Санкт-Петербургский государственный университет
информационных технологий, механики и оптики
Кафедра «Компьютерные технологии»

Е.В. Смирнов

Бакалаврская работа

Приведение объектно-ориентированных программ к автоматному виду

Научный руководитель: докт. техн. наук, профессор Шалыто А.А.

Санкт-Петербург
2009

Оглавление

Введение

В рамках парадигмы автоматного программирования, предложенной в работе [1], построение и реализация алгоритма выполняется в виде конечного автомата. Основное отличие такого подхода от традиционного заключается в том, что автоматная программа в своей основе содержит не понятия «действие» и «условие», а понятие «состояние». По этой причине автоматные программы также называют «программами с явно выделенными состояниями».

В случае систем, основанных на событиях, автоматный подход является достаточно естественным и широко применяется (в парсерах, драйверах устройств, микроконтроллерах и т. д.). Для вычислительных алгоритмов понятие состояния не столь естественно, однако автоматный подход может быть применен и к ним. Вопросам реализации программных систем с использованием явного выделения состояний посвящен ряд работ, например [2, 3].

Использование явного выделения состояний позволяет упростить решение многих задач, связанных с отладкой, документацией и демонстрацией работы алгоритма. Можно указать следующие преимущества автоматных программ:

Наглядность. Граф переходов автомата может быть понят даже неспециалистом.

Простота документирования. Автомат полно и наглядно (в отличие от кода на каком-либо языке программирования) документирует поведение программы.

Граф переходов является моделью программы. Для обработки программы разнообразными алгоритмами анализа и верификации требуется построить модель, с которой они будут работать. Авторы работы [4] выделяют следующие возможные типы моделей:

- абстрактное синтаксическое дерево (*AST*);
- абстрактный семантический граф (*ASG*);

- граф потока управления (*CFG*);
- граф зависимости по данным (*DDG*);
- представление на основе статического однократного присваивания (*SSA*).

В работе [4] делается вывод о том, что наиболее оптимальным и универсальным является представление программы в виде графа потока управления, построенного на основе *SSA*-представления. В свою очередь, представление программы на основе автомата является одним из видов *CFG*. Использование же *SSA*-представления может быть легко добавлено в любую из указанных моделей (алгоритм добавления можно найти во многих пособиях по созданию компиляторов, например, в книге [5]).

При работе с программами, написанными с использованием традиционных подходов, первичным является сам код. Построение модели осуществляется в виде дополнительного этапа и может являться весьма непростой задачей. Далее все алгоритмы анализа и верификации работают с полученной моделью, причем надежность их результатов будет сильно зависеть от точности модели. Полученные результаты их работы необходимо перенести из модели обратно в исходную программу, что тоже может быть не всегда легко реализуемо.

В случае автоматного подхода первичной является именно модель. Создание программы можно разбить на два этапа: построение графа переходов и создание по нему кода. Подробное описание первого этапа можно найти в книге [6], где на конкретных примерах продемонстрирован полный цикл проектирования программной системы на основе автоматного подхода. Второй этап разработки может быть выполнен автоматически с использованием существующих средств, например, в работе [7]. При данном подходе модель будет идеально описывать программную систему, что положительно скажется на точности работающих с ней алгоритмов. Более подробно данная тема на примере автоматической верификации рассмотрена в работах [8,9].

Малая распространенность автоматного подхода препятствует использованию перечисленных преимуществ. Если при написании новой программы с нуля программист еще может выбрать использование данного подхода, то ручная переделка уже готового кода является сложной и дорогостоящей задачей. По этой причине достаточно остро стоит вопрос о наличие формального алгоритма преобразования «традиционной» программы в автоматную. При наличии такого алгоритма станет возможным создание инструментальных средств, которые возьмут эту работу на себя.

В работе [10] был предложен метод преобразования итеративных программ к автоматному виду, а в работе [11] те же авторы предложили метод для преобразования кода, содержащего рекурсивные вызовы. Само преобразование осуществлялось вручную. В работе [12] были формализованы преобразования базовых блоков программ, таких как операторы ветвления и циклов, вызова процедуры и т. д. Для каждого оператора был предоставлен соответствующий ему фрагмент автомата, с доказательством его корректности. В этой работе также была описана процедура построения *обратных автоматов,* позволявших осуществлять трассировку (пошаговое выполнение) программы назад.

Все перечисленные работы рассматривали только императивные программы, никак не затрагивая объектно-ориентированные. Этот факт делает невозможным применение методов преобразования к коду, написанному на таких языках, как *C#* и *Java*, которые являются полностью объектно-ориентированными и не предусматривают возможности написания программ без использования классов.

Целью данной работы является заполнение данного пробела. В последующих главах будут рассмотрены алгоритмы преобразования кода, использующего разнообразные аспекты объектно-ориентированного программирования, такие как наследование, полиморфизм, интерфейсы, конструкторы, деструкторы, и некоторые другие.

Глава 1. Основные понятия

Для формального описания метода преобразования программы в систему взаимодействующих конечных автоматов потребуется описать исходную программу (разд. 1.1) и определить понятия *фрагмента автомата*, *замыкания*, *обратного автомата* (разд. 1.2).

1.1. Исходная программа

Исходные программы далее будем рассматривать на языках *C++*, *Java* и, иногда, *C#*, так как они на данный момент являются наиболее широко распространенными объектно-ориентированными языками. Каждый из перечисленных языков имеет свои действия над классами. При этом отметим, что некоторые концепции *C++* и *Java* являются взаимоисключающими (такие как деструкторы и `finalize`-функции), а некоторые значительно отличаются (реализация наследования, интерфейсов). Поэтому не удается совместить все возможные ситуации применения конструкций языка в рамках одного общего псевдоязыка. В дальнейшем все примеры будут приводиться на том языке, для которого они наиболее типичны.

Далее будем полагать, что известен метод корректного построения фрагментов автоматов для операторов, находящихся внутри тела функции, таких как:

- оператор присваивания;
- последовательность операторов;
- полный и укороченный операторы ветвления;
- цикл с предусловием;
- вызов процедуры.

Их реализация подробно описана в работе [12]. Там же описана процедура приведения кода к виду, содержащему только перечисленные операторы.

В разд. 3.1 будет представлен несколько модифицированный (по сравнению с предложенным в работе [12]) алгоритм построения автоматов

для оператора вызова функций.

1.2. Фрагменты автоматов

Фрагмент автомата — набор состояний и переходов. Переходы, у которых не определено начальное либо конечное состояние, называются соответственно *входами* и *выходами* фрагмента.

Входы и выходы позволяют объединять фрагменты посредством операции замыкания — вход одного фрагмента и выход другого объединяются в один переход, у которого определено и начальное, и конечное состояние.

Для преобразования программы в конечный автомат будем строить фрагменты автоматов, соответствующих операторам, а затем объединять их во фрагменты, соответствующие последовательности операторов и т. д.

Данные определения взяты из работы [12].

1.3. Обратный автомат

Для демонстрационных и отладочных целей может оказаться полезной возможность выполнять программу по шагам не только вперед, но и назад. Такое выполнение в обратную сторону называется *обратной трассировкой* (*backtracking*).

Сама идея обратного выполнения программы не является новой. Она впервые была реализована в 1960 году в отладчике *EXDAMS* для языка программирования *Fortran*, однако широкого распространения не получила. На эту тему был написан ряд работ (например, [13]) и была предпринята попытка интеграции обратного трассировщика в набор средств разработки *GCC* (*GNU Compiler Collection*), однако она ни к чему не привела (на данный момент в *GCC* поддержки обратной трассировки нет). Автору настоящей работы удалось найти только один современный отладчик, предоставляющий функцию обратного выполнения [14], однако он является коммерческим продуктом с закрытыми кодами и документацией.

Существуют два подхода к реализации восстановления состояний

программы, выделенные в работе [13]:

- с полным сохранением истории выполнения. С каждой операцией, изменяющей значения переменных, ассоциируется *изменяемое множество* (*change set*). При прямом проходе перед выполнением каждой такой операции элементы ее изменяемого множества размещаются в стеке, откуда извлекаются при обратном проходе;
- структурный. В этом случае изменяемое множество ассоциируется с управляющими операторами (ветвления, цикла и т. д.). Как и в первом случае, при прямом проходе значения переменных из изменяемого множества сохраняются, а при обратном проходе – восстанавливаются. При шаге назад выполняется переход не к предыдущему оператору, а к оператору, предшествовавшему текущему блоку.

Различие этих двух подходов можно пояснить на следующем примере. Рассмотрим фрагмент программы:

```
оператор1;
while(условие)
{
     оператор2;
     оператор3;
}
```

Предположим, на некоторой итерации цикла был выполнен `оператор3`. После этого потребовалось сделать один шаг назад. При *использовании подхода с полным сохранением истории* был бы выполнен переход к состоянию, предшествовавшему выполнению `оператора3` на данной итерации (после выполнения `оператора2`). Продолжая делать шаги назад, обращение тела цикла было бы выполнено столько раз, сколько раз оно было выполнено при прямом проходе. После этого был бы сделан переход к обращению `оператора1` и всего, что ему предшествовало.

При *использовании структурного подхода* сразу было бы восстановлено состояние, соответствовавшее моменту входа в цикл (после выполнения `оператора1`). Если бы затем понадобилось восстановить состояние на какой-то конкретной итерации цикла, то пришлось бы выполнять тело необходимое число раз для того, чтобы достичь заданного положения.

Преимуществом структурного подхода является намного меньший объем используемой памяти для сохранения состояний. Однако этот подход не гарантирует правильного обращения в случае зависимости выполнения программы от внешних воздействий. Если бы в указанном выше примере внутри цикла находились операторы ветвления с условиями, зависящими от действий пользователя (например, проверка данных от клавиатуры), и эти условия не совпали бы с условиями первого выполнения, то результат мог бы сильно измениться и не соответствовал бы уже «одному шагу назад».

Перейдем теперь к рассмотрению автоматных программ.

Для сохранения хода выполнения программы в этом случае будем использовать *обратный автомат*.

Автомат, обратный данному – такой автомат, чье множество состояний совпадает с состояниями исходного, все действия в состояниях являются обратными к соответствующим действиям прямого автомата, а все переходы ведут в обратную сторону.

Действие обратное данному – такое действие, после выполнения которого программа придет к состоянию на момент до выполнения прямого действия.

Прямой и обратный автоматы будут иметь общее множество состояний и отличаться только множеством переходов. При этом для трассировки вперед будет использоваться граф переходов прямого автомата, а для трассировки назад – обратного.

