(X)$ как множество положительных булевых формул над *X* (булевых формул, построенных из элементов *X*, которые соединены операторами $\vee$ и $\wedge$) и формул **истина** и **ложь**.

Альтернирующим автоматом Бюхи называется набор $A = (\Sigma, S, s^0, \rho, F)$ [12], где Σ – непустой конечный алфавит, *S* – непустое конечное множество состояний, $s^0 \in S$ –начальное состояние, *F* – множество *принимающих состояний*, а $\rho : S \times \Sigma \to B^+(S)$ – функция перехода.

Запуском альтернирующего автомата Бюхи на бесконечном слове $\omega = \{a_0, a_1, \ldots\}$ называется *S*-помеченное дерево *r* такое, что его корень помечен значением s^0 и справедливо следующее: если вершина дерева *x* глубиной *i* помечена значением *s* и $\rho(s, a_i) = \theta$, то эта вершина имеет k детей $x_1, x_2 \ldots x_k$, где $k \le |S|$, и множество их пометок $\{r(x_1), r(x_2), \ldots r(x_k)\}$ обращает формулу θ в истину. Запуск называется *принимающим*, если на каждой его ветви принимающие состояния или переход $\rho(s, a_i)$= **истина** встречаются бесконечно часто.

Таким образом, каждому альтернирующему автомату A соответствует множество бесконечных слов, для которых существует принимающий запуск. Это множество называется *языком автомата A* и обозначается *L(A)*.

Можно построить аналог альтернирующего автомата Бюхи для конечных слов. Такой автомат будет называться *альтернирующим автоматом*.

В работе [13] показано, что для любой *LTL*-формулы φ можно построить альтернирующий автомат Бюхи A такой, что *L(A)=L(*φ*)*, причем число состояний автомата A линейно зависит от размера формулы φ.

Таким образом, для верификации соответствия протокола работы программы спецификации, заданной в виде *LTL*-формулы, достаточно по этой спецификации построить альтернирующий автомат и проверить, является ли данный протокол элементом языка полученного автомата. В работе [10] предложены три алгоритма построения принимающего запуска для данных альтернирующего автомата и протокола.

Первый алгоритм (*обход в глубину)* пытается построить принимающий запуск, обходя альтернирующий автомат рекурсивно в глубину из начального состояния. Этот алгоритм наиболее прост в реализации, но часто требует обхода протокола несколько раз. Например, для спецификации вида **GF**φ **(**где Fφ = **истина U** φ и $G\varphi = \overline{F\overline{\varphi}}$**)**, «хвост» протокола будет повторно проанализирован на каждом шаге. Таким образом, для длинных протоколов алгоритм обхода в глубину работает неприемлемо медленно.

Второй алгоритм (*обход в ширину)* обходит автомат в ширину, поддерживая на каждом шаге все возможные комбинации записей протокола, которые могут являться элементами принимающего запуска в данный момент. Алгоритм обхода в глубину анализирует протокол лишь однажды, но вычислительное состояние алгоритма имеет размер, экспоненциально зависящий от размера спецификации. Для небольших формул множества состояний, допустимых в данный момент, невелики, но с ростом сложности формул размер этих множеств может представлять проблему.

Экспоненциальный рост вычислительного состояния алгоритма вызван недетерминизмом формулы. Эту проблему можно решить, анализируя протокол от «хвоста к голове» [14]. Третий из предложенных в работе [10] алгоритмов похож на обход в глубину, но вместо рекурсивных вызовов использует уже вычисленное для «хвоста» протокола состояние.

В следующей части статьи предлагается метод динамической верификации автоматных программ, основанный на описанном подходе. Также проанализирована эффективность работы описанных трех алгоритмов для динамической верификации автоматных программ.

1.3.3. Метод динамической верификации автоматных программ

В настоящий время для верификации программ, заданных в виде системы автоматов Мили, в основном используется статическая верификация – метод *Model Checking*. Однако, существующие реализации этого метода позволяют верифицировать лишь относительно несложные системы автоматов (сотни состояний во всей системе) [9].

Дадим краткое описание системы автоматов. Имеется набор конечных детерминированных автоматов Мили [15] с несколькими выходными воздействиями на ребрах. Для автомата задается набор событий, с которыми он может вызываться. Для каждого события определяется: может ли это событие поступать от источника событий или только от автомата?

Каждый переход может содержать:

- событие, при котором он происходит;
- условие, при котором он происходит. В условии в качестве атомарных утверждений можно использовать входные переменные $x_1,x_2,\ldots,x_k$ и выражения вида $s_{i,j}$ (оно истинно, если автомат A_i находится в состоянии $s_{i,j}$);
- последовательность действий. Она может содержать выходные воздействия z_i и передачу управления другим автоматам $A_i(e_j)$.

События обрабатываются последовательно – одно событие за один раз. В качестве входных переменных используются булевы переменные или состояния другого автомата. Переменные других типов необходимо преобразовывать в булевы.

На рис. 19 изображен пример простой системы автоматов Мили, которая может использоваться для управления грузовым лифтом. Автомат *A1* моделирует поведения лифта (*s1* – «Ожидание, двери закрыты», *s2* – «Движение», *s3* – «Ожидание, двери открыты»). Автомат *A2* моделирует лампу в кабине лифта (*s1* – «Свет выключен», *s2*– «Свет включен»). Когда автомат *A1* получает событие *e1* («Вызов»), он переходит в состояние *s2*, в котором он ожидает событие *e2* («Прибытие»). При получении этого события в автомате *A1* осуществляется переход в состояние *s3* и вызывается автомат *A2* с событием *e3*, включая тем самым свет. При получении события *e4* («Двери закрыты») автомат *A1* пересылает это событие автомату *A2*, который в ответ выключает свет. Конечно, это довольно простой пример, но он иллюстрирует взаимодействие основных компонентов автоматной системы.

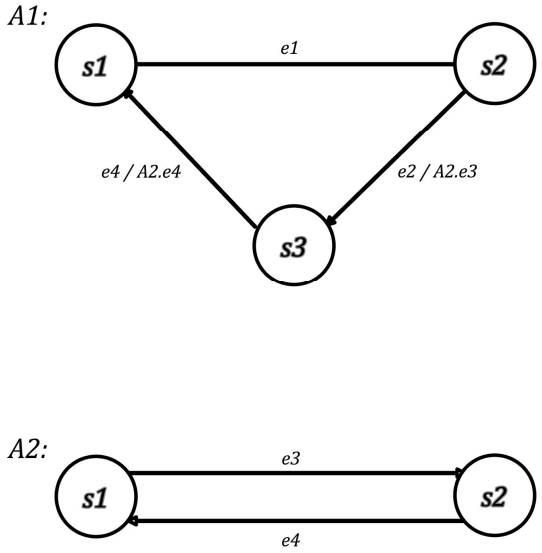


Рис. 19. Диаграммы состояний автоматов системы управления лифтом

Опишем метод динамической верификации такой системы автоматов. Он состоит из следующих простых шагов:

- запись отрицания верифицируемой спецификации в виде *LTL*-формул. Это преобразование производится вручную в соответствии с рекомендациями, изложенными в работе [16];
- автоматическое построение по полученным формулам альтернирующего автомата;
- запуск системы автоматов и автоматическое построение протокола ее работы;
- автоматический обход альтернирующего автомата в соответствии с полученным протоколом с помощью одного из алгоритмов, описанных в предыдущем разделе;
- автоматическое построение контрпримера в модели при обнаружении нарушения спецификации.

Заметим, что все шаги метода, кроме первого, выполняются автоматически, а первый шаг в том или ином виде присутствует во всех подходах к верификации автоматов по методу *Model Checking*. Таким образом, предлагаемый метод не увеличивает объем ручной работы по сравнению с другими существующими автоматическими методами.

В качестве примера приведем верификацию описанной выше автоматной системы управления лифтом. В качестве спецификации выберем условие «через некоторое время после вызова у лифта открываются двери».

Для верификации системы автоматов Мили с использованием метода динамической верификации требуется определить:

- набор атомарных предикатов, которые разрешено использовать в *LTL*-формулах спецификации;
- способ построения протокола выполнения автомата;
- правила вычисления значений атомарных предикатов в записях протокола.

Начнем с описания набора предикатов, на которых можно основывать спецификации. Обработка одного события системой автоматов состоит из следующих шагов [9]:

- получение события;
- вычисление условий на переходах из текущего состояния;
- выполнение действий на переходе. При выполнении вызова другого автомата управление передается этому автомату с соответствующим событием, вызванный автомат совершает переход по этому событию и затем возвращает управление вызвавшему автомату;
- переход в новое состояние.

В соответствии с это схемой необходимо предоставить возможность установить факт совершения того или иного этапа обработки события. Таким образом, достаточным представляется следующий набор базовых предикатов:

- $e_{i,j}$ – автомат A_i обрабатывает событие e_j;
- x_i – при обработке текущего события значение входной переменной x_i истинно;
- z_i – выполняется выходное воздействие z_i;
- $s_{i,j}$ – A_i находится в состоянии $s_{i,j}$.

Предложенный набор атомарных предикатов покрывает описательные возможности, предложенные в работе [9].

Попробуем перевести спецификацию для системы управления лифтом («через некоторое время после вызова у лифта открываются двери») на язык формул, использующих предложенные предикаты. Она будет записана как $e_{1,1} \rightarrow \mathrm{F}\ s_{1,3}$.

Далее определим структуру протокола, позволяющую однозначно вычислять значения предложенных атомарных предикатов в каждой записи. Эту структуру будем основывать на разбиении работы системы автоматов на промежуточные состояния, данной в работе [9]. Протоколом является некоторое конечное слово над конечным алфавитом. В предложенном методе алфавитом протокола системы автоматов Мили из n автоматов (по $S_i\Big|_{i=1}^{n}$ состояний в каждом) с m событиями, k входными переменными и l выходными воздействиями является множество со следующими элементами: $e_{1,1},\ldots,e_{n,m},s_{1,S_1},\ldots,s_{n,S_n},x_1,\ldots,x_k,\overline{x_1},\ldots,\overline{x_k},z_1,\ldots,z_t$. В момент начала протоколирования в протокол делаются записи $s_{i,j}$ о текущих состояниях автоматов (эту последовательность записей мы назовем *заголовком протокола*). Протоколирование может быть начато лишь при условии, что ни один автомат не производит обработку события. При получении входного события e_j автоматом A_i в протокол делается запись $e_{i,j}$. Затем происходит вычисление входных переменных и для каждой переменной, значение которой – **истина**, в протокол делается запись x_i, а для каждой со значением **ложь** – $\overline{x_i}$. При выполнении выходного воздействия в протокол делается запись z_i. Наконец, после обработки события автоматом A_i, в протокол делается запись $s_{i,j}$, указывающая новое состояние автомата, даже если произошел переход по петле и состояние автомата при переходе не изменилось.

Остается определить значения описанных предикатов для каждой записи в протоколе. Назовем *секцией обработки события* автоматом A_i последовательность записей $e_{i,j},\ldots,s_k$. *Заголовком секции* назовем начальную подпоследовательность записей длины $1 + k$: $e_{i,j},\ldots,x_k$. Секции могут быть вложены. При этом вложенные друг в друга секции обязательно соответствуют разным автоматам [9].

Тогда предикат $e_{i,j}$ имеет значение **истина** во всех записях секции обработки события, начинающейся с записи $e_{i,j}$. Предикат x_i имеет значение **истина** для всех записей секции, если в заголовке секции встречается запись x_k и **ложь**, если встречается запись $\overline{x_k}$. Отметим, что в каждом заголовке для каждой входной переменной обязана присутствовать ровно одна из этих записей. При вызове вложенного автомата значения x_i, построенные на основании заголовка секции обработки вложенного события, перекрывают предыдущие значения предикатов. Предикат z_i имеет значение **истина** только в записи z_i. Предикат $s_{i,k}$ принимает значение **истина** в записи $s_{i,k}$ и сохраняет это значение во всех записях протокола до следующей записи вида $s_{i,k'}$, не включая саму эту запись. Во всех остальных случаях предикаты имеют значение **ложь**.

Например, при одиночном вызове лифта и прибытии лифта на этаж, рассматриваемая система управления лифтом произведет действия, описываемые следующим протоколом (в квадратные скобки взяты секции обработки событий): $s_{1,1}s_{2,1}[e_{1,1}s_2][e_{1,2}[e_{2,3}s_{2,3}]s_{1,3}]$.

Отметим, что в результате разделения записей заголовка протокола ($e_{i,j},\ldots,x_k$), некоторые моменты времени, на которых производится обход альтернирующего автомата, могут быть представлены несколькими записями протокола. Так как всему заголовку каждой секции соответствует один момент времени, то для определения значений предикатов в этот момент времени требуется наличие информации обо всех записях заголовка. Эту особенность необходимо учитывать при реализации предложенного метода на практике, но, так как заголовок всегда имеет фиксированную длину, то при получении последней записи заголовка можно немедленно выяснить значения всех предикатов, описывающих состояние системы, в момент начала обработки события.

При обнаружении несоответствия записей протокола заданной спецификации требуется построить контрпример: путь в системе автоматов, который приводит к нарушению условий спецификации. При использовании предложенного метода для построения контрпримера достаточно взять заголовки секций протокола от начала до момента обнаружения нарушения спецификации. Эта последовательность записей в совокупности с заголовком протокола и даст путь в автоматной системе, являющийся контрпримером.

1.3.4. Функциональные особенности и характеристики метода

Для проверки возможности практического использования предложенного метода динамической верификации была осуществлена простейшая реализация алгоритмов построения и обхода альтернирующих автоматов. Реализация выполнена на языке *C#* и исполнялась в среде *Microsoft CLR 2.0 SP1* под операционной системой *Microsoft Windows Vista 64*. Код написан без применения специальных оптимизаций, применение которых может дополнительно ускорить скорость работы верификатора. Для тестирования в качестве примера использовалась две спецификациию. Одна из них – простая спецификация вида $\boldsymbol{G}x1 \rightarrow \boldsymbol{F}z1$: после получения события, при котором значение входной переменной *x1* – **истина**, всегда выполняется выходное воздействие *z1*. Вторая – более сложная: $\boldsymbol{G}x1 \vee (\boldsymbol{F}z1 \wedge \boldsymbol{F}z2)$, соответствует утверждению «при получении любого события либо значение переменной *x1* – **истина** либо в некоторый момент времени после него будут выполнены выходные воздействия *z1* и *z2*.

Так как алгоритм анализа протокола не зависит от структуры автомата, то для тестирования были искусственно построены протоколы различной длины, удовлетворяющие и не удовлетворяющие спецификации.

Для спецификации φ и протокола π алгоритм обратного обхода имеет трудоемкость $O(|\varphi| * |\pi|)$ и требует $O(|\varphi|)$ единиц памяти. Это делает его наиболее эффективным алгоритмом при наличии всего протокола в целом. Тестовая реализация алгоритма проводила верификацию протокола относительно спецификации вида $\boldsymbol{G}x1 \rightarrow \boldsymbol{F}z1$ с производительностью около 200 000 записей в секунду.

При отсутствии протокола в целом выбор алгоритма ограничен обходом автомата в глубину и в ширину. Алгоритм обхода в глубину в худшем случае имеет трудоемкость $O\left(|\varphi| * |\pi|^2\right)$ и требует $O(|\pi|)$ единиц памяти, что в ряде случаев (например, при верификации спецификации вида **GF**φ) делает его не лучшим выбором. На спецификации ***Gx1*** $\rightarrow$ ***Fz1*** скорость работы алгоритма составила около 100 000 записей в секунду.

Наконец, обход в ширину позволяет верифицировать протоколы с трудоемкостью $O(|\varphi| * |\pi|)$, не требуя наличия всего протокола целиком. Однако оценка потребления памяти этим алгоритмом в худшем случае составляет $O\left(2^{|\varphi|}\right)$.

В целом полученные результаты подтверждают рекомендации по использованию указанных алгоритмов, приведенные в работе [10]: при возможности рекомендуется использовать алгоритм обратного обхода. При невозможности его применения следует выбирать между обходом в ширину и глубину в зависимости от вида формулы.

