Опубликована в журнале «RSDN Magazine». 2005. № 5.

# Реализация систем, управляемых событиями Использование конечных автоматов

**Авторы: А.Рахимбердыев**
**The RSDN Group**
**А.Ксенофонтов**
**The RSDN Group**
**Е.Адаменков**
**The RSDN Group**
**Д.Антонов**
**The RSDN Group**
**Р.Степанов**
**The RSDN Group**

## Введение

Многим разработчикам приходилось хотя бы однажды сталкиваться с необходимостью создания систем, поведение которых определяется внешними событиями (так называемых *реактивных систем*). Типичным примером такой системы является реализация стека протоколов TCP/IP. Подобные задачи возникают при организации взаимодействия с внешними устройствами, использовании удаленных компонентов и сервисов, создании приложений с распределенной архитектурой.

Опыт авторов показывает, что решения, получающиеся при традиционном подходе к реализации реактивных систем, редко оказываются удобными и простыми для дальнейшего расширения и поддержки, особенно в случаях, когда поведение системы нетривиально. В данной статье рассматривается методика разработки, эффективно решающая большую часть подобных проблем. Поведение системы в целом описывается в виде конечного автомата на диаграмме состояний UML. Локальные действия в отдельных состояниях системы определяются при помощи соответствующих классов и функций C++. В статье также описывается расширение средства моделирования ArgoUML, предназначенное для автоматизации процесса разработки.

## Практическое использование конечных автоматов

*Threads are for people who can't*
*program state machines*
*Alan Cox*

Конечные автоматы уже давно используются для формального описания поведения систем. Являясь распространенной и общепринятой абстракцией, они с успехом применяются при написании документации и спецификаций. Целью авторов, однако, было использовать конечные автоматы не только для описания, но и для реализации систем.

Прежде всего, нами была разработана универсальная реализация машины состояний, позволяющая создать и выполнить произвольный конечный автомат. Чтобы воспроизвести конкретный конечный автомат, необходимо параметризовать машину состояний экземплярами классов, разработанных для данного автомата. Из соображений производительности и удобства интеграции с остальными компонентами проекта для реализации машины состояний был выбран язык C++.

В качестве средства описания конечных автоматов было выбрано средство моделирования ArgoUML. В начале проекта синхронизация представления машины на диаграмме и ее программной реализации производилась вручную, что служило источником многочисленных трудно обнаруживаемых ошибок. Для решения этой проблемы нами было создано расширение ArgoUML, автоматически генерирующее код C++, создающий машину состояний по ее спецификации на диаграмме UML.

Далее в этом разделе мы подробно описываем каждый из упомянутых компонентов.

## Традиционный подход

В простейшем случае реализация системы, реагирующей на внешние события, выглядит подобным образом:

```
if ( /* Event A */ )
{
  // обработка события A
  ...
}
else if ( /* Event B */ )
{
  // обработка события B
  ...
}
else ...
```

Для улучшения читаемости кода можно анализировать поступающие события в отдельной функции, которая возвращает тип очередного события как элемент перечисления:

```
enum Event
{
  EventA,
  EventB,
  ...
};

Event next_event();

switch (next_event())
{
  case EventA:
    // обработка события A
    ...
  case EventB:
    // обработка события B
    ...
```

В действительности реакция системы на определенное событие часто зависит от того, что происходило до его возникновения. С обработчиками таких событий приходится связывать переменные, сохраняющие текущее состояние обработчика.

```
switch (next_event())
{
  case EventA:
    switch (substate_a) {
  case StateA_1:
    // обработка события
    substate_a = StateA_2;
    break;
  case StateA_2:
    ...
```

Подобный код достаточно эффективен, но, к сожалению, при большом количестве событий и состояний он становится трудным для понимания. Чтобы представить себе логику работы системы в целом, необходимо проанализировать весь исходный код, объем которого часто достигает нескольких сотен килобайт.