Действия в состояниях обратного автомата должны обращать соответствующие действия в состояниях прямого. В работах [12, 13] предложены следующие способы обращения:

- ручной реализацией обратных действий. Так, например, для прямого действия `x++` обратным будет `x--`. Недостатки этого подхода: не для каждого действия может быть построено обратное, необходимость большого объема ручной работы;
- автоматическим обращением с использованием изменяемых множеств, рассмотренным выше. Этот подход не работает в том случае, если невозможно точно указать изменяемое множество для каждой операции. К сожалению, это случается довольно часто при вызове библиотечных функций, которые изменяют скрытое от пользователя состояние библиотеки.

Возможность выполнения обратной трассировки позволяет существенно сократить затрачиваемое на отладку время. Сравнение процессов «обычной» отладки и отладки с использованием шагов назад можно найти в работе [15]. При этом можно указать следующие достоинства отладки с применением обратной трассировки:

- возможность многократного исследования поведения произвольного фрагмента программы;
- возможность восстанавливать прошлые состояния программы без ее перезапуска;
- значительное сокращение времени отладки при поиске трудновоспроизводимых ошибок, так как нет необходимости при каждом перезапуске ожидать выполнения всех условий их проявления.

1.4. Модель данных

Модель данных представляет собой структуру, которая хранит в себе все переменные, объявленные в программе, и предоставляет к ним доступ. Она предназначена для хранения вычислительных состояний [6] программы и для их отделения от управляющих состояний, которые представлены в виде автомата или системы автоматов. Помимо этого, модель данных используется

для преодоления некоторых ограничений языка. Автор работы [1] предлагает реализовать код автоматной программы с использованием оператора `switch`. В силу правил всех рассматриваемых языков переменная, объявленная внутри одной из меток `case` (внутри кода одного из состояний) будет локальной для этой блока и не будет видна в остальных. По этой причине нельзя просто перенести объявления переменных напрямую в автомат из исходного текста программы

Построение модели данных состоит из двух этапов: определения соответствующей структуры, содержащей в себе объявления всех переменных, и приведение исходной программы к виду, использующему модель.

Каждый экземпляр автомата будет хранить свою собственную модель данных, содержащую объявления локальных переменных. При таком подходе может быть легко реализована рекурсия: каждый рекурсивный вызов будет создавать экземпляр автомата, содержащий свои собственные локальные переменные, не зависящие от значений в других экземплярах.

Для всех глобальных переменных будет создана отдельная модель данных.

Выводы по главе 1

1. Рассмотрен набор понятий, необходимых для выполнения процедуры преобразования. Понятия «фрагмент автомата» и «замыкание» позволяют разбить процесс преобразования программы на подзадачи (построение фрагментов автомата по отдельным частям кода), которые могут быть выполнены независимо друг от друга. Таким образом, процесс преобразования можно выполнять параллельно в нескольких потоках, сокращая общие затраты времени.

2. Приведен обзор методов реализации обратной трассировки. По мнению автора, возможность выполнения программы в обратную сторону является очень удобной и полезной в процессе отладки, именно поэтому в данной работе ей уделено значительное внимание.

Глава 2. Модификация кода исходной программы

Приведение программы к автоматному виду будем выполнять в два этапа. На первом из них (рассматриваемом в этом разделе) преобразуем исходный код так, чтобы исключить из него сложные и неудобные в обработке конструкции (так называемый *синтаксический сахар*) и заменить их набором базовых операторов языка. На втором этапе, описанном в главе 3, произведем собственно построение по коду системы автоматов.

2.1. Области видимости переменных

В *C*-подобных языках, когда переменная, находящаяся в статической памяти, выходит из области видимости, она уничтожается. Если переменная была объектом некоторого класса — вызывается ее деструктор. В автоматной программе нет понятия области видимости, все переменные хранятся в модели данных, поэтому автоматически удалены они будут только после окончания работы. Это может привести к отличиям в ходе выполнения от исходной программы.

Автоматическое удаление вышедших из области видимости переменных является важным аспектом при работе с внешними ресурсами по методу *RAII* (*Resource Acquisition Is Initialization*). Это иллюстрирует следующий пример (язык *C*++):

```
class ResourceHolder
{
  ResourceHolder(…) {…} // В конструкторе создается некоторый
                        // ресурс
  ~ResourceHolder() {…} // В деструкторе ресурс удаляется
…
};
…
{
ResourceHolder res(…);
```

```
… // Операции с ресурсом
} // Когда выполнение дойдет до конца блока, если произойдет
  // исключение или же работа функции будет завершена
  // оператором return, вызовется деструктор res и ресурс
  // будет корректно удален
```

Для решения проблемы отсутствия областей видимости разместим все переменные в динамической памяти и будем их сами явно удалять. В этом случае использование ресурса будет выглядеть следующим образом:

```
{
  ResourceHolder* res = new ResourceHolder(…);
  …
  delete res;
}
```

Оператор удаления всех таких объектов также следует добавить перед всеми возможными выходами из функции.

В языках с автоматическим управлением памятью объект может быть удален, только если на него не остается указателей. Когда переменная-указатель выходит из области видимости, общее число указателей на данный объект может стать равным нулю. После этого объект подлежит уничтожению. В автоматной программе, как уже отмечалось выше, переменные всегда находятся в области видимости. Для уменьшения числа ссылок на объект необходимо вручную в соответствующем месте присвоить указателю значение NULL:

До преобразования:

```
{
  A a = new A();
  … // Здесь не создается новых указателей на объект
}   // В этом месте указатель выйдет из области видимости.
    // Общее число указателей на объект станет равным нулю, и
    // при следующем запуске сборщика мусора объект будет
    // удален
```

После:

```
{
```

```
    A a = new A();
    …
    a = null; // После такого присваивания число указателей на
              // объект станет равным нулю
}
```

После построения по данному коду автомата одно из его состояний (соответствующее концу блока кода) будет в явном виде выполнять всю работу по удалению объекта. Таким образом, сохранится поведение исходной программы. После окончания выполнения блока кода либо все локальные для него объекты будут удалены (в случае ручного управления памятью), либо будут удалены все локально созданные ссылки на них (в случае автоматического управления памятью).

В том случае, если внутри блока возможно возникновения исключения, к блоку следует добавить обработчик, который удаляет все объекты и приравнивает нулю все указатели, а затем производит повторный выброс полученного исключения. Преобразование обработчиков исключений описано в разд. 3.10.

Аналогичные преобразование следует совершить и над объектами, являющимися полями другого класса. Их следует заменить на указатели и явно создавать и уничтожать в конструкторе и деструкторе соответственно.

2.2. Перегрузка операторов

Некоторые языки, такие как *C++* и *C#*, позволяют выполнять перегрузку операторов для пользовательских типов данных. Перегрузка позволяет добиться естественного использования типа данных во многих операциях и поэтому достаточно широко используется. Классическим примером использования перегрузки операторов является написанием классов для выполнения математических операций. Рассмотрим следующий класс-обертку над типом целого числа (код на языке *C++*):

```
class MyNumber
{
     int num;
public:
     …
     MyNuber& operator=(const MyNumber& number)   { … }
     MyNumber& operator+(const MyNumber& number)  { … }
};
```

С переопределенными операторами сложения и присваивания становится возможным написать следующий фрагмент:

```
MyNumber a, b, c;
…
a = b + c;
```

Перегруженные операторы применяются только для удобства чтения кода программы человеком. С точки зрения компилятора такой код будет эквивалентен следующему:

```
a.operator=(b.operator+(c));
```

Для такого кода уже известен алгоритм построения фрагмента автомата, так как применение операторов сводится к простому вызову соответствующих функций. Таким образом, следующим шагом преобразования исходного кода будет замена всех операторов на явный вызов их как функций.

2.3. Приведение к виду, использующему модель данных

Модифицируем программу таким образом, чтобы она использовала только модель данных. Сначала построим модель по следующим правилам:

1. Создадим глобальную модель данных. Далее будем считать, что она доступна как глобальная переменная с именем `globalData`.
2. Поместим в нее все объявления глобальных переменных.
3. Изменим в коде программы все обращения к глобальным

переменным на обращение через `globalData.<имя переменной>`.

4. Для каждой процедуры создадим локальную модель данных. Будем полагать, что она доступна как локальная переменная с именем `data`.
5. Поместим в локальные модели объявления всех переменных, локальных для соответствующих процедур.
6. Поместим в локальные модели объявления всех параметров соответствующих процедур.
7. Добавим в каждую локальную модель переменную с именем `<имя процедуры>_`, имеющую такой же тип, как и у возвращаемого функцией значения. В эту переменную будет записываться результат работы функции.
8. Модифицируем исходный код так, чтобы он использовал переменные через конструкцию вида `data.<имя переменной>`.

Выводы по главе 2

1. Рассмотрены преобразования, позволяющие привести код программы к виду, подходящему для построения системы автоматов.
2. Все рассмотренные преобразования фактически являются рефакторингом исходного кода, так как не влияют на результат его работы, а только изменяют внешний вид.
3. Преобразования могут быть выполнены в произвольном порядке с единственным ограничением: приведение к виду, использующему модель данных, должно быть выполнено последним.

Глава 3. Построение системы автоматов

В этом разделе будут рассмотрены алгоритмы преобразования конкретных структур исходной программы в соответствующие структуры автоматной программы. Перед применением рассматриваемых ниже алгоритмов, должны быть выполнены все преобразования, описанные в главе 2.

3.1. Вызов процедур

Сложность при преобразовании объектно-ориентированной программы к автоматному виду связана с дополнительными возможностями вызова функций-членов (конструкторов, деструкторов, обычных методов) и ограничениями на них (интерфейсы). Для выполнения преобразования сначала рассмотрим фрагмент автомата, соответствующий обычному вызову функции, а в последующих разделах покажем, как свести работу с методами класса только к явным вызовам функций.

Оператор вызова процедуры преобразуем в одно состояние, которое будет создавать, запускать и ожидать окончания работы автомата для процедуры (такой автомат будем далее называть вложенным). Для синхронизации работы вложенных автоматов будем использовать методы `isAtStart()` и `isAtEnd()`, показывающие что автомат находится в начальном и конечном состояниях соответственно. Таким образом, в состоянии должны быть выполнены следующие действия:

- конструирование вложенного автомата;
- передача параметров;
- пошаговое выполнение вложенного автомата, пока тот не придет в конечное состояние;
- получение результата;
- удаление вложенного автомата.

3.1.1. Передача параметров

Передача параметров и получение результата производится посредством дополнительных функций вида `void setParameters(/*список параметров такой же, как и у исходной функции*/)` и `/*тип возвращаемого значения, как и у исходной функции*/ getResult()`.

Проблема может возникнуть при передаче параметра по значению в том случае, когда он является экземпляром класса. Приведем простой пример:

```
class A
{
  public:
  A(int x) {…} // Конструктор от целого числа
  A(const A& a) { … } // Конструктор копирования
}
void f(A a) { … }
...
A a(5);
f(a); // Вызов от объекта класса
f(5); // Вызов от целого числа
```

При каждом вызове функции `f` будет создан локальный объект типа `A` – в первом случае с помощью конструктора копирования, а во втором – конструктором от целого числа. После завершения работы функции этот локальный объект удаляется.