Для изучения характеристик метода на практике был построен набор протоколов различной длины – от 10 до 10^6 записей, в которых записи x_i, z_1 и z_2 распределены равномерно. На каждом протоколе был несколько раз запущен алгоритм обратного обхода для каждой из двух рассматриваемых спецификаций. В результате было вычислено среднее время верификации протокола заданной длины. Результаты измерений приведены на рис. 20.

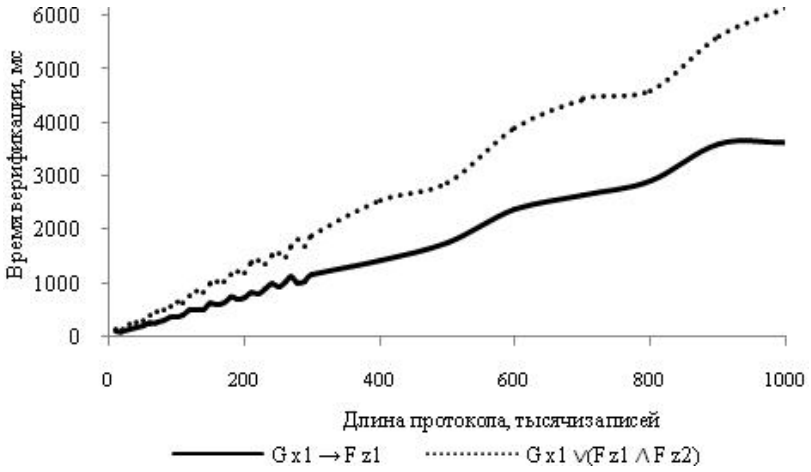


Рис. 20. Скорость работы алгоритма обратного обхода на протоколах различной длины

Важной особенностью динамической верификации программ является зависимость от точки прерывания протокола. Например, если оборвать протокол, используемый в описанном тесте, на записи x_1, то спецификация окажется не выполненной, так как соответствующая этой записи запись z_1 о выполнении выходного воздействия просто не успела попасть в протокол. В связи с этой особенностью исследователей могут интересовать не столько факт нарушения спецификации, сколько статистика выполнения или не выполнения спецификации. В работе [10] приведены модифицированные алгоритмы обхода, позволяющие собирать такого рода статистику.

Предложенный в данной работе метод можно использовать также с этими алгоритмами: последний шаг (построение и анализ контрпримера) следует заменить на анализ полученной статистики.

1.4. Разработка, программная реализация и экспериментальное исследование методов применения контрактов при разработке объектно-ориентированных автоматных систем

В данном разделе приводятся разработка, программная реализация и экспериментальное исследование методов применения контрактов при разработке объектно-ориентированных автоматных систем.

1.4.1. Преобразование требований к автоматной программе к форме, пригодной для автоматической проверки

Требования, предъявляемые к разрабатываемым программам, обычно формируются словесно. Для автоматической проверки их необходимо формализовать.

Отметим факторы, затрудняющие автоматическую проверку программ. Во-первых, для того чтобы в процессе разработки программы можно было постоянно контролировать ее качество необходимо сделать саму спецификацию исполняемой. При этом спецификации, сформулированные словесно, невозможно использовать для автоматической верификации. Во-вторых, не существует универсального способа формализации требований. В настоящее время для формализации требований к автоматным программам используют темпоральные спецификации и модульное тестирование. В данной работе предлагается применять для этой цели также и контракты. Будем различать *внешние* и *внутренние* контракты:

- внешние – для интерфейсов автоматизированных объектов управления [5] (относятся к группе состояний, их структура неизвестна);
- внутренние – для состояний автоматной модели (относятся к отдельным состояниям, их структура известна).

Внешние контракты (по аналогии с тестированием «черного ящика») применимы в тех случаях, когда структура автоматной программы неизвестна. К этой группе относятся, в частности, контракты на интерфейсы автоматизированного объекта управления – совокупности управляющего автомата и объекта управления [20]. Кроме того, внешние контракты позволяют описывать требования, предъявляемые к некоторой совокупности состояний, не производя при этом жесткой фиксации ее внутренней структуры.

Внутренние контракты (как и в случае с тестированием «белого ящика») применимы лишь к тем системам, чья структура известна. Например, внутренние контракты могут фиксировать инварианты, постусловия и предусловия, свойственные отдельному состоянию автоматной программы. Особенностью данного типа контрактов является возможность явно выражать требования не к отдельному методу (как это обычно делается в объектно-ориентированном программировании), а к целому состоянию автоматной программы.

В рамках объектно-ориентированного программирования применяются внешние контракты, способные производить проверки, использующие лишь интерфейс некоторого класса. Такие же ограничения имеют место и для программирования с явным выделением состояний.

Основываясь на введенном в работе [5] понятии «автоматизированного объекта управления», рассмотрим применимость контрактов для программирования с явным выделением состояний. На основании того, что класс (в объектно-ориентированном подходе) является естественной реализацией для автоматизированного объекта управления, предлагается записывать контракты не на отдельные классы [28], а на целые состояния.

Обратимся к описанному выше разделению контрактов на внутренние и внешние. Оно отражает применимость предложенного подхода по записи предусловий, инвариантов и постусловий для заданного состояния автоматной модели. Действительно, опираясь на имеющийся интерфейс состояния, можно записать требования, не учитывающие его структуру. Другими словами, как и в случае с тестированием «черного ящика» (в объектно-ориентированном программировании), можно проверять различные свойства отдельной сущности, не имея данных о ее реализации.

Достоинством записи контрактов для некоторого состояния автоматной программы является также возможность использования сведений о его внутренней структуре. Действительно, при необходимости можно объединять сразу несколько состояний в отдельные группы и уже для них записывать предусловия, постусловия или же инвариант.

1.4.1.1. Пример – диспетчер заданий

Приведем пример, который наглядно иллюстрирует применимость данного подхода. Рассмотрим систему (рис. 21), в которой централизованный диспетчер, управляющий запуском подзадач.

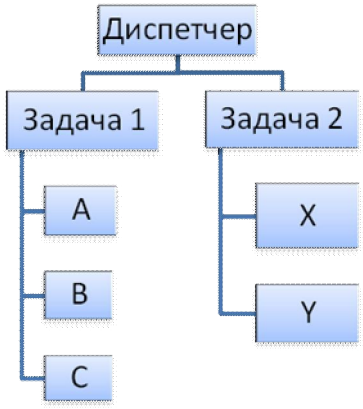


Рис. 21. Пример системы, для которой запись спецификаций получается громоздкой

Предположим, что выполнение каждой новой задачи (их число может быть больше двух) требует некоторых подготовительных действий, осуществляемых диспетчером. Другими словами, в качестве требования к такой системе можно выдвинуть условие на запуск каждой из подзадач только после того, как закончит работу диспетчер. Попытка записать темпоральную спецификацию для данной системы приводит к громоздкой формуле, длина которой растет экспоненциально с увеличением числа подзадач. Если же использовать внешние контракты, то придется сформулировать множество однотипных условий, для каждой из подзадач (A, B, C, X, Y).

В этом случае *целесообразнее использовать внутренние контракты*. При этом подзадачи A, B и C объединим в первую группу («Задача 1»), а подзадачи X и Y – во вторую. Таким образом, вместо множества формул вида $!(A \rightarrow \mathbf{X}Y)$, $!(B \rightarrow \mathbf{X}Y)$ и т. д. достаточно потребовать для любой задачи с порядковым номером i (в нашем случае $i = 1, 2$), чтобы выполнялось предусловие:

$$\underline{\mathbf{Pred}}[Задача_i] = Диспетчер.$$

1.4.1.2. На входе – требования на естественном языке, на выходе – спецификация, контракт или тесты

В качестве примера, наглядно иллюстрирующего проблему формализации вербальных требований в форме, пригодной для автоматической проверки, рассмотрим модель холодильника. Зададим список требований, предъявляемых к разрабатываемой программе:

1. При закрытой дверце внутренняя лампа не должна гореть.
2. При выходе показаний датчика напряжения за пределы допустимого диапазона управляющее реле выключит питание.
3. В момент отключения охлаждающего элемента показания термостатов не должны превышать допустимые.
4. Пока дверца не будет открыта, лампа не будет включена.
5. Если открыть дверцу холодильника, прогреть его основной объем и закрыть дверь, то:
   - будет включена лампа;
   - при достижении критической температуры будет включен охлаждающий элемент;
   - лампа будет выключена.

Приведенный список требований можно условно разделить на три группы. В первую группу отнесем требования 1–3. Они подходят для записи контрактов. Первое из них («*При закрытой дверце внутренняя лампочка не должна гореть*») является инвариантом – условием, которое должно соблюдаться в течение всего процесса выполнения программы. В требовании 2 проверяется выполнимость некоторого условия при выходе из заданного состояния, а в требовании 3 – при входе в заданное состояние. Формализуем их при помощи пост- и предусловий, оперирующих показаниями термодатчиков.

Ко второй группе можно отнести требование 4: «*Пока дверца не будет открыта, лампочка не будет включена*». Оно содержит выражение, зависящее от последовательности выполнения программы во времени, и его можно компактно записать в качестве темпоральной спецификации:

(*Лампа_не_включена* **U** *Дверь_открыта*)

Ее можно использовать для верификации на модели. Занесем полученные результаты в таблице, используя в формальной записи инварианты (`Inv`), постусловия (`Post`) и предусловия (`Pred`).

Таблица 1. Примеры записи формальных спецификаций

1. При закрытой дверце внутренняя лампочка не должна гореть	**Inv** [DoorClosed]: lamp.*isTurnedOff*
2. При выходе показаний датчика напряжения из допустимого диапазона управляющее реле выключит питание	**Post** [VoltageOutOfRange] : powerAdapter.*isTurnedOff*
3. В момент отключения охлаждающего элемента показания термостатов не должны превышать допустимые значения	**Pred** [FreezerTurnedOff] : thermoSensor.*valuesInRange*
4. Пока дверца не будет открыта, лампочка не будет включена	lamp.*isTurnedOff* **U** DoorOpened

Заметим, что не всякую темпоральную спецификацию удается легко верифицировать. Предположим, что проверке подлежит требование «*лампа в холодильнике зажигается строго после открытия двери*». В таком случае, придется исключить из темпоральной спецификации все возможные переходы, допустимые заданной автоматной моделью при открытии двери за

исключением собственно зажигания лампы. Полученная формула стала бы громоздкой и перестала бы быть удобочитаемой.

Более того, полученная таким образом спецификация полностью становилась бы привязанной к конкретной реализации модели, а это обстоятельство затруднило бы процесс контроля качества программы при изменении ее реализации.

Для описанного случая хорошо подходит тестирование. Именно к нему целесообразно прибегнуть и при проверке последнего, пятого требования. Оно представляет собой сценарий исполнения. Его легко записать и проверить, используя, например, модульное тестирование.

Отвечающая требованиям из табл. 1 модель холодильника была реализована как с явным выделением состояний, так и без него. При этом была выявлена тенденция, наблюдаемая во многих системах, реализующих сложное поведение – трудность верификации неавтоматной программы.

Попытки отказаться от автоматного подхода в больших проектах, реализующих сложное поведение, могут дать выигрыш по времени на начальном этапе. Дело в том, что отпадает необходимость строить автоматную модель программы, проектировать ее состояния, переходы и т. д. Сложности возникают на этапе обеспечения качества программы, построенной традиционным путем [22].

Верифицировать можно и объектно-ориентированную программу без явного выделения состояний. При ее верификации на модели должна быть построена структура Крипке, описывающая возможные переходы между состояниями программы, которая затем передается на вход верификатору [20]. При этом возникают сложности, связанные с построением модели Крипке по заданной неавтоматной программе и возрастает вероятность появления ошибок.

Неудобство формулировки требований к неавтоматной программе затягивает весь процесс контроля ее качества. При этом затраты на построение автоматной модели может с лихвой окупиться удобством ее верификации. Описанный выше пример с моделью холодильника – подтверждение тому. Сложности с записью темпоральной спецификации для неавтоматных программ осложняет их верификацию на модели. Поэтому для таких программ на практике наиболее распространенным средством контроля их качества остается тестирование [25].

1.4.1.3. Выводы о применимости

Как было показано выше, при проверке некоторых условий удобно применять верификацию на модели. Однако проверять таким образом требование «*событие **A** произошло сразу же после события **B***» весьма сложно, так как темпоральные спецификации при этом получаются громоздкими. В некоторых случаях удается сократить, а порой и существенно упростить спецификации, представив ее в качестве контракта. В большей степени это касается «внешних» инвариантов, объединяющих в себе требования к целой группе состояний. Отметим, что обратное преобразование (из контрактов в темпоральную спецификацию) можно задать тремя правилами:

- Инвариант «всегда верно P» записывается как **G** P.
- Предусловие «перед выполнением P должно соблюдаться Q» записывается как
  $!(!Q \rightarrow \mathbf{X}P)$.
- Постусловие «после выполнения P должно быть соблюдено Q» записывается как
  $P \rightarrow \mathbf{X}Q$.

При этом подобный переход к темпоральным спецификациям не всегда бывает оправдан. На практике часто встречаются ситуации, при которых и контракт, и темпоральная спецификация не поддаются упрощению. В таких случаях предлагается применять тестирование, описывая сценарий исполнения программы. При этом тестирование, правда, не гарантирует отсутствия ошибок в программе, а позволяет лишь проверить определенные сценарии исполнения.

Все неявные предположения, сделанные в процессе написания программы, в настоящей работе предлагается выражать с помощью контрактов: записывать инварианты класса, проверять предусловия при входе в тело метода и постусловия при выходе из него. Подобная практика приводит к повышению качества кода, делая его более понятным и надежным. Важно отметить, что ее применению мешает отсутствие единого средства управления процессом контроля качества ПО [29].

1.4.2. Существующие технологии интеграции процессов разработки и контроля качества в процессе разработки автоматных программ

Процесс верификации на модели обычно состоит из следующих шагов.

1. Построение модели программы.
2. Задание требований в терминах выбранного типа темпоральной логики.
3. Верификация модели с целью проверки выполнения формализованных требований.
4. Анализ контрпримера в случае несоответствия программы требованиям.

Для того чтобы проверять свойства автоматных моделей с помощью уже существующего верификатора, необходимо последовательно реализовать следующие операции:

- транслировать автоматную модель во входной язык верификатора;
- в случае если спецификация неверна, транслировать полученный контрпример обратно в автоматную модель.

Существует несколько работ [9, 18], в которых упомянутые операции выполнялись вручную. При этом шаги 1–4 осуществляются, как правило, в различных программах, а при переходе от одного шага к другому, соответствующие преобразования делаются вручную, что может приводить к ошибкам.

Такие ошибки опасны, как минимум, по двум причинам. Первая из них состоит в том, что транслированная с ошибками модель программы может лишиться некоторых своих свойств, которые не удовлетворяют требованиям спецификации. В этом случае, даже если оставшиеся операции будут произведены правильно, результаты верификации будут неверны. В частности, возможно получение сообщения о соответствии модели указанным требованиям, в то время как исходная программа данным требованиям не удовлетворяет. Вторая причина состоит в том, что при верно заданной модели программы вероятно появление ошибки при записи спецификации. В некоторых случаях, об ошибке сообщит верификатор, но если спецификация записана синтаксически правильно, то процесс верификации может запуститься. В этом случае будет осуществлена проверка не той спецификации, которая требовалась, соответственно, и результаты всей верификации будут неверны.

Разработанное в рамках автоматного подхода, инструментальное средство *UniMod* позволяет моделировать и реализовывать объектно-ориентированные программы со сложным поведением на основе автоматного подхода. Отметим, что модель системы с указанного средства строится с помощью двух типов *UML*-диаграмм: диаграмм связей (в форме диаграмм классов) и диаграмм состояний [22].

С использованием инструментального средства *UniMod* выполнены проекты, доступные по адресу http://is.ifmo.ru/unimod-projects/. Эти проекты показали эффективность применения автоматного программирования и средства *UniMod* при реализации систем со сложным поведением, но также выявили и ряд недостатков:

- ввод диаграмм состояний с помощью графического редактора трудоемок;
- данное инструментальное средство помогает создавать и модифицировать автоматы, но позволяет производить проверку корректности только синтаксиса;

- отсутствуют современные средства эффективной разработки автоматных программ с интеграцией средств верификации.

