Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

«Текстовый язык автоматного программирования ТАВР»

Автор: Цымбалюк Е. А.

Научный руководитель: Шалыто А. А.

Санкт-Петербург
2008

Содержание

Введение

В современном мире информационных технологий для разработки программного обеспечения, как правило, используются объектно-ориентированные языки программирования (такие как *C++*, *Java* или *C#*) . При этом существует целый ряд задач, к которым предъявляются высокие требования по качеству программного обеспечения: встроенные системы и системы реального времени.

Одним из эффективных подходов к решению подобных задач является автоматное программирование. Это стиль программирования, основанный на применении конечных автоматов для описания поведения программ, или программирование с явным выделением состояний [1]. Однако использование автоматного программирования в рамках «стандартных» объектно-ориентированных языков программирования затруднено тем, что эти языки позволяют оперировать понятием «объект», но не предоставляют никаких языковых средств для работы с состоянием объекта.

В настоящее время существует достаточно большое количество графических сред, позволяющих создавать и редактировать графы переходов автоматов, тем самым упрощая применение автоматного программирования. Одним из примеров подобных графических сред является среда разработки *UniMod* [2]. Это надстройка для среды разработки *Eclipse* [3], позволяющая строить графы переходов автоматов, интерпретировать и компилировать автоматные программы в один из объектно-ориентированных языков (в частности язык *Java*), проверять корректность построения графов переходов, а также отлаживать созданную программу непосредственно в среде разработки.

Графические среды являются удобным средством автоматного программирования. Однако процесс программирования в графической системе организован иначе, чем при программировании с использованием текстового языка. Сам принцип написание программы не в виде текста, но в виде графического изображения требует от программиста иных навыков и иного способа мышления. Это в определенном смысле препятствует развитию автоматного программирования.

В рамках данной работы рассматриваются уже существующие текстовые языки программирования. На основе обзора формируются требования к текстовым языкам программирования, и на основе сформулированных требований предлагается новый текстовый язык автоматного программирования *TABP*, отвечающий поставленной задаче.

1. Обзор текстовых языков автоматного программирования

В данной главе представлен обзор текстовых языков автоматного программирования, а также рассмотрены проблемы, возникающие при реализации конечных автоматов на текстовых языках. Для выявления ключевых задач, встающих перед текстовыми языками автоматного программирования. Далее рассматривается реализация конкретного автомата кнопки лифта. Помимо специализированных языков автоматного программирования в данной главе рассматривается «классический» язык программирования общего назначения *C++*.

1.1. Пример конечного автомата

В качестве примера выбран конечный автомат, реализующий поведение кнопки вызова лифта (рис. 1). В начальном состоянии кнопка лифта отжата, автомат находится в состоянии `Released`. При нажатии на кнопку (событие `push`) происходит вызов лифта (действие `call`), и автомат переходит в нажатое состояние `Pushed`. Кнопка остается нажатой до тех пор, пока не приедет лифт (так поступают кнопки многих лифтов Санкт-Петербурга). Функция `isReady()` возвращает булево значение «истина», если лифт приехал, иначе – «ложь». После приезда лифта автомат отпускает кнопку (действие `release()`) и переходит в исходное состояние `Released`.

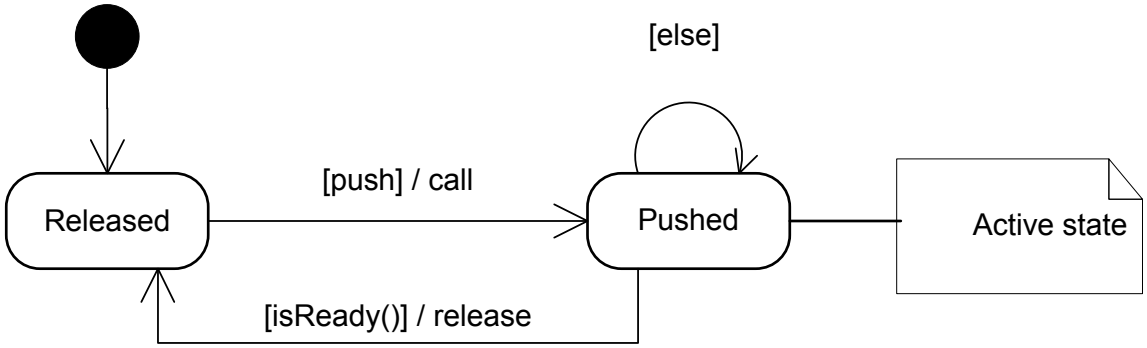

Рис. 1. Конечный автомат кнопки лифта

Отметим, что состояние `Released` является пассивным – при переходе в данное состояние автомат теряет управление. Управление возвращается обратно при возникновении нового события.

В отличие от состояния `Released`, состояние `Pushed` является активным, то есть оно не теряет управление. На рис. 1 автомат в состоянии `Pushed` проверяет значение `isReady()` в цикле до тех пор, пока не приедет лифт.

1.2. **Язык *SMC***

В языке *SMC* (State Machine Compiler, [4]) автомат описывается в виде набора состояний и переходов. Используя диаграмму переходов автомата, можно легко построить соответствующую программу на *SMC*. Однако язык *SMC* предоставляет лишь базовые механизмы описания автомата: язык позволяет описывать действия на переходе, действия при входе/выходе в состояние и т.д., но в рамках языка невозможно описывать подавтоматы и активные состояния.

Таким образом, для реализации автомата кнопки лифта необходимо модифицировать исходную диаграмму переходов, так как в языке *SMC* недостаточно языковых средств, позволяющих описать активное состояние. Существует несколько вариантов решения:

- Сделать состояние `Pushed` пассивным. Для этого необходимо заменить обработку функции `isReady()` на событие `ready`, посылаемое готовым лифтом.
- Эмулировать активное состояние. Для этого автомату необходимо посылать самому себе вспомогательное событие, при обработке которого проверяется значение `isReady()`, и осуществляются необходимые действия.
- Перенести работу из состояния `Pushed` в действие при переходе. В этом случае состояние `Pushed` перестает быть нужным: состоянию `Released` достаточно единственной петли, имеющей цикл ожидания в качестве действия на переходе.
- Для реализации автомата выберем первый вариант, как наиболее подходящий для языка *SMC* (рис. 2, листинг 1).

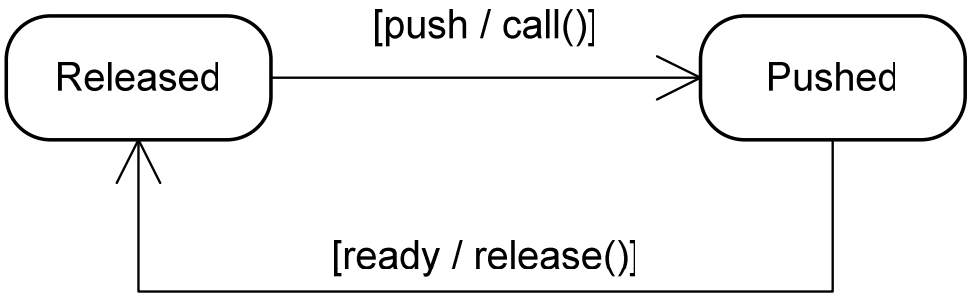

Рис. 2. Модифицированный конечный автомат кнопки лифта

Второй и третий варианты решения не являются естественными для автоматного программирования. В частности третий вариант не удовлетворяет правилу: действия на переходах (входе/выходе в состояние) являются атомарными, то есть с точки зрения автомата выполняются мгновенно. Однако цикл ожидания `isReady()` имеет неопределенное время работы и должен иметь явно выделенное состояние.

Листинг 1. Реализация автомата кнопки лифта на языке *SMC*

```
Released {
     push Pushed {
          call();
     }
}
Pushed {
     ready Released {
          release();
     }
}
```

1.3. Язык *State Machine*

Язык *State Machine* является расширением языка *Java* и основан на одноименном паттерне *State Machine* [5], расширяющем паттерн State [20].

Аналогично языку *SMC*, язык *State Machine* рассчитан на пассивные состояния, то есть в рамках языка невозможно реализовать исходный пример без изменений. Реализуем автомат кнопки лифта, используя модифицированный автомат, приведенный на рис. 2.

В языке *State Machine* автоматы являются объектами в смысле объектно-ориентированного программирования. Благодаря этому не возникает проблем при организации взаимодействия автоматов. Однако язык *State Machine* не позволяет описывать подавтоматы, а также синтаксис языка достаточно громоздок. Каждое состояние автомата является отдельный классом *Java*, а автомат – множеством определенных состояний и переходов между ними. При таком дизайне программы возникает проблема с общими данными автомата (являющимися общими для всех состояний). Каждое состояние, будучи самостоятельным классом, должно иметь метод-конструктор, сохраняющий указатель на объект данных автомата.

Еще одним требованием языка *State Machine* является то, что автомат и все состояния должны реализовывать общий интерфейс, предоставляющий связь автомата с внешним миром. В случае нашего примера данный интерфейс (назовем его `IButton`) должен содержать методы `push` и `ready` для того, чтобы внешняя система могла сообщить автомату о возникновении соответствующего события.

В языке *State Machine* для осуществления переходов используются «события» (в терминах языка *State Machine*), которые передаются в качестве аргумента метода `castEvent`. Данный метод выполняет переход в соответствии с таблицей переходов, которая задается при объявлении автомата.

Заметим, что данный механизм переходов автомата является несколько необычным с точки зрения автоматного программирования. События, используемые для переходов,

могут быть аналогами событий `push` и `ready`, однако интерфейс `IButton` уже содержит соответствующие методы. К тому же метод `castEvent` вызывается изнутри автомата, то есть при выполнении методов `push` и `ready`. Выходит, что у внешней системы нет возможности использовать данные «события», как события в понимании автоматного программирования. Получается, что «события» `PUSH` (предназначенное для перехода в состояние `Pushed` при возникновении события `push`) и `READY` (предназначенное для перехода в состояние `Released` при возникновении события `ready`) являются лишь механизмом перехода в другое состояние. Было бы логично в таком случае не создавать лишнюю сущность, а непосредственно задать имя состояния, в которое будет осуществлен переход автомата, но синтаксис *State Machine* таков, какой он есть…[1]

Ниже предложен код реализации модифицированного конечного автомата кнопки лифта (листинг 2).

Листинг 2. Реализация модифицированного автомата кнопки лифта на языке *State Machine*

```
public interface IButton {
     void push();
     void ready();
}

public state Released implements IButton events PUSH {
     public void push() {
          call();
          castEvent(PUSH);
     }
     public void ready() {}
}

public state Pushed implements IButton events READY {
     public void push() {}
     public void ready() {
          release();
          castEvent(READY);
     }
}

public automaton Button implements IButton {
     state Released released  (PUSH -> pushed);
     state Pushed pushed      (READY -> released);
```

[1] К сожалению, нам пришлось остановиться более подробно на деталях синтаксиса языка *State Machine*, однако в рамках данного обзора крайне познавательно рассмотреть, какие трудности возникают при проектировании текстового языка автоматного программирования. В частности, какие результаты могут быть получены при выборе неудачных дизайнерских решений: невозможность реализации простейшего автомата без существенных модификаций и (в случае языка *State Machine*) крайняя громоздкость.

```
    public Button() {
        released  @= new Released();
        pushed    @= new Pushed();
    }
}
```

1.4. **Язык *C++***

В отличие от рассмотренных выше языков, язык *C++* не является текстовым языком автоматного программирования, однако он является языком общего назначения и предназначен для реализации любых программ (в том числе и нашей :). Рассмотрим реализацию исходного автомата кнопки лифта на языке *C++*.

Для реализации события `push` воспользуемся механизмом вызова метода (так же, как это сделано в языке *State Machine*). Состояние автомата задает переменная `state`, обрабатываемая конструкцией `switch` [1]. Поскольку состояние `Released` – пассивное, оно должно возвращать управление внешней системе. Для передачи управления автомату достаточно вызвать вспомогательный метод `auto`, реализующий поведение автомата (листинг 3).

Листинг 3. Реализация автомата кнопки лифта на языке *C++*

```
class Button {
public:
    void push(void) {
        isPushed = true;
        auto();
        isPushed = false;
    }
private:
    enum State {
        Released,
        Pushed
    };
    State state;
    bool isPushed;
    void auto(void) {
        for (;;) {
            switch (state) {
            case Released:
                if (isPushed) {
                    call();
                    state = Pushed;
                    break;
                }
                return ;
            case Pushed:
                if (isReady()) {
                    release();
```

```
                        state = Released;
                    }
                    break;
                }
            }
        }
    };
```

Приведенный код программы на *C++* не такой компактный, как реализация на языке *SMC*, однако язык *C++* не является автоматным языком программирования. В программе передача управления и смена состояния автомата реализованы вручную. Также для обработки событий использована вспомогательная переменная `isPushed` (значение данной переменной изменяется при вызове метода `push()`, после чего автомату передается управление с помощью метода `auto()`). Тем не менее, текст представленной реализации намного понятнее аналогичной реализации на языке *State Machine*, и для реализации автомата не пришлось модифицировать исходный пример. Как ни странно, язык, который не разрабатывался специально для автоматного программирования, предоставляет все необходимые средства, позволяющие *беспрепятственно* реализовать поставленную задачу.

1.5. **Язык автоматного программирования**

В работе [8] представлен новый текстовый язык автоматного программирования, созданный на базе системы *MPS* [9, 11]. Данный язык позволяет описывать автоматы для реактивных систем. В работе [12] приведен пример использования данного языка для описания автоматов системы дорожного движения. Для реализации выбранного примера автомата воспользуемся модифицированной версией автомата (рис. 2), аналогично примеру с языком *SMC*. Отметим, что в работе [12] для решения схожей проблемы введено специальное событие `e0` (единственное для всей системы дорожного движения), возбуждаемое внешней системой с периодом времени `dt`.

```
statemachine Button {
    initial state s0 {
        transiteto Released;
    }
    state Released {
        on push() execute call() transiteto Pushed;
    }
    state Pushed {
        on ready() execute release() transiteto Released;
    }
}
```

Отметим, что язык В.С. Гурова и М.А. Мазина позволяет описывать вложенные состояния автомата, причем состояния верхнего уровня могут иметь собственные обработчики событий, что позволяет описывать обобщенные переходы для группы состояний. Помимо языковых возможностей представленный язык привязан к конкретной системе разработки *MPS*, однако имеет встроенную систему визуализации описываемого автомата, позволяя программисту сразу увидеть разрабатываемый автомат. Данный язык находится в стадии разработки и может измениться с момента написания данного текста.

1.6. **Текстовый язык автоматного программирования (*TABP)***

В данной главе рассмотрены два текстовых языка автоматного программирования и язык общего назначения на примере реализации конечного автомата, описывающего поведение кнопки лифта.

Несмотря на то, что первые два языка являются, по сути, специализированными языками, они не позволили реализовать исходный конечный автомат без существенных модификаций. Существуют ситуации, в которых автомат может иметь не только пассивные, но и активные состояния (в том числе и в «реактивных» системах [6]), а также ситуации, при которых автомату необходимо выполнить несколько переходов подряд (то есть ситуации, когда автомат переходит в пассивное состояние, но при этом не теряет управление). Как было отмечено выше, существуют действия, которые нельзя считать атомарными, для которых естественно выделить отдельное состояние или даже множество состояний.

Язык *C++,* хотя и не является языком автоматного программирования, позволяет реализовать исходный пример без изменений. Данный факт крайне важен, так как подчеркивает, что на текущий момент языки общего назначения являются лучшей альтернативой существующим текстовым языкам автоматного программирования. В качестве недостатков реализации на *C++*, можно отметить то, что организация механизмов передачи управления автомата осуществлялась вручную (использовались оператор `return`, прекращающий выполнение процедуры `auto`, и переменна `state`, позволяющая восстановить пассивное состояние автомата). Также, обработка событий была осуществлена через вызов метода и вспомогательную переменную. Заметим, что можно расширить механизм посылки сообщений для повышения удобства и универсальности, но все равно это останется ручной реализацией, а не особенностью языка *C++*.

Среди текстовых языков автоматного программирования должны быть представлены языки, сочетающие в себе набор инструментов, позволяющих описывать

широкий класс автоматов (поддержка пассивных и активных состояний, подавтоматов и т.д.). Также, неотъемлемым свойством данных языков должно быть то, что механизмы автоматного программирования (такие, как передача событий, сохранение и восстановление контекста работы автомата при переходе в пассивное состояние и т.д.) являются частью языка. В языке *TABP* [7] предпринята попытка создания языка такого типа. Ниже представлена реализация конечного автомата кнопки лифта на языке *TABP* (листинг 4).

