Санкт-Петербургский государственный университет информационных технологий, механики и оптики

На правах рукописи

Мазин Максим Александрович

Автоматное программирование для среды языково-ориентированного программирования

Специальность 05.13.11 – «Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей»

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель –
доктор технических наук,
профессор А. А. Шалыто

Санкт-Петербург
2010

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Актуальность проблемы. С момента рождения программирования одной из основных проблем является написания программных систем со сложным поведением. В последнее время для создания таких систем все чаще используется автоматное программирование. Существует несколько подходов адаптации автоматного программирования к различным техникам и парадигмам программирования, среди которых доминирующей является объектно-ориентрованное программирование. Однако эта парадигма существует в практически неизменном виде уже более тридцати лет и требует дальнейшего развития, что может быть выполнено на основе языково-ориентированного программирования. Это должно повысить уровень абстракции программного кода, упростить его реализацию и сопровождение. Поэтому адаптация автоматного программирования применительно к языково-ориентированной парадигме является актуальной.

Цель диссертационной работы – адаптация автоматного программирования для среды языково-ориентированного программирования.

Основные задачи исследования. Для достижения указанной цели диссертации решены следующие задачи, решения которых выносятся на защиту:

1. Разработка текстового языка автоматного программирования (абстрактный и конкретный синтаксисы, операционная семантика, системы типов, кодогенератор и т. д.).
2. Разработка средств валидации автоматов в среде языково-ориентированного программирования.
3. Создание языка для многопоточного автоматного программирования, основанного на акторах.

4. Внедрение результатов работы в практику программирования в среде языково-ориентированного программирования *MPS* (*Metaprogramming System*).

Научная новизна. Научная новизна предлагаемых подходов состоит в том, что автоматное программирование адаптировано для языко-ориентированного программирования в части обеспечения языковой поддержки, валидации и многопоточности.

Методы исследования. В работе использованы методы объектно-ориентированного проектирования, метапрограммирования, теории автоматов, теории формальных грамматик, теории графов, теории алгоритмов, исчисление секвенций, теории акторных моделей.

Достоверность научных положений и практических рекомендаций, полученных в диссертации, подтверждается корректным обоснованием постановок задач, точной формулировкой критериев, компьютерным моделированием, а также результатами внедрения предложенной технологии.

Практическое значение полученных результатов состоит в том, что они успешно используются и будут использоваться в дальнейшем при разработке промышленных и учебных программных проектов на основе автоматного подхода и языково-ориентированной парадигмы.

Внедрение результатов работы. Результаты диссертации использованы на практике в компании *JetBrains* (Санкт-Петербург) при разработке коммерческой системы учета ошибок *YouTrack*, сданной в эксплуатацию. Кроме того, автор был одним из разработчиков инструментального средства с открытым кодом для поддержки автоматного программирования *UniMod*. Это средство опубликовано на сайте *sourceforge.net* и скачано более 60 тысяч раз.

Полученные результаты используются также в учебном процессе на кафедре «Компьютерные технологии» СПбГУ ИТМО при выполнении

курсовых и бакалаврских работ по курсу «Теория автоматов в программировании».

Апробация диссертации. Основные положения диссертационной работы докладывались на конференциях и семинарах: II конференции молодых ученых СПбГУ ИТМО (2005 г.); XXXV, XXXVI научных учебно-методических конференциях СПбГУ ИТМО «Достижения ученых, аспирантов и студентов СПбГУ ИТМО в науке и образовании» (2006, 2007 гг.); «Телематика-2003», «Телематика-2004», «Телематика-2005», «Телематика-2006», «Телематика-2007» (СПбГУ ИТМО); на семинаре «Автоматное программирование» в рамках международной конференции «International Computer Symposium in Russia (CSR-2006)» (ПОМИ им. В. А. Стеклова, 2006 г.); Второй Всероссийской научной конференции «Методы и средства обработки информации» (МГУ, 2005 г.); Четвертой Всероссийской межвузовской конференция молодых ученых (СПбГУ ИТМО, 2009 г.); международной конференции «Software Engineering Conference (Russia)» (М., 2005 г.); форуме по открытому программному коду «Open Source Forum» (М., 2005 г.); второй международной научной конференции «Компьютерные науки и информационные технологии» (Саратов, 2007 г.); международной конференции «110 Anniversary of Radio Invention» (СПбГЭТУ «ЛЭТИ», 2005 г.).

Публикации. По теме диссертации опубликовано 31 научных работ, из которых 24 печатные. Результаты диссертации опубликованы в следующих журналах из списка ВАК: «Программирование», «Информационно-управляющие системы», «Научно-технический вестник СПбГУ ИТМО» и «Компьютерные инструменты в образовании».

Свидетельства об официальной регистрации программ для ЭВМ. На инструментальное средство для поддержки автоматного программирования, разработанное в рамках диссертации, получены следующие свидетельства: «Ядро автоматного программирования»

№2006 613249 от 14.09.2006, «Встраиваемый модуль автоматного программирования для среды разработки *Eclipse*» №2006 613817 от 7.11.2006.

Участие в научно-исследовательских работах. Основные результаты по теме диссертации получены в ходе научно-исследовательских и опытно-конструкторской работ, проводимых на кафедре «Компьютерных технологий» СПбГУ ИТМО в:

- 2002, 2003 гг. по теме «Разработка технологии автоматного программирования» (грант Российского фонда фундаментальных исследований по проекту 02–07–90114);
- 2002 – 2004 гг. по теме «Разработка технологии создания программного обеспечения систем управления на основе автоматного подхода» (Министерство образования и науки РФ);
- 2005, 2006 гг. по теме «Автоматное программирование» (Федеральная целевая научно-техническая программа «Исследования и разработки по приоритетным направлениям науки и техники» на 2002 – 2006 гг.);
- 2008 г. по теме «Разработка основных положений создания программных систем управления со сложным поведением на основе объектно-ориентированного и автоматного подходов» (Министерство образования и науки РФ);
- 2009 г. по теме «Методы повышения качества при разработке автоматных программ с использованием функциональных и объектно-ориентированных языков программирования» (Федеральная целевая программа «Научные и научно-педагогические кадры инновационной России» на 2009 – 2013 гг.).

Структура диссертации. Диссертация изложена на 136 страницах и состоит из введения, пяти глав и заключения. Список литературы содержит 119 наименований. Работа иллюстрирована 65 рисунками.

В первой главе приведен обзор особенностей автоматного программирования в зависимости от использования различных парадигм программирования. Приведен обзор различных взглядов на метапрограммирование и обоснована целесообразность исследования в области автоматного программирования для среды языково-ориентированного программирования. Сформулированы особенности среды языково-ориентированного программирования, приводящие к необходимости создания новых подходов к автоматному программированию.

Вторая глава посвящена созданию языка и инструментального средства для автоматного программирования в среде *MPS*. Описан подход к автоматной разработке, при котором код автомата пишется в виде текста, а диаграмма переходов восстанавливается из кода автоматически.

В третьей главе описан метод валидации автоматов, в условиях на переходах которых, существуют булевы формулы с предикатами сравнения входных переменных с константой и входных переменных друг с другом.

В четвертой главе описаны акторный язык и инструментальное средство для многопоточного автоматного программирования в среде *MPS*. Язык *actors* позволяет в привычной для *Java*-программистов манере использовать автоматически распараллеливающиеся функциональные конструкции. Описан способ применения языка *actors* совместно с языком автоматного программирования для многопоточного автоматного программирования.

Пятая глава содержит описание результатов внедрения разработанных средств автоматного программирования в промышленную разработку программного обеспечения.

ГЛАВА 1. АВТОМАТНАЯ И ЯЗЫКОВО-ОРИЕНТИРОВАННАЯ ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

1.1. Парадигма автоматного программирования

Автоматное программирование, иначе называемое «программирование от состояний» или «программирование с явным выделением состояний» – это подход к разработке программного обеспечения (ПО), основанный на расширенной модели конечных автоматов и ориентированный на создание широкого класса приложений. При этом речь идет не только и не столько об использовании конечных автоматов в программировании, сколько о методе создания программ в целом, поведение которых описывается автоматами. При этом *программы предлагается представлять, как совокупность автоматизированных объектов управления*. Автоматное программирование относится к такому направлению создания ПО, как *программная кибернетика* [1], так как оно базируется на идеях теории управления и теории автоматов.

Программирование с использованием автоматов имеет достаточно богатую историю развития. Различные аспекты и понятия, связанные с этой идеей, рассматривались в работах многих авторов с самых разных точек зрения и применительно к различным конкретным вопросам [2, 3]. Однако систематически автоматы использовались только при разработке компиляторов и протоколов. В настоящее время программирование от состояний рассматривается как один из стилей программирования [4].

Как парадигма разработки ПО, автоматное программирование сформировалось, в основном, благодаря усилиям А. А. Шалыто, который в 1991 г. предложил технологию для поддержки этого стиля программирования [5]. В его работах нашли отражение различные аспекты этого подхода к программированию, а краткое описание предлагаемой

парадигмы программирования содержится, например, в работе [6]. Более полное и исчерпывающее изложение сути автоматного программирования, как парадигмы и метода разработки программных систем, дано в работе [7].

Термин *автоматное программирование* родился в 1997 г. и был впервые использован во введении к работе [8]. На английский язык этот термин переводится как *Automata-based Programming*. Англоязычное название было предложено в работе [9]. Также этот подход известен под названием *SWITCH*-технология.

Парадигма автоматного программирования состоит в представлении сущностей со сложным поведением в виде автоматизированных объектов управления. Для этого сущность со сложным поведением разделяется на две части:

- управляющую часть, ответственную за поведение – выбор выполняемых действий, зависящий от текущего состояния и входного воздействия, а также за переход в новое состояние;
- управляемую часть, ответственную за выполнение действий, выбранных управляющей частью, и, возможно, за формирование некоторых компонентов входных воздействий для управляющей части – *обратных связей*.

В соответствии с теорией управления, управляемая часть называется *объектом управления*, а управляющая часть – *системой управления*. В рамках рассматриваемой парадигмы для реализации управляющей части используется система взаимосвязанных автоматов, каждый из которых называется *управляющим автоматом* или просто *автоматом*.

После разделения сущности со сложным поведением на объекты управления и автоматы реализовать ее уже несложно, а главное, ее реализация становится понятной и удобной для модификации. Вся логика поведения сущности сосредоточена в управляющих автоматах. Объекты управления, в свою очередь, обычно обладают простым поведением –

независящим от состояний. Поэтому объекты управления могут быть легко реализованы традиционными «неавтоматными» методами.

Таким образом, в соответствии с автоматным подходом, сущности со сложным поведением следует представлять в виде *автоматизированных объектов управления* – так в теории управления называют объект управления, интегрированный с системой управления в одно «устройство».

1.2. Особенности поддержки автоматного программирования на различных программных платформах

Автоматное программирование до последнего времени поддерживалось следующими программными платформами:

- специализированные языки программируемых логических контроллеров, например, лестничные схемы, функциональные блоки [10, 11];
- процедурные языки программирования, например, язык *C* [12];
- объектно-ориентированные языки программирования, например, языки *C++* [13] и *Java* [14];
- функциональные языки программирования, например, языки *Erlang* [15] и *Haskel* [16];
- динамические языки программирования, например, язык *Ruby* [17].

1.2.1. Специализированные языки программируемых логических контроллеров

Программирование не сводится к программированию на языках общего назначения. Существует также специализированные языки программирования логических контроллеров и языки для программирования, например, аппаратуры *VHDL* [18].

Среди языков программирования логических контроллеров рассмотрим *язык функциональных блоков* (*Function Block Diagram*). В работе

[19] приведено несколько примеров преобразования автоматов в функциональные блоки. Наиболее интересным из них является использование цифровых мультиплексоров. Реализация функционального блока цифровым мультиплексором может быть осуществлена с помощью конструкции *switch* какого-нибудь алгоритмического языка. Поэтому реализация графов переходов автоматов на цифровых мультиплексорах осуществляется практически также как и на конструкции *switch* в алгоритмических языках.

Для логических систем автоматное программирование напрямую не поддерживается, поэтому предлагается рисовать диаграммы состояний вручную, проводить формальный перевод в лестничные схемы или в функциональные блоки, а по построенной схеме генерировать программу.

1.2.2. Процедурное программирование

При применении автоматного программирования для процедурного подхода используется либо *проектирование сверху вниз*, либо *проектирование от объектов управления и событий*. Ниже эти подходы рассматриваются для одного автомата.

В случае проектирования сверху вниз:

1. По словесному описанию поведения системы строится набор *управляющих состояний*.
2. Строится *управляющий автомат* программной системы: управляющие состояния связываются между собой переходами, добавляются входные и выходные переменные, необходимые для реализации заданного в словесном описании поведения. Если система является событийной, то определяется набор событий, обрабатываемых автоматом. Они включаются в условия переходов наряду с входными переменными.

3. Входным переменным автомата сопоставляются *запросы*: булевы функции или (реже) двоичные переменные. Выходным переменным – *команды*: процедуры. Если упомянутые подпрограммы являются недостаточно простыми для непосредственной реализации, для каждой из них производится цикл традиционного проектирования сверху вниз.
4. Вводятся переменные, необходимые для корректной реализации упомянутых запросов и команд. Совокупность значений этих переменных определяет множество *вычислительных состояний* системы.

Автоматное проектирование сверху вниз целесообразно применять при проектировании всей системы с нуля.

При применении автоматного программирования в случае, когда объекты управления реализованы аппаратно или в виде готовых программных компонент, чаще используется проектировании от объектов управления и событий:

1. Исходными данными задачи считается не только словесное описание целевого поведения системы, но и (более или менее) точная спецификация набора *событий*, поступающих в систему из внешней среды, и множеств *запросов* и *команд* всех объектов управления.
2. Строится набор *управляющих состояний*.
3. Каждому запросу объектов управления ставится в соответствие входная переменная автомата, каждой команде – выходная. На основе управляющих состояний, событий, входных и выходных переменных строится автомат, обеспечивающий заданное поведение системы.

В качестве спецификации при процедурном автоматном программировании используются диаграммы двух видов:

- диаграмма переходов, описывающая состояния управляющего автомата, правила переходов между состояниями, действия в состояниях и на переходах, вложения автоматов в состояния и вызов автоматов;

- схема связей, описывающая связи автомата с объектами управления, множества входных и выходных воздействий автомата.

Для реализации автоматных программ при процедурном программировании используется изоморфное преобразование диаграмм переходов в код [8]. При этом каждому автомату соответствует конструкция *switch* (или аналогичная), ветвление в которой производится по значению переменной состояния автомата. Каждому состоянию соответствует конструкция *case*, внутри которой происходит дешифрация входных воздействий, выполнение выходных воздействий, выбор перехода и смена состояния. Например, шаблон для преобразования тела автомата Мура в код на языке *C* выглядит следующим образом:

```
void A() {
    switch (y) {            // Дешифратор состояний
        case 1:
            // Блок формирования выходных воздействий
            z1 = <0|1>; ...; zm = <0|1>;
            if (<метка_1>) {            // Дешифратор входных воздействий
                y = <1..s>;             // Обновление состояния
            } else if (<метка_2>) {
                y = <1..s>;
            } ... else if (<метка_k>) {
                y = <1..s>;
            }
            break;
        case 2:
        ...
        case <s>:
            ...
    }
}
```

В качестве примера инструментального средства для поддержки автоматного программирования на процедурных языках может быть приведена система *StateFlow* [20], входящая в широко известный программный пакет *Matlab* [21], которая обеспечивает визуальное автоматное программирование. Известны также инструментальные средства:

- *Visio2Switch* [22] для генерации кода на языке *C* по диаграмме, созданной в пакете *Microsoft Visio* [23];

- *MetaAuto* [24] для генерации тела автоматной программы на различных языках;
- *Visio2Auto* [25] для преобразования диаграмм *Microsoft Visio* в код на языке *C*.

1.2.3. Объектно-ориентированное программирование

Для создания объектно-ориентированных автоматных программ в работе [26] предложен метод их разработки. При использовании этого метода при построении диаграмм в рамках *SWITCH*-технологии сохраняется автоматный подход, но применяется стандартная *UML*-нотации. При этом предлагается, используя нотацию *UML*-диаграмм классов, строить схемы связей автоматов, а графы переходов – используя нотацию *UML*-диаграмм состояний. При наличии нескольких автоматов их схема взаимодействия не строится, а они все изображаются на диаграмме классов. Диаграммы классов (как схема связей) и диаграммы состояний образуют предлагаемый графический язык для описания структуры и поведения программ.

Для проектирования программ с использованием этого языка предлагается следующий метод:

1. На основе анализа предметной области в виде *UML*-диаграммы классов разрабатывается концептуальная модель системы, определяющая сущности и отношения между ними.
2. В отличие от традиционных для объектно-ориентированного программирования подходов [27], из числа сущностей выделяются источники событий, объекты управления и автоматы. Источники событий активны – они по собственной инициативе воздействуют на автоматы. Объекты управления пассивны – они выполняют действия, которые вызываются автоматами. Объекты управления также могут формировать значения входных переменных для автоматов. Автоматы

активируются источниками событий и на основании значений входных переменных и текущих состояний воздействуют на объекты управления, переходя в новые состояния.

3. Используя нотацию диаграммы классов, строится схема связей автоматов, которая задает интерфейс каждого их них. На этой схеме слева изображаются источники событий, в центре – автоматы, а справа – объекты управления. Источники событий с помощью *UML*-ассоциаций связываются с автоматами, которым они поставляют события. Автоматы связываются с объектами, которыми они управляют, а также с другими автоматами, которые они вызывают или которые вложены в их состояния.
4. Схема связей, кроме задания интерфейсов автоматов, выполняет функцию, характерную для диаграммы классов – задает объектно-ориентированную структуру программы.
5. Каждый объект управления содержит два типа методов, реализующих входные переменные (`xj`) и выходные воздействия (`zk`). При этом отметим, что объект управления инкапсулирует вычислительные состояния системы, которые обычно явно не выделяются из-за большого их числа.
6. Для каждого автомата с помощью нотации диаграммы состояний строится граф переходов типа *Мура-Мили*, в котором дуги могут быть помечены событием (`ei`), логической формулой из входных переменных и формируемыми на переходах выходными воздействиями.
7. В каждом состоянии могут указываться выходные воздействия, выполняемые при входе и имена вложенных автоматов, которые активны, пока активно состояние, в которое они вложены.

8. Кроме вложенности, автоматы могут взаимодействовать по вызываемости. При этом вызывающий автомат передает вызываемому событие, что указывается на переходе или в состоянии в виде выходного воздействия. Во втором случае посылка события вызываемому автомату происходит при входе в состояние.
9. Каждый автомат имеет одно начальное и произвольное число конечных состояний.
10. Состояния на графе переходов могут быть простыми и сложными. Если в состояние вложено другое состояние, то оно называется сложным. В противном случае состояние простое. Основной особенностью сложных состояний является то, что дуга, исходящая из такого состояния, заменяет однотипные дуги, исходящие из каждого вложенного состояния.
11. Все сложные состояния неустойчивы, а все простые, за исключением начального – устойчивы. При наличии сложных состояний в автомате, появление события может привести к выполнению более одного перехода. Это происходит в связи с тем, что, как отмечено выше, сложное состояние является неустойчивым, и автомат выполняет переходы до тех пор, пока не достигнет первого из простых (устойчивых) состояний. Отметим, что если в графе переходов сложные состояния отсутствуют, то, как и в *SWITCH*-технологии, при каждом запуске автомата выполняется не более одного перехода.
12. Каждая входная переменная и каждое выходное воздействие являются методами соответствующего объекта управления, которые реализуются вручную на целевом языке программирования. Источники событий также реализуются вручную.
13. Использование символьных обозначений в графах переходов позволяет весьма компактно описывать сложное поведение проектируемых систем.

Смысл таких символов задает схема связей. При наведении курсора на соответствующий символ на графе переходов во всплывающей подсказке отображается его текстовое описание.

Для поддержки этого метода автором и В. Гуровым было создано инструментальное средство *UniMod*, на основе которого В. Гуров защитил диссертацию на тему «Технология проектирования и разработки объектно-ориентированных программ с явным выделением состояний (метод, инструментальное средство, верификация)» [28]. Средство *UniMod* подробно описано в ряде работ [26, 29 – 40], опубликованных, в том числе и в журнале «Программирование» [41]. Это инструментальное средство обеспечивает валидацию автоматов и поддержку автозавершения пользовательского ввода. Для программ, построенных с помощью средства *UniMod*, обеспечивается их автоматическая верификация [42 – 44].

В качестве еще одного инструментальных средства, которое базируется на использовании автоматов можно привести *IBM Rhapsody*, развившееся из системы *Statemate* [45], разработанной еще в 1983 году. За эту систему в 2007 году коллектив разработчиков, возглавляемый Д. Харелом, получил премию *ACM Software System Award*.

1.2.4. Функциональные языки программирования

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 сравнении «чистого» 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. Это объясняется тем, что состояние автомата, по сути, является глобальной переменной, а обработка последовательностей событий в императивных языках производится при помощи циклов.

В работе [46] предложен метод реализации *структурных автоматов* на функциональных языках программирования на примере языка *Haskell*, при котором

- автомат представляется функцией переходов от состояния и события;
- события задаются при помощи алгебраических типов данных;
- цикл обработки событий реализуется в виде левой свертки списка событий по функции переходов.

Преимуществом использования функциональных языков программирования для реализации автоматов по сравнению с императивными языками программирования является строгая типизация составных частей конечного автомата.

1.2.5. Динамические языки

Реализация автоматов в коде с помощью описанных выше методов обладает следующими недостатками: в случае ручной реализации – трудоемкость изоморфного перехода от диаграммы к коду, в случае автоматической генерации кода – ухудшение читаемости кода. В работах [47, 48] предлагается использовать динамические возможности языка *Ruby* для создания проблемно-ориентированного языка для наиболее точной и изоморфной реализации автоматов в коде.

Для преобразования диаграммы переходов в код на этом языке предлагается использовать следующий метод:

1. Для каждой диаграммы создается новый класс с именем, совпадающим с названием автомата.
2. Внутри класса с помощью конструкций предлагаемого проблемно-ориентированного языка, описывается диаграмма переходов:
   - состояниям присваиваются идентификаторы, соответствующие их именам на диаграмме;
   - в описаниях переходов ссылки на состояния производятся по именам.
3. Класс автомата связывается с неавтоматной частью программы через подписку на выходные воздействия автомата.

Ключевыми моментами использования языка *Ruby* являются полностью декларативное описание поведения системы и автоматическое порождение специальных методов, позволяющих переносить диаграмму переходов с незначительными изменениями.

1.3. Языково-ориентированное программирование

В последние годы популярен подход к программированию, у которого много названий: модельно-ориентированный поход (*model driven architecutre*) [49], порождающее программирование (генеративное

программирование) [50], метапрограммирование [51], ментальное программирование или программирование намерений (*intentional programming*) [52], языково-ориентированное программирование [53]. Все эти названия отражают *различные аспекты* общего подхода к разработке ПО, и, так или иначе, связаны с разработкой проблемно-ориентированных или предметно-ориентированных языков (*domain specific language*) [54].

1.3.1. Модельно-ориентированный подход

Модельно-ориентированный подход к разработке программных систем состоит в описании функциональности систем в виде платформо-независимых моделей (*platform-independent model*) с помощью специальных проблемно-ориентированных языков программирования. Далее, имея модели описания платформы (*platform definition model*), такие как *CORBA* [55], *.NET* [56], *WEB* и т. п., платформо-независимые модели транслируются в платформо-зависимые модели (*platform-specific model*), которые могут быть запущены на компьютерах. Подобная трансляция требует задания правил отображения, которые также предлагается моделировать.

Платформо-зависимые модели могут быть описаны с помощью различных проблемно-ориентированных языков или языков общего назначения (например, *Java*, *C#*, *PHP*, *Python*). Как правило, для трансляции моделей используются автоматизированные средства.

Принципы модельно-ориентированного подхода применимы и в других областях, например, для моделирования бизнес-процессов [57], где платформо-независимые модели «транслируются» в автоматизированные или ручные процессы.

1.3.2. Порождающее программирование

Порождающее программирование рассматривает другой аспект программирования. В работе [50] авторы сравнивают традиционное программирование с самостоятельной сборкой автомобиля конечным

пользователем. Имеется в виду сборка программных систем из готовых компонент, в процессе которой программисту приходится дополнительно писать низкоуровневый код для интеграции компонент друг с другом.

При применении порождающего программирования предлагается настраивать и связывать компоненты с помощью высокоуровневых средств, а низкоуровневый код порождать автоматически. В качестве таких высокоуровневых средств могут быть использованы, например, проблемно-ориентированные языки программирования.

1.3.3. Метапрограммирование

Метапрограммированием называется написание программ (например, кодогенераторов), которые используют код других программ или свой собственный в качестве входных данных. Также под метапрограммированием понимается выполнение во время компиляции части работы, которую в противном случае пришлось бы выполнять во время исполнения. Во многих случаях применение метапрограммирования позволяет решать задачи более гибко и эффективно, чем при написании кода вручную.

Языки написания метапрограмм называются метаязыками. Языки, код на которых используется в качестве данных, называются *объектными языками*. Способность языков быть метаязыками для самих себя называется рефлексией (*reflection*) [58].

Языки и средства программирования обычно предоставляют возможности метапрограммирования одним из трех способов. Первый из них состоит в предоставлении программному коду доступа к внутренностям среды исполнения в виде программных интерфейсов приложений (*application programming interface*). Второй – динамическое выполнение строковых выражений, содержащих программный код. В этом случае «программы могут

писать программы». Несмотря на то, что оба подхода могут совмещаться в одном языке, большинство языков ориентированы на один из них.

