МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

Факультет Информационных технологий и программирования
Направление Прикладная математика и информатика
Специализация Математические модели и алгоритмы в разработке программного обеспечения
Академическая степень Бакалавр прикладной математики и информатики

Кафедра Компьютерных технологий **Группа** 4539

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

Построение автоматной модели по спецификации на примере банковского сервера аутентификации

Автор квалификационной работы **Д. М. Пенькин**

Научный руководитель **А. А. Шалыто**

Санкт-Петербург
2009 г.

Оглавление

Введение

В настоящее время все более и более сложные и ответственные процессы автоматизируются, а порой и полностью выполняются на компьютерах, а человеку остается только наблюдать за корректностью их работы. Программы и программные комплексы проходят многостадийные тестирования, использующие разные методики (начиная с функционального тестирования, стресс-тестирования, заканчивая более сложным и эффективным методом – верификацией, и т. д.), выявляющиеся ошибки исправляются. Однако зачастую ошибки возникают уже не во время разработки, а в период использования программного продукта. В некоторых областях такие ошибки могут нанести большой экономический ущерб или даже привести к гибели людей. Примерами таких ситуаций могут быть:

- Падение ракеты *«Ариан-5»*. Первый испытательный полет этой ракеты состоялся 4 июня 1996 года и был неудачным. На 39-й секунде полета ракета потерпела крушение по причине неисправности в управляющем программном обеспечении, которая считается самой дорогостоящей компьютерной ошибкой в истории. Часть программного обеспечения ракеты была заимствована из ее предыдущей версии – «Ариан-4», но при переносе этой системы для использования на новой ракете, разработчиками не были учтены все особенности. Из-за иной траектории выведения ракеты через 30 секунд после запуска значение горизонтальной скорости превысило установленные в программе ограничения и вызвало сбой в работе компьютера. Это привело к выдаче ложной команды на отклонение сопел ускорителей, а позже и основного двигателя. В результате на 39-й секунде полета ракета разрушилась под действием аэродинамических сил. На ракете *«Ариан-4»* сбоя не происходило из-за отличий в характеристиках траектории полета [1].
- Медицинский ускоритель *Therac-25*. Доза лечебного облучения, предназначенного для пациентов, была превышена в 100 раз.

Неисправность в программном обеспечении прибора была зафиксирована только через полгода; за это время было зафиксировано шесть смертельных случаев вследствие переоблучения. Основная ошибка в собственной многозадачной операционной системе реального времени, устанавливавшейся на контроллере ускорителя, заключалась в отсутствии синхронизации и возможного вследствие этого так называемого состояния гонки (ошибка программирования многозадачной системы, при которой работа системы зависит от того, в каком порядке выполняются части кода). [2]

- Сбой в телефонной сети *AT&T* 15 января 1990 г. Ошибка в новой версии прошивки междугородних коммутаторов привела к тому, что коммутатор перезагружался, если получал специфический сигнал от соседнего коммутатора. Проблема состояла в том, что этот сигнал генерировался в тот момент, когда коммутатор восстанавливал свою работу после сбоя. В один прекрасный день один из коммутаторов в Нью-Йорке перезагрузился и подал тот самый злополучный сигнал. Вскоре 114 соседних коммутаторов непрерывно перезагружались каждые шесть секунд, а 60 тысяч человек остались без междугородней связи на девять часов, пока инженеры не установили на коммутаторы предыдущую версию прошивки.

В данной работе рассматривается иной подход к повышению качества программ, более того, строится метод автоматического построения по предъявляемым требованиям таких программ, которые сразу соответствовали бы этим требованиям. Таким образом, единственной задачей проверки при проектировании системы останется составление и проверка корректной спецификации к этой системе.

Помимо этого будет приведено построение автоматной модели банковского сервера аутентификации, как пример применения синтезирующего алгоритма к прототипу реальной задачи.

Глава 1. Основные понятия и обзор существующих исследований

1.1. Верификация и синтез

Спецификация – инженерный термин, обозначающий набор требований и параметров, которым удовлетворяет некоторая сущность. В информатике, в частности, при разработке программ, однако, под спецификацией подразумевается формальная спецификация – это математической описание программной или аппаратной системы, которая может быть реализована в соответствии с этим описанием. Специфицируется, что должна делать система, но не то, как она должна это делать. Если существует спецификация системы, то можно применить методы формальной верификации, о которых речь пойдет ниже, чтобы продемонстрировать, что система удовлетворяет (или будет удовлетворять) спецификации. Таким образом, можно проверить, будет ли конкретная спроектированная модель удовлетворять требованиям после реализации [3].

В настоящее время для выявления ошибок и повышения качества разработки программ применяются четыре основных метода:

- имитационное моделирование – тестирование прототипа программы;
- тестирование полной программы – применяется после окончательного написания программы, однако, как известно, даже при положительном результате не гарантирует отсутствия ошибок;
- дедуктивный метод (метод доказательства теорем) – является очень трудоемким и к тому же привязан к семантике языка программирования;
- метод проверки модели (*model checking*) – метод автоматической верификации программных систем – проверка программы на

соответствие спецификации, которая может быть выражена, например, на языке темпоральной логики.

В последнее время очень активно развивается автоматный подход к программированию [4], который является очень эффективным средством при построении систем логического управления, а также «реактивных» систем [5, 6]. При использовании технологии автоматного программирования исключается проблема адекватности модели программе, поскольку взаимодействующие автоматы, описывающие логику программы, уже являются адекватной конечной моделью, по которой формально и изоморфно можно построить программу. Помимо этого, свойства программной системы в виде автоматов формулируются и специфицируются в частности, на языках темпоральных логик, естественным образом [7]. С другой стороны, проверка этих свойств осуществляется в терминах, которые так же естественно вытекают из автоматной модели программы.

Все вышеперечисленные эти методы позволяют проверить лишь уже написанный или, по крайней мере, спроектированный программный модуль на предмет соответствия спецификации, а не синтезировать его из спецификации напрямую. Этой проблеме и будет посвящена данная работа.

Построение программы по спецификации является одной из известных фундаментальных проблем. Помимо теоретического интереса, этот вопрос имеет огромное практическое значение, поскольку нахождение хорошего алгоритма синтеза программы по строго заданным требованиям приведет к кардинальному повышению надежности разработки самых разнообразных сложных систем.

1.2. Спецификация на языке темпоральной логики

Первые работы по исследованию возможности автоматического синтеза программ появились в конце 1980-х годов. В них рассматривался процесс генерации так называемых «закрытых» систем – систем, не взаимодействующих с окружающей средой, на основе спецификации на

языке темпоральной логики [8]. В этом случае программа могла быть «извлечена» из конструктивного доказательства выполнимости формул спецификации. При использовании доказательного метода синтеза программ используется два подхода: логический и аналитический.

При логическом подходе спецификация трактуется как формулировка теоремы, утверждающей существования решения задачи. Синтез программы состоит в поиске доказательства этой теоремы существования в некотором конструктивном логическом исчислении. Если такое доказательство удается построить, то существуют формальные правила, позволяющие преобразовать доказательство в программу, находящую решение задачи. Важно заметить, что существует универсальная контролирующая процедура, которая проверяет правильность доказательства и применения правил перехода от доказательства к программе.

При аналитическом подходе спецификация трактуется как уравнение относительно программы. Такое уравнение можно подвергать символическим преобразованиям, при которых символ неизвестной программы «рассыпается» на систему вспомогательных неизвестных, а они, в свою очередь, заменяются либо другими неизвестными, либо конкретными программными конструкциями. Таким образом, по мере преобразования синтезируемая программа постепенно «прорастает» в тексте спецификации, в результате превращая ее в законченный программный текст. Так же, как при предыдущем подходе, правильность выполнения символических преобразований может формально проверяться на каждом шагу.

К сожалению, этот метод не подходит для синтеза «открытых» систем (взаимодействующих с окружающей средой), поскольку выполнимость подразумевает существование такого окружения, в котором программа удовлетворяла бы формулам, однако программа не может ограничивать окружение [9].

Более поздние работы разрешили проблему синтеза «открытых» систем из спецификаций, заданных на языке линейной темпоральной логики

(*Linear Temporal Logic, LTL*). Проблема осуществимости синтеза сведена к проверке непустоты иерархического автомата, а программа с конечным числом состояний могла быть сгенерирована из бесконечного дерева, принимаемого на вход автоматом [10, 11]. В этих же работах показано, что в общем случае задачи определения осуществимости и, собственно, генерации программы из спецификации на *LTL* являются *2EXPTIME*-полными. Синтез программ из спецификаций на языке ветвящейся темпоральной логики (*Computational Tree Logic*, *CTL*) и комбинации *LTL* и *CTL*, имеющей обозначение *CTL**, является *EXPTIME*- и *2EXPTIME*-полной задачей соответственно, что является результатом работы [12].

Исследованию подвергались также и распределенные открытые системы [13]. Предлагается архитектура, представляющая собой множество процессоров и схему их соединений, решением проблемы синтеза для которой является набор программ с конечным числом состояний, по одной на каждый из процессоров, поведение которых в совокупности удовлетворяет спецификации. В работе [13] показано, что осуществимость синтеза для заданной спецификации на заданной архитектуре неразрешима. Предыдущие исследования подразумевали простую архитектуру с одним процессором, поэтому проблема осуществимости была разрешимой. Что касается самого синтеза, то рассматривается следующий метод: сначала строится однопроцессорная программа, которая затем разбивается на множество подпрограмм для каждого из процессоров архитектуры. Проблема декомпозиции за конечное число шагов проще, чем разработка: показано, что возможность декомпозиции программы с конечным числом состояний на множество подпрограмм для заданной архитектуры является разрешимой задачей.

В заключение стоит заметить, что в упомянутых работах по синтезу программы из спецификации на языке темпоральной логики предполагается, что система и окружение поочередно делают «одношаговые ходы».

1.3. Спецификация в виде диаграмм последовательности сообщений

Диаграмма последовательности сообщений (*Message Sequence Chart, MSC*) – диаграмма, на которой показаны взаимодействия объектов, упорядоченные по времени их проявления. Используется, в частности, в языке *UML*. Основными элементами диаграммы последовательностей являются обозначения объектов (прямоугольники), вертикальные линии, отображающие течение времени при деятельности объекта, и стрелки, показывающие выполнение действий объектами. В дальнейшем будем обозначать диаграмму последовательности сообщений по первым буквам ее английского названия – *MSC,* а также называть сценарием программы. Применительно к синтезу программ, объектами *MSC* являются компоненты программы, которые могут обмениваться информацией – сообщениями в терминах *MSC*.

Каждая диаграмма, составляющая спецификацию, имеет предусловие для входа – только при выполнении этого условия компоненты программы будут взаимодействовать согласно именно этой диаграмме. В ходе исследований в данной сфере возникли два направления: взаимоисключающие или пересекающиеся сценарии. Отличие состоит в том, может ли программа «находиться» одновременно в двух диаграммах последовательности сообщений.

1.3.1. *ROOM*-диаграммы

В работе [14] рассматривается метод, в котором *MSC* разделяются на базовые (экзистенциальные), обозначаемые *bMSC*, и высокоуровневые – *hMSC*. Последние предоставляют необходимые операторы описания композиции и иерархического упорядочения *bMSC*. Семантика такой спецификации предполагает взаимоисключающие диаграммы – во время исполнения программа всегда находится ровно в одной *bMSC*.

В результате синтеза, описанного в работе [14], генерируются структурные и поведенческие компоненты *ROOM*-моделей (*Real-Time Object Oriented Modeling*) [15]. Структурными компонентами являются деятели, протоколы, порты и связи, а поведенческими – вариант диаграммы состояний без параллелизма. Однако семантика с взаимоисключением *bMSC* позволяет реализовать только очень примитивный алгоритм синтеза.

1.3.2. Синтез из диаграмм трассировки

В проекте *SCED* (название применяемого в проекте редактора трассировочных диаграмм – *SCenario EDitor*) [16, 17] семантика используемого *MSC*-языка разрешает пересекающиеся сценарии – система во время исполнения может находиться в нескольких сценариях одновременно. Это предположение делает синтез более сложным. Синтезирующий алгоритм для выбранного объекта генерирует автомат, который, по сути, является реализацией раннего алгоритма Бирмана-Кришнасвами (*Biermann and Krishnaswamy*), конструирующего программу из примеров расчета. Суть этого метода состоит в следующем: пользователь выполняет расчеты, которые система запоминает, а после этого автоматически синтезирует минимальную по количеству состояний программу, способную произвести те же расчеты, что были выполнены пользователем [18]. Более того, в работе [18] доказывается, что если пользователь задумал программу P и в качестве примера привел расчеты T_1, T_2, ..., производимые программой P в любом подмножестве ее запусков, то после конечного числа таких примеров система сможет синтезировать программу P_0, которая будет способна выполнить те же расчеты, что и программа P.

Синтезирующий алгоритм в проекте *SCED* модифицирует результаты работы [18] для использования стандартных (экзистенциальных) диаграмм последовательности сообщений. Язык *MSC* был расширен за счет условных структур и циклов, подсценариев, формальных утверждений, действий и состояний.

Серьезным ограничением метода *SCED* является то, что синтезированный автомат должен быть детерминированным. В некоторых случаях это приводит к синтезу автоматов с дизъюнктными наборами состояний. К тому же, в синтезирующем алгоритме неполностью определено начальное состояние, что может привести к генерации невыполнимых автоматов, если спецификация была недостаточно определенной. Для синтезированных автоматов с дизъюнктным набором состояний невозможно определить начальное состояние, не нарушая детерминированность автомата [17].

Помимо указанных недостатков, с помощью *SCED*-метода нельзя описать частичный порядок в трассировках в терминах *MSC*, например, оставить события, относящиеся только к одному из объектов, но не к другим, позволяя всем остальным событиям (незадействованным) случаться в любое время [16]. Это важно для обеспечения должного уровня абстракции при составлении спецификации программы, однако делает синтез программы значительно более сложным.