Листинг 4. Реализация автомата кнопки лифта на языке *TABP*

```
auto Button {
     fixed Released {
          on push {
               call();
               goto Pushed;
          }
     }
     loop Pushed {
          if (isReady()) {
               release();
               goto Released;
          }
     }
}
```

TABP предоставляет набор языковых элементов, комбинация которых позволяет описать требуемый конечный автомат. Такими элементами являются: шаги `step`, операции перехода, механизм обработки событий, механизм сохранения контекста выполнения автомата при переходе в пассивное состояние, а также конструкции (так называемый, «синтаксический сахар»[2]) `fixed` и `loop`, позволяющие описать более компактно наиболее часто встречающиеся комбинации элементов языка. В частности, приведенную реализацию автомата можно переписать без использования `fixed` и `loop` (листинг 5).

Листинг 5. Реализация автомата кнопки лифта на языке *TABP* без использования конструкций `fixed` и `loop`

```
auto Button {
     step Released {
          on push {
               call();
               goto Pushed;
          }
          relax;
```

[2] http://ru.wikipedia.org/wiki/Синтаксический_сахар

```
        }
        step Pushed {
            if (isReady()) {
                release();
                goto Released;
            }
            repeat;
        }
    }
```

Отметим, что *TABP* является языком общего назначения, помимо автоматных конструкций *TABP* предоставляет те же инструменты, что и «классические» объектно-ориентированные языки (классы, методы, процедуры и т.д.). По нашему мнению, текстовые языки автоматного программирования должны не просто позволить программировать с явным выделением состояний, но и *расширить* объектно-ориентированное программирование новой парадигмой.

2. Элементы языка *TABP*

В данной главе рассматриваются отдельные элементы *TABP*, и обсуждается, почему было принято то или иное дизайнерское решение.

2.1. Язык общего назначения

Важной особенностью языка *TABP*, является то, что *TABP* – самостоятельный язык общего назначения, а не расширением одного из уже существующих языков (таких как *Java* или *C#*, как это сделано в случае со *State Machine* [5] или в работе [13]).

Оба подхода имеют свои достоинства и недостатки. Язык-расширение наследует все свойства расширяемого языка. То есть, он имеет обратную совместимость с оригинальным языком (программы, написанные на оригинальном языке, автоматически являются программами на расширенном языке). С точки зрения совместимости со старыми проектами, подход с расширением существующего языка имеет серьезное преимущество. Однако язык-расширение имеет и недостатки. При практическом использовании языка основным языком программирования остается оригинальный язык, а *специализированное* расширение используется лишь в тех случаях, когда без него какие-то вещи сделать гораздо неудобнее и сложнее (как в случае с расширением *LINQ*, используемом для формирования *SQL*-запросов [14]).

Одной из основных задач при создании *TABP* является создание текстового языка автоматного программирования, позволяющего применять парадигму автоматного программирования [15, 16] как базовую модель для всего проекта в целом. Создание специализированного расширения изначально ставит перед собой иные задачи – решение частных задач, наиболее характерных для автоматного программирования.

Таким образом, способ расширения существующего языка программирования не позволяет решить те задачи, которые ставятся перед разрабатываемым языком. Поэтому было принято решение о создании самостоятельного языка программирования общего назначения, в основе которого лежат идеи автоматного программирования.

2.2. Совместимость с *.NET*

Как говорилось в предыдущем параграфе, серьезным недостатком нового самостоятельного языка программирования является отсутствие обратной совместимости с уже существующими проектами, реализованными на другом языке программирования. Данную проблему позволяет решить технология *.NET* [17, 18]. Система *.NET* изначально

проектировалась как кросс-языковая платформа. Это достигается посредством использования единого промежуточного языка программирования *MSIL* (*Microsoft Intermediate Language*) [19]. Все языки, исполняемые под платформой *.NET*, транслируются в промежуточный язык. Таким образом, вне зависимости от языка программирования, код программы преобразуется в единый формат, и проект (или часть проекта), написанный на одном языке, может беспрепятственно использоваться с проектом, реализованном на другом языке.

При разработке языка *TABP* было принято решение сделать язык *.NET*-совместимым, для обеспечения совместимости со всем языками, реализованными под платформу *.NET*. Для обеспечения такой совместимости все языки программирования *.NET* должны поддерживать некоторый набор базовых элементов объектно-ориентированного программирования (такие, как наследование, защищенный доступ к переменным и т. д.), а также набор элементов, специфичных для платформы *.NET* (интерфейсы, делегаты, события и т. д.).

Стоит также отметить, что помимо совместимости с другими языками, платформа *.NET* предоставляет систему сборки мусора и стандартную библиотеку (вернее, множество библиотек, входящих в состав *.NET Framework*), а также кросс-платформенность в целом – возможность запуска программ, написанных на языке *TABP*, под множеством операционных систем, на различных компьютерах и портативных устройствах.

2.3. **Выразительные возможности языка**

В первой главе данной работы было выполнено сравнение нескольких текстовых языков автоматного программирования. Одним из недостатков рассмотренных язык является то, что программисту предоставляется возможность использовать лишь какую-то конкретную автоматную модель. На практике заданная модель может оказаться неудобной или даже непригодной (как в примере с кнопкой лифта – при наличии и пассивных, и активных состояний в рамках одного автомата). Поскольку язык *TABP* является языком общего назначения, он не должен иметь подобные недостатки. Соответственно, механизм описания автоматов не должен навязывать фиксированную модель. Для достижения данной цели выбран несколько иной принцип построения автомата, нежели в языках *SMC* и *State Machine*. В языке *TABP* автомат строится из составных «блоков» – элементов языка, позволяющих «выстраивать» автомат. Основными элементами построения автомата являются «шаги» (блоки **`step`**). Используя операторы перехода **`goto`** и **`relax`**, можно определить активные и пассивные состояния.

Примеры реализаций различных моделей автоматов представлены в главе 3. Элементы, из которых составляется автомат, подробно описаны в следующих параграфах.

2.4. **Активные и пассивные состояния**

Язык *TABP* позволяет описывать состояния двух типов: активные и пассивные. Отличие пассивного состояния заключается в том, что при переходе в пассивное состояние автомат теряет управление. Автомат снова получает управление при возникновении события, либо в случае явной передачи управления автомату при вызове встроенного метода **`auto`**.

Для выполнения требований к выразительности языка *TABP* необходимо поддерживать оба типа состояний, так как они предназначены для решения разных задач. Пассивные состояния используется в реактивных системах [6] – в системах, в которых действия автомата управляются внешними событиями. При возникновении события автомат получает управление, выполняет какие-то действия и возвращает управление возбудившей событие системе. Системы, позволяющие описывать только пассивные состояния (такие, как *SMC*), требуют наличие внешней системы, продуцирующей события, что накладывает определенные ограничения при разработке (с этими же ограничениями мы столкнулись при реализации примера с кнопкой лифта). Однако язык программирования не должен накладывать ограничения на реализуемую задачу. Если исходная система требует, чтобы автомат сам опрашивал управляемый объект [9] об изменившемся состоянии, то язык должен предоставить такую возможность – определить активное состояние.

В языке *TABP* для определения состояния используются блоки **`step`**, задающие так называемые «шаги»:

```
step StepName {
}
```

Шаг сам по себе представляет именованный блок кода. Если не использовать операторы перехода, то после выполнения тела шага, выполняется код, стоящий вслед за ним. Шаг лишь добавляет имя некоторому блоку кода, однако вместе с операторами перехода шаг задает состояние. В языке *TABP* существует два основных оператора перехода: **`goto`** и **`relax`**, использование которых определяет, является ли состояние пассивным или активным:

```
goto StepName;
relax StepName;
```

Оператор **goto** осуществляет переход в начало именованного шага. Соответственно, шаг, в который осуществляется переход, представляет собой активное состояние, так как после перехода автомат не теряет управления.

Для поддержки пассивных состояний в языке *TABP* введен оператор перехода **relax**. Данный оператор осуществляет переход аналогично **goto**, однако после перехода автомат переходит в пассивное состояние. Таким образом, состояние, в которое переходит автомат, является пассивным.

В языке *TABP* введены еще два типа шагов (так называемый «синтаксический сахар») блоки **loop** и **fixed**. Первый блок предназначен для описания активных состояний с петлей (рис. 3).

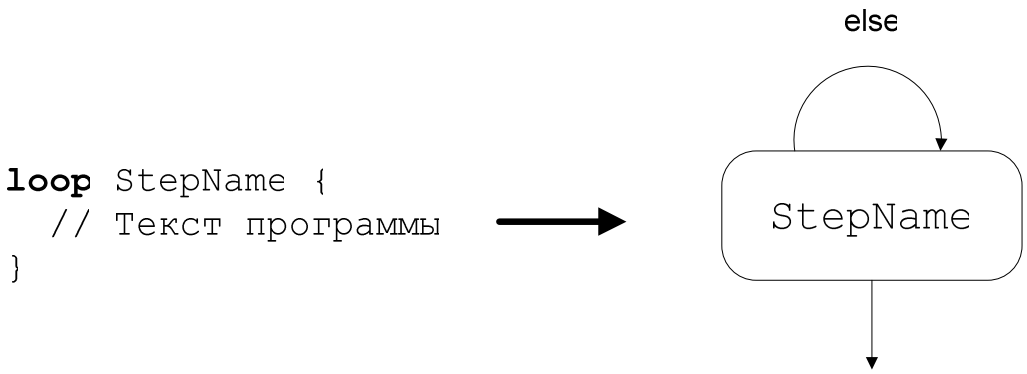

Рис. 3. Состояние с петлей, определяемое блоком `loop`

Данный блок аналогичен блоку **step** с оператором **goto** на конце:

```
step StepName {
     // Текс программы
     goto StepName;
}
```

Блок **fixed** аналогичен блоку **loop**, однако предназначен для описания пассивных состояний:

```
fixed StepName {
     // Текст программы
}
```

Блок **fixed** соответствует шагу с оператором **relax** в конце блока:

```
step StepName {
     // Текст программы
     relax StepName;
}
```

Блок **fixed** подходит для любых пассивных состояний, он удобен для обработки событий. После обработки события автомат переходит в пассивное состояние:

```
fixed {
      on Event1 {
      }
      on Event2 {
      }
      // …
}
```

Имя блока может быть опущено, так как не во всех ситуациях оно необходимо.

Стоит отметить, что данный способ описания состояний более «низкоуровневый», нежели способ описания состояний в языке *SMC*. Можно было бы ввести два типа блока «активное состояние» и «пассивное состояние», но данный подход ограничил бы программиста. В языке *TABP* программист может самостоятельно определять семантику поведения автомата и конкретных состояний. Данный подход позволяет создавать так называемые «гибридные» состояние, то есть состояния, которые в зависимости от внешней системы могут не возвращать управление, если этого требует логика системы.

2.5. **Операторы перехода**

Операторы перехода соответствуют стрелкам на диаграмме переходов автомата. Язык *TABP* предоставляет два основных оператора **goto** и **relax**, описанных выше. Данные операторы осуществляют переход к именованному шагу. Оператор **relax** может быть вызван без указания имени шага, тогда переход будет осуществлен к началу текущего блока:

```
// Аналогично блоку fixed
step { // Безымянное пассивное состояние
      relax ;
}
```

Для активных состояний аналогичную функцию выполняет оператор **repeat**:

```
// Аналогично блоку loop
step { // Безымянное активное состояние с петлей
      repeat ;
}
```

Помимо операторов, задающих петли, в рамках шага можно использовать оператор **break**, осуществляющий переход к концу блока:

```
step {
      break; // Переход к концу блока
}
```

Язык *TABP* предоставляет еще один интересный оператор **relax step**. Данный оператор также является синтаксическим сахаром, так как может быть выражен с помощью оператора **relax** и блока **step:**

```
relax step;
// аналогично
relax StepName;
step StepName;
```

При выполнении данного оператора автомат теряет управление. При возвращении управления автомату, выполнение начинается с оператора **relax step**. В качестве примера использования данного оператора можно привести пример с реализацией действий на входе в пассивное состояние. Дело в том, что при осуществлении перехода в состояние посредством оператора **relax**, автомат теряет управление сразу же после перехода:

```
step { // Пассивное состояние с действием на входе
     // Действие на входе
     relax step;
     // …
}
```

Помимо действий на входе данный оператор может быть использован при реализации итераторов [20]:

```
for (/*…*/) {
     // Итерация
     relax step;
}
```

2.6. **Условные операторы**

Поскольку *TABP* является языком общего назначения, он должен предоставлять возможность решать более широкий класс задач, нежели другие языки программирования. Язык *TABP* является *C*-подобным – имеет синтаксис похожий на синтаксис языка *C*. Соответственно, *TABP* предоставляет набор «стандартных» операторов, таких как операторы **if**, **switch**, циклы **for**, **while** и **do while** [21], имеющих аналогичную семантику поведения. Условные операторы могут быть использованы внутри блоков **step**, **loop**, **fixed**, что позволяет описывать действия автомата непосредственно в теле состояний.

2.7. **Взаимодействие автоматов**

Одним из важных вопросов разработки языка автоматного программирования является взаимодействие автоматов. Например, язык может быть построен так, что вся

программа в целом – это один большой автомат (как это сделано в *SMC*). При таком подходе увеличивается связность компонентов системы, так как нет четких языковых средств, для их логического разграничения (качество дизайна программы полностью зависит от опыта программиста). На сегодняшний день системы проектируются как объектно-ориентированные. В контексте автоматного программирования поведение каждого объекта описывается конечным автоматом [16]. Соответственно, разумно позволить описывать произвольное количество автоматов, а также предоставить языковые средства для обеспечения взаимодействия автоматов.

В языке *TABP* взаимодействия автоматов реализовано, на основе схемы взаимодействия автомата (рис. 4), предложенной Новиковым Ф. А. [22].

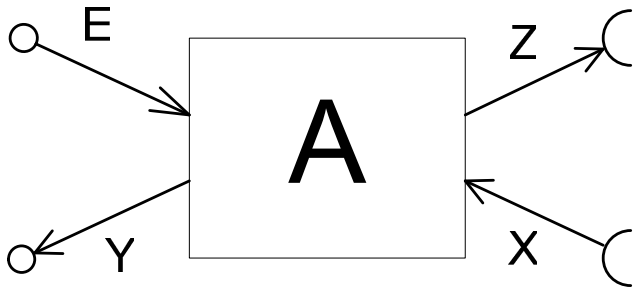

Рис. 4. Интерфейс взаимодействия автоматов

На рис. 4 множество `E` задает входные события автомата `A`, множество `Y` описывает переменные автомата, доступные другим объектам системы, множество `Z` описывает исходящие события и воздействия автомата, и множество `X` задает множество используемых переменных внешних объектов. Данная схема примечательна тем, что позволяет естественным образом «соединены» автоматы, обеспечивая их взаимодействие (рис. 5).

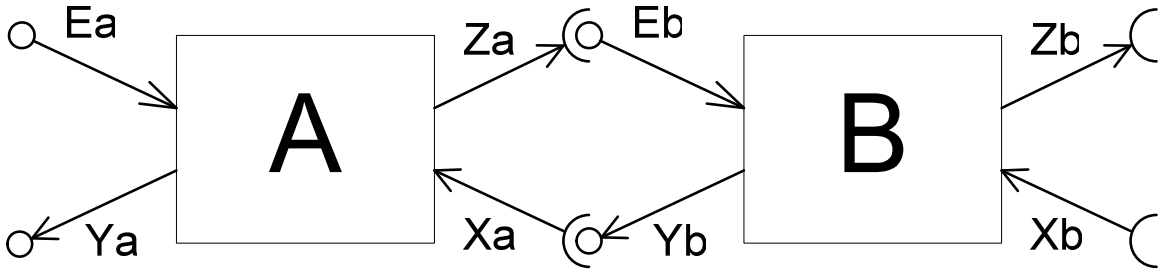

Рис. 5. «Соединение» двух автоматов

Для поддержки описанной схемы взаимодействия в языке *TABP* введено понятие события (ключевое слово **`event`**), подписка на события (ключевое слово **`out`**), а так же автоматы могут иметь переменные и методы. В соответствии с принципами объектно-

ориентированного программирования переменные и методы объекта могут быть открытые (**`public`**), закрытые (**`private`**) и защищенные (**`protected`**).

Стоит отметить, что программист может не использовать автоматных возможностей языка *TABP* и, пользуясь лишь переменными и методами объектов, программировать в «классическом» объектно-ориентированном стиле. Данная возможность необходима, так как позволяет проще решить те задачи, для которых неавтоматный подход лучше подходит. Возможность программировать без использования автоматных возможностей языка не противоречит концепции и основной парадигме языка. Более того, это усиливает гибкость языка – по аналогии можно привести пример с языком *C++*, который позволяет программировать в процедурно-ориентированном стиле вместо объектно-ориентированного, либо языки *Nemerle* [23] и *Scala* [24] – функциональные языки программирования, объединяющие две парадигмы: декларативное (функциональное) программирование и императивное. Многие задачи удобнее решать в императивном стиле, однако, основная парадигма этих языков – функциональное программирование.