1.4.3. Мультиязыковая среда MPS

Изложенные в разделе 1.4.1 методы контроля качества автоматных программ предлагается интегрировать в процесс их разработки, учитывая недостатки существующих подходов, описанные в разделах 1.4.1 и 1.4.2. Таким образом, в настоящей работе предлагается объединить достоинства сразу трех подходов для проверки качества программ с явным выделением состояний:

- проверка спецификаций (проверка на модели);
- проверка предусловий и постусловий, инвариантов (контракты);
- unit testing (модульное тестирование).

В результате процесс создания автоматных программ (с внедренными в него указанными подходами) становится эффективным, так как будут сочетаться возможности современной среды разработки (статические проверки, рефакторинг, автодополнение и т.д.) и семантическая проверка автоматного кода.

Для устранения основных недостатков инструментального средства *UniMod*, рассмотренных в разд. 1.4.1, было предложено использовать новый подход к разработке автоматных программ [30]. При этом используется мультиязыковая среда *MPS* (http://www.jetbrains.com/mps), которая позволяет не только создавать новые языки, так и расширять языки, уже созданные с ее помощью.

Языки, разрабатываемые с помощью *MPS*, не являются текстовыми в традиционном понимании, так как пользователь пишет не текст программы, а вводит программу в виде *абстрактного синтаксического дерева* (АСД). Такой подход позволяет обойтись без создания лексических и синтаксических анализаторов при создании новых языков, а также настроить преобразования АСД в код на конкретном языке программирования и задать удобную среду для его редактирования. Кроме того, пользователь получает возможность после трансляции программы, написанной на языке, созданном в *MPS*, получить код, не зависящий от этой системы.

Каждый язык в среде MPS определяется набором *концептов* (сущностей), определяющих его возможности. Всякий концепт содержит описание детей, ссылок и свойств. *Дети концепта* определяют то, какие узлы могут находиться в поддереве экземпляра этого концепта. Описание ребенка состоит из *роли*, концепта ребенка и *кардинальности связи*. Кардинальность определяет число детей с этой ролью, которое может быть у экземпляра данного концепта.

Ссылки концепта определяют набор ссылок вне собственного поддерева, которые может иметь экземпляр данного концепта. Описание ссылки аналогично описанию ребенка, за исключением того, что кардинальность может принимать только значения 0..1 и 1. Свойства концепта это описания строковых, булевских и числовых атрибутов экземпляров этого концепта. Описание свойства состоит из имени и типа его значения. Примером свойства является имя переменной у концепта «объявление переменной».

Выше было отмечено, что редактор абстрактного синтаксического дерева в *MPS* реализован таким образом, чтобы воспроизвести основные особенности процесса редактирования текстовых программ, привычного для программистов. Для этого в основу редактора была положена клеточная структура. Все редактируемое дерево представлено в виде набора вложенных клеток: корневому узлу соответствует клетка, занимающая все поле редактора. Дети узлов представлены клетками внутри клетки родительского узла. Клетки могут быть конечными и составными. Клетки второго типа представляют собой набор вложенных клеток, расположенных на экране с соответствии с выбранной политикой: вертикально, горизонтально или горизонтально с переносом на следующую строку. Атомарные клетки могут быть константными, содержимое которых задано, и редактируемыми, содержимое которых может изменяться пользователем. Редактируемые клетки могут, например,

использоваться для отображения и редактирования атрибутов узла. Кроме того, клетки могут реагировать на нажатие определенных клавиш и изменять соответствующим образом дерево.

Описание редактора концепта определяет набор клеток, которыми узлы этого концепта будут представлены в редакторе. Процесс создания редактора реализует принцип *WYSIWYG* (*What You See Is What You Get*) – позволяет оценивать результат непосредственно при редактировании.

Помимо редактирования значений свойств, редактор должен обеспечивать удобство создания новых экземпляров концептов. За это отвечает меню подстановки. При нажатии комбинации клавиш *Control+Space* активизируется меню, с помощью которого можно выбрать концепт создаваемого узла. По умолчанию список содержит все доступные конкретные концепты, расширяющие требуемый, но это поведение возможно изменить. Другой важной возможностью является механизм левых и правых трансформаций, позволяющих преобразовывать дерево при нажатии определенных клавиш. Например, при нажатии после числовой константы клавиши «+», она будет преобразована в узел операции сложения.

Генератор языка содержит набор правил, преобразующих синтаксическое дерево программы на одном языке в дерево на другом языке, для которого уже задана семантика. Обычно таким языком является *Java*, для которого используется стандартный компилятор и среда исполнения. Если один язык расширяет другой, то его генератор может преобразовывать программу в дерево на расширяемом языке. После этого стандартный генератор преобразует его в код на языке *Java*.

1.4.4. Совмещение императивного кода, автоматного кода, спецификаций и контрактов в одной программе

Предложенный в настоящей работе способ создания безопасных программ с явным выделением состояний базируется на совмещении в процессе разработки императивного кода, автоматного представления, а также трех вышеописанных подходов к проверке качества ПО: верификации на модели, контрактного программирования и тестирования.

Преимуществом такого подхода является привязка спецификаций и проверяемых контрактов к программе, записанной в терминах конечных автоматов. При этом все управление осуществляется централизовано на базе свободно распространяемой среды метапрограммирования *MPS*. Отдельно отметим наличие в *MPS* языка *stateMachine* [30], позволяющего для *Java*-класса, реализующего сложное поведение добавить автоматную модель, указав набор состояний и переходов между ними.

1.4.5. Реализация предложенного подхода

Для добавления верификации и проверки контрактов в процесс разработки автоматных программ потребуется:

- создать язык темпоральных спецификаций и контрактов, который был назван *stateSpec*;
- описать операторы языка темпоральной логики *LTL*;
- настроить систему типов темпоральных операторов;
- внедрить секцию со спецификацией в описание автоматных программ на языке *stateMachine*;
- разработать плагин для запуска внешнего верификатора *NuSMV*;
- указать сочетания клавиш, запускающие верификацию или проверку контрактов;
- преобразовать автоматную модель к форме, пригодной для автоматической верификации;
- настроить обработку полученных данных верификации;
- выделить сообщение о результатах верификации;

- в случае обнаружения контрпримера, заменить исходными названия состояний и событий в цепочке исполнения программы, приводящей к нарушению спецификации или контракта.

1.4.5.1. Создание языка темпоральных спецификаций и контрактов

В мультиязыковой среде *MPS* язык представляет собой набор *концептов* (*concepts*). Наиболее важные из них: структура языка, редактор, генератор и система типов (рис. 22).

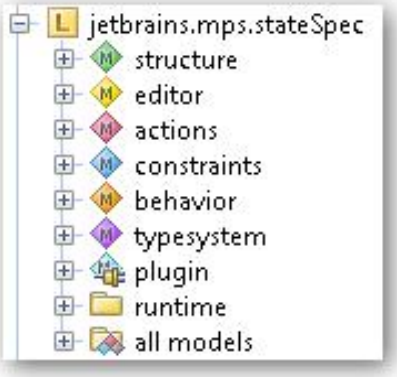


Рис. 22. Составные части языка stateSpec в среде MPS

Структура языка определяет набор узлов для построения синтаксического дерева (например, операторы и операнды). Редактор для данного концепта определяет то, как будет отображены его экземпляры в коде самой программы. Генератор задает семантику языка (явно описывая операции, необходимые для получения исполняемого кода), а система типов позволяет наложить ограничения на использование экземпляров данного концепта. Концепты могут наследовать друг друга и образовывают иерархию.

Это позволяет выделять общую функциональность в базовые концепты с целью их повторного использования. В частности, далее будет описано создание концептов *BaseTemporalExpression* и *State-Contract* для представления темпоральных операторов и контрактов, соответственно.

1.4.5.2. Описание сущностей для языка LTL

Опишем все операторы языка темпоральной логики *LTL*. Как известно, этот язык состоит из трех унарных (**X**, **G**, **F**) и двух бинарных операторов (**U**, **R**):

- Next (X) – в следующий момент времени верно, что …;
- Global (G) – в любой момент времени верно, что …;
- Future (F) – когда-либо будет верным, что …
- Until (U) – первое условие верно, пока не будет выполнено второе;
- Release (R) – первое условие верно, пока не будет выполнено второе.

Создадим иерархию сущностей для работы с темпоральными операторами языка *LTL*, во главе которой будет абстрактный concept *BaseTemporalExpression*. Его задачей является хранение ссылки на выражение типа *Expression*, которое в дальнейшем можно будет полиморфно анализировать. Также в

эту сущность будет добавлено ограничение: применять темпоральную спецификацию разрешено лишь в секции для спецификаций (рис. 23).

```
concepts constraints BaseTemporalExpression {
  default concrete concept: <no defaultConcreteConcept>

  can be child
  (operationContext, scope, parentNode, link, childConcept)->boolean {
    parentNode.ancestor<concept = Specification, +> != null;
  }

  can be parent <none>

  <<property constraints>>

  <<referent constraints>>

  default scope
  <no default scope>

}
```

Рис. 23. Запрет на использование темпоральных операторов вне заданной секции для спецификаций

Затем опишем две сущности – отдельно для унарных и бинарных операторов, настроив для каждого редактор и возможность иметь псевдоним (рис. 24, рис. 25).

```
concept UnaryTemporalExpression extends    BaseTemporalExpression
                                implements <none>
  children:
  BaseTemporalExpression | expression | 1 | specializes: | <none>
  concept properties:
  abstract
```

Рис. 24. Описание сущности, базовой по отношению к унарным темпоральным операторам

```
concept BinaryTemporalExpression extends BaseTemporalExpression
                                 implements <none>
  children:
    BaseTemporalExpression | leftOperand  | 1 | specializes: | <none>
    BaseTemporalExpression | rightOperand | 1 | specializes: | <none>
  concept properties:
  abstract
```

Рис. 25. Описание сущности, базовой по отношению к бинарным темпоральным операторам

Определим также удобный редактор для работы с выражениями на языке темпоральных спецификаций. В случае с бинарными операторами редактор выглядит следующим образом (рис. 26).

```
editor for concept BinaryTemporalExpression
  node cell layout:
    [> ^% | leftOperand | % ^{{ alias }} ^% | rightOperand | % <]
```

Рис. 26. Настройка редактора для бинарных темпоральных операторов

Это позволит в дальнейшем записывать бинарные операторы в спецификациях с использованием ввода следующего вида: оператор – операнд – оператор.

Проделаем аналогичные операции с *UnaryTemporalExpression* и реализуем все необходимые операторы языка *LTL* (рис. 27).

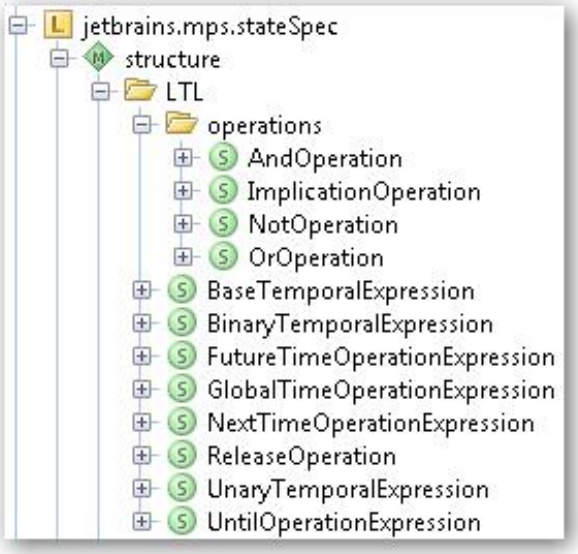


Рис. 27. Структура созданного языка с поддержкой темпоральных операторов и секций для спецификаций

При этом создадим сущности, описывающие логические операторы булевой логики:

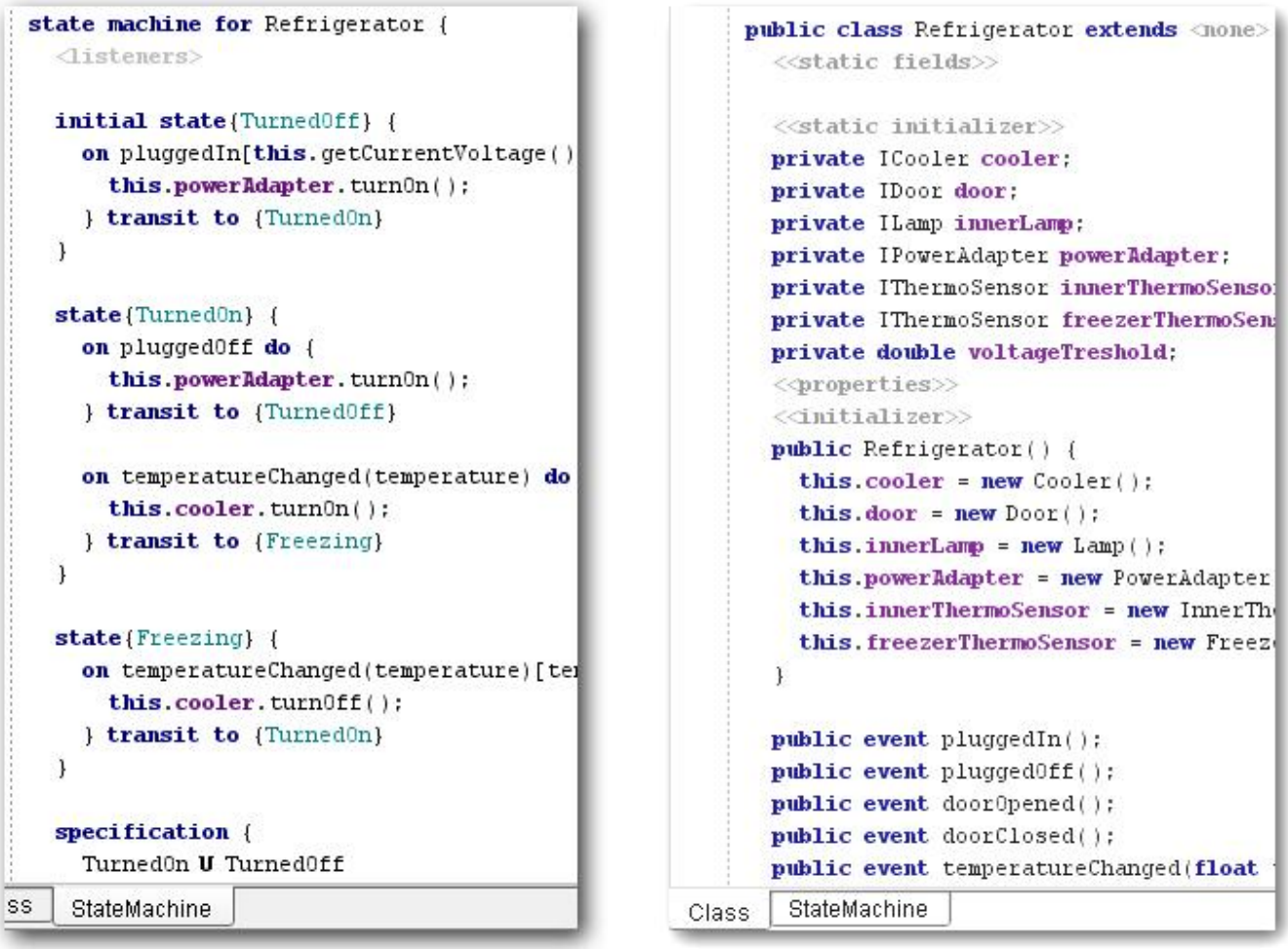
- And (&&) – И;
- Or (||) – ИЛИ;
- Not (!) – НЕ;
- Implication(→) – Импликация.

Проблема добавления в созданный нами язык поддержки логических булевых операторов решается настройкой системы типов: в качестве типа возвращаемого значения и типа операнда темпоральным операторам разрешим использовать встроенный тип *Boolean*.

1.4.6. Интеграция в автоматную модель

На рис. 27, а также в язык *temporalLogicLanguage* были добавлены сущности *StateReference* и *Specification*. Первая из них представляет собой ссылку на состояние автоматной модели, описанную на языке *StateMachine* [31].