1.3.3. Верификация временных диаграмм

Временные диаграммы [19] предоставляют графический язык спецификаций, который особенно подходит для описания дизайна аппаратных систем. В работе [20] приведено формальное определение временных диаграмм и их семантики на основе трансляции в темпоральную логику. Временные диаграммы способны выразить частичный порядок сообщений, а также ограничения в виде причинно-следственных связей сообщений. Помимо этого, в работе [20] показано, что получаемый тип формулы на языке темпоральной логики имеет эффективную процедуру проверки модели (*Model Checking Procedure*).

1.3.4. Трансляция спецификации MSC в автоматную модель

В 1999 году немецкими исследователями был разработан способ трансляции *MSC* в диаграмму состояний [21], являющуюся, по сути, автоматную модель программы, такую, какая, например, используется в современном программном средстве *UniMod* [22]. В работе [21] описывается техника записи спецификации в таком *MSC*-подобном виде, который имел бы идентичную с диаграммами состояний семантику. Эта техника позволила бы осуществить автоматическую трансляцию *MSC* в готовую автоматную модель. Компонента, записанная в виде *MSC* или же диаграммы состояний, отражает его, возможно, недетерминированное поведение. Недетерминированная семантика позволяет обрабатывать абстрактные спецификации и конкретные предписания на уровне реализации с помощью одного общего математического аппарата. Более того, такой подход позволяет создавать смешанную спецификацию, состоящую из диаграмм последовательности сообщений и диаграмм состояний для разных компонент системы. Такое свойство может быть полезно, например, когда различные компоненты системы находятся на разных стадиях разработки и реализации.

Проектируемая система представляется в виде множества компонент и направленных каналов. Компоненты, которым требуется обмениваться сообщениями, должны быть связаны этими каналами. Каждая компонента постоянно прослушивает свой входной канал, читает входящие сообщения, вычисляет каким-либо способом свой ответ и отправляет в выходной канал. Кроме того, подразумевается, что в системе присутствуют глобальные часы. Набор сообщений, прошедший через канал в течение отрезка времени, называется историей канала. Основным предположением метода является асинхронность общения компонент: отправляющая сообщение компонента никогда не блокируется его получателем. Добиться такого поведения можно, например, с помощью буферизации сообщений.

Для данной спецификации S определим, как она соотносится с диаграммой последовательности сообщений M. Каждое такое отношение определяет возможную интерпретацию M. Существуют два вопроса, связанные с семантикой любой данной *MSC*:

- какие последовательности сообщений она представляет;
- когда эти последовательности случаются во время выполнения программы.

Если ответ на первый вопрос содержится в самих диаграммах, то второй вопрос стоит отдельного рассмотрения. Предполагается, что система S характеризуется набором историй каналов V. Авторы работы [21] предлагают различать следующие интерпретации *MSC* относительно S:

- точная – последовательности взаимодействий, указанные в M, случаются ровно один раз во всех $v \in V$, а все остальные взаимодействия невозможны;
- экзистенциальные – существуют элементы $v \in V$, выражающие последовательности взаимодействий, указанные в M;
- универсальные все $v \in V$ выражают последовательности взаимодействий, указанные в M;
- условие инициации – говорят, что M_0 инициирует M_1, если, после того, как последовательность взаимодействий, описываемая в M_0, случилась в одной из $v \in V$, то через конечное время в этой же v произойдет и последовательность, описываемая в M_1;
- отрицание – используя логическое отрицание, каждой из перечисленных выше интерпретаций можно сопоставить противоположную.

Предполагается, что каждая из таких интерпретаций может использоваться в течение процесса разработки программы. Например, для трансляции из *MSC* в диаграммы состояний используется точная интерпретация; для тестирования и документации наиболее часто

используются экзистенциальная и универсальная интерпретации, а также их отрицания. Авторы делают акцент на том, что на какой стадии ни применялись бы *MSC*, необходимо ясно представлять, с какой интерпретацией это делается.

Как уже было сказано, для трансляции в диаграмму состояний авторами работы [21] была выбрана точная интерпретация MSC. Далее начинается процесс трансляции в предположении, что каждой компоненте *MSC* будет соответствовать автомат в результирующей программе. Каждое из условий *MSC* преобразуется в состояние этого автомата. Помимо этого, необходимо определить начальное состояние каждого из полученных состояний. Для получения автомата последовательно проделываются следующие действия (рассмотрим их для одной из компонент – `C`):

1) *Проекция*. Во время этой фазы каждая из данных *MSC* «проецируется» на компоненту `C`. Таким образом, убираются все остальные компоненты и каналы сообщения, никаким из концов не связанные с `C`.

2) *Нормализация*. Во второй фазе происходит нормализация полученных проекций *MSC*. В обычной форме *MSC* состоит ровно из двух символов условия и (возможно, пустой) последовательностью сообщений между ними. Для нормализации *MSC*, которая не начинается или не заканчивается символом условия, добавим такое, обозначающее начальное состояние. Разделим *MSC*, которые содержат более чем два условия в любом из промежуточных состояний. Таким образом, из двух полученных *MSC* первое будет в силе до условия, по которому было произведено деление, включительно, а второе – сразу после этого условия в `C`.

3) *Трансформация в MSC-автомат*. Для получения такого автомата метки условий сопоставляются с состояниями автомата компоненты `C`. Переход из состояния s_0 в состояние s_1 создается в том и только том случае, если существует нормализованная *MSC* `M`, начальное и конечное

условия которой совпадают с метками s_0 и s_1 соответственно. В качестве начального условия для такого автомата берется то, которое было выбрано на входе в эту фазу. Результатом трансформации является автомат, являющийся высокоуровневой абстракцией состояний рассматриваемой компоненты.

4) *Трансформация в автомат*. На этой фазе переходы, отмеченные *MSC*, расширяются за счет промежуточных состояний и переходов. Все переходы, отмеченные *MSC*, содержащими только условия, ε-переходами. Пусть M будет какая-либо иная *MSC*-метка. Эта диаграмма описывает положительное число событий коммуникации, обозначаемых $(d_1, n_1, c_1), ..., (d_k, n_k, c_k)$, записанных в порядке соответствующих обозначений в *MSC* (стрелок). Здесь d_i, n_i, c_i обозначают направление соответствующих стрелок в M («!» для исходящих, «?» для входящих стрелок), метку сообщения и имя адресанта или адресата соответственно (для $1 \le i \le k$). При $k = 1$ переход с меткой M заменяется переходом с меткой (d_1, n_1, c_1). Для $k > 1$ к автомату добавляются k−1 новых состояний, обозначенных с s_1 до s_{k-1} и k новых переходов. Первый переход начинается в состоянии, соответствующем начальному условию M и заканчивается в состоянии s_1, а его меткой является тройка (d_1, n_1, c_1). В свою очередь, k-й переход с меткой (d_k, n_k, c_k), начинается в созданном состоянии s_{k-1} и заканчивается в состоянии, соответствующем финальному условию M. Все остальные переходы ведут из состояния s_{i-1} в состояние s_i и имеют метку (d_i, n_i, c_i), при $1 < i < k$. В конце рассмотренной фазы получается недетерминированный автомат с входными и выходными действиями.

5) *Оптимизация*. Для оптимизации полученного автомата используются стандартные алгоритмы из теории автоматов, такие как алгоритм исключения ε-переходов и т. д. Последующая оптимизация зависит от семантики системы. Например, в диаграмме состояний можно

заменить последовательные однонаправленные сообщения в одно. Входное сообщение и инициированные им выходные сообщения записываются через знак «/».

1.4. Модифицированные диаграммы последовательности сообщений

К сожалению, все разновидности языка *MSC*, включая стандарты *ITU* [23] и диаграммы последовательностей, адаптированные для языка *UML*, имеют весьма слабую выразительность. Их семантика немногим превосходит множество простых ограничений на частичный порядок возможных событий времени работы программы. Строго говоря, исходя из *MSC*, нельзя ничего сказать о том, что система будет фактически делать во время работы. Эти диаграммы описывают, что может произойти, а не то, что должно происходить. Таким образом, в большинстве возможных семантик языка *MSC* пустая система, не делающая ничего в ответ на любое внешнее событие, удовлетворяет такой диаграмме. Учитывая это, обычно добавляется минимальное, иногда скрытое требование, заключающееся в том, что хотя бы один запуск системы должен корректно пройти через каждую из специфицированных диаграмм последовательности сообщений [24].

Еще одна проблема заключается в том, что с помощью *MSC* невозможно описать нежелательный сценарий. Хотелось бы иметь возможность запретить критические для обеспечения безопасности последовательности действий.

В 1998 году была предложена идея модификации *MSC*, которая, в частности, добавила в них живучесть (или универсальность), позволяющая описывать не только возможное, но и необходимое, обязательное поведение системы. Более того, оба варианта поведения стало можно обозначать как на уровне всей диаграммы, так и локально, при указании отдельных событий, условий и временного прогресса внутри диаграммы. Таким образом, появился способ уйти от чисто экзистенциальной интерпретации диаграмм.

Живучесть позволила специфицировать и «запрещенные» последовательности действий, формализовала логические структуры в диаграммах: ветвление, итерирование и поддиаграммы [25]. Новый тип диаграмм последовательности сообщений получил название *LSC* (*Live Sequence Chart*). Рассмотрим его подробнее.

Итак, в диаграммах *LSC* существует возможность указать необходимость или только возможность поведения системы в отдельных частях диаграммы, причем эти метки имеют графическое представление. Авторы *LSC* предлагают различать это свойство во внутренней схеме элемента как «температуру» этого элемента: обязательные элементы – «горячие», а возможные – «холодные». В графическом отображении диаграмм введены перечисленные ниже новые обозначения.

В горизонтальном направлении:

- асинхронный и синхронный обмен сообщениями обозначается с помощью стрелок, имеющих закрашенные полностью или контурные концы;
- пунктирная стрелка обозначает обмен сообщениями, который может случиться (возможное поведение системы), а сплошная – тот, который должен случиться (обязательное поведение системы).

В вертикальном направлении:

- пунктирный участок линии обозначает возможный прогресс компоненты (ход программы может пойти далее по этой диаграмме), в то время как сплошной – обязательный прогресс (ход программы обязательно продолжится по этой диаграмме);
- с каждой компонентой ассоциируется набор положений, которые имеют температуру прогресса в компоненте в качестве метки. Например, как указано выше, возможный прогресс между положениями отмечается пунктирной линией.

Изменения коснулись и условий, поскольку в диаграммах *LSC* они играют очень важную роль. Условия также разделяются на «горячие» – обязательные, – и «холодные» – возможные. Первые помещаются в

контейнер из сплошных линий, вторые же – в контейнер из пунктирных линий. Если во время исполнения система обнаруживает невыполненное «горячее» условие, возникает ошибка и программа завершается некорректно. Если же обнаруживается невыполненное «холодное» условие, что является нормальной ситуацией, то система выходит из текущей поддиаграммы или из всей диаграммы, если текущей была диаграмма верхнего уровня (в этом случае программа корректно завершается).

Такая интерпретация условий оправдывает себя. Обязательные условия и «горячие» элементы позволяют специфицировать запрещенные сценарии развития событий – такие, которые система не должна совершать. Помимо этого, с помощью «холодных» условий становится возможной запись условного поведения и различных вариантов итераций [25].

Обобщая все вышесказанное, представим все введенные обозначения и скрывающуюся за ними семантику в табл. 1.

Таблица 1. Нововведенные обозначения

		Обязательное	**Возможное**
Диа-грамма	***Тип***	Универсальная	Экзистенциальная
	Семантика	Любой запуск системы соответствует диаграмме	Хотя бы один запуск системы соответствует диаграмме
Поло-жение	***Температура***	«Горячее»	«Холодное»
	Семантика	Прогресс в компоненте должен продвинуться за положение	Прогресс в компоненте необязательно продвинется за положение
Сооб-щение	***Температура***	«Горячее»	«Холодное»
	Семантика	Если сообщение послано, оно будет получено	Получение сообщения не гарантировано
Условие	***Температура***	«Горячее»	«Холодное»
	Семантика	Условие должно быть выполнено, иначе ошибка	Если условие не выполнено, выход из текущей (под)диаграммы

Выводы по главе 1

В данной работе будет исследоваться возможность построения автоматной модели по спецификации, заданной на рассмотренном выше языке *LSC*. Этот способ задания требований к проектируемой программе выбран из-за его гибкости, мощности, возможности простого и интуитивно понятного графического представления. Считается, что именно «визуальное программирование» является одним из наиболее понятных и наглядных методов проектирования систем [26]. В работе [27] показано, что *LSC* могут быть записаны с помощью выражений на языке темпоральной логики и наоборот, а это важное свойство означает, что язык *LSC* может иметь широкие перспективы применения при проектировании реальных систем, поскольку требования к таким системам нередко составляются именно на языке темпоральной логики. В ходе построения алгоритма синтеза также будут привлекаться результаты некоторых других рассмотренных выше исследований.

Глава 2. Построение синтезирующего алгоритма

2.1. Спецификация на языке *LSC*

Для начала определимся, что же есть спецификация на языке *LSC* и в каком случае можно считать, что система соответствует спецификации. Возьмем определение из работы [25]:

$$LS = \langle M, amsg, mode \rangle,$$

где `M` – множество диаграмм, `amsg` – активирующие сообщения, а `mode` – типы диаграмм (экзистенциальные или универсальные). В данной работе будут рассматриваться именно активизирующие сообщения, поскольку доказательства для такого случая не столь громоздки, как при использовании активизирующих диаграмм. Это, однако, не запрещает возможность подобного использования.

Система удовлетворяет *LSC*-спецификации, если при каждом запуске для каждой универсальной диаграммы верно, что в случае возникновения активирующего сообщения ход исполнения программы удовлетворяет этой диаграмме, и если для каждой экзистенциальной диаграммы верно, что существует такой ход программы, в котором возникает соответствующее активирующее сообщение, и далее ход программы удовлетворяет этой диаграмме. Запишем формальное определение:

Определение

Система S ***удовлетворяет*** *спецификации* LS, *если:*

$1. \forall m \in M, mode(m) = universal \Rightarrow \forall n L_S^n \subseteq L_m$;

$2. \forall m \in M, mode(m) = existential \Rightarrow \exists n L_S^n \cap L_m \neq \varnothing$.