2.8. **События и обработчики событий**

Одним из важнейших способов взаимодействия автоматов является передача событий. В реактивных системах получение события приводит к передаче управления автомату. Есть несколько подходов при организации обработки событий, а также в понимании, что есть событие, и как автомат должен вести себя при их обработке. Например, в языке *State Machine* используются «внутренние» события, используемые для связки состояний автомата и осуществления переходов. В общем случае событие – это некоторое «сообщение», передающее информацию о природе события (само имя говорит об этом), а так же через атрибуты события – дополнительные данные, несущие полезную информацию.

При обработке событий могут быть введены строгие требования: автомат должен обрабатывать все входящие события и либо производить какое-то действия, либо перейти в исключительное состояние. На практике такие требования слишком строги и неудобны, поэтому для языка *TABP* выбраны более «мягкие» правила обработки событий, схожие с сигналами в языке *UML* [25]. Все события возникают и обрабатываются последовательно, то есть два события не могут произойти одновременно (вернее, не могут быть обработаны одним объектом). Если произошло несколько событий, то они помещаются в специальную очередь и обрабатываются в порядке возникновения. Если автомат не ожидает события (в

контексте языка *TABP* – если автомат переходит в пассивное состояние), то необработанное событие игнорируется.

Для удобства в языке *TABP* введено несколько способов описания событий. Первый способ – это описание простого события без атрибутов:

```
event EventName;
```

Второй способ – это описание события вместе с атрибутами. В языке *TABP* событие – это объект, соответственно, события могут иметь помимо атрибутов еще и методы:

```
event EventName {
      float eventAttrib;
      void eventMethod() {}
}
```

Для обработки событий используется оператор **on**. Оператор обработки событий абстрагирует программиста от работы с очередью сообщений. При обработке событие автоматически извлекается из очереди сообщений:

```
on EventName {
      // …
}
```

При описании автоматов в виде диаграммы переходов принято использовать сторожевые условия [25] – логические условия, использующие атрибуты полученного события. На рис. 6 приведен пример автомата, использующего сторожевые условия на числовой атрибут `value` события `E1`.

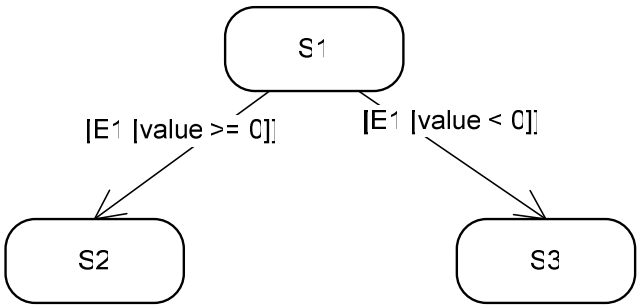

Рис. 6. Пример автомата со сторожевым условием

Язык *TABP* предоставляет синтаксический сахар, для описания сторожевых условий – можно задать логическое условие в скобках после имени события:

```
fixed S1 {
      on E1(value >= 0) {
            relax S2;
      }
      on E1(value < 0) {
            relax S3;
```

```
        }
}
```

Событие – это объект, имеющий свои атрибуты и методы, для работы с этим объектом можно использовать ключевое слово **event**, имеющее в рамках блока **on** тип обрабатываемого события.

Помимо оператора **on** в язык *TABP* введен оператор **otherwise**, предназначенный для обработки произвольного события, тип которого неизвестен. Данный оператор может быть использован, например, для перехода в исключительное состояние при реализации строгой схемы обработки событий, а также для действий «по умолчанию» :

```
otherwise {
      // …
}
```

2.9. **Посылка событий**

Существует два принципиально разных способа передачи событий – синхронный и асинхронный. Передача синхронного события аналогична вызову функции или метода объекта: программа останавливается, передает управление автомату, которому адресовано событие, и дожидается окончания обработки события и возвращения управления обратно. Если в очереди автомата уже есть какие-то события, то сначала обрабатываются более ранние события. При асинхронной посылке события адресату не передается управление, а само событие добавляется в очередь автомата.

Одним из примеров подобной схемы передачи событий является механизм посылки сообщений в операционной системе *Windows*. Посылка сообщений осуществляется посредством вызова функций `SendMessage` и `PostMessage` – для синхронных и асинхронных сообщений соответственно.

В языке *TABP* реализована вышеописанная схема. Она наиболее естественно подходит к выбранному механизму обработки событий, описанному в предыдущем разделе, а также данная схема включает в себя обе стратегии передачи управления.

Для посылки событий используются два оператора **send to** и **post to**:

```
send EventName to anAutomaton;
post EventName to anAutomaton;
```

В случае асинхронного события вторая часть **to** может быть опущена. В таком случае автомат пошлет событие самому себе. Если вызов происходит в контексте оператора **on**, то имя события может быть опущено, в этом случае обрабатываемое событие пошлется выбранному адресату:

```
send to anAutomaton;
post to anAutomaton;
post ; // to this;
```

2.10. **Подписка на события**

Язык *TABP* предоставляет средства для подписки автоматов на события. Механизм подписки на сообщения похож на аналогичный механизм в языке *C#*. Однако в языке *C#* на события подписываются делегаты [26], и сами события – это свойства объектов, а не самостоятельные объекты. В языке *TABP* на события подписываются автоматы, и при возникновении события происходит его посылка аналогично выполнению операторов **send to** и **post to**:

```
auto A1 {
     out EventName z1;
}
```

Исходящее событие описывается с помощью ключевого слова out и активируется аналогично вызову функции, при этом аргументами функции активации являются аргументы конструктора соответствующего события:

```
z1(); // Активация исходящего события
```

Для подписки на событие используется имя исходящего события и оператор **+=**:

```
a.z1 += myAuto;
```

При подписке посредством оператора **+=** посылка события происходит синхронно. Однако это лишь один способ передачи сообщения. Для поддержки асинхронных событий в языке *TABP* введена расширенная форма подписки **+= post** и **+= send**, с помощью которой программист явно задает, какой тип события должен произойти:

```
a.z1 += post myAutomaton;
a.z1 += send myAutomaton;
```

Отметим, что подписка на события соответствует множеству Z исходящих воздействий, предназначенных для связки автоматов. Достоинством и в тоже время недостатком :-) является косвенное связывание автоматов. Поэтому автомату не известно, на события какого автомата он подписан. Это подходит для разработки «библиотечных» автоматов общего назначения и в целом для повторного использования кода. Недостатки подписки на события аналогичны недостаткам событий в языке *C#*: в больших системах сложно отследить связи между автоматами, посылающими друг другу события.

2.11. **Сохранение контекста выполнения**

В данном разделе хотелось бы остановиться отдельно на важной особенности языка *TABP*, а именно на механизме сохранения контекста выполнения программы при вызове оператора **relax** и переходе автомата в пассивное состояние. Дело в том, что оператор **relax** может быть вызван в любом месте внутри блока, описывающего состояние автомата. В области видимости оператора **relax** могут находиться переменные, значения которых должны быть восстановлены при возвращении управления автомату:

```
step {
     int a = 12345;
     relax PassiveState;
     // …
     fixed PassiveState {
          // После возвращения управления автомату
          // значение переменной a останется корректным
     }
}
```

Данный механизм похож на принцип работы оператора **yield** в языке *C#* [26], предназначенного для более удобного описания итераторов. Однако в отличие от оператора **yield** механизм, предлагаемый языком *TABP*, более общий и предоставляет больше возможностей (оператор **yield** предназначен только для описания итераторов и не предназначен для другого применения).

Для поддержки сохранения контекста автомата при переходе в пассивное состояние автомат хранит стек для хранения значений переменных. Программист описывает переменные так же, как обычные локальные переменные на стеке, и компилятор самостоятельно определяет, где именно должна располагаться переменная: во временном стеке процедуры автомата, либо в постоянном стеке объекта автомата.

2.12. **Обратный вызов**

В языке *TABP* при синхронной посылке события автомат находится в виртуальном состоянии ожидания завершения обработки события. Находясь в состоянии ожидания, автомат не может обрабатывать входные события. Если в момент ожидания автомату будет послано синхронное событие, то произойдет ошибка времени выполнения. В частности, синхронное событие может послать тот автомат, от которого мы ожидаем завершения обработки. Для разрешения данной ситуации язык *TABP* предоставляет возможность «расширить» виртуальное состояние ожидания, позволив автомату обрабатывать события. В качестве примера можно привести алгоритм сортировки, в

котором автомат A посылает автомату B событие начала сортировки, а автомат B посылает автомату A запросы для вычисления результатов сравнения элементов:

```
auto A {
      B b = new B();
      send Sort to b {
            on Compare {
                  send CompareResult(event.a < event.b) to b;
            }
      };
}
auto B {
      // …
      send Compare(a, b) to a {
            on CompareResult {
                  // …
            }
      }
}
```

Для поддержки возможности описания обработчиков для виртуальных состояний ожидания в язык *TABP* введена расширенная запись посылки событий:

```
send EventName to a {
      // …
}
```

Описываемое виртуальное состояние предназначено только для обработки событий. Поэтому на тело обработчика, описанного в блоке **send to**, наложены ограничения: из описываемого блока невозможно выйти наружу (например, посредством оператора **goto** или **relax**), а также данное состояние не может иметь вложенных пассивных подсостояний. Стоит отметить, что внутри обработчика виртуального состояния видны все переменные автомата, которые видны в соответствующем месте описания оператора **send to**.

Посредством данной механики можно описывать протоколы взаимодействия на основе синхронных событий.

2.13. **Суперсостояния**

При описании диаграммы переходов конечного автомата часто используется специальное обозначение, задающее общий переход для множества состояний (рис. 7).

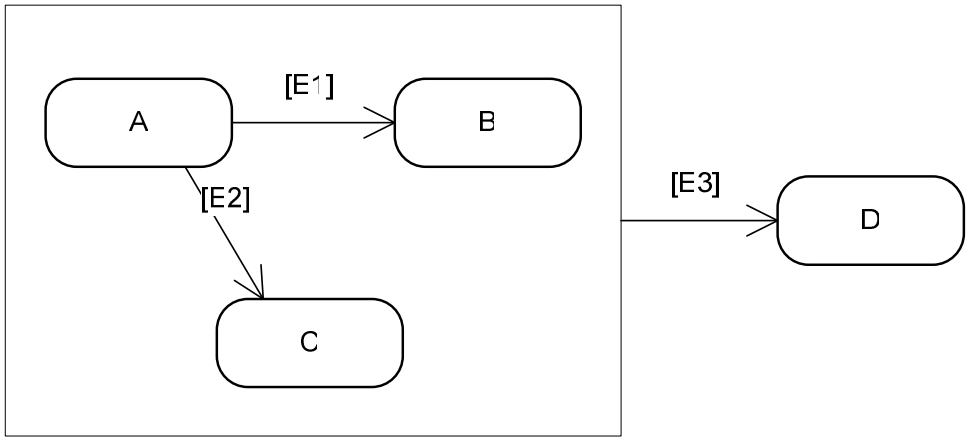

Рис. 7. Пример использования перехода, общего для группы состояний

Данная нотация аналогична записи автомата, приведенной на рис. 8.

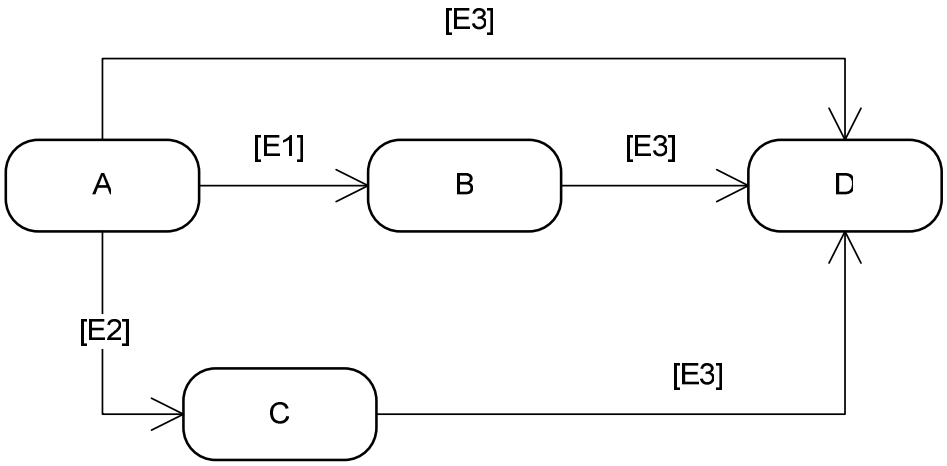

Рис. 8. Автомат соответствующий общему переходу

Из приведенных примеров видно, что нотация общего перехода гораздо удобнее стандартной записи. При описании автомата в текстовом виде также удобно описывать часто повторяемые переходы в виде аналогичной нотации общего перехода.

Для поддержки обобщенного перехода в язык *TABP* введено понятие «суперсостояния», описываемое с помощью блока **`super`**:

```
super {
      // …
}
```

Код, описанный внутри блока, автоматически добавляется в конец последующих шагов, в области которых видно описанное состояние. Идея заключается в том, что обработчики внутри блока **`super`** имеют более низкий приоритет. Например, если в обработчике `super` есть блок, обрабатывающий некоторое событие `E`, и внутри

некоторого шага в области действия суперсостояния также находится обработчик события E, то сработает последний обработчик события:

```
super {
     on E {
          // …
     }
}
step {
     on E {
          // Более «низкоуровневое» состояние имеет
          // более высокий приоритет
     }
}
```

Суперсостояния могут использоваться иерархически – на состояние могут действовать несколько вложенных суперсостояний. Действия выполняются в порядке возрастания уровня абстракции – от низкоуровнего к более высокоуровневому суперсостоянию:

```
super {
     // A
}
super {
     // B
}
step {
     // C
}
// Последовательность выполнения: C, B, A
```

Поскольку суперсостояния предназначены для описания общих обработчиков событий для пассивных состояний, код суперсостояний выполняется в момент вызова оператора **relax**. Таким образом, пока автомат находится в активном состоянии, обработчики суперсостояний неактивны. Также, суперсостояние не может содержать пассивных подсостояний, однако могут осуществляться выходы за пределы блока посредством операторов **goto** и **relax**.

Концепция суперсостояний в языке *TABP* позволяет описывать обобщенные переходы автомата и в частности позволяет реализовать пример, приведенный на рис. 7:

```
step {
     super {
          on E3 {
               relax D;
          }
     }
     fixed A {
          on E1 goto B;
```

```
            on E2 goto C;
        }
        fixed B;
        fixed C;
}
fixed D;
```

2.14. **Подавтоматы**

В работе [27] предложен паттерн *O-State*, в частности, рассматриваются *AND*-состояния [28] для слабодетерминированных систем. На рис. 9 приведен пример *AND*-состояния: если *AND*-состояние активно, то активны оба состояния `A` и `B`, если *AND*-состояние неактивно, то также неактивны состояния `A` и `B` (вместо состояний `A` и `B` могут быть подавтоматы).

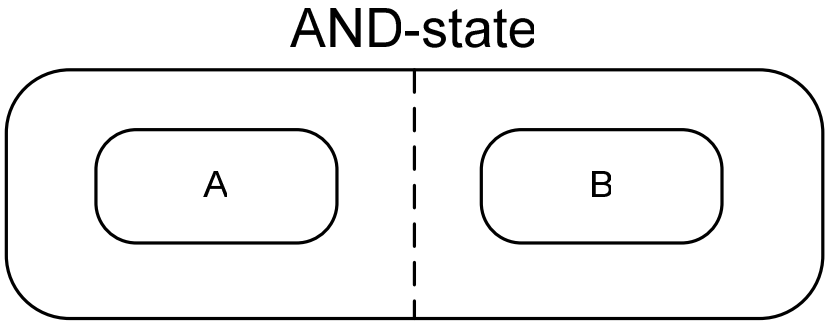

Рис. 9. Пример *AND*-состояния

В работе [27] показывается, что автомат с *AND*-состояниями является слабодетерминированным – при возникновении некоторого события (вне зависимости от последовательности, в которой возникшее событие обрабатывается подсостояниями) автомат осуществляет одинаковые переходы.

По аналогии с данной схемой в язык *TABP* введены «подавтоматы». Для некоторого автомата может быть задан набор подавтоматов, поведение которых аналогично поведению *AND*-состояния. Каждый подавтомат имеет собственное независимое состояние. При получении события главный автомат транслирует его своим подавтоматам.