Вторая сущность требуется для того, чтобы интегрировать секцию спецификаций в описание автоматной модели. В качестве примера на рис. 28 приведены составные части модели холодильника из разд. 1.4.1: императивный код в виде *Java*- класса и описание конечного автомата. В качестве примера спецификации в секции *specification* приведена формула «пока холодильник не выключат, он будет оставаться включенным».



а б

Рис. 28. Описание автоматной модели холодильника с помощью конечного автомата (а), императивное (б)

После создания языка для описания темпоральных спецификаций, необходимо решить проблему записи автоматной модели (заданной в среде *MPS*) на языке *SMV*, с которым работает свободно распространяемый верификатор *NuSMV* (http://nusmv.irst.itc.it/).

1.4.7. Поддержка контрактов

Для того чтобы поддержать проверку контрактов в разрабатываемом языке *stateSpec* воспользуемся результатами разд. 1.4.1. В нем были предложены формулы для записи инвариантов, предусловий и постусловий через темпоральные спецификации.

На основе результатов, полученных ранее, можно переформулировать заданные к автоматной программе контракты на языке *LTL*, а затем провести их статическую проверку, используя все тот же верификатор *NuSMV*. Таким образом, пользователь сможет записывать и проверять темпоральные спецификации или контракты в тех случаях, когда это будет удобнее.

Следует отметить, что верификатор *NuSMV* работает со своим языком описания автоматных программ. При этом, как уже отмечалось выше, преобразование модели к этому языку вручную

чревато появлением привнесенных ошибок. Поэтому преобразование автоматной модели, созданной в среде *MPS* к форме, понятной верификатору *NuSMV*, необходимо автоматизировать. Остановимся на этом вопросе подробнее.

1.4.8. Описание автоматной модели на языке SMV

Модель на языке *SMV* содержит набор перечислимых переменных и правила переходов. Для записи модели необходимо явно задать правила, согласно которым эти переходы могут осуществляться. Наглядно проиллюстрируем синтаксис данного языка на примере:

```
MODULE main
VAR
  request : boolean;
  state : ready, busy;
ASSIGN
  init(state) := ready;
  next(state) := case
  state = ready & request = 1 : busy;
  1 : ready, busy;
esac;
```

Описанная модель имеет два состояния: `ready` (начальное) и `busy`. Если переменная `request` имеет в данный момент значение `true`, а также `state` = `ready`, то состояние изменяется на `busy`. Иначе, смена состояний происходит недетерминировано.

В случае, когда автоматная программа написана на языке *stateMachine*, для ее записи на языке *SMV* достаточно проанализировать условия на переходах и заменить их на булевские переменные, а также добавить информацию о возможных состояниях и правилах переходов. На данном этапе можно сохранить в памяти таблицу соответствия исходных условий на переходах, а также имен булевских переменных, которыми они были заменены. Она понадобится в дальнейшем, для преобразования результатов работы верификатора в термины исходной модели.

1.4.9. Автоматизация проверки контрактов и верификации

В разделах 1.4.3-1.4.8 был описан процесс разработки языка спецификаций *LTL* и перевод автоматной модели (со всеми ее темпоральными спецификациями и контрактами) в формат *SMV*. Для того чтобы полностью автоматизировать процесс верификации автоматных программ и проверки контрактов, описанных в среде *MPS*, осталось разработать инструмент, который позволил бы выполнять последовательно все вышеописанные преобразования, запускать верификатор *NuSMV* и выводить полученные результаты (в терминах исходной модели) пользователю.

Для этого создадим в *MPS* отдельный *плагин* (дополнительную функциональность, доступную по нажатию заданного сочетания клавиш). В его тело добавим вызовы соответствующих методов, разработанных для преобразования контрактов и темпоральных спецификаций. После выше описанных преобразований исходной модели будет запускаться внешний верификатор *NuSMV* (работа с ним описана в классе *NuSMVProcess*, рис. 29).

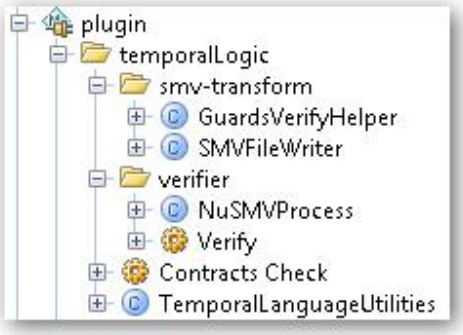


Рис. 29. Плагин для запуска верификатора NuSMV

Для удобства зададим сочетание клавиш, которое будет его вызывать (в данном случае, *ALT+V*) и ограничим область его использования секциями с темпоральными спецификациями и контрактами. Как сказано выше, для концептов, описывающих автоматную модель в среде *MPS* (как и для обычных *Java*-классов) доступны инструменты модульного тестирования. В случаях, когда требования к программе удобнее записать в виде темпоральных спецификаций или контрактов, необходимо создать соответствующую секцию в описании автомата.

Для наглядности приведем в качестве примера запись требований, сформулированных для модели холодильника, рассмотренной в разд. 1.4.1.2. В качестве контрактов запишем инвариант на допустимые значения напряжения при работе холодильника, а также постусловие, требующее, чтобы в состояние готовности он приходил лишь строго после того как будет включен (рис. 30).

```
initial state(ReadyToUse) {

  invariant {
    voltageTreshold == 220
    currentVoltage < voltageTreshold
  }

  require for ReadyToUse
    TurnedOn
  }
}
```

Рис. 30. Контракты, записанные для состояния ReadyToUse

В секции для темпоральных спецификаций запишем требование: лампочка в холодильнике выключена до тех пор, пока не будет открыта его дверца (рис. 31).

```
specification {
   this.lamp.isTurnedOff() U DoorOpened
}
```

Рис. 31. Темпоральная спецификация

Далее можно запустить проверку контрактов или спецификаций, выбрав соответствующий пункт из контекстного меню (рис. 32).

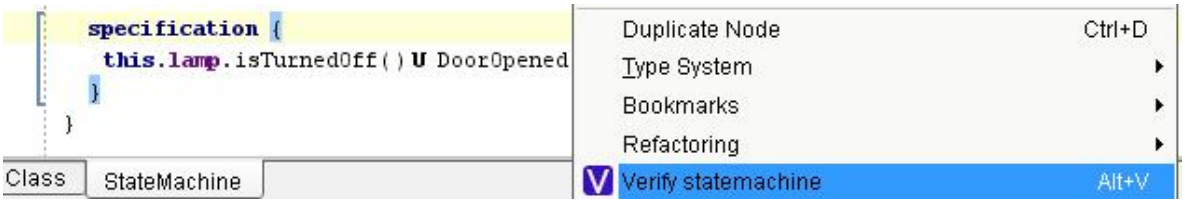


Рис. 32. Запуск процесса верификации

Полученные результаты выводятся пользователю в диалоговом окне (рис. 33)

Рис. 33. Сообщение о результатах верификации

В случае если для какой-либо из формул будет найден контрпример, он будет сформулирован в терминах исходной автоматной модели.

1.4.10. Внедрение полученных результатов

Для практического внедрения описанного в главе 1.4.5 инструментального средства была выбрана программа учета дефектов *YouTrack*, разработанная в компании ООО «ИнтеллиДжей Лабс» (работает на мировом рынке под брендом JetBrains). Программа *YouTrack* представляет собой интернет-приложение для работы с базами дефектов. С ее помощью можно добавлять информацию о новых дефектах, а также осуществлять поиск дефектов, уже имеющихся в базе данных. Также имеется возможность редактирования различной сопутствующей информации о каждом из дефектов. Интерфейс программы содержит систему контекстных подсказок и автодополнение вводимых запросов, повышая тем самым удобство работы с программой (рис. 34).

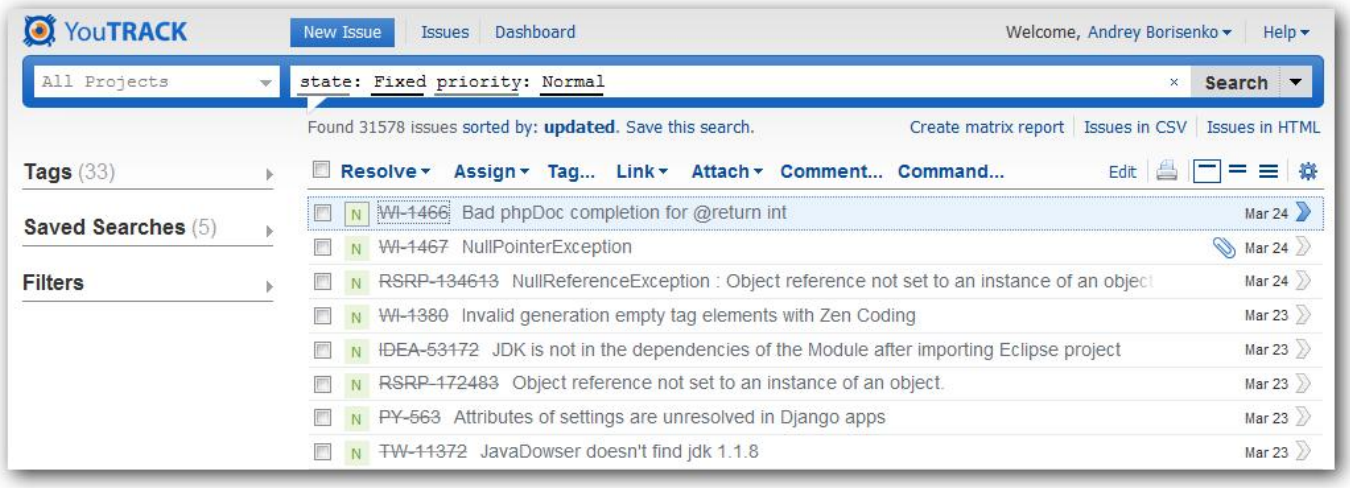


Рис. 34. Пользовательский интерфейс программы *YouTrack*

1.4.10.1. Область внедрения

YouTrack реализована в виде системы следующих взаимодействующих модулей:

- серверный модуль, работающий с базой данных;
- серверный модуль, реализующий логику приложения;
- клиентский модуль, работающий в браузере пользователя и реализующий пользовательский интерфейс.

Часть поведения этой программы реализована в виде автоматов. В частности, автоматы используются в серверной части для синтаксического анализа запросов и в клиентской части для реализации логики пользовательского интерфейса. В качестве области внедрения в рамках данной работы была выбрана именно клиентская часть программы *YouTrack*. Остановимся подробнее на ее устройстве.

1.4.10.2. Особенности системы учета дефектов YouTrack

Технической особенностью программы *YouTrack* является то, что она реализуется на базе системы *JetBrains MPS*, описанной в разд. 1.4.3. Кроме упомянутого выше языка описания автоматов *stateMachine*, также используются следующие предметно-ориентированные языки системы *MPS*:

- *baseLanguage* – язык, повторяющий синтаксис языка *Java*, используется для написания императивного кода;
- *dnq* – язык работы с данными, позволяющий выполнять запросы к спискам записей;
- *webr* – язык для разработки веб-приложений;

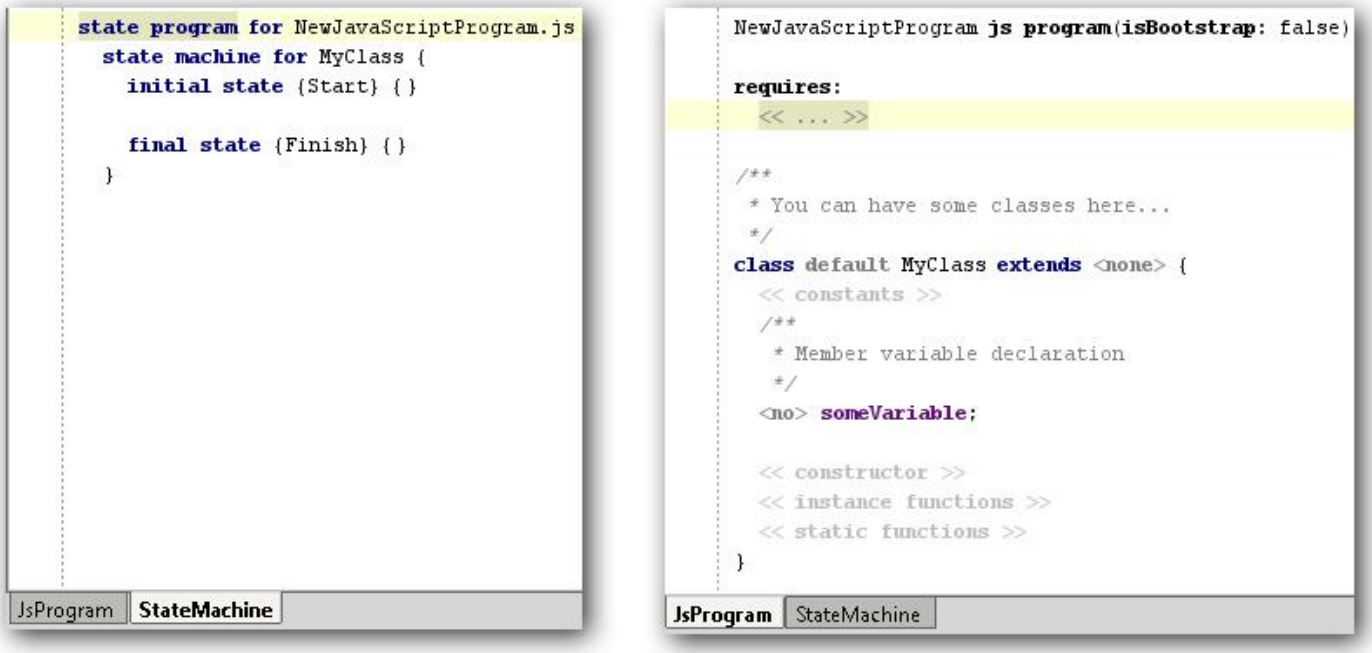
При генерации автоматов, работающих на сервере используется *Java*, в то время, как пользовательский интерфейс для работы через браузер основан на *JavaScript*. Это обстоятельство привело к тому, что язык *stateSpec* получил отдельную реализацию для работы с программами на *JavaScript*. Далее приводится подробное описание соответствующих модификаций.

1.4.10.3. Практическая реализация

Последовательно выполнив шаги, описанные в главе 1.4.5, получаем язык для записи и проверки контрактов и темпоральных спецификаций для автоматных программ. До этого момента

рассматривалась автоматная модель, соответствующая *Java*-классу, реализующему объекты со сложным поведением. Другими словами, для заданного класса описывалась автоматная модель, использующая его интерфейс (методы, события). По аналогии с предложенным в разд. 1.4.4 принципом совмещения императивного кода и описания автомата, будем хранить рядом с кодом JavaScript-программы ее автоматную модель. Для этого добавим в язык stateSpec скриптовую версию языка stateMachine. В *MPS* для этого достаточно импортировать язык `jetbrains.mps.webr.javascript.stateMachine`, нажав CTRL+L, либо явно указать зависимость в свойствах создаваемого языка.

На рис. 35 приведен пример записи кода программы на *JavaScript* и соответствующей автоматной модели. Для того чтобы хранить темпоральные спецификации и контракты непосредственно в автоматной модели необходимо повторить действия, аналогичные описанным в разд. 1.4.6. А именно, создать сущности *StateContract* и *Specification*, но хранить в них ссылку не на *Java*-класс, а на *JavaScript*-программу. При необходимости проверки сложных сценариев исполнения можно воспользоваться модульным тестированием – YouTrack имеет встроенную поддержку JUnit и Selenium.



а б

Рис. 35. Описание автоматной модели программы на JavaScript с помощью конечного автомата (а), императивное (б)

В качестве примера, иллюстрирующего контроль качества программ, разрабатываемых в рамках системы учета дефектов *YouTrack*, рассмотрим автомат, отвечающий за работу списка дефектов.

1.4.10.4. Автомат управлением списком дефектов IssueList.js

Список дефектов – один из наиболее важных элементов интерфейса программы *YouTrack*. Именно в нем выводится информация об имеющихся в базе дефектах с учетом различных фильтров

поиска. Для навигации по списку используются клавиши управления кареткой. Пример работы списка дефектов изображен на рис. 36.