Теперь попытаемся установить, какие действия необходимо выполнить для повторения в автоматной программе поведения, описанного в предыдущем абзаце. Обычный подход – присвоение значений параметров переменным локальной модели данных в функции `setParameters`. Он в данном случае не работает. Ключевым моментом является именно «присвоение переменным локальной модели», так как это потребует лишнего использования оператора присваивания, которого не было в исходной программе.

Выходом из данного затруднения является использование шаблонов в *C++* и *generic*-классов в *Java* в сочетании с выносом локальной переменной-параметра функции из статической памяти в динамическую. При таком подходе функция установки параметров для автоматной реализации функции `f` будет иметь следующий вид (язык *C++*):

```
struct DataModel
{
  A* a;
  …
};

template <class FirstParameterType>
  void setParameters(const FirstParameterType& param)
{
  dataModel = new DataModel();
  dataModel.a = new A(param);
}
```

Данная функция будет работать как в случае копирования объекта, так и в случае его создания от некоторого параметра. При таком подходе ровно один раз будет вызван соответствующий конструктор и не будет добавлено никаких «лишних» (отсутствовавших в исходной программе) вызовов методов класса. Результатом вызова конструкторов будет добавление в очередь отложенных вызовов их автоматных реализаций.

Перед завершением работы автомата для функции `f` необходимо будет произвести вызов автоматов для деструкторов всех таких объектов, а затем удалить их. Тем самым будет полностью воспроизведено поведение исходной программы.

Использовать такой подход необходимо только в случае передачи по значению объектов. Для переменных простых типов подходит обычное копирование в локальную модель данных.

3.1.2. Фрагмент автомата

В работе [12] предлагалось удалять вложенный автомат после прямого прохода по нему, а для обратной трассировки создавать его заново, выполнять его до достижения конечного состояния, а затем уже начинать пошаговое выполнение в обратную сторону. К сожалению, данный подход не работает при наличии в системе выражений с побочными эффектами (в случае, когда ход выполнения функции зависит не только от ее параметров, но и от окружения: глобальных переменных, данных, вводимых пользователем и т.п.). По этой причине не будем удалять вложенный автомат. Для каждого состояния будем хранить стек из всех вложенных автоматов, вызванных когда-либо из него. При выполнении обратной трассировки снимем самый верхний автомат со стека. Так как ранее он уже был вызван и завершил свою работу, то он уже находится в конечном состоянии и готов к выполнению в обратную сторону.

Для данного подхода характерны значительные затраты памяти на хранение всех вложенных автоматов, их моделей данных и стеков сохраненных значений переменных, но этот недостаток компенсируется корректностью обращения систем, зависящих от контекста.

Построенный рассмотренным методом фрагмент автомата показан на Рис. 1.

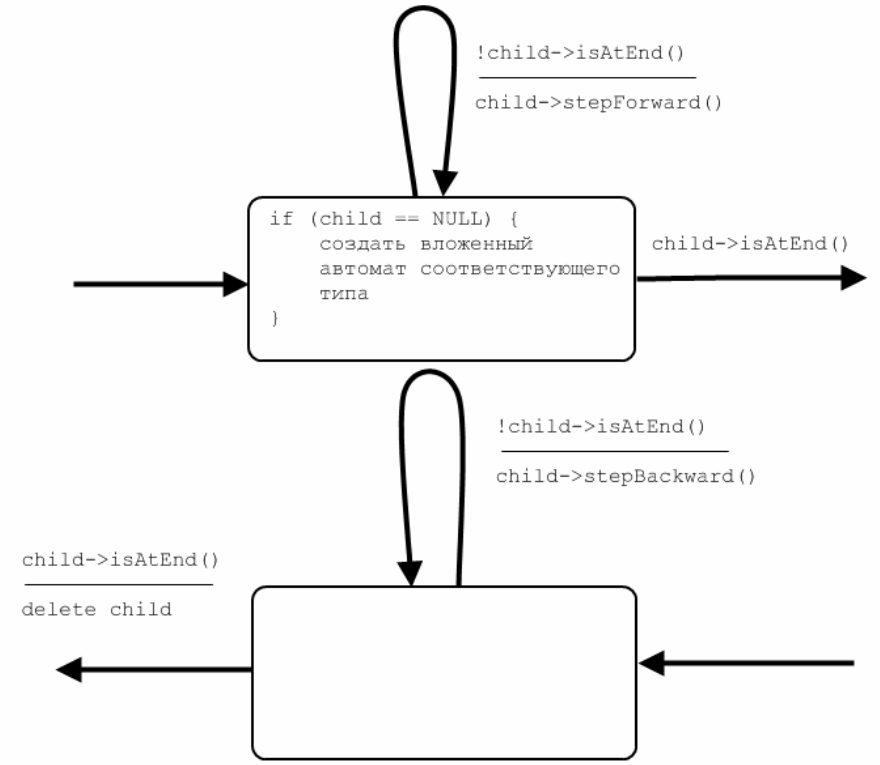


Рис. 1. Фрагмент автомата для вызова процедуры

3.2. Отложенные вызовы автоматов

В качестве одного из отличий объектно-ориентированных программ можно выделить наличие большого объема кода, неявно созданного компилятором, в частности, вызовы конструкторов и деструкторов классов. Например, при использовании *finalize*-функций в языке *Java* нет никаких сведений о том, когда именно они будут вызваны. Для таких функций нет конкретного места в коде программы, которое можно было бы обработать с помощью метода, рассмотренного в предыдущем разделе.

Рассмотрим пример (язык *C++):*

```
class String
{
public:
  String(const char* str){…}
};

void f(String s)
```

```
{
  …
}

void g()
{
  f("Hello world");
}
```

В данном примере при вызове функции `f()` произойдет неявный вызов конструктора класса `String` от переданной ему строки, имеющей тип `const char*`. Если в автоматной программе в одном из состояний просто вызвать вложенный автомат, соответствующий функции `f()`, то конструктор будет выполенен один раз и не будет возможности пройти его по шагам. Для того чтобы сохранить возможность пошагового выполнения, создадим автомат для конструктора *отложенным*.

Отложенный автомат будет добавляться в специальную очередь у текущего выполняемого автомата. Текущий автомат перед выполнением каждого шага проверяет эту очередь, и если она не пуста – запускает первый полученный из нее автомат обычным образом. В рассматриваемом примере автомат для конструктора будет добавлен в очередь отложенных автоматов для `f` в момент вызова сгенерированного конструктора при передаче параметра (разд. 3.4).

Для реализации добавления в очередь отложенных вызовов необходимо реализовать получение текущего выполняемого автомата. Так как у каждого автомата в любой момент времени может быть не более одного активного вложенного, то достаточно просто пройти по цепочке вложенных автоматов, начиная с самого первого вызванного.

При необходимости вызова автоматов, обратных отложенным (при выполнении обратной трассировки), то потребуется сохранение номера состояния, на котором произошел отложенный вызов. При этом на каждом

шаге назад текущий автомат будет проверять, не происходил ли при прямом проходе в данном состоянии вызов отложенного автомата, и если происходил – запускать соответствующий обратный автомат.

Идея отложенного вызова уже выдвигалась ранее в работе [16], однако там она использовалась для другой цели: реализации многопоточности и решения проблем, связанных с реентерабельностью (возможностью повторного вызова из параллельного потока до завершения работы) автоматных функций.

Следует отметить, что теоретически возможно приведение программ к автоматному виду и без использования отложенных вызовов. Для этого необходимо фактически реализовать компилятор языка, который найдет все неявные вызовы функций и заменит их явным вызовом автоматов. Однако такой подход, по мнению автора, слишком сложен для решения рассматриваемой задачи, особенно для языков со сложной грамматикой наподобие *C++*.

3.3. Представление классов

Для каждого класса исходной программы будем строить соответствующий ему класс автоматной программы (в дальнейшем будем называть его *производным* классом, не стоит путать это со значением «класс, наследник данного»). Построенный класс будет содержать:

- те же поля, что и исходный;
- автоматные реализации методов;
- дополнительные функции для вызова автоматных методов.

Иерархия классов будет полностью сохранена. Если два класса находились в отношении наследования в исходной программы, то их производные классы будут находиться в таком же отношении в автоматной программе. То же самое касается и интерфейсов.

3.4. Методы

Построение автомата для операторов, находящихся в тела метода, можно осуществить с использованием уже известных алгоритмов. Пусть этот автомат реализует следующий интерфейс:

```
interface Automaton
{
   void stepForward();
   void stepBackward();
   bool isAtEnd();
   bool isAtStart();
   Automaton getChild();
}
```

Передача параметров в автомат и возвращение значения будет выполняться дополнительными функциями, как описано в разд. 3.1.

Основное отличие методов класса от обычных членов — наличие скрытого параметра `this`, указывающего на конкретный объект, у которого был вызван метод. Второе отличие — наличие доступа к скрытым членам класса. Таким образом, для полной реализации автомата для метода необходимо предоставить ему указатель `this` и дать доступ к скрытой части класса.

Первое пункт выполним в явном виде: заведем указатель на экземпляр объекта в классе автомата и изменим все обращения к полям.

В результате, класс с одним методом и одним полем

```
class A
{
   int a;
   f() { a++; }
}
```

будет преобразован в:

```
class A
{
   int a;
   class f_automaton implements Automaton
```

```
    {
        A myThis;
        f_automaton(A myThis) { this.myThis = myThis; }
        …
            myThis.a++;
        …
    }
}
```

При этом вызовы вида

```
A a;
…
a.f();
```

будут преобразованы по правилам вызова функций к виду:

```
A a;
…
Automaton f = new A.f_automaton(a);
while (!f.isAtEnd()) {
    f.stepForward();
}
```

Реализация доступа к закрытым полям класса зависит от используемого языка программирования: в языке *C++* классы автоматов для методов должны быть «друзьями» (`friend`) своего класса-владельца, в языке *Java* – достаточно сделать их вложенными.

3.5. Виртуальные функции

При использовании виртуальных функций на этапе построения автоматного кода неизвестно, какой именно автомат следует вызывать. Выяснится это только на этапе выполнения программы.

Решим эту проблему построением автомата-*диспетчера*. Диспетчер будет во время выполнения проверять реальный тип объекта и запускать нужный

вложенный автомат (рис. 2). Наличие явно выделенного диспетчера позволяет избавить вызывающий автомат от необходимости выполнять проверки. Более того, вызывающий автомат может и не иметь информации о том, что именно он вызывает: автомат для обычной функции или диспетчер для виртуальной.