Для описания подавтомата используется блок **`subauto`**, внутри которого описываются состояния подавтомата. Подавтомат может иметь имя либо быть анонимным. Имя автомата используется для подписки конкретного подавтомата на события, а также для посылки сообщений (в том числе для пересылки сообщений между подавтоматами) :

```
auto Automaton {
    subauto Subautomaton {
```

```
            // …
        }
        subauto /* Anonymous subautomaton */ {
            // …
        }
}
```

Подавтоматы могут иметь собственный набор событий, на которые он реагирует. Например, подавтомат, описывающий поведение включения-выключения объекта, может обрабатывать событие «включить». Для удобства подавтомат может быть подписан независимо от других подавтоматов – в этом случае событие будет получать только подписанный подавтомат:

```
a.outEvent += b.SubAuto;
```

Также, при посылке событий можно явно задать конкретный подавтомат (аналогично подавтомат может послать событие другому подавтомату собственного объекта) :

```
send EventName to a.SubAuto;
```

2.15. **Мультисостояния**

Язык *TABP* предоставляет возможность описывать *OR*-состояния языка *UML*. Данные состояния в один момент времени имеют от одного и более активных подсостояний. Для описания *OR*-состояний используется блок **multi**:

```
multi {
        step A {
        }
        step B {
        }
        // …
}
```

Состояния, описанные внутри блока **multi**, выполняются параллельно. Соответственно, выход из блока **multi** происходит, когда все подсостояния возвращают управление.

В качестве примера использования *OR*-состояний можно привести задачу *ABRO* [32]. Суть задачи заключается в том, что автомат должен дождаться события A и B, после чего сгенерировать событие O. Если в любой момент времени пришло событие R, то автомат должен прекратить ожидание событий A и B (листинг 6).

Листинг 6. Решение задачи *ABRO*

```
auto ABRO {
  out O onO;
  step { // Состояния обработки событий A и B
    super {
      // При возникновении события R завершить ожидание
      on R goto Reset;
    }
    multi {          // OR-состояние
      fixed {        // Ожидание события A
        on A break;
      }
      fixed {        // Ожидание события B
        on B break;
      }
    }
    onO();           // Генерация собятия O
  }
  step Reset;
}
```

2.16. **Наследование автоматов**

В языке *TABP* один автомат может быть унаследован от другого автомата в объектно-ориентированном смысле. Автомат-наследник может замещать виртуальные методы автомата, добавлять новые методы и переменные и т. д. Помимо стандартных возможностей, доступных в объектно-ориентированных языках программирования, язык *TABP* предоставляет инструменты для расширения наследуемых автоматов.

В ряде случаев деревья состояний автоматов (все состояния автомата с учетом их вложенности) могут быть одинаковыми и различаются лишь на некотором уровне вложенности (рис. 10).

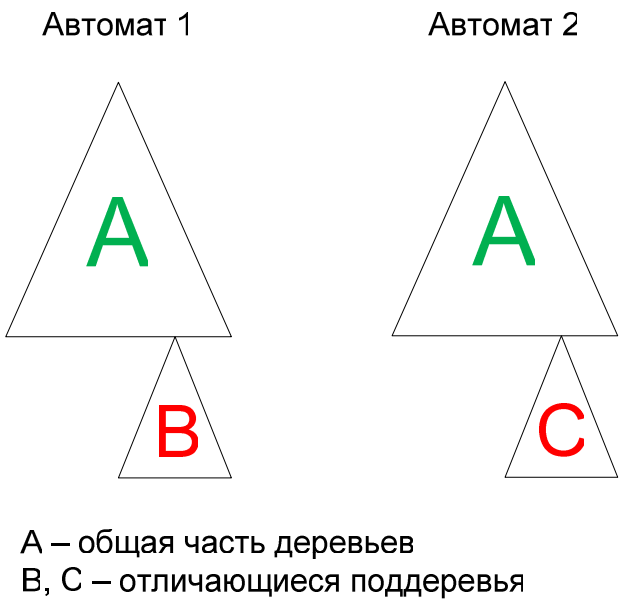

Рис. 10. Абстрактный пример автоматов с отличающимися поддеревьями состояний

Язык *TABP* предоставляет возможность вынести в отдельный автомат общую часть, а в автоматах-наследниках реализовать отличающиеся поддеревья (рис. 11).

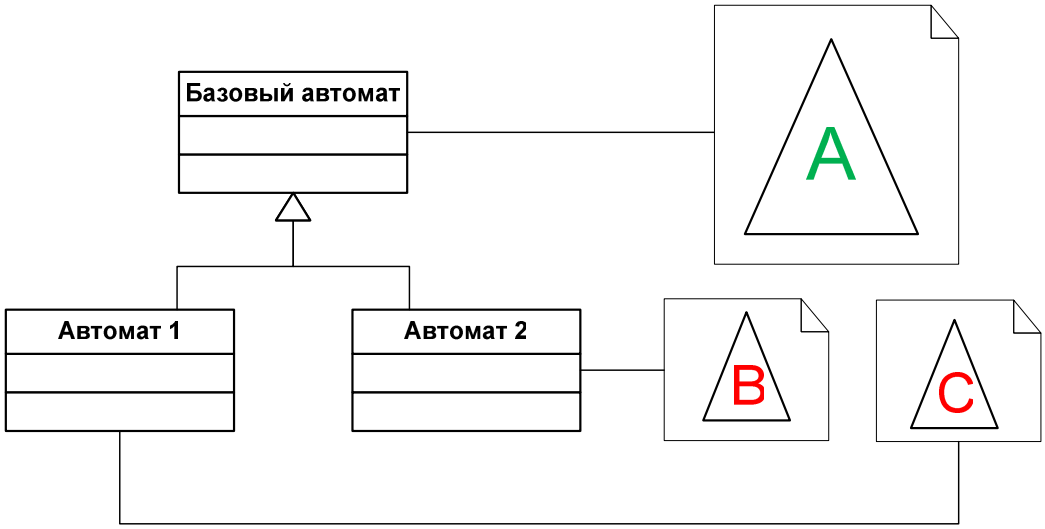

Рис. 11. Вынесение общей части в базовый автомат

Для поддержки расширения базового автомата в языке *TABP* используется специальный блок **`virtual`**, с помощью которого описывается виртуальное состояние. В базовом автомате объявляется виртуальное состояние с уникальным именем, тем самым помечая место, в котором должно быть расширение:

```
auto BaseAutomaton {
     step A {
          // …
          virtual ExtensionPlace; // Место расширения
     }
```

```
}
```

Автоматы-наследники реализуют виртуальное состояние, расширяя базовый автомат:

```
auto Automaton1 : BaseAutomaton {
     virtual ExtensionPlace {
          // Расширение базового автомата
          step B {
               // …
          }
     }
}
```

И базовый автомат, и автоматы-наследники могут иметь произвольное число виртуальных состояний.

Отметим, что механизм наследования, предложенный в языке *TABP*, не является единственно возможным. В частности, в работе [29] предложена нотация наследования автоматных классов. Однако, применение данной нотации в языке *TABP* затруднительно, так как нотация требует декомпозиции описываемых автоматов, что требует существенных изменений в синтаксисе языка.

3. Примеры использования языка *TABP*

В этом разделе приведены примеры использование языка *TABP* на основе автоматов, рассмотренных в работе [16]. В примерах рассматривается система объектов, состоящая из двери и лампы. В начальном состоянии дверь закрыта, лампа выключена. При открытии двери снаружи лампа включается. При повторном открытии двери изнутри лампа выключается.

3.1. Активный автомат Мура

На рис. 12 приведена схема связей, а на рис. 13 – граф переходов данного примера.

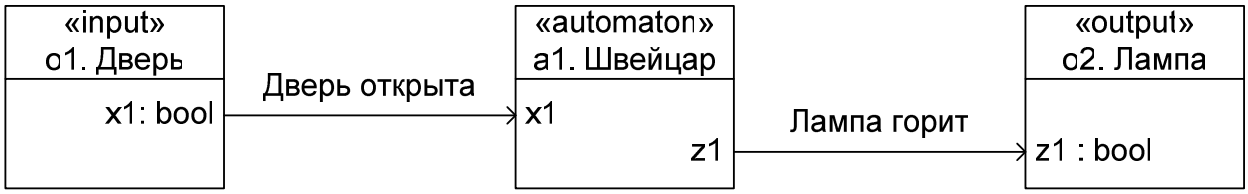

Рис. 12. Схема связей активного автомата Мура для примера с дверью и лампочкой

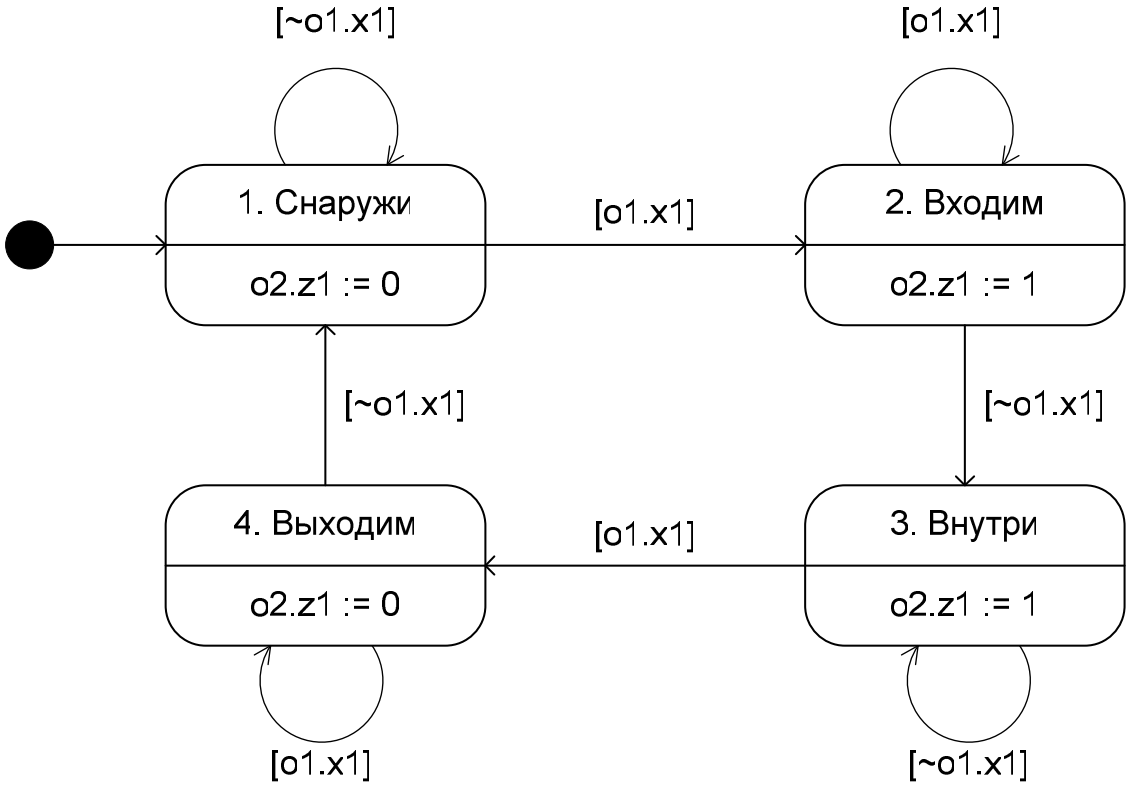

Рис. 13. Граф переходов автомата Мура для примера двери с лампочкой

Далее приведена реализация данного автомата на языке *TABP* (листинг 7).

Листинг 7. Реализация автомата Мура для системы из двери и лампочки

```
auto A1 {
  bool x1(void) { /*...*/ } // Дверь открыта
  void z1(bool) { /*...*/ } // Лампа горит

  loop Outside { // Снаружи
    z1(false);
    if (x1()) {
      z1(true);
      step { // Enter, Входим
        if (x1()) repeat;
      }
      break; // goto Inside;
    }
  }

  loop Inside {
    z1(true);
    if (x1()) {
      z1(false);
      step { // Exit, Выходим
        if (x1()) repeat;
      }
      goto Outside;
    }
  }
}
```

Автомат состоит из двух состояний `Outside` (снаружи) и `Inside` (внутри), анонимные состояния, помеченные соответственно `Enter` и `Exit`, являются вспомогательными состояниями, предназначенными для ожидания закрытия двери.

3.2. **Пассивный автомат Мура**

В данном примере имеется один объект, являющийся одновременно и объектом управления и источником событий. На рис. 14 приведена схема связей, а на рис. 15 – граф переходов. После них приведена реализация автомата швейцара на языке *TABP* (листинг 8).

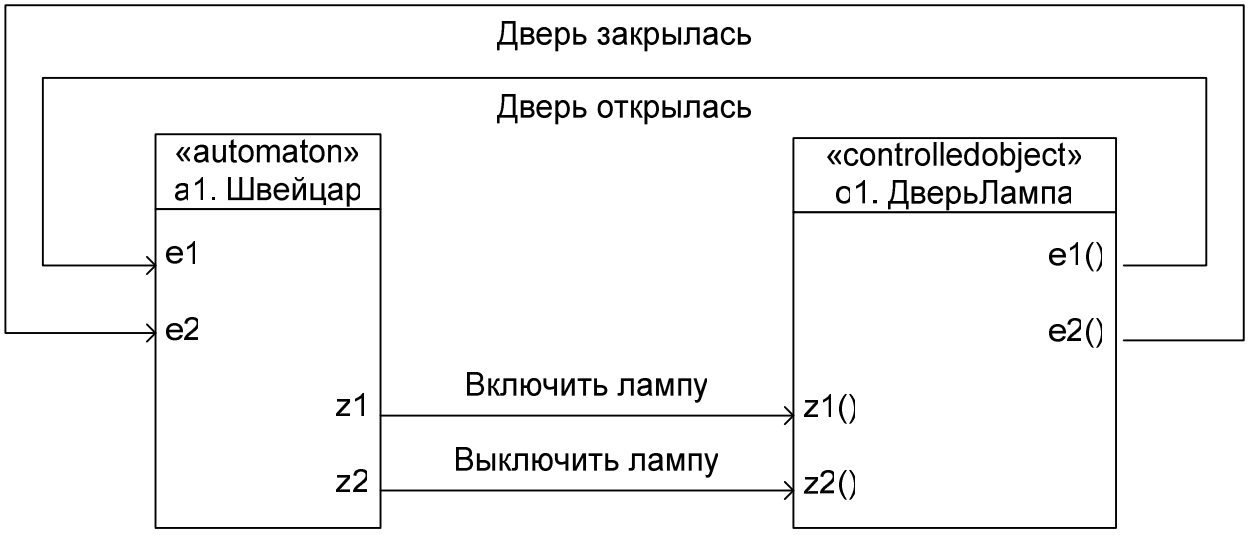

Рис. 14. Схема связей пассивного автомата Мура для примера с дверью и лампой

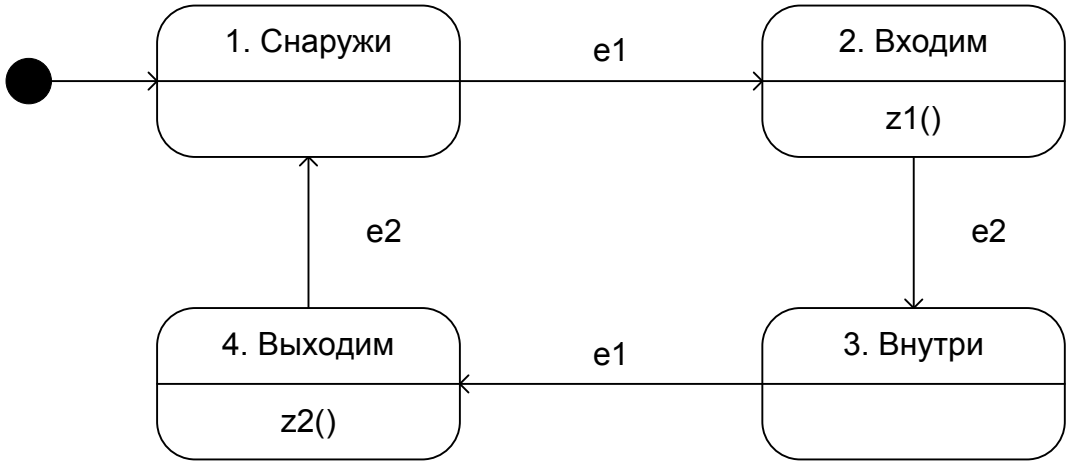

Рис. 15. Граф переходов пассивного автомата Мура для примера с дверью и лампой

Листинг 8. Реализация пассивного автомата Мура

```
event e1; // Дверь открылась
event e2; // Дверь закрылась
event z1; // Включить лампу
event z2; // Выключить лампу

auto A2 { // Швейцар
  out z1 z1;
  out z2 z2;

  fixed Outside { // Снаружи
    z2();
    on e1 {
      z1();
      fixed { // Enter, Входим
        on e2 {
          break;
        }
      }
```

```
      relax Inside;
    }
  }

  fixed Inside { // Внутри
    z1 ();
    on e1 {
      z2();
      fixed { // Exit, Выходим
        on e2 {
          break;
        }
      }
      relax Outside;
    }
  }
}
```

3.3. **Связь автоматов по состояниям**

Поведение системы может быть описано не одним автоматов, а несколькими, которые взаимодействуют различным образом, например, через переменную другого автомата для управления своими переходами.