Третий способ состоит в том, чтобы выйти за рамки языка и использовать средства трансформации программ, которые принимают на вход описания языков и могут применять различные правила трансформации к коду на этих языках. Эти средства метапрограммирования общего назначения могут применяться к любым целевым языкам вне зависимости от того существует ли в этих языках встроенные средства метапрограммирования.

1.3.4. Ментальное программирование

Предложенная Чарльзом Симони концепция ментального программирования состоит в приведении программного кода к виду как можно более точно отражающему ход мысли программиста. Точное воспроизведение уровня абстракции, на котором программист мыслит, упрощает навигацию и сопровождение программ.

Ч. Симони предлагает строить программные системы следующим образом. Программист сначала создает инструментальные средства разработки, специфичные для заданной предметной области. Затем эксперты в этой предметной области, применяя созданные средства разработки, описывают требуемое поведение приложения. После этого автоматизированная среда разработки порождает по описанному поведению целевой код программы. Компания *Intentional Software* [52], основанная Ч. Симони, занимается разработкой подобной среды программирования, которая называется *Intentional Programming* (*IP*).

Ключевой особенностью среды *IP* является то, что исходные коды хранятся не в текстовых файлах, а в бинарных. Такой формат хранения подобен языку *XML* [59] в том, что так же как и язык *XML* не требует

специального синтаксического анализатора для каждой конструкции языка. Это упрощает создание инструментов для работы с таким кодом.

Тесная интеграция редактора кода с бинарным форматом хранения позволяет привнести в код полезные свойства нормализации баз данных. Избыточность устраняется присваиванием каждому объявлению уникального идентификатора, что позволяет хранить имена всех переменных и операторов в единственном экземпляре. Многие задачи рефакторинга кода [60] и навигации при этом существенно упрощаются. Также в бинарном представлении не сохраняется информация о пробелах и символах разделителях, и таким образом каждый программист, работающий с проектом, может выбрать размер отступов при отображении кода в редакторе.

Разработчики среды *IP* предполагают совершить прорыв в порождающем программировании. Поскольку предлагаемый ими подход позволяет создавать проблемно-ориентированные языки без вложений в написание компилятора и среды разработки.

1.3.5. Языково-ориентированное программирование

При рассмотрении подходов, требующих разработки проблемно-ориентированных языков, часто основным достоинством считается то, что, создав однажды проблемно-ориентированный язык для некоторой предметной области, можно в дальнейшем использовать его многократно для более выразительного и компактного написания программ в данной предметной области. Предложенное Мартином Вардом [53] и развитое Сергеем Дмитриевым [61] языково-ориентированное программирование допускает создание проблемно-ориентированных языков даже для однократного применения.

М. Вард предлагает вместо традиционной «водопадной» модели разработки ПО [62] использовать модель «изнутри-наружу». В этой модели

разработка программной системы начинается с создания проблемно-ориентированного языка программирования, затем одна группа программистов реализует отображение конструкций этого языка в конечный код, а другая – описывает требуемое поведение системы на этом языке. Таким образом, устраняются риски свойственные моделям «сверху-вниз» и «снизу-вверх».

В модели «сверху-вниз» система изначально проектируется в самом общем виде, а затем последовательно уточняется. При этом существующие спецификации часто неформальны и допускают разночтения. Это приводит к тому, что на ранних стадиях разработки возникают ошибки, которые в дальнейшем крайне трудно устранить, так как они проникают во все уровни системы. При программировании «изнутри-наружу» такой проблемы нет, так как проблемно-ориентированный язык подходит для формальной спецификации поведения программы. Более того, изменения спецификации в процессе разработки не требуют изменений в реализации проблемно-ориентированного языка.

При использовании модели «снизу-вверх» сначала создаются низкоуровневые утилиты и процедуры, которые используются для создания более высокоуровневых процедур и типов данных и так далее до тех пор, пока не будет реализован самый верхний уровень системы. Такой подход позволяет последовательно создавать и тестировать компоненты системы, но на ранних стадиях такой разработки не вполне ясно, какие именно утилиты и процедуры потребуются, что приводит к созданию большого числа лишнего кода.

1.3.6. Среда языково-ориентированного программирования

Проблемно-ориентированными языками являются языки, специально предназначенные для решения задач из некоторой предметной области. Например, такими языками являются язык структурных запросов (*SQL*) [63]

для работы с базами данных, язык задания регулярных выражений (*regular expressions*) [64] для поиска подстрок по шаблону, язык запросов к элементам *XML*-документа (*XPath*) [65] для манипуляций с *XML*-документами. Сильная связь проблемно-ориентированных языков с предметной областью позволяет с их помощью описывать и решать задачи из данной предметной области очень компактно и выразительно. В теории применение проблемно-ориентированных языков выглядит достаточно просто: выделить для некоторой предметной области набор абстракций; создать язык, поддерживающий эти абстракции; решить программистскую задачу на этом языке. Тем не менее, в настоящее время при практическом программировании проблемно-ориентированные языки создаются редко.

В работе [66] М. Фаулер предполагает, что главным образом препятствует этому две причины. Во-первых, на практике, вместо расширений к существующим языкам общего назначения, прежде всего, рассматривается разработка замкнутых [67] проблемно-ориентированных языков. Это, возможно, связано с потенциальной неоднозначностью в текстовых языках программирования конструкций из разных языковых расширений. Во-вторых, создание развитой интегрированной среды разработки, поддерживающей созданный язык, является крайне трудоемкой задачей, а современная промышленная разработка ПО без использования такой среды невозможна в силу малой эффективности [68]. Далее эти причины будут рассмотрены подробнее, и будет показано, как они устраняются при применении среды метапрограммирования *MPS*.

Несмотря на то, что значительная часть усилий разработчиков проблемно-ориентированных языков направлена на создание самостоятельных языков, большей ценностью могут обладать языковые расширения, добавляющие новые проблемно-ориентированные конструкции в языки общего назначения, например, в язык *Java*. Языковые расширения, которые можно создавать и использовать также легко, как сейчас создаются

и используются библиотеки и каркасы, значительно ценнее для разработчика, чем самостоятельный проблемно-ориентированный язык, используемый лишь для части программы. Подобный подход позволяет разработчику одновременно пользоваться высоким уровнем абстракции проблемно-ориентированных языков программирования и мощью языков общего назначения.

Создаваемые расширения к языкам общего назначения должны быть совместимы друг с другом. Без совместимости переиспользование языковых расширений крайне затруднено. Совместимость означает, что если существует пара языковых расширений, например, одно, добавляющее в язык работу с денежными значениями, и другое, расширяющее математическую нотацию языка общего назначения новыми операциями, то должна существовать возможность использовать эти расширения совместно, несмотря на то, что они могут быть созданы независимо разными разработчиками.

Для решения задач расширяемости языков общего назначения и совместимости расширений, необходимо решить проблему неоднозначности конструкций в текстовых языках программирования. Большинство существующих языков реализовано с помощью текстовых грамматик, но такие грамматики могут быть неоднозначны – могут допускать различные интерпретации одинаковых программ. Более того, если удается по отдельности устранить неоднозначность для языка общего назначения с расширением *A* и для того же языка общего назначения с расширением *B*, то это еще не означает отсутствия неоднозначности в грамматике этого языка общего назначения с расширениями *A* и *B* одновременно. Это в результате приводит к возможной несовместимости таких языковых расширений.

Для продуктивной работы разработчикам необходимы «умные» средства разработки. С тех пор как получили распространение такие редакторы кода, как *IntelliJ IDEA* [69] и *Eclipse* [70], в промышленной

разработке обычные текстовые редакторы практически перестали использоваться. В обычных редакторах, например, нет подсветки ошибок, контекстной справки, функций автодополнения ввода. Существуют программные каркасы для «умных» редакторов, позволяющие добавлять поддержку для произвольных языков программирования, например, *IntelliJ IDEA Language API*, *XText* [71] и *Oslo* [72]. Однако эти каркасы не поддерживают совместимые языковые расширения. Кроме того, даже использование этих каркасов для создания языковой поддержки промышленного качества занимает много времени и требует высокой квалификации в области теории построения компиляторов.

Для устранения перечисленных препятствий к языково-ориентированной разработке в компании *JetBrains* была разработана среда метапрограммирования *MPS*, позволяющая быстро создавать языки программирования. Исторически термин «языково-ориентированное программирование» получил распространение позже начала работы над средой *MPS*, поэтому в названии использован близкий по смыслу термин «метапрограммирование». Тем не менее, среда, прежде всего, предназначена именно для языково-ориентированного программирования.

Неоднозначность текстовых грамматик устраняется в среде *MPS* отказом от хранения кода программ в текстовом виде. Отсутствие текстового кода избавляет от необходимости в текстовых грамматиках и, как следствие, от необходимости борьбы с неоднозначностями в синтаксических анализаторах. Вместо этого код хранится и редактируется его в виде абстрактного синтаксического дерева (АСД) [73]. Из этого, однако, не следует, что у языков среды *MPS* нет грамматик, но это не текстовые, а абстрактные грамматики. Абстрактная грамматика непосредственно описывает абстрактный синтаксис языка. Подобным образом язык *XML Schema* описывает грамматику *XML*-документов.

Отсутствие грамматической неоднозначности позволяет легко комбинировать языки друг с другом. Среда *MPS* позволяет добавлять новые конструкции в существующие языки (языковые расширения) и внедрять языки общего назначения в проблемно-ориентированные языки программирования. Поэтому языки в среде *MPS* совместимы. Например, для среды *MPS* создано множество совместимых расширений языка *Java*:

- язык для работы с коллекциями в функциональном стиле (*collections*);
- язык для поддержки дат и операций над ними (*dates*);
- язык, позволяющий использовать различные математические операции в привычной нотации, например, суммы и интервалы и т.д.

Так как языки в среде *MPS* не хранятся в текстовом виде, для редактирования кода невозможно использовать обычные текстовые редакторы. Для непосредственного редактирования АСД в среде *MPS* применяются *проекционные редакторы* [52]. При этом для каждого узла синтаксического дерева создается *проекция* – элемент графического пользовательского интерфейса. Взаимодействуя с проекциями узлов синтаксического дерева, пользователь редактирует его. Проекционные редакторы в среде *MPS* реализованы таким образом, что использование их практически не отличается от редактирования обычного текста. Например, если пользователь символ за символом вводит выражение «1 + 2 + 3», то среда *MPS* построит правильное АСД для этого выражения. Не все особенности ввода текстового кода удается воспроизвести для проекционных редакторов. Однако опыт промышленного применения среды *MPS* показал [66], что для адаптации нового разработчика к среде *MPS* требуется около двух недель, после которых разработчик редактирует код в среде *MPS* не менее эффективно, чем делал это ранее при помощи «умных» редакторов кода.

Хранение кода в виде АСД упрощает разработку возможностей «умного» редактирования кода. Многие из таких возможностей среда *MPS*

предоставляет для всех новых языков автоматически, например, автодополнение кода, поиск использований, рефакторинги переименования и безопасного удаления, поддержка систем контроля версий. Ранее в компании *JetBrains* была создана интегрированная среда разработки *IntelliJ IDEA*, для которой была добавлена поддержка большого числа текстовых языков и технологий программирования. В работе [66] указывается, что поддержка одного языка или технологии для среды *IntelliJ IDEA* занимает несколько человеко-месяцев, а реализация такой же функциональности в среде *MPS* занимает всего несколько дней. Такая эффективность достигается наличием в указанной среде готовой инфраструктуры и проблемно-ориентированных языков для конфигурирования этой инфраструктуры под разрабатываемый язык. Таким образом, в среде *MPS* по принципу раскрутки (*bootstrapping*) созданы проблемно-ориентированные языки для предметной области «разработка языков».

Кроме создания «умного» редактора среда *MPS* позволяет задавать и другие аспекты языков: системы типов, генераторы кода, потоки данных и т. д.

1.4. Задачи в области автоматного языково-ориентированного программирования, требующие своего решения

В настоящее время среда *MPS* успешно применяется в промышленной разработке. При участии автора в процессе создания в указанной среде системы учета программных ошибок *YouTrack*, было разработано около несколько проблемно-ориентированных языков программирования различных классов. В силу специфики и новизны подхода, реализация каждого из языков требует дополнительных исследований, так как простота создания языков в среде *MPS* открывает множество качественно новых возможностей. В ходе разработки указанной системы выявилась необходимость создания инструментальных средств для поддержки

автоматного программирования, так как оно наилучшим образом подходит для описания сущностей со сложным поведением.

Выбор автоматного программирования в качестве предмета диссертации связан с тем, что автор в течение последних восьми лет проводил исследования в этой области, используя и создавая различные средства автоматного программирования, начиная от применения автоматного программирования в разработке на *Macromedia Flash* [74] интерактивных обучающих программ [75 – 79], и кончая соавторством в разработке инструментального средства *UniMod*, которое опубликовано на сайте *sourceforge.net*, и скачано более 60 000 раз.

1.4.1. Валидация автоматных моделей

Для уменьшения числа ошибок при разработке автоматных программ необходимо проводить валидацию разрабатываемых автоматных моделей. Валидацией таких моделей называется проверка свойств автоматов, которые не зависят от пользовательской спецификации. Такими свойствами являются:

- достижимость любого состояния из начального состояния;
- достижимость из любого состояния конечного состояния;
- полнота и непротиворечивость [8] условий на переходах.

Задача автоматической валидации автоматных моделей с одноместными предикатами сравнения в условиях на переходах была решена автором ранее [80]. Для дальнейшего увеличения выразительной мощности автоматных моделей необходимо расширить класс формул для выражений условий на переходах двуместными предикатами сравнения входных переменных.

1.4.2. Автоматы в многопоточных средах

В связи с тем, что дальнейшее повышение производительности вычислительных систем более не может достигаться увеличением тактовой

частоты процессоров, и главной тенденцией становится рост числа ядер в них [81], растет актуальность разработки параллельных программ [82]. Однако для того, чтобы программа выполнялась эффективно на нескольких ядрах, она должна быть написана с применением техник параллельного программирования [83].

Универсальные императивные языки программирования [84], наиболее распространенные на сегодняшний день, хорошо подходят для реализации последовательных алгоритмов, но плохо приспособлены для написания алгоритмов параллельных. Это связано с тем, что программа, написанная на императивном языке, представляет собой последовательность инструкций процессору, изменяющих состояние памяти [85]. В виду того, что порядок этих команд зафиксирован, выполнение программы не может быть распределено между несколькими процессорами. Поэтому программисты вынуждены самостоятельно разбивать свою программу на параллельно выполняющиеся потоки [86].

Автоматическому распараллеливанию хорошо поддаются программы, написанные на функциональных языках программирования [4, 87]. Но функциональные языки, в отличие от императивных, значительно менее распространены. Тем не менее, некоторые функциональные конструкции в последнее время стали активно проникать в универсальные императивные языки программирования. Например, становится популярным использование в императивных языках замыканий [88] для обработки списков, так как в некоторых случаях [89] это позволяет выполнять автоматическое распараллеливание по данным [90].

Поэтому естественно продолжить адаптацию абстракций параллельных функциональных программ для императивных языков программирования и, в частности, для автоматного программирования.

1.4.3. Задачи, решаемые в диссертации

Среда *MPS* обладает следующими особенностями, которые приводят к необходимости новых разработок в рамках автоматного программирования:

1. Специфика создания языков, без разработки компиляторов.
2. Возможность встраивать сообщения об ошибках на лету в процессе программирования позволяет сообщать об ошибках валидации. В известных системах программирования автоматов, например, *Rational Rose* (*IBM*) отсутствовала автоматическая проверка полноты и непротиворечивости автоматов. Поэтому во многих случаях реализованные исходно таким образом автоматы функционировали некорректно. В инструментальном средстве *UniMod* валидация выполнялась, но увеличение выразительной мощности выражений условий на переходах, приводит к необходимости разработки новых методов валидации автоматов.
3. Возможность интеграции в среде *MPS* языков друг с другом. Поэтому открылась возможность построения языковых средств многопоточного автоматного программирования. Традиционная реализация многопоточного автоматного взаимодействия предполагает использование общей памяти, что приводит к необходимости синхронизации потоков непосредственно в программном коде.

Для устранения указанных недостатков были выполнены исследования, которые легли в основу данной диссертации. При этом задачи, решаемые в диссертационной работе, состоят в следующем:

1. Разработка текстового языка автоматного программирования (абстрактного и конкретного синтаксиса, операционной семантики, системы типов, генераторов).
2. Разработка средства валидации автоматов в среде *MPS*.
3. Создание языка для многопоточного программирования и интеграция его с автоматным языком, позволяющих отказаться от использования

разделяемой памяти при многопоточном автоматном программировании.

4. Внедрение результатов работы в учебный процесс и в практику проектирования языков и программных систем в среде *MPS*.

Выводы по главе 1

1. Выполнен обзор особенностей автоматного программирования в зависимости от использования различных парадигм программирования.
2. Выполнен обзор различных взглядов на метапрограммирование и обоснована целесообразность исследования в области автоматного программирования для среды языково-ориентированного программирования.
3. Сформулированы особенности среды языково-ориентированного программирования, приводящие к необходимости создания новых подходов к автоматному программированию.
4. Сформулированы задачи, решаемые в диссертации.

ГЛАВА 2. ТЕКСТОВЫЙ ЯЗЫК АВТОМАТНОГО ПРОГРАММИРОВАНИЯ

В рамках проекта *UniMod* [32] предложены метод и средство для моделирования и реализации объектно-ориентированных программ со сложным поведением на основе автоматного подхода. Модель системы в рамках указанного проекта предлагается строить с помощью двух типов *UML*-диаграмм: диаграмм классов и диаграмм состояний [30]. При этом на диаграмме классов, которая представляется в виде схемы связей и взаимодействия автоматов, изображаются источники событий, автоматы и объекты управления, которые реализуют функции входных и выходных воздействий. Код для источников событий и объектов управления пишется на языке *Java*. Поведение автоматов описывается с помощью диаграмм состояний.

С использованием инструментального средства *UniMod* выполнен ряд проектов, который доступны по адресу http://is.ifmo.ru/unimod-projects/. Данные 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;
- многие программисты предпочитают работать с текстовым представлением программы, несмотря на то, что диаграммы позволяют представлять информацию более обозримо;
- невозможно в одном *Java*-классе совместить автомат и объект управления, что не позволяет прозрачно использовать автоматное программирование совместно с объектно-ориентированным, так как в

настоящее время код, генерируемый из автоматной модели, не является в полной мере объектно-ориентированным.

2.1. Текстовые языки автоматного программирования

В настоящее время существует несколько текстовых языков, поддерживающих программирование автоматов, как в рамках автоматной парадигмы (*UniMod FSML* [91], *State Machine* [92]), так и вне ее (*SMC* [93], *TABP* [94], *SCXML* [95], *Ragel* [96]). Каждый из этих языков программирования обладает некоторыми недостатками и ограничениями.

Язык *State Machine*, представляет собой расширение языка *Java*, основанное на паттерне проектирования *State Machine* [97]. Этот паттерн позволяет в объектно-ориентированном стиле описывать структуру автомата, однако такое описание оказывается громоздким, что препятствует использованию языка *State Machine* в реальной разработке.

Язык *SMC* – универсальный компилятор автоматов. Этот язык позволяет описывать в автоматы в едином формате и генерировать целевой код на различных языках общего назначения. Однако интеграция этого языка с целевыми языками требует от разработчиков дополнительных усилий – необходимо существенно модифицировать код класса для того, чтобы связать его с автоматом, описывающим его поведение.

В универсальном языке программирования *TABP* предусмотрены встроенные конструкции, облегчающие написание автоматов. Однако синтаксис этого языка вынуждает разработчика оперировать понятиями более низкоуровневыми, чем «состояние». Кроме того, универсальность языка *TABP* доставляет дополнительные трудности разработчику, связанные с необходимостью изучать конструкции нового языка даже для тех задач, для которых подходят более распространенные универсальные языки.

Язык *Ragel* предназначен для описания конечных автоматы с помощью регулярных выражений. Такой подход ограничивает область

применимости этого языка задачами лексического анализа и спецификации протоколов,

Стандарт языка *SCXML* – спецификация *XML*-представления автоматов Харела [2]. Использовать этот язык непосредственно для написания кода автоматов на нем крайне затруднительно, скорее он подходит для переноса автоматов Харела из одного приложения в другое.

Существуют и другие языки, так или иначе позволяющие использовать автоматы при программировании. Это доказывает популярность автоматного подхода для решения различных задач программирования. Однако перечисленные недостатки затрудняют его применение. Кроме того для перечисленных языков не существует развитых инструментальных средств разработки. Поэтому большинство программистов отдает предпочтение языкам общего назначения, такие средства существуют.

2.2. Создание языков в среде *MPS*

Для устранения указанных недостатков при участии автора был предложен новый подход к разработке автоматных программ и применению автоматов в объектно-ориентированных системах [98, 99]. В рамках этого подхода предлагается использовать *среду языково-ориентированного программирования MPS* [61], которая позволяет создавать проблемно-ориентированные языки.

Для задания языка в среде *MPS* требуется разработать:

- структуру абстрактного синтаксического дерева (АСД) [73] для разрабатываемого языка. Узлам АСД могут соответствовать такие понятия как «объявление класса», «вызов метода», «операция сложения» и т. п.;
- модель проекционного редактора для каждого типа узла АСД. Задание редактора для узла АСД равноценно заданию конкретного синтаксиса

для этого узла. При этом если для традиционных текстовых языков программирования создание удобного редактора – отдельная сложная задача, то для языков, созданных с помощью среды *MPS*, редакторы являются частью языка. Эти редакторы поддерживают автоматическое завершение ввода текста, навигацию по коду, выделение цветом ошибок в коде программы и т. д.;

- модель системы типов [100] для языка;
- модель трансформации программы на задаваемом языке в исполняемый код.

Таким образом, создание языка в среде *MPS* сопровождается созданием инструментального средства разработки программ на этом языке.

Отметим, что среда *MPS* позволяет, как создавать новые языки, так и расширять языки уже созданные с помощью этой системы.

В отличие от традиционных языков, языки, созданные с помощью среды *MPS*, не являются текстовыми в традиционном понимании, так как при программировании на них пользователь пишет не текст программы, а вводит ее в виде АСД с помощью специальных проекционных редакторов. Структура и внешний вид этих редакторов таковы, что работа с моделью программы для пользователя выглядит, как традиционная работа с текстом программы. Отказ от традиционного текстового ввода программ значительно упрощает создание новых языков [52] – исчезает необходимость в разработке лексических и синтаксических анализаторов, и, как следствие, перестают действовать ограничения на класс грамматик языков. Недостатком такого подхода является зависимость языков от среды *MPS* – невозможно разрабатывать программы без этой среды. Однако подобное ограничение присуще и традиционным, чисто текстовым языкам, которые зависят от компиляторов. Впрочем, после трансляции программы, написанной на языке, созданном в среде *MPS*, исполняемый код перестает зависеть от этой среды.

Ядро среды *MPS* написано на кросс-платформенном языке *Java*. В связи с этим в среде *MPS* существуют развитые средства для взаимодействия с *Java*-платформой. Среда *MPS* позволяет писать код только на тех языках, которые созданы в этой среде, поэтому для написания *Java*-кода в среде *MPS* разработан язык *baseLanguage*. Этот язык является почти полной реализацией спецификации *Java* 5 [101]. В нем определены такие конструкции, как «класс», «интерфейс», «метод», «предложение», «выражение» и т. д.

2.3. Язык *stateMachine*

Разработанный автором язык автоматного программирования в среде *MPS* назван *stateMachine*. Он представляет собой автоматное расширение языка *baseLanguage.* Основная цель, которая преследовалась при разработке языка *stateMachine*, – создание языка, позволяющего описывать поведение классов в виде автоматов, не накладывая на сами классы никаких дополнительных ограничений. Более того, автоматное описание поведения класса должно быть инкапсулировано – код, использующий класс не должен «знать» о том, каким образом задано поведение класса. Этот подход отличается от предложенного в работе [32] тем, что позволяет использовать автоматы не только в программах, написанных в соответствии со *SWITCH*-технологией, но и в традиционных объектно-ориентированных программах.

В качестве примера использования языка *stateMachine* рассмотрим систему управления лифтом [102]. При этом отметим, что функциональность этой системы меньше по сравнению с системой, описанной в работе [103]. Поэтому логика управления всеми подсистемами лифта реализована в одном автомате.

Каждый автомат в языке *stateMachine* связан с некоторым классом и описывает его поведение. Для того чтобы задать поведение класса с помощью автомата необходимо в этом классе определить события, на

которые будет реагировать автомат. Особенностью языка *stateMachine* является то, что события в нем – это методы специального вида. В языке *Java* декларации методов различаются по способу реализации:

- обычные методы, реализация которых следует сразу за объявлением метода;
- нативные методы, реализованные на платформо-зависимых языках программирования;
- абстрактные методы, которые вообще не имеют реализации, непосредственно связанной с декларацией.