В этих обозначениях L_S^n – множество вариантов исполнений системы `S` при последовательности направленных сообщений `n`, а L_m – язык диаграммы `m`, содержащий все возможные варианты исполнения системы, удовлетворяющие ему.

В свою очередь, *LSC*-спецификация называется удовлетворяемой, если существует система, удовлетворяющая ей.

Целью работы является автоматическое построение системы объектов, которая функционировала бы согласно данной спецификации. В процессе построения системы необходимо следить за тем, чтобы сама спецификация была корректной, непротиворечивой. В противном случае ни одна система не сможет удовлетворить такой спецификации.

Рассмотрим, к примеру, спецификацию, состоящую всего из двух универсальных диаграмм, приведенных на рис. 1. Сообщение `login`, присланное объекту `Core` окружением, активирует первую диаграмму, которая требует указанный дальнейший порядок сообщений. Однако после того как объект `SessionMgr` отвечает сообщением `updateOk`, активируется вторая диаграмма, которая специфицирует сообщение `startSession` до процедуры добавления сессии (`addSession`, `added`). С другой стороны, первая диаграмма требует сначала добавления сессии, и уже потом ее старта с помощью команды `startSession`. Получили противоречие. Такая спецификация некорректна.

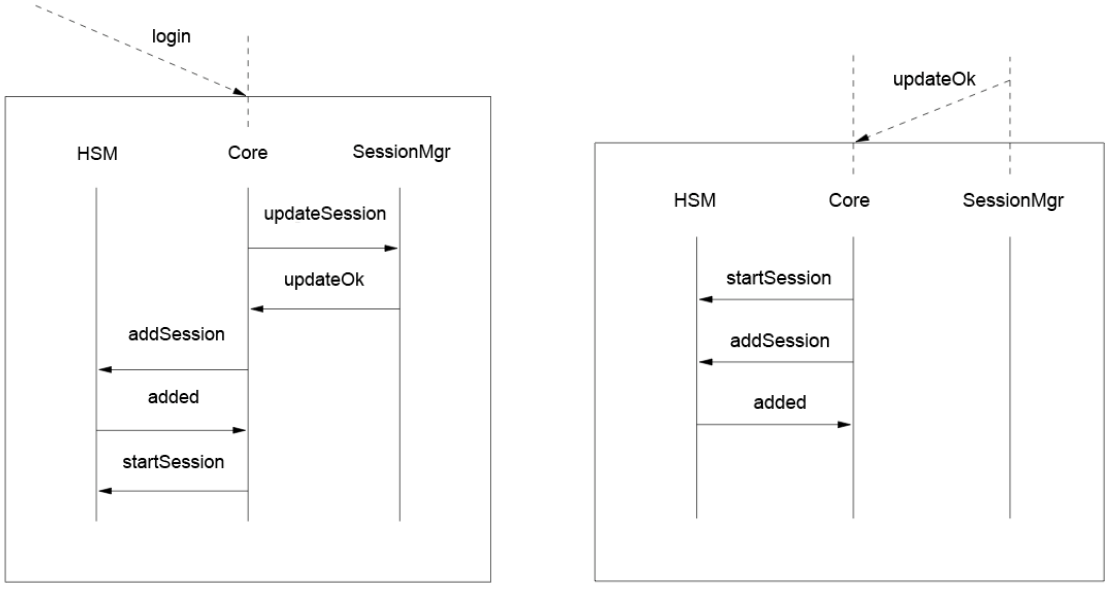

Рис. 1. Противоречивая спецификация

Для того чтобы установить корректность спецификации, необходимо, в частности, убедиться, что каждая из универсальных диаграмм

удовлетворяется любым запуском системы. В дальнейшем будем использовать следующие обозначения:

- A_{in} – алфавит сообщений, посылаемых системе от окружения;
- A_{out} – алфавит сообщений между объектами системы.

Определение

LSC-спецификация $LS = \{M, amsg, mode\}$ ***корректна****, если существует непустой регулярный язык* $L_1 \subseteq (A_{in} \cdot A_{out}^*)^*$*, удовлетворяющий следующим критериям:*

1. $L_1 \subseteq \bigcap_{m_j \in M, mode(m_j)=universal} L_{m_j}$;
2. $\forall w \in L_1 \forall a \in A_{in} \exists r \in A_{out}^* : w \cdot a \cdot r \in L_1$;
3. $\forall w \in L_1, w = x \cdot y \cdot z, y \in A_{in} \Rightarrow x \in L_1$;
4. $\forall m \in M, mode(m) = existential \Rightarrow L_m \cap L_1 \neq \varnothing$

Язык L_1 – это язык удовлетворяющих спецификацию запусков программы. Первое утверждение требует удовлетворения каждой из универсальных диаграмм любым запуском системы, второе – продолжения работы программы в случае возникновения нового сообщения от окружения, третье – завершения выполнения цепочки сообщений прежде, чем будут обработаны новые сообщения от окружающей среды, и, наконец, четвертое утверждение требует удовлетворения экзистенциальных диаграмм всеми запусками из языка L_1.

2.2. Корректность и существование удовлетворяющей системы

Перейдем к определению системы объектов. В работе [28] представлена Базовая Вычислительная Модель для Объектно-Ориентированного Дизайна, поведение которой задается автоматами. В

данной работе принимается эта модель с некоторыми упрощениями, облегчающими дальнейшие рассуждения. Рассмотрим их подробнее:

- предполагается, что каждый объект системы существует в единственном экземпляре в течение всего периода функционирования системы, не допускается создание или удаления объектов во время работы системы;
- все сообщения в системе синхронные, а также любое отправленное сообщение гарантированно доходит до получателя;
- при рассмотрении языков слов, генерируемых системой, возьмем такой порядок генерации запросов, чтобы он соответствовал тому порядку, в котором эти запросы были сделаны, в отличие от работы [28], где соответствие устанавливается с порядком выполнения запросов;
- рассматриваемая система генерирует язык над алфавитом $A = (O \cup env) \times (O.\sum)$, в котором символ $(O_j, O_i.\sigma)$ означает, что объект O_j выполняет запрос σ у объекта O_i; а символ $(env, O_i.\sigma)$ означает, что подобный запрос исходит от окружения (в работе [28] отправитель не является частью языка, поэтому используется алфавит $A = O.\sum$);
- не допускаются взаимные блокировки: если запрос направлен объекту, находящемуся в состоянии без соответствующих переходов, или объекту, находящемуся в процессе перехода, то этот запрос считается полученным, однако не влияет на указанный объект;
- допускаются безусловные переходы: после входа в новое состояние осуществляется дальнейший безусловный переход, если это возможно. Помимо этого, добавляется *условие честности*: если безусловный переход возможен бесконечное число раз, то и выполняется он бесконечное число раз. Учитывая это, в данной системе запрещены циклы из состояний, соединенных безусловными переходами, которые могут быть осуществлены без нарушения рассмотренного условия

честности. Таким образом достигается конечность времени реакции системы на события.

С перечисленными изменениями определения системы и ее поведения становится возможным установить необходимый и достаточный признак существования системы, удовлетворяющей спецификации на языке *LSC*. Более того, таким признаком является корректность самой спецификации. Прообраз данной теоремы взят из работы [25], в доказательстве были сделаны модификации для случая автоматной программы в качестве результата.

Теорема

Спецификация LS удовлетворяема тогда и только тогда, когда она корректна.

Доказательство

I. Пусть система S удовлетворяет спецификации LS. Покажем, что язык $L_S = \bigcup_{n \in A_{in}^*} L_S^n$ удовлетворяет четырем критериям языка L_1, упомянутым в определении корректности спецификации.

1) Из определения системы объектов следует, что L_S – регулярный и непустой. Система S удовлетворяет спецификации LS. Тогда, если положить $L = \bigcap_{m_j \in M, mode(m_j)=universal} L_{m_j}$, то по определению удовлетворения системой спецификации, верно $\forall n L_S^n \subseteq L$. Отсюда получаем: $L_S = \bigcup_n L_S^n \subseteq L$.

2) Пусть $w \in L_S$. Существует такая последовательность сообщений ($n = O^0.\sigma^0 \cdot O^1.\sigma^1 \cdot \ldots \cdot O^p.\sigma^p$), посланных окружением, что w представляет собой поведение системы S в ответ на эту последовательность. Тогда $w = w_0 \cdot w_1 \cdot \ldots \cdot w_p, w_i \in A^*, first(w_i) = (env, O^i.\sigma^i)$ и существует последовательность таких состояний $c_0, c_1, \ldots, c_{p+1}$, что c_0 является начальным состоянием и для $0 \le i \le p, leads(c_i, w_i, c_{i+1})$. Значение

предиката `leads` следующее: он описывает реакцию системы на сообщение от окружения, которое вызывает переход системы из состояния c_i в новое состояние c_{i+1}, возможно, проходя через «нестабильные» состояния (то есть через безусловные переходы). Путь такого выполнения – w_i.

Возвращаясь к пункту 2 определения корректности спецификации, заметим, что система попадает в состояние c_{p+1} в конце реакции на последовательность запросов от окружения n. Для любого объекта O_i и запроса σ существует реакция системы на запрос $O_i.\sigma$ из состояния c_{p+1}. Если обозначить путь этой реакции как w_{p+1}, то верно $w \cdot w_{p+1} \in L_S$.

3) Полагая $w = x \cdot y \cdot z, y \in A_{in}$, заметим, что $\exists i : x = w_0 \cdots w_i$, а следовательно, $x \in L_S$, что и требовалось.

4) Система `S` удовлетворяет спецификации LS. Из определения удовлетворения системой спецификации следует:

$$\forall m \in M, mode(m) = existential \Rightarrow \exists n L_S^n \cap L_m \neq \varnothing .$$

Поскольку $L_S^n \subseteq L_S = \bigcup_n L_S^n$, отсюда получаем, что:

$$\forall m \in M, mode(m) = existential \Rightarrow L_S \cap L_m \neq \varnothing .$$

Полученное утверждение соответствует пункту 4 определения корректности спецификации и завершает доказательство следствия корректности спецификации из существования системы, удовлетворяющей ей.

II. Пусть спецификация LS корректна. Необходимо доказать, что существует система `S`, удовлетворяющая ей. Для этого введем понятие *глобального автомата*, покажем, что существует таковой, удовлетворяющий спецификацию, и что с его помощью можно построить искомую систему `S`.

Глобальный автомат A, описывающий систему объектов $O = \{O_1, ..., O_p\}$ и множество сообщений $\Sigma = \Sigma_{in} \cup \Sigma_{out}$ – это тройка $A = \langle Q, q_0, \delta \rangle$, где:

- Q – конечный набор состояний;
- q_0 – начальное состояние;
- $\delta \subseteq Q \times B \times Q$ – отношение перехода, в котором B – набор меток вида σ / τ, где $\sigma \in A_{in} = (env) \times (O.\Sigma_{in})$, а $\tau \in A^*_{out} = ((O) \times (O.\Sigma_{out}))^*$.

Пусть теперь $n = a^0 \cdot a^1 \cdot \ldots$, где $a^i \in A_{in}$. Множество запусков автомата A при таких сообщениях от окружения – это язык $L^n_A \subseteq (A^* \cup A^w)$ – такой, что слово $w = w_0 \cdot w_1 \cdot \ldots$ входит в L^n_A тогда и только тогда, когда $w_i = a^i \cdot x^i$, где $x^i = x^{i_0} \cdot \ldots \cdot x^{i_{k_i - 1}} \in A^*_{out}$, и существует такая последовательность состояний $q^{0_1}, q^{1_1}, \ldots, q^{1_{k_1}}, q^{2_1}, \ldots q^{2_{k_2}}, \ldots; q^{0_1} = q_0$, что $\forall i, j \, (q^{i_j}, / x^{i-1_j}, q^{i_{j+1}}) \in \delta, \; (q^{ik_i}, a^i / x^{i_0}, q^{i+1_1}) \in \delta$.

Критерий удовлетворения глобальный автомат спецификации задается таким же образом, как и для системы объектов.

Определение.

Глобальный автомат A удовлетворяет спецификации $LS = \langle m, amsg, mode \rangle$, *если верно:*

- $\forall m \in M, mode(m) = unviersal \Rightarrow \forall n L^n_A \subseteq L_m$;
- $\forall m \in M, mode(m) = existential \Rightarrow \exists n L^n_A \cap L_m \neq \varnothing$.

Поскольку LS корректна, то, по определению, существует соответствующий язык L_1. Из его регулярности следует существование детерминированного конечного автомата $\Re = (A, S, s_0, F)$, разрешающего L_1. Предположим что $\Re$ минимален, поэтому все состояния из S достижимы и из каждого можно попасть в какое-либо допускающее состояние.

По пункту 2 определения корректности, для каждого допускающего состояния s из $\Re$ и для каждого $a \in A_{in}$ существует переход с меткой a, ведущий в состояние, из которого есть путь к допускающему состоянию, помеченный $r \in A^*_{out}$. Формально:

$$\forall s \in F \; \forall a \in A_{in} \;\; \rho(s, a) = s' \Rightarrow \exists r \in A^*_{out} : \rho(s', r) \in F \; .$$

Из пункта 3 того же определения следует, что ни одно недопускающее состояние в $\Re$ не имеет переходов с меткой $a \in A_{in}$. Это так, поскольку если бы существовало состояние $s \notin F$, достижимое из начального состояния по пути x, то $\rho(s_0, x) = s$ и $\rho(s, a) = s'$, причем из s' достижимо допускающее состояние: $\rho(s', z) \in F$. Тогда $w = x \cdot a \cdot z$ нарушает условие 3 корректности спецификации.