На рис. 16 приведена схема связей автоматов, а на рис. 17 – их графы переходов. После которых приведена реализация автоматов на языке *TABP* (листинг 9).

Рис. 16. Схема связей автоматов, связанных по состояниям

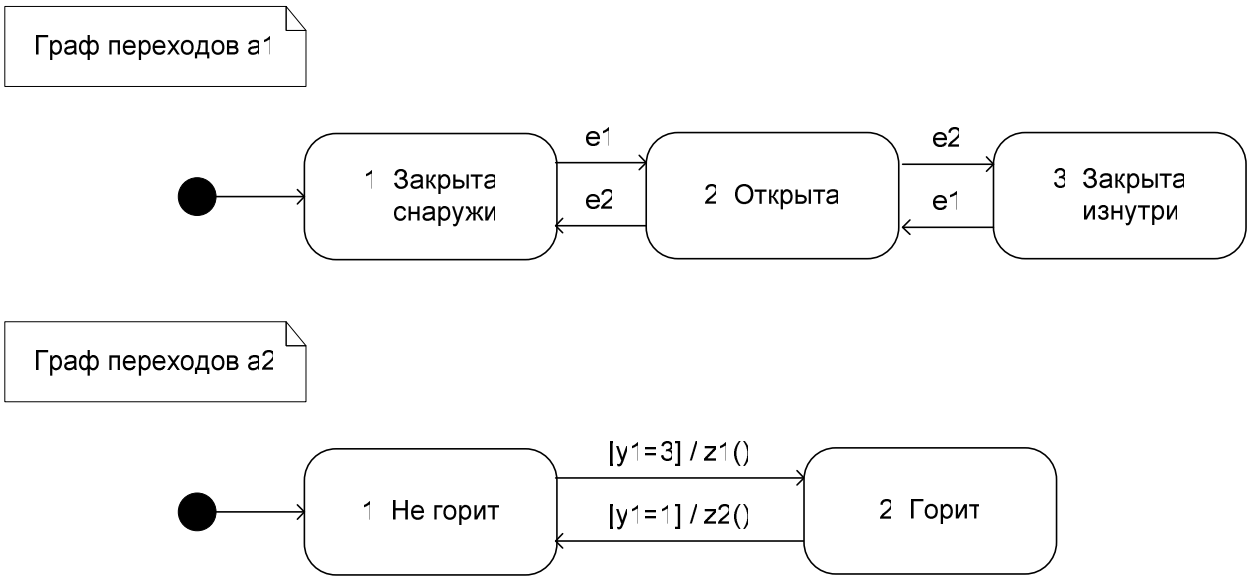

Рис. 17. Граф переходов автоматов, связанных по состояниям

Листинг 9. Реализация автоматов на языке *TABP* автоматов, связанных по состояниям

```
event e1; // Дверь открылась
event e2; // Дверь закрылась

auto A1 { // Швейцар
  int y1; // Состояние автомата

  fixed ClosedOutside { // Закрыта снаружи
    y1 = 1;
    on e1 {
      fixed { // OpennedOutside, Открыта
        y1 = 2;
        on e2 {
          break;
        }
      }
      fixed { // ClosedInside, Закрыта изнутри
        y1 = 3;
        on e1 {
          break;
        }
      }
      fixed { // OpennedInside, Открыта
        y1 = 2;
        on e2 {
          goto ClosedOutside;
        }
      }
    }
  }
}
```

```
event z1; // Включить лампу
event z2; // Выключить лампу

auto A2 { // Электрик
  out z1 z1;
  out z2 z2;

  A1 a1; // Ссылка на автомат A1 для доступа к переменной y1

  A2(A1 a1) {
     this.a1 = a1;
  }

  loop { // Off, Не горит
    if (a1.y1 == 3) {
      z1();
      loop { // On, горит
        if (a1.y1 == 1) {
          z2();
          break;
        }
      }
    }
  }
}
```

Автомат A1 является пассивным, в то время как автомат A2 – активным. Чтобы он тоже был пассивным, требуется дополнительный источник событий, так как автомат не может подписаться на событие «изменение переменной y1». Это может быть осуществлено, если, например, автомат A1 при изменении переменной y1 будет генерировать событие StateChanged, а автомат A2, соответственно, подпишется на него (листинг 10).

Листинг 10. Реализация автоматов, связанных по состояниям. Вариант с пассивным автоматом электрика

```
event e1; // Дверь открылась
event e2; // Дверь закрылась
event StateChanged; // Состояние переменной y1 изменилось

auto A1 { // Швейцар
  out StateChanged onStateChanged;
  int y1; // Состояние автомата

  void setY1(int value) {
     y1 = value;
     onStateChanged();
  }

  fixed ClosedOutside { // Закрыта снаружи
```

```
    setY1(1);
    on e1 {
      fixed { // OpennedOutside, Открыта
        setY1(2);
        on e2 {
          break;
        }
      }
      fixed { // ClosedInside, Закрыта изнутри
        setY1(3);
        on e1 {
          break;
        }
      }
      fixed { // OpennedInside, Открыта
        setY1(2);
        on e2 {
          goto ClosedOutside;
        }
      }
    }
  }
}

event z1; // Включить лампу
event z2; // Выключить лампу

auto A2 { // Электрик
  out z1 z1;
  out z2 z2;

  A1 a1;  // Ссылка на автомат A1

  A2(A1 a1) {
     this.a1 = a1;                // Сохраняем ссылку на A1
     a1.onStateChanged += this; // Подписка на событие
  }

  fixed { // Off, Не горит
    on StateChanged(a1.y1 == 3) {
      z1();
      fixed { // On, горит
        on StateChanged(a1.y1 == 1) {
          z2();
          break;
        }
      }
    }
  }
}
```

Во втором варианте при каждом изменении переменной `y1` автомат `A1` генерирует событие `StateChanged`. Автомат `A2` при инициализации подписывается на соответствующее событие в конструкторе автомата. Ко всем блокам **`if`** в новом примере добавляется обработчик сообщения `StateChanged`, и вместо операторов **`repeat`** используется оператор **`relax`** (блок **`fixed`** вместо **`step`**).

3.4. **Использование событий с параметрами**

До сих пор использовались события без параметров. В этом заключается некоторая избыточность, так как достаточно использовать единственное событие `e1` – «дверь сдвинулась», а в качестве параметра передавать булеву переменную, принимающую значение «истина», если дверь открылась, и «ложь» в обратном случае.

На рис. 18 представлена схема связей автомата при использовании параметризованных входных и выходных воздействий; соответственно, на рис. 19 представлен граф переходов автомата. Реализация автомата на языке TABP приведена в листинг 11.

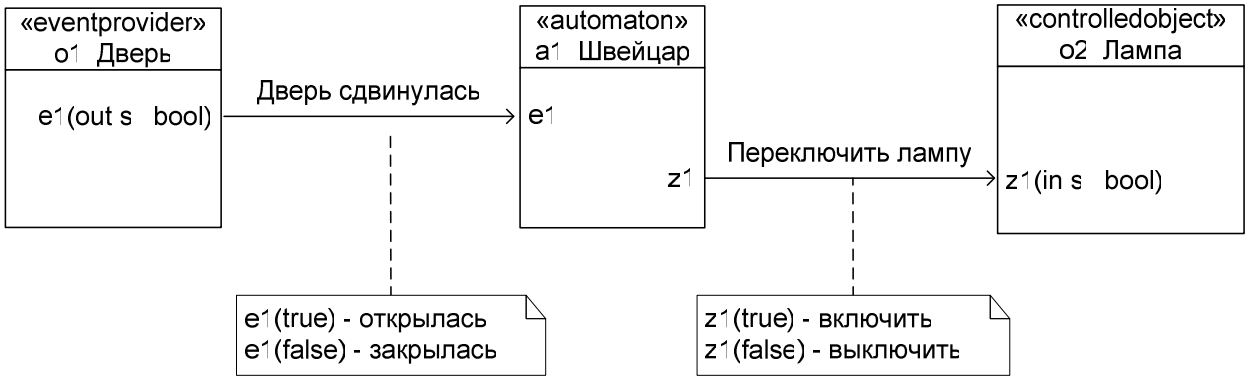

Рис. 18. Схема связей автомата при использовании параметризованных входных и выходных воздействий

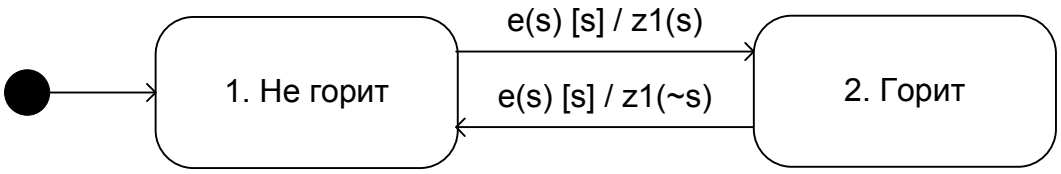

Рис. 19. Граф переходов автомата при использовании параметризованных входных и выходных воздействий

Листинг 11. Реализация автомата на языке TABP при использовании параметризованных входных и выходных воздействий

```
event e1 {
  bool open; // Дверь сдвинулась
             // (true - открылась, false - закрылась)
}

event z1 {
  bool turnOn; // Переключить лампу
               // (true - включить, false - выключить)
}

auto A1 { // Швейцар
  out z1 z1;

  fixed { // Off, Не горит
    on e1(event.open) {
      z1(true);
      fixed { // On, Горит
        on e1(event.open) {
          z1(false);
          break;
        }
      }
    }
  }
}
```

В автомате используются события со сторожевым условием.

3.5. **Вызов автомата возбуждением события**

В разд. 3.3 был рассмотрен вариант реализации примера с пассивным автоматом электрика. Его можно усовершенствовать, – дело в том, что можно обойтись без лишней связи – переменной, состояния автомата швейцара. Для требуемого поведения лапочки достаточно одного события, извещающего об изменении состояния. Схема связей усовершенствованного автомата приведена на рис. 20.

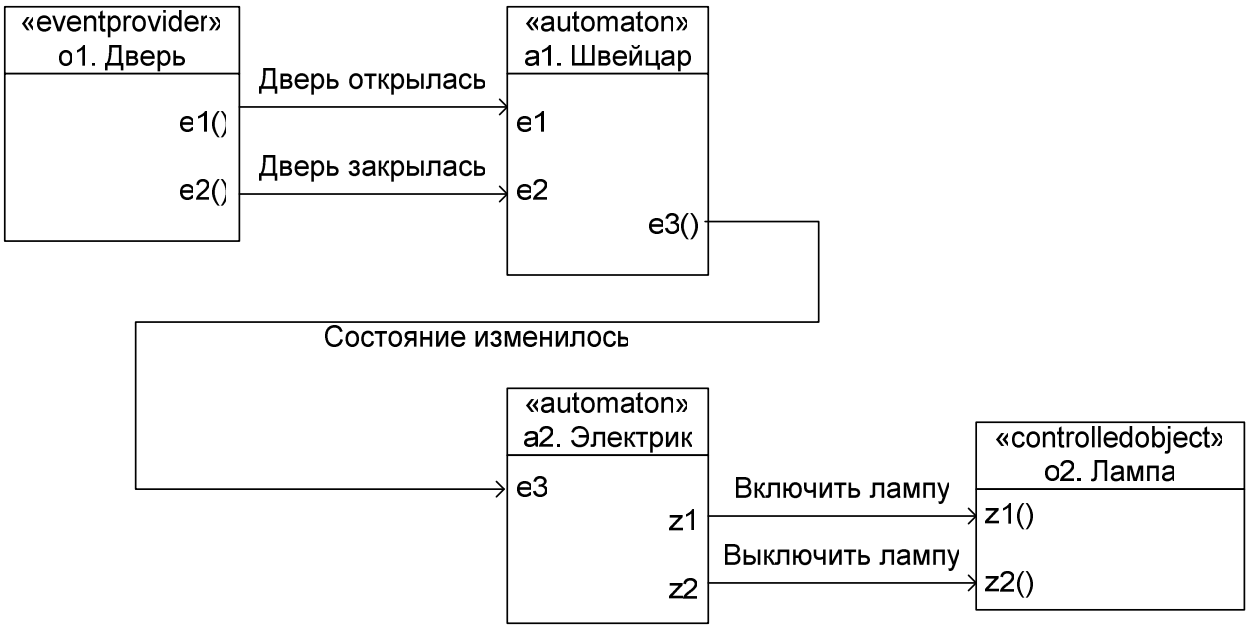

Рис. 20. Схема связей автоматов возбуждением события

Отметим, что вызов автомата возбуждением события может осуществляться несколькими способами. Первый из них был показан в разд. 3.3 – вызов автомата посредством активации выходного события. Такой вариант не всегда подходит, так как на выходное событие могут подписаться и другие автоматы. Другой способ вызова автомата возбуждением события, это непосредственная посылка события конкретному автомату. На рис. 21 представлены графы переходов, за которым следует реализация автоматов швейцара и электрика (листинг 12).

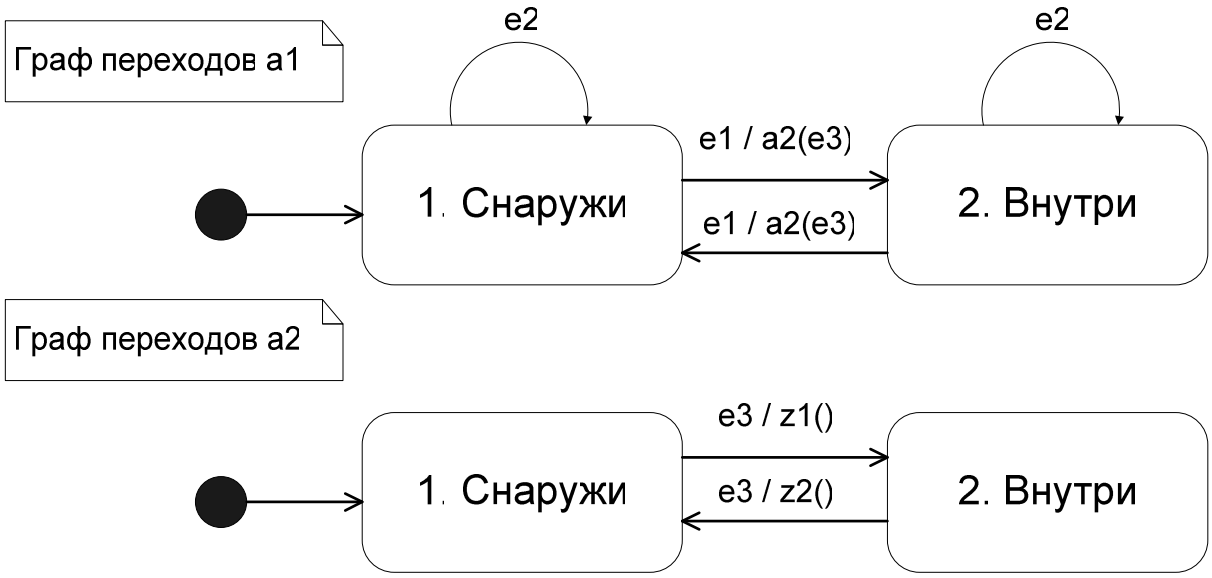

Рис. 21. Графы переходов автоматов, связанных возбуждением события

Листинг 12. Реализация на языке ТАВР автоматов, связанных возбуждением события

```
event e1; // Дверь открылась
event e2; // Дверь закрылась
event e3; // Изменение состояние
event z1; // Включить лампу
event z2; // Выключить лампу

auto A2 { // Электрик
  out z1 z1;
  out z2 z2;

  fixed { // Снаружи
    on e3 {
      z1();
      fixed { // Внутри
        on e3 {
          z2();
          break;
        }
      }
    }
  }
}

auto A1 { // Швейцар
  A2 a2;

  A1() {
    a2 = new A2();
  }

  fixed { // Снаружи
    on e1 {
      send e3 to a2;
      fixed { // Внутри
        on e1 {
          send e3 to a2;
          break;
        }
      }
    }
  }
}
```

Автомат `A1` содержит переменную `a2` типа `A2`, которая инициализируется в конструкторе автомата. Данный механизм весьма полезен для повторного использования кода – можно выделять общую логику в отдельный автомат, причем, в отличие от предыдущего варианта (листинг 11), автомат `A2` ничего не знает об `A1`, то есть его может

использовать любой автомат, в том числе и другая усовершенствованная версия автомата. Швейцара.