В диссертации предлагается еще один способ реализации – декларация метода, реализация которого находится в автомате и зависит от текущего его состояния. Для этого язык *stateMachine* расширяет язык *baseLanguage* конструкцией *событие* (*EventMethodDeclaration*), соответствующего указанной декларации метода. Например, в классе *Elevator* объявлены следующие события (рис. 1):

- «*doorsOpened*», «*doorsClosed*» – события, которые посылает объект управления *DoorsEngine,* когда двери оказываются в максимально открытом и полностью закрытом положении соответственно;
- «*floorReached*» – событие, которое посылает объект управления *ElevatorEngine,* когда лифт достигает очередного этажа;
- «*call*» – событие, которое получает автомат, когда пассажир нажимает кнопку вызова лифта на этаже или в кабине лифта;
- «*openDoors*» – событие, которое получает автомат, когда пассажир нажимает кнопку экстренного открытия дверей в кабине лифта;
- «*loadingTimeout*» – событие, которое посылает объект управления *LoadingTimer*, когда истекает время ожидания погрузки пассажиров;
- «*fire*» – событие, которое автомат получает в случае обнаружения возгорания.

```
public class Elevator extends <none>  implements DoorsEngineListener    {
                                                  ElevatorEngineListener
                                                  LoadingTimerListener
  <<initializer>>
  <<static fields>>
  public int currentFloor = 1;
  public TaskList tasks = new TaskList();
  public ElevatorEngine elevatorEngine;
  public DoorsEngine doorsEngine;
  public LoadingTimer loadingTimer;
  <<properties>>

  public Elevator(ElevatorEngine elevatorEngine, DoorsEngine doorsEngine,
    this.elevatorEngine = elevatorEngine;
    this.doorsEngine = doorsEngine;
    this.loadingTimer = loadingTimer;
    this.elevatorEngine.addElevatorEngineListener(this );
    this.doorsEngine.addDoorsEngineListener(this );
    this.loadingTimer.addLoadingTimerListener(this );
  }


  public event doorsOpened( );
  public event doorsClosed( );
  public event floorReached(int floor);
  public event call(int floor, CallType callType);
  public event openDoors( );
  public event loadingTimeout( );
  public event fire( );
```

Рис. 1. События автомата управляющего лифтом

Так как события являются обычными методами, их можно использовать в качестве реализации абстрактных методов. Это позволяет уменьшить число зависимостей в программе, реализуя паттерн проектирования «Обозреватель» [97]. Например, в рассматриваемой системе объект управления *ElevatorEngine* не имеет непосредственных ссылок на класс *Elevator*. Вместо этого в программе управления лифтом объявлен интерфейс *ElevatorEngineListener*, и объект управления *ElevatorEngine* извещает о событиях экземпляры этого интерфейса. Класс *Elevator* реализует интерфейс *ElevatorEngineListener* и поэтому может обрабатывать события объекта управления *ElevatorEngine*.

Реакция автомата на событие зависит от состояния, в котором автомат находится. В каждом состоянии для каждого события может быть определен набор переходов. Для перехода может быть задано *условие на переходе*, *действие на переходе* и *целевое состояние*. В процессе обработки события

для текущего состояния перебираются все переходы, помеченные данным событием, до тех пор, пока не будет найден первый переход, условие на котором выполняется. Если такой переход найден, то выполняется действие на переходе и осуществляется переход в целевое состояние. Отсутствие условия на переходе интерпретируется, как тождественно истинное условие.

События в языке *stateMachine* могут иметь параметры. На переходе с каждым параметром события может быть связано *фильтрующее значение*. Если такое значение задано, то переход выполняется только, если значение параметра обрабатываемого события совпадает с фильтрующим значением.

Состояния автомата бывают двух типов: обычные и конечные. Конечные состояния отличаются тем, что не могут иметь исходящих переходов. Следовательно, когда автомат оказывается в конечном состоянии, он перестает обрабатывать события.

Для состояния могут быть определены действия при входе в состояние и действия при выходе из состояния.

Первое по порядку состояние, объявленное в автомате, считается начальным. При создании экземпляра класса, для которого определен автомат, после выполнения конструктора осуществляется переход в начальное состояние.

Обычные состояния могут быть вложены друг в друга. При этом если состояние содержит другие состояния, то оно называется составным. При переходе в составное состояние выполняется переход в первое по порядку вложенное в него состояние. Поэтому автомат после обработки события не может оказаться в составном состоянии. Каждый исходящий из составного состояния переход работает так, как если бы такой переход был добавлен к каждому вложенному состоянию.

Каждый автомат в языке *stateMachine* связан с некоторым классом, имеет доступ к его полям и может вызывать методы этого класса. Поэтому в языке *stateMachine* нет необходимости в специальных конструкциях для

взаимодействия между автоматами, так как вместо вложения одного автомата в другой можно использовать агрегацию одного *автоматного класса* другим автоматным классом, а посылка события из одного автомата в другой не что иное, как вызов метода.

С каждым автоматом может быть связано несколько объектов-слушателей (*listener*), которые оповещаются о стадиях обработки события. Например, таким слушателем может быть объект, выводящий сообщения о действиях автомата в специальный файл протокола работы автомата. Слушатели оповещаются о следующих стадиях работы автомата:

- начало и завершение обработки события;
- переход в состояние и переход из состояния;
- проверка условия на переходе;
- обнаружение для текущего состояния и данного события переход, условие на котором выполняется;
- игнорирование события, для которого не обнаружено перехода из текущего состояния;
- начало и завершение выполнения действия на переходе;
- начало и завершение выполнения действия при входе в состояние;
- начало и завершение выполнения действия при выходе из состояния.

После написания программы на языке *stateMachine*, она сначала транслируется в *Java*-код, а затем компилируется стандартным *Java*-компилятором. Достоинством этого языка является простота его использования в объектно-ориентированных приложениях, написанных на языке *Java*. При применении этого языка проверка корректности программы выполняется на стадии ее написания, а не в процессе компиляции.

2.4. Структура языка *stateMachine*

2.4.1. Метаязык описания структуры языков

Абстрактный синтаксис языка задается в среде *MPS* при помощи проблемно-ориентированного метаязыка *structure*. Этот язык позволяет описывать типы конструкций языка – *концепты*, их свойства и отношения между ними в объектно-ориентированном стиле. На рис. 2 в качестве примера приведен концепт *Transition*, описывающий конструкцию *переход* языка *stateMachine*.

```
concept Transition extends     BaseConcept
                   implements  IStateElement

  instance can be root: false

  properties:
  << ... >>

  children:
```

TransitionTrigger	transitionTrigger	1	**specializes:**	<none>
TransitionAction	transitionAction	1	**specializes:**	<none>

```
  references:
  << ... >>

  concept properties:
  alias = on

  concept links:
  << ... >>

  concept property declarations:
  << ... >>

  concept link declarations:
  << ... >>
```

Рис. 2. Объявление концепта *Transition*

Объявление концепта определяет структуру экземпляра концепта (узла АСД) – какие свойства узел может иметь, на какие узлы он может ссылаться, и какие узлы он может содержать в качестве дочерних. При

определении концепта также задаются *метасвойства* (*concept property*) и *метасвязи* (*concept links*), описывающие структуру самого концепта, а не его узлов.

2.4.2. Иерархия концептов

Существует два типа концептов: собственно концепты и *интерфейсы концепта*. Интерфейсы концепта применяются для описания особенностей, которые могут быть реализованы несколькими разными концептами. Например, если некоторый концепт имеет имя, по которому на него можно ссылаться, то он может реализовывать интерфейс концепта *INamedConcept*. Используются интерфейсы концептов так же, как интерфейсы в объектно-ориентированных языках программирования.

Любой концепт может наследоваться от другого концепта и реализовывать несколько интерфейсов концепта. Эта система похожа на иерархию классов в языке *Java*, где класс может наследоваться от другого класса и реализовывать несколько интерфейсов, а интерфейсы могут наследоваться от других интерфейсов. Когда концепт наследуется от другого концепта или реализует интерфейс, он транзитивно наследует все члены родительских концептов и реализованных интерфейсов.

2.4.3. Свойства

Для каждого концепта и интерфейса концепта может быть определен набор свойств (*properties*). Значения свойств хранятся внутри экземпляров концепта. Каждое свойство имеет один из следующих типов:

- примитивный тип, например, логический, строковый и целочисленный;
- перечислимый тип, имеющий значение из предопределенного набора значений;
- ограниченный тип – строковый тип, множество значений которого ограничено регулярным выражением.

На рис. 3 приведен пример задания свойства *name* для интерфейса концепта *INamedConcept*.

```
interface concept INamedConcept extends <none>

  properties:
  name : string

  children:
  << ... >>

  references:
  << ... >>
```

Рис. 3. Интерфейс концепта *INamedConcept* со свойством *name*

2.4.4. Ссылки

Узлы *АСД* могут иметь ссылки (*references*) на другие узлы. В разделе ссылок объявления концепта могут быть объявлены ссылки, которые могут иметь экземпляры концепта. Каждое объявление ссылки состоит из роли (названия), типа и мощности. Тип ссылки накладывает ограничение на концепт узла, на который ведет ссылка. Мощность определяет, сколько ссылок данного типа может иметь узел. Ссылки могут иметь только два вида мощностей: «0..1» и «1». Мощность «0..1» означает, что каждый узел данного концепта может иметь, а может и не иметь соответствующую ссылку, а мощность «1» означает, что узел должен иметь соответствующую ссылку в единственном экземпляре. Например, концепт *EventParameterReference*, соответствующий выражению использования параметра события, имеет одну ссылку на концепт *EventParameterDeclaration*, с ролью *eventParameterDeclaration* и мощностью «1» (рис. 4).

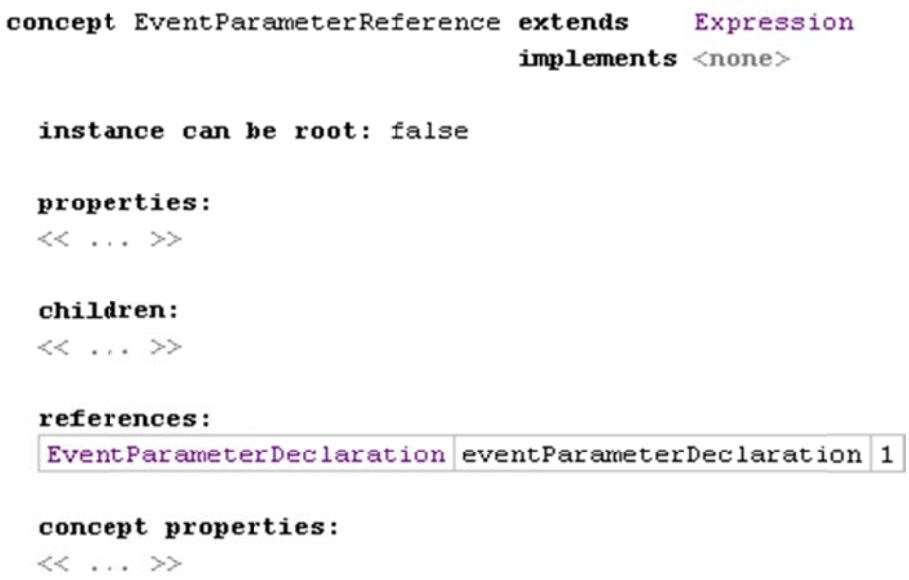

Рис. 4. Концепт *EventParameterReference* со ссылкой *eventParameterDeclaration*

2.4.5. Дочерние узлы

В разделе дочерних узлов (*children*) объявления концепта определяется, какие узлы могут быть агрегированы внутрь экземпляра концепта. Каждое объявление группы дочерних узлов состоит из роли (названия), типа и мощности. Тип дочернего узла накладывает ограничение на концепты узлов, которые могут быть дочерними для экземпляров данного концепта. Дочерним может быть узел концепта, указанного в качестве типа, или любого унаследованного от него концепта. Роль определяет название данной группы дочерних узлов. Наконец, мощность определяет то, сколько дочерних узлов данной группы может содержаться внутри одного узла. Всего существует четыре разрешенных вида мощностей для дочерних узлов:

- 1 – ровно один дочерний узел;
- 0..1 – ноль или один дочерний узел;
- 0..n – ноль или более дочерних узлов;
- 1..n – один или более дочерних узлов.

2.4.6. Метасвойства

Для того чтобы определить члены специфические для самого концепта, а не для его узлов, применяются *метасвойства* и *метасвязи*. *Метасвойства* и *метасвязи* используются не для задания значений общих для всех узлов концепта, а для описания метаособенностей концепта. Этим *метасвойства* и *метасвязи* отличаются от статических полей класса в объектно-ориентированных языках.

Например, для того чтобы помечать концепты как абстрактные, в среде *MPS* используется метасвойство *abstract*. Оно объявлено в самом общем концепте *BaseConcept* и имеет логический тип (рис. 5).

```
concept BaseConcept extends    <default>
                    implements <none>

  instance can be root: false

  properties:
  shortDescription : string
  alias            : string
  virtualPackage   : string

  children:
  << ... >>

  references:
  << ... >>

  concept properties:
  abstract

  concept links:
  << ... >>

  concept property declarations:
  boolean abstract                    <inheritable: false >
  string  alias                       <inheritable: false >
  string  shortDescription            <inheritable: false >
  boolean dontSubstituteByDefault     <inheritable: true  >
  string  deprecated104               <inheritable: false >
  boolean substituteInAmbigousPosition <inheritable: false >
```

Рис. 5. Объявление метасвойства *abstract*

Все концепты в среде *MPS* прямо или косвенно наследуются от концепта *BaseConcept* так же, как в языке *Java* все классы прямо или косвенно наследуются от класса *Object*. Поэтому для каждого концепта в среде *MPS* может быть определено свое значение метасвойства *abstract*. Истинные значения метасвойств логического типа у концепта задаются упоминанием метасвойства в объявлении концепта. Например, концепт *AbstractState* в языке *stateMachine* является абстрактным. В разделе метасвойства (*concept properties*) этого концепта упомянуто метасвойство *abstract* (рис. 6).

```
concept AbstractState extends     BaseConcept
                      implements  INamedConcept
                                  IStateElementHolder
                                  IStateElement

  instance can be root: false

  properties:
  << ... >>

  children:
  << ... >>

  references:
  << ... >>

  concept properties:
  abstract

  concept links:
  applicableStateElement = OnEnterAction

  concept property declarations:
  << ... >>
```

Рис. 6. Задание метасвойства *abstract* для концепта *AbstractState*

2.4.7. Шаблон метапрограммирования *список опций*

Для задания метаотношений между концептами применяются метасвязи. Например, в языке *stateMachine* для описания элементов автомата используется шаблон метапрограммирования «список опций» (*option list)*.

Этот шаблон применяется для описания концептов, дочерние узлы которых опциональны и могут идти вперемешку. Допустимые концепты опциональных дочерних узлов задаются в виде метасвязи.

Базовым концептом для всех элементов (опций) автомата является интерфейс концепта *IStateElement*. Базовым концептом контейнером элементов автомата является интерфейс концепта *IStateElementHolder* (рис. 7).

```
interface concept IStateElementHolder extends <none>

  properties:
  << ... >>

  children:
  IStateElement | stateElement | 0..n | specializes: | <none>

  references:
  << ... >>

  concept properties:
  << ... >>

  concept links:
  << ... >>

  concept property declarations:
  << ... >>

  concept link declarations:
  reference applicableStateElement target concept: ConceptDeclaration
```

Рис. 7. Объявление интерфейса концепта *IStateElementHolder*

Концепты, являющиеся элементами автомата, например, состояния, переходы, действия при входе в состояние и при выходе из него, должны реализовывать интерфейс концепта *IStateElement*. Концепты, содержащие элементы автомата, например, состояния или сам автомат, должны реализовывать интерфейс концепта *IStateElementHolder* и указывать с помощью метасвязи *applicableStateElement* концепты, которые допустимы в качестве элементов данного концепта. Ранее на рис. 6 было представлено

объявление концепта *AbstractState*, который, с одной стороны, является концептом-элементом автомата, с другой, – концептом-контейнером для элементов автомата и может содержать узлы концепта *OnEnterAction*.

2.4.8. Диаграмма классов для структуры языка *stateMachine*

Объектно-ориентированный стиль задания абстрактного синтаксиса языка в среде *MPS* позволяет использовать диаграмму классов для представления структуры и отношений концептов языка. На рис. 8 представлена диаграмма классов для концептов языка *stateMachine*.

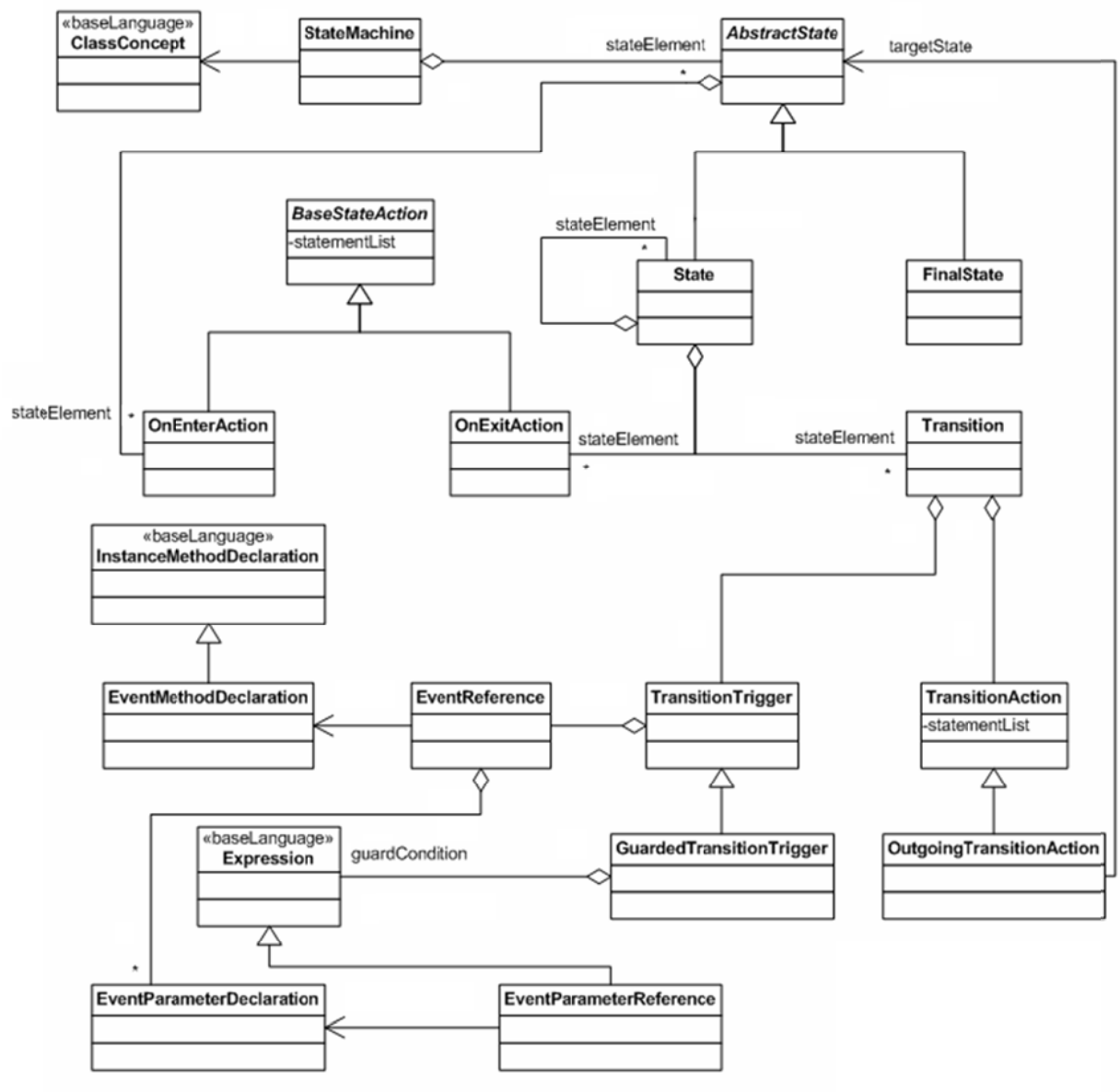

Рис. 8. Диаграмма классов для концептов языка *stateMachine*

Концепты, помеченные стереотипом «*baseLanguage*», принадлежат языку *baseLanguage*. Для упрощения диаграммы на ней не упомянуты интерфейсы концептов *IStateElement* и *IStateElementHolder*.

2.5. Конкретный синтаксис языка *stateMachine*

Для задания конкретного синтаксиса языка в среде *MPS* применяется проблемно-ориентированный метаязык *editor*. Этот метаязык позволяет для каждого концепта указать правила отображения экземпляров этого концепта в набор ячеек проекционного редактора.

Проекционный редактор среды *MPS* состоит из ячеек, которые могут содержать другие ячейки, некоторый текст или компонент графического пользовательского интерфейса. Проекционный редактор для узла является его отображением и контроллером. Проекционный редактор отображает узел и позволяет редактировать его свойства, добавлять и удалять его дочерние узлы.

Правило отображения экземпляров концептов в набор ячеек проекционного редактора называется редактором концепта. С каждым неабстрактным концептом связан редактор. Если редактор не задан для концепта явно, то используется редактор его базового концепта.

2.5.1. Типы ячеек в языке *editor*

Для задания редакторов концептов метаязык *editor* позволяет использовать ячейки следующих типов:

- *константная ячейка*. Ячейки этого типа могут содержать только фиксированный, предопределенный при объявлении текст. Константные ячейки, как правило, используются для отображения *ключевых слов* языка, операторных скобок и т. п.;
- *коллекция ячеек* – ячейка, в которую вложены другие ячейки. Коллекция ячеек может быть:

- горизонтальной – вложенные ячейки размещаются в строку;
- вертикальной – вложенные ячейки размещаются одна под другой;
- горизонтальной с переносами (*indent layout*) – вложенные ячейки размещаются горизонтально, но если строка из ячеек слишком длинная, то они переносятся, как текст на следующую строку, с отступом перед каждой следующей строкой;

- *ячейка свойства* связана с некоторым свойством концепта. Эта ячейка для экземпляра концепта отображает текущее значение свойства. Изменение пользователем содержимого ячейки приводит к изменению значения свойства;
- *ячейка ссылки* используется для представления ссылок. Каждая ячейка ссылки состоит из двух частей: левая часть указывает на объявление ссылки, правая определяет внешний вид ссылки. Например, редактор концепта *EventParameterReference* (рис. 9),

```
editor for concept EventParameterReference
  node cell layout:
    ( % eventParameterDeclaration % -> ( name ) )

  inspected cell layout:
    <choose cell model>
```

Рис. 9. Использование ячейки ссылки в редакторе концепта *EventParameterReference*

соответствующего конструкции использования параметра события, состоит из ячейки для ссылки *eventParameterDeclaration*, для отображения которой будет использована ячейка свойства *name* концепта *EventParameterDeclaration*. Таким образом, конструкция использования параметра события в проекционном редакторе будет представлена в виде имени применяемого параметра;

- *ячейка дочернего узла* позволяет указать среде *MPS* место в редакторе концепта, в котором должен быть размещен редактор для дочернего узла;
- *ячейка коллекции дочерних узлов* – ячейка, содержащая несколько ячеек дочерних узлов одной роли. Например, редактор для события на переходе (концепт *EventReference*) содержит ячейку для коллекции дочерних узлов с ролью *eventParameterDeclaration* (рис. 10).

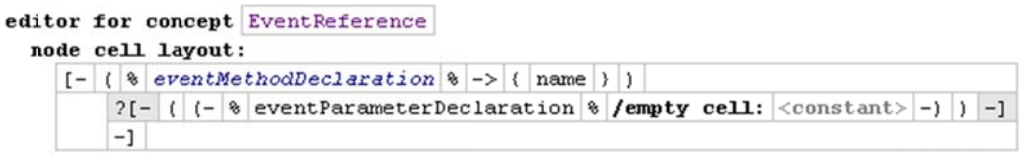

Рис. 10. Использование ячейки коллекции дочерних узлов в редакторе концепта *EventReference*

Так же, как и коллекция ячеек, ячейка коллекции дочерних узлов может иметь горизонтальное, вертикальное или горизонтальное с переносами размещение вложенных узлов. Ячейка коллекции дочерних узлов поддерживает добавление и удаление детей заданной роли. Также для этой ячейке может быть определен разделитель, который будет автоматически вставляться между ячейками дочерних узлов;

- *ячейка метасвойства* – ячейка, отображающая метасвойство концепта;
- *условная ячейка* позволяет скрывать вложенную в нее ячейку, если некоторое условие не выполняется. Условные ячейки удобны для сокрытия опциональных конструкций языка. Например, нет необходимости показывать скобки вокруг списка параметров у события на переходе, если у события нет параметров (рис. 11);

```
initial state{S1} {
  on e1 do {<no statements>}

  on e2(p1) do {<no statements>}
}
```

Рис. 11. Пример событий с параметрами и без

- *ячейка компонента* используется для декомпозиции редакторов концептов. Если редакторы нескольких концептов содержат один и тот же фрагмент, то этот фрагмент может быть вынесен в компонент редактора (рис. 12), а в редакторах концептов достаточно использовать ячейку компонента со ссылкой на этот компонент (рис. 13).

```
editor component AbstractState_component
  applicable concept:
      AbstractState

  component cell layout:
      [-
      ? initial {{ alias }} { { name } } {
        (- % stateElement % /empty cell: <constant> -)
      }
      -]
```

Рис. 12. Компонент редактора общий для концептов *State* и *FinalState*

```
editor for concept State
  node cell layout:
    # AbstractState_component #
```

Рис. 13. Использование ячейки компонента в редакторе концепта *State*

Существуют также и другие типы ячеек.