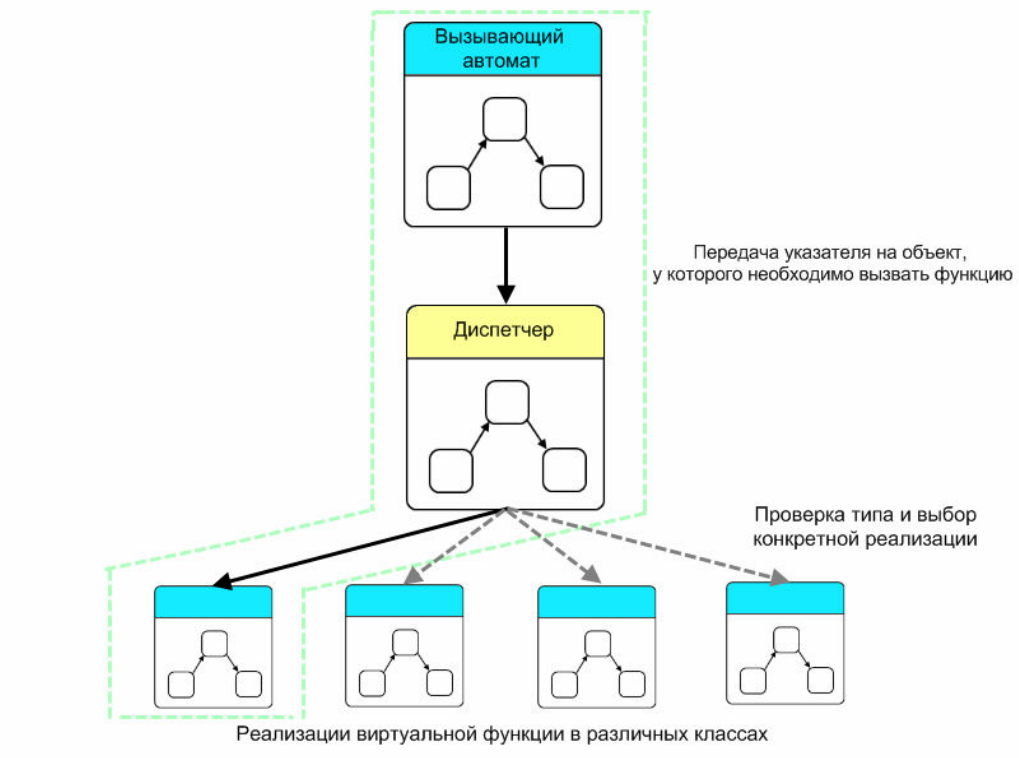


Рис. 2. Схема использования диспетчера

Автомат-диспетчер будет состоять из начального и конечного состояний, а также будет иметь по одному состоянию для каждого возможного реального типа объекта, у которого мы хотим вызвать виртуальную функцию. В каждом из этих состояний будет производиться вызов вложенного автомата, который отвечает данной функции у данного типа. Назовем эти состояния *вызывающими*. Добавим переходы из начального состояния в каждое вызывающее. Условиями на переходах будут совпадения типов, ожидаемых в соответствующих вызывающих состояниях и реального типа объекта. Завершим построение добавлением переходов из вызывающих состояний в конечное.

Приведем пример.

Пусть исходная иерархия выглядит следующим образом:

```
class A
{
     virtual void f();
};
class B: public A
{
     virtual void f();
};
```

Тогда для вызова функции `f` у переменной `a`, имеющий тип указателя на `A`, необходимо построить автомат-диспетчер, показанный на рис. 3 (пример для языка *Java*).

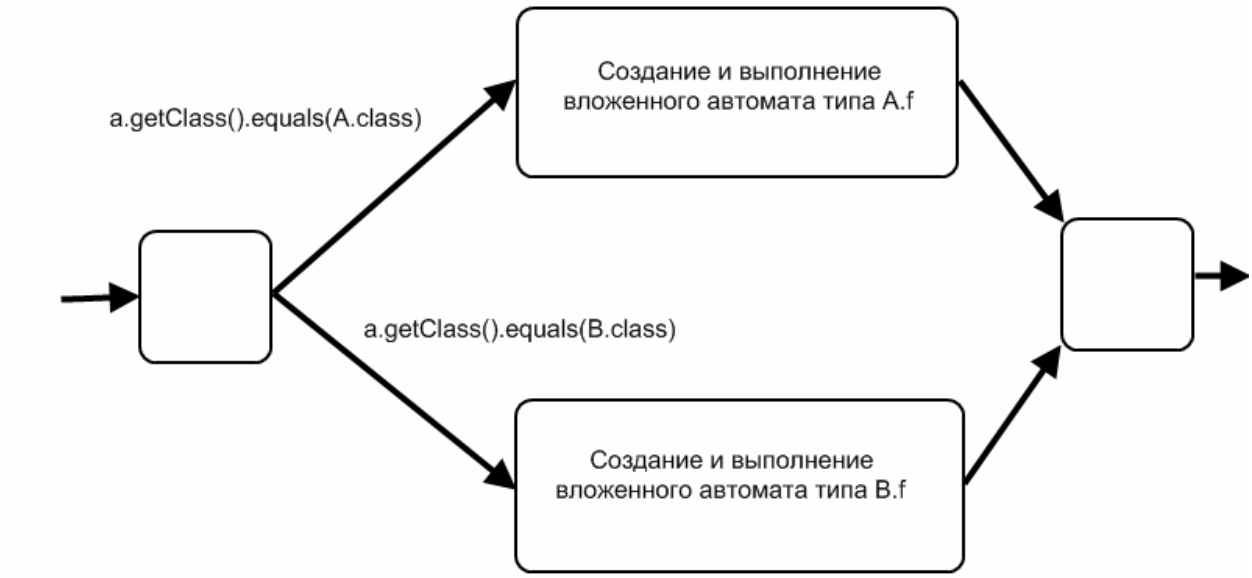


Рис. 3. Фрагмент автомата для вызова виртуальной функции

С помощью динамической проверки типа проверим, что полученная автоматная программа будет вести себя так же, как исходная. Следовательно, будет вызван автомат нужного типа – соответствующего функции, которая была бы вызвана при тех же условиях в исходной программе.

Выбор вида вызова (с использованием диспетчера или без него) можно осуществить на этапе построения автоматного кода. Тогда же можно определить конкретный класс диспетчера. Его реализацию можно поместить либо в отдельный класс, либо внутри того класса, которому принадлежит вызываемая функция.

3.6. Конструкторы

Конструкторы классов – специальные функции, вызывающиеся после выделения памяти под объект для его инициализации. По типу вызова конструкторы можно разделить на две группы:

- явно вызываемые пользователем;
- неявно вызываемые.

3.6.1. Явно вызываемые конструкторы

Наиболее простой случай. В программе присутствует фрагмент кода, явно создающий объект. Например,

```
A a = new A(); // Java
A* a = new A(); // C++
A a(); // C++
```

Такой вызов нельзя просто заменить на вызов автоматной реализации, так как сперва необходимо создать объект, с которым она будет работать. Для этой цели создадим в производном классе конструктор без параметров с пустым телом:

Исходный класс:

```
class A
{
  A() {…}
  A(int i) {…}
}
```

Производный класс:

```
class A
{
  A() { return; } // Конструктор по умолчанию,
                  // не делает ничего
  class ctor_automaton implements Automaton
  {
  …
  }
```

```
    class ctor_int_automaton implements Automaton
    {
    …
    }
}
```

С его помощью создадим объект, а после этого уже вызовем автоматную реализацию. Создание объекта строкой вида:

```
A* a = new A(5)
```

изменится на

```
A* a = new A();
Automaton a_ctor =  new A::ctor_int_Automaton(a, 5);
```

3.6.2. Неявно вызываемые конструкторы

Вызов конструкторов копирования и при неявном преобразовании типов лежит на компиляторе, поэтому в самой программе нет необходимости в коде, явно создающем объект класса. Воспользуемся для разрешением данной проблемы механизмом отложенных автоматов, рассмотренным ранее в разд. 3.2.

Для каждого конструктора исходного класса добавим в производный класс автомат, реализующий поведение данного конструктора, и конструктор с такой же сигнатурой, как и в исходном классе. Этот конструктор и будет выполнять добавление автомата в очередь отложенных вызовов.

3.7. Интерфейсы

Интерфейсы – это своеобразные контракты, заключаемые классом и гарантирующие для стороннего пользователя наличие определенной функциональности. По этой причине было бы нежелательно изменять интерфейсы в автоматной программе, так как это может привести к потере совместимости с кодом, который взаимодействует с классом через интерфейс. По этой же причине может возникнуть необходимость в том, чтобы при вызове интерфейсного метода он выполнялся не по шагам, а сразу целиком –

имитировал работу неавтоматной функции. Это может понадобиться, если класс загружается во время выполнения (в виде *DLL* в случае *C++* или через *ClassLoader* в случае *Java*) некоторого стороннего кода, который не может быть изменен. Этот сторонний код, естественно, ожидает немедленного получения результата при вызове функции.

Реализуем в производном классе все функции интерфейса следующим образом: они будут создавать экземпляр автомата и выполнять шаги пока он не придет в конечное состояние.

Исходная иерархия:

```
Interface ISaver
{
   void saveInt(int i);
   …
}
class MySaver implements ISaver
{
   void saveInt(int i)
  {
  … // Реализация
   }
  …
}
```

Производная иерархия. Интерфейс остался без изменений, производный класс `MySaver` по-прежнему его реализует:

```
Interface ISaver
{
   void saveInt(int i);
   …
}
class MySaver implements ISaver
{
   class saveInt_automaton implements Automaton
  {
```

```
    … // Автоматная реализация метода saveInt
   }
    void saveInt(int i)
    {
         saveInt_automaton a = new saveIntAutomaton(this);
         a.setParameters(i);
         while (!a.isAtEnd()) {
              a.stepForward();
         }
    }
}
```

Все вызовы интерфейсной функции из доступного для преобразования кода будем по-прежнему изменять на создание соответствующего автомата. Таким образом, автоматный код можно будет использовать вместо исходного кода везде, где требуется один из реализованных интерфейсов.

3.7.1. Прокси-функции

Прокси-функции являются обобщением рассмотренного выше метода для сохранения интерфейсов. Прокси-функция в автоматной программе может быть построена для любой функции исходной программы, для которой является желательным сохранить возможность неавтоматного вызова. Как уже было упомянуто, это может понадобиться при использовании (линковке или динамической загрузке) модифицированного кода некоторым внешним приложением.

Прокси-функция – функция, имеющая такую же сигнатуру и ведущая себя так же, как и исходная, но реализующая ее поведение через автомат. Прокси-функция создает внутри себя экземпляр соответствующего автомата, запускает его на выполнение, а по окончании его работы – возвращает полученное значение.