3.6. **Демонстрационный проект «Шарики»**

Последним примером данной главы является демонстрационное приложение «Шарики» (рис. 22), реализованное на языке *TABP* с использованием тестовой версии компилятора. Рабочая грамматика реализованного компилятора приведена в приложении 1 (грамматика описана в формате генератора компиляторов *ECG* [30]).

Проект «Шарики» представляет собой «созерцательную» игру, на поле в центре экрана находится центр гравитации, к которому притягиваются шарики. При приближении шарики сталкиваются друг с другом. Каждый шарик имеет собственный радиус и цвет. Если при столкновении шары имеют схожий цвет, то они объединяются в один, образуя новый шар, площадь которого равна сумме площадей объединяемых шаров. Если после объединения размер шарика превышает некоторый барьер, то шар взрывается, разлетаясь на множество разноцветных шаров со случайным цветом и размером.

В игре «Шарики» пользователь может влиять на поведение программы: изменять позицию центра гравитации (правая кнопка мыши), отключать гравитацию (средняя кнопка мыши), а также «перетаскивать» шарики (левая кнопка мыши).

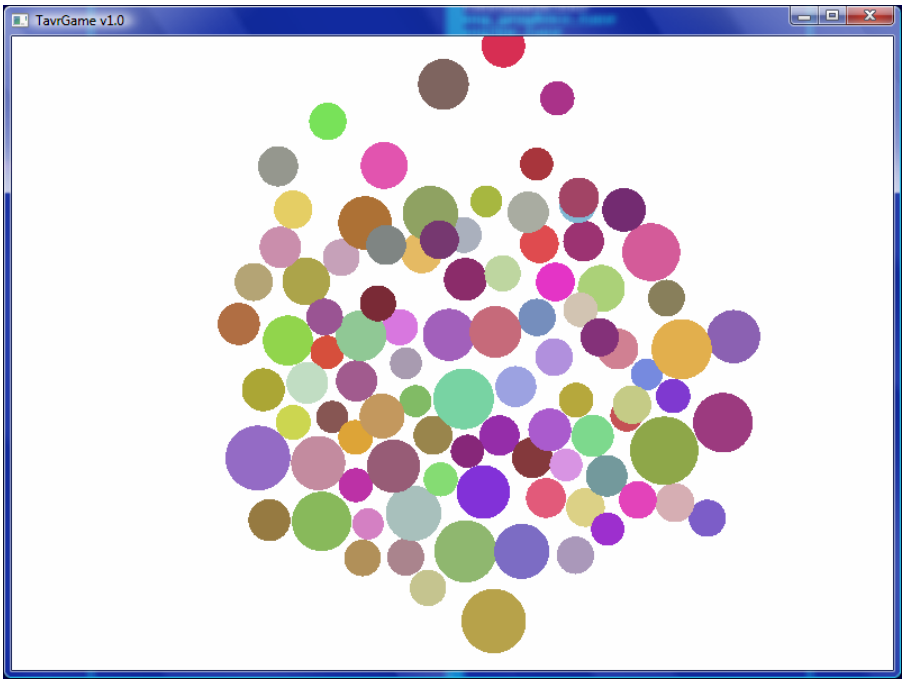

Рис. 22. Окно программы демонстрационного проекта «Шарики»

Игра представляет собой систему взаимодействующих объектов:

- `Engine` – система, абстрагирующая функции операционной системы (создание окна программы, графический вывод, пользовательский ввод). Данная система является поставщиком следующих событий.
  - `EngMouse` – движение мыши и нажатие кнопок.
  - `EngResize` –изменение размера окна.
  - `EngUpdate` – событие, генерируемое окном программы, при отсутствии событий операционной системы.
- `Game` – базовая система игры, хранящая список шариков. Данная система посылает шарикам события `GameUpdate` и `GameRender` для обновления позиции и отрисовки шарика на экран.
- `GameController` – автомат, отвечающий за обработку пользовательского ввода, при перетаскивании шариков посылает событие `GameChangeGravity`.
- `GameBall` – автомат, отвечающий за поведение шарика.

На рис. 23 приведена схема взаимодействия объектов игры.

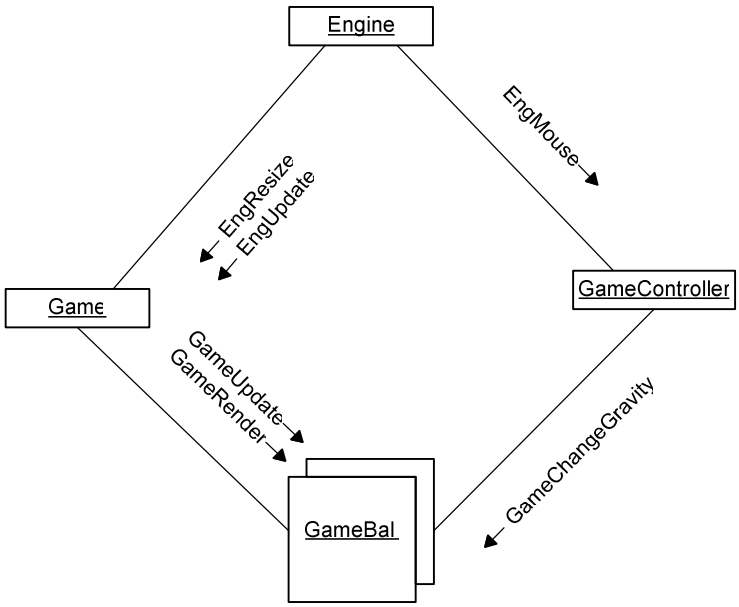

Рис. 23. Схема взаимодействия объектов в проекте «Шарики»

На рис. 24 приведен граф переходов автомата `Game`. На запуске автомат создает шарики и переходит в состояние обработки событий системы.

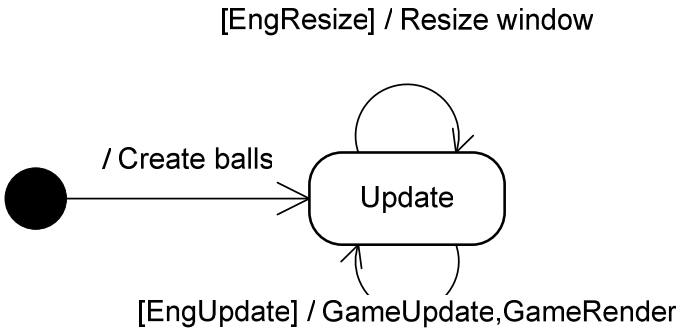

Рис. 24. Граф переходов автомата `Game`

На рис. 25 приведен граф переходов автомата `GameController`, который по событиям мыши изменяет центр гравитации, включает-отключает гравитацию и управляет «перетаскиванием» шариков.

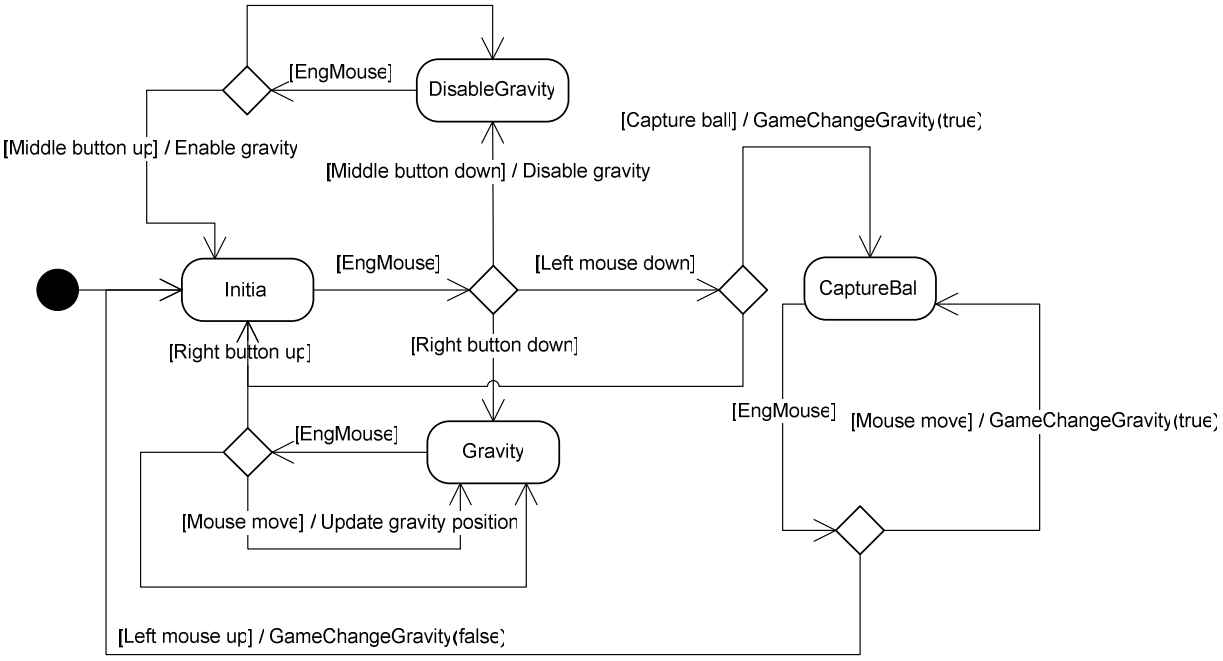

Рис. 25. Граф переходов автомата `GameController`

На рис. 26 приведен автомат `GameBall`, представляющий собой комбинацию двух подавтоматов (в нотации *UML* – два составных *AND*-состояния). Автомат `GameBall` описывает объект игрового шарика. Два подавтомата описывают разные аспекты объекта: отрисовка шарика и обновление его позиции.

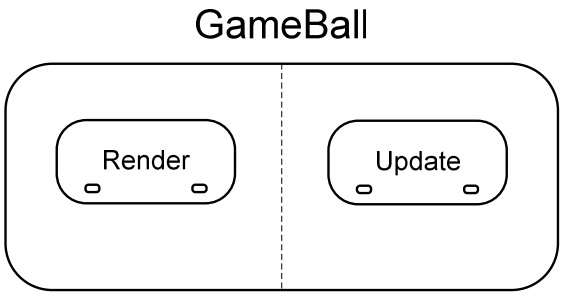

Рис. 26. Автомат `GameBall` с подавтоматами `Render` и `Update`

На рис. 27 приведен граф переходов подавтомата `GameBall.Render`, отвечающий за вывод шарика на экран.

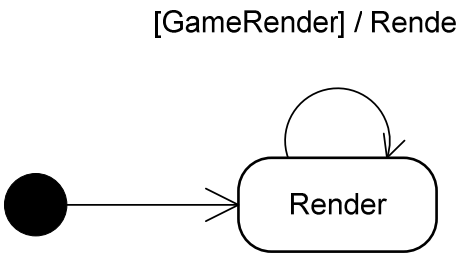

Рис. 27. Граф переходов подавтомата `GameBall.Render`

На рис. 28 приведен граф переходов подавтомата `GameBall.Update`, отвечающий за обновление позиции шарика, также осуществляющий перетаскивание шарика и анимацию размера шара при объединении шариков схожего цвета.

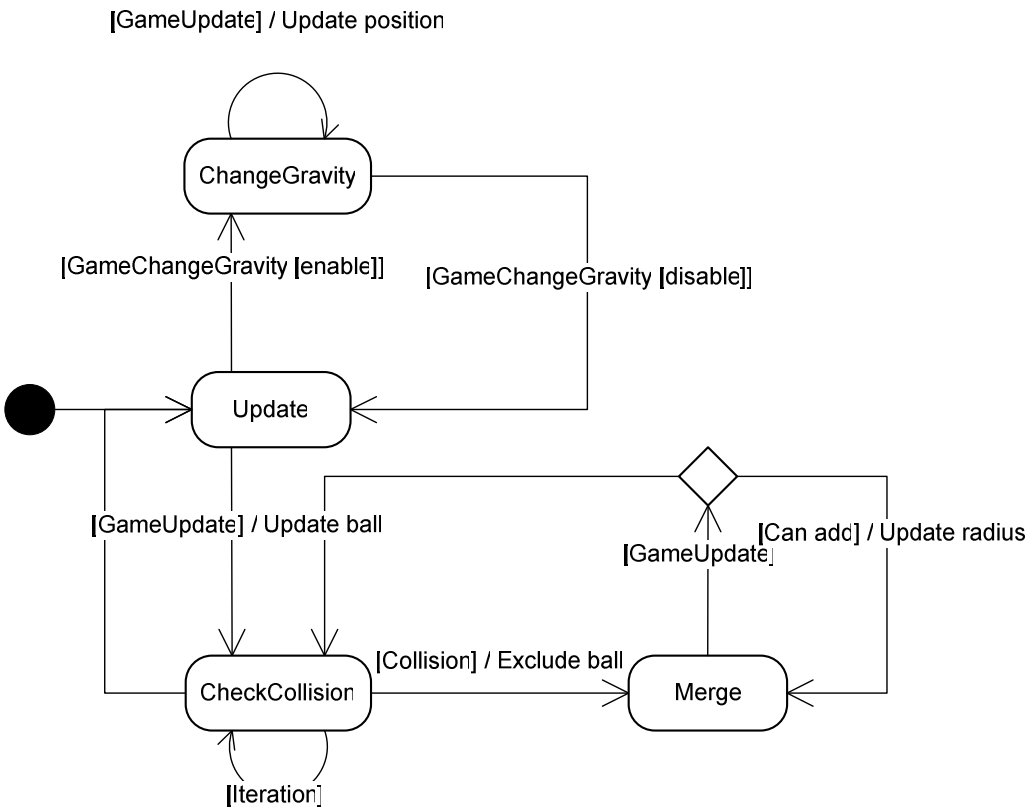

Рис. 28. Граф переходов подавтомата `GameBall.Update`

Описанные автоматы, реализованные на языке *TABP*, приведены в приложениях 2–4.

Заключение

В данной работе выполнен обзор текстовых языков автоматного программирования. В ходе обзора были определены основные задачи текстовых языков автоматного программирования. В ходе работы был предложен текстовый язык автоматного программирования TABP, имеющий следующие особенности:

- кросс-языковость языков, совместимость с семейством языков *.NET*;
- поддержка разных моделей автоматов (от автоматов Мура до более сложных комбинаций с вложенными состояниями и обобщенными обработчиками событий);
- выразительные возможности языка: возможность описания активных и пассивных состояний, вложенных состояний и т. д.;
- возможность задавать несколько автоматов в рамках одной программы и выстраивать цепочки взаимодействия между ними;
- посылка и обработка событий, а так же событий со сторожевыми условиями;
- подписка на синхронные и асинхронные события;
- механизм сохранения контекста выполнения при переходе в пассивное состояние;
- подавтоматы с поведением, аналогичным *AND*-состояниям *UML*;
- наследование автоматов, с возможностью расширения дерева состояний автомата с помощью виртуальных состояний.

Язык TABP позволяет задавать граф переходов автомата в неявном виде, используя блок **`step`** – универсальную конструкцию для описания состояний. Операторы **`goto`** и **`relax`** позволяют осуществлять переходы в другие состояния, а также определяющие, является состояние пассивным или активным.

Язык TABP предоставляет возможность описывать события с параметрами и без, при этом событие является самостоятельным объектом, которое может иметь свои методы и конструкторы. Для посылки событий в разработанном языке используются операторы **`send to`** и **`post to`** для синхронной и асинхронной посылки сообщения соответственно. Для обработки событий используется блок **`on`**, позволяющий помимо обрабатываемого события задавать еще сторожевое условие. Механизмы работы с событиями полностью абстрагируют программиста от работы с очередью сообщений автомата, позволяя использовать события наиболее естественным образом.

Разработанный язык является языком общего назначения, благодаря чему он лишен недостатков специализированных языков-расширений, используемых лишь в отдельных местах программы. Язык *TABP* предназначен для разработки приложения в целом, позволяя описывать объекты системы в виде конечных автоматов.

В качестве развития языка могут быть проведены исследования по следующим вопросам:

- верификация автоматов, описываемых в программах на языке *TABP*;
- в текстовый редактор для программ на языке *TABP* может быть встроен визуализатор описываемых автоматов, строящий граф переходов автомата на лету;
- механизм наследования автоматов может быть расширен, так как текущая версия языка позволяет расширять лишь фиксированные виртуальные состояния, которые предусмотрены на стадии разработки базового автомата. (В будущем хотелось бы иметь возможность расширять граф переходов базового автомата, даже если он не имеет виртуальных состояний.)
- языковые средства для поддержки многопоточности. Может быть введен аналог ключевого слова **`synchronized`**, используемого в языке *Java*, либо могут быть введены языковые расширения, позволяющие распараллеливать алгоритмы, подобно системе *OpenMP* [31].
- автоматные средства для поддержки многопоточности, в частности, поддержка асинхронной посылки событий между разными потоками, выделение отдельного потока конкретному автомату и т. д.