2.5.2. Стили ячеек

Каждая ячейка имеет некоторые настройки поведения и представления. Например, цвет текста, стиль шрифта, можно ли ячейку выделять в редакторе и т. д. Эти свойства задаются в виде таблицы стилей ячейки. Таблица стилей может быть либо встроенной (описываться вместе с ячейкой в редакторе концепта), либо может быть определена отдельно и использована в нескольких ячейках.

Хорошей практикой является определять несколько таблиц стилей для различных целей. Например, в языке *baseLanguage* определены таблицы стилей для ключевых слов – применяются для тех константных ячеек в

редакторах языка *baseLanguage*, которые соответствуют ключевым словам в языке *Java*; статических полей – применяются для определений статических полей и ссылок на статические поля; полей объектов; числовых литералов; строковых литералов и т. д. Язык *stateMachine* является расширением языка *baseLanguage*, поэтому в языке *stateMachine* используются таблицы стилей, объявленные в языке *baseLanguage*. Это позволяет сохранить стилистическое единообразие в коде, написанном на смеси этих языков.

Таблицы стилей ячеек задаются в виде подобном заданию таблиц стилей в языке *CSS* [104]. На рис. 14 представлен фрагмент набора классов стилей языка *baseLanguage*.

```
stylesheet BaseLanguageStyle {
  style KeyWord {
    text-foreground-color : darkBlue
    font-style : bold
  }

  style Field {
    text-foreground-color : darkMagenta
    font-style : bold
  }

  style Comment {
    text-foreground-color : gray
    font-style : bold
  }
```

Рис. 14. Классы стилей в языке *baseLanguage*

2.5.3. Редакторы концептов языка *stateMachine*

Ниже для конструкций языка *stateMachine* приведены примеры редакторов (рис. 15 – рис. 19).

```
state machine for A1 {
  <listeners>

  initial state(S1) {
    on e1 do {<no statements>} transit to (S2)
  }

  final state(S2) { }
}
```

Рис. 15. Редактор автомата (концепт *StateMachine*)

```
initial state(S1) {
  initial state(S1_1) {
    on e2 do {<no statements>} transit to (S1_2)
  }

  state(S1_2) {
    on e2 do {<no statements>} transit to (S1_1)
  }

  on e1 do {<no statements>} transit to (S2)
}
```

Рис. 16. Редактор составного состояния с вложенными простыми состояниями (концепт *State*)

```
final state(S2) { }
```

Рис. 17. Редактор конечного состояния (концепт *FinalState*)

```
а)  on e1 do {<no statements>} transit to (S2)

б)  on e2[this.x1] do {<no statements>}

в)  on e2[!(this.x1)] do {
      this.z1();
    } transit to (S3)
```

Рис. 18. Редактор перехода а) без условия, без действия, со сменой состояния; б) с условием $x1$, без действия, без смены состояния; в) с условием $\neg x1$, с действием $z1$, со сменой состояния (концепт *Transition*)

```
enter do {
  this.z1();
}

exit do {
  this.z2();
}
```

Рис. 19. Редактор действий по входу в состояние и по выходу из состояния (концепты *EnterAction* и *ExitAction*)

2.6. Система типов языка *stateMachine*

Для описания системы типов языка [100] в среде *MPS* применяется метаязык *typesystem* [105]. Этот метаязык позволяет задавать правила вывода типов для конструкций пользовательского языка и определять правила приводимости типов. Среда *MPS*, используя эти правила, автоматически выводит типы для кода, написанного на пользовательском языке, выявляет ошибки приведения типов и сообщает об этих ошибках пользователю «на лету» – в процессе написания кода.

Система типов языка *stateMachine* является расширением системы типов языка *baseLanguage*. Поэтому для того чтобы система проверки типов среды *MPS* проверяла, например, тип выражения условия на переходе, достаточно задать правило, требующее приводимости типа этого выражения к типу *boolean* языка *baseLanguage*.

Среда *MPS* позволяет использовать в качестве типов экземпляры любых концептов. В языке *baseLanguage* для типов применяется иерархия концептов, базовым концептом которой является концепт *Type*. С узлами АСД, для концептов которых определены правила вывода типов, связывается узел типа, который может быть использован разработчиком языка, например, в процессе генерации целевого кода или при выполнении рефакторинга.

2.6.1. Правила вывода типов

Основной конструкцией языка *typesystem* является *правило вывода* (*Inference Rule*), которое состоит из условия, определяющего применимость правила к конкретному узлу, и тела, представляющего собой набор операторов, выполняемых в момент применения правила к узлу АСД.

Условие может быть задано одним из двух способов: либо в виде ссылки на концепт, тогда правило применимо ко всем узлам этого концепта и его наследников, либо в виде шаблона, тогда правило применимо ко всем узлам, соответствующим шаблону. Узел соответствует шаблону, если имеет такие же свойства и ссылки, как у шаблона, а дочерние узлы соответствуют дочерним узлам шаблона. Шаблон также в качестве свойств, ссылок и дочерних узлов может иметь переменные, которые соответствуют любому значению.

2.6.2. Ти́повые уравнения и неравенства

Для задания отношений между типами конструкций языка в теле правила вывода применяются операторы *ти́повых уравнений* и *ти́повых неравенств*. Подсистема вывода типов среды *MPS* на основе типовых уравнений и неравенств, заданных разработчиком языка, строит для АСД программы систему типовых уравнений с граничным условиями [105]. Решая эту систему уравнений, указанная подсистема выводит тип для узлов АСД.

Неразрешимости системы типовых уравнений свидетельствует о наличии в программе ошибки приведения типов. Среда *MPS* обнаруживает такие ошибки и сообщает о них пользователю. Например, на рис. 20 целочисленная константа использована в качестве условия на переходе. Однако семантика языка *stateMachine* допускает в качестве условий на переходах лишь выражения логического типа.

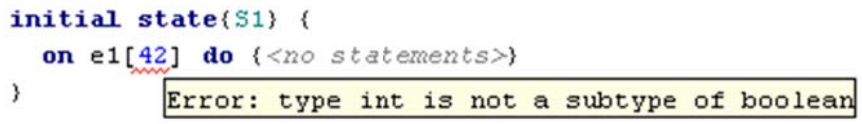

Рис. 20. Сообщение об ошибке вывода типа для выражения условия на переходе

Требование того, что выражение для условия на переходе должно иметь логический тип задается правилом вывода, приведенным на рис. 21. Это правило вывода является частью системы типов языка *stateMachine*. Для целочисленной константы в языке *baseLanguage* задано правило вывода, представленное на рис. 22.

```
rule typeof_GuardedTransitionTrigger {
  applicable for concept = GuardedTransitionTrigger as guard
  overrides false

  do {
    typeof(guard.guardCondition) :<=: <boolean>;
  }
}
```

Рис. 21. Правило вывода для выражения условия на переходе

```
rule typeOf_IntegerLiteral {
  applicable for concept = IntegerLiteral as integerLiteral
  overrides false

  do {
    typeof(integerLiteral) :==: <int>;
  }
}
```

Рис. 22. Правило вывода для целочисленной константы

На приведенных выше рисунках использованы следующие конструкции языка *typesystem*:

- «*a* :<=: *b*» задает типовое неравенство – тип *b* приводим к типу *a*;
- «*a* :==: *b*» задает типовое уравнение – тип *a* равен типу *b*. Так как в качестве типов в среде *MPS* используются узлы АСД, равенство типов

означает их структурное совпадение: узлы являются экземплярами одного и того же концепта, значения их свойств, ссылок и дочерних узлов совпадают;

- «typeof(*expr*)» – ссылка на переменную типа, связанного с узлом *expr*. Типы узлов на момент применения правил вывода еще не известны, поэтому код в этих правилах оперирует не типам узлов, а переменными типов узлов. Решая систему типовых уравнений, подсистема вывода типов вычисляет значения этих переменных и ассоциирует их с узлами АСД.

Тела правил вывода могут содержать не только операторы для задания типовых уравнений и неравенств, но произвольный код на языке *baseLanguage*. Например, в языке *stateMachine* конструкция события на переходе является ссылкой на объявление события в автоматном классе. При этом события могут иметь параметры, типы которых совпадают с типами параметров в объявлении события. С параметром события на переходе может быть связано фильтрующее значение. Тип выражения, фильтрующего значения, должен приводиться к типу параметра, с которым это выражение связано. Указанные требования реализованы в языке *stateMachine* правилом вывода для концепта *EventParameterDeclaration* (рис. 23).

```
rule typeof_EventParameterDeclaration {
  applicable for concept = EventParameterDeclaration as event
  overrides false

  do {
    node<ParameterDeclaration> d = event.getAssociatedParameter();
    if (d != null) {
      typeof(event) :==: d.type;
    } else {
      typeof(event) :==: <void>;
    }
    if (event.matchValue != null) {
      typeof(event.matchValue) :<=: typeof(event);
    }
  }
}
```

Рис. 23. Правило вывода для концепта *EventParameterDeclaration*

2.6.3. Правила подтиповости

Для того чтобы подсистема вывода типов могла найти решение системы типовых уравнений с граничными условиями, ей требуется информация о том, какие типы являются подтипами других. Для задания *отношения подтиповости* в языке *typesystem* применяются *правила подтиповости*. Эти правила позволяют задать функцию, которая для данного типа возвращает его прямые надтипы.

Правило подтиповости состоит из условия, которое может быть ссылкой на концепт или шаблоном, и тела, которое представляет собой набор операторов, вычисляющих узел или список узлов, являющихся прямыми надтипами для данного узла. Когда подсистема вывода типов определяет, является ли тип *a* подтипом *b*, она применяет подходящие правила подтиповости к *a*. К полученным типам она снова применяет правила подтиповости, затем снова и т. д., транзитивно замыкая отношение подтиповости. Если среди всех полученных таким образом типов найдется тип *b*, то указанная подсистема считает, что тип *a* является подтипом *b*.

2.7. Генератор кода для языка *stateMachine*

Генерация кода в среде *MPS* происходит путем последовательной трансформации *моделей-в-модели*. Правила перевода конструкций входного языка в конструкции выходного языка в среде *MPS* задаются при помощи проблемно-ориентированного метаязыка *generator*. Процесс генерации моделей-в-модели может состоять из множества шагов, на которых происходит понижение уровня абстракции кода путем замены конструкций промежуточных входных моделей на конструкции промежуточных выходных моделей. Этот процесс продолжается до тех пор, пока промежуточная выходная модель, полученная на очередном шаге, не будет состоять исключительно из конструкций, для которых правила трансформации не определены.

Например, для большинства конструкций языка *baseLanguage*, который является *MPS*-представлением языка *Java*, операционная семантика уже определена компилятором *Java*. Для генерации кода на языке *baseLanguage* используется простая трансформация *моделей-в-текст* на языке *Java*. Поэтому для языка *stateMachine* язык *baseLanguage* используется, как конечный выходной язык пошаговой трансформации моделей.

Преобразование моделей описывается посредством задания шаблонов генерации. Шаблоны записываются на выходном языке и редактируются теми же редакторами, что и весь остальной код на этом языке. Поэтому в редакторах шаблонов генерации доступен тот же уровень инструментальной поддержки: подсветка синтаксиса и ошибок, автоматическое дополнение кода и т. д.

2.7.1. Правила генерации

В процессе генерации кода по входной модели среда *MPS* «собирает» правила генерации из всех языков, на которых эта модель написана.

Последовательно применяя эти правила к модели и ее узлам, среда *MPS* вычисляет целевую модель.

Каждое правило генерации состоит из *предпосылки* и *следствия*. Предпосылка определяет применимость данного правила к входной модели или узлу АСД. Следствие описывает характер трансформации, которая должна быть применена к входной модели. В метаязыке *generator* существует пять видов правил генерации:

1. *Правило создания корневых узлов* (*conditional root rules*) – создание корневых узлов в выходной модели, которым не соответствуют узлы входной модели. Предпосылка такого правила – *выражение условия применимости* – логическое выражение, при истинном значении которого должно применяться данное правило. Следствие – шаблон, по которому генерируется корневой узел в выходной модели.
2. *Правило отображения корневых узлов* (*root mapping rules*) – трансформация узлов входной модели в корневые узлы выходной модели. Предпосылка – ссылка на концепт, к экземплярам которого применимо данное правило, и выражение условия применимости. Следствие – шаблон преобразования узла входной модели в корневой узел выходной модели.
3. *Правило редукции* (*reduction rules*) – трансформация узлов входной модели в некорневые узлы выходной модели. Предпосылка и следствие такие же, как и у правила отображения корневых узлов.
4. *Правило прошивания* (*weaving rules*) – трансформация узлов входной модели в некорневые узлы выходной модели и вставка полученных узлов в произвольные места выходной модели. В отличие от правил редукции, следствия правил прошивания содержат также выражения, вычисляющие узлы целевой модели, в которые полученные после трансформации узлы необходимо вставить в качестве дочерних.

5. *Правило исключения корневых узлов* (*abandon root rules*) – удаление корневых узлов входной модели из выходной модели. Предпосылка – ссылка на концепт, экземпляры которого должны быть удалены из выходной модели, и выражение условия применимости.

Узлы входной модели, для которых не было обнаружено правил преобразования, копируются в выходную модель без изменений.

2.7.2. Макросы языка *generator*

Шаблон генерации может быть параметризирован посредством макросов языка *generator*. Эти макросы представляются собой выражения, связанные с узлами, ссылками узлов и свойствами узлов шаблона. В процессе генерации среда *MPS* формирует каждый узел выходной модели путем копирования узла шаблона и замены всех, помеченных макросами, вложенных в него узлов, значений ссылок и свойств, на значения, вычисляемые выражениями макросов.

В языке генераторов существует три разновидности макросов:

1. *Макросы узлов* (*node macro*) – вычисляют вложенные узлы.
2. *Макросы ссылок* (*reference macro*) – вычисляют целевые узлы для ссылок.
3. *Макросы свойств* (*property macro*) – вычисляют значения свойств.

2.7.3. Трансформация конструкций языка *stateMachine*

Язык *stateMachine* расширяет язык *baseLanguage* конструкцией *событие* (*EventMethodDeclaration*) – методом, реализация которого находится в автомате и зависит от текущего состояния автомата. В процессе генерации кода конструкции *EventMethodDeclaration* заменяются на обычные методы языка *baseLanguage* (рис. 24).

```
concept    EventMethodDeclaration  --> reduce_EventMethodDeclaration
inheritors false
condition  <always>
```

Рис. 24. Правило редукции для концепта *EventMethodDeclaration*

В класс, с которым связан автомат, *прошивается* объявление поля, предназначенного для хранения текущего состояния, и перечисление (*enum*) для всех состояний автомата (рис. 25).

```
weaving rules:
  concept    StateMachine                                                    -->
  inheritors false
  condition  (node, genContext, operationContext)->boolean {
               StateMachineGenerator.supports(node);
             }
    weave_StateMachine_fields
    context : (node, genContext, operationContext)->node< > {
                genContext.get copied output for (node.classConcept);
              }
```

Рис. 25. Правило прошивания поля для хранения текущего состояния и перечисления для множества состояний автомата

В теле метода, генерируемого из объявления события автомата, происходит ветвление по значению переменной автомата (рис. 26). Для каждого перехода по данному событию генерируется оператор *if* (рис. 27), условием которого является условие на переходе, а телом – действия по выходу из текущего состояния, действие на переходе, изменение значения поля состояния, и действия по входу в целевое состояние.

```
public void $[eventMethodName]($COPY_SRCL$[int eventParameter])
    throws $COPY_SRCL$[Exception] {
  $INCLUDE$ [// Create event context]
  $INCLUDE$ [// Notify: event processing started]
  switch (this.->$[myState]) {
    $LOOP$[case ->$[State].->$[stateName] :
             $COPY_SRCL$[// Transitions]
             $INCLUDE$ [// Notify: event skipped]
             break;]
    default :
      <no statements>
  }
  $INCLUDE$ [// Notify: event processing finished]
}
```

Рис. 26. Шаблон генерации для концепта *EventMethodDeclaration*

```
$INCLUDE$ [// Create transition match parameters]
$INCLUDE$ [// Notify: test transition]
if ($COPY_SRC$["guard condition"]) {
  $INCLUDE$ [// Notify: found transition]
  $COPY_SRC$[// Transition action]
  break;
}
```

Рис. 27. Шаблон генерации для концепта *Transition*

В класс, с которым связан автомат, для каждого действия по входу и по выходу из состояния *прошивается* метод (рис. 28). Этот метод вызывается, когда должно быть выполнено соответствующее действие.

```
private void $[state_onEnterOrExit]() {
  $COPY_SRCL$[// on enter/exit action]
}
```

Рис. 28. Шаблон генерации для концепта *BaseStateAction*

Для того чтобы автомат начинал работу в начальном состоянии, в конструкторы класса автомата прошивается переход в начальное состояние (рис. 29).

```
public SomeClass($COPY_SRCL$[string s]) throws $COPY_SRCL$[Exception] {
  $COPY_SRCL$[// Statements of class constructor]
  $LOOP$[this.smListeners.add($COPY_SRC$[null]);]
  $LOOP$[this.->$[onEnter]();]
  this.->$[myState] = ->$[State].->$[stateName];
}
```

Рис. 29. Шаблон генерации для конструктора класса автомата

2.8. Автоматическое построение диаграммы переходов

Ранее в работе отмечалось, что многие программисты предпочитают набирать код программы в текстовом виде. Поэтому разработанный автором язык *stateMachine* является текстовым языком, реализованным в среде *MPS*. Однако диаграммы переходов позволяют представлять автоматы более наглядно. Поэтому *предлагается по текстовому описанию автомата автоматически строить диаграмму переходов*.

Далее приведен листинг кода для автомата, управляющего лифтом:

```
state machine for Elevator {
  initial state{Stopped} {
    on call(toFloor, from) do {
      this.tasks.addTask(toFloor, from);
    } transit to {ClosingDoors}
  }

  state{ClosingDoors} {
    enter do {
      this.doorsEngine.close();
    }
    on call(toFloor, from)[toFloor != this.floor] do {
      this.tasks.addTask(toFloor, from);
    }
    on call(toFloor, from)[toFloor == this.floor] do {
    } transit to {OpeningDoors}
    on openDoors do {} transit to {OpeningDoors}
    on doorsClosed[this.floor > this.tasks.target()] do {
    } transit to {MovingDown}
    on doorsClosed[this.floor < this.tasks.target()] do {
    } transit to {MovingUp}
  }
```

```
state{OpeningDoors} {
  enter do {
    this.doorsEngine.open();
  }
  on call(toFloor, from) do {
    this.tasks.addTask(toFloor, from);
  }
  on doorsOpened do {} transit to {LoadingPeople}
}

state{LoadingPeople} {
  enter do {
    this.tasks.removeTasks(this.floor);
    this.loadingTimer.start();
  }
  on call(toFloor, from)[toFloor != this.floor] do {
    this.tasks.addTask(toFloor, from);
  }
  on loadingTimeout[!(this.tasks.hasTasks())] do {
  } transit to {Stopped}
  on loadingTimeout[this.tasks.hasTasks()] do {
  } transit to {ClosingDoors}
}

state{MovingUp} {
  enter do {
    this.elevatorEngine.turnUp();
  }
  on reached(floor)[this.tasks.shouldStopAscending(floor)] do {
    this.elevatorEngine.stop();
    this.floor = floor;
  } transit to {OpeningDoors}
  on call(toFloor, from) do {
    this.tasks.addTask(toFloor, from);
  }
}

state{MovingDown} {
  enter do {
    this.elevatorEngine.turnDown();
```

```
      }
      on reached(floor)[this.tasks.shouldStopDescending(floor)] do {
        this.elevatorEngine.stop();
        this.floor = floor;
      } transit to {OpeningDoors}
      on call(toFloor, from) do {
        this.tasks.addTask(toFloor, from);
      }
    }
  }
```

Средства интеграции языка *stateMachine* со средой *MPS* для автомата, код которого приведен выше, автоматически построили диаграмму переходов, приведенную на рис. 30.

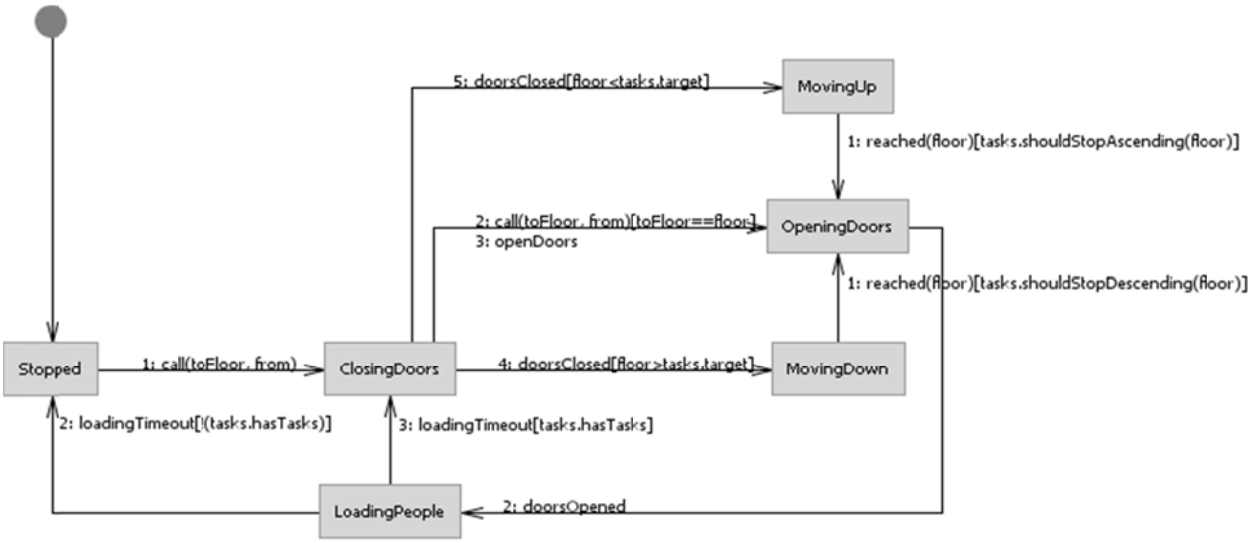

Рис. 30. Диаграмма переходов для автомата *Elevator*

Предлагаемый подход отличается от традиционного автоматного программирования, для которого первичным артефактом является диаграмма переходов, а код, реализующий ее, пишется по ней вручную или генерируется автоматически.

Выводы по главе 2

1. Описаны средства и процесс создания языка в среде языково-ориентированного программирования *MPS*.

2. Описан разработанный автором язык автоматного программирования в среде *MPS*.
3. Описан шаблон метапрограммирования *OptionList*.
4. Предложен и реализован подход к автоматной разработке, при котором код автомата пишется в виде текста, а диаграмма переходов восстанавливается из кода автоматически.

ГЛАВА 3. ВАЛИДАЦИЯ АВТОМАТНЫХ МОДЕЛЕЙ

Для уменьшения числа ошибок при разработке автоматных программ необходимо проводить валидацию разрабатываемых автоматных моделей. Валидацией автоматных моделей называется проверка свойств автоматов, не зависящих от пользовательской спецификации. В этой главе рассматриваются методы валидации автоматов в нотации, предложенной в работе [106], с двухместными предикатами в условиях на переходах, и реализация этих методов в среде языково-ориентированного программирования.

В настоящей работе рассматривается валидация следующих свойств автоматных моделей:

- полнота условий на переходах;
- непротиворечивость условий на переходах;
- достижимость любого состояния из начального состояния;
- достижимость из любого состояния конечного состояния.

Задача автоматической валидации автоматных моделей с одноместными предикатами в условиях на переходах была решена автором ранее [80]. Для дальнейшего увеличения выразительной мощности автоматных моделей необходимо расширить класс формул для выражения условий на переходах двуместными предикатами сравнения входных переменных. Практическая ценность такого расширения состоит в том, что оно позволяет проверять корректность автоматных моделей, содержащих в условиях на переходах сравнения переменных друг с другом. Предыдущий результат в этой области, описанный автором в работе [41], позволял проверять условия на переходах, содержащие лишь одноместные предикаты, что с практической точки зрения позволяло сравнивать переменную лишь с константой.

Достижимость любого состояния из начального и достижимость конечного состояния из любого состояния может быть проверена модификацией алгоритма поиска в глубину [107].

3.1. Формальная модель автомата

Для проведения валидации автоматных моделей необходимо зафиксировать формальную модель автомата [108].

Автоматом называется $A = (Q, \Sigma, d, q_0, F, X)$, где

Q – множество состояний;

Σ – множество событий;

q_0 – начальное состояние, $q_0 \in Q$;

F – множество конечных состояний, $F \subseteq Q$;

d – функция переходов;

X – набор входных переменных с множеством значений K.

Множество конечных состояний F – набор таких состояний $f \epsilon F$, при переходе в любое из которых работа автомата завершается. Подобное понимание конечного состояния, характерное для структурных автоматов, отличает рассматриваемую модель от принятой в теории формальных грамматик, где понятие *конечное состояние* – синоним *допускающего состояния*.

3.1.1. Функция переходов

Функция переходов автомата определяется как

$$d = d(q, \sigma, v_1, \dots, v_n): (Q \setminus F) \times \Sigma \times K^n \to Q,$$

где $q \in Q \setminus F$, $\sigma \in \Sigma$, $v_i \in K$. Таким образом, функция переходов для данного состояния q, пришедшего события σ и набора значений переменных однозначно определяет новое состояние, в которое перейдет автомат. Это означает, что рассматриваются только детерминированные автоматы. На конечных состояниях функция перехода не определена. При переходе

автомата в конечное состояние его выполнение завершается, и дальнейшие переходы не осуществляются.