При использовании автоматного представления программы в отладочных целях, прокси-функция может обмениваться сообщениями с внешним

отладчиком. Взаимодействуя с отладчиком, пользователь сможет контролировать выполнение автоматного кода, вручную выполнять переключение состояний, использовать обратную трассировку при наличии обратного автомата.

3.8. Деструкторы

Ситуация с деструкторами аналогична ситуации с конструкторами: они также могут быть вызваны как явно, так и неявно. Работа с деструкторами сильно зависит от политики управления памятью, принятой в используемом языке программирования. Разобьем вызов деструкторов на три группы:

- при выходе статического объекта из области видимости;
- при ручной очистке памяти;
- при автоматической очистке памяти сборщиком мусора.

3.8.1. Деструкторы при выходе из области видимости

Наибольшую сложность представляет именно эта группа. Это связано с тем, что в автоматной программе нет понятия «область видимости». То, что в исходной программе находилось в одном блоке, в автоматной программе будет разбито на несколько состояний. Здесь неприменим отложенный вызов: так как областей видимости нет, соответственно не будет и автоматического уничтожения объектов. Именно для этой цели на этапе предварительной обработки кода создание локальных объектов в статической памяти должно быть заменено на ручное создание и удаление в динамической памяти (разд. 2.2). После такой обработки все явные вызовы операторов удаления объектов можно обработать по алгоритму, указанному в следующем параграфе.

3.8.2. Деструкторы при ручном удалении объектов

Данный случай практически не отличается от соответствующего случая для конструкторов, только действия выполняются в обратном порядке. В

производном классе реализуется деструктор, который не выполняет никаких действий, и автомат, в котором и будет выполнена вся необходимая работа. При этом первым будет осуществлен вызов автомата, а затем, после завершения его работы, объект будет удален. Рассмотрим пример.

Исходный код:

```
class A {
   ~A() { … }
}
// Создание указателя a на объект типа A
…
delete a;
```

Код после преобразования:

```
class A {
   class dtor_automaton() : public Automaton
   {
        …
   }
   ~A() { return; }
}
// Создание указателя a на объект типа A
…
// Выполнение действий, ранее бывших в деструкторе
Automaton automaton = new A::dtor_automaton(a);
… // Выполнение, по шагам или целиком, автомата для
 // деструктора
delete automaton;
//Удаление объекта
delete a;
```

3.8.3. Деструкторы при использовании автоматической сборки мусора

Такие языки как *Java* и *C#* поддерживают реализацию так называемых

finalize-функций. Они аналогичны деструкторам в языках с ручным управлением памятью за одним исключением: их вызов осуществляется сборщиком мусора во время уничтожения объекта. При этом невозможно дать никакие гарантии относительно времени и места вызова этих функций. Повторный вызов этих функций, как правило, невозможен. Кроме того, эти функции запускаются из отдельного потока.

Для разрешения этой проблемы снова воспользуемся отложенным вызовом автоматов. Для каждого исходного класса, имевшего *finalize*-метод, реализуем автомат и новый *finalize*-метод. Автомат будет выполнять всю необходимую работу (как правило, это освобождение ресурсов, занятых объектом). Новый метод будет добавлять этот автомат в список отложенных вызовов, а затем будет ожидать окончания работы автомата, не давая тем самым сборщику мусора уничтожить объект (уничтожение объекта может произойти только после окончания работы *finalize*-метода).

Покажем преобразование на примере.

Исходный класс:

```
class A
{
…
  protected void finalize()
  {
   …
  }
}
```

Класс после преобразования:

```
class  A
{
    class finalize_automaton: extends Automaton
  {
   …
  }
```

```
protected void finalize()
  {
  finalizeAutomaton = new finalize_automaton(this);
  currentAutomaton.addDeferredCall(finalizeAutomaton);
    while (!finalizeAutomaton.isAtEnd()) {
        Thread.yield();
    }
  }
}
```

3.9. Глобальные переменные

Глобальные переменные и статические члены классов автоматически инициализируются в момент запуска программы и уничтожаются перед самым ее завершением. Для реализации такого поведения неприменим метод отложенного вызова для конструкторов, так как на этапе инициализации еще не существует текущего автомата и очереди отложенных вызовов. Невозможно хранить очередь в глобальной переменной, поскольку, как правило, нет никаких гарантий насчет порядка инициализации таких переменных.

Будем хранить все глобальные переменные в отдельной модели данных. В автомат, соответствующий самой первой запускаемой функции (обычно это функция `main`), добавим два дополнительных состояния. Одно из них расположим в начале (до состояний, соответствующих коду исходной программы), но после всей необходимой инициализации структур автоматов. Это состояние будет создавать глобальную модель данных. При ее создании, если среди глобальных переменных были объекты, то в очередь отложенных вызовов добавятся вызовы всех их конструкторов. Второе состояние добавим после всех состояний, реализующих логику функции. В нем осуществим вызов автоматов для деструкторов объектов и уничтожение глобальной модели данных.

3.10. Обработка исключений

Использование исключений не относятся непосредственно к объектно-ориентированному программированию, однако является достаточно важной темой для того, чтобы быть рассмотренным.

Синтаксис обработки исключений схож в разных языках: это блок кода, помеченный специальным образом (обычно конструкцией вида `try {…}`, будем называть его *try-блок*) и последовательность обработчиков.

Введем понятие *исключительного состояния*. Исключительное состояние – такое состояние, в которое переходит автомат в случае возникновения исключения. Выделим два вида таких состояний:

- *глобальное*. В него переходит автомат в случае возникновения необработанного исключения;
- *локальное*. Соответствует *try*-блоку. В него переходит автомат в случае возникновения исключения внутри блока.

Построение автоматной реализации обработчиков проведем в несколько этапов:

1. Заведем в каждом автомате переменную `currentException`, в которой будем хранить указатель на текущее исключение. По умолчанию значение переменной примем равным `null`;
2. Для каждого *try*-блока построим фрагмент автомата, соответствующий коду блока.
3. Добавим локальное исключительное состояние.
4. В каждое состояние фрагмента автомата, соответствующего коду блока, добавим переход в локальное исключительное состояние с условием `currentException != null`. Условия на всех остальных переходах изменим с `условие` на `(currentExcption == null) && (условие);`
5. Построим фрагменты автоматов для обработчиков.
6. Добавим в исключительное состояние переходы во фрагменты для

обработчиков. Условиями на переходах будут соответствия типа исключения ожидаемого обработчиком и реального типа объекта, хранящегося в `currentException`;

7. Добавим в автомат глобальное исключительное состояние.
8. В каждое состояние, не имеющее перехода в локальное исключительное состояние, добавим переход в глобальное. Условия изменим аналогично п. 4.
9. Для каждого локального исключительного состояния: если соответствующий ему *try*-блок находился внутри другого *try*-блока, добавим переход в локальное исключительное состояние для внешнего блока. Иначе добавим переход в глобальное состояние. Условием на переходе будет невыполнение условий на всех прочих переходах.
10. В состояния, отвечающие вызову вложенного автомата, добавим проверку на нахождение вложенного автомата в глобальном исключительном состоянии. Если это условие выполнено, то требуется присвоить переменной `currentException` значение, взятое из вложенного автомата.
11. Весь код, реализующий логику переходов автомата, следует поместить в *try*-блок, перехватывающий все виды исключений и помещающий их экземпляры в переменную `currentException`.

Таким образом, фрагменту программы:

```
operator1;
try {
   operator2;
} catch (Ex1 ex1) {
   Handler1;
} catch (Ex2 ex2) {
   Handler2;
}
```

соответствует фрагмент автомата, показанный на рис. 3

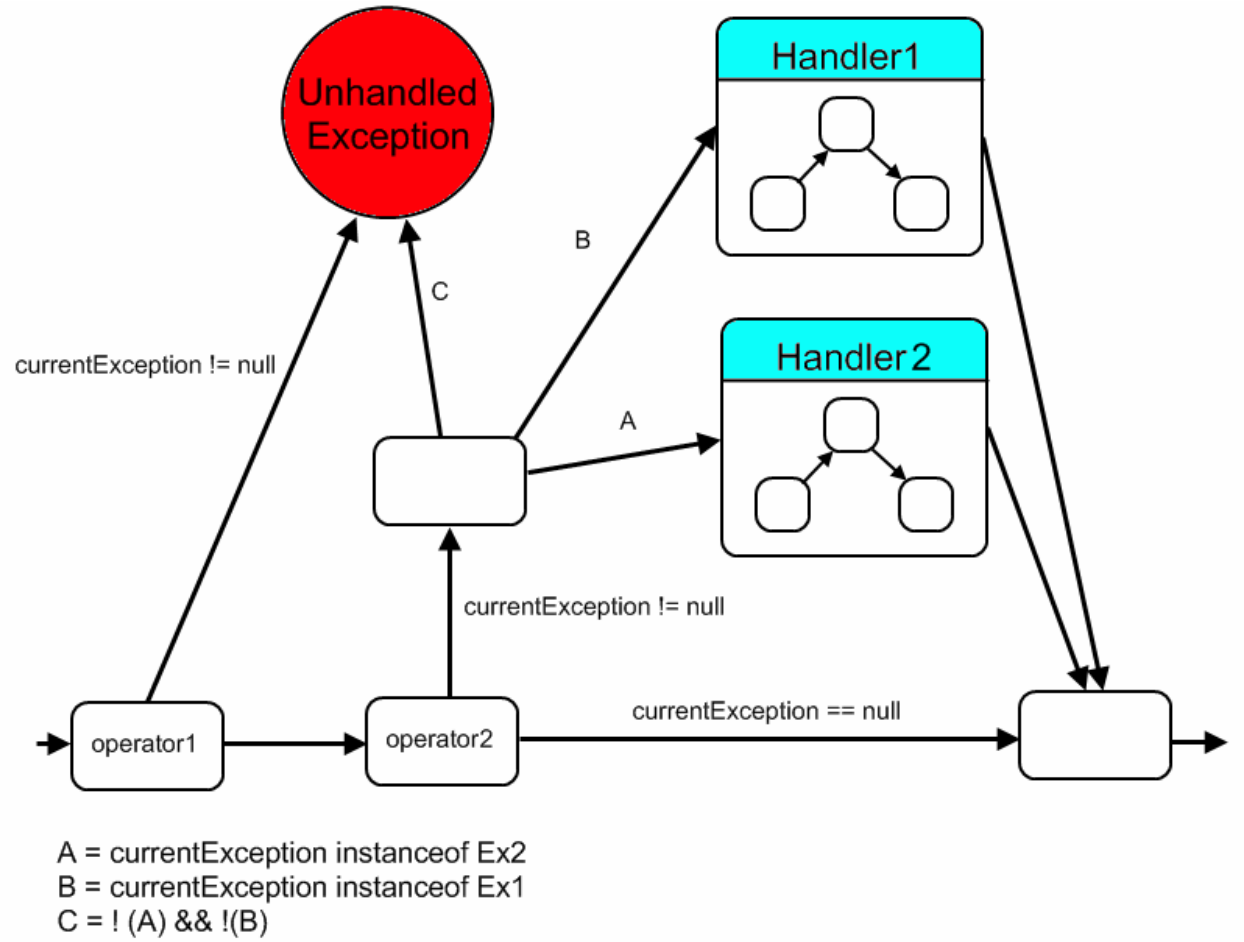


Рис. 4. Фрагмент автомата для операторов, находящихся в *try*-блоке

Для реализации обратной трассировки по данному фрагменту необходимо запоминать, какой из обработчиков был вызван. Операторы, находившиеся внутри *try*-блока, должны быть помещены каждый в свое отдельное состояние для того, чтобы после обратной трассировки по обработчику был совершен переход именно к тому месту, где возникло исключение.