Для языка *TABP* разработан тестовый компилятор, на базе которого создано демонстрационное приложение «Шарики». Разработанный компилятор поддерживает не все возможности языка *TABP*, описанные в данной работе, в частности, этот компилятор транслирует программу на языке *TABP* в программу на языке *C*. После этого выполняется компиляция *C*-кода. В дальнейшем предполагается реализовать язык *TABP* для платформы *.NET* с поддержкой всех языковых возможностей. Язык *TABP* – язык общего назначения и разрабатывался для промышленного использования.

Список литературы

1. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch
2. *UniMod.* http://unimod.sf.net
3. *Eclipse.* http://www.eclipse.org
4. *SMC.* http://smc.sf.net
5. *Шамгунов Н. Н.* Разработка методов проектирования и реализации поведения программных систем на основе автоматного подхода // Диссертация на соискание ученой степени кандидата технических наук. СПбГУ ИТМО, 2004. http://is.ifmo.ru/disser/shamg_disser.pdf
6. *Шалыто А. А.* Алгоритмизация и программирование для систем логического управления и «реактивных» систем // Автоматика и телемеханика. 2000. № 1, с.3–39. http://is.ifmo.ru/works/arew/1/
7. *Цымбалюк Е. А.* Текстовый язык автоматного программирования. // Бакалаврская работа. СПбГУ ИТМО, 2006.
8. *Гуров В. С., Мазин М. А., Шалыто А. А.* Текстовый язык автоматного программирования // Научно-технический вестник. Вып. 42. Фундаментальные и прикладные исследования информационных систем и технологий. СПбГУ ИТМО. 2007, с. 29–32. http://vestnik.ifmo.ru/ntv/42/ntv_42.1.4.pdf
9. *Шалыто А. А.* Автоматное проектирование программ. Алгоритмизация и программирование задач логического управления. http://is.ifmo.ru/works/app-aplu/1/
10. *Дмитриев С.* Языково-ориентированное программирование: следующая парадигма // RSDN Magazine. 2005, № 5. http://www.rsdn.ru/article/philosophy/LOP.xml
11. *Fowler M.* A language Workbench in Action – MPS. http://martinfowler.com/articles/mpsAgree.html
12. *Красильников Н. Н., Шалыто А. А.* Мультиагентная система дорожного движения. Реализация на языке *Java* и текстовом языке автоматного программирования // СПбГУ ИТМО, 2007. http://is.ifmo.ru/works/_26_01_2008_multi_agent_sys.pdf
13. *Раер М. Г.* Автоматное расширение языка *C#* // Магистерская работа. СПбГУ ИТМО, 2006. http://vestnik.ifmo.ru/ntv/39/ntv_39.3.5.pdf

14. *The LINQ Project*. http://msdn.microsoft.com/en-us/netframework/aa904594.aspx
15. *Шалыто А. А.* Парадигма автоматного программирования. http://is.ifmo.ru/works/_2007_09_27_shalyto.pdf
16. *Новиков Ф. А.* Визуальное конструирование программ // Информационно-управляющие системы. 2005. № 6, с. 9–22. http://is.ifmo.ru/works/visualcons/
17. *.NET Framework*. http://msdn.microsoft.com/en-us/netframework/default.aspx
18. *Duffy J.* Professional .NET Framework 2.0 // Wrox Press.
19. *Lidin S.* Expert .NET 2.0 IL Assembler // Apress, 2006.
20. *Gamma E., Helm R., Johnson R., Vlissides J.* Design Patterns // Addison-Wesley Professional, 1994.
21. *Kernighan B. W., Ritchie D. M.* C Programming Language // Prentice Hall PTR, 1988.
22. *Новиков Ф.А.* Частное сообщение.
23. *Язык Nemerle*. http://nemerle.org
24. *Язык Scala*. http://scala-lang.org
25. *Booch G., Rumbaugh J., Jacobson I.* The Unified Modeling Language User Guide. Addison-Wesley Professional, 1998.
26. *Richter J.* CLR via C#. Microsoft Press, 2006.
27. *Маврин П. Ю.* Реализация диаграмм состояний // Бакалаврская работа. СПбГУ ИТМО, 2006. http://is.ifmo.ru/download/statecharts.pdf
28. *Harel D*. Statecharts: A Visual Formalism for Complex Systems // Science of Computer Programming. Vol. 8.1987. June, pp. 31–274.
29. *Шопырин Д. Г., Шалыто А. А.* Графическая нотация наследования автоматных классов // Программирование. 2007. № 5, с. 62 – 74. http://is.ifmo.ru/works/_12_12_2007_shopyrin.pdf
30. *Цымбалюк Е.А.* ECG: Генератор компиляторов // Курсовая работа. СПбГУ ИТМО, 2007.
31. *OpenMP*. http://openmp.org
32. *Berry G.* Esterel technologies. http://www.esterel-technologies.com/company/management-team/gerard-berry.html

Приложение 1. Грамматика языка *TABP*

```
#name TAVR_LOADER

program      -> decl_list;
    <<< Program             >>>

decl_list    -> decl *;
    <<< NodeList             >>>
decl      -> attrib_list func_header ";";
    <<< NodeArg("NT(DECL_PROC)")    >>>
decl      -> attrib_list func_header code_block;
    <<< NodeArg("NT(DECL_PROC)")    >>>
decl      -> $error "}";
    <<< NodeError            >>>
decl      -> $error ";";
    <<< NodeError            >>>

code_block    -> "{" stat_list "}";
    <<< CodeBlock            >>>
code_stat    -> stat;
    <<< CodeBlock            >>>

stat_list    -> stat *;
    <<< NodeList             >>>

// int a, b = 0, c;
stat       -> expr decl_item_list ";";
    <<< Node("NT(DECL_VAR)")      >>>

decl_item_list  -> decl_item % ",";
    <<< NodeList            >>>
decl_item    -> $string | $string "=" expr;
    <<< Node("NT(ARGUMENT)")      >>>
decl_item    -> $string "(" expr_list ")";
    <<< Node("NT(DECL_VAR_CONSTR)")    >>>

// returns and etc.
stat      -> ";";
    <<< Node              >>>
stat      -> expr ";";
stat      -> "return" ";";
    <<< Node("NT(RETURN)")        >>>
stat      -> "return" expr ";";
    <<< Node("NT(RETURN)")        >>>
stat      -> "return" $error;
    <<< Error("ERR(TVR1002)")      >>>

stat      -> expr $error;
```

```
    <<< Error("ERR(TVR1002)")      >>>
stat      -> $error ";";
    <<< NodeError            >>>

// if
stat      -> "if" "(" expr ")" code_stat; shift("else")
    <<< StatIf            >>>
stat      -> "if" "(" expr ")" code_stat "else" code_stat;
    <<< StatIf            >>>
stat      -> "if" "(" expr $error;
    <<< Error("ERR(TVR1003)")      >>>

// for
stat      -> "for" "(" for_stat ";" for_stat ";"
             for_stat ")" code_stat;
    <<< NodeArg("NT(FOR)") >>>
for_stat    -> expr;
for_stat    -> expr decl_item_list;
    <<< Node("NT(DECL_VAR)")      >>>
for_stat    -> ;
    <<< Node              >>>

// while
stat      -> "while" "(" expr ")" code_stat;
    <<< Node("NT(WHILE)")        >>>
stat      -> "do" code_stat "while" "(" expr ")";
    <<< Node("NT(DO_WHILE)")      >>>

// code block
stat      -> code_block;

// step
stat      -> step_type arg_name code_block;
    <<< Step              >>>
stat      -> step_type $string ";";
    <<< Step              >>>

// send
stat      -> "send" send_event_val send_to ";";
    <<< NodeArg("NT(SEND)")        >>>
stat      -> "post" send_event_val send_to ";";
    <<< NodeArg("NT(POST)")        >>>

send_event_val  -> ;
    <<< Node              >>>
send_event_val  -> typename send_event_args;
    <<< NodeArg("NT(ARGUMENT)")      >>>
send_event_args -> [ "(" expr_list ")" ];
    <<< Node              >>>
send_to      -> [ "to" expr ];
    <<< Node              >>>

// on
```

```
stat      -> "on" typename code_block;
    <<< NodeArg("NT(ON)")        >>>
stat      -> "otherwise" code_stat;
    <<< Node("NT(OTHERWISE)")      >>>

// += send
stat      -> expr "+=" "send" expr ";";
    <<< Node("NT(REG_SEND)")      >>>
stat      -> expr "+=" expr ";";
    <<< Node("NT(REG_SEND)")      >>>
stat      -> expr "+=" "post" expr ";";
    <<< Node("NT(REG_POST)")      >>>
stat      -> expr "-=" expr ";";
    <<< Node("NT(UNREG)")        >>>

// break, continue
stat      -> "break" ";";
    <<< Node("NT(BREAK)")        >>>
stat      -> "continue" ";";
    <<< Node("NT(CONTINUE)")      >>>
stat      -> "repeat" ";";
    <<< Node("NT(REPEAT)")        >>>
stat      -> "goto" $string ";";
    <<< Node("NT(GOTO)")        >>>

stat      -> "relax" ";";
    <<< Node("NT(RELAX)")        >>>
stat      -> "relax" $string ";";
    <<< Node("NT(RELAX)")        >>>
stat      -> "relax" "step" ";";
    <<< Node("NT(RELAX_STEP)")      >>>

// FUNCTION HEADER
func_header    -> typename $string "(" arg_list ")";
    <<< FuncHeader            >>>

arg_list    -> ["void"]; reduce("," ")")
    <<< Node              >>>
arg_list    -> arg_type arg_name % ",";
    <<< ArgList             >>>
arg_type    -> "inout" typename;
    <<< Node("ET(INOUT)")        >>>
arg_type    -> "out" typename;
    <<< Node("ET(OUT)")          >>>
arg_type    -> typename;
arg_name    -> $string;
arg_name    -> ;
    <<< ArgEmpty            >>>

// TYPENAME
typename    -> "void";
    <<< Node("NT(NAME)", L"void")    >>>
typename    -> "bool";
```

```
    <<< Node("NT(NAME)", L"bool")    >>>
typename    -> "char";
    <<< Node("NT(NAME)", L"char")    >>>
typename    -> "string";
    <<< Node("NT(NAME)", L"string")    >>>

typename    -> "byte";
    <<< Node("NT(NAME)", L"byte")    >>>
typename    -> "word";
    <<< Node("NT(NAME)", L"word")    >>>
typename    -> "int";
    <<< Node("NT(NAME)", L"int")    >>>
typename    -> "uint";
    <<< Node("NT(NAME)", L"uint")    >>>
typename    -> "long";
    <<< Node("NT(NAME)", L"long")    >>>
typename    -> "ulong";
    <<< Node("NT(NAME)", L"ulong")    >>>
typename    -> "float";
    <<< Node("NT(NAME)", L"float")    >>>
typename    -> "double";
    <<< Node("NT(NAME)", L"double")    >>>

typename    -> "global";
    <<< Node("ET(GLOBAL)")        >>>
typename    -> $string;  reduce
    <<< Node("NT(NAME)")        >>>
typename    -> typename "." $string;
    <<< Node("NT(SUBNAME)")        >>>
typename    -> "typeof" expr;
    <<< Node("ET(TYPEOF)")        >>>
typename    -> typename "[" "]";
    <<< Node("ET(ARRAY)")        >>>
typename    -> "(" typename ")";

// STRUCT
decl      -> attrib_list "struct" $string
             "{" struct_decl_list "}";
    <<< NodeArg("NT(STRUCT)")  >>>
struct_decl_list -> struct_decl *;
    <<< NodeList            >>>

// struct variables
struct_decl    -> attrib_list typename struct_item_list ";";
    <<< NodeArg("NT(DECL_MEMBER_VAR)") >>>
struct_item_list -> struct_item % ",";
    <<< NodeList            >>>
struct_item    -> $string | $string "=" expr;
    <<< Node("NT(ARGUMENT)")      >>>

// struct method
struct_decl    -> attrib_list func_header code_block;
    <<<  NodeArg("NT(DECL_METHOD)")    >>>
```

```
struct_decl    -> attrib_list func_header ";";
    <<<  NodeArg("NT(DECL_METHOD)")    >>>
attrib_list    -> attrib *;
    <<< NodeList            >>>
attrib      -> "public";
    <<< Node("NT(ATTRIB_PUBLIC)",    L"public")    >>>
attrib      -> "private";
    <<< Node("NT(ATTRIB_PRIVATE)",    L"private")    >>>
attrib      -> "protected";
    <<< Node("NT(ATTRIB_PROTECTED)",  L"protected")  >>>
attrib      -> "static";
    <<< Node("NT(ATTRIB_STATIC)",    L"static")    >>>
attrib      -> "extern";
    <<< Node("NT(ATTRIB_EXTERN)",    L"extern")    >>>

struct_decl    -> $error ";";
    <<< NodeError            >>>

// constructor
struct_decl    -> attrib_list constr_header code_block;
    <<< NodeArg("NT(DECL_CONSTR)")    >>>
struct_decl    -> attrib_list constr_header ";";
    <<< NodeArg("NT(DECL_CONSTR)")    >>>
constr_header  -> $string "(" arg_list ")";
    <<< FuncHeader            >>>

// operators
struct_decl    -> attrib_list operator_header code_block;
    <<< NodeArg("NT(DECL_OPERATOR)")  >>>
struct_decl    -> attrib_list operator_header ";";
    <<< NodeArg("NT(DECL_OPERATOR)")  >>>
operator_header -> "operator" "=" "(" arg_list ")";
    <<< OperatorHeader(L"=")      >>>
operator_header  -> "operator" typename "(" arg_list ")";
    <<< OperatorHeader(L"(T)")      >>>

// steps
struct_decl    -> step_type arg_name code_block;
    <<< Step              >>>
struct_decl    -> step_type $string ";";
    <<< Step              >>>

step_type    -> "step" | "loop" | "fixed" | "subauto";

// out
struct_decl    -> "out" typename $string ";";
    <<< Node("NT(OUT)")          >>>

// AUTO
decl      -> attrib_list "auto" $string
             "{" struct_decl_list "}";
    <<< NodeArg("NT(AUTO)")  >>>
```

```
    // EVENT
    decl      -> attrib_list "event" $string event_body;
        <<< NodeArg("NT(EVENT)")      >>>
    event_body    -> ["{" struct_decl_list "}"];
        <<< Node              >>>

    decl      -> ";";
        <<< Node              >>>

    right  "ref" "typeof" "sizeof";
    right   "=" "+=" "-=" "*=" "/=" "%=" "<<=" ">>=" "&=" "|="
"^=";
    right  "?";
    left   "||";
    left   "&&";
    left   "<" "<=" ">" ">=" "==" "!=";
    left   "<<" ">>";
    left   "&" "|" "^";
    left   "+" "-";
    left   "*" "/" "%";
    right  unar;
    left   "++" "--";
    left   "[" "(";
    left   ".";

    expr      -> $string;
        <<< Node("NT(NAME)")        >>>
    expr      -> expr "." $string;
        <<< Node("NT(SUBNAME)")        >>>
    expr      -> expr "." "auto";
        <<< Node("NT(SUBNAME)", L"auto")  >>>
    expr      -> expr "." $error;
        <<< Error("ERR(TVR1006)")      >>>

    expr      -> expr "=" expr;
        <<< Node("ET(ASSIGN)")        >>>

    // ARITHMETIC OPERATIONS

    /*
    expr      -> expr "+=" expr;
        <<< Node("ET(ASSIGN_ADD)")      >>>
    expr      -> expr "-=" expr;
        <<< Node("ET(ASSIGN_SUB)")      >>>
    expr      -> expr "*=" expr;
        <<< Node("ET(ASSIGN_MUL)")      >>>
    expr      -> expr "/=" expr;
        <<< Node("ET(ASSIGN_DIV)")      >>>
    expr      -> expr "%=" expr;
        <<< Node("ET(ASSIGN_MOD)")      >>>

    expr      -> expr "<<=" expr;
        <<< Node("ET(ASSING_SHL)")      >>>
```

```
expr      -> expr ">>=" expr;
    <<< Node("ET(ASSING_SHR)")      >>>

expr      -> expr "&=" expr;
    <<< Node("ET(ASSING_BIT_AND)")    >>>
expr      -> expr "|=" expr;
    <<< Node("ET(ASSING_BIT_OR)")    >>>
expr      -> expr "^=" expr;
    <<< Node("ET(ASSING_BIT_XOR)")    >>>

expr      -> expr "&" expr;
    <<< Node("ET(BIT_AND)")        >>>
expr      -> expr "|" expr;
    <<< Node("ET(BIT_OR)")        >>>
expr      -> expr "^" expr;
    <<< Node("ET(BIT_XOR)")        >>>

expr      -> expr "<<" expr;
    <<< Node("ET(SHL)")          >>>
expr      -> expr ">>" expr;
    <<< Node("ET(SHR)")          >>>
/**/
expr      -> expr "+" expr;
    <<< Node("ET(ADD)")          >>>
expr      -> expr "-" expr;
    <<< Node("ET(SUB)")          >>>
expr      -> expr "*" expr;
    <<< Node("ET(MUL)")          >>>
expr      -> expr "/" expr;
    <<< Node("ET(DIV)")          >>>
expr      -> expr "%" expr;
    <<< Node("ET(MOD)")          >>>

expr      -> expr "||" expr;
    <<< Node("ET(LOG_OR)")         >>>
expr      -> expr "&&" expr;
    <<< Node("ET(LOG_AND)")        >>>
expr      -> expr "==" expr;
    <<< Node("ET(EQ)")             >>>
expr      -> expr "!=" expr;
    <<< Node("ET(NOT_EQ)")         >>>
expr      -> expr "<"  expr;
    <<< Node("ET(LESS)")           >>>
expr      -> expr "<=" expr;
    <<< Node("ET(LESS_EQ)")        >>>
expr      -> expr ">"  expr;
    <<< Node("ET(GREATER)")        >>>
expr      -> expr ">=" expr;
    <<< Node("ET(GREATER_EQ)")      >>>

expr      -> "!" expr; prior(unar)
    <<< Node("ET(LOG_NOT)")        >>>
expr      -> "~" expr; prior(unar)
```

```
    <<< Node("ET(BIT_NOT)")        >>>
expr      -> "+" expr; prior(unar)
    <<< Node("ET(POS)")          >>>
expr      -> "-" expr; prior(unar)
    <<< Node("ET(NEG)")          >>>

expr      -> "++" expr;
    <<< Node("ET(PRE_INC)")        >>>
expr      -> "--" expr;
    <<< Node("ET(PRE_DEC)")        >>>
expr      -> expr "++";
    <<< Node("ET(POST_INC)")       >>>
expr      -> expr "--";
    <<< Node("ET(POST_DEC)")       >>>

expr      -> "true";
    <<< Node("ET(BOOLEAN)",  L"true")  >>>
expr      -> "false";
    <<< Node("ET(BOOLEAN)",  L"false")  >>>
expr      -> "null";
    <<< Node("ET(NULL)")        >>>
expr      -> "this";
    <<< Node("ET(THIS)")        >>>
expr      -> "event";
    <<< Node("ET(EVENT)")        >>>
expr      -> $integer;
    <<< Node("ET(INT)")          >>>
expr      -> $float;
    <<< Node("ET(FLOAT)")        >>>
expr      -> $double;
    <<< Node("ET(DOUBLE)")        >>>
expr      -> $char;
    <<< Node("ET(CHAR)")        >>>
expr      -> $literal;
    <<< Node("ET(STRING)")        >>>

expr      -> "auto";
    <<< Node("NT(NAME)", L"auto")    >>>

expr      -> typename; reduce("." "[" ")")
expr      -> "sizeof" expr;
    <<< Node("ET(SIZEOF)")        >>>

expr      -> "new" typename "(" expr_list ")";
    <<< Node("ET(NEW_AUTO)")      >>>
expr      -> "new" typename "[" expr "]";
    <<< Node("ET(NEW_ARRAY)")      >>>

expr      -> "(" expr ")" expr; prior(unar)
    <<< Node("ET(TYPE_CAST)")      >>>
expr      -> expr "?" expr ":" expr;
    <<< Node("ET(IF)")          >>>
```

```
expr      -> "(" expr ")"; prior(unar) shift($string)
expr      -> expr "(" expr_list ")";
    <<< Node("ET(CALL)")        >>>
expr      -> expr "[" expr_list "]";
    <<< Node("ET(ARRAY)")        >>>

expr_list    -> expr % ",";
    <<< NodeList             >>>
expr_list    -> ;
    <<< NodeList             >>>

//*        : PNODE
decl,      decl_list,
decl_item,    decl_item_list,
stat,      stat_list,
func_header,  constr_header,    operator_header,
code_block,    code_stat,
arg_type,    arg_list,
typename,    expr,        expr_list,
struct_decl,  struct_decl_list,
struct_item_list, struct_item,
attrib,      attrib_list,
send_event_args, send_event_val,  send_to,
event_body,
for_stat    : PNODE

"this"      |
"step"      | "loop"  | "fixed"  | "subauto"  |
$string     | $integer  | $float  | $double  |
$char      |
$literal
    <<< StrVal >>>

arg_name,
"this",
step_type, "step", "loop", "fixed", "subauto",
$string,
$integer, $float, $double,
$char,
$literal    : STRVAL

skip       = \s
cline      = //
cbegin      = /{*}
cend      = {*}/

$string      = {\S^\y\d@}{\S^\y@}*
$integer    = \d+
$float      = (\d+.\d*|\d*.\d+)[{eE}[{+\-}]\d+]f
$double      = (\d+.\d*|\d*.\d+)[{eE}[{+\-}]\d+]
$char      = '(\\{^\n}|{^\\\n\'})'
$literal    = "(\\{^\n}|{^\\\n\"})*"
```