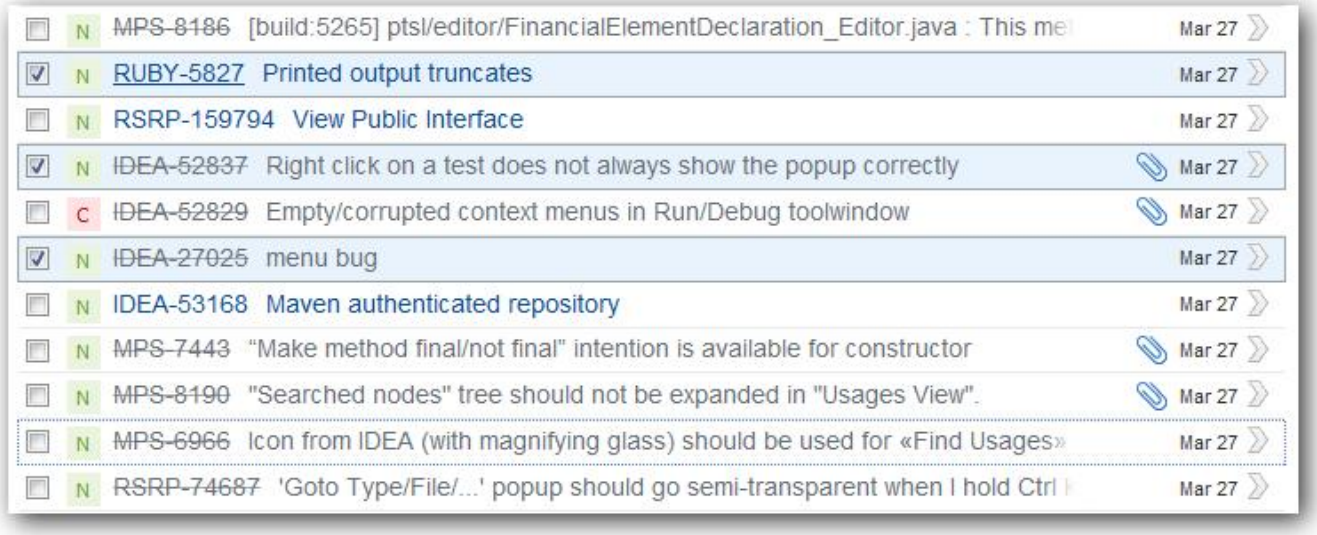


Рис. 36. Пример работы списка дефектов

Выбранные дефекты (выделены синим фоном на рисунке) а также тот дефект, который в данный момент принимает команды пользователя (обведен пунктирной рамкой) являются состоянием списка дефектов. Логика его работы реализована в классе `IssueList.js`. Поведение этого класса задается автоматом, реализованным на языке *stateMachine*, текст которого приведен ниже:

```
state program for IssueList.js
  state machine for charisma.smartui.IssueList {
    initial state {NOT_SHIFTED} {
      exit do {this.ch =
        !(this.getSelected().checkbox.checked);}
      on shiftDown() do {<< statements >>} transit to SHIFTED
    }

  state {SHIFTED} {
    initial state {INITIAL} {
      on kdown() do {<< statements >>}
        transit to MOVE_DOWN
      on kup() do {<< statements >>}
        transit to MOVE_UP
      on shiftUp() do {<< statements >>}
      transit to NOT_SHIFTED
    }

  state {MOVE_DOWN} {
    enter do {this.getSelected().setChecked(this.ch);}
    on kdown() do {this.getSelected().setChecked(this.ch);}
```

```
    on kup() do {this.ch = !this.ch;}
      transit to MOVE_UP
    on shiftUp() do {<< statements >>}
      transit to NOT_SHIFTED
  }

  state {MOVE_UP} {
    enter do {this.getSelected().setChecked(this.ch);}
    on kup() do {this.getSelected().setChecked(this.ch);}
    on kdown() do {this.ch = !this.ch;}
      transit to MOVE_DOWN
    on shiftUp() do {<< statements >>}
      transit to NOT_SHIFTED
    }
  }
}
```

Данный автомат обрабатывает четыре события:

- `kdown` (нажата клавиша «Вниз»)
- `kup` (нажата клавиша «Вверх»)
- `shiftDown` (нажата клавиша «Shift»)
- `shiftUp` (отпущена клавиша «Shift»).

Внутренним вычислительным состоянием является значение флага `ch`, указывающее на то, добавляется ли выбранный элемент в выделенный набор или исключается из него.

Выходными воздействиями автомата являются инвертирование флага `ch` («`this.ch = !this.ch`»), включение/исключение выбранного элемента из набора выделенных элементов («`this.getSelected().setChecked(this.ch)`»), а также инициализация флага `ch` («`this.ch = !(this.getSelected().checkbox.checked)`»).

Автомат содержит два состояния верхнего: `NOT_SHIFTED` («Клавиша *Shift* не нажата») и `SHIFTED` («Клавиша *Shift* нажата»). Второе состояние включает в себя три вложенных состояния: `INITIAL` («Начальное»), `MOVE_UP` («Перемещение фокуса вверх») и `MOVE_DOWN` («Перемещение фокуса вниз»).

Требования к автоматизированному объекту управления `IssueList.js` формулируются следующим образом:

- в состояние `MOVE_UP` автомат переходит только после нажатия клавиши «Вверх»;
- в состояние `MOVE_DOWN` автомат переходит только после нажатия клавиши «Вниз»;
- при движении курсором в одном направлении не должно производиться инвертирование флага `ch`.

Два первых пункта требований были формализованы в виде **контрактов-предусловий** на состояния `MOVE_UP` и `MOVE_DOWN` с формулами `kup` и `kdown` соответственно (рис. 37).

```
state (MOVE_UP) {
  require {
    kup()
  }
```

```
state (MOVE_DOWN) {
  require {
    kdown()
  }
```

а б

Рис. 37. Контракты-предусловия на состояния MOVE_UP (а), MOVE_DOWN (б)

Третий пункт был представлен в виде двух темпоральных формул (рис. 38):

```
specification {
  kdown() --> !(this.ch == !(this.ch)) R kup() || shiftUp() || shiftDown()
  kup() --> !(this.ch == !(this.ch)) R kdown() || shiftUp() || shiftDown()
}
```

Рис. 38. Темпоральные формулы, описывающие движение курсора в одном направлении

Верификация автомата *IssueList.js* по этим формулам была успешно проведена.

1.4.11. Демонстрационный пример

Рассмотрим работу описанного выше инструментального средства проверки контрактов и темпоральных спецификаций на конкретном примере. Пусть задана автоматная модель кофеварки, схематично изображенная на рис. 39.

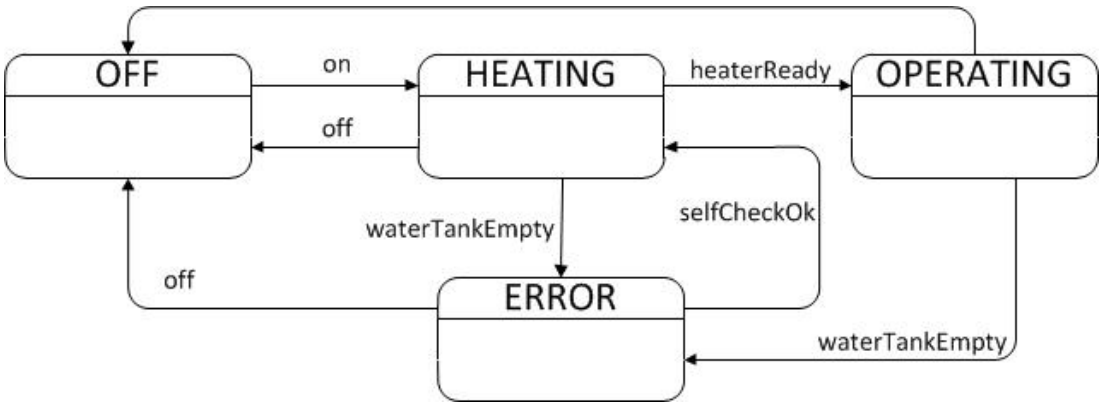


Рис. 39. Автомат, отвечающий за работу кофеварки

Условимся считать стартовым состояние `OFF` (когда кофеварка выключена). Не вдаваясь в подробности поведения данной модели в каждом из выше указанных состояний, потребуем лишь

соблюдения очередности переходов, указанной на рис. 39 (соответствующие события подписаны над стрелками).

Для реализации программы потребуется подключить язык *stateMachine* и созданный в рамках данной работы язык *stateSpec*. Контроль качества разрабатываемой программы будем осуществлять на основе некоторого набора требований, предъявляемой к модели кофеварки.

1.4.11.1. Требования к модели кофеварки

Допустим, что проверке подлежат следующие требования:

1. Кофеварка, находящаяся в состояние нагревания (`HEATING`) или приготовления кофе (`OPERATING`) при опустошении водяного резервуара перейдёт в аварийное состояние (`ERROR`);
2. Если на выключенной кофеварке (`OFF`) нажать кнопку `включить`, то кофеварка активирует нагревательный элемент;
3. Резервуар с водой остаётся пустым пока нагревательный элемент не придёт в состояние готовности;
4. Кофеварка может перейти в состояние приготовления кофе (`OPERATING`) лишь из состояния нагрева (HEATING) и только в случае готовности нагревательного элемента;
5. Во время приготовления кофе нагревательный элемент остается включенным.

1.4.11.2. Формализация требований

На основании рекомендаций, описанных в разд. 1.4.1.2, представим требования 4 и 5 как предусловия на состояние приготовления кофе. Соответствующие контракты запишем в require-секцию состояния OPERATING. Получаем описание автоматной модели, в которое интегрированы последние два требования из вышеуказанного списка (рис. 40).

```
state program for CoffeeMachine.js
  state machine for CoffeeMachine {
    initial state {OFF} {
      on on() do {
        this.turnHeaterOn;
        this.makeCoffee();
      } transit to HEATING
    }

    state {HEATING} {
      on off() do {<< statements >>} transit to OFF

      on heaterReady() do {<< statements >>} transit to OPERATING

      on waterTankEmpty() do {<< statements >>} transit to ERROR
    }

    state {OPERATING} {
      require {
        heaterReady() && HEATING
        this.turnHeaterOn U makeCoffee()
      }

      on off() do {<< statements >>} transit to OFF

      on waterTankEmpty() do {<< statements >>} transit to ERROR

      on makeCoffee() do {this.makeCoffee();}
    }

    state {ERROR} {
      on off() do {<< statements >>} transit to OFF

      on selfCheckOk() do {<< statements >>} transit to HEATING
    }
```

Рис. 40. Автоматная модель кофеварки

Требования 1–3 удобно записать в качестве темпоральных спецификаций. Полученные формулы довольно компактны и просты в понимании. Содержащая их секция представлена на рис. 41.

```
specification {
  G HEATING || OPERATING && waterTankEmpty() --> F ERROR
  waterTankEmpty() U heaterReady()
  OFF && on() --> F this.turnHeaterOn
}
```

Рис. 41. Темпоральные формулы, описывающие требования к кофеварке

1.4.11.3. Запуск проверки контрактов

Процедура запуска проверки контрактов предельно проста: в контекстном меню редактора автоматной модели MPS пользователю достаточно выбрать команду «Check the contracts». Все необходимые преобразования (см. разд. 1.4.9) произойдут автоматически. А именно, все контракты будут преобразованы в темпоральные спецификации по формулам:

- Инвариант «всегда верно P» записывается как **G** P.
- Предусловие «перед выполнением P должно соблюдаться Q» записывается как
  !(!Q → **X**P).
- Постусловие «после выполнения P должно быть соблюдено Q» записывается как
  P → **X**Q

После этого будет запущен верификатор NuSMV, о результатах работы которого пользователю будет сообщено в специальном диалоговом окне (рис. 42).

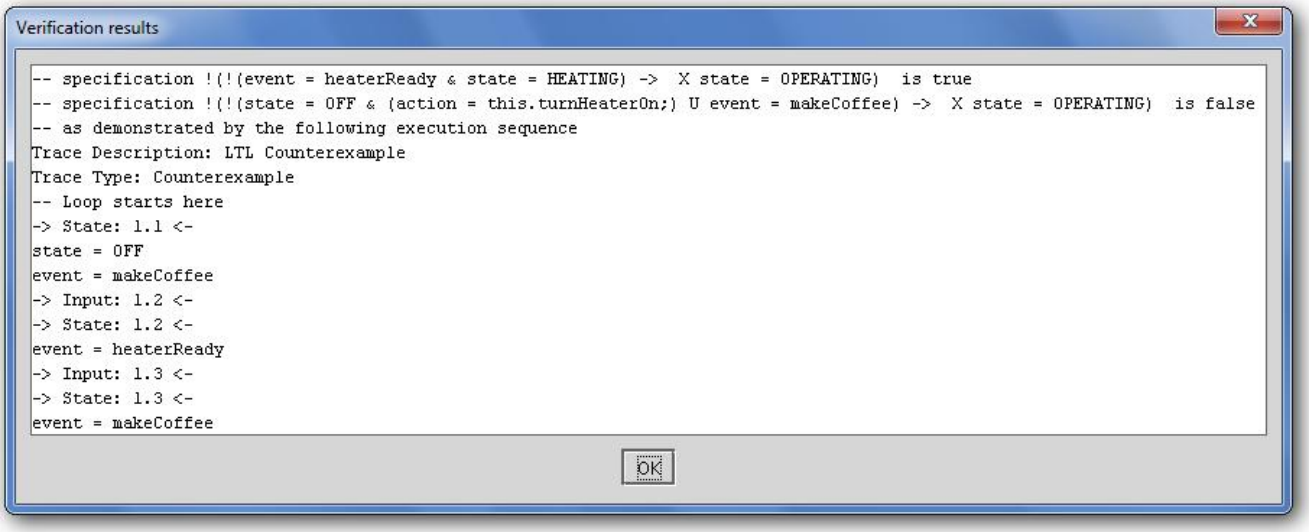


Рис. 42. Результат проверки контрактов

В данном случае, контракт, соответствовавший требованию 4 не выполняется. В состояние `OPERATING`, достижимое из стартового состояния `OFF`, кофеварка может перейти с нарушением предусловия `this.turnHeaterOn U makeCoffee()`. Соответствующий контрпример приводится в диалоговом окне на рис. 42.

1.4.11.4. Запуск проверки спецификаций

Для того чтобы проверить темпоральные спецификации, сформулированные для данной модели (рис. 41), пользователю необходимо вызвать контекстное меню «Check specification» в окне редактора MPS. После автоматических преобразований, описанных в главе 1.4.5, результаты работы верификатора NuSMV будут выведены в диалоговом окне (рис. 43).

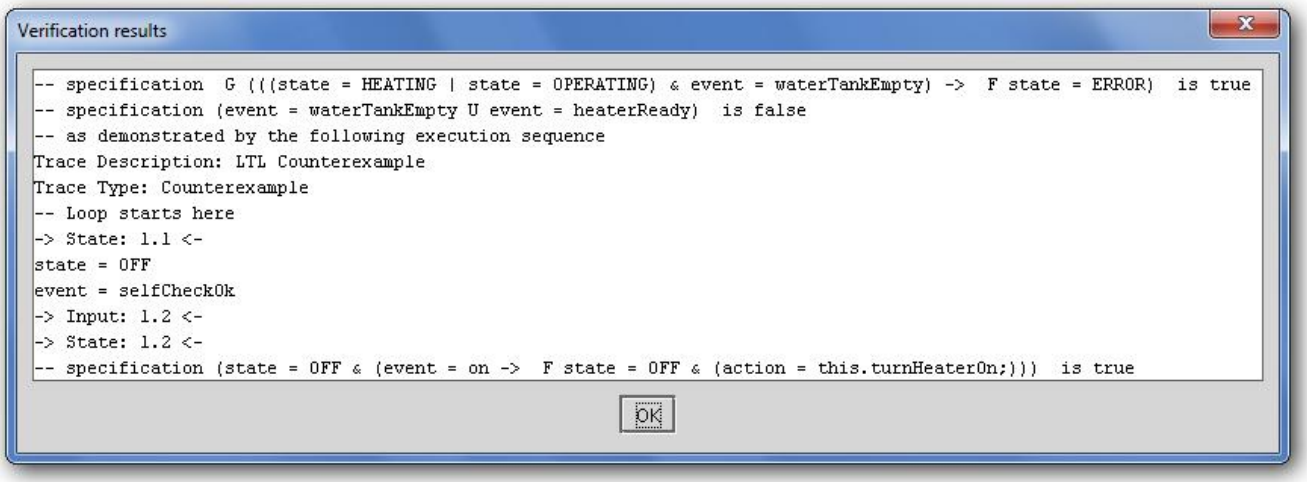


Рис. 43. Результат проверки спецификаций

Согласно результатам верификации, требования 1 и 2 выполняются. Требование 3 («Резервуар с водой остаётся пустым пока нагревательный элемент не придёт в состояние готовности») выполнено не всегда. В качестве контрпримера приводится бесконечный цикл: кофеварка находится в выключенном состоянии (`OFF`), а других событий, кроме `selfCheckOk`, не происходит. Нагревательный элемент никогда не придёт в состояние готовности. Следовательно, темпоральная формула
`waterTankEmpty` **`U`** `heaterReady` ложна.