Использование объектно-ориентированного дизайна и, в частности, паттерна проектирования State, позволяет значительно улучшить легкость и удобство модификаций поведения системы. В качестве типичного примера использования паттерна State можно привести следующий фрагмент кода:

```
class State
{
  virtual State handle_a_1() = 0;
  ...
  virtual State handle_a_m() = 0;
}

class Automaton
{
  State current_state;

  Automaton(State initial_state)
  {
    currentState = initial_state;
  }

  void handle_a_1()
  {
    currentState = currentState.handle_a_1();
  }
  ...
  void handle_a_m()
  {
    current_state = currentState.handle_a_m();
  }
};
```

К сожалению, несмотря на очевидные преимущества использования паттерна State, логика работы автомата по-прежнему распределена по исходному коду. Разработчик, желающий понять поведение всей системы, должен внимательно изучить реализацию всех составляющих ее классов.

## Конечные автоматы и UML

*Конечный автомат* представляет собой хорошо известную математическую абстракцию. Он состоит из множества *состояний* (state), соединенных между собой *переходами* (transition). В любой конкретный момент времени автомат находится в одном или нескольких из своих состояний, называемых *активными состояниями*. При наступлении внешнего *события* (event), иногда называемого *триггером* (trigger) или выполнении некоторого *сторожевого условия* (guard condition) автомат может перейти из текущего активного состояния в новое по одному из переходов.

С переходами могут быть связаны действия. Такие действия выполняются при активации соответствующих переходов. Заметим также, что переходы могут быть безусловными, то есть разрешенными вне зависимости от внешних событий или условий.

Каким образом конечный автомат реагирует на внешние события? Для каждого активного состояния определяется множество разрешенных исходящих переходов для поступившего события (при этом считается ошибкой, если для одного состояния оказываются активными более одного перехода). Множество активных состояний автомата (напомним, что их может быть более одного) обновляется согласно разрешенным переходам. После этого изменение конфигурации автомата не завершается – множество активных состояний обновляется до тех пор, пока существует хотя бы один разрешенный переход (разумеется, при этом переходы могут происходить только по сторожевым условиям, поскольку поступившее событие считается уже обработанным).

Обработка события не может быть прервана. Если во время обработки события поступает новое событие, оно помещается в очередь и будет обработано позднее. Не является ошибкой, если поступившее событие не вызвало изменения конфигурации автомата – оно просто игнорируется.

Состояния могут быть простыми или *композитными* (composite), содержащими в себе вложенные состояния. Если некоторое состояние активно, то и все включающие его сложные состояния также активны.

С каждым состоянием могут быть ассоциированы *действия* (actions), выполняемые при входе, выходе и во время нахождения автомата в данном состоянии. Они называются соответственно entry action, exit action и do activity. Последнее действие интересно тем, что его выполнение может происходить в течение всего времени нахождения автомата в состоянии и быть прерванным при поступлении внешнего события.

С точки зрения внешнего наблюдателя, поведение конечного автомата полностью определяется последовательностью состояний, которые он принимает за время своей жизни. Если в данный момент времени автомат находится в некотором состоянии, то путь, по которому активное состояние было достигнуто, не должен влиять на его дальнейшее поведение. На практике все же оказывается удобным сохранять дополнительную информацию, связанную с историей переходов, в глобальном контексте и учитывать ее при выборе следующего активного состояния.

Главным достоинством конечных автоматов является то, что в них естественным образом описываются системы, управляемые внешними событиями – такие, как реализация коммуникационного протокола. При этом событиями представляется не только получение пакетов и транзакций, являющихся частью протокола (например, пакет ACK в протоколе TCP), но и получение запросов пользователя, пришедших через API (например, запрос на установление соединения с заданным хостом).