Таким образом, показано, что в $\Re$ переходы с меткой из A_{in} существуют только для допускающих состояний и что для допускающего состояния есть переход для каждого символа из алфавита A_{in}. Теперь необходимо сконструировать недетерминированный конечный автомат $\Re'$ с такими же свойствами, который бы запрещал для допускающих состояний переходы с метками из алфавита A_{out}. Это можно сделать, добавив каждому состоянию $s \in F$ дополнительное состояние $s' \notin F$ следующим образом: все входящие в s переходы, помеченные символом из A_{out}, копируются в s'. Тогда $\Re'$ разрешает тот же язык, что и $\Re$, поскольку за счет недетерминизма этот автомат может решить, осуществить ли переход в s или в s'.

Теперь построим из автомата $\Re'$ глобальный автомат B заменой всех переходов с метками из A_{out} на безусловные переходы с символом этой метки в качестве действия (выходного воздействия). Все переходы с метками из A_{in} остаются неизменными. Осталось показать, что построенный таким образом глобальный автомат B удовлетворяет спецификации LS.

По построению B, верно, что $L_B = \bigcup_n L_B^n = L_1$. Из корректности LS следует, что $L_1 \subseteq \bigcap_{m_j \in M, mode(m_j)=universal} L_{m_j}$. Отсюда:

$$\forall m \in M, mode(m) = universal \Rightarrow L_1 \subseteq L_m,$$

а из этого приходим к:

$$\forall m \in M, mode(m) = universal \Rightarrow \forall n L_B^n \subseteq L_1 \subseteq L_m,$$

что соответствует пункту 1 определения удовлетворения системой спецификации.

Из корректности LS также следует, что:

$$\forall m \in M, mode(m) = existential \Rightarrow L_m \cap L_1 \neq \varnothing .$$

Однако, поскольку $L_1 = \bigcup_n L_B^n$, то верное также и следующее:

$$\forall m \in M, mode(m) = existential \Rightarrow \exists n L_B^n \cap L_m \neq \varnothing$$

Последнее соответствует пункту 2 определения удовлетворения системой спецификации, что завершает доказательство следования существования системы, удовлетворяющей спецификации, из корректности этой спецификации.

Теорема доказана

Вернемся к понятию глобального автомата. Глобальный автомат – это конечный автомат с входным алфавитом, состоящим из сообщений от окружения системе (A_{in}), и выходным алфавитом, состоящим из сообщений между объектами системы (A_{out}). Глобальный автомат может иметь безусловные переходы, также выполняется *условие честности*, рассмотренное выше, и запрещены циклы безусловных переходов. Напомним, что последнее условие гарантирует конечность времени реакции системы на события.

Как показано в доказательстве, с помощью глобального автомата можно построить систему объектов, где каждому объекту сопоставлен конечный автомат, удовлетворяющую данной корректной спецификации. В дальнейшем будут предложены методы генерации автоматов для объектов из глобального автомата.

2.3. Алгоритм определения корректности спецификации

Из доказанной в предыдущем разделе теоремы следует, что достаточным условием существования удовлетворяющей спецификации LS системы есть корректность этой спецификации. В этом разделе приведен алгоритм определения корректности спецификации.

Базовой конструкцией, используемой в алгоритме, является конечный автомат, разрешающий язык универсальной диаграммы. Для универсальной диаграммы m определим детерминированный конечный автомат $\Re = (A, S, s_0, \rho, F)$, где:

- $A = A_{in} \cup A_{out}$ – алфавит;
- набор состояний S состоит из срезов m и из состояния s_0:

$$S = \{c \mid c \in cuts(m)\} \cup s_0;$$

- полагая отображение f $(dom(m) \cup env) \times \Sigma \times dom(m)$ в алфавит A естественным, определим функцию переходов ρ таким образом:
  - $\rho(s_0, a) = c_0$, если $a = f(amsg(m))$ и c_0 – начальный срез;
  - $\rho(c, a) = c$, если a не запрещено в диаграмме m;
  - $\rho(c, a) = c'$, если $succ_m(c, < j, l_j >, c'')$ и $succ_m(c'', < j', l_{j'} >, c')$, а $f(msg(m)(< j, l_j >)) = a$, где $< j, l_j >< j', l_{j'} >$ – события отправки и получения одного и того же сообщения, и если не все положения среза c' «холодные»;
  - $\rho(c, a) = s_0$, если $succ_m(c, < j, l_j >, c')$ и $f(msg(m)(< j, l_j >)) = a$ все положения среза c' «холодные»;
- $F = \{s_0\}$ – множество допускающих состояний.

Докажем теперь, что язык, разрешаемый построенным таким образом автоматом совпадает с языком L_m диаграммы m.

Теорема

$L(\Re) = L_m$.

Доказательство

I. Покажем, что $L(\Re) \subseteq L_m$. Положим $w = w_1 \cdot w_2 \cdot \ldots \cdot w_p \in L(\Re)$ и $w_i = f(amsg(m))$. Необходимо убедиться, что диаграмма *m* удовлетворяется, то есть, что $\exists i_1, i_2, \ldots, i_k \ \exists v = v_1 \cdot v_2 \cdot \ldots \cdot v_k \in L_m^{trc} : i < i_1 < i_2 < \ldots < i_k;\ \forall j : 1 \le j \le k \Rightarrow w_{i_j} = v_j$ и $\forall j' : i \le j' \le i_k; j' \notin \{i_1, \ldots, i_k\}; w_{j'} \notin f(Messages(m))$.

Пусть $q^0, q^1, \ldots, q^p$ – последовательность состояний, которые $\Re$ проходит, читая слово w. Очевидно, $q^{i-1} = s_0$, иначе $\Re$ не допустил бы слово из-за символа w_i, поскольку только из s_0 есть переход по условию $amsg(m)$. Пусть i_k – наименьший из индексов больше i, такой, что $q_{i_k} = s_0$. Такой индекс существует, поскольку из условия допустимости $\Re$ следует $q^p = s_0$ и предположения, что $w \in L(\Re)$.

Пусть $i_1, i_2, \ldots, i_k$ – упорядоченный список индексов в последовательности $q^i, \ldots, q^{i_k}$, в которой также $q^{i_j} \neq q^{i_{j-1}}$. Из определения функции перехода ρ следует, что срезы диаграммы, соответствующие $q^i, q^{i_1}, \ldots, q^{i_k}$, следуют друг за другом, q^i соответствует начальному срезу c_0, а q^{i_k} – срез, в котором все положения «холодные». Таким образом, последовательность срезов есть ход выполнения программы, генерирующий след $v = v_1 \cdot v_2 \cdot \ldots \cdot v_k \in L_m^{trc}$. Для всех индексов $j' : i \le j' \le i_k, j' \notin \{i_1, \ldots, i_k\}$ верно, что $q_{j'} = q_{j'-1}$. Отсюда, по определению функции перехода ρ вытекает, что $w_{j'} \notin f(Messages(m))$.

II. Осталось показать, что $L_m \subseteq L(\Re)$. Рассуждения аналогичны первой части, если предположить, что новое активирующее сообщение не посылается до тех пор, пока система обрабатывает предыдущее активирующее сообщение.

Теорема доказана

Автомат, допускающий только такие варианты исполнения системы, которые удовлетворяют *всем* универсальным диаграммам спецификации, может быть построен путем пересечения этих отдельных автоматов (полученных для каждой из универсальных диаграмм). Этот автомат будет использоваться в алгоритме определения корректности спецификации. Идея состоит в том, чтобы начать с этого автомата, который разрешает «наибольший» регулярный язык, удовлетворяющий все универсальным диаграммам, и постепенно сужать его, исключая состояния, из которых система могла бы под воздействием окружения (сообщений от него) нарушить спецификацию. После редуцирования необходимо убедиться, что все еще существуют варианты выполнения программы, удовлетворяющие каждой из экзистенциальных диаграмм.

Алгоритм

1. Найти минимальный детерминированный конечный автомат $\Re = (A, S, s_0, \rho, F)$, разрешающий язык $L = \bigcap_{m_j \in M, mode(m_j)=universal} L_{m_j}$.
2. Определим множества $Bad_i \subseteq S, i = 0,1,...$ следующим образом:
   - $Bad_0 = \{s \in S \mid \exists a \in A_{in} : \forall x \in A_{out}^* \ \rho(s, a \cdot x) \notin F\}$;
   - $Bad_i = \{s \in S \mid \exists a \in A_{in} : \forall x \in A_{out}^* \ \rho(s, a \cdot x) \notin F \setminus Bad_{i-1}\}$.

   Полученная последовательность множеств Bad_i монотонно возрастает, причем $Bad_i \subseteq Bad_{i+1}$, а поскольку S конечно, то она сходится. Обозначим ее предел Bad_{max}.
3. Построим из $\Re$ новый автомат $\Re' = (A, S, s_0, \rho, F')$, где множество допускающих состояний сужено до $F' = F - Bad_{max}$.
4. Продолжим редукцию $\Re$ путем удаления всех переходов с меткой A_{in}, ведущих в состояния из множества $S \setminus F'$. Получим новый автомат $\Re''$.

5. Проверим свойство $L(\mathfrak{R}'') \neq \varnothing$, а также $L_{m_i} \cap L(\mathfrak{R}'') \neq \varnothing$ для каждого $m_i \in M, mode(m_i) = existential$. Если оба свойства сохраняются, то результат алгоритма положителен, если же хотя бы одно нарушается, то результат отрицателен.

Теорема

Описанный алгоритм верный: при заданной спецификации LS он останавливается и выдает положительный результат тогда и только тогда, когда LS корректна.

Доказательство

I. Допустим, алгоритм выдал положительный результат. Покажем, что язык $L(\mathfrak{R}'')$ удовлетворяет всем свойствам языка L_1 из определения корректности спецификации.

1) По построению, автомат $\mathfrak{R}$ есть пересечение вариантов выполнения программы, удовлетворяющих универсальным диаграммам, поэтому $L(\mathfrak{R}) = \bigcap_{m_j \in M, mode(m_j)=universal} L_{m_j}$. При переходе от $\mathfrak{R}$ к $\mathfrak{R}''$ некоторые допускающие состояния были сделаны недопускающими, а некоторые переходы – удалены. Это не могло добавить новых слов к языку автомата, поэтому $L(\mathfrak{R}'') \subseteq L(\mathfrak{R})$.

2) Допустим, $w \in L(\mathfrak{R}'')$ и $s = \rho(s_0, w)$. Поскольку w допускается детерминированным автоматом $\mathfrak{R}''$, то $s \in F'$, а следовательно, $s \notin Bad_{max}$. В частности, $s \notin Bad_0 \Rightarrow \forall a \in A_{in} \exists x \in A_{out}^* : \rho(s, a \cdot x) \in F \setminus Bad_{max}$. Удаление переходов с метками из A_{in} не влияет на существование x, так как $x \in A_{out}^*$. Это означает, что $\forall w \in L(\mathfrak{R}'')\ \forall a \in A_{in}\ \exists x \in A_{out}^* : w \cdot a \cdot x \in L(\mathfrak{R}'')$.

3) Допустим, $w \in L(\mathfrak{R}''),\ w = x \cdot y \cdot z,\ y \in A_{in}$. Пусть $s = \rho(s_0, x)$. Тогда существует переход с меткой y из состояния s, иначе w не было бы допущено. В ходе построения $\mathfrak{R}''$ были удалены все переходы с

метками A_{in} из состояний множества $S \setminus F'$, поэтому $s \in F'$. Отсюда, x допускается языком $L(\Re'')$.

4) Последнее свойство следует напрямую, поскольку алгоритм выдает положительный результат если $\forall m_i \in M, mode(m_i) = existential$ верно $L_{m_i} \cap L(\Re'') \neq \varnothing$.

II. Покажем, что для любого языка L_1, удовлетворяющего свойствам определения корректности, верно $L_1 \subseteq L(\Re'')$. В этом случае будет также верно, что $\forall m_i \in M, mode(m_i) = existential$ $L_{m_i} \cap L_1 \neq \varnothing$, откуда вытекает $L_{m_i} \cap L(\Re'') \neq \varnothing$.

Пусть L_1 – любой язык, удовлетворяющий свойствам определения корректности. Тогда $L_1 \subseteq L(\Re)$. Покажем, что $\forall w : \rho(s_0, w) \in Bad_{max}$ выполняется $w \notin L_1$. Тогда справедливо и $L_1 \subseteq L(\Re')$, поскольку $\Re'$ получен из $\Re$ путем превращения состояний из множества Bad_{max} в недопускающие.

Пусть $s = \rho(s_0, w), s \in Bad_{max}$ и пусть $w \in L_1$. Учитывая то, что $Bad_{max} = Bad_i$, получаем: $\exists a \in A_{in} \forall x \in A_{out}^*$ $\rho(s, a \cdot x) \notin F - Bad_{i-1}$. Снова используя определение корректности и предположение $w \in L_1$, имеем: $\exists r \in A_{out}^* : w \cdot a \cdot r \in L_1$, откуда следует $\rho(s, a \cdot r) \in F$ и $L_1 \subseteq L(\Re)$.

Из того, что $\rho(s, a \cdot r) \in F$ и $\rho(s, a \cdot r) \notin F - Bad_{i-1}$, следует, что $\rho(s, a \cdot r) \in Bad_{i-1}$. Тогда слово $w^1 = w \cdot a \cdot r$ верно $w^1 \in L_1$, $\rho(s_0, w^1) \in Bad_{i-1}$. Таким же образом можно найти такие $w^2, w^3, ..., w^i$, что $\forall j$ $w^j \in L_1, \rho(s_0, w^j) \in Bad_{i-j}$. Получаем, что для w^i справедливо: $w^i \in L_1$, $\rho(s_0, w^i) \in Bad_0$. Пусть $s' = \rho(s_0, w^i), s' \in Bad_0$, тогда $\exists a \in A_{in} \forall x \in A_{out}^*$ $\rho(s', a \cdot x) \notin F$, но поскольку $w^i \in L_1$, то также верно, что $\exists r \in A_{out}^* : w^i \cdot a \cdot r \in L_1$. Учитывая тот факт, что $L_1 \subseteq L(\Re)$, получаем $\rho(s', a \cdot x) \in F$ и приходим к противоречию.