Если в пределах обрабатываемой процедуры возникло необработанное исключение, то оно будет передано вверх по цепочке вызывавших друг друга автоматов, пока не достигнет места, где в исходной программе вызов процедуры находился внутри *try*-блока. При обратной трассировке, поскольку все вложенные автоматы сохраняются, после выполнения обработчика верхнего уровня далее трассировка пойдет по процедуре, выкинувшей начальное исключение.

Если исключение не было обработано в исходной программе, автоматная программа может либо завершить работу (копируя поведение исходной), либо

обработать исключение и предоставить, например, возможность воспользоваться обратной трассировкой для определения источника ошибки.

Выводы по главе 3

1. Были рассмотрены алгоритмы, позволяющие повторить в автоматной программе поведение базовых концепций объектно-ориентированного программирования.
2. Полученный в результате работы код можно назвать «автоматно-объектно-ориентированным», так как он сочетает в себе оба этих подхода. Преобразованный код остается объектным: сохраняется иерархия классов и все интерфейсы, данные и методы работы с ними по-прежнему четко структурированы. За интерфейсами же скрыты автоматные реализации, которые позволяют воспользоваться всеми преимуществами программ с явно выделенными состояниями.
3. Рассмотренных методов должно быть достаточно для преобразования произвольной программы, написанной на *C++* или *Java*.

Глава 4. Реализация

Для подтверждения работоспособности указанных в предыдущих главах методов, автором была разработана реализующая их программа-преобразователь. Программа позволяет преобразовывать объектно-ориентированный код на языке *Java* в соответствующий ему автоматный код. Преобразование разбито на несколько этапов, как показано на рис. 3.

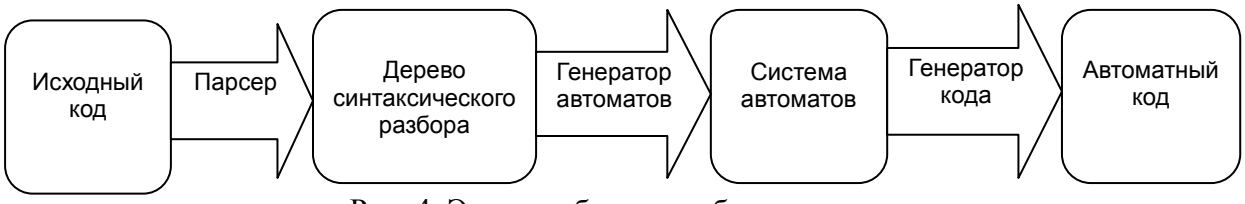


Рис. 4. Этапы работы преобразователя

Первый этап (разбор входного файла) выполняется с помощью *Java Compiler Api* – программного интерфейса компилятора *javac*. Данный интерфейс появился в *Java 1.6*, поэтому для работы преобразователя необходимо наличие этой или более поздней версии. На этом этапе происходит построение деревьев синтаксического разбора для поданных на вход файлов с исходным кодом.

На втором этапе происходит обход полученного дерева, создание по нему автоматов, добавление состояний и переходов. Так как преобразователь предназначен для демонстрационных целей, в него была добавлена поддержка ограниченного набора конструкций языка (например, из всех возможных вариантов циклов поддерживается только цикл с предусловием). Тем не менее, этих конструкций достаточно для написания различных демонстрационных примеров.

На третьем этапе происходит вывод полученной системы в виде кода на языке *Java*. Как говорилось в разд. 3, иерархия классов полностью сохраняется, поэтому полученный на выходе набор файлов будет таким же, как и исходный. Помимо сгенерированных файлов, для работы автоматной программы потребуются дополнительно служебные классы, находящихся в пакете `ru.ifmo.auto`.

Продемонстрируем работу преобразователя на примере, включающем в

себя иерархию классов, виртуальные функции и обработку исключений.

Рассмотрим набор классов, реализующих дерево вычисления значений арифметических выражений в целых числах. Для простоты ограничимся только операцией деления. Получаем иерархию классов, указанную на рис. 5.

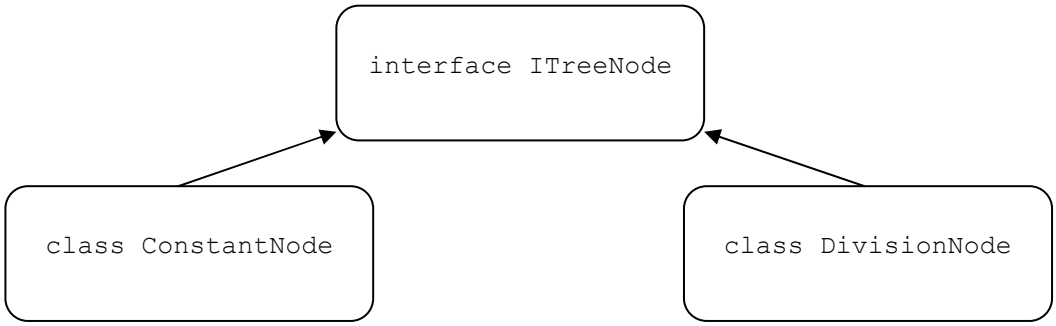


Рис. 5. Иерархия классов в примере

Интерфейс определяет единственную функцию `int compute()`, рассчитывающую значение выражения.

Класс `ConstantNode` хранит в себе константу и возвращает ее значение.

Класс `DivisionNode` хранит в себе указатели на делимое и делитель и возвращает результат деления.

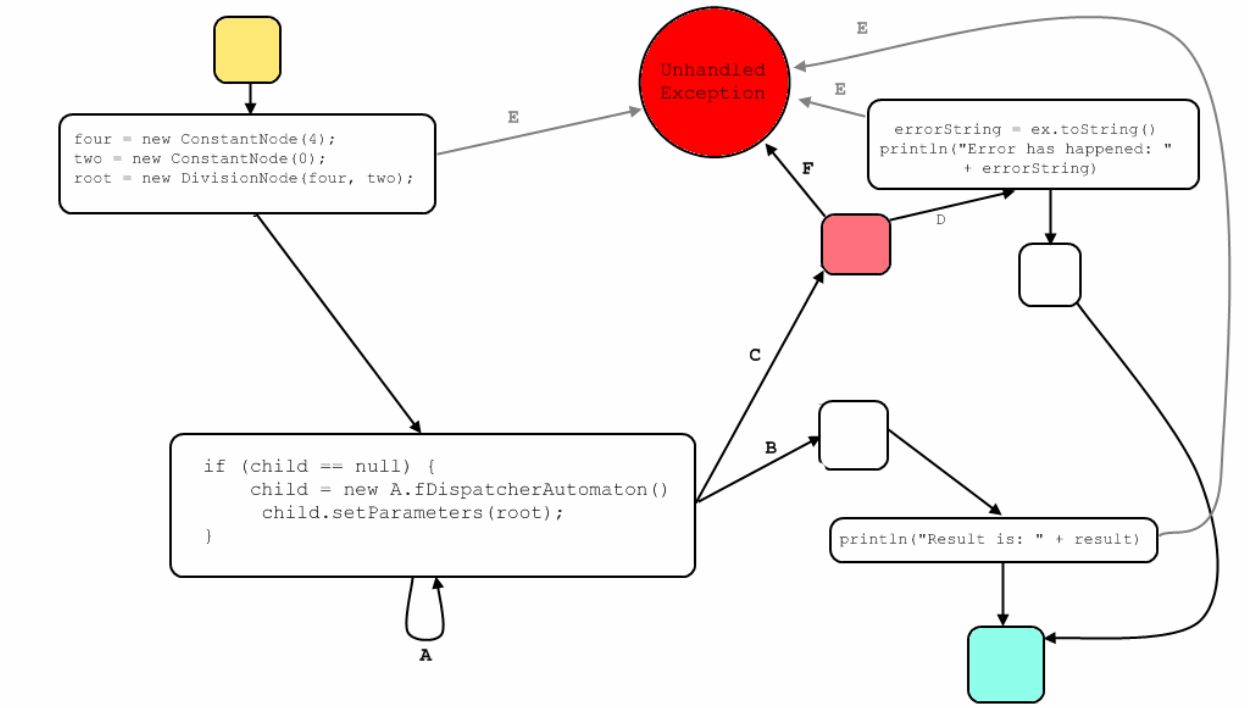
Добавим к этому набору класс `Main`, который будет строить дерево и вычислять значение. Определим в нем функцию `main` следующим образом:

```
public static void main(String[] args)
{
        ITreeNode four = new ConstantNode(4);
        ITreeNode zero = new ConstantNode(0);
        ITreeNode root = new DivideNode(four, zero);
        double result;
        try {
             result = root.compute();
        } catch (ArithmeticException ex) {
             String errorString = ex.toString();
             System.out.println("Error has happened: " +
   errorString);
             return;
```

```
        }
        System.out.println("Result is: " + result);
    }
```

Результатом выполнения данной программы будет возникновение исключения при попытке деления на ноль, его перехват и вывод сообщения пользователю.

Передадим перечисленные классы преобразователю. Полный код полученных в результате процесса классов находится в приложении. На рис. 6 приведена схема полученного автомата для функции `main`.



A = (!child.isAtEnd()) && (!child.isAtExceptionState())
B = (child.isAtEnd()) && (!child.isAtExceptionState()); result = child.getResult(), child = null
C = child.isAtExceptionState(); currentException = child.getException()
D = currentException instanceof ArithmeticException; ex = currentException, currentException = null
E = currentException != null
F = !(currentException instanceof ArithmeticException)

Рис.6. Схема автомата для функции `main`

На рис. 7 приведена схема автомата-диспетчера для функции `compute`.