Для описания машин состояний нами была выбрана нотация UML. Язык UML на сегодняшний день является фактическим стандартом для моделирования процессов и систем при разработке программного обеспечения. В настоящее время язык активно развивается; последняя версия спецификации UML, созданная в рамках концерна OMG, имеет номер 2.0. Как показал наш опыт, набор языковых конструкций, определенный для версии 1.3 (которую поддерживает ArgoUML), оказывается достаточным для описания внешнего поведения системы и особенностей реализации ее поведенческой модели. Благодаря этому нам удалось использовать только стандартные элементы UML без определения специальных расширений языка.

В нотации UML конечные автоматы описываются *диаграммами состояний* (statechart diagram). Диаграмма состояний представляет собой ориентированный граф, вершины которого (состояния и так называемые псевдосостояния) соединены ребрами-переходами. События и условия, связанные с переходом, на диаграмме отображаются в виде метки на соответствующем ребре.

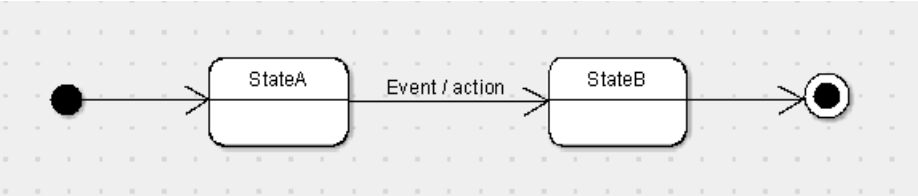


*Диаграмма 1.*

Диаграмма 1 изображает два состояния StateA и StateB, связанных переходом, активируемым по событию Event. При переходе также вызывается действие action. На диаграмме присутствуют начальное и конечное состояния, являющиеся разновидностями так называемых *псевдосостояний* (pseudo-state).

Часто в конечном автомате удобно использовать семантику *конкурентных* (concurrent) состояний. Предположим, что мы моделируем выполнение инструкций в суперскалярном процессоре. На определенном этапе декодирования инструкции необходимо запросить ресурсы для ее выполнения – функциональный блок процессора и физические регистры. Эти ресурсы запрашиваются параллельно и независимо друг от друга, но продолжение выполнения инструкции возможно только после получения всех ресурсов.

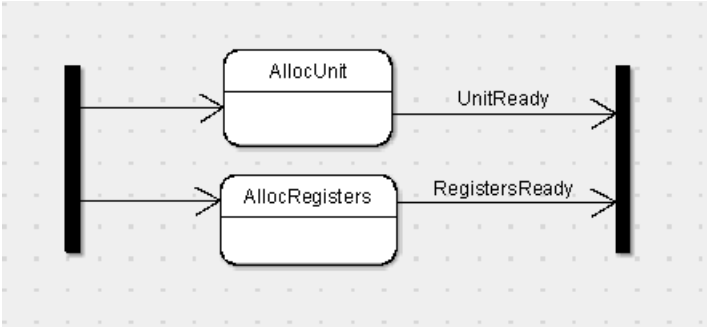


*Диаграмма 2.*

Состояния AllocUnit и AllocRegisters являются конкурентными – после прохождения псевдосостояния fork (слева на диаграмме) автомат находится этих двух состояниях *одновременно*. Псевдосостояние join (справа на диаграмме) становится активным после наступления событий UnitReady и RegistersReady (их порядок не играет роли).

## Реализация на C++

Попробуем спроектировать библиотеку, которая позволит максимально просто создавать конечные автоматы. Основные требования к ее функциональности:

- При помощи параметризации и расширения на основе библиотеки можно создавать конкретные конечные автоматы; код самой библиотеки при этом не должен модифицироваться;
- Библиотека должна поддерживать конкурентные активные состояния (необходимо для работы с композитными состояниями и поддержки семантики fork/join);
- Ключевое требование для реализации больших систем – возможность повторного использования компонентов при помощи вложенных машин состояний.
- Как можно большее число проверок корректности описания автомата должно производиться при создании исходного кода, а не во время выполнения приложения.