Было показано, что $L_1 \subseteq L(\Re')$, и осталось убедиться, что $L_1 \subseteq L(\Re'')$. Пусть $w \in L_1$ и $w \notin L(\Re'')$. Поскольку $L_1 \subseteq L(\Re')$, то $w \in L(\Re')$, следовательно,

$\rho(s_0, w) \in F'$. По предположению, в процессе перехода от $\mathfrak{R}'$ к $\mathfrak{R}''$ должен был быть удален один из переходов, который использовался в $\mathfrak{R}'$ во время чтения слова w, иначе автомат $\mathfrak{R}''$ допустил бы это слово. Пусть $w = x \cdot y \cdot z$, $y \in A_{in}$, $s = \rho(s_0, x)$ и пусть в автомате $\mathfrak{R}''$ был удален переход $\rho(s, a)$. Из определения корректности вытекает, что $x \cdot y \cdot z \in A_{in}$, а условие $y \in A_{in}$ подразумевает, что $x \in L_1 \Rightarrow x \in L(\mathfrak{R}')$, следовательно, $s = \rho(s_0, x) \in F'$, что противоречит предположению об удалении перехода $\rho(s, a)$, поскольку при построении $\mathfrak{R}''$ переходы удалялись только из множества состояний $S \setminus F'$.

Теорема доказана

2.4. Распределение глобального автомата

Рассмотрим процесс синтеза автоматной системы объектов по спецификации. Сначала необходимо применить описанный выше алгоритм для определения корректности спецификации, поскольку корректность является достаточным условием существования искомой системы. При положительном результате будет получен глобальный автомат, удовлетворяющий спецификацию. После этого необходимо неким образом получить из глобального автомата автоматы для каждого из объектов системы. Будем в дальнейшем называть этот процесс распределением глобального автомата. В данном разделе для наглядности в качестве примера будет использован фрагмент глобального автомата для системы, рассмотренной в главе 3.

Итак, пусть глобальный автомат $A = \langle Q, q_0, \delta \rangle$, описывающий систему объектов $O = \{O_1, ..., O_n\}$ и сообщения $\Sigma = \Sigma_{in} \cup \Sigma_{out}$, удовлетворяет спецификации LS. В ходе распределения будут использованы новые сообщения – из множества $\Sigma_{col} : \Sigma_{col} \cap \Sigma = \varnothing$. Эти сообщения служат для организации связи объектов и не запрещены диаграммами спецификации.

Далее будет предложено несколько методов распределения глобального автомата:

- использование объекта-координатора;
- полное дублирование;
- частичное дублирование.

Первый метод тривиальный и будет приведен только лишь с целью примера построения реальной автоматной системы, удовлетворяющей исходной корректной спецификации. Второй метод является предпосылкой на пути к третьему, более эвристическому.

Для наглядности, продемонстрируем предложенные методы на примере части глобального автомата системы, рассматриваемой в главе 3. Будем считать эту часть отдельным новым глобальным автоматом, – он приведен на рис. 2.

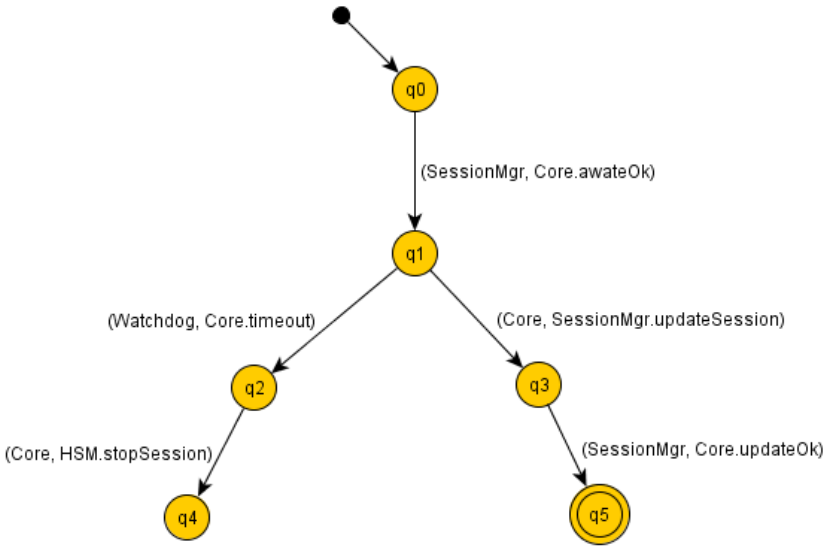

Рис. 2. Глобальный автомат, подлежащий распределению

2.4.1. Использование объекта-координатора

Добавим к множеству объектов $O = \{O_1, ..., O_n\}$ в системе новый объект O_{crd}, который будет координировать деятельность системы, рассылая команды остальным объектам. Последним же будут соответствовать простые автоматы для обработки этих команд.

Пусть $|\Sigma_{col}| = |A_{in}| + |A_{out}|$, а функция $f : A_{in} \cup A_{out} \to \Sigma_{col}$ будет соответствующей проекцией. Определим тогда автомат для

объекта-координатора O_{crd} как $\langle Q, q_0, \delta_{crd} \rangle$, а автоматы для остальных объектов $O_i \in O$ как $\langle \{q_{O_i}\}, q_{O_i}, \delta_{O_i} \rangle$. Состояния (в том числе и допускающее) для автомата объекта-координатора полностью идентичны состояниям глобального автомата. Функции перехода δ_{crd} и δ_{O_i} определяются следующим образом:

- если $(q, a, q') \in \delta$, где $a \in A_{in}$, $a = (env, O_i.\sigma_i)$, то $(q, f(a), q') \in \delta_{crd}$ и $(q_{O_i}, \sigma_{O_i} / O_{crd}.f(a), q_{O_i}) \in \delta_{O_i}$;
- если $(q, /a, q') \in \delta$, где $a \in A_{out}$, $a = (O_i, O_j.\sigma_j)$, то $(q, /O_i.f(a), q') \in \delta_{crd}$ и $(q_{O_i}, f(a) / O_j.\sigma_j, q_{O_i}) \in \delta_{O_i}$.

Автоматы этой конструкции, полученной из рассматриваемого глобального автомата, приведены на рис. 3–6. Размер автомата для объекта-координатора равен размеру глобального автомата, тогда как автоматы для объектов имеют по одному состоянию.

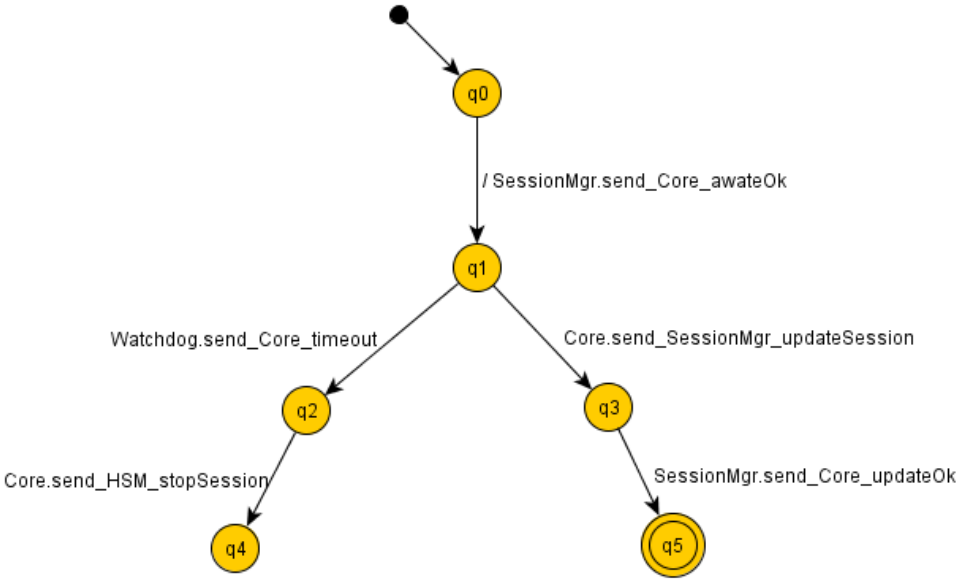

Рис. 3. Автомат для объекта-координатора

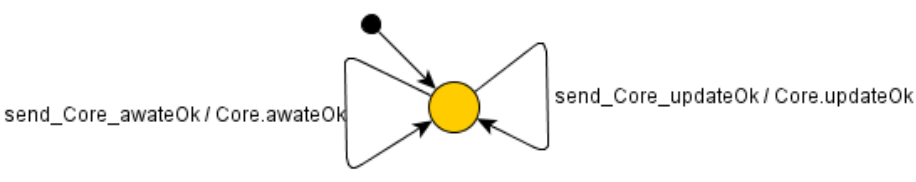

Рис. 4. Автомат для объекта `SessionMgr`

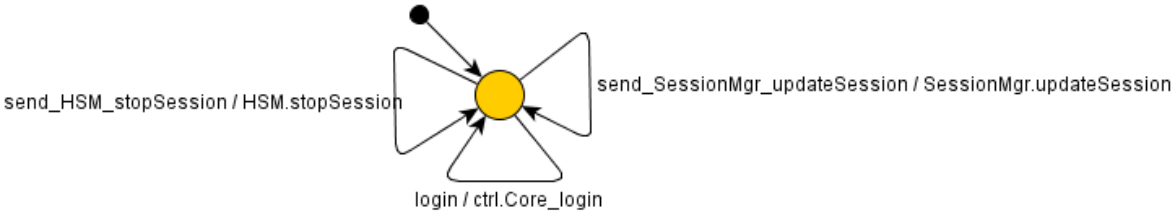

Рис. 5. Автомат для объекта `Core`

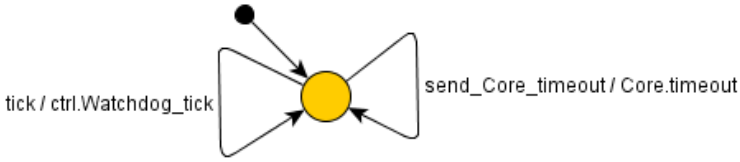

Рис. 6. Автомат для объекта `Watchdog`

2.4.2. Полное дублирование

В этом методе предлагается в автомате для каждого из объектов системы полностью дублировать распределяемый глобальный автомат. Вспомним, что глобальный автомат задан как $A = \langle Q, q_0, \delta \rangle$. Обозначим как k максимальную полустепень исхода состояний в множестве Q. Тогда множество меток переходов в A получено с помощью проекции $tn : \delta \to \{1,...,k\}$. Пусть $|\Sigma_{col}| = k$ и пусть функция $f : \{1,...,k\} \to \Sigma_{col}$ будет соответствующей проекцией.

Автомат для объекта O_i определим как $\langle Q, q_{O_i}, \delta_{O_i} \rangle$, где функция переходов задается следующим образом:

- если $(q, a, q') \in \delta$, где $a \in A_{in}$, $a = (env, O_i.\sigma_i)$, а $a' = f(tn(q, a, q')) \in \Sigma_{col}$, то $(q, \sigma_i / O_{i+1}, a') \in \delta_{O_i}$ и $\forall j \neq i \ (q, a' / O_{j+1}.a', q') \in \delta_{O_j}$;
- если $(q, /a, q') \in \delta$, где $a \in A_{out}$, $a = (O_i, O_j.\sigma_j)$, а $a' = f(tn(q, /a, q')) \in \Sigma_{col}$, то $(q, /O_i.\sigma_i; O_{i+1}.a', q') \in \delta_{O_i}$ и $\forall j \neq i \ (q, a' / O_{j+1}.a', q') \in \delta_{O_j}$.

Автоматы этой конструкции, полученной из рассматриваемого глобального автомата, приведены на рис. 7–9. Максимальная полустепень исхода в примере равна двум. Поэтому множество дополнительных сообщений $\Sigma_{col} = \{1, 2\}$.