2. РЕЗУЛЬТАТЫ ТЕОРЕТИЧЕСКИХ И ЭКСПЕРИМЕНТАЛЬНЫХ ИССЛЕДОВАНИЙ

В рамках НИР были разработаны методы совместного использования автоматного и функционального программирования, позволяющие производить автоматическую валидацию автоматных программ в процессе компиляции. Также были разработаны методы совместного использования автоматного и объектно-ориентированного программирования, поддерживающие автоматическую статическую и динамическую верификацию.

В современных проектах вся сопутствующая информация должна обновляться в процессе разработки программы. Это относится и к спецификации. Хранить ее «рядом» с кодом недостаточно. Необходимо сделать ее исполнимой, то есть внедрить контроль автоматический контроль качества программы в процесс ее разработки. Спецификация, которая не отвечает данному требованию, не является эффективной еще и потому, что она никак не реагирует на изменение модели программы и быстро устаревает.

В целях создания среды разработки надежных программ, реализующих системы со сложным поведением, в рамках НИР предложено совместить сразу несколько подходов к проверке качества ПО. Тестирование и верификация на модели были внедрены в процесс разработки автоматных программ в среде MPS. Также была изучена роль контрактов в преобразовании требований к программе, сформулированных словесно, к форме, пригодной для автоматической проверки.

В качестве направлений дальнейших исследований можно выделить внедрение проверки контрактов во время выполнения программы в предложенной среде контроля качества ПО, изучение ее свойств, а также более глубокое описание контрактного программирования с явным выделением состояний. Отдельной областью для дальнейших исследований является итеративная верификация, позволяющая существенно ускорить получение результата, опираясь на данных предыдущих процессов верификации.

Изложенное позволяет утверждать, что результаты выполнения второго этапа научно-исследовательской работы превышают мировой уровень разработок в рассматриваемой области.

Результаты НИР внедрены в реальном секторе экономики при разработке системы учета дефектов YouTrack, а также могут быть использованы при разработке других программных систем.

Результаты НИР будут внедрены при создании и модернизации научно-образовательных курсов в рамках третьего этапа настоящего Государственного контракта.

3. ПУБЛИКАЦИИ РЕЗУЛЬТАТОВ НИР

По результатам НИР были опубликованы следующие статьи (копии статей приведены в приложении 1):

1. Малаховски Я. М., Шалыто А. А. Реализация конечных автоматов на функциональных языках программирования //Информационно-управляющие системы. 2009. №6, с. 30–33.
2. Клебанов А.А., Степанов О.Г., Шалыто А.А. Применение шаблонов требований к формальной спецификации и верификации автоматных программ//Научно-технический вестник Санкт-Петербургского государственного университета информационных технологий, механики и оптики. 2010. №5(69), с. 91-95.

Кроме этого, по результатам НИР сделаны следующие доклады на конференциях:

1. Малаховски Я.М. Валидация автоматов с переменными на функциональных языках программирования / Труды VII Всероссийской межвузовской конференции молодых ученых. СПб: СПбГУ ИТМО. 2010.
2. Федотов П.В. Рефакторинг автоматных программ / Труды VII Всероссийской межвузовской конференции молодых ученых. СПб: СПбГУ ИТМО. 2010.
3. Борисенко А.А. Повышение качества автоматных программ / Труды VII Всероссийской межвузовской конференции молодых ученых. СПб: СПбГУ ИТМО. 2010.
4. Клебанов А.А. О применимости шаблонов требований к формальной спецификации и верификации автоматных программ / Труды VII Всероссийской межвузовской конференции молодых ученых. СПб: СПбГУ ИТМО. 2010.
5. Клебанов А.А., Степанов О.Г., Шалыто А.А. Применение шаблонов требований к формальной спецификации и верификации автоматных программ /Труды семинара «Семантика, спецификация и верификация программ: теория и приложения». Казань. 2010, с. 124–130.
6. Klebanov A.A. On the Formal Specification of Automata-based Programs via Specification Patterns /Proceeding of the 4 Spring/Summer Young Researcher`s Colloquium on Software Engineering (SYRCoSE 2010). Nizhny Novgorod. 2010, pp.97–99.

В процессе выполнения НИР также получены два свидетельства о регистрации программы для ЭВМ (копии свидетельств приведены в приложении 2):

1. Малаховски Я. М., Шалыто А. А. Библиотека для поддержки автоматного программирования на языке Haskell. Свидетельство о государственной регистрации программы для ЭВМ. № 2010 614196. Дата регистрации – 29.05.2010.
2. Борисенко А.А., Шалыто А. А. Программное средство для автоматической проверки контрактов и темпоральных спецификаций в среде MPS. Свидетельство о государственной регистрации программы для ЭВМ. № 2010 613401. Дата регистрации – 15.06.2010.

ЗАКЛЮЧЕНИЕ

В результате исследований, выполненных на втором этапа работ по Государственному контракту, были проведены следующие работы:

1. Разработана программная библиотека для поддержки автоматного программирования для функционального языка программирования Haskell.
2. Проведена апробация разработанной библиотеки на примере ответственной системы — сервера обмена мгновенными сообщениями.
3. Введено понятие рефакторинга для автоматных программ – такой модификации программ, при которой ее поведение сохраняется.
4. Выделен набор рефакторингов автоматных программ, для каждого рефакторинга дано описание и доказано, что выполнение рефакторинга не изменяет поведение программы.
5. Предложен метод произведения изменений в автоматных программах, уменьшающий число модификаций, которые могут привести к появлению ошибок.
6. Произведено экспериментальное применение разработанного метода модификации автоматных программ.
7. Разработан метод автоматической динамической верификации программ. Предложенный метод основан на известном подходе к верификации программ, в котором используются альтернирующие автоматы, однако позволяет верифицировать гораздо более сложные системы автоматов, нежели традиционный метод Model Checking.
8. Проанализирована применимость данного метода к верификации автоматных программ; в частности, исследованы особенности использования различных алгоритмов обхода альтернирующего автомата.
9. Произведено экспериментальное применение разработанного метода динамической верификации автоматных программ.
10. Предложен подход к программированию по контрактам (контрактному программированию) с явным выделением состояний. При этом отдельно рассмотрены внешние и внутренние контракты.
11. Разработан метод, позволяющий выбирать оптимальный способ формализации спецификаций к автоматным системам в зависимости от характера спецификации и особенностей автоматной системы.
12. Предложен способ интеграции действий по обеспечению соответствия реализованной автоматной системы спецификации в процесс разработки программного обеспечения;
13. Реализован инструмент, позволяющий проводить верификацию автоматной системы во время разработки.
14. Произведено экспериментальное применение разработанных методов и разработанного инструмента на нескольких примерах. В том числе, произведено внедрение результатов исследований в реальный сектор экономики.

Кроме этого, по результатам НИР были опубликованы две статьи в журналах из переченя ВАК и сделано шесть докладов на трех конференциях.

Копия экспертного заключения о возможности опубликования приведена в приложении 3.

Результаты выполненных работ позволяют утверждать, что научно-технический уровень исследований превышает уровень исследований в рассматриваемой области, проводимых в лучших исследовательских центрах мира.

ИСТОЧНИКИ

1. Промежуточный научно-технический отчет о выполнении Государственного контракта по теме "Методы повышения качества при разработке автоматных программ с использованием функциональных и объектно-ориентированных языков программирования" (I этап). СПбГУ ИТМО. 2009.
2. Фаулер М. Рефакторинг: улучшение существующего кода. СПб.: Символ-Плюс, 2004.
3. Kurzweil R. The Singularity Is Near: When Humans Transcend Technology. Penguin, 2006.
4. Кафедра «Технологии программирования». Раздел «Проекты». http://is.ifmo.ru/projects/
5. Поликарпова Н. И., Шалыто А. А. Автоматное программирование. СПб.: Питер, 2010.
6. Козлов В. А., Комалёва В. А. Моделирование работы банкомата. СПбГУ ИТМО, 2006. http://is.ifmo.ru/unimod-projects/bankomat/
7. Балтийский И. А., Гиндин С. И. Моделирование работы банкомата. СПбГУ ИТМО, 2008. http://is.ifmo.ru/unimod-projects/atm
8. Кларк Э., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. М.: МЦНМО, 2002.
9. Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Теоретические исследования поставленных перед НИР задач. СПбГУ ИТМО. 2007. http://is.ifmo.ru/verification/_2007_02_report-verification.pdf
10. Finkbeiner B., Sipma H. Checking Finite Traces Using Alternating Automata //Form. Methods Syst. Des. 2004. 24, 2.
11. Emerson E. A. Temporal and modal logic / In «Handbook of theoretical Computer Science (Vol. B): Formal Models and Semantics». MA: MIT Press, 1990, pp. 995–1072.
12. Vardi M. Y. Alternating Automata and Program Verification / Computer Science Today. Recent Trends and Developments. Vol. 1000 of LNCS. Springer–Verlag. 1995.
13. Vardi M. Y. Alternating Automata: Checking Truth and Validity for Temporal Logics / Proc. 14th International Conference on Automated Deduction. Vol. 1249 of LNCS. Springer–Verlag. 1997.
14. Havelund K., Roşu G. Testing Linear Temporal Logic Formulae on Finite Execution Traces. Technical Report TR 01–08, RIACS. 2001.
15. Mealy G. A. Method to Synthesizing Sequential Circuits // Bell System Technical J. 1955. 34. pp. 1045–1079.
16. Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Выбор направления исследований и базовых компонентов. СПбГУ ИТМО. 2007. http://is.ifmo.ru/verification/_2007_01_report-verification.pdf
17. Риган П., Хемилтон С. NASA: миссия надежна // Открытые системы. 2004, №3, с. 12–17. http://www.osp.ru/text/302/184060.html
18. Вельдер С. Э., Шалыто А. А. Введение в верификацию автоматных программ на основе метода Model checking. 2006. http://is.ifmo.ru/download/modelchecking.pdf
19. Корнеев Г. А., Парфенов В. Г., Шалыто А. А. Верификация автоматных программ. Саратов: СГУ, 2007. http://is.ifmo.ru/verification/_KNIT-2007.pdf
20. Поликарпова Н. И. Объектно-ориентированный подход к моделированию и спецификации сущностей со сложным поведением. СПбГУ ИТМО, 2006. http://is.ifmo.ru/papers/oosuch
21. Курбацкий Е. А., Шалыто А. А. Верификация программ, построенных при помощи автоматного подхода / Материалы международной научно-методической конференции «Высокие интеллектуальные технологии и инновации в образовании и науке». СПбГПУ. 2008, с. 293–296. http://is.ifmo.ru/download/2008-02-25_politech_verification_kurb.pdf

22. Кузьмин Е. В., Соколов В. А. Моделирование спецификация и верификация «автоматных» программ // Программирование. 2008. № 1, с. 2–5. http://is.ifmo.ru/download/2008-03-12_verification.pdf
23. Мейер Б. Объектно-ориентированное конструирование программных систем. М.: Интернет-университет информационных технологий, 2005.
24. Кулямин В.В. Методы верификации программного обеспечения. Институт системного программирования РАН, 2008. http://www.ict.edu.ru/ft/005645/62322e1-st09.pdf
25. Веденеев В. В. Автоматизация тестирования использования программных интерфейсов приложений на основе моделирования конечными автоматами. СПбГУ ИТМО, 2007. http://is.ifmo.ru/testing/vedeneev/
26. Винниченко И. В. Автоматизация процессов тестирования. СПб.: Питер, 2005.
27. Мейер Б. Семь принципов тестирования программ // Открытые системы. 2008. № 7, с. 13–29. http://www.osp.ru/os/2008/07/5478839/
28. Polikarpova N., Ciupa I., Meyer B. A comparative study of programmer-written and automatically inferred contracts / Proceedings of ISSTA 2009: International Symposium on Software Testing and Analysis Chair of Software Engineering. Chicago, IL, USA. 2009.
29. Бромберг И. Автоматизация тестирования // Открытые системы. 2002. №5, с.45–47. http://www.osp.ru/os/2002/05/181457
30. Гуров В. С., Мазин М. А., Шалыто А. А. Текстовый язык автоматного программирования / Тезисы докладов Международной научной конференции, посвященной памяти профессора А. М. Богомолова «Компьютерные науки и технологии». Саратов: СГУ. 2007, с. 66–69. http://is.ifmo.ru/works/_2007_10_05_mps_textual_language.pdf
31. Гуров В. С., Шалыто А. А., Яминов Б. Р. Технология верификации автоматных программ без их трансформации во входной язык верификатора / Международная научно-техническая мультиконференция «Проблемы информационно-компьютерных технологий и мехатроники». Материалы международной научно-технической конференции «Многопроцессорные вычислительные и управляющие системы (МВУС'2007)». Таганрог: НИИ МВС. 2007. Т.1, с. 198–203. http://is.ifmo.ru/verification/_jaminov.pdf

ПРИЛОЖЕНИЕ 1. КОПИИ СТАТЕЙ

ПРОГРАММНЫЕ И АППАРАТНЫЕ СРЕДСТВА

УДК 004.4'242

РЕАЛИЗАЦИЯ КОНЕЧНЫХ АВТОМАТОВ НА ФУНКЦИОНАЛЬНЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

Я. М. Малаховски,
магистрант
А. А. Шалыто,
доктор техн. наук, профессор
Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Рассматриваются вопросы реализации на функциональных языках программирования событийных структурных конечных автоматов, используемых в автоматном программировании. На примерах показаны решения, имеющие преимущества перед реализациями на императивных языках программирования.

Ключевые слова *— конечные автоматы, функциональное программирование, Haskell.*

Введение

Несмотря на все более широкое использование автоматных моделей при разработке программного обеспечения, в настоящее время не известно методов реализации на функциональных языках программирования [1] событийных структурных конечных автоматов, используемых в автоматном программировании [2].

Основное отличие функциональных программ от императивных состоит в том, что функциональная программа представляет собой некоторое выражение (в математическом смысле), а выполнение программы означает вычисление значения этого выражения. При этом слово «вычисление» вовсе не означает, что операции производятся только над числами.

При сравнении «чистого» (pure) функционального и императивного подходов к программированию можно отметить следующие свойства функциональных программ:

- в чистых функциональных программах отсутствуют побочные эффекты, поэтому в них не существует прямого аналога глобальным переменным и объектам императивных языков программирования;
- в функциональных программах не используется оператор присваивания;
- в функциональных программах нет циклов, а вместо них применяются рекурсивные функции;
- выполнение последовательности команд в функциональной программе бессмысленно, поскольку одна команда не может повлиять на выполнение следующей.

Отсюда следует, что при использовании чистых функциональных языков программирования (например, языка *Haskell*) не удается непосредственно применять методы, используемые при реализации конечных автоматов на императивных языках программирования (например, языка *C++*). Это объясняется тем, что состояние автомата, по сути, является глобальной переменной, а обработка последовательностей событий в императивных языках производится при помощи циклов.