Переменная v_i может соответствовать переменной в программе, быть членом класса объекта управления или параметром события σ. Тип переменной v_i в программе может логическим, целочисленным, вещественным или перечислимым. В рамках теоретической части работы значения переменных v_i принадлежат множеству K, для элементов которых определена операция сравнения.

Для формального описания валидируемых свойств удобно ввести обозначение $s_q^\sigma = s_q^\sigma(v_1, \dots, v_n)$ для функции переходов из состояния q при возникновении события σ:

$$s_q^\sigma(v_1, \dots, v_n) = d(q, \sigma, v_1, \dots, v_n).$$

В общем случае функция s_q^σ может быть произвольной. В данной работе рассматриваются лишь такие функции, которые состоят из набора пар

$$s_q^\sigma = \{(\varphi_i, q_i), i = 1..k\},$$

где φ_i – булева функция, соответствующая условию на переходе из состояния q в состояние q_i по событию σ. В качестве аргументов этой функции допускаются произвольные одноместные предикаты и предикаты сравнения переменных $v_1, \dots, v_k$ друг с другом [109].

Если на некотором наборе значений переменных $\{v_1, \dots, v_k\}$ предикат φ_i принимает истинное значение, то значение функции s_q^σ на этом наборе определяется как равное q_i:

$$\varphi_i(v_1, \dots, v_k) = 1 \Leftrightarrow s_q^\sigma(v_1, \dots, v_k) = q_i.$$

Такое задание функции s_q^σ корректно только, если выполняются условия:

$$\forall \{v_1, \dots, v_n\}\ \exists \varphi_i(v_1, \dots, v_n) = 1,$$

$$\forall \{v_1, \dots, v_n\}, i \neq j, \varphi_i(v_1, \dots, v_n) = 1 \Rightarrow \varphi_j(v_1, \dots, v_n) = 0.$$

Первое условие соответствует свойству полноты и означает существование

перехода для любого набора значений входных переменных. Второе условие соответствует свойству непротиворечивости и означает единственность перехода для любого набора входных переменных.

3.1.2. Достижимость состояний

В автомате *определен переход* qq^* из состояния q в состояние q^*, если существует такое событие σ, что функция перехода s_q^σ содержит для некоторого условия φ пару (φ, q^*). При этом к условию φ не предъявляется требование выполнимости для какого-либо набора $v_1, \dots, v_n$. Таким образом, при определении наличия перехода из состояния q в состояние q^* условия на переходах фактически игнорируются.

Состояние q^* *достижимо из состояния* q ($q \rightsquigarrow q^*$), если $q = q^*$ или существуют такие состояния $q = q^0, q^1, \dots, q^n = q^*$, для которых определены переходы $q^{i-1}q^i$ для $\forall i = 1..n$.

3.1.3. Граф изоморфный автомату

Для применения результатов теории графов для валидации автоматов каждому автомату $A = (Q, \Sigma, d, q_0, F, X)$ необходимо сопоставить изоморфный гиперграф [107] состояний и переходов $G = (V, E)$. Вершинами такого графа будут состояния автомата $V = Q$. Направленные ребра графа будут соединять одно состояние с другим, если определен переход из первого во второе. При этом на ребрах графа записываются событие σ, при котором совершается данный переход, и выражение φ, соответствующее условию на переходе. Ребро $e = (q, q^*, \sigma, \varphi)$ принадлежит множеству E, если функция перехода s_q^σ для состояния q и события σ содержит (φ, q^*). Так как для одной пары состояний может быть определено несколько переходов между ними, построенная конструкция будет гиперграфом.

Из изоморфизма гиперграфа G автомату A следует изоморфизм между реберными путями в графе путями из переходов в автомате. Таким образом,

запись $q \rightsquigarrow q^*$ одновременно означает достижимость состояния q^* из состояния q и достижимость вершины q^* из вершины q.

3.1.4. Валидируемые свойства

Валидация автомата с учетом данных выше определений состоит из проверки следующих свойств:

1. Полнота условий на переходах – существование выполнимого выражения для любой функции перехода s_q^σ и при любых возможных значениях параметров функции $v_1, \dots, v_n$.
2. Непротиворечивость условий на переходах – единственность выполнимого выражения для любой функции перехода s_q^σ при всех возможных значениях параметров функции $v_1, \dots, v_n$.
3. Достижимость любого состояния q из начального состояния q_0

$$\forall q\colon q_0 \rightsquigarrow q.$$

4. Достижимость конечного состояния из любого состояния автомата

$$\forall q\ \exists f \in F\colon q \rightsquigarrow f.$$

Не все перечисленные свойства являются обязательными для корректной работы автомата. Свойства 1 и 2 вытекают из данного определения автомата. Операционная семантика языка автоматного программирования, описанного в главе 2, частично снимает необходимость выполнения свойства 1 в силу того, что отсутствие явного перехода по некоторому событию и некоторому набору входных переменных трактуется как наличие петли без выходных воздействий. Необходимость свойства 2 умаляется существованием фиксированного порядка, в котором происходит перебор переходов при обработке события. Таким образом, данная операционная семантика для функции переходов s_q^σ заменяет явные условия на переходах $\{(\varphi_i, q_i), i = 1..k\}$, на неявные $\{(\varphi_i', q_i), i = 1..k\} \cup \{(\varphi_{k+1}', q)\}$, где $\varphi_i' = \varphi_i \wedge \left(\neg \bigvee_{j=1..k-1} \varphi_k\right)$. Тем не менее, проверка свойств полноты и

непротиворечивости явных условий на переходах позволяет выявлять ошибки программирования в процессе разработки.

Невыполнение свойства 3 на практике, как правило, указывает на ошибки в проектировании автомата и, соответственно, программы, основанной на нем. Нарушение этого свойства означает, что в программе существует код, который не будет выполнен ни при каких условиях.

Невыполнение утверждения 4 для автомата означает, что в нем существуют состояния, при попадании в которые программа никогда не завершится. Такой автомат может быть создан и иметь практическое применение. Более того, для описания системы, работа которой не предполагает явного завершения, может быть создан автомат, не имеющий конечных состояний вообще. Если же автоматная программа реализует конечный по времени процесс, то наличие в нем состояния, из которого программа не достигает конечного, явным образом указывает на ошибку.

Таким образом, проверка перечисленных выше свойств весьма желательна и имеет большую практическую ценность.

3.2. Полнота и непротиворечивость условий на переходах

Проверка свойств полноты и непротиворечивости условий на переходах автомата $A = (Q, \Sigma, d, q_0, F, X)$, с учетом изложенного определения, состоит в проверке истинности двух утверждений:

- $\forall q \in Q, \forall \sigma \in \Sigma: \bigvee \varphi_i \equiv 1$, где φ_i – условия на переходах для функции s_q^σ, для свойства полноты;
- $\forall q \in Q, \forall \sigma \in \Sigma: \forall i \neq j: \varphi_i \wedge \varphi_j \equiv 0$, где φ_i и φ_j – условия на переходах для функции s_q^σ, для свойства непротиворечивости.

Таким образом, задача сводится к проверке либо свойства тавтологичности, либо свойства выполнимости набора выражений.

3.2.1. Булевы формулы

Ранее в настоящей работе указывалось, что условия на переходах φ определены, как булевы формулы, в качестве аргументов которых выступают произвольные одноместные предикаты и предикаты сравнения входных переменных автомата. В работе [41] определение булевой формулы вводится следующим образом:

Пропозициональная переменная – элементарное логическое высказывание, которое принимает значение из множества $\{0, 1\}$.

Пропозициональной формулой является:

- всякая пропозициональная переменная;
- если A – пропозициональная формула, то $\neg A$ – пропозициональная формула;
- если A и B – пропозициональные формулы, то $(A \wedge B)$, $(A \vee B)$, $(A \to B)$ – пропозициональные формулы.

Пропозициональная формула – это высказывание, которое может быть истинным или ложным. Пусть пропозициональная формула f содержит n (пропозициональных) переменных, тогда подстановка на их место конкретных значений 0 или 1 («истина» или «ложь»), выдаст истинное или ложное значение для формулы f. Таким образом, f задает отображение φ: $\{0, 1\}^n \to \{0,1\}$, которое называется булевой функцией.

Итак, булева функция задается пропозициональной формулой, значение которой определяется значениями пропозициональных переменных, содержащихся в формуле, и операторами, связывающими эти переменные. В настоящей работе в качестве пропозициональных переменных выступают одноместные предикаты и предикаты сравнения входных переменных автомата.

3.2.2. Предикаты

Входные переменные автомата в рассматриваемой модели могут иметь логический, целочисленный, вещественный или перечислимый тип. В качестве пропозициональных переменных, допустимых в функциях условий на переходах, рассматриваются логические переменные и операторы сравнения переменных с константой (одноместные предикаты) и друг с другом (двухместные предикаты): операторы строгого и нестрогого неравенства ($<$, $\leq$, $>$, $\geq$), оператор равенства ($=$) и оператор неравенства ($\neq$). Последние два оператора сводятся к четырем предыдущим следующим образом:

$$x = y \equiv (x \geq y) \wedge (x \leq y),$$
$$x \neq y \equiv (x < y) \vee (x > y).$$

Таким образом, в дальнейшем достаточно рассмотреть только предикаты с первыми четырьмя операторами.

3.2.3. Проверка тавтологичности формулы

Проверка невыполнимости булевой формулы φ аналогична проверке тавтологичности булевой формулы $\neg\varphi$, поэтому для проверки свойств полноты и непротиворечивости условий на переходах достаточно предложить метод проверки тавтологичности булевой формулы φ указанного класса. Такую проверку предлагается проводить следующим образом:

1. Формула, для которой необходимо доказать тавтологичность представляется в виде *секвенции* $\vdash \varphi$ [109].
2. При помощи исчисления секвенций формула сводится к набору *элементарных секвенций* (*аксиом исчисления секвенций*) – таких секвенций, у которых и в левой, и в правой части существуют только пропозициональные переменные: логические входные переменные и операции сравнения входных переменных.

3. Если ни одна из полученных элементарных секвенций не является *контрпримером*, то, в соответствии с теорией исчисления секвенций, исходная формула является тавтологией. Контрпримером называется такая секвенция, для которой существует набор значений входных переменных, при котором все пропозициональные переменные в левой части истинны, а все пропозициональные переменные в правой части ложны.

Если в некоторую элементарную секвенцию некоторая логическая входная переменная входит и слева, и справа, то такая секвенция не является контрпримером, так как никакая логическая переменная не может принимать и истинное, и ложное значение одновременно.

Значения операций сравнения могут быть связаны друг с другом через сравниваемые входные переменные, поэтому для того, чтобы элементарная секвенция являлась контрпримером, недостаточно того, чтобы ни одна операция сравнения не входила в обе части секвенции одновременно. Для секвенций с операциями сравнения необходимо искать набор значений входных переменных, при которых все неравенства слева истинны, а все неравенства справа ложны.

Все операции сравнения в элементарной секвенции могут быть перенесены в левую часть по правилам:

$$\frac{(v_1 \geq v_2), \Gamma \vdash \Delta}{\Gamma \vdash (v_1 < v_2), \Delta}, \quad \frac{(v_1 \leq v_2), \Gamma \vdash \Delta}{\Gamma \vdash (v_1 > v_2), \Delta},$$

$$\frac{(v_1 > v_2), \Gamma \vdash \Delta}{\Gamma \vdash (v_1 \leq v_2), \Delta}, \quad \frac{(v_1 < v_2), \Gamma \vdash \Delta}{\Gamma \vdash (v_1 \geq v_2), \Delta},$$

которые вытекают из правила вывода для секвенций:

$$\frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \neg A, \Delta}$$

и свойств операторов сравнения:

$$\overline{(v_1 < v_2)} \equiv (v_1 \geq v_2),$$

$$\overline{(v_1 > v_2)} \equiv (v_1 \leq v_2).$$

Таким образом, задача поиска контрпримера может быть сведена к доказательству выполнимости системы неравенств. Аргументами неравенства может быть пара входных переменных (двухместные предикаты) или входная переменная и константа (одноместные предикаты).

3.2.4. Выполнимость системы неравенств

Если в получившейся системе неравенства вида $v_1 < v_2$ заменить на эквивалентные неравенства $v_2 > v_1$, а неравенства вида $v_1 \leq v_2$ – на неравенства $v_2 \geq v_1$, то полученная система неравенств будет задавать на множестве своих аргументов отношение частичного нестрогого порядка [110]. Иначе говоря, предикат вида $(v_1 \geq v_2)$ определяет порядок для своих аргументов v_1 и v_2. Предикат вида $v_1 > v_2 \equiv (v_1 \geq v_2) \wedge (v_1 \neq v_2)$ также упорядочивает пару v_1, v_2.

Если для какой-либо пары аргументов v_1, v_2 из заданного порядка следует, что $v_1 > v_2 \wedge v_2 > v_1$ или $v_1 > v_2 \wedge v_2 \geq v_1$, то такой порядок не корректен, а система неравенств невыполнима.

Для этого целесообразно ввести понятие *графа отношений* – ориентированного графа $G = (V, E)$, вершинами V которого является множество входных переменных и констант, использующихся в неравенствах. Множество ребер E строится следующим образом:

- каждому строгому неравенству системы $v_1 < v_2$ соответствует помеченное оператором сравнения ребро $e(v_1, v_2, <) \in E$;
- каждому нестрогому неравенству $v_1 \leq v_2$ – ребро $e(v_1, v_2, \leq) \in E$;
- каждой паре сравнимых констант c_1, c_2, являющихся аргументами в неравенствах – ребро $e(c_1, c_2, <) \in E$, если $c_1 < c_2$, и ребро $e(c_2, c_1, <)$, в противном случае – система неравенств дополняется набором естественных неравенств для констант.

Если пара аргументов v_1, v_2 связывается одновременно и строгим, и нестрогим неравенством в системе, то множество ребер будет содержать ребро только для строго неравенства.

Для того чтобы проверить, что задаваемое системой неравенств отношение частичного порядка корректно, необходимо построить транзитивное замыкание графа отношений $G^*(V, E^*)$ и проверить условие

$$\forall v_1, v_2 (e(v_1, v_2, <) \in E^* \Rightarrow e(v_2, v_1, <) \notin E^* \wedge e(v_2, v_1, \leq) \notin E^*).$$

Построение транзитивного замыкания графа G предлагается осуществлять при помощи модифицированного алгоритма Флойда–Уоршолла [107]. Граф отношений G может быть представлен в виде матрицы смежности W. Каждой строке и каждому столбцу матрицы W соответствует вершина графа G. В ячейках матрицы смежности хранятся числа соответствующие операторам отношения, которыми помечены ребра:

$$e(v_i, v_j, sign) \notin E \Leftrightarrow W_{ij} = 0,$$
$$e(v_i, v_j, \leq) \in E \Leftrightarrow W_{ij} = 1,$$
$$e(v_i, v_j, <) \in E \Leftrightarrow W_{ij} = 2.$$

Приведем алгоритм Флойда-Уоршалла, модифицированный для построения транзитивного замыкания графа отношений:

```
1. Для всех k от 1 до |V|
2.   Для всех i от 1 до |V|
3.     Для всех j от 1 до |V|
4.       Если W_ik ≠ 0 ∧ W_kj ≠ 0
5.         W_ij = max(W_ij, W_ik, W_kj)
```

В терминах графа, значения операторов на ребрах являются весовой функцией. Предложенный алгоритм находит пути наибольшего веса между всеми вершинами. При этом вес пути определяется как вес ребра с наибольшим весом в данном пути. С точки зрения отношений для всех пар вершин ищется наиболее строгое отношение, определенное для них. Таким

образом, полученная после работы алгоритма матрица W является матрицей смежности для графа G^*.

3.2.5. Полнота условий на переходах

Проверка свойства полноты условий на переходах для автомата $A = (Q, \Sigma, d, q_0, F, X)$ состоит в проверке истинности утверждения:

$$\forall q \in Q, \forall \sigma \in \Sigma: \bigvee \varphi_i \equiv 1,$$

где φ_i – условия на переходах для функции s_q^σ.

В предыдущих разделах был описан метод сведения проверки свойства тавтологичности булевой формулы указанного класса к поиску контрпримера для секвенции и предложен алгоритм такого поиска. Таким образом, проверка полноты условий на переходах сведена к поиску контрпримера для секвенции $\vdash \{\bigvee \varphi_i\}$.

3.2.6. Непротиворечивость условий на переходах

Проверка свойства непротиворечивости для автомата $A = (Q, \Sigma, d, q_0, F, X)$ состоит в проверке истинности утверждения:

$$\forall q \in Q, \forall \sigma \in \Sigma: \forall i \neq j: \varphi_i \wedge \varphi_j \equiv 0,$$

где φ_i и φ_j – условия на переходах для функции s_q^σ.

Для проверки истинности этого утверждения предлагается представить его в виде секвенции $\{\varphi_i \wedge \varphi_j\} \vdash$. Как было показано в предыдущих разделах, свойство непротиворечивости условий на переходах выполняется тогда и только тогда, когда для такой секвенции с помощью предложенного выше алгоритма невозможно найти контрпример.

3.3. Достижимость из начального состояния

Проверка свойства достижимости любого состояния из начального для автомата $A = (Q, \Sigma, d, q_0, F, X)$ состоит в проверке истинности

утверждения $\forall q \in Q: q_0 \rightsquigarrow q$, для проверки которого предлагается использовать следующий алгоритм:

1. Множество состояний, пройденных при обходе графа из начального состояния на нулевом шаге $S_0 = \emptyset$.
2. В гиперграфе G, изоморфном автомату A, из вершины q_0 выполняется обход вершин в глубину [107]. На каждом шаге обхода строится множество состояний, пройденных при обходе графа из начального состояния на i-ом шаге $S_i = S_{i-1} \cup \{q_i\}$, где q_i – вершина, пройденная на i-ом шаге обхода.
3. Если после завершения обхода за N шагов $S_N = Q$, то любое состояние автомата достижимо из начального. Иначе $Q \setminus S_N$ – множество недостижимых состояний.

Доказательство корректности решения.

Обход в глубину, запущенный для вершины q_0, достигает саму вершину q_0 и все вершины, достижимые из нее. Из этого следует верность утверждения $\forall q \in S_N: q_0 \rightsquigarrow q$. По изоморфизму $G \leftrightarrow A$ множество S_N – множество состояний, достижимых из начального, а все остальные состояния $q \in Q \setminus S_N$ недостижимы. Если множество S_N совпадает с множеством Q, то все состояния в автомате достижимы из начального состояния.

3.4. Достижимость конечного состояния

Проверка свойства достижимости конечного состояния из любого состояния автомата $A = (Q, \Sigma, d, q_0, F, X)$ состоит в проверке утверждения $\forall q \in Q\ \exists f \in F: q \rightsquigarrow f$, для проверки которого предлагается использовать модифицированный алгоритм обратного обхода в глубину графа:

1. В стек T, используемый для обратного обхода графа, помещаются все состояния из множества F. Множество состояний, посещенных при обходе графа на нулевой итерации, полагается пустым $S_0 = \emptyset$. Множество помеченных вершин также полагается пустым.

2. Если стек T пуст, то алгоритм заканчивает свою работу.
3. Если стек T не пуст на i-ой итерации, то
   a) из вершины стека извлекается состояние q_i;
   b) множество состояний, пройденных при обходе графа на i-ой итерации, полагается равным $S_i = S_{i-1} \cup \{q_i\}$;
   c) складываются в стек T и помечаются все вершины, которые не были помечены на предыдущих шагах, и из которых существует хотя бы один переход в состояние q_i.
4. Выполняется переход к шагу 2.
5. Если после N итераций множество S_N совпадает с множеством Q, то конечные состояния достижимы из всех вершин автомата, иначе конечные состояния не достижимы из вершин $q \in Q \setminus S_N$.

Конечность алгоритма [102] вытекает из того, что

- конечные состояния помещаются в стек один раз на шаге 1, и не могут быть заново помещены в стек на шаге 3, так как в автомате не существует переходов из конечных состояний;
- неконечные состояния не могут несколько раз попадать в стек на шаге 3, так как после помещения в стек состояния помечаются, а помеченные ранее состояния заново в стек не складываются;
- на каждой итерации из стека извлекается по одному состоянию. При этом каждое состояние может побывать в стеке не более одного раза. Поэтому алгоритм закончит свою работу за число итераций, не превышающее число состояний в автомате.

Корректность алгоритма доказывается аналогично доказательству корректности обратного алгоритма обхода в глубину.

3.5. Реализация в среде *MPS*

От системы валидации автоматов в среде *MPS* требовалось предоставить пользователю простой и удобный способ проверки автоматов, а

также предложить ему инструменты для исправления в автоматах ошибок валидации.

Для этого описанные методы валидации применяются в реальном времени по мере написания пользователем кода автомата. Сообщения об ошибках валидации выводятся пользователю путем выделения в редакторе цветом или подчеркиванием конструкций, содержащих ошибки. Например, при проверке достижимости конечных состояний выделяются состояния, из которых конечные состояния не достигаются (рис. 31). При наведении курсора мыши на элементы, содержащие ошибки, выводится текст сообщения об ошибке.

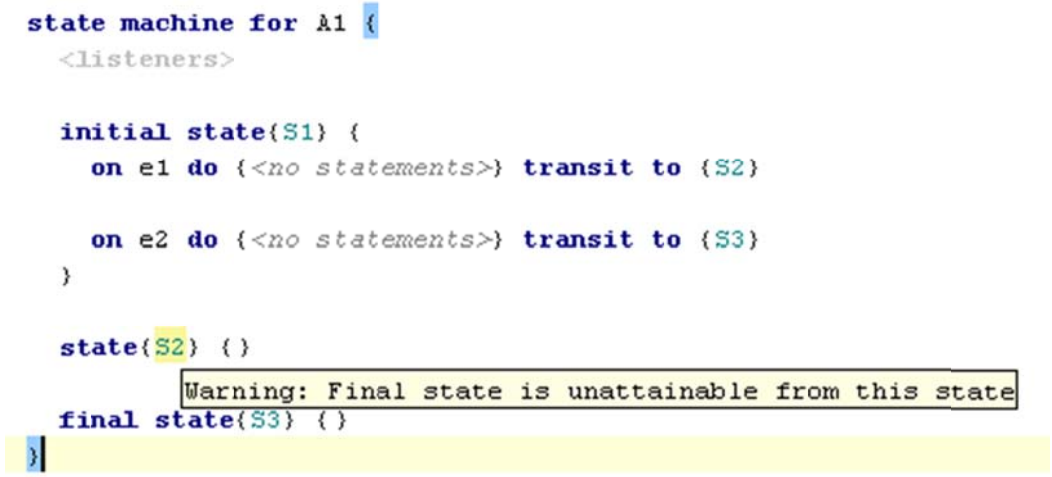

Рис. 31. Сообщение о недостижимости конечного состояния из состояния *S2*. Название состояния выделено цветом. Рядом с названием состояния текст с сообщением об ошибке

Для реализации подобных проверок в среде *MPS* применяется проблемно-ориентированный язык *typesystem*. Этот язык позволяется описывать не только систему типов для разрабатываемого языка, но и другие аспекты его семантики, например, проверку условий несинтаксической корректности. Код проверки записывается на языке *baseLanguage*. Язык *typesystem* расширяет язык *baseLanguage* конструкциями для вывода сообщений об ошибках. Такие конструкции позволяют, помимо сообщения об ошибке, указать узел *АСД*, к которому эта ошибка относится. Среда *MPS*

автоматически выделяет в редакторе узлы, с которыми связана хотя бы одна ошибка, и выводит текст сообщения об ошибке при наведении курсора мыши.

На рис. 32 показан пример правила для проверки достижимости конечных состояний из всех состояний автомата. Это правило выводит сообщения об ошибке для каждого состояния автомата, из которого недостижимо ни одно конечное состояние.

```
non type system rule validate_StateMachine_AttainableFinalState {
  applicable for concept = StateMachine as stateMachine
  overrides false

  do {
    foreach state in stateMachine.getUnattainableToFinalState() {
      warning "Final state is unattainable from this state" -> state;
    }
  }
}
```

Рис. 32. Правило на языке *typesystem*, выводящее для состояния *state* сообщение о недостижимости из него конечного состояния

С выводом сообщения об ошибке также может быть связан код действия для автоматического исправления этой ошибки. Эта возможность используется для создания средств автоматизированного исправления ошибок валидации состояний с неполными или противоречивыми исходящими переходами. Для ошибочных переходов, при наведении на них курсора, появляется меню с вариантами преобразования кода для исправления ошибок. При выборе пользователем варианта осуществляется исправление кода.

3.5.1. Реализация проверки полноты и непротиворечивости

Автоматный язык в среде *MPS* определяет условие на переходе, как узел концепта *Expression*. Этот концепт является базовым для всех выражений, существующих в языке *baseLanguage*. Поэтому в общем случае

условие на переходе не обязательно является булевой формулой из класса, для которого предложен метод валидации.