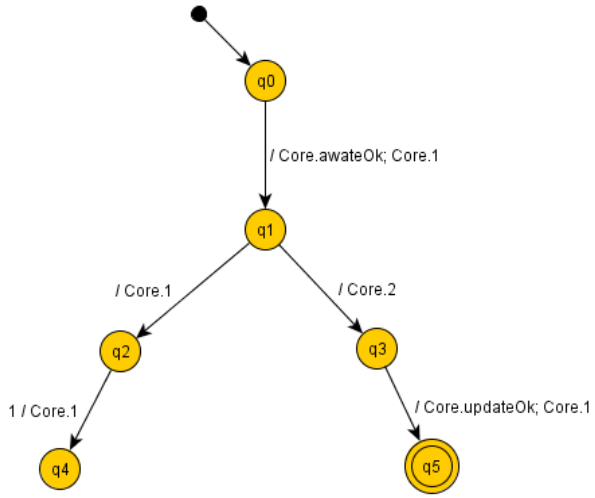

Рис. 7. Полное дублирование: автомат для объекта `SessionMgr`

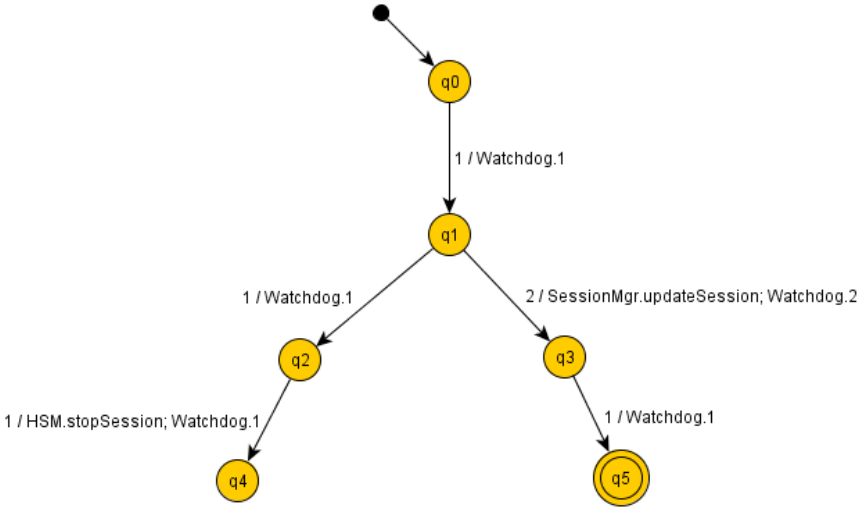

Рис. 8. Полное дублирование: автомат для объекта `Core`

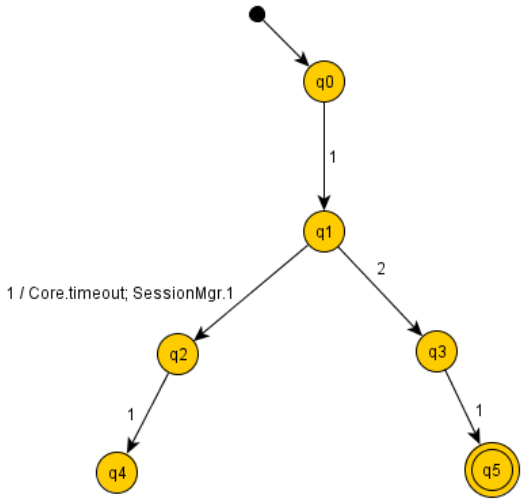

Рис. 9. Полное дублирование: автомат для объекта `Watchdog`

2.4.3. Частичное дублирование

Идея данного метода состоит в том, чтобы продублировать глобальный автомат, как и в предыдущем случае, а затем объединить те состояния в полученных автоматах для объектов системы, которые несут в себе информацию, не относящуюся к соответствующим объектам. В некоторых случаях это сильно уменьшит размер результирующих автоматов системы, хотя в общем случае их размер останется равным размеру распределяемого глобального автомата.

Автомат для объекта O_i определим как $\left\langle Q_{O_i} \cup q_{idle}, q_{O_i}, \delta_{O_i} \right\rangle$, где множество $Q_{O_i} \subseteq Q$ задается следующим образом:

$$Q_{O_i} = \left\{ q \in Q \left| \begin{bmatrix} \exists q' \in Q \;\; \exists a \notin A_{out} : a = (O_i, O_j.\sigma_j), \;\; (q, /a, q') \in \delta \\ \exists q' \in Q \;\; \exists a \notin A_{in} : a = (env, O_i.\sigma_i), \;\; (q', a, q) \in \delta \end{bmatrix} \right. \right\}$$

Таким образом, в автомате для объекта O_i остаются состояния, в которые глобальный автомат попадает при получении сообщения от окружения, и состояния, в которых O_i посылает сообщения.

Пусть $|\Sigma_{col}| = |Q|$ и пусть функция $f : Q \to \Sigma_{col}$ будет соответствующей проекцией. Функцию переходов δ_{O_i} определим таким образом:

- если $(q, a, q') \in \delta$, где $a = (env, O_i.\sigma_i)$, то $(q, \sigma_i / O_{i+1}.f(q'), q') \in \delta_{O_i}$;
- если $(q, /a, q') \in \delta$, где $a = (O_j, O_i.\sigma_i)$, то верно одно из двух:
  - $q' \in Q_{O_j}$, тогда $(q, /O_i.\sigma_i; O_{i+1}.f(q'), q') \in \delta_{O_i}$;
  - $q' \notin Q_{O_j}$, тогда $(q, /O_i.\sigma_i; O_{i+1}.f(q'), q_{idle}) \in \delta_{O_i}$;
- если $q \in Q_{O_j}$ и $q' \in Q_{O_j}$, то $(q, f(q'), q') \in \delta_{O_i}$;
- если $q \in Q_{O_j}$, а $q' \notin Q_{O_j}$, то $(q, f(q'), q_{idle}) \in \delta_{O_i}$.

Автоматы этой конструкции, полученной из рассматриваемого глобального автомата, приведены на рис. 10–12.

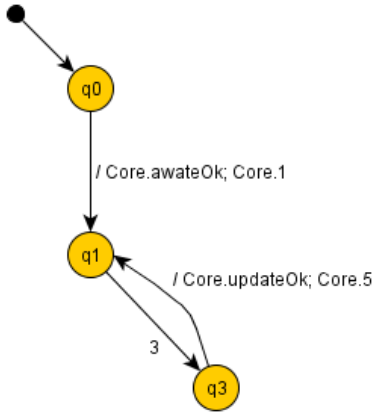

Рис. 10. Частичное дублирование: автомат для объекта `SessionMgr`

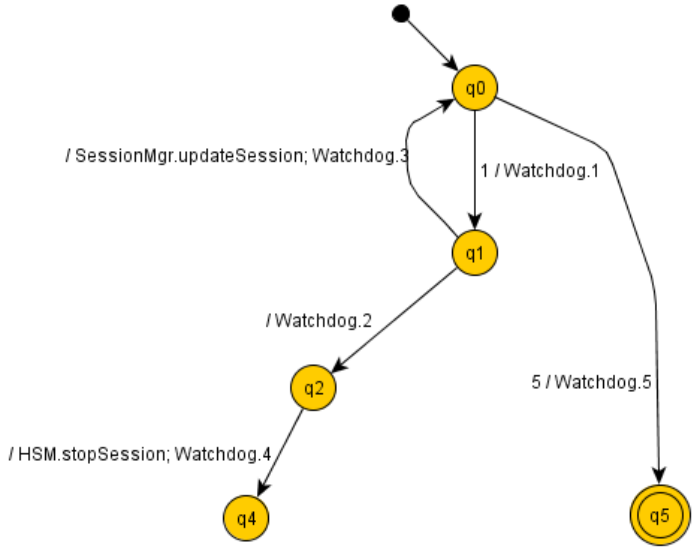

Рис. 11. Частичное дублирование: автомат для объекта `Core`

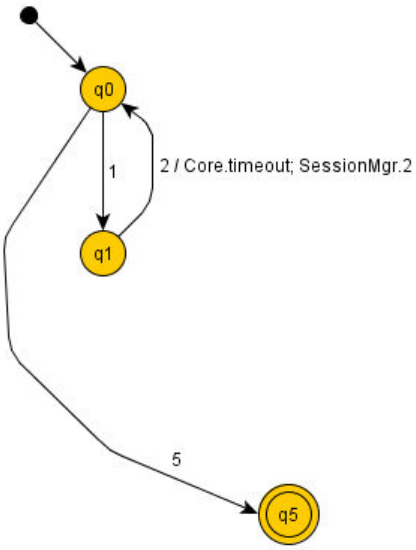

Рис. 12. Частичное дублирование: автомат для объекта `Watchdog`

2.5. Оценка сложности синтезированной системы

В этом разделе оценим размер синтезированной системы по отношению к спецификации LS. Для этого будем использовать обозначения и методы работы [25]: под размером *LSC*-диаграммы будем понимать число положений в ней – $|m| = |dom(m)|$, а под размером спецификации $LS = \langle M, amsg, mode \rangle$ – сумму размеров всех ее диаграмм: $|LS| = \sum_{m \in M} |m|$. Размером глобального автомата $A = \langle Q, q_0, \delta \rangle$ будем считать количество состояний в нем, то есть $|Q|$. Функция переходов δ – полиномиальная относительно числа состояний. Соответственно, в результирующей системе под размером автомата будем понимать число состояний в нем.

Пусть LS – корректная спецификация, в которой универсальные диаграммы обозначены как $\{m_1, m_2, ..., m_t\}$. Пусть A – соответствующий этой спецификации глобальный автомат, полученный с помощью описанного ранее алгоритма. Он был получен путем пересечения автоматов $A_1, A_2, ..., A_t$, допускающих выполнение программы по диаграммам $m_1, m_2, ..., m_t$, соответственно, и затем выполнением некоторых изменений, не влияющих на количество состояний.

Теорема

Число срезов диаграммы m, в которой участвуют k сущностей, ограничена числом $|m|^k$.

Доказательство

Срез диаграммы есть множество положений всех участвующих сущностей. Поскольку в срезе присутствует ровно одно положение для каждого из объектов, а число положений для объекта i равно $|dom(m,i)|$, то всего может существовать, как максимум, $\prod_{i \in inst(m)} |dom(m,i)|$. В реальности

это число обычно меньше, поскольку порядок сообщений запрещает множество срезов.

Теорема доказана

Теорема

Размер глобального автомата A удовлетворяет неравенству $|A| \le \prod_{i=1}^{t} |m_i|^{k_i} \le |LS|^{kt}$, *где* k_i *– число объектов в диаграмме* m_i, k *– общее число сущностей в спецификации, а* t *– число универсальных диаграмм в спецификации.*

Доказательство

Поскольку глобальный автомат получен из пересечения автоматов, справедливо:

$$|A| = \prod_{i=1}^{t} |A_i| \le \prod_{i=1}^{t} \prod_{j \in inst(m_i)} |dom(m_i, j)|.$$

Тогда искомое неравенство следует напрямую из предыдущей теоремы. Заметим, что построенные автоматы, соответствующие универсальным диаграммам, являются детерминированными, следовательно, и пересечение их детерминировано, и поэтому, нет необходимости применять процедуру детерминизации к полученному глобальному автомату, которая могла бы экспоненциально увеличить количество состояний в автоматах синтезированной системы.

Теорема доказана

Исходя из вышесказанного, размер глобального автомата является полиномом относительно размера спецификации, если считать количество диаграмм в спецификации и количество объектов в системе постоянными. Временная сложность алгоритма синтеза является, в свою очередь, полиномом относительно размера глобального автомата.

Наконец, оценим размер системы в предложенных выше методах:

- в случае использования объекта-координатора, размер автомата, соответствующего координатору, равен размеру глобального автомата, а автоматы для объектов имеют в качестве размера константу: они включают в себя лишь одно состояние;
- в случае полного дублирования размер каждого из автоматов, соответствующего объекту системы, равен размеру глобального автомата;
- в случае частичного дублирования размер автомата, соответствующего объекту, может быть меньше размера глобального автомата, однако суммарное количество состояний в автоматах системы не может быть меньше размера глобального автомата.

Выводы по главе 2

В этой главе был предложен и обоснован метод синтеза автоматной программы для системы объектов по спецификации на языке модифицированных последовательностей диаграмм *LSC*, а также приведена оценка размера результирующей системы.

Предложенный алгоритм заключается в построении промежуточных автоматов, соответствующих диаграммам спецификации, по которым затем строится так называемый глобальный автомат, и распределении этого автомата на автоматы для каждого из объектов системы.

Стоит заметить, что изначально наложенные на свойства системы ограничения (число объектов постоянно, сообщения синхронны и т. д.) не влияют на возможную сложность и жизнеспособность системы – система не становится тривиальной. Таким образом, приведенный алгоритм имеет большую перспективу применения в проектировании и синтезе реальных систем.

Глава 3. Применение алгоритма на примере синтеза программы банковского сервера аутентификации

3.1. Спецификация системы

В данной главе будет рассмотрен пример применения предложенного ранее алгоритма синтеза программ по спецификации на примере управляющей программы банковского сервера аутентификации. Речь идет о компоненте, которая в том или ином виде должна присутствовать в любой ситуации, когда клиенту предоставляется возможность удаленно каким-либо образом управлять своим банковским счетом: совершать переводы средств или оплачивать квитанции и т. д. – в любой ситуации, когда требуется авторизация клиента.

Сервер аутентификации есть ни что иное, как своего рода «прослойка» между клиентом и бизнес-приложениями, которые, в свою очередь, имеют связь с, собственно, базой клиентов банка. В последней как раз и хранятся все данные о доступных средствах на счетах каждого из клиентов, история транзакций и т. п. В дальнейшем базу будем обозначать как `Base`. Было бы нецелесообразно возлагать на `Base` излишнюю функциональность, как, например, аутентификация и авторизация пользователей, и уж тем более возможность изменения данных напрямую самим клиентом. Это бы сделало программную основу компоненты Base слишком большой и неудобной для отладки, поддержки и модификации. В реальности для изменения данных в Base пользователю предоставляются бизнес-приложения. Примером такого приложения является Интернет-банкинг, позволяющий производить операции со счетами через Интернет, или Мобильный банкинг, отличающийся от первого тем, что устройством, с которого эти изменения производятся, является мобильный телефон. Можно привести еще много примеров бизнес-приложений. Однако любому из них необходимо проверить, что за пользователь пытается их

использовать, имеет ли он на это право (банк может предоставлять какие-то услуги только определенному кругу своих клиентов). Было бы логично унифицировать метод аутентификации клиента для того, чтобы не поддерживать соответствующие программные модули в каждом из бизнес-приложений. Поэтому решение вынести модуль аутентификации в отдельную сущность является эффективным средством унификации и обеспечения единообразности выполнения операций в разных приложениях. Как результат, поддержка такого решения осуществляется гораздо проще, а значит, и обходится намного дешевле.

Сам сервер аутентификации будем обозначать как `AS`. Это сложный объект, состоящий из ядра (`Core`), менеджера сессий аутентификации (`SessionMgr`), криптографического устройства (`HSM`) и сущности, следящей за временем и текущими сессиями (`Watchdog`).

Строго говоря, процедура аутентификации может быть самой различной. В данном случае главным устройством для обеспечения безопасности аутентификации является криптографическое устройство `HSM` и его «следящий» поток `Watchdog`, рассматриваемый в данной системе как отдельный объект. `HSM` представляет собой физически отдельное от всех систем устройство, имеющее некоторый программный интерфейс, и способное генерировать ключи для шифрования сообщений к клиенту и от него. `HSM` оперирует понятием сессии аутентификации, в контексте которой происходит общение с одним клиентом. Каждая сессия имеет свой криптоключ. Периодически `HSM` должен менять криптоключ в целях защиты от взлома этого ключа в реальном времени. За временем следит объект `Watchdog`, являющийся, по сути, таймером для каждой из сессий. Для того чтобы не слишком усложнять пример и не вдаваться в подробности процедуры аутентификации, будем рассматривать только абстрактные сессии аутентификации и объекты, которые тем или иным образом связаны с процессами управления этими сессиями.