В теории конечных автоматов существуют два класса: абстрактные и структурные автоматы. Абстрактные автоматы обычно используются при разработке систем генерации парсеров по контекстно-свободным грамматикам и регулярным выражениям (см., например, работы [3, 4]). Функции переходов в таких задачах обычно строятся не по диаграммам состояний, а по множеству порождающих правил. В свою очередь, реализации на императивных языках программирования, построенные по диаграммам состояний, обычно валидируются сторонними утилитами, а не компилятором.

В событийных системах управления применяются структурные автоматы, ранее использовавшиеся в аппаратных реализациях. В общем случае такие автоматы содержат два типа входных воздействий: события и входные переменные. Неизвестны работы, в которых описано, как на функциональных языках программирования ре-

ализовать автоматы указанного класса, так как для таких автоматов требуется формирование выходных воздействий, а чистые функции не позволяют реализовать их непосредственно. В настоящей работе рассматриваются автоматы, управляемые только событиями. Реализация таких автоматов в функциональном программировании обычно не рассматривается.

Таким образом, авторами решаются следующие задачи: реализация на функциональных языках программирования событийных структурных автоматов, в которых входные переменные отсутствуют, и валидация функций переходов таких автоматов в процессе компиляции.

В работе используется язык Haskell [5–8], так как:

- в отличие от иных подобных языков, он близок к *типизированному лямбда-исчислению;*
- в нем отсутствуют побочные эффекты;
- он не нарушает функциональные концепции при вводе-выводе.

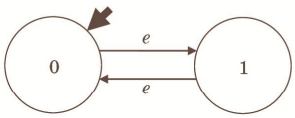
Кроме того, этот язык достаточно популярен в академической среде, и для него существует несколько качественных компиляторов.

Реализация автомата по диаграмме состояний

Суть предлагаемого подхода к реализации функции переходов на функциональных языках программирования состоит в представлении событий автоматов при помощи алгебраических типов данных, а состояний — кортежами или структурами соответствующих переменных. Продемонстрируем различные реализации счетного триггера на примерах.

Обработка одиночных событий.

Пусть требуется реализовать счетный триггер (рис. 1), реагирующий на события. Другими словами, требуется построить конечный автомат с двумя состояниями, кодируемыми нулем и единицей, который управляется кнопкой. Каждое нажатие кнопки порождает событие e. К выходу z конечного автомата подключена лампа. Каждое событие e переводит автомат в состояние $(1 - y)$, где y — текущее состояние. При этом переменная состояния одновременно является выходной переменной ($z = y$). Таким образом, каждое нажатие кнопки будет приводить то к включению лампы, то к ее выключению. Исходный код, реализующий данный счетный триггер, приведен в листинге 1.



■ ***Рис. 1.*** *Диаграмма состояний счетного триггера*

Листинг 1. *Реализация счетного триггера на Haskell.*

```
-- Событие: «Нажатие на кнопку».
data Event = ButtonClick deriving Show
-- Состояния: «Выключено» и «Включено».
data State = LampOff | LampOn deriving Show

-- Функция переходов, изоморфная рис. 1.
gotEvent :: State -> Event -> State
gotEvent LampOff ButtonClick = LampOn
gotEvent LampOn ButtonClick = LampOff

-- Функция, вызываемая системой.
-- Начальное состояние – LampOff.
main = print $ gotEvent LampOff ButtonClick
```

Рассмотренная реализация функции переходов не является единственно возможной. Например, для этой цели можно использовать конструкции *pattern matching*. Однако в этом случае исходный код обычно получается длиннее, поскольку такая конструкция требует частичного дублирования. Поэтому в дальнейшем будет использоваться оператор *case*.

Последовательности событий.

Для того чтобы добавить возможность применения последовательности событий к начальному состоянию автомата, введем функцию *applyEvents*.

Листинг 2. *Функция applyEvents и ее использование.*

```
-- Исходный код листинга 1 за исключением функции main
...
-- Функция, применяющая события к начальному состоянию.
applyEvents :: State -> [Event] -> State
-- Результат = состояние, если событий больше нет.
applyEvents st [] = st
-- Иначе делаем переход и вызываемся рекурсивно.
applyEvents st (e:es) = applyEvents (gotEvent st e) es

-- Новая функция main.
main = print $ applyEvents LampOff [ButtonClick]
```

Обобщение на произвольные типы данных для событий и состояний.

Если бы в реализуемом примере было более одного конечного автомата, то пришлось бы писать несколько функций, аналогичных *applyEvents*. Поэтому можно выделить общую их часть в библиотечный код, пригодный для написания других конечных автоматов. В листинге 3 приведен разработанный библиотечный код, а также реализация счетного триггера с его использованием.

Листинг 3. *Библиотечная функция applyEvents и ее использование.*

```
-- Тип функции переходов.
type SwitchFunc state event = state -> event -> state
```

```
-- Функция, применяющая список событий к начальному
-- состоянию автомата при помощи функции переходов.
applyEvents :: SwitchFunc st ev -> st -> [ev] -> st
applyEvents _ st [] = st
applyEvents swF st (ev:evs) = applyEvents swF
    (swF st ev) evs
-----------------------------------------------------
-- Реализация счетного триггера
-- при помощи приведенного выше библиотечного кода.

-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonClick deriving Show
data TriggerState = LampOff | LampOn deriving Show
-- Функция переходов для счетного триггера.
triggerSwF LampOff ButtonClick = LampOn
triggerSwF LampOn ButtonClick = LampOff

-- Функция, вызываемая системой.
main = print $ applyEvents triggerSwF LampOff [ButtonClick]
```

Отметим, что новая версия функции *applyEvents* (листинг 3) является левой сверткой (одна из стандартных операций функционального программирования) списка событий по функции переходов. Поэтому можно заменить все определение функции *applyEvents* на *applyEvents = foldl*.

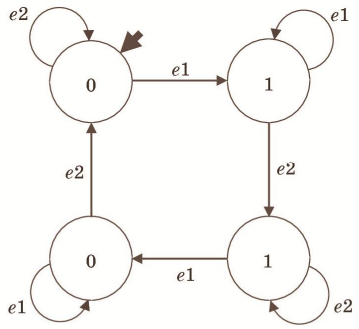
Преимущества функциональной реализации.

Алгебраические типы данных позволяют производить более строгие проверки по сравнению с конструкциями, сходными с конструкцией *enum* императивного языка C. Например, при использовании алгебраических типов данных невозможно случайно проверить на равенство элемент множества состояний с элементом множества событий. Еще одним преимуществом функционального подхода является то, что компилятор способен самостоятельно проверить полноту и непротиворечивость веток оператора *case*, тем самым осуществляя валидацию функции переходов на этапе компиляции.

Выходные воздействия

Функции переходов в предыдущих примерах самостоятельно не производили выходных воздействий, а только возвращали результирующее состояние, которое печаталось на консоль функцией *main*. На практике автоматам требуются и другие классы выходных воздействий для того, чтобы выводить сообщения на экран, отправлять пакеты данных в сеть, обмениваться событиями и т. д.

В качестве примера реализации автомата с выходными воздействиями рассмотрим счетный триггер с четырьмя состояниями (рис. 2). Его отличие от предыдущего триггера заключается в том, что лампа будет включаться или выключаться только после отпускания кнопки, а повторное нажатие или отпускание кнопки не будет производить никакого эффекта.



■ ***Рис. 2.*** *Диаграмма переходов счетного триггера с четырьмя состояниями*

Общую часть реализаций этого примера для экономии места вынесем в отдельный листинг 4.

Листинг 4. *Общая часть реализаций счетного триггера с четырьмя состояниями.*

```
-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonDown
                  | ButtonUp deriving Show

data TriggerState = LampOffButtonUp
                  | LampOffButtonDown
                  | LampOnButtonUp
                  | LampOnButtonDown deriving Show
```

Первый вариант реализации выходных воздействий.

Один из вариантов реализации выходных воздействий — изменить функцию переходов так, чтобы она возвращала не только новое состояние автомата, но и список выходных воздействий некоторого типа. Это может быть как арифметический тип данных, представляющий собой множество возможных выходных воздействий, так и тип *IO ()*. В этом случае ответственность за выполнение выходных воздействий лежит на той части кода, которая вызывает функцию переходов автомата. Достоинством данного подхода является его гибкость. Листинг 5 демонстрирует данный подход.

Листинг 5. *Функция переходов, возвращающая список выходных воздействий.*

```
-- Абстрактный тип функции переходов,
-- возвращающей новое состояние и список выходных воздействий.
type SwitchFunc state input output = state -> input
  -> (state, [output])

-- Функция applyEvents, реализованная через свертку.
applyEvents :: SwitchFunc state input output -> state
  -> [input] -> (state, [output])
applyEvents switchFunc state events =
  foldl (switchToAcc switchFunc) (state, []) events

-- Функция, «конвертирующая» функцию переходов в
```

```
-- аккумулятор выходных воздействий.
switchToAcc :: SwitchFunc state input output -> (state, [output])
  -> input -> (state, [output])
switchToAcc switchFunc (state, output) event =
  (nstate, output ++ noutput)
  where (nstate, noutput) = switchFunc state event

-- Функция переходов для счетного триггера.
triggerSwitchFunc state event = case state of
  LampOffButtonUp -> case event of
    ButtonDown -> (LampOffButtonDown, [])
    _ -> (state, [])
  LampOnButtonUp -> case event of
    ButtonDown -> (LampOnButtonDown, [])
    _ -> (state, [])
  LampOffButtonDown -> case event of
    ButtonUp -> (LampOnButtonUp, [putStrLn «LampOn»])
    _ -> (state, [])
  LampOnButtonDown -> case event of
    ButtonUp -> (LampOffButtonUp, [putStrLn «LampOff»])
    _ -> (state, [])


-- Функция, вызываемая системой.
main = do
  sequence_ o
  putStrLn $ show $ s
  where
    (s, o) = applyEvents triggerSwitchFunc LampOffButtonUp
      [ButtonDown, ButtonUp, ButtonDown, ButtonUp]
```

Второй вариант реализации выходных воздействий.

Проблема реализации выходных воздействий может быть решена также модификацией функции переходов. При этом в качестве возвращаемого значения она будет иметь тип *IO state*, где *state* — тип состояния автомата, а сами выходные воздействия будут выполняться непосредственно в самой функции переходов (листинг 6).

Листинг 6. *Функция переходов, возвращающая список выходных воздействий.*

```
-- Абстрактный тип функции переходов,
-- возвращающей новое состояние и осуществляющей IO.
type SwitchFunc state input output = state -> input -> IO state

-- Функция applyEvents, реализованная через монаду IO.
applyEvents :: SwitchFunc state input output -> state
  -> [input] -> IO state

applyEvents switchFunc state [] = return state
applyEvents switchFunc state (event:eventsTail) = do
  newstate <- switchFunc state event
  applyEvents switchFunc newstate eventsTail

-- Функция переходов для счетного триггера.
triggerSwitchFunc state event = case state of
  LampOffButtonUp -> case event of
    ButtonDown -> return LampOffButtonDown
    _ -> return state
  LampOnButtonUp -> case event of
    ButtonDown -> return LampOnButtonDown
    _ -> return state
  LampOffButtonDown -> case event of
    ButtonUp -> do
      putStrLn «LampOn»
      return LampOnButtonUp
    _ -> return state
  LampOnButtonDown -> case event of
    ButtonUp -> do
      putStrLn «LampOff»
      return LampOffButtonUp
    _ -> return state

-- Функция, вызываемая системой.
main = do
  result <- applyEvents triggerSwitchFunc LampOffButtonUp
    [ButtonDown, ButtonUp, ButtonDown, ButtonUp]
  putStrLn $ show $ result
```

Второй подход предоставляет больше свободы, так как в данном случае функция переходов не является чистой. Данный способ реализации является аналогом подхода, используемого в императивных языках программирования, поскольку все вычисления выполняются строго в контексте *IO*.

Заключение

В работе предложены методы реализации событийных структурных конечных автоматов на языке Haskell. При этом продемонстрированы преимущества этих подходов по сравнению с реализациями на императивных языках программирования. Такими преимуществами являются строгая типизация составных частей конечного автомата и валидация функции переходов компилятором.

Литература

1. **Abelson H., Sussman G.** Structure and Interpretation of Computer Programs. — MIT Press, 1985. — 634 p.
2. **Поликарпова Н. И., Шалыто А. А.** Автоматное программирование. — СПб.: Питер, 2009. — 176 с.
3. **Parsec.** http://www.haskell.org/haskellwiki/Parsec (дата обращения: 09.07.2009)
4. **The Parser Generator for Haskell.** http://www.haskell.org/happy/ (дата обращения: 09.10.2009).
5. **Bird R.** Introduction to Functional Programming using Haskell. — NY.: Prentice Hall, 1998. — 448 p.
6. **Davie A.** Introduction to Functional Programming System Using Haskell. — Cambrige: Cambrige University Press, 1992. — 304 p.
7. **Hudak P., Peterson J., Fasel J.** A Gentle Introduction to Haskell 98. http://www.haskell.org/tutorial/ (дата обращения: 09.07.2009) — 64 p.
8. **Кирпичев Е.** Монады // RSDN Magazine. 2008. N 3. http://www.rsdn.ru/article/funcprog/monad.xml (дата обращения: 09.07.2009).

А.А. Клебанов, О.Г. Степанов, А.А. Шалыто

УДК 004.4'242

ПРИМЕНЕНИЕ ШАБЛОНОВ ТРЕБОВАНИЙ К ФОРМАЛЬНОЙ СПЕЦИФИКАЦИИ И ВЕРИФИКАЦИИ АВТОМАТНЫХ ПРОГРАММ

А.А. Клебанов, О.Г. Степанов, А.А. Шалыто

Верификация на модели *(model checking)* является глубоко проработанной технологией проверки корректности программного обеспечения, однако при этом она недостаточно широко используется на практике. Одной из причин является сложность составления формальных требований на языке темпоральных логик. В настоящей работе описывается подход для записи верифицируемых требований на подмножестве естественного языка в контексте автоматного программирования. В частности, приводится анализ применимости шаблонов требований к формальной спецификации автоматных программ, а также описывается грамматика для вывода требований.
Ключевые слова: автоматное программирование, верификация на модели, темпоральные логики.

Введение

Автоматное программирование [1] – это метод разработки программного обеспечения (ПО), основанный на расширенной модели конечного автомата. В рамках данного подхода программы представляются системой автоматизированных объектов управления, логика поведения которых задается системой взаимодействующих управляющих автоматов.

В ряде работ [2, 3] показано, что к автоматным программам хорошо применима верификация на модели (*Model Checking*) [4]. Суть такой верификации состоит в проверке соответствия модели с конечным числом состояний (*структуры Крипке*) формальной спецификации, заданной в виде набора формул темпоральной логики. При верификации преимуществом автоматного подхода перед традиционными подходами к разработке ПО является высокая степень автоматизации, так как в автоматных программах модель поведения задается априори. Разработаны методы [5–7], позволяющие автоматически преобразовывать как управляющие автоматы в модель, пригодную для верификации, так и построенный верификатором контрпример в автоматную модель. Однако, как при верификации автоматных программ, так и при верификации программ общего вида, существует следующая проблема – необходимость записи формальных требований в виде формул темпоральных логик, работа с которыми достаточно трудоемка и требует значительной математический подготовки.

В работе [8] эта проблема частично решается использованием контрактов [9]. Хотя контракты являются более простым формализмом (и, как следствие, ошибки в подобной формальной спецификации менее вероятны), рассматриваемый подход имеет несколько недостатков. Во-первых, контракты значительно уступают темпоральным логикам в выразительных возможностях. К записи требований они применимы только в том случае, когда необходимо специфицировать свойства инвариантности, предусловия или постусловия. Во-вторых, специфицирование требований, распространяющихся на группу состояний, может быть достаточно трудоемким процессом, поскольку для каждого состояния из группы потребуется разработать свою спецификацию. Таким образом, описанную выше проблему нельзя считать полностью решенной.

В настоящей работе описывается подход к записи требований, скрывающий сложность темпоральных логик. Предлагается записывать требования на подмножестве естественного языка, заданного приводимой ниже формальной грамматикой. Грамматика основывается на наборе шаблонов требований [10, 11] – обобщенном описании (формальном и на естественном языке) часто встречающихся ограничений на допустимые последовательности состояний в модели системы с конечным числом состояний. Таким образом, для каждого полученного требования существует эквивалентная формальная запись, позволяющая осуществить верификацию.