В реализации метода валидации выполняется разбор условий на переходах и проверка принадлежности этих условий к описанному классу булевых формул. Поддерживаются следующие выражения языка *baseLanguage*:

- *AndExpression* – конъюнкция;
- *OrExpression* – дизъюнкция;
- *NotExpression* – логическое отрицание;
- *ParenthesizedExpression* – выражение в скобках;
- *BooleanConstant* – булева константа;
- *LessThanExpression* – оператор «меньше»;
- *LessThanOrEqualsExpression* – оператор «меньше или равно»;
- *GreaterThanExpression* – оператор «больше»;
- *GreaterThanOrEqualsExpression* – оператор «больше или равно»;
- *EqualsExpression* – оператор «равно»;
- *NotEqualsExpression* – оператор «неравно»;
- *StaticFieldReference* – ссылка на статическое поле класса, если поле *финальное*, интерпретируется как константа, иначе – как переменная;
- *StringLiteral* – строковая константа;
- *IntegerConstant*, *LongLiteral*, *HexIntegerLiteral*, *CharConstant*, *FloatingPointConstant*, *FloatingPointFloatConstant* – числовые константы;
- *EventParameterReference* – параметр события, который при валидации интерпретируется как входная переменная;
- *FieldReferenceOperation* – поле класса, которое интерпретируется как входная переменная;
- *InstanceMethodCallOperation* – метод, который интерпретируется как входная переменная.

Если поддерево выражения для условия на переходе содержит конструкции, отличные от перечисленных выше, то пользователю будет выдано сообщение о том, что такое выражение не может быть валидировано.

Несмотря на то, что для разработки надежных систем необходимо явно проверять полноту и непротиворечивость, описанная во второй главе операционная семантика для автоматного языка программирования, позволяет корректно исполнять автоматы даже в тех случаях, когда для них не выполняются свойства полноты и непротиворечивости переходов. В случае если в некотором состоянии не определен переход для некоторой комбинации события и значений входных переменных, то событие игнорируется. Таким образом, обеспечивается полнота переходов для любых автоматов. Перебор переходов при обработке события выполняется в порядке определения этих переходов в коде автомата. Поэтому если для некоторой комбинации входных переменных выполняются условия сразу на двух переходах, то будет выбран переход, определенный в коде раньше. Это обеспечивает непротиворечивость переходов для любых автоматов. Средства автоматической валидации автоматов в среде *MPS* позволяют по желанию пользователя отключать проверку полноты и непротиворечивости переходов для автомата.

Проверка полноты и непротиворечивости переходов реализована для каждого стабильного состояния q следующим образом:

1. Исходящие переходы из состояния q и состояний, в которые состояние q вложено, группируются по событиям. Для каждого события σ, обрабатываемого в состоянии q, определяется набор связанных с ним переходов $s_q^\sigma = \{\varphi_i, i = 1..k\}$.
2. Для каждого перехода определяется, принадлежит ли формула условия на переходе классу, для которого предложен метод проверки свойств полноты и непротиворечивости. Если какая-то из формул не

принадлежит этому классу, то пользователю выводится сообщение об ошибке и валидация прекращается.

3. Для каждого события строится формула $\varphi_{\text{п}} = \vee \varphi_i$ – дизъюнкция всех условий на переходах из состояния q по событию σ, и формулы $\varphi_{\text{н}ij} = \varphi_i \wedge \varphi_j$ – попарные конъюнкции всех условий на переходах из состояния q по событию σ.
4. Для формулы $\varphi_{\text{п}}$ строится секвенция $\vdash \varphi_{\text{п}}$. Для этой секвенции осуществляется поиск контрпримера при помощи предложенного метода. Наличие хотя бы одного контрпримера указывает на то, что свойство полноты переходов не выполняется. В этом случае пользователю выводится сообщение об ошибке, связанное с каждым из переходов из состояния q по событию σ (рис. 33), а также строятся выражения для автоматизированного исправления ошибки.

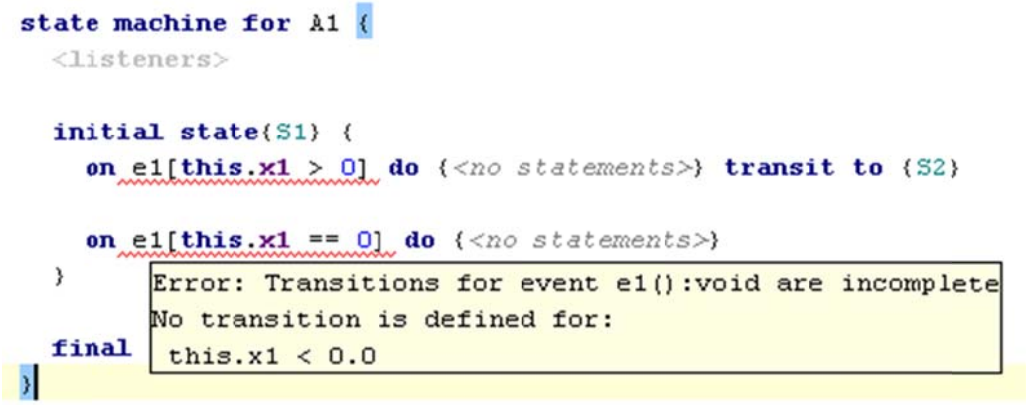

Рис. 33. Сообщение о неполноте переходов

5. Для каждой формулы $\varphi_{\text{н}ij}$ строится секвенция $\varphi_{\text{н}ij} \vdash$. Для этой секвенции выполняется поиск контрпримера при помощи предложенного метода. Наличие хотя бы одного контрпримера указывает на то, что свойство непротиворечивости переходов не выполняется. В этом случае пользователю выводится сообщение об ошибке, связанное с переходами i и j (рис. 34), а также строятся выражения для автоматизированного исправления ошибки.

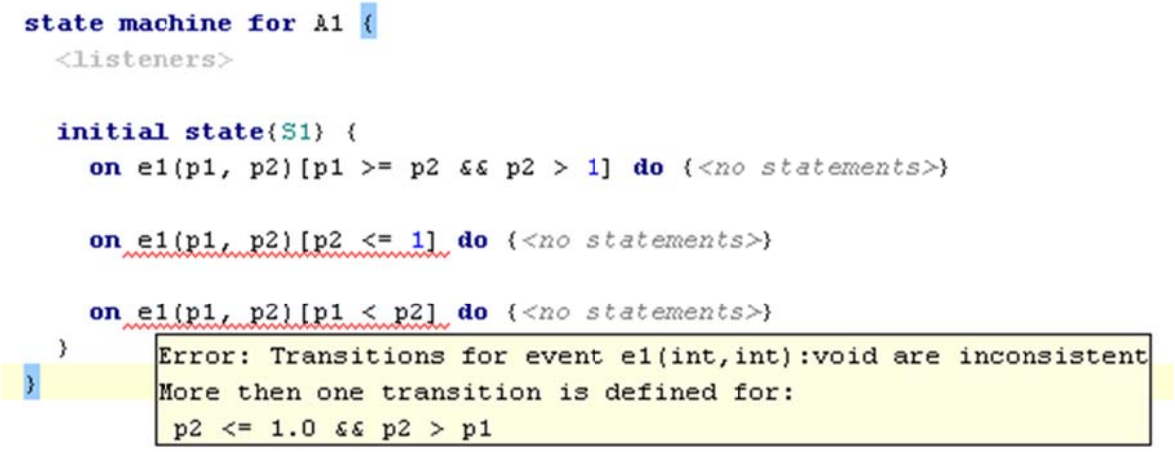

Рис. 34. Сообщение о противоречивости переходов

3.5.2. Исправление неполноты переходов

Набор контрпримеров для секвенции $\vdash \varphi_{\text{п}}$ фактически определяет множества значений входных переменных, для которых не определен переход. Если набор секвенций $\{C_l = \{\gamma_{ls}\} \vdash \{\delta_{lt}\}\}$ – множество всех контрпримеров для секвенции $\vdash \varphi_{\text{п}}$, то наборы значений входных переменных, выполняющих формулы $\psi_l = (\bigwedge \gamma_{ls}) \wedge (\bigwedge \neg \delta_{lt})$, являются также наборами значений входных переменных, для которых не определен переход. Таким образом, если любое из условий $s_q^{\sigma} = \{\varphi_i\}$ на переходах из состояния q по событию σ заменить на условие $\varphi_i' = \varphi_i \vee (\vee \psi_l)$, то свойство полноты переходов начнет выполняться.

На рис. 35 показан пример диалога исправления ошибки для состояния с неполным набором исходящих переходов, а на рис. 36 последствие применения автоматизированного исправления неполноты.

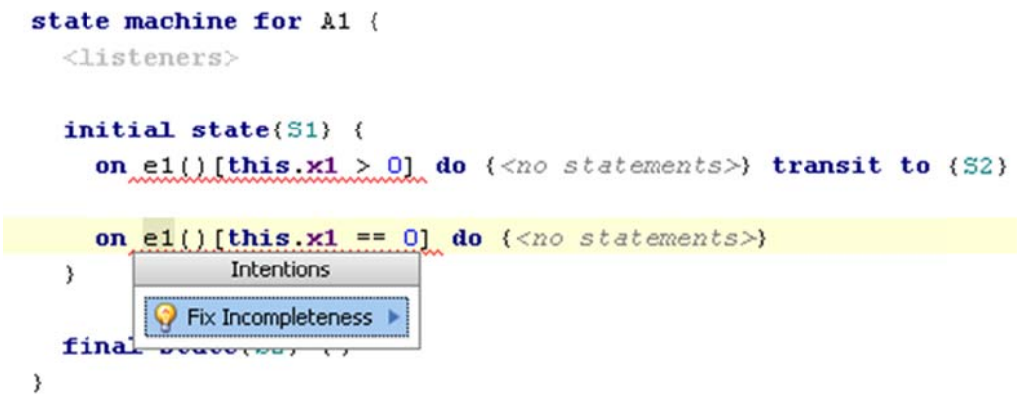

Рис. 35. Диалог исправления неполноты переходов

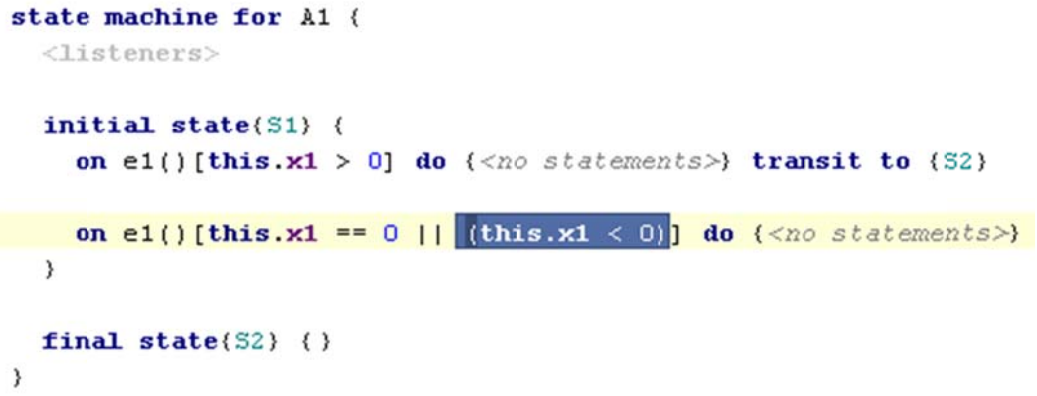

Рис. 36. Код автомата после применения автоматизированного исправления ошибки неполноты переходов

3.5.3. Исправление противоречивости переходов

Набор контрпримеров для секвенции $\varphi_{\text{н}ij} \vdash$ определяет множества значений входных переменных, для которых условия φ_i и φ_j выполняются одновременно. Если набор секвенций $\{C_l = \{\gamma_{ls}\} \vdash \{\delta_{lt}\}\}$ – множество всех контрпримеров для секвенции $\varphi_{\text{н}ij} \vdash$, то наборы значений входных переменных, выполняющих формулы $\psi_l = (\wedge\gamma_{ls}) \wedge (\wedge\neg\delta_{lt})$, являются также наборами значений входных переменных, для которых условия φ_i и φ_j выполняются одновременно. Таким образом, если условие φ_i заменить на

условие $\varphi_i' = \varphi_i \wedge \neg(\vee \psi_l)$ или условие φ_j заменить на условие $\varphi_j' = \varphi_j \wedge \neg(\vee \psi_l)$, то противоречие условий на переходах будет устранено.

На рис. 37 показан пример диалога исправления ошибки для состояния с противоречивым набором исходящих переходов, а на рис. 38 приведен результат применения автоматизированного исправления противоречивости переходов.

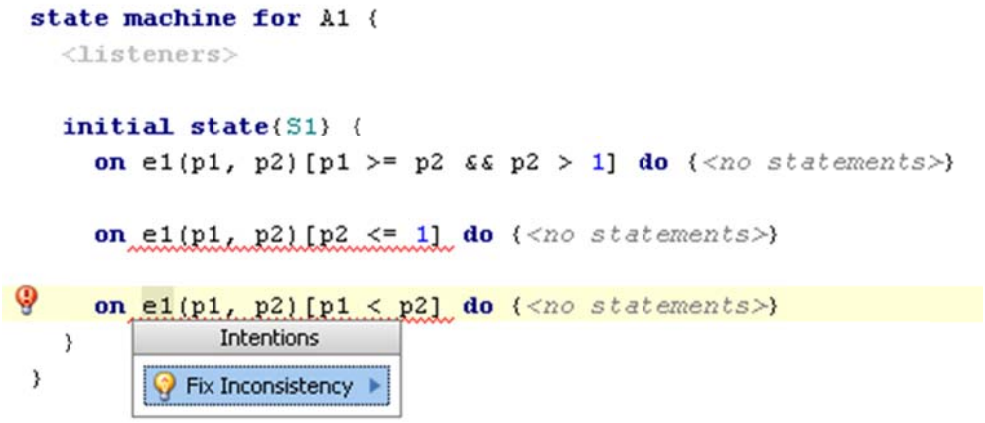

Рис. 37. Диалог исправления противоречивости переходов

```
state machine for A1 {
  <listeners>

  initial state(S1) {
    on e1(p1, p2)[p1 >= p2 && p2 > 1] do {<no statements>}

    on e1(p1, p2)[p2 <= 1] do {<no statements>}

    on e1(p1, p2)[p1 < p2 && !(p2 <= 1 && p2 > p1)] do {<no statements>}
  }
}
```

Рис. 38. Код автомата после применения автоматизированного исправления ошибки противоречивости переходов

3.5.4. Реализация проверки достижимости состояний

Алгоритмы проверки достижимости всех состояний из начального и достижимости конечного состояния из каждого состояния автомата,

описанные выше, не учитывают наличие вложенных состояний. Однако составные состояния, имеющие исходящие переходы и содержащие вложенные состояния, являются лишь краткой записью для общих исходящих переходов у вложенных состояний. Наличие исходящего перехода из составного состояния эквивалентно наличию такого же исходящего перехода из каждого вложенного состояния.

Поэтому обход в глубину для поиска всех состояний, достижимых из начального, реализован следующим образом:

1. Обход начинается с загрузки начального состояния автомата в стек.
2. Если стек пуст, то завершить работу алгоритма.
3. Из стека извлекается текущее состояние q.
4. Если состояние q составное, то в стек загружается начальное состояние, вложенное в состояние q. Выполняется переход к шагу 2.
5. Если состояние q ранее было помечено как пройденное, то осуществляется переход к шагу 2.
6. Состояние q помечается как пройденное.
7. Все состояния, в которые состояние q вложено, помечаются как пройденные.
8. Все состояния, в которые имеется переход из состояния q, загружаются в стек. Все состояния, в которые имеется переход из состояний, в которые состояние q вложено, также загружаются в стек. Выполняется переход к шагу 2.

Благодаря шагу 7, составные состояния считаются достижимыми, если они содержат хотя бы одно достижимое состояние.

Все состояния, недостижимые из начального состояния автомата, помечаются как ошибочные (рис. 39). Правило, обеспечивающее вывод пользователю сообщения о недостижимости состояния, приведено на рис. 40.

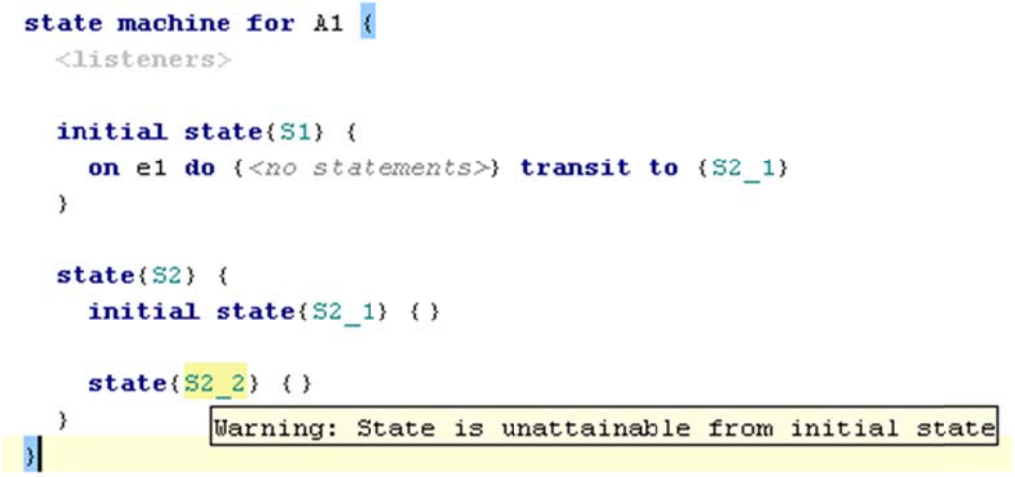

Рис. 39. Сообщение о недостижимости состояния из начального состояния автомата

```
non type system rule validate_StateMachine_AttainableFromInitialState {
  applicable for concept = StateMachine as stateMachine
  overrides false

  do {
    foreach state in stateMachine.getUnattainableFromInitialStates×() {
      warning "State is unattainable from initial state" -> state;
    }
  }
}
```

Рис. 40. Правило на языке *typesystem*, выводящее для состояния *state* сообщение о его недостижимости из начального состояния автомата

Выводы по главе 3

1. Описана формальная модель автомата, пригодная для описания формального метода валидации автоматов.
2. Предложен алгоритм проверки тавтологичности и невыполнимости булевых формул с предикатами сравнения входных переменных с константой и входных переменных между собой.
3. Предложены методы валидации свойств автоматной модели.
4. Предложенные методы валидации автоматов реализованы в среде *MPS*.

ГЛАВА 4. ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРИ МНОГОПОТОЧНОМ АВТОМАТНОМ ПРОГРАММИРОВАНИИ

В настоящее время растет актуальность разработки параллельных программ. Это связано с тем, что дальнейшее увеличение производительности вычислительных систем более не может достигаться увеличением тактовой частоты процессоров, и главной тенденцией их развития становится увеличение числа ядер в системах [81]. Однако для того, чтобы программа выполнялась эффективно на нескольких ядрах, она должна быть написана с применением техник параллельного программирования [83].

Универсальные императивные языки программирования [84], наиболее распространенные на сегодняшний день, хорошо подходят для реализации последовательных алгоритмов, но плохо приспособлены для написания параллельных алгоритмов. Это связано с тем, что программа, написанная на императивном языке, представляет собой последовательность инструкций процессору, изменяющих состояние памяти [85]. Из-за того, что порядок этих команд фиксирован, выполнение программы не может быть распределено между несколькими процессорами. Поэтому программисты вынуждены самостоятельно разбивать свою программу на параллельно выполняющиеся потоки [86].

Автоматическому распараллеливанию хорошо поддаются программы, написанные на функциональных языках программирования [4, 87]. Однако применение функциональных языков, в отличие от императивных, значительно менее распространено. Тем не менее, некоторые функциональные конструкции в последнее время стали активно проникать в универсальные императивные языки программирования. Например, становится популярным использование в императивных языках замыканий

[88] для обработки списков. В некоторых случаях [89] это позволяет выполнять автоматическое распараллеливание по данным [90].

Поэтому естественно продолжить адаптацию абстракций параллельных функциональных программ для императивных языков программирования. К таким абстракциям, в частности, относится *акторная модель* [111], реализованная в языках программирования *Erlang* [15] и *Scala* [112].

Акторная модель была предложена К. Хьюиттом, П. Бишопом и Р. Штайгером в 1973 г. Они ввели понятие *актора* – примитива параллельных вычислений, обладающего потоком выполнения и способного обмениваться сообщениями с другими акторами. Акторная модель обладает следующими особенностями:

- все является актором (в объектно-ориентированном программировании все является объектом);
- у разных акторов нет общих данных;
- акторы взаимодействуют посредством асинхронной посылки сообщений друг другу;
- каждый актор имеет очередь для буферизации входящих сообщений.

В этой модели акторы обладают следующими свойствами и возможными действиями:

- в качестве реакции на входящее сообщение они могут:
  - изменять только свое внутреннее состояние;
  - создавать другие акторы;
  - посылать сообщения другим акторам;
- акторы никогда не используют общее состояние и, поэтому у них нет необходимости заботиться о блокировках и других видах синхронизации;
- акторы не блокируются, ожидая ответов на свои запросы.

4.1. Акторное расширение языка *Java* в среде *MPS*

Разработанный при участии автора язык *actors* [113, 114] для среды *MPS* позволяет использовать акторные конструкции при программировании на языке *Java*. На рис. 41 приведен пример кода, содержащего объявления

типа актора, объявления сообщения, создания нового актора и посылки сообщения. Из рисунка видно, что такой код синтаксически почти не отличается от объявления класса, объявления метода, создания нового объекта и вызова метода в языке *Java*. Поэтому *Java*-программисту потребуются минимальные усилия для изучения синтаксиса акторного языкового расширения.

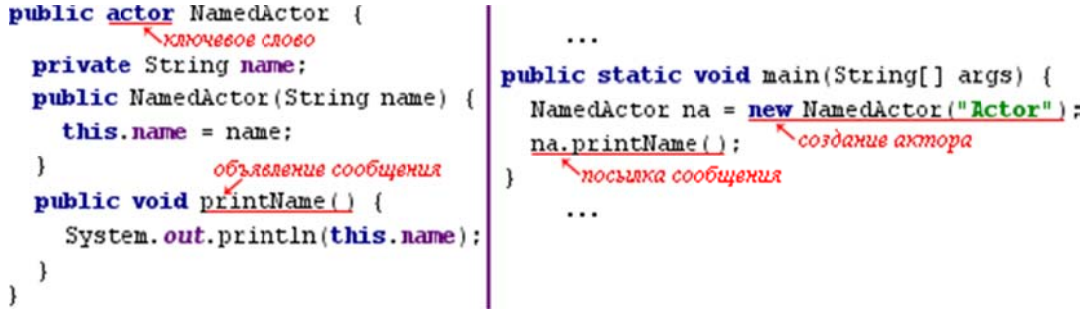

Рис. 41. Пример синтаксиса акторного расширения языка *Java*

Разработанный язык позволяет использовать акторы наряду с обычными объектами при программировании на языке *Java*. Декларация типа актора, поддерживаемая языком, синтаксически совпадает с декларацией *Java*-класса с той лишь разницей, что вместо ключевого слова *class* используется ключевое слово *actor*. Для типа актора могут быть определены конструкторы, поля и объявления типов обрабатываемых сообщений. Синтаксис последних полностью совпадает с синтаксисом объявлений обычных методов в языке *Java*. Аналогично, синтаксис отправки сообщений актору, не отличается от синтаксиса вызова методов у объекта. Однако, в отличие от вызова методов, отправка сообщений происходит асинхронно: сообщения помещаются в индивидуальную очередь актора и обрабатываются актором последовательно.

Содержание потока в языке *Java* – ресурсоемкий процесс. Поэтому для обработки сообщений актору не выделяется отдельный экземпляр класса *java.lang.Thread* [115]. Вместо этого, отправка всех сообщений всем акторам происходит посредством добавления сообщений в дополнительно введенную

очередь общего диспетчера. Диспетчер обладает пулом потоков [116], размер которого либо задается программистом явно, либо вычисляется, исходя из числа доступных ядер в системе. По мере появления новых сообщений в очереди диспетчера и освобождения потоков из пула, сообщения последовательно извлекаются из очереди для обработки акторами.

При этом гарантируется, что сообщения, направленные одному и тому же актору, не могут обрабатываться параллельно двумя разными потоками. Таким образом, эмулируется обладание актором потоком управления.

На рис. 42 приведен пример объявления двух типов акторов: *Ping* и *Pong*. При отправке актору *Ping* сообщения *start*, между акторами начинается обмен сообщениями *ping* и *pong*. После отправки заданного параметром *pingCount* числа сообщений актор *Ping* отправляет актору *Pong* сообщение *stop* и прекращает работу. В ответ на сообщение *stop* актор *Pong* также завершает работу.

```
public actor Ping {
  private int pingsLeft;
  public Ping() { }
  public void pong(Pong pong) {
    if (this.pingsLeft-- >= 0) {
      System.out.println("Ping: pong");
      pong.ping(this);
    } else {
      System.out.println("Ping: stop");
      pong.stop();
    }
  }
  public void start(int count, Pong pong) {
    System.out.println("Ping: start");
    this.pingsLeft = count;
    this.pong(pong);
  }
}
```

```
public actor Pong {
  public Pong() { }
  public void ping(Ping ping) {
    System.out.println("Pong: ping ");
    ping.pong(this);
  }
  public void stop() {
    System.out.println("Pong: stop");
  }
}
```

Рис. 42. Пример объявления акторов

Для запуска обмена сообщениями необходимо создать акторы *Ping* и *Pong* и отправить актору *Ping* сообщение *start*. Код, выполняющий эти действия, и протокол работы этого кода приведены на рис. 43.

```
...
public static void main(String[] args) {
  Pong pong = new Pong();
  Ping ping = new Ping();
  ping.start(3, pong);
}
...
```

```
Ping: start
Ping: pong
Pong: ping
Ping: pong
Pong: ping
Ping: pong
Pong: ping
Ping: stop
Pong: stop
```

Рис. 43. Запуск обмена сообщениями между акторами и вывод программы

4.2. Обработка сообщений с задержкой

Язык *actors* позволяет при посылке сообщения указать минимальную задержку, с которой оно должно быть обработано. Для этого в этот язык введена конструкция *defer*, которая может быть применена к оператору посылки сообщения. В качестве параметра конструкция *defer* принимает величину минимальной задержки, для задания которой используется встроенное в среду *MPS* языковое расширение *Dates*.