Рассмотрим схему работы `AS` и построим спецификацию на языке модифицированных последовательностей диаграмм *LSC*:

1. Возникающий запрос `login` обрабатывается компонентой `Core`, которая создает новую сессию в `SessionMgr` с помощью запроса `updateSession`, дожидается ответа `updateOk`, затем регистрирует эту сессию в `HSM` с помощью запроса `addSession`, на который он отвечает added, и, наконец, стартует криптосчетчик сессии с помощью запроса `startSession`. Все общение с клиентом в дальнейшем происходит в менеджере конкретной сессии (например, поток в `Core`), который не рассматривается в данной работе. Соответствующая *LSC*-диаграмма приведена на рис. 13.

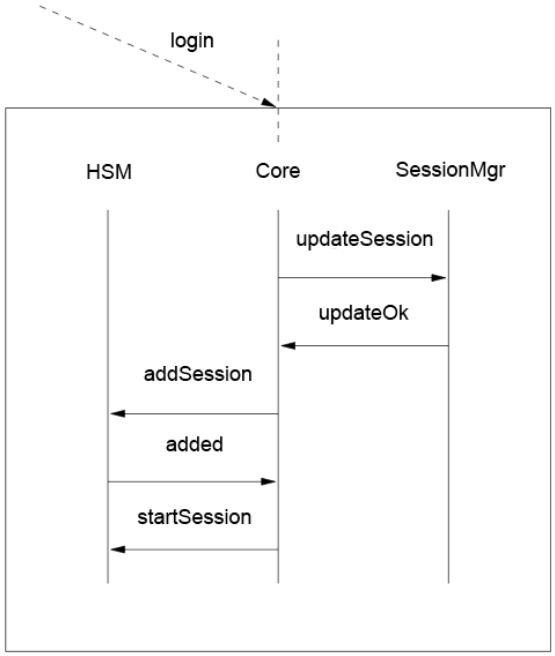

Рис. 13. Поведение объектов при получении сигнала `login`

2. Как было упомянуто, `Watchdog` – сущность, следящая за временем и текущими сессиями. При возникновении сигнала `tick` от системного окружения (от системных часов), `Watchdog` посылает `Core` сигнал `changeSID` о предстоящей смене криптоключа. Соответствующая диаграмма приведена на рис. 14.

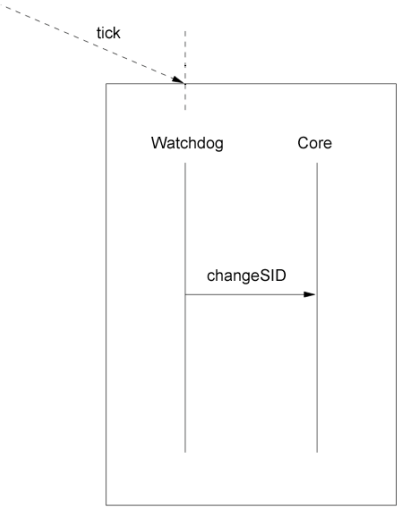

Рис. 14. Поведение объектов при получении сигнала `tick`

3. После получения сигнала `changeSID` о предстоящей смене криптоключа `Core` посылает сигнал `awateSession` объекту `SessionMgr`, подготавливая тем самым сессию к смене ключа или удалению, и дожидается подтверждающего ответа `awateOk`. Соответствующая диаграмма приведена на рис. 15.

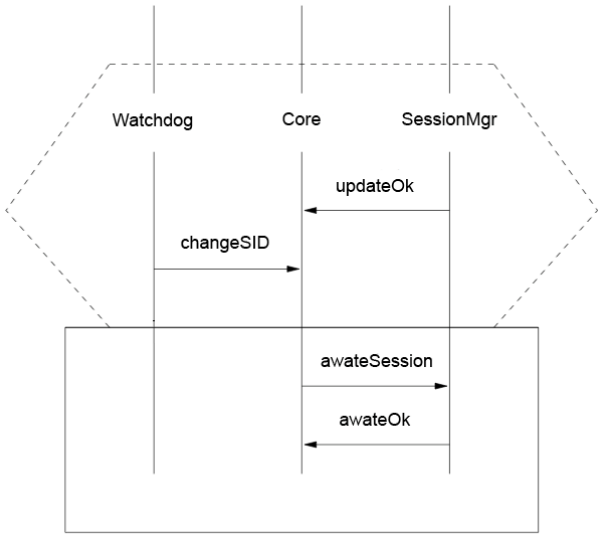

Рис. 15. Поведение объектов при получении сигналов `updateOk` и `changeSID`

4. Может произойти ситуация, когда после обмена сигналами `awate` от `Watchdog` придет сигнал `timeout`, сообщающий о завершении сессии. По логике работы `AS` эта ситуация имеет место в случае прекращения активности в сессии аутентификации, и такая сессия должна быть завершена и удалена. В этом случае `Core` посылает

объекту `HSM` сигналы `stopSession` и `removeSession`. Соответствующая диаграмма приведена на рис. 16.

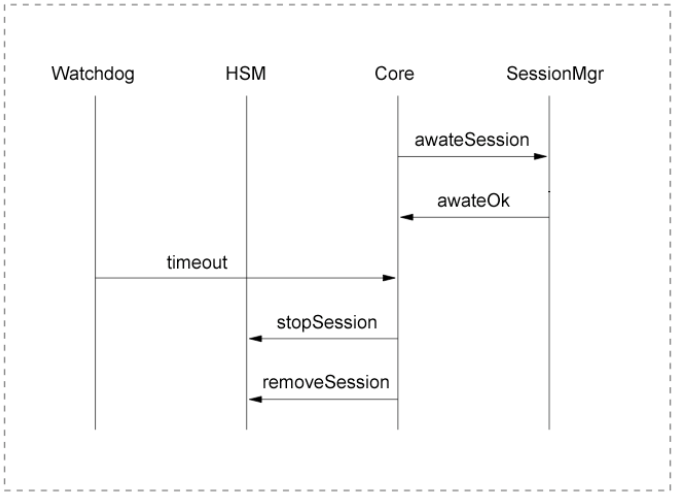

Рис.16. Возможное поведение объектов (экзистенциальная диаграмма).

5. Может произойти и другая ситуация − в том случае если в сессии все еще есть активность. Тогда сессию необходимо «обновить» с новым криптоключом, что и делает `Core`, посылая объекту `SessionMgr` сообщение `updateSession` и дожидаясь подтверждения `updateOk`. Соответствующая диаграмма приведена на рис. 17.

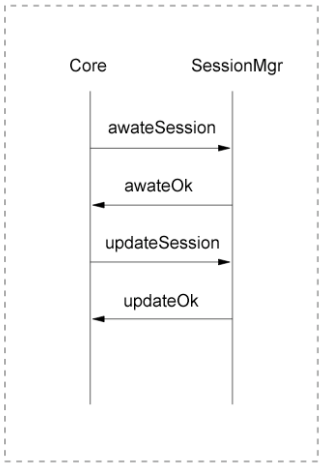

Рис. 17. Возможное поведение объектов (экзистенциальная диаграмма)

Приведенные выше пять диаграмм и будем считать спецификацией банковского сервера аутентификации. Применим предложенный алгоритм для синтеза автоматной программы по этой спецификации.

3.2. Синтез автоматной программы для системы

Итак, приступим к синтезу автоматной программы, используя предложенный в данной работе метод.

Определим множество запрещенных сообщений для каждой из универсальных диаграмм. В нашем случае это три первые диаграммы. Для них, соответственно, получаем такие множества:

- $\{timeout,\ changeSID,\ awateSession,\ awateOk,\ stopSession,\ removeSession\}$;
- $\{updateOk\}$;
- $\{updateSession,\ addSession,\ added,\ startSession\}$.

Построенные для каждой из универсальных диаграмм автоматы приведены на рис. 18–20. Заметим, что в автомате на рис. 19 два допускающих состояния, поскольку в активирующей диаграмме два сообщения: `updateOk` и `changeSID`. Также стоит отметить, что каждое состояние имеет переходы, приводящие в него же. Для недопускающих состояний это переходы по незапрещенным сообщениям между объектами системы (*NRM*), а для допускающих – все сообщения, не приводящие к смене состояния, включая сообщения от окружения (*NTM*).

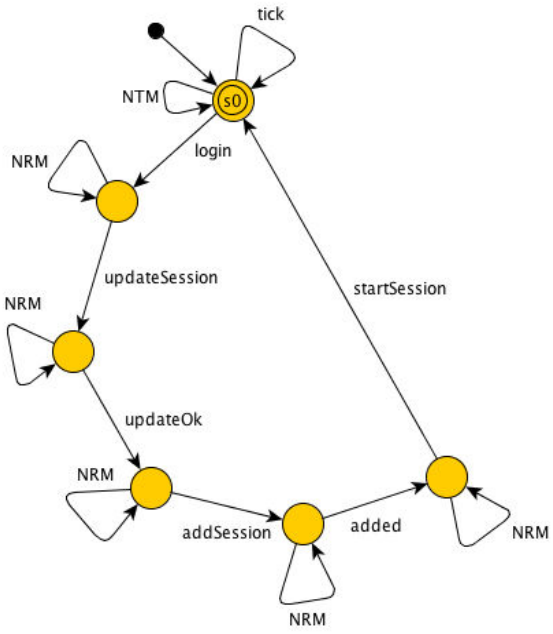

Рис. 18. Автомат для ситуации получения сигнала `login`

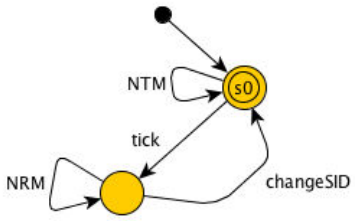

Рис. 19. Автомат для ситуации получения сигнала `tick`

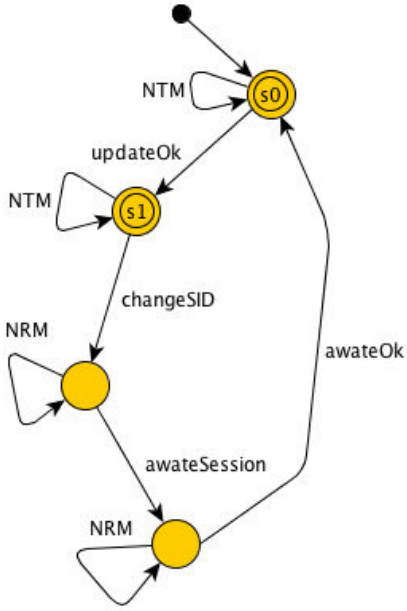

Рис. 20. Автомат для ситуации получения сигнала `updateOk` и `changeSID`

Далее пересечем полученные автоматы, получая новый автомат, допускающий все варианты исполнения программы, удовлетворяющие универсальным диаграммам спецификации. Автомат, полученный пересечением, приведен на рис. 21.

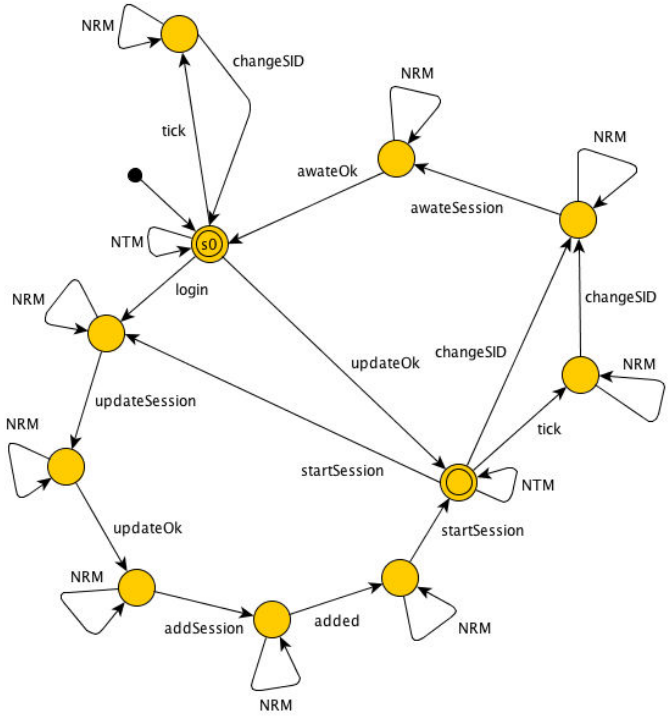

Рис. 21. Пересечение автоматов, соответствующих универсальным диаграммам

Затем с помощью механизма редукции отсечем состояния, из которых система может нарушить спецификацию, и получим глобальный автомат. По построению, два допускающих состояния глобального автомата имеют переходы только для сообщений от окружения. Заметим, что в глобальном автомате существуют варианты исполнения программ, удовлетворяющие каждой из универсальных диаграмм. Полученный глобальный автомат представлен на рис. 22.