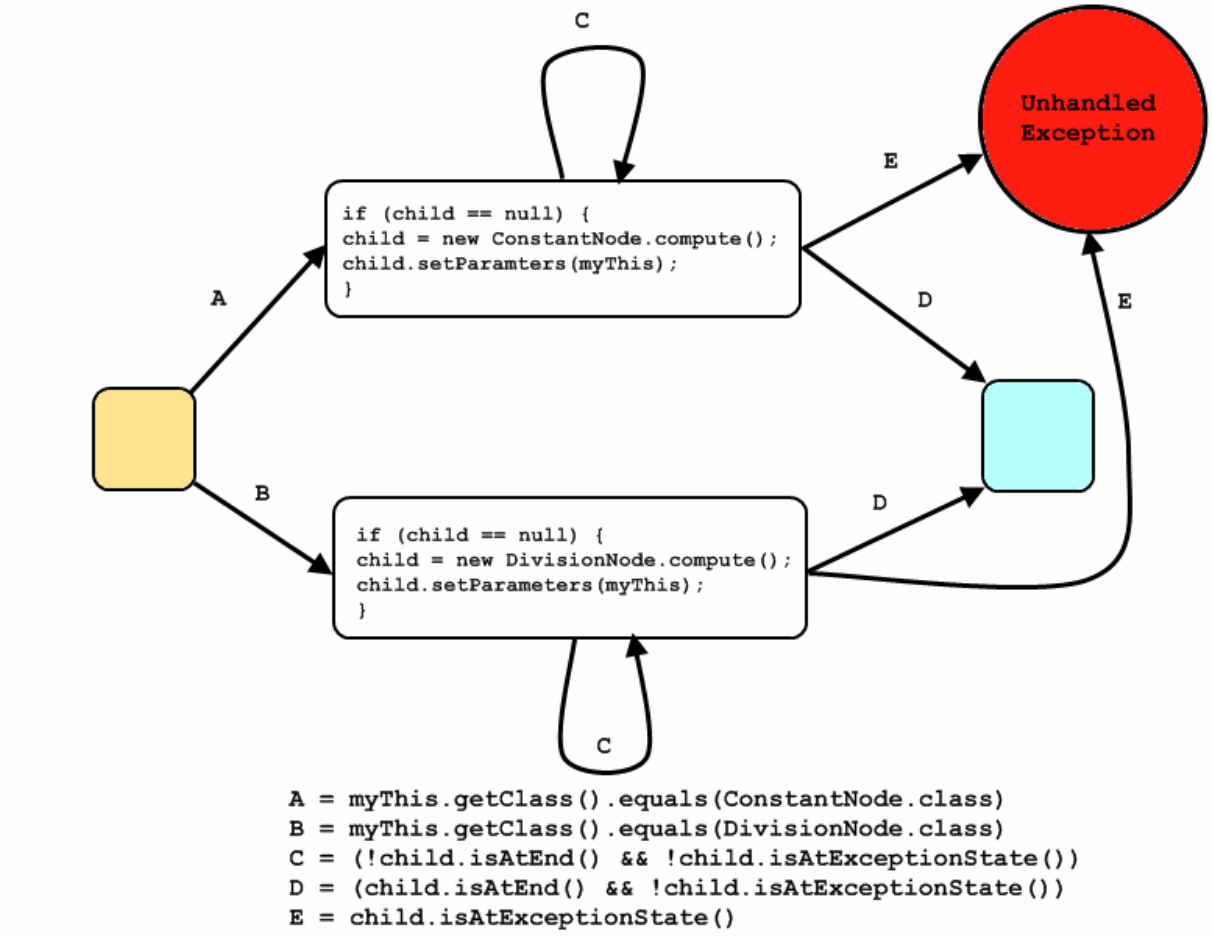


Рис. 7. Схема диспетчера для функции `compute`

В производный класс `Main` была добавлена новая функция `main`, выполняющая создание и запуск автомата, соответствующего старой функции `main`. В данной реализации система автоматов выполняется сразу до достижения конечного состояния. Возможна реализация других подходов, например передача пользователю контроля над выполнением переходов.

Заключение

Рассмотренные в работе методы позволяют решить задачу приведения к автоматному виду объектно-ориентированной программы. В сочетании с методами, изложенными в работе[12], это позволяет преобразовать в автоматную форму практически любую программу, написанную на любом из современных распространенных языков программирования, таких как *C, C++, C#, Java* (пример такого преобразования в главе 4 и приложении). Таким образом, исчезает одно из основных препятствий, затрудняющих внедрение и применение автоматного подхода – необходимость переписывать с нуля существующие системы.

Используя перечисленные выше алгоритмы, становится возможным построить автоматический преобразователь (либо расширить возможности предложенного в главе 4 преобразователя), который будет получать на вход программу, написанную в «традиционном» стиле, а на выход выдавать соответствующую ей автоматную программу. Полученную в результате систему автоматов можно использовать либо вместо исходной (так как ее поведение будет точно таким же), либо в качестве ее модели для применения разнообразных алгоритмов визуализации, анализа и верификации.

Источники

1. *Шалыто А. А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.
2. *Шалыто А. А., Наумов Л. А.* Методы объектно-ориентированной реализации реактивных агентов на основе конечных автоматов // Искусственный интеллект. 2004. № 4, с.756–762.
3. *Зюбин В. Е.* Программирование информационно-управляющих систем на основе конечных автоматов. Новосибирск, 2006.
4. *Ицыксон В. М., Глухих М. И., Зозуля А. В. и др.* Исследование средств построения моделей исходного кода программ на языках *C* и *C++* // Научно-технические ведомости СПбГПУ. 2009. № 1.
5. *Appel A. W.* Modern Compiler Implementation in *Java*. Cambridge University Press, 1998.
6. *Поликарпова Н. И., Шалыто А. А.* Автоматное программирование. СПб.: Питер, 2009.
7. *UniMod*, инструмент для построения схем автоматов в формате *UML* и генерации по ним исходного кода. http://unimod.sourceforge.net
8. *Вельдер С. Э., Шалыто А. А.* О верификации простых автоматных программ на основе метода Model Checking // Программные и аппаратные средства. 2007. № 3, с.27–38.
9. *Корнеев Г. А., Парфенов В. Г., Шалыто А. А.* Верификация автоматных программ // Тезисы докладов Международной научной конференции, посвященной памяти профессора А. М. Богомолова. 2007, с. 66–69.
10. *Туккель Н. И., Шалыто А. А.* Преобразование итеративных алгоритмов в автоматные // Программирование. 2002. № 5, с.12–26.
11. *Туккель Н. И., Шалыто А. А., Шамгунов Н. Н.* Реализация рекурсивных алгоритмов на основе автоматного подхода // Телекоммуникации и информатизация образования. 2002. № 5, с.72–99.

12. *Корнеев Г. А.* Автоматизация построения визуализаторов алгоритмов дискретной математики на основе автоматного подхода. Диссертация на соискание ученой степени кандидата технических наук.. http://is.ifmo.ru/disser/korn_disser.pdf
13. *Agrawal H., Spafford E.* An execution backtracking approach to program debugging //Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference. 1988, pp. 283–299.
14. *Сайт компании Undo Software* – разработчика коммерческого обратного трассировщика и отладчика *UndoDB*. http://www.undo-software.com/
15. *Agrawal H., DeMillo R., Spafford E.* Efficient debugging with slicing and Backtracking. Technical Report SERC-TR-80-P. Software Engineering Research Center. Purdue University, 2004.
16. *Канжелев С. Ю., Шалыто А. А.* Моделирование кнопочного телефона с использованием SWITCH-технологии. http://is.ifmo.ru/download/phone.pdf

Приложение. Код преобразованной программы

Код до преобразования

Интерфейс:

```
interface ITreeNode
{
    int compute();
}
```

Классы, реализующие его:

```
class ConstantNode implements ITreeNode
{
   private int _value;
   ConstantNode(int value)
   {
        this._value = value;
   }
   public int compute()
    {
        return _value;
   }
}

class DivisionNode implements ITreeNode
{
   private ITreeNode _left;
   private ITreeNode _right;

   public DivisionNode(ITreeNode left, ITreeNode right)
   {
        this._left = left;
        this._right = right;
   }

   public int compute()
   {
        int leftRz = _left.compute();
        int rightRz = _right.compute();

        return leftRz / rightRz;
   }
}
```

Код класса `Main` уже был приведен в главе 4.