В этом расширении предусмотрены специальные проблемно-ориентированные конструкции для работы с датами, временем и промежутками времени. В частности, оно позволяет записывать периоды времени наглядно, указывая их величину и размерность: 5 *minutes*, 1 *year*.

Пример использования конструкции *defer* приведен на рис. 44: код актора *Pong* отредактирован таким образом, чтобы посылка сообщения *pong* происходила не сразу, а с задержкой в пять миллисекунд.

```
public actor Pong {
  public Pong() { }
  public void ping(Ping ping) {
    System.out.println("Pong: ping ");
    ping.pong(this).defer(5 milliseconds);
  }
  public void stop() {
    System.out.println("Pong: stop");
  }
}
```

Рис. 44. Отправка сообщения с задержкой

4.3. Отложенный результат

В традиционном объектно-ориентированном программировании некоторые методы в результате своего выполнения возвращают вычисленные значения. Эти значения становятся доступны вызывающему коду сразу по окончанию синхронного выполнения таких методов. Так может быть организованно взаимодействие между вызываемым и вызывающим кодом.

При программировании с использованием акторов нет возможности использовать возвращаемые значения, так как к тому моменту, когда операция асинхронной посылки сообщения оказывается выполненной, само сообщение может быть еще не обработано, а возвращаемое значение не вычислено.

Для того чтобы вызываемый код мог оповестить вызывающий код об окончании обработки события и передать вычисленное значение, можно передавать вызывающий актор в качестве параметра сообщения. По окончании обработки сообщения вызываемый актор должен послать вызывающему актору ответное сообщение. Именно так и организовано взаимодействие в приведенном выше примере (рис. 42).

Однако при таком подходе код обрабатывающего сообщение актора должен «знать» о структуре посылающего сообщение актора. Таким образом возникает нежелательная сильная связанность кода [97].

Для того чтобы обойти эту проблему и позволить использовать возвращаемые обработчиками сообщений значения, в языке *actors* существует поддержка механизма отложенных результатов. Для любого сообщения актора, так же как и для любого метода в языке *Java*, могут быть заданы тип возвращаемого значения и набор генерируемых исключений. Код обработчика сообщения должен либо вычислить значение соответствующего типа, либо сгенерировать исключение.

Оператор посылки сообщения имеет специальный тип *deferred*, параметризованный типом возвращаемого сообщением значения. Например,

если обработчик сообщения возвращает значение типа *String*, то асинхронная посылка этого сообщения вернет значение типа *deferred*<*String*>. Используя это значение, код, посылающий сообщение, может определить действия как в связи с успешной обработкой сообщения, так и в связи с возникновением исключительной ситуации. Действия задаются в виде замыканий – анонимных функций, языковая поддержка которых осуществляется встроенным в среду *MPS* расширением *Closures*.

Использование отложенного значения может быть продемонстрировано на примере программы редактора графов. Одной из функций такой программы является чтение графов из файлов, реализованное актором *GraphReader*. При получении сообщения *read*, актор *GraphReader* начинает выполнять длительную операцию чтения и разбора файла. Результатом этой операции является экземпляр класса *GraphModel*, но в процессе ее выполнения может возникнуть исключение *IOException*. Часть программы, использующая актор *GraphReader* (рис. 45), посылает ему сообщение *read*, и сохраняет полученное отложенное значение в переменной *model*. Далее с помощью оператора *addCallback* назначается действие в связи с успешной обработкой сообщения, а с помощью оператора *addErrback* – обработчик исключения *IOException*.

```
...
GraphReader reader = new GraphReader();
deferred< GraphModel > model = reader.read(new File("./graphModel.grph"));
model.addCallback({GraphModel result => frame.setGraphModel(result); });
model.addErrback({IOException e =>
    JOptionPane.showMessageDialog(fileChooser, "Cannot open the file"); });
...
```

асинхронный вызов метода

обработка результата

обработка исключения IOException

Рис. 45. Действия по окончанию обработки сообщения

4.4. Генерация кода для языка *actors*

Для того чтобы обрабатывать вызовы методов асинхронно, для каждого метода, объявленного внутри конструкции *actor* языка *actors*, в

классе, генерируемом из актора, дополнительно создается метод асинхронного вызова. Этот метод принимает в качестве параметров величину задержки вызова и параметры исходного метода. В качестве типа возвращаемого значения метода асинхронного вызова имеет тип *deffered*.

В теле метода асинхронного вызова выполняется добавление в очередь сообщений актора замыкания, вызывающего исходный метод, и формируется объект отложенного результата.

Например, для метода *ping* актора *Pong*, приведенного ранее на рис. 42, будет сгенерирован метод асинхронного вызова, приведенный на рис. 46.

```
public Deferred<Void> ping_returningDeferred(
    final period delay, Ping ping) {
  Deferred<Void> deferred = new Deferred<Void>(true);
  try {
    Dispatcher.getDispatcher().
        addTask(new Task(now + delay, { =>
      try {
        Dispatcher.setCurrentDeferred(deferred);
        Void result = null;
        this.ping(ping);
        deferred.setResult(result);
        Dispatcher.setCurrentDeferred(null);
      } catch (Throwable e) {
        deferred.setException(e);
        Dispatcher.setCurrentDeferred(null);
      }
    }, this));
  } catch (InterruptedException e) {
    deferred.setException(e);
  }
  return deferred;
}
```

Рис. 46. Метод асинхронного вызова для метода *ping* актора *Pong*

Вызовы методов акторов заменяются в процессе генерации на вызовы методов асинхронного вызова. Например, код, сгенерированный для вызова метода *ping* актора *Pong*, приведен на рис. 47.

```
public void pong(Pong pong) {
  if (this.pingsLeft-- >= 0) {
    System.out.println("Ping: pong");
    pong.ping_returningDeferred(0 seconds, this);
  } else {
    System.out.println("Ping: stop");
    pong.stop_returningDeferred(0 seconds);
  }
}
```

Рис. 47. Код, сгенерированный для вызова метода *ping* актора *Pong*

4.5. Совместное использование языков *stateMachine* и *actors*

Описанные языки *stateMachine* и *actors* являются расширениями языка *baseLanguage*. В совокупности они образуют язык многопоточного автоматного программирования.

Концепт *ActorConcept*, использующийся в языке *actors* для представления актора, унаследован от концепта *ClassConcept* языка *baseLanguage* (рис. 48).

```
concept ActorConcept extends     ClassConcept
                     implements IBLDeprecatable

  instance can be root: true

  properties:
  << ... >>

  children:
  << ... >>

  references:
  << ... >>

  concept properties:
  alias            = actor
  shortDescription = actor concept
```

Рис. 48. Объявление концепта *ActorConcept*

Каждый автомат в языке *stateMachine* связан с классом, автоматное поведение которого он описывает. В частности, таким классом может быть экземпляр концепта *ActorConcept*. Таким образом, комбинация языков

stateMachine и *actors* позволяет описывать автоматы, обладающие собственным потоком управление и обрабатывающие события асинхронно. Такие автоматы сочетают в себе достоинства автоматного программирования и акторной модели. Следовательно, они могут быть использованы для *многопоточного автоматного программирования*.

4.5.1. Пример использования многопоточного автоматного программирования

В качестве примера, приведем систему управления лифтом, описанную в главе 2, в которой и автоматный класс, и классы объектов управления являются акторами (рис. 49 – рис. 51).

```
public actor ActorElevator extends <none> implements
    ElevatorEngineListener, DoorsEngineListener,
    LoadingTimerListener {
  <<static fields>>

  <<static initializer>>
  public int floor = 1;
  public TaskList tasks = new TaskList();
  public ActorElevatorEngine elevatorEngine;
  public ActorDoorsEngine doorsEngine;
  public LoadingTimer loadingTimer;
  <<properties>>
  <<initializer>>
  public ActorElevator(
      ActorElevatorEngine elevatorEngine,
      ActorDoorsEngine doorsEngine,
      LoadingTimer loadingTimer) {
    this.elevatorEngine = elevatorEngine;
    this.doorsEngine = doorsEngine;
    this.loadingTimer = loadingTimer;
    this.elevatorEngine.addElevatorEngineListener(this);
    this.doorsEngine.addDoorsEngineListener(this);
    this.loadingTimer.addLoadingTimerListener(this);
  }

  public event doorsOpened();
  public event doorsClosed();
  public event reached(int floor);
  public event call(int toFloor, CallType from);
  public event openDoors();
  public event loadingTimeout();
  public event fire();
```

Рис. 49. Объявление актора *ActorElevator*, поведение которого задано автоматом

```
public actor ActorElevatorEngine extends <none> implements
    <none> {
  private static int TICK_DELAY = 50;
  private static double VELOCITY = 1.0 / 64;
  private static double DELTA = 0.000001;

  <<static initializer>>
  private double velocity = 0;
  private list<ElevatorEngineListener> listeners
      = new arraylist<ElevatorEngineListener>;
  private list<ChangeListener> changeListeners
      = new arraylist<ChangeListener>;
  public int floorsCount {get; private set;}
  public double position {get; private set;}
  <<initializer>>
  public ActorElevatorEngine(int floorsCount) {
    this.floorsCount = floorsCount;
    this.position = 1;
  }

  public void turnUp() {
    if (this.floorsCount - DELTA > this.position) {
      this.velocity = VELOCITY;
      this.run();
    }
  }

  public void turnDown() {
    if (DELTA < this.position - 1) {
      this.velocity = 0 - VELOCITY;
      this.run();
    }
  }
```

Рис. 50. Объявление актора *ActorElevatorEngine*, являющегося объектом управления «Двигатель лифта»

```
public actor ActorDoorsEngine extends <none> implements
    <none> {
  private static int TICK_DELAY = 50;
  private static double VELOCITY = 1.0 / 64;
  private static double DELTA = 0.000001;

  <<static initializer>>
  private double velocity = 0;
  private list<DoorsEngineListener> listeners
      = new arraylist<DoorsEngineListener>;
  private list<ChangeListener> changeListeners
      = new arraylist<ChangeListener>;
  public double position {get; private set;}
  <<initializer>>
  public ActorDoorsEngine() {
    this.position = 1;
  }

  public void open() {
    if (1 - DELTA > this.position) {
      this.velocity = VELOCITY;
      this.run();
    }
  }

  public void close() {
    if (DELTA < this.position) {
      this.velocity = 0 - VELOCITY;
      this.run();
    }
  }
```

Рис. 51. Объявление актора *ActorDoorsEngine*, являющегося объектом управления «Двигатель, открывающий двери лифта»

Таким образом, *для перехода от однопоточного автоматного программирования к многопоточному достаточно перейти от использования классов к использованию акторов.* Так как синтаксис акторов в языке *actors* незначительно отличается от синтаксиса классов в языке *Java*, такой переход может быть осуществлен достаточно просто.

Моделирование системы управления лифтом в виде взаимодействующих акторов лучше, отражает реальное взаимодействие компонентов такой системы, чем однопоточное моделирование, так как при

аппаратной реализации системы управления различные компоненты функционируют параллельно.

4.5.2. Генерация кода при совместном использовании языков *stateMachine* и *actors*

Генерация кода акторов, поведение которых задано автоматом, осуществляется в два этапа: сначала редуцируются конструкции языка *stateMachine*, затем редуцируются конструкции языка *actors* (рис. 52 – рис. 54).

```
public actor A  {
  public A() { }

  public event e1();
}

state machine for A {
  initial state{S1} {
    on e1 do {} transit to {S2}
  }
  state{S2} {
    on e1 do {} transit to {S1}
  }
}
```

Рис. 52. Код на языках *baseLanguage*, *stateMachine* и *actors*

```
public actor A {
  private State state;
  public A() {
    this.state = State.S1;
  }
  public void e1() {
    switch (this.state) {
      case State.S1 :
        if (true) {
          this.state = State.S2;
          break;
        }
      case State.S2 :
        if (true) {
          this.state = State.S1;
          break;
        }
    }
  }
  public enum State implements <none> {
    S1()
    S2()
    public State() { }
  }
}
```

Рис. 53. Код на языках *baseLanguage* и *actors*

```
public class A {
  private State state;
  public A() {
    this.state = State.S1;
  }
  public void e1() {
    switch (this.state) {
      case State.S1 :
        if (true) {
          this.state = State.S2;
          break;
        }
      case State.S2 :
        if (true) {
          this.state = State.S1;
          break;
        }
    }
  }
  public Deferred<Void> e1_returningDeferred(
      final period delay) {
    Deferred<Void> deferred = new Deferred<Void>(true);
    try {
      Dispatcher.getDispatcher().
          addTask(new Task(now + delay, { =>
        try {
          Dispatcher.setCurrentDeferred(deferred);
          Void result = null;
          this.e1();
          deferred.setResult(result);
          Dispatcher.setCurrentDeferred(null);
        } catch (Throwable e) {
          deferred.setException(e);
          Dispatcher.setCurrentDeferred(null);
        }
      }, this));
    } catch (InterruptedException e) {
      deferred.setException(e);
    }
    return deferred;
  }
  public enum State implements <none> {
    S1()
    S2()
    public State() { }
  }
}
```

Рис. 54. Код на язык *baseLanguage*

Для управления порядком применения правил генерации среда *MPS* позволяет задавать приоритет применения генераторов разных языков. Для генератора языка *stateMachine* указано, что он должен применяться до генератора языка *actions* (рис. 55).

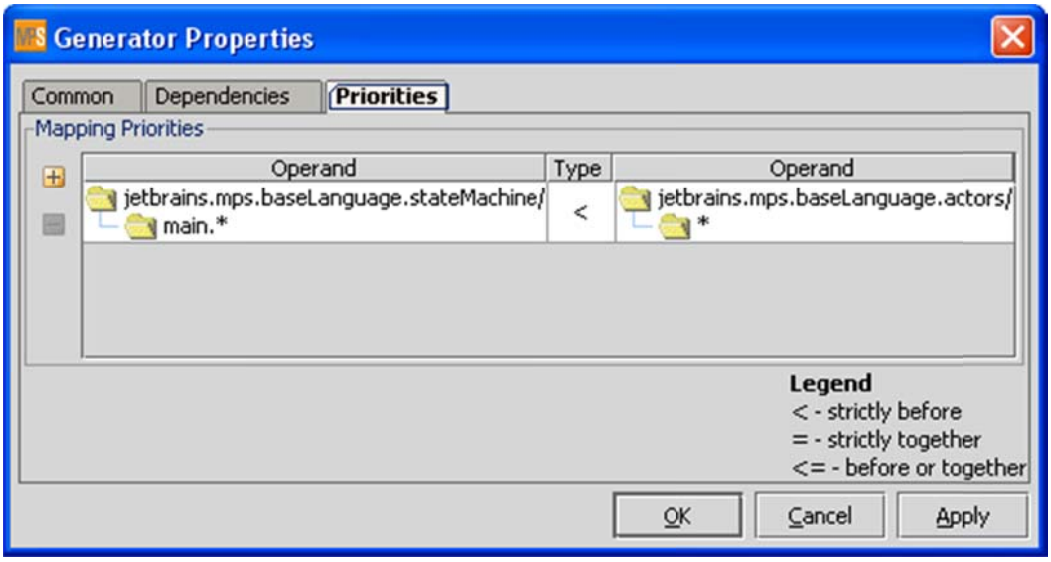

Рис. 55. Диалог задания приоритетов генерации для языка *stateMachine*

Выводы по главе 4

1. Разработан и описан язык *actors* и инструментальные средства, позволяющие в привычной для *Java*-программистов манере использовать автоматически распараллеливающиеся функциональные конструкции. Язык упрощает создание программ, которые эффективно используют возможности многоядерных компьютеров.
2. Выполнена интеграция акторного языка и его инструментального средства с языком и инструментальным средством для автоматного программирования. Таким образом, обеспечена поддержка многопоточного автоматного программирования.

ГЛАВА 5. ПРИМЕНЕНИЕ ТЕКСТОВОГО ЯЗЫКА ДЛЯ АВТОМАТНОГО ПРОГРАММИРОВАНИЯ В ПРОЕКТЕ *YOUTRACK*

Разработанный язык автоматного программирования был применен при разработке нескольких подсистем коммерческой системы учета программных дефектов *YouTrack*. В качестве примера в диссертации описывается реализация подсистемы управления и слежения за парком серверов с развернутыми на них экземплярами системы *YouTrack*.

Эта подсистема реализована в виде отдельного *WEB*-приложения [117] с базой данных и позволяет выполнять следующие задачи:

- запускать и останавливать виртуальные хосты (компьютеры), работающие под управлением *Amazon Elastic Cloud* [118];
- создавать виртуальное устройство хранения данных (жесткий диск) и подключать его к запущенному виртуальному хосту;
- загружать новые или обновлять существующие версии программного обеспечения на запущенном виртуальном хосте;
- запускать и останавливать *Java* процессы на указанных виртуальных хостах;
- запускать и останавливать экземпляры системы *YouTrack* внутри *Java*-процессов;
- следить за состоянием всех запущенных процессов;
- внешним пользователям регистрировать заявки на создание и запуск собственного экземпляра *YouTrack*.

В результате анализа предметной области были выделены следующие классы:

- *YouTrackInstance* – соответствует экземпляру системы *YouTrack* и ассоциирован с *Java*-процессом, в котором запущен;

- *YouTrackServer* – соответствует *Java*-процессу и ассоциирован с множеством объектов класса *YouTrackInstance*.
- *EC2Host* – отвечает за представление виртуального хоста из *Amazon Elastic Cloud* и агрегирует множество экземпляров класса *YouTrackServer*, которые запущены на виртуальном хосте.

На рис. 56 представлена диаграмма классов описанной предметной области.

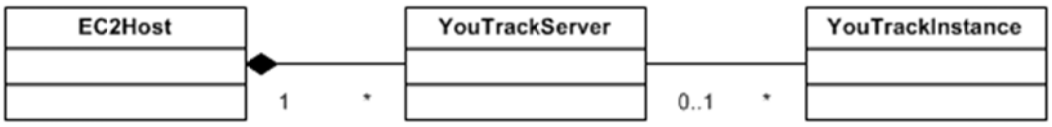

Рис. 56. Диаграмма классов предметной области

5.1. Автоматы с состояниями, хранимыми в базе данных

5.1.1. Язык для хранения объектов в базе данных

Состояние перечисленных классов должно сохраняться при перезагрузках *WEB*-приложения. Для этого целесообразно использовать базу данных. В рамках проекта *YouTrack* для хранения объектов в базе данных при участии автора был разработан проблемно-ориентированный язык *DNQ* (*Data Navigation and Query*) [119]. В этом языке определен концепт *хранимый класс* (*PersistentClassConcept*), который расширяет понятие *Java*-класса языка *baseLanguage*. Вместо полей, хранимый класс может иметь ассоциации следующих видов:

- простая ассоциация для хранения примитивных типов данных, таких как строка (*string*), целое число (*int*) или дата (*instant*);
- направленная ассоциация для хранения связи с другими хранимыми классами;

- двунаправленная ассоциация – играет роль двух направленных в разные стороны ассоциаций. При этом изменения, вносимые в первую ассоциацию, автоматически влияют на значение второй;
- агрегация – позволяет строить отношения вложения дочерних объектов в родительские. Особенностью этого типа ассоциаций является то, что при удалении родительского объекта автоматически удаляются все вложенные в него дочерние объекты.

Помимо хранимых классов в языке *DNQ* существуют *хранимые перечисления* (*PersistentEnumConcept*), расширяющие стандартные *Java*-перечисления языка *baseLanguage*.

Таким образом, язык *DNQ* позволяет проектировать схему базы данных в виде набора хранимых классов. Запросы к данным записываются в виде выражений языка для работы с коллекциями (*collections*). Эти выражения при генерации редуцируются в оптимизированные запросы к базе данных.

Хранимые классы *DNQ* – это расширенные классы языка *baseLanguage*. Поэтому к ним применимы все расширения последнего, в том числе, и автоматное расширение, описанное в данной работе. С помощью хранимых перечислений задается множество состояний, в которых может находиться хранимый класс, также как с помощью обычных *Java*-перечислений задается множество состояний *Java*-класса.

5.2. Реализация подсистемы в проекте *YouTrack*

Далее приведен код программы на языках *DNQ* и *stateMachine*, описывающий структуру и поведение классов, рассматриваемой предметной области (рис. 57 – рис. 62).

5.2.1. Класс *YouTrackInstance*

На рис. 57 приведено описание полей хранимого класса *YouTrackInstance* на языке *DNQ*, а на рис. 58 – описание поведения хранимого класса *YouTrackInstance* на языке *stateMachine*.

```
public persistent class YouTrackInstance extends <none> implements <none> {

    version mismatch resolution: ignore

    public static string NAME_PATTERN = "^[0-9A-Za-z][\\-0-9A-Za-z]+";
    private static Pattern P = Pattern.compile(NAME_PATTERN);

    public simple string name unique;
    public simple string ownerEmail unique;
    public simple string ownerJabber opts;
    public simple instant created required;
    public simple string secretKey required;
    public bidirectional association YouTrackServer[0..1] server onDelete(clear);
    public bidirectional association EC2Volume[0..1] volume onDelete(clear);
    << composite unique constraints >>
```

Рис. 57. Описание полей хранимого класса *YouTrackInstance* на языке *DNQ*

```
state machine for YouTrackInstance {
  <listeners>

  initial state{UNKNOWN} {
    on startup[this.server == null] do {<no statements>} transit to {DOWN}
    on serverUp(runningInstances) do {<no statements>} transit to {DOWN}
    on serverDown do {<no statements>} transit to {DOWN}
    on delete[this.server == null] do {<no statements>} transit to {DELETED}
  }

  state{RUNNING} {
    on startup do {<no statements>} transit to {UNKNOWN}
    on serverDown do {<no statements>} transit to {DOWN}
    on stop do {
      this._stop();
    } transit to {DOWN}
  }

  state{DOWN} {
    on startup[this.server != null] do {<no statements>} transit to {UNKNOWN}
    on serverUp(runningInstances)[!(runningInstances.contains(this.name))] do {
      this._start();
    } transit to {RUNNING}
    on delete do {<no statements>} transit to {DELETED}
    on start do {
      this._start();
    } transit to {RUNNING}
  }

  state{DELETED} {
    enter do {
      this.delete();
    }
  }
```

Рис. 58. Описание поведения хранимого класса *YouTrackInstance* на языке *stateMachine*

5.2.2. Класс YouTrackServer

На рис. 59 приведено описание полей хранимого класса *YouTrackServer* на языке *DNQ*, а на рис. 60 – описание поведения хранимого класса *YouTrackServer* на языке *stateMachine.*

```
public persistent class YouTrackServer extends <none> implements <none> {

  version mismatch resolution: ignore

  public static int START_PORT = 8080;

  public bidirectional association EC2Host[0..1] host opts;
  public simple int port required;
  public simple int adminPort required;
  public simple string stateMessage opts;
  public unordered bidirectional association YouTrackInstance[0..n] instances onDelete(clear);
```

Рис. 59. Описание полей хранимого класса *YouTrackServer* на языке *DNQ*

```
state machine for YouTrackServer {
  <listeners>

  initial state{DOWN} {
    on startup do {<no statements>} transit to {UNKNOWN}
    on tick[this.ping()] do {<no statements>} transit to {RUNNING}
    on delete do {
      this._delete();
    }
    on start[this.host.isInStateUP()] do {
      this._start();
    } transit to {STARTING}
  }

  state{UP} {
    on startup do {<no statements>} transit to {UNKNOWN}
    initial state{STARTING} {
      on tick[this.ping()] do {
        this._startInstances();
      } transit to {RUNNING}
    }
    state{RUNNING} {
      on tick[!(this.ping())] do {<no statements>} transit to {DOWN}
      on stop do {
        this._stop();
      } transit to {STOPPING}
    }
    state{STOPPING} {
      on tick[!(this.ping())] do {<no statements>} transit to {DOWN}
    }
  }

  state{UNKNOWN} {
    on tick[this.ping()] do {<no statements>} transit to {RUNNING}
    on tick do {<no statements>} transit to {DOWN}
  }
}
```

Рис. 60. Описание поведения хранимого класса *YouTrackServer* на языке *stateMachine*

5.2.3. Класс *EC2Host*

На рис. 61 приведено описание полей хранимого класса *EC2Host* на языке *DNQ*, а на рис. 62 – описание поведения хранимого класса *EC2Host* на языке *stateMachine*.

```
public persistent class EC2Host extends AbstractHost implements <none> {

  version mismatch resolution: ignore

  << static fields >>

  public simple long number unique, nullObjectValue(0L);
  public simple int waitCount opts;
  private simple string ec2InstanceId opts;
  public simple string ec2type opts;
  public simple string ec2securityGroup opts;
  public simple string ec2keyName opts;
  public simple string ec2privateKey opts;
  private simple string publicDnsName opts;
  public unordered bidirectional association YouTrackServer[0..n] servers opts;
```