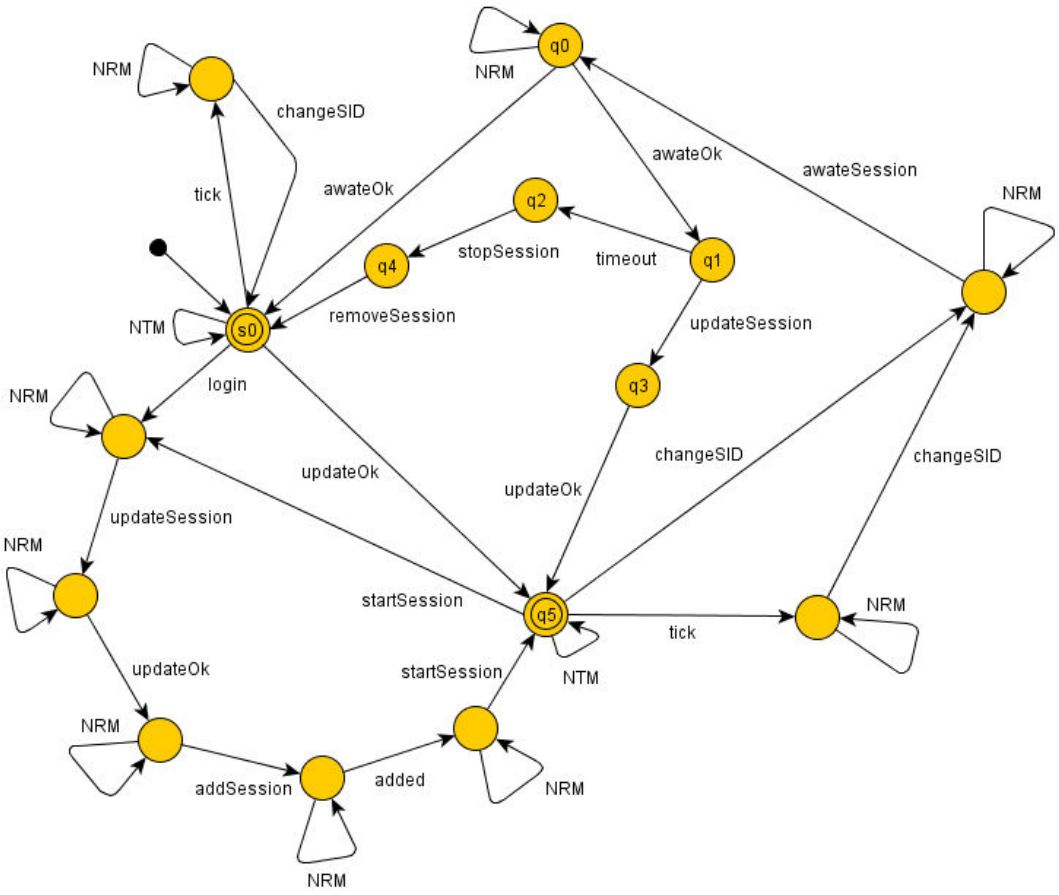

Рис. 22. Глобальный автомат

Теперь необходимо распределить глобальный автомат на автоматы для каждого из объектов системы. В главе 2 были рассмотрены три метода распределения на примере части данного глобального автомата, состоящей из множества состояний $q_0, q_1, ..., q_5$. Распределение полного глобального автомата является чисто технической задачей, поэтому приведем ее результат, представленный на рис. 23–26.

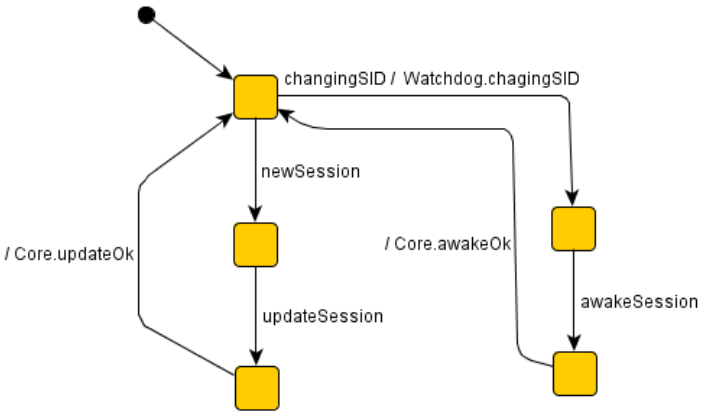

Рис. 23. Автомат для объекта `SessionMgr`

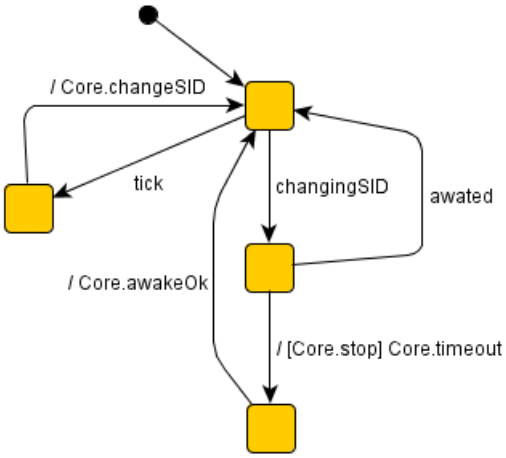

Рис. 24. Автомат для объекта `Watchdog`

Рис. 25. Автомат для объекта `HSM`

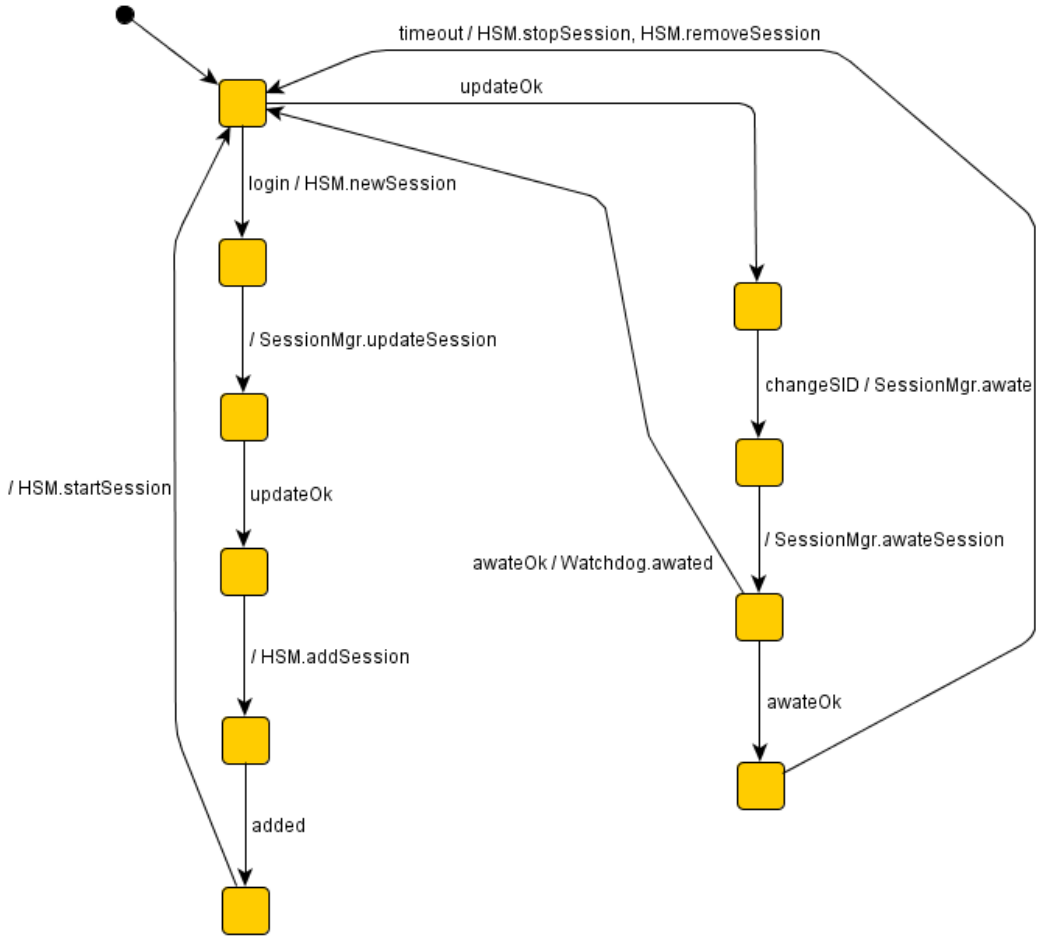

Рис. 26. Автомат для объекта `Core`

Выводы по главе 3

Приведен пример спецификации реальной системы и последующего синтеза, использующего предложенный в данной работе алгоритм. Алгоритм может стать актуальным при проектировании и реализации программных систем в различных областях, и этот пример является тому подтверждением.

Заключение

В данной работе получила развитие идея принципиально нового подхода к программированию, направленного на кардинальное повышение качества программ. Этот подход заключается в автоматическом построении программы, использующей автоматный подход, по заданной спецификации, то есть по предъявляемым требованиям.

Предложенный алгоритм не только позволил синтезировать программу, соответствующую заданным требованиям, но и попутно проверяет семантическую корректность этих требований. Для задания спецификации был выбран наглядный и в то же время достаточно мощный язык модифицированных последовательностей диаграмм. На этапе проектирования и разработки этот язык более понятен широкому числу даже неподготовленных людей, чем, например, формулы на языке темпоральной логики. В связи с этим, данный подход является весьма актуальным во многих сферах, где применяются программные системы.

Вдобавок к теоретической части, в работе рассмотрен синтез автоматной модели банковского сервера аутентификации, как пример применения предложенного алгоритма к прототипу реальной задачи. Этот пример еще раз подтверждает актуальность и применимость предложенного подхода.

Список литературы

1. *Ariane 5 Flight 501* / Wikipedia. http://en.wikipedia.org/wiki/Ariane_5_Flight_501.
2. *Therac-25*, / Wikipedia. http://en.wikipedia.org/wiki/Therac-25.
3. *Specification*, / Wikipedia. http://en.wikipedia.org/wiki/Specification.
4. *Шалыто А. А., Туккель Н. И.* SWITCH-технология – автоматный подход к созданию программного обеспечения «реактивных» систем // Программирование. 2001. № 5, с. 45–62. http://is.ifmo.ru/works/switch/1/
5. *Егоров К. В., Шалыто А. А.* Методика верификации автоматных программ // Информационно-управляющие системы. 2008. № 5, с. 15–21. http://is.ifmo.ru/works/_egorov.pdf
6. *Курбацкий Е. А., Шалыто А. А.* Верификация программ, построенных на основе программного подхода / Материалы XV Международной научно-методической конференции «Высокие интеллектуальные технологии и инновации в образовании и науке». СПбГ ПУ. 2008, с. 293–296. http://is.ifmo.ru/download/2008-02-25_politech_verification_kurb.pdf
7. *Вельдер С. Э., Шалыто А. А.* Введение в верификацию автоматных программ на основе метода *Model checking*. СПбГУ ИТМО. 2006. http://is.ifmo.ru/download/modelchecking.pdf
8. *Manna Z., Waldinger R.J.* A deductive approach to program synthesis / ACM Trans. Prog. Lang. Sys. 2. 1980, pp. 90–121.
9. *Emerson E. A., Clarke E. M.* Using branching time temporal logic to synthesize synchronization skeletons // Sci. Comp. Prog. 2.1982, pp. 241–266.
10. *Pnueli A., Rosner R.* On the Synthesis of a Reactive Module / Proc. 16th ACM Symp. on Principles of Programming Languages, Austin, TX, 1989.
11. *Wong-Toi H., Dill D.L.* Synthesizing processes and schedulers from temporal specifications / Computer-Aided Verification'90, DI-MACS Series in Discrete Mathematics and Theoretical Computer Science. 1991, vol. 3, pp. 177–186.
12. *Kupferman O., Vardi M.Y.* Synthesis with incomplete information / Proc. 2nd Int. Conf. on Temporal Logic (ICTL'97). 1997, pp. 91–106.
13. *Pnueli A., Rosner R.* Distributed Reactive systems are Hard to Synthesize / Proc. 31st IEEE Symp. on Foundations of Computer Science. 1990, pp. 746-757.
14. *Leue S., Mehrmann L., Rezai M.* Synthesizing ROOM Models from Message Sequence Chart Specifications / University of Waterloo Tech. Report 98–06, 1998.
15. *Selic B., Gullekson G., Ward P.* Real-Time Object-Oriented Modeling. John Wiley & Sons, New York, 1994.
16. *Koskimies K., Makinen E.* Automatic Synthesis of State Machines from Trace Diagrams // Software Practice and Experience. 1994, vol. 24, pp. 643–658.
17. *Koskimies K., Systa T., Tuomi J., Mannisto T.* Automated Support for Modeling OO Software // IEEE Software, 1998, vol. 15, pp. 87–94.

18. *Biermann A.W., Krishnaswamy R.* Constructing Programs from Example Computations // IEEE Trans. Softw. Eng., SE-2. 1976, pp. 141–153.
19. *Amon T., Boriello G., Sequin C.* Operation/Event Graphs: A design representation for timing behavior / Proc. '91 Conf. on Computer Hardware Description Language. 1991, pp. 241–260.
20. *Schlor R., Damm W.* Specification and verification of system-level hardware designs using timing diagrams / Proc. European Conference on Design Automation, Paris, France, IEEE Computer Society Press. 1993, pp. 518-524.
21. *Krüger I., Grosu R., Scholz P., Broy M.* From MSCs to Statecharts / Proc. DIPES'98, Kluwer, 1999.
22. *Сайт проекта UniMod*. http://unimod.sf.net
23. *MSCs*: ITU-T Recommendation Z.120: Message Sequence Chart (MSC). ITU-T, Geneva, 1996.
24. *Harel D.* From Play-In Scenarios To Code: An Achievable Dream // IEEE Computer. 2001, vol. 34(1), pp. 53–60.
25. *Damm, W., Harel, D.* LSCs: Breathing Life into Message Sequence Charts /Formal Methods in System Design. Kluwer Academic Publishers, vol. 19. 2001, pp. 293–312.
26. *Johnston W.M.; Hanna J.R.P., Millar R.J.* Advances in dataflow programming languages // ACM Computing Surveys (CSUR) 36. 2004, vol. 1, pp. 1–34.
27. *Kugler H., Harel D., Pnueli A., Lu Y., Bontemps Y.* Temporal Logic for Scenario-Based Specifications / In 11th Int. Conf. TACAS'05. Springer-Verlag, 2005.
28. *Harel D., Kupferman O.* On the Inheritance of State-Based Object Behavior // Proc. 34th Int. Conf. on Component and Object Technology, IEEE Computer Society. 2000, vol. 1, pp. 45–62.