Актуальность применения шаблонов требований в контексте автоматного программирования отмечается в работе [2]: «… важным является вопрос о шаблонах (структуре) темпоральных свойств, наиболее применимых и адекватных для верификации автоматных программ. Наличие таких шаблонов позволяло бы говорить о классах темпоральных свойств автоматных моделей, что, несомненно, облегчало бы построение технологической схемы проверки автоматных программ на корректность относительно спецификации». Однако в указанной работе выделяется только одно требование, являющееся частным случаем существующего шаблона, и дальше этот вопрос никак не прорабатывается.

В начале настоящей работы кратко описываются шаблоны требований, затем приводится анализ их применимости к спецификации автоматных программ, и, наконец, вводится формальная грамматика для записи требований. В заключении сделаны выводы по работе.

Шаблоны требований

В работах [10, 11] предлагается система шаблонов требований, разработанная на основе спецификаций для программ общего вида. Шаблоны можно классифицировать в соответствии с иерархической структурой, основанной на их семантике. В настоящее время выделено восемь основных шаблонов («Отсутствие», «Существование», «Всеобщность», «Ограниченное существование»,

«Предшествование», «Ответ», «Цепное предшествование», «Цепной ответ»), которые делятся на две группы – «Наличие» и «Порядок». В группу «Наличие» входят шаблоны, описывающие наличие или отсутствие состояний, в которых выполняется заданное требование. В группу «Порядок» – описывающие порядок состояний.

Описание шаблона состоит из его имени (или списка имен), цели, записи на различных формализмах (LTL, CTL и т.п.), примера использования и связи с другими шаблонами.

Каждое требование имеет *ограничение* – ту часть пути исполнения, на котором это требование должно выполняться. Всего выделены пять видов ограничений.

1. Глобально – на всем пути исполнения.
2. До – на пути до заданного состояния.
3. После – на пути после заданного состояния.
4. Между – на пути между двумя заданными состояниями.
5. После-до – аналогично ограничению «Между», однако наличие правой границы интервала не является обязательным.

Отметим, что стандартно для формализмов, ориентированных на состояние, интервал, на котором должно выполняться требование, замкнут на левом конце и открыт на правом.

В табл. 1 приведен пример шаблона «Всеобщность». Оригинальный пример использования, предложенный в работе [10], заменен примером более применимым в контексте автоматного программирования. В этом состоит задача адаптации системы шаблонов для автоматного программирования.

<table>
<tr><td colspan="2">Цель</td><td colspan="2">Используется для описания части пути исполнения системы, в которой содержатся только те состояния, в которых выполняется необходимое требование. Известен также как «Впредь» и «Всегда».</td></tr>
<tr><td rowspan="7">Запись</td><td rowspan="6">LTL</td><td>Ограничение</td><td>Запись</td></tr>
<tr><td>Глобально</td><td>□(P)</td></tr>
<tr><td>До R</td><td>◊R → (P U R)</td></tr>
<tr><td>После Q</td><td>□(Q → □(P))</td></tr>
<tr><td>Между Q и R</td><td>□((Q & !R & ◊R) → (P U R))</td></tr>
<tr><td>После Q до R</td><td>□(Q & !R → (P W R))</td></tr>
<tr><td rowspan="6">CTL</td><td>Ограничение</td><td>Запись</td></tr>
<tr><td>Глобально</td><td>AG(P)</td></tr>
<tr><td>До R</td><td>A[(P | AG(!R)) W R]</td></tr>
<tr><td>После Q</td><td>AG(Q → AG(P))</td></tr>
<tr><td>Между Q и R</td><td>AG(Q & !R → A[(P | AG(!R)) W R])</td></tr>
<tr><td>После Q до R</td><td>AG(Q & !R → A[P W R])</td></tr>
<tr><td colspan="2">Пример использования</td><td colspan="2">Этот шаблон может быть применен для описания общих свойств модели в целом или отдельной группы состояний. Например, в том случае, когда необходимо выразить свойство вида: «Если автомат находится в состоянии s, то верно свойство P».
При использовании ограничения «Глобально» подстановка темпоральных выражений <>f или AF(f) в качестве параметра P позволяет выразить свойство справедливости.</td></tr>
<tr><td colspan="2">Связь с другими шаблонами</td><td colspan="2">Этот шаблон тесно связан с шаблонами «Отсутствие» и «Существование». Наличие состояния, в котором выполняется требование, может рассматриваться как отрицание его отсутствия.</td></tr>
</table>

Таблица 1. Шаблон «Всеобщность»

Применимость шаблонов требований к спецификации автоматных программ

Рассмотрим вопрос применимости шаблонов требований к формальной спецификации автоматных программ. Для этого проанализируем требования к различным программам, разработанным в СПбГУ ИТМО, Ярославском государственном университете, ОАО «Концерн «НПО «АВРОРА» и доступным на сайте [12], и проверим, как они выражаются при помощи шаблонов. Пример организации промежуточных результатов анализа приводится в табл. 2. В столбцах «Требование» и «Исходная формальная запись» приводятся оригинальные требования из источника (столбец «Источник»), записанные на естественном языке и одном из формализмов соответственно. В столбце «Шаблон,

Ограничение» приводится запись шаблона, подстановка необходимых утверждений в который, даст эквивалентную исходной формальную запись. В случае необходимости приводится формальное доказательство эквивалентности.

Требование	Исходная формальная запись	Шаблон, Ограничение	Источник
Если произошла поломка нагревателя или одного из клапанов, то кофеварка (основной автомат A_0) обязательно перейдет в состояние 5.	$AG((y_{31} = 4 \mid y_{32} = 4 \mid y_2 = 4) \& y_0 = 2 \rightarrow A(y_0 = 2 \text{ U } y_0 = 5))$	Ответ (ограниченный), AG(P → A(S)), P: $(y_{31} = 4 \mid y_{32} = 4 \mid y_2 = 4) \& y_0 = 2$, S: $y_0 = 2 \text{ U } y_0 = 5$	2

Таблица 2. Анализ применимости шаблонов требований к спецификации автоматных программ

Всего было рассмотрено 118 требований и их вариантов из 20 источников. Установлено, что 85% требований покрывается пятью шаблонами. Оставшиеся 15% не удается записать при помощи шаблонов из-за ряда причин, таких как ограниченность системы шаблонов и некоторых особенностей конкретной автоматной модели рассматриваемого приложения. Процентное соотношение между использованными шаблонами приведено на рис. 1, а, между использованными ограничениями – на рис. 2.

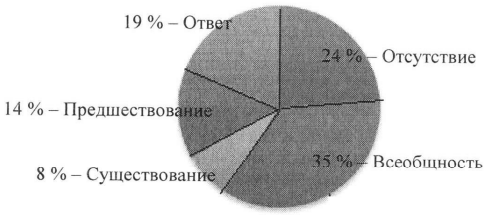


Рис. 1. Процентное соотношение между использованными шаблонами

Рис. 2. Процентное соотношение между использованными ограничениями

Метод записи верифицируемых требований на естественном языке

Существуют различные подходы к выделению формальных (верифицируемых) требований из спецификаций, записанных на естественном языке. Отметим два основных направления [13] – синтаксический разбор текстов и использование формальных грамматик, ограничивающих естественный язык.

В настоящей работе предлагается использовать формальную грамматику, которая приводится в табл. 3. Как отмечалось выше, грамматика основывается на основных шаблонах требований и их вариантах. Это позволяет иметь запись требования, как на естественном языке, так и на любом из формализмов, запись на которых разработана для шаблонов. Моноширинным шрифтом выделены указатели места заполнения шаблона реальными требованиями.

<требование>	::= <ограничение> <шаблон>
<ограничение>	::= «Для любого состояния верно, что» | «До состояния, в котором Q, верно что» | «После состояния, в котором Q, верно что» | «Между состоянием, в котором Q, до состояния, в котором R, верно что» | «После состояния, в котором Q, до состояния, в котором R, верно что»
<шаблон>	::= <отсутствие> | <всеобщность> | <существование> | <предшествование> | <ответ> | <ответ на следующем шаге>
<отсутствие>	::= «никогда не выполняется P»
<всеобщность>	::= «всегда выполняется P»
<существование>	::= «когда-нибудь выполнится P»
<предшествование>	::= «всегда верно, что если выполнено P, то до этого было выполнено S»
<ответ>	::= «всегда верно, что если выполнится P, то когда-нибудь выполнится S»
<ответ на следующем шаге>	::= «всегда верно, что если выполнится P, то в следующем состоянии выполнится S»

Таблица 3. Грамматика верифицируемых требований на подмножестве русского языка

В качестве примера рассмотрим реальное требование к системе управления кофеваркой, которое приводится в работе [3]: «Система управления кофеваркой никогда не попадет в такое состояние, в котором она не реагирует ни на события системного таймера, ни на нажатие кнопок «ОК» и «С». В автоматной модели кофеварки требованию «Никак не реагирует ни на события системного таймера, ни на нажатие кнопок «ОК» и «С» соответствует предикат act = end. Наречие «никогда» подсказывает, что должен быть использован шаблон «Отсутствие» с ограничением «Глобально».

Выполним порождение:

<требование> → <ограничение> <шаблон> → Для любого состояния верно, что <шаблон> → Для любого состояния верно, что <отсутствие> → Для любого состояния верно, что никогда не выполняется P

Подставив вместо P реальное требование, получим искомое формальное требования на естественном языке: «Для любого состояния верно, что никогда не выполняется act = end». Этому требованию на языках CTL и LTL соответствуют выражения AG(! act = end) и □(! act = end). Эти выражения совместно с автоматной моделью системы подаются на вход инструментальному средству, осуществляющему проверку на модели.

Заключение

Запись требований в виде формул темпоральной логики является неотъемлемой частью верификации на модели. В настоящей работе в рамках автоматного программирования предложен подход, который, во-первых, значительно упрощает этот процесс, а во-вторых, минимизирует число потенциальных ошибок в самой спецификации.

В качестве приоритетного направления дальнейших исследований можно выделить инструментальную поддержку предложенного подхода. В работе [8] показывается, как система метапрограммирования *JetBrains MPS* [14] может быть использована как для разработки, так и для верификации автоматных программ. На текущий момент требования записываются в виде формул темпоральных логик, что может быть улучшено на основе описанного выше решения. Аналогично работам [13, 15, 16] может быть реализован помощник для интерактивного выбора необходимого ограничения и шаблона. Наконец, остается открытым ряд теоретических вопросов, таких как расширение набора шаблонов или запись существующих шаблонов на новом формализме.

Исследование проводится в рамках Федеральной целевой программы «Научные и научно-педагогические кадры инновационной России на 2009–2013 годы»

Литература

1. Поликарпова Н.И., Шалыто А.А. Автоматное программирование. – СПб: Питер, 2009. – 176 с. [Электронный ресурс]. – Режим доступа: http://is.ifmo.ru/books/_book.pdf, своб.
2. Васильева К.А., Кузьмин Е.В. Верификация автоматных программ с использованием LTL // Моделирование и анализ информационных систем. – 2007. – Т. 14. – № 1. – С. 3–14.
3. Кузьмин Е.В., Соколов В.А. Моделирование, спецификация и верификация «автоматных» программ // Программирование. – 2008. – № 1. – С. 38–60.
4. Кларк Э., Грамберг О., Пелед Д. Верификация моделей программ. Model Checking. – М.: Изд-во МЦНМО, 2002. – 416 с.
5. Гуров В.С., Яминов Б.Р. Технология верификации автоматных моделей программ без их трансляции во входной язык верификатора // Тезисы научно-технической конференции «Научное программное обеспечение в образовании и научных исследованиях». – СПбГУ ПУ, 2008. – С. 36–40 [Электронный ресурс]. – Режим доступа: http://is.ifmo.ru/download/2008-02-25_politech_tezis.pdf, своб.
6. Кулаков А.Ю., Шалыто А.А. Автоматизация верификации визуальных автоматных программ / Материалы XV Международной научно-методической конференции «Высокие интеллектуальные технологии и инновации в образовании и науке». – СПбГПУ, 2008. – С. 296–297 [Электронный ресурс]. – Режим доступа: http://is.ifmo.ru/download/2008-02-25_politech_tezis.pdf, своб.
7. Kurbatsky E. Verification of Automata-Based Programs // Proceedings of the Second Spring Young Researchers Colloquium on Software Engineering. – 2008. – V. 2. – P. 15–17 [Электронный ресурс]. – Режим доступа: http://is.ifmo.ru/verification/_kurbatsky_syrcse.pdf, своб.
8. Степанов О.Г. Методы реализации автоматных объектно-ориентированных программ. Диссертация на соискание ученой степени кандидата технических наук. – СПбГУ ИТМО, 2009. – Режим доступа: http://is.ifmo.ru/disser/stepanov_disser.pdf, своб.
9. Мейер Б. Объектно-ориентированное конструирование программных систем. – М.: Русская редакция, 2005. – 1204 с.

10. Dwyer M.B., Avrunin G.S., Corbett J.C. Property Specification Patterns for Finite-state Verification // Proceedings of the 2nd Workshop on Formal Methods in Software Practice. – 1998.
11. Dwyer M.B., Avrunin G.S., Corbett J.C. Patterns in Property Specifications for Finite-state Verification // Proceedings of the 21st International Conference on Software Engineering. – 1999.

ПРИЛОЖЕНИЕ 2. КОПИИ СВИДЕТЕЛЬСТВ О РЕГИСТРАЦИИ ПРОГРАММ ДЛЯ ЭВМ

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2010614196

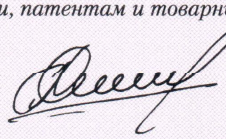
Библиотека поддержки автоматного программирования для языка Haskell

Правообладатель(ли): ***Государственное образовательное учреждение высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» (RU)***

Автор(ы): ***Малаховски Ян Михайлович, Шалыто Анатолий Абрамович (RU)***

Заявка № **2010612469**
Дата поступления **5 мая 2010 г.**
Зарегистрировано в Реестре программ для ЭВМ
29 июня 2010 г.

Руководитель Федеральной службы по интеллектуальной собственности, патентам и товарным знакам



Б.П. Симонов

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2010615076

Программное средство для автоматической проверки контрактов и темпоральных спецификаций в среде MPS

Правообладатель(ли): ***Государственное образовательное учреждение высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» (RU)***

Автор(ы): ***Борисенко Андрей Андреевич, Шалыто Анатолий Абрамович (RU)***

Заявка № **2010613401**
Дата поступления **15 июня 2010 г.**
Зарегистрировано в Реестре программ для ЭВМ
5 августа 2010 г.

Руководитель Федеральной службы по интеллектуальной собственности, патентам и товарным знакам

Б.П. Симонов

ПРИЛОЖЕНИЕ 3. КОПИЯ ЭКСПЕРТНОГО ЗАКЛЮЧЕНИЯ О ВОЗМОЖНОСТИ ОПУБЛИКОВАНИЯ

УТВЕРЖДАЮ
Начальник НИЧ
_____ Л. М. Студеникин
«13» 09 2010 г.

ЭКСПЕРТНОЕ ЗАКЛЮЧЕНИЕ О ВОЗМОЖНОСТИ ОПУБЛИКОВАНИЯ

Экспертная комиссия Санкт-Петербургского государственного университета информационных технологий, механики и оптики Министерства образования и науки РФ рассмотрела на заседании (протокол № 1 от «13» 09 2010 г.)

научно-технический отчет о выполнении 2 этапа по государственному контракту №П2373 от 18 ноября 2009 года по проекту «Методы повышения качества при разработке автоматных программ с использованием функциональных и объектно-ориентированных языков программирования»
(вид, название материала, Ф.И.О. автора (ов))

подтверждает, что в материале не содержатся сведения, предусмотренные разделом 3 Положения 88.
(содержатся ли сведения, предусмотренные разделом 3 Положения 88)

На публикацию материала не следует получить разрешение Министерства образования и науки РФ.

Заключение: материал разрешен к публикации в открытой печати РФ

Председатель комиссии _____ / Зубок Д.А. /

Представитель ОИСиНТИ _____ / Горкина Н.М. /

Секретарь комиссии _____ / Павлов В.В. /