Код, сгенерированный преобразователем

```
  class Main
  {

      static AutomatonRuntime getmainAutomaton()
      {
          return new mainSelectorAutomaton();
      }


      // This secton is for proxy functions
      interface mainAutomaton extends AutomatonRuntime<Void>
      {
          ;
      }
      static class mainSelectorAutomaton extends AutomatonRuntimeImpl< Void>
implements mainAutomaton{
          class mainDataModel
          {
              ArithmeticException ex;
              double result;
              ITreeNode two;
              ITreeNode root;
              String[] args;
              ITreeNode four;
              String errorString;
          }
          private mainDataModel dataModel = new mainDataModel();

          public void setParameters(String[] args)
          {
              this.dataModel.args = args;
          }

          public void stepForward() {
              System.out.println("Main.mainSelector, state #" + _state);
              switch(_state) {
                  case 0:
                      if (_child == null) {
                          _child = new main();
                          ((main)_child).setParameters( dataModel.args);
                      }
                      if (!_child.isAtEnd() && !_child.isAtExceptionState())
_child.stepForward();
                      if (_child.isAtExceptionState()) {_state = -1;
_currentException = _child.getCurrentException();}
                      if (_child.isAtEnd() && ! _child.isAtExceptionState())
_state = 1 ;
              }
          }
          public mainSelectorAutomaton() {
              super( 1 );
          }
          public Void getResult() {
              //Yes i know it looks fun
```

```
            return null;
        }
    }
    public static void main(String[] args)
    {
        Main.mainSelectorAutomaton a =
(Main.mainSelectorAutomaton)Main.getmainAutomaton();
        a.setParameters(args);
        while (!a.isAtEnd() && ! a.isAtExceptionState()) a.stepForward();
        if (a.isAtExceptionState()) {
            System.out.println("Unhandled exception: " +
a.getCurrentException().toString());
        }
    }
    static class main extends AutomatonRuntimeImpl<Void > implements
mainAutomaton
    {
        main()
        {
            super(11);
        }
        class mainDataModel
        {
            ArithmeticException ex;
            double result;
            ITreeNode two;
            ITreeNode root;
            String[] args;
            ITreeNode four;
            String errorString;
        }
        private mainDataModel dataModel = new mainDataModel();

        public void setParameters(String[] args)
        {
            this.dataModel.args = args;
        }

        public Void getResult()
        {
            return null;
        }
        public void stepForward()
        {
            System.out.println("Current automaton Main.main, state # " +
_state);
            if (checkAndExecuteDelayedCall()) return;
            try {
                switch (_state) {
                    case 0:
                        if (_currentException != null) {
                            _state = -1;
                            break;
                        }
                        _state = 1;
```

```
                            break;
                        case 1:
                            if (_currentException != null) {
                                _state = -1;
                                break;
                            }
                            dataModel.four = new ConstantNode(4);
                            dataModel.two = new ConstantNode(0);
                            dataModel.root = new DivisionNode(dataModel.four,
dataModel.two);
                            _state = 2;
                            break;
                        case 2:
                            if (_currentException != null) {
                                _state = 4;
                                break;
                            }
                            if (_child == null) {
                                _child = ConstantNode.getcomputeAutomaton();

((ConstantNode.computeSelectorAutomaton)_child).setParameters(dataModel.root)
;
                            }
                            if (_child.isAtExceptionState()) {
_currentException = _child.getCurrentException(); break; }
                            ;
                            if (!_child.isAtEnd() &&
!_child.isAtExceptionState()) {
                                _child.stepForward();
                                _state = 2;
                                break;
                            }
                            if (_child.isAtEnd() &&
!_child.isAtExceptionState()) {
                                dataModel.result =
((ConstantNode.computeSelectorAutomaton)_child).getResult(); _child = null;
                                _state = 3;
                                break;
                            }
                            break;
                        case 3:
                            if (_currentException != null) {
                                _state = 4;
                                break;
                            }
                            _state = 9;
                            break;
                        case 4:
                            if (_currentException instanceof
ArithmeticException) {
                                dataModel.ex =
(ArithmeticException)_currentException; _currentException = null;
                                _state = 5;
                                break;
                            }
```

```
                            _state = -1;
                            break;
                        case 5:
                            if (_currentException != null) {
                                _state = -1;
                                break;
                            }
                            dataModel.errorString = dataModel.ex.toString();
                            _state = 6;
                            break;
                        case 6:
                            if (_currentException != null) {
                                _state = -1;
                                break;
                            }
                            System.out.println("Error has happened: " +
dataModel.errorString);
                            _state = 7;
                            break;
                        case 7:
                            if (_currentException != null) {
                                _state = -1;
                                break;
                            }
                            _state = 11;
                            break;
                        case 8:
                            if (_currentException != null) {
                                _state = -1;
                                break;
                            }
                            _state = 9;
                            break;
                        case 9:
                            if (_currentException != null) {
                                _state = -1;
                                break;
                            }
                            _state = 10;
                            break;
                        case 10:
                            if (_currentException != null) {
                                _state = -1;
                                break;
                            }
                            System.out.println("Result is: " +
dataModel.result);
                            _state = 11;
                            break;
                        case 11:
                            break;
                    }
                } catch (Throwable t) {
                    _currentException = t;
                }
```

```
            }
        }
    }
    class DivisionNode implements ITreeNode
    {
        private ITreeNode _left;private ITreeNode _right;
        static AutomatonRuntime getDivisionNodeCtorAutomaton()
        {
            return new DivisionNodeCtorSelectorAutomaton();
        }


        static AutomatonRuntime getcomputeAutomaton()
        {
            return new computeSelectorAutomaton();
        }


        // This secton is for proxy functions
        public Integer compute()     {
            compute auto = new compute();
            auto.setParameters( this );
            while (!auto.isAtEnd() && !auto.isAtExceptionState())
auto.stepForward();
            if (auto.isAtExceptionState()) throw
(RuntimeException)auto.getCurrentException();
            return auto.getResult();
        }
        interface DivisionNodeCtorAutomaton extends AutomatonRuntime<Void>
        {
            ;
        }
        static class DivisionNodeCtorSelectorAutomaton extends
AutomatonRuntimeImpl< Void> implements DivisionNodeCtorAutomaton{
            class DivisionNodeCtorDataModel
            {
                DivisionNode myThis;
                ITreeNode left;
                ITreeNode right;
            }
            private DivisionNodeCtorDataModel dataModel = new
DivisionNodeCtorDataModel();

            public void setParameters(DivisionNode myThis, ITreeNode left,
ITreeNode right)
            {
                this.dataModel.myThis = myThis;
                this.dataModel.left = left;
                this.dataModel.right = right;
            }

            public void stepForward() {
                System.out.println("DivisionNode.DivisionNodeCtorSelector,
state #" + _state);
                switch(_state) {
```

```
                case 0:
            }
        }
        public DivisionNodeCtorSelectorAutomaton() {
            super(1);
        }
        public Void getResult() {
            //Yes i know it looks fun
            return null;
        }
    }
    interface computeAutomaton extends AutomatonRuntime<Integer>
    {
        ;
    }
    static class computeSelectorAutomaton extends AutomatonRuntimeImpl<
Integer> implements computeAutomaton{
        class computeDataModel
        {
            ITreeNode myThis;
            Integer leftRz;
            Integer rightRz;
            Integer _compute;
        }
        private computeDataModel dataModel = new computeDataModel();

        public void setParameters(ITreeNode myThis)
        {
            this.dataModel.myThis = myThis;
        }

        public void stepForward() {
            System.out.println("DivisionNode.computeSelector, state #" +
_state);
            switch(_state) {
                case 0:
                    if
(dataModel.myThis.getClass().equals(ConstantNode.class)) {
                        _state = 1;
                        break;                  }
                    if
(dataModel.myThis.getClass().equals(DivisionNode.class)) {
                        _state = 2;
                        break;                  }
                case 1:
                    if (_child == null) {
                        _child = new ConstantNode.compute();

((ConstantNode.compute)_child).setParameters((ConstantNode)dataModel.myThis
);}
                    if (!_child.isAtEnd() && !_child.isAtExceptionState())
_child.stepForward();
                    if (_child.isAtExceptionState()) {_state = -1;
_currentException = _child.getCurrentException();}
                    if (_child.isAtEnd() && ! _child.isAtExceptionState())
```

```
_state = 3;
                        break;
                    case 2:
                        if (_child == null) {
                            _child = new DivisionNode.compute();

((DivisionNode.compute)_child).setParameters((DivisionNode)dataModel.myThis
);}
                        if (!_child.isAtEnd() && !_child.isAtExceptionState())
_child.stepForward();
                        if (_child.isAtExceptionState()) {_state = -1;
_currentException = _child.getCurrentException();}
                        if (_child.isAtEnd() && ! _child.isAtExceptionState())
_state = 3;
                        break;
                }
            }
            public computeSelectorAutomaton() {
                super(3);
            }
            public Integer getResult() {
                //Yes i know it looks fun
                return ((AutomatonRuntime<Integer>)_child).getResult();
            }
        }
        DivisionNode(ITreeNode left, ITreeNode right)
        {
            DivisionNodeCtor auto = new DivisionNodeCtor();
            auto.setParameters( this , left, right);
            AutomatonRuntimeImpl.addDelayedCallToCurrentAutomaton(auto);
        }
        static class DivisionNodeCtor extends AutomatonRuntimeImpl<Void >
implements DivisionNodeCtorAutomaton
        {
            DivisionNodeCtor()
            {
                super(2);
            }
            class DivisionNodeCtorDataModel
            {
                DivisionNode myThis;
                ITreeNode left;
                ITreeNode right;
            }
            private DivisionNodeCtorDataModel dataModel = new
DivisionNodeCtorDataModel();

            public void setParameters(DivisionNode myThis, ITreeNode left,
ITreeNode right)
            {
                this.dataModel.myThis = myThis;
                this.dataModel.left = left;
                this.dataModel.right = right;
            }
```

```
            public Void getResult()
            {
                return null;
            }
            public void stepForward()
            {
                System.out.println("Current automaton
DivisionNode.DivisionNodeCtor, state # " + _state);
                if (checkAndExecuteDelayedCall()) return;
                try {
                    switch (_state) {
                        case 0:
                            if (_currentException != null) {
                                _state = -1;
                                break;
                            }
                            _state = 1;
                            break;
                        case 1:
                            if (_currentException != null) {
                                _state = -1;
                                break;
                            }
                            dataModel.myThis._left = dataModel.left;
                            dataModel.myThis._right = dataModel.right;
                            _state = 2;
                            break;
                        case 2:
                            break;
                    }
                } catch (Throwable t) {
                    _currentException = t;
                }
            }
        }
        static class compute extends AutomatonRuntimeImpl<Integer > implements
computeAutomaton
        {
            compute()
            {
                super(4);
            }
            class computeDataModel
            {
                DivisionNode myThis;
                Integer leftRz;
                Integer rightRz;
                Integer _compute;
            }
            private computeDataModel dataModel = new computeDataModel();

            public void setParameters(DivisionNode myThis)
            {
                this.dataModel.myThis = myThis;
            }
```

```
        public Integer getResult()
        {
            return dataModel._compute;
        }
        public void stepForward()
        {
            System.out.println("Current automaton DivisionNode.compute,
state # " + _state);
            if (checkAndExecuteDelayedCall()) return;
            try {
                switch (_state) {
                    case 0:
                        if (_currentException != null) {
                            _state = -1;
                            break;
                        }
                        _state = 1;
                        break;
                    case 1:
                        if (_currentException != null) {
                            _state = -1;
                            break;
                        }
                        if (_child == null) {
                            _child = ConstantNode.getcomputeAutomaton();

((ConstantNode.computeSelectorAutomaton)_child).setParameters(dataModel.myThi
s._left);
                        }
                        if (_child.isAtExceptionState()) {
_currentException = _child.getCurrentException(); break; }
                        ;
                        if (!_child.isAtEnd() &&
!_child.isAtExceptionState()) {
                            _child.stepForward();
                            _state = 1;
                            break;
                        }
                        if (_child.isAtEnd() &&
!_child.isAtExceptionState()) {
                            dataModel.leftRz =
((ConstantNode.computeSelectorAutomaton)_child).getResult(); _child = null;
                            _state = 2;
                            break;
                        }
                        break;
                    case 2:
                        if (_currentException != null) {
                            _state = -1;
                            break;
                        }
                        if (_child == null) {
                            _child = ConstantNode.getcomputeAutomaton();
```

```
((ConstantNode.computeSelectorAutomaton)_child).setParameters(dataModel.myThi
s._right);
                            }
                            if (_child.isAtExceptionState()) {
_currentException = _child.getCurrentException(); break; }
                            ;
                            if (!_child.isAtEnd() &&
!_child.isAtExceptionState()) {
                                _child.stepForward();
                                _state = 2;
                                break;
                            }
                            if (_child.isAtEnd() &&
!_child.isAtExceptionState()) {
                                dataModel.rightRz =
((ConstantNode.computeSelectorAutomaton)_child).getResult(); _child = null;
                                _state = 3;
                                break;
                            }
                            break;
                        case 3:
                            if (_currentException != null) {
                                _state = -1;
                                break;
                            }
                            dataModel._compute =
dataModel.leftRz/dataModel.rightRz;
                            _state = 4;
                            break;
                        case 4:
                            break;
                    }
                } catch (Throwable t) {
                    _currentException = t;
                }
            }
        }
    }


    .
```