Рис. 61. Описание полей хранимого класса *EC2Host* на языке *DNQ*

```
state machine for EC2Host {
  <listeners>

  initial state{DOWN} {
    on startup do {<no statements>} transit to {UNKNOWN}
    on start(volume) do {
      this.startEC2Instance();
      this.attachVolume(volume);
    } transit to {STARTING}
  }

  state{UNKNOWN} {
    on tick[this.isEC2State("pending")] do {<no statements>} transit to {STARTING}
    on tick[this.isEC2State("running")] do {<no statements>} transit to {RUNNING}
  }

  state{UP} {
    on tick[!(this.ping())] do {<no statements>} transit to {UNKNOWN}
    on startup do {<no statements>} transit to {UNKNOWN}
    on terminate do {
      this.stopAllServers();
    } transit to {TERMINATING}
    initial state{STARTING} {
      on tick[this.isEC2State("running")] do {<no statements>} transit to {UPDATING}
    }
    state{UPDATING} {
      enter do {
        this._update();
      }
      on softwareUpdated do {<no statements>} transit to {RUNNING}
    }
    state{RUNNING} {
      on update do {<no statements>} transit to {UPDATING}
    }
    state{TERMINATING} {
      on tick[this.isAllServersDown()] do {
        this._terminate();
      } transit to {DOWN}
    }
  }
```

Рис. 62. Описание поведения хранимого класса *EC2Host* на языке *stateMachine*

5.3. Совместное использование языков *stateMachine* и *DNQ*

При применении языка *stateMachine* для описания поведения обычных *Java*-классов языка *baseLanguage*, в процессе генерации кода автоматная часть редуцируется до конструкций языка *baseLanguage*. После этого генерируется текст программы. При использовании языка *stateMachine* для описания поведения хранимых классов языка *DNQ* генерация кода выполняется в несколько этапов:

1. Программа на языках *DNQ*, *stateMachine* и *baseLanguage* редуцируются до программы, содержащей только конструкции языков *DNQ* и *baseLanguage*. При этом для хранения состояния создается хранимое перечисление, содержащее все состояния класса, а в соответствующий хранимый класс добавляется ассоциация с созданным перечислением для хранения текущего состояния. На рис. 63 приведен пример хранимого перечисления, созданного для состояний класса *EC2Host*.

```
persistent enum EC2HostState implements <none> {
  DOWN()
  UNKNOWN()
  DETERMINE_STATE()
  NO_PING()
  UP()
  STARTING()
  UPDATING()
  RUNNING()
  TERMINATING()
```

Рис. 63. Хранимое перечисление состояний класса *EC2Host*

2. Код на языке *DNQ* редуцируется до кода на языке *baseLanguage*. При этом конструкции вида *switch(state)*, где *state* – имя ассоциации с хранимым перечислением, редуцируются в последовательность операторов *if*, *else-if*. Это связано с тем, что целевой код, генерируемый из хранимых перечислений, не является *Java*-перечислением и поэтому не может быть использован в качестве аргумента конструкции *switch*. На рис. 64 приведен пример такой редукции для метода *EC2Host.update*, реализующего обработку события *update*. Для упрощения примера действия, выполняемые на переходах, не показаны.

```
public void update() {
  switch (this.__state__) {
    case EC2HostState.DOWN :
      break;
    case EC2HostState.UNKNOWN :
      break;
    case EC2HostState.DETERMINE_STATE :
      break;
    case EC2HostState.NO_PING :
      break;
    case EC2HostState.UP :
      break;
    case EC2HostState.STARTING :
      break;
    case EC2HostState.UPDATING :
      break;
    case EC2HostState.RUNNING :
      break;
```

```
public void update() {
  do {
    if (this.__state__ == EC2HostState.DOWN) {
      break;
    } else if (this.__state__ == EC2HostState.UNKNOWN) {
      break;
    } else if (this.__state__ == EC2HostState.DETERMINE_STATE
      break;
    } else if (this.__state__ == EC2HostState.NO_PING) {
      break;
    } else if (this.__state__ == EC2HostState.UP) {
      break;
    } else if (this.__state__ == EC2HostState.STARTING) {
      break;
    } else if (this.__state__ == EC2HostState.UPDATING) {
      break;
    } else if (this.__state__ == EC2HostState.RUNNING) {
```

Рис. 64. Пример редукции оператора *switch* в набор операторов *if*, *if-else*

3. Из модели на языке *baseLanguage* выполняется генерация текста программы. На рис. 65 показан финальный код метода *EC2Host.update*, в котором присутствуют только конструкции на языке *Java*.

```
public void update(final Entity entity) {
  do {
    if (EntityOperations.equals(AssociationSemantics.getToOne(entity, "__state__"), EC2HostStateImpl.DOWN)) {
      break;
    } else if (EntityOperations.equals(AssociationSemantics.getToOne(entity, "__state__"), EC2HostStateImpl.
        UNKNOWN)) {
      break;
    } else if (EntityOperations.equals(
        AssociationSemantics.getToOne(entity, "__state__"), EC2HostStateImpl.DETERMINE_STATE)) {
      break;
    } else if (EntityOperations.equals(AssociationSemantics.getToOne(entity, "__state__"), EC2HostStateImpl.
        NO_PING)) {
      break;
    } else if (EntityOperations.equals(AssociationSemantics.getToOne(entity, "__state__"), EC2HostStateImpl.UP)) {
      break;
    } else if (EntityOperations.equals(AssociationSemantics.getToOne(entity, "__state__"), EC2HostStateImpl.
        STARTING)) {
      break;
    } else if (EntityOperations.equals(AssociationSemantics.getToOne(entity, "__state__"), EC2HostStateImpl.
        UPDATING)) {
      break;
    } else if (EntityOperations.equals(AssociationSemantics.getToOne(entity, "__state__"), EC2HostStateImpl.
        RUNNING)) {
```

Рис. 65. Код на языке *Java*, реализующий обработку события *EC2Host.update*

Выводы по главе 5

1. В качестве примера применения предложенного языка автоматного программирования *stateMachine* описано поведение хранимых классов – классов, состояние которых сохраняется в базе данных.

2. При этом за счет механизма расширения *MPS*-языков удается полностью переиспользовать все синтаксические конструкции автоматного языка при его применении для описания поведения хранимых классов.
3. Изложенный подход позволяет унифицировать описание поведения всех классов в программе, что, несомненно, повышает читаемость кода и упрощает его поддержку.

ЗАКЛЮЧЕНИЕ

В диссертационной работе решены следующие задачи:

1. В среде языково-ориентированного программирования разработан текстовый язык и инструментальное средство для автоматного программирования. Для автоматов, написанных на этом языке, при необходимости обеспечивается автоматическое построение диаграммы состояний, а не наоборот, что важно для профессиональных программистов, привыкших к текстовому представлению программ.
2. Разработаны средства валидации автоматов для среды языково-ориентированного программирования.
3. Разработан акторный язык и инструментальное средство для многопоточного программирования, и выполнена их интеграция с языком и инструментальным средством для автоматного программирования. Таким образом, обеспечена поддержка многопоточного автоматного программирования.
4. Результаты работы внедрены при создании в среде языково-ориентированного программирования *MPS* коммерческой системы учета программных дефектов *YouTrack* в компании *JetBrains* (Санкт-Петербург).

Все полученные результаты обладают новизной, так среда языково-ориентированного программирования *MPS* построена на новых принципах создания языков программирования и является первой в мире промышленной системой этого класса.

ЛИТЕРАТУРА

1. **Cai K., Cangussu J. W., DeCarlo R. A., Mathur A. P.** An overview of software cybernetics /Software Technology and Engineering Practice. 2003, pp. 77 – 86.

2. **Harel D.** Statecharts: A visual formalism for complex systems //Sci. Comput. Program. 1987. №. 8, с. 231–274.

3. **Ахо А., Ульман Дж.** Теория синтаксического анализа, перевода и компиляции (Том 1. Синтаксический анализ). М: Мир, 1978.

4. **Непейвода Н. Н.** Стили и методы программирования. Курс лекций. Учебное пособие. М.: Интернет-университет информационных технологий, 2005.

5. **Шалыто А. А.** Программная реализация управляющих автоматов. //Судостроительная промышленность. Сер. «Автоматика и телемеханика». 1991. Вып.13, с. 41, 42.

6. **Шалыто А. А.** Парадигма автоматного программирования //Научно-технический вестник СПбГУ ИТМО. Автоматное программирование. 2008. Вып. 53, с. 3–23.

7. **Поликарпова Н., Шалыто А.** Автоматное программирование. СПб: Питер, 2009.

8. **Шалыто А. А.** Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.

9. **Shalyto A. A.** Technology of Automata-Based Programming, 2004. http://www.codeproject.com/KB/architecture/abp.aspx

10. International Standard IEC 1131-3. Programmable Controllers. Part 3. Programming languages. s.l. : International Electrotechnical Commission. 1993.

11. **Петров И. В.** Программируемые контроллеры. Стандартные языки и приемы прикладного проектирования. М : СОЛОН-Пресс, 2004.

12. **Керниган Б., Ритчи Д.** Язык программирования *C*. М.: Вильямс, 2009.

13. **Страуструп Б.** Язык программирования C++. Специальное издание. СПб.: Бином, Невский Диалект, 2008.

14. **Eckel B.** Thinking in Java (4th Edition). NJ: Prentice Hall PTR, 2005.

15. **Armstrong J.** Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf, 2007.

16. **Душкин Р. В.** Справочник по языку Haskel. М.: ДМК Пресс, 2008.

17. **Фултон Х.** Программирование на языке Ruby. М.: ДМК Пресс, 2007.

18. **Бибило П. Н.** Основы языка VHDL. М.: Либроком, 2009.

19. **Шалыто А. А.** Реализация алгоритмов логического управления программами на языке функциональных блоков //Промышленные АСУ и контроллеры. 2000. № 4, с. 45–50.

20. **Colgren R.** Basic Matlab, Simulink And Stateflow (Aiaa Education Series). American Institute of Aeronautics & Ast, 2006.

21. **Поршнев С. В.** Matlab 7. Основы работы и программирования. М.: Бином-Пресс, 2009.

22. **Головешин А.** Конвертор Visio2Switch. http://is.ifmo.ru/progeny/visio2switch.

23. **Бьяфоре Б.** Microsoft Visio 2007. Библия пользователя. М.: Вильямс, 2009. .

24. **Канжелев С. Ю., Шалыто А. А.** Преобразование графов переходов, представленных в формате MS Visio, в исходные коды программ для различных языков программирования (инструментальное средство MetaAuto). Проектная документация. СПбГУ ИТМО, 2006. http://is.ifmo.ru/projects/metaauto/.

25. **Столяров Л. В.** Трансляция описаний автоматов, представленных в формате Microsoft Visio, в исходный код на языке С //Компьютерные инструменты в образовании. 2009. №5, с. 35 – 44.

26. **Мазин М. А., Гуров В. С., Нарвский А. С., Шалыто А. А.** UniMod: Метод и средство разработки реактивных объектно-ориентированных программ с явным выделением состояний /Труды Второй Всероссийской научной конференции «Методы и средства обработки информации». МГУ. 2005, с. 361–366.

27. **Грехем И.** Объектно-ориентированные методы. Принципы и практика. М: Вильямс, 2004.

28. **Гуров В. С.** Технология проектирования и разработки объектно-ориентированных программ с явным выделением состояний (метод, инструментальное средство, верификация). Диссертация на соискание ученой степени кандидата технических наук. СПбГУ ИТМО, 2008.

29. **Гуров В. С., Мазин М. А., Шалыто А. А.** UNIMOD — программный пакет для разработки объектно-ориентированных приложений на основе автоматного подхода /Труды XI Всероссийской научно-методической конференции «Телематика-2004». Т. 1. СПбГУ ИТМО. 2004, с. 189–191.

30. **Гуров В. С., Мазин М. А., Шалыто А. А.** Операционная семантика UML-диаграмм состояний в программном пакете UniMod /Труды XII Всероссийской научно-методической конференции «Телематика-2005». Т. 1. СПбГУ ИТМО. 2005, с. 74–76.

31. **Гуров В. С., Мазин М. А., Зубок Д. А., Парфенов В. Г., Шалыто А. А.** Два подхода к созданию программ с использованием инструментального средства UniMod /Труды XIV Всероссийской научно-методической конференции «Телематика-2007». СПбГУ ИТМО. 2007, с. 428.

32. **Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.** UML. SWITCH-Технология. Eclipse //Информационно-управляющие системы. 2005. № 6, с. 12–17. http://is.ifmo.ru/works/uml-switch-eclipse/

33. **Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.** Разработка средств автоматизации построения объектно-ориентированных программ с явным выделением состояний //Научно-технический вестник СПбГУ ИТМО. Актуальные

проблемы современных оптико-информационных систем и технологий. 2004. Вып. 16, с. 88 – 100.

34. **Гуров В. С., Мазин М. А., Шалыто А. А.** Автоматическое завершение ввода условий в диаграммах состояний //Информационно-управляющие системы. 2008. №1, с. 24 – 33.

35. **Мазин М. А., Гуров В. С., Шалыто А. А.** Операционная семантика UML-диаграмм состояний в программном пакете UniMod /Труды XII Всероссийской научно-методической конференции «Телематика-2005». Т. 1. СПбГУ ИТМО. 2005, с. 74–76.

36. **Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.** Исполняемый UML. Проект UniMod. /Software Engineering Conference (Russia). РусСофт. 2005.

37. **Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.** Проект с открытым кодом UniMod – инструментальное средство для автоматного программирования на платформе Eclipse. /Open Source Forum. РусСофт, 2005.

38. **Гуров В. С., Мазин, М. А., Шалыто А. А.** UniMod – инструментальное средство для автоматного программирования //Научно-технический вестник СПбГУ ИТМО. Фундаментальные и прикладные исследования информационных систем и технологий. 2006. Вып. 30, с. 32–44.

39. **Мазин М. А., Гуров В. С.** Создание системы автоматического завершения ввода с использованием пакета UniMod /Вестник II межвузовской конференции молодых ученых. Т.1. СПбГУ ИТМО. 2005, с. 73–87.

40. **Gurov V., Mazin M., Narvsky A., Shalyto A.** UniMod: Method and Tool for Development of Reactive Object-Oriented Programs with Explicit States Emphasis /Proceedings 2005 of St.Petersburg IEEE Chapters «110 Anniversary» of Radio Invention. 2005. Vol. II, pp. 106–110.

41. **Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.** Инструментальное средство для поддержки автоматного программирования //*Программирование.* 2007. № 6, с. 65–80.

42. **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

43. **Гуров В. С., Яминов Б. Р.** Технология верификации автоматных моделей программ без их трансляции во входной язык верификатора. //Научное программное обеспечение в образовании и научных исследованиях. СПбГПУ. 2008, с. 36–40.

44. **Лукин М. А., Шалыто А. А.** Автоматизация верификации визуальных автоматных программ /Материалы XV Международной научно-методической конференции «Высокие интеллектуальные технологии и инновации в образованиии науке». СПбГПУ. 2008, с. 296, 297.

45. **Harel D., Naamad A.** The STATEMATE semantics of statecharts //ACM Trans. Softw. Eng. Methodol. Vol. 5. 1996. № 4, pp. 293–333.

46. **Малаховский Я. М., Шалыто А. А.** Реализация конечных автоматов на функциональных языках программирования //Информационно-управляющие системы. 2009. № 6, с. 30 – 33.

47. **Степанов О. Г., Шалыто А. А., Шопырин Д. Г.** Предметно-ориентированный язык автоматного программирования на базе динамического языка Ruby //Информационно-управляющие системы. 2007. № 4, с. 22–27.

48. **Тимофеев К. И., Астафуров А. А., Шалыто А. А.** Наследование автоматных классовс использованием динамических языковпрограммирования (на примере языка RUBY) //Информационно управляющие системы. 2009. № 4, с. 21 – 25.

49. **Frankel D.** Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley, 2003.

50. **Чарнецки К., Айзенекер У.** Порождающее программирование. Методы, инструменты, применение. Для профессионалов. СПб.: Питер, 2005.

51. **Surhone L., Timpledon M., Marseken S.** Template Metaprogramming: Metaprogramming, Generic Programming, Compiler, Compile-Time

Execution, C+ +, Curl Programming Language. Betascript Publishing, 2010.

52. **Simonyi C.** The death of computer languages, the birth of Intentional Programming //Microsoft Research. 1995. http://citeseer.nj.nec.com/simonyi95death.html

53. **Ward M.** Language Oriented Programming //Software – Concepts and Tools. 1994. № 15.

54. **Fowler M.** Language Workbenches: The Killer-App for Domain-Specific Languages, 2005.

55. **Цимбал А.** Технология CORBA для профессионалов. СПб.: Питер, 2001.

56. **Троелсен Э.** C# и платформа .NET. СПб.: Питер Пресс, 2007.

57. **Laguna M., Marklund J.** Business Process Modeling, Simulation and Design. Prentice Hall, 2004.

58. **Sobel J., Friedman D.** An Introduction to Reflection-Oriented Programming. 1996.

59. **Wagner W., Hilken R.** XML: Introduction to Applied XML–Technologies in Business. Prentice Hall, 2002.

60. **Fowler M.** Crossing Refactoring's Rubicon //ThoughtWorks. 2001.

61. **Дмитриев С.** Языково-ориентированное программирование: следующая парадигма //RSDN Magazine. 2005. № 5, http://rsdn.ru/article/philosophy/LOP.xml

62. **Wirth N.** Program Development by Stepwise Renement //Comm. ACM. 1971. 14, pp. 221–227.

63. **Дейт К.** Введение в системы баз данных. М.: Вильямс, 2006.

64. **Гойвертс Я., Левитан С.** Регулярные выражения. Сборник рецептов. СПб.: Символ-Плюс, 2010.

65. **Kay M.** XPath 2.0 Programmer's Reference (Programmer to Programmer). Wrox, 2004.

66. **Solomatov K.** DSL Adoption with JetBrains MPS //DZone, Architect Zone. 2009.

67. **Fowler M.** A Language Workbench in Action – MPS //ThoughtWorks. 2005. http://www.martinfowler.com/articles/mpsAgree.html

68. **Fowler M.** *PostIntelliJ* //ThoughtWorks. http://martinfowler.com/bliki/PostIntelliJ.html.

69. **Давыдов С., Ефимов А.** IntelliJ IDEA. Профессиональное программирование на Java. Наиболее полное руководство. СПб.: БХВ-Петербург.

70. **Карлсон Д.** Eclipse. М.: Лори, 2008.

71. **Gronback R.** Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley Professional, 2009.

72. **Surhone L., Tennoe M., Henssonow S.** Oslo (Microsoft). Betascript Publishing, 2010.

73. **Ахо А., Лам М., Сети Р., Ульман Дж.** Компиляторы. Принципы, технологии и инструментарий. М.: Вильямс, 2008.

74. **Инглиш Дж.** Macromedia Flash 8. М.: Эком, 2007.

75. **Мазин М. А., Шалыто А. А.** Анимация. Flash-технология. Автоматы //Компьютерные инструменты в образовании. 2003. № 4, с. 39–47.

76. **Мазин М. А., Парфенов, В.Г., Шалыто А. А.** Автоматная реализация интерактивных сценариев образовательной анимации /Труды X Всероссийской научно-методической конференции «Телематика-2003». СПбГИТМО (ТУ), 2003.

77. **Мазин М. А., Шалыто А.** А. Macromedia Flash и автоматы //Мир ПК. Диск. 2004. № 2.

78. **Мазин М. А., Шалыто А. А.** Преступники и автоматы //Мир ПК. 2004. № 9, с. 82–84.

79. **Мазин М. А., Шалыто А. А.** Анимация. Flash-технология. Автоматы //Информатика. 2006. № 11, с. 36–47.

80. **Мазин М. А.** Методы валидации автоматных моделей. Магистерская работа. СПбГУ ИТМО, 2006.

81. **Adve S., Adve V., Agha G., Frank M., Garzarán M., Hart J., Hwu W., Johnson R., Kale L., Kumar R., Marinov D., Nahrstedt K., Padua D., Madhusud.** Parallel Computing Research at Illinois: The UPCRC Agenda. University of Illinois at Urbana-Champaign. 2008.

82. **Sutter H.** The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software //Dr. Dobb's Journal. 2005. Vol. 30. № 3, pp. 202–210. http://www.gotw.ca/publications/concurrency-ddj.htm

83. **Воеводин В. В., Воеводин Вл. В.** Параллельные вычисления. СПб.: БХВ-Петербург, 2004.

84. **Sebesta R. W.** Concepts of Programming Languages. MA: Addison-Wesley Longman Publishing Co., Inc., 2001.

85. **Godfrey M.** First Draft of a Report on the EDVAC //IEEE Annals of the History of Computing. Vol. 15. 1993, № 4, pp. 28–75.

86. **Эндрюс Г.** Основы многопоточного, параллельного и распределенного программирования. М.: Вильямс, 2003.

87. **Trinder P., Mattson J., Partridge A., Jones S., Hammond K.** GUM: a portable parallel implementation of Haskell. 1996.

88. **Fowler M.** Closure. 2004. http://martinfowler.com/bliki/Closure.html

89. **Duffy J., Essey E.** PARALLEL LINQ: Running Queries On Multi-Core Processors //MSDN Magazine. Microsoft. 2007.

90. **Hillis W., Steele G.** Data Parallel Algorithms //Commun. ACM. Vol. 29. 1986. № 12, pp. 1170–1183.

91. **Лагунов И. А.** Разработка текстового языка автоматного программирования и его реализация для инструментального средства UniMod на основе автоматного подхода. Бакалаврская работа. СПбГУ ИТМО, 2008.

92. **Шамгунов Н. Н., Корнеев Г. А., Шалыто А. А.** State Machine – расширение языка Java для эффективной реализации автоматов // Информационно-управляющие системы. 2005. № 1, с. 16–24.

93. SMC – State Machine Compiler. http://smc.sourceforge.net/

94. **Цымбалюк Е. А.** Текстовый язык автоматного программирования TABP. Магистерская работа. 2008.

95. **State Chart XML (SCXML)**: State Machine Notation for Control Abstraction. W3C, 2008. http://www.w3.org/TR/2008/WD-scxml-20080516/

96. **Thurston A.** Parsing Computer Languages with an Automaton Compiled from a Single Regular Expression //In 11th International Conference on Implementation and Application of Automata (CIAA 2006). Taipei, Taiwan: 2006. Lecture Notes in Computer Science. Vol. 4094, pp. 285, 286.

97. **Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.** Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2007.

98. **Гуров В. С., Мазин М. А., Шалыто А. А.** Текстовый язык автоматного программирования //Научно-технический вестник СПбГУ ИТМО. Фундаментальные и прикладные исследования информационных систем и технологий. 2007. Вып. 42, с. 29–32.

99. **Мазин М. А., Гуров В. С., Шалыто А. А.** Текстовый язык автоматного программирования //Труды второй международной научной конференции «Компьютерные науки и информационные технологии». Саратов: СГУ. 2007, с. 33 – 35.

100. **Luo Z.** Computation and Reasoning: A Type Theory for Computer Science. Oxford University Press, 1994.

101. **Gosling J., Joy B., Steele G., Bracha G.** The Java Language Specification. Addison Wesley, 2005.

102. **Кнут Д.** Искусство программирования. Том 1. Основные алгоритмы. М.: Вильямс, 2008.

103. **Наумов Л. А., Шалыто А. А.** Искусство программирования лифта. Объектно-ориентированное программирование с явным выделением состояний //Мир ПК. Диск. 2004. № 2.

104. **Мейер Э.** CSS. Каскадные таблицы стилей. Подробное руководство. СПб.: Символ-Плюс, 2008.

105. **Конопко К. С.** Helgins: универсальный язык для написания анализатора типов //Компьютерные инструменты в образовании. 2007. № 4.

106. **Шалыто А. А., Туккель Н. И.** SWITCH-технология – автоматный подход к созданию программного обеспечения «реактивных» систем //Программирование. 2001. № 5, с. 45–62.

107. **Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.** Алгоритмы. Построение и анализ. М.: Вильямс. 2010.

108. **Hopcroft J., Motwani R., Ullman J.** Introduction to Automata Theory, Languages and Computation. NJ: Pearson Addison-Wesley, 2007.

109. **Верещагин Н., Шень А.** Лекции по математической логике и теории алгоритмов. Часть 2. Языки и исчисления. М.: МЦНМО, 2008.

110. **Верещагин Н., Шень А.** Начала теории множеств. Математическая логика и теория алгоритмов. М.: МЦНМО, 2008.

111. **Hewitt C., Bishop P., Steiger R.** A Universal Modular ACTOR Formalism for Artificial Intelligence. 1973, pp. 235–245.

112. **Odersky M., Spoon L., Venners B.** Programming in Scala: A Comprehensive Step-by-step Guide. Artima Inc, 2008.

113. **Мазин М. А., Жукова А. Р.** Акторное расширение языка Java в среде MPS. СПбГУ ИТМО, 2009.

114. **Мазин М. А., Жукова А. Р.** Акторное расширение языка Java в среде MPS //Научно-технический вестник СПбГУ ИТМО . 2010. Вып. 66, с. 72 – 77.

115. **Lewis B., Berg D.** Multithreaded Programming with Java Technology. Prentice Hall, 1999.

116. **Goets B.** Java theory and practice: Thread pools and work queues //developerWorks. 2002. http://www.ibm.com/developerworks/java/library/j-jtp0730.html

117. **Гуров В. С., Новиков Б. А., Горшкова Е. А., Белов Д. Д., Спиридонов С. В.** Моделирование контроллера Web-приложений с использованием UML //Программирование. 2005. № 1, с. 44–51.

118. **Reese G.** Cloud Application Architectures: Building Applications and Infrastructure in the Cloud. O'Reilly Media, 2009.

119. **Никитин П. А.** Применение генеративного программирования для создания объектной базы. Диплом. СПб : СПб ГУ, 2010.