Приложение 2. Реализация автомата `Game`

```
/**
  File: game.tavr
  Description: Game automaton.

  Copyright: Evgeniy A. Cymbalyuk, 2008
*/

event GameUpdate {
  float   elapsedTime; // in sec
  Game    game;
}

event GameRender {
  Game    game;
}

auto Game {
  Engine engine;

  EngShader shaderCircle;
  EngShader shaderTransform;

  //EngTexture  textureTest;

  Game(void) {
    engine = new Engine("TavrGame v1.0", 800, 600);
    engine.update += this;
    engine.resize += this;
    engine.mouse  += new GameController(this);

    shaderCircle   =
      engine.loadPixelShader("data\\circle.psh");
    shaderTransform  =
      engine.loadVertexShader("data\\transform.vsh");

    shaderTransform.set("width",  width);
    shaderTransform.set("height", height);
  }

  void run(void) {
    engine.run();

    shaderCircle.release();
    shaderTransform.release();

    engine.release();

    // break reference cycles
    engine = null;
    for (int i=0; i<balls.length; ++i) {
```

```
    update -= balls[i];
    render -= balls[i];
  }
  balls  = null;
}

// game options
float windowWidth   = 800.f;
float windowHeight  = 600.f;

float width     = 1024.f;
float height    = 768.f;

EngV2 wnd2scr    = EngV2(width / windowWidth,
                         height / windowHeight);

EngV2 defGravityCenter = EngV2(width / 2.f, height / 2.f);
EngV2 gravityCenter    = defGravityCenter;
bool  gravityEnable    = true;
float gravityCoef      = 10000.f;

float maxSpeed       = 25.f;
float maxAcceler     = 75.f;
float massCoef       = 0.1f;
float rigidCoef      = 100000000.f;

float mergeBarrier   = 0.15f; //0.2f
float ballRadius     = 19.f;
float ballRadiusRangeCoef = 0.4f;
float explodeBarrier =
  ballRadius * 5.5f * ballRadiusRangeCoef;

float timeFactor     = 3.f;
float changeGravTimeCoef = 1.5f;
float changeSingleGravTimeCoef = 1.8f;

int   explodeBallsCount = 20;
float explodeSpeed   = maxAcceler * 2.f;
int   ballsCountMax  = 100;
int   ballsCount;

GameBall [] balls;

// game events
out GameUpdate    update;
out GameRender    render;

void excludeBall(GameBall ball) {
  for (int i=0; i < balls.length; ++i) {
    if (balls[i] != ball) {
      continue;
    }
```

```
    balls[i] = null;
    update  -= ball;
    render  -= ball;

    --ballsCount;
    return ;
  }
}

void includeBall(GameBall ball) {
  for (int i=0; i<balls.length; ++i) {
    if (balls[i] != null) {
      continue;
    }

    balls[i] = ball;
    update  += ball.Update;
    render  += ball.Render;
    return ;
  }
}

step {
  // create balls
  balls = new GameBall[ballsCountMax];

  for (int i=0; i<ballsCountMax; ++i) {
    balls[i] = new GameBall(this);

    update  += balls[i].Update;
    render  += balls[i].Render;
  }
  ballsCount = ballsCountMax;
}

fixed {
  on EngUpdate {
    update(timeFactor * event.time / 1000.f, this);

    engine.beginScene();
    engine.clear(EngColor(1.f, 1.f, 1.f));

    render(this);

    engine.endScene();
  }

  on EngResize {
    windowWidth   = event.width;
    windowHeight  = event.height;

    if (width > height) {
      width  = height * windowWidth / windowHeight;
```

```
      } else {
        height = width * windowHeight / windowWidth;
      }

      wnd2scr =
        EngV2(width / windowWidth, height / windowHeight);

      gravityCenter = defGravityCenter =
        EngV2(width / 2.f, height / 2.f);

      if (event.isReset) {
        shaderCircle.release();
        shaderTransform.release();

        shaderCircle     =
          engine.loadPixelShader("data\\circle.psh");
        shaderTransform  =
          engine.loadVertexShader("data\\transform.vsh");
      }

      shaderTransform.set("width",  width);
      shaderTransform.set("height", height);
    }
  }
}

int main(string [])
{
  new Game().run();
  return 0;
}
```

Приложение 3. Реализация автомата `GameController`

```
/**
  File: game_ctrl.tavr
  Description: Tavr Game input controller.

  Copyright: Evgeniy A. Cymbalyuk, 2008
*/

event GameChangeGravity {
  Game   game;
  bool   enable;
  EngV2   pos;
}

auto GameController {
  Game game;

  GameController(Game game) {
    this.game = game;
```

```
}

float clickTimer        = 150.f;
float doubleClickTimer  = 350.f;

float lastDown, lastClick;

GameBall captureBall(inout EngV2 pos) {
  for (int i=0; i<game.balls.length; ++i) {
    if (game.balls[i] == null) {
      continue;
    }

    GameBall ball = game.balls[i];
    float dist_2 = pos.sub(ball.pos).length_2();
    if (dist_2 < ball.radius * ball.radius) {
      return ball;
    }
  }

  return null;
}

fixed Initial {
  on EngMouse {
    if (event.isRight(true)) {
      game.gravityCenter = game.wnd2scr.mul(event.pos);
      game.timeFactor    =
        game.timeFactor * game.changeGravTimeCoef;
      relax Gravity;
    }
    if (event.isMiddle(true)) {
      game.gravityEnable = false;
      relax DisableGravity;
    }

    if (event.isLeft(true)) {
      EngV2 pos = game.wnd2scr.mul(event.pos);

      GameBall ball = captureBall(pos);

      if (ball != null) {
        send GameChangeGravity(game, true, pos) to ball;

        fixed {
          on EngMouse {
            if (event.isLeft(false)) {
              send GameChangeGravity(game, false, pos)
                to ball;
              break;
            }
            if (event.isMove()) {
              pos = game.wnd2scr.mul(event.pos);
```

```
                send GameChangeGravity(game, true, pos)
                  to ball;
              }
            }
          }
        }
      }
    }

    fixed Gravity {
      on EngMouse {
        if (event.isRight(false)) {
          game.gravityCenter = game.defGravityCenter;
          game.timeFactor    =
            game.timeFactor / game.changeGravTimeCoef;
          relax Initial;
        }
        if (event.isMove()) {
          game.gravityCenter = game.wnd2scr.mul(event.pos);
        }
      }
    }

     fixed DisableGravity {
       on EngMouse {
         if (event.isMiddle(false)) {
           game.gravityEnable = true;
           relax Initial;
         }
       }
     }
}
```

Приложение 4. Реализация автомата `GameBall`

```
/**
  File: game_ball.tavr
  Description: Tavr game Ball auto.

  Copyright: Evgeniy A. Cymbalyuk, 2008
*/

auto GameBall {
  EngV2 pos;
  EngV2 speed;
  float radius;
  EngColor color;

  GameBall(Game game) {
    pos.x = random() * game.width;
    pos.y = random() * game.height;
```

```
    radius = (game.ballRadiusRangeCoef * random() + 1.f)
             * game.ballRadius;
    color = EngColor(0.45f + 0.45f * random(),
                     0.15f + 0.75f * random(),
                     0.2f + 0.7f * random());
  }

  void update(Game game, float elapsedTime) {
    // calc acceleration
    EngV2 acc;
    float mass = game.massCoef * radius * radius;

    // ignore close gravity center
    EngV2 toGravity = game.gravityCenter.sub(pos);
    if (game.gravityEnable && toGravity.length() > 0.0001f){
      float grav =
        mass * game.gravityCoef / toGravity.length();

      acc.updateAdd( toGravity.length(grav / mass) );
    }

    // check collision
    for (int i=0; i < game.balls.length; ++i) {
      GameBall ball = game.balls[i];
      if (ball == this || ball == null) {
        continue;
      }

      float dist_2  = ball.pos.sub(pos).length_2();
      float minDist = radius + ball.radius;
      if (dist_2 + 0.001f > minDist * minDist) {
        continue;
      }

      float dist  = sqrt(dist_2);
      float power = game.rigidCoef * (minDist - dist);

      EngV2 dir   = pos.sub(ball.pos);
      dir.updateLength(power / mass);

      acc.updateAdd( dir.mul(elapsedTime) );
    }

    acc.updateMaxLength(game.maxAcceler);

    speed.updateAdd( acc.mul(elapsedTime) );
    float speedLength = speed.length();
    if (speedLength > game.maxSpeed) {
      speed.updateMul( (speedLength - min(speedLength
        - game.maxSpeed, elapsedTime * game.maxAcceler))
        / speedLength );
    }
```

```
  pos.updateAdd( speed.mul(elapsedTime) );
}

subauto Update {
  on GameUpdate {
    Game game = event.game;
    update(game, event.elapsedTime);

    for (int i=0; i<game.balls.length; ++i) {
      GameBall ball = game.balls[i];
      if (ball == null || ball == this) {
        continue;
      }

      float dist_2  = ball.pos.sub(pos).length_2();
      float minDist = radius + ball.radius;
      if (dist_2 + 0.001f > minDist * minDist) {
        continue;
      }

      // merge balls
      EngV3 color3 = color.toV3();
      EngV3 ball_color3 = ball.color.toV3();

      EngV3 t = color3.sub(ball_color3);
      float color_dist =
        max(abs(t.x), abs(t.y), abs(t.z));

      if (color_dist < game.mergeBarrier) {
        game.excludeBall(ball);

        float blend =
          (radius * radius) / (ball.radius * ball.radius);

        color  = EngBlend(blend, color, ball.color);
        pos    = EngBlend(blend, pos,   ball.pos);
        speed  = EngV2();

        float dstRadius = sqrt(radius * radius
          + ball.radius * ball.radius);
        radius = max(radius, ball.radius);
        float radiusSpeed = dstRadius - radius;

        fixed {
          on GameUpdate {
            float toAdd = radiusSpeed * event.elapsedTime;
            if (toAdd > dstRadius - radius) {
              radius = dstRadius;
              break;
            }

            radius = radius + toAdd;
```

```
            update(game, event.elapsedTime);
          }
        }
        break;
      }

      // explode ball
      if (radius > game.explodeBarrier) {
        game.excludeBall(this);

        int to_create = min(game.explodeBallsCount,
          game.ballsCountMax - game.ballsCount);

        for (int k=0; k<to_create; ++k) {
          GameBall b = new GameBall(game);
          b.pos = pos.add(
            EngV2(radius * (2.f * random() - 1.f),
            radius * (2.f * random() - 1.f)) );
          b.speed =
            b.pos.sub(pos).length(game.explodeSpeed);

          game.includeBall(b);
        }

        break;
      }
    }
  }
  on GameChangeGravity {
    if (!event.enable) {
      break;
    }

    EngV2 dstPos = event.pos;

    fixed {
      on GameUpdate {
        EngV2 dir = dstPos.sub(pos);
        float len = dir.length();
        if (len > 0.01f) {
          float speedCoef =
            max(len / event.game.maxSpeed, 1.3f);

          pos.updateAdd(
            dir.length(speedCoef * event.elapsedTime *
            min(len, event.game.maxSpeed)) );
        }
      }
      on GameChangeGravity {
        if (!event.enable) {
          speed = event.game.gravityCenter.sub(this.pos)
            .length(2.f * event.game.explodeSpeed);
```

```
            break;
          }

          dstPos = event.pos;
        }
      }
    }
  }

  subauto Render {
    on GameRender {
      EngShader vs = event.game.shaderTransform;
      EngShader ps = event.game.shaderCircle;

      ps.enable();
      ps.set("center", pos);
      ps.set("radius_2", radius * radius);
      ps.set("color", color);

      vs.enable();

      event.game.engine.renderQuad(pos.x - radius,
        pos.y - radius, 2.f * radius, 2.f * radius);

      ps.disable();
      vs.disable();
    }
  }
}
```