Учитывая эти требования, можно разделить любой конечный автомат на следующие компоненты:

- Объекты, представляющие состояния и переходы между ними, события, механизм вычисления активных состояний;
- Код инициализации, создающий указанные выше объекты и компонующий их в соответствии с описанием автомата (генерируется автоматически);
- Контекст, хранящий информацию для вычисления сторожевых условий.

Очевидно, что код инициализации и определение контекста специфичны для каждого конечного автомата и не могут быть помещены в библиотеку.

Какие классы должны быть включены в библиотеку? Начнем с самого простого: событий. События, определяемые пользователем, должны быть наследованы от класса BaseEvent:

```
struct BaseEvent
{
  virtual ~BaseEvent() {}
};
```

Хотя это и менее эффективно с точки зрения производительности, мы определяем события как объекты, а не элементы перечисления, поскольку это дает возможность хранить в объекте информацию, связанную с событием. Например, для события «в торговый автомат опущена монета» это может быть номинал монеты:

```
class EvReceivedCoin : public BaseEvent {
  int value_;
};
```

Следующий из основных классов библиотеки – базовый класс состояния. Состояния должны позволять:

Определять entry- и exit-действия в пользовательских наследованных классах.

Устанавливать переходы в другие состояния по событиям или сторожевым условиям.

Определять множество исходящих разрешенных переходов для очередной конфигурации машины состояний.

Листинг ниже содержит интерфейс и реализацию ключевых функций класса BaseState (здесь и далее реализация упрощена за счет отсутствия композитных состояний). Отметим, что BaseState параметризован классом контекста, поскольку для вычисления условий и вызова действий состояния должны «знать» тип своего контекста.

```
template<class CONTEXT>
class BaseState
{
public:
  // добавить триггерный переход
  template<class E_TYPE> void connect(BaseState<CONTEXT> *target, void
(CONTEXT::*action)() = 0)
  {
    new TriggeredTransition(this, target, &typeid(E_TYPE), action);
  }

  // добавить переход по условию
  template<class Predicate> void connect(BaseState<CONTEXT> *target,
    const Predicate &guard, void (CONTEXT::*action)() = 0)
  {
    new GuardedTransition<Predicate>(this, target, guard, action);
  }

protected:
  friend class StateMachine;

  struct BaseTransition
  {
    BaseState<CONTEXT> *source_;
    BaseState<CONTEXT> *target_;
    void (CONTEXT::*action_)();

    BaseTransition(BaseState<CONTEXT> *source, BaseState<CONTEXT> *target,
      void (CONTEXT::*action)()) :
      source_(source), target_(target), action_(action)
    {
      source_->outgoing_.push_back(this);
      target_->incoming_.push_back(this);
    }
    virtual ~BaseTransition() {}

    virtual bool enabled(CONTEXT *, const type_info *) = 0;
  };

  vector<BaseTransition *> outgoing_;
  vector<BaseTransition *> incoming_;

  virtual bool is_initial() { return false; }
  virtual bool is_final() { return false; }
  virtual bool is_fork() { return false; }

  CONTEXT *context() { return context_; }

  virtual void entry_action(const BaseEvent *event) {}
  virtual void exit_action(const BaseEvent *event) {}

  virtual bool can_activate() { return true; }

private:
  template<class Predicate>
```

```
  struct GuardedTransition : public BaseTransition
  {
    Predicate guard_;

    GuardedTransition(BaseState<CONTEXT> *source, BaseState<CONTEXT> *target,
      const Predicate &guard, void (CONTEXT::*action)()) :
      BaseTransition(source, target, action), guard_(guard) {}

    virtual bool enabled(CONTEXT *context, const type_info *)
    {
      return guard_(context);
    }
  };

  struct TriggeredTransition : public BaseTransition
  {
    const type_info *event_id_;

    TriggeredTransition(BaseState<CONTEXT> *source, BaseState<CONTEXT> *target,
      const type_info *event_id, void (CONTEXT::*action)()) :
      BaseTransition(source, target, action), event_id_(event_id) {}

    virtual bool enabled(CONTEXT *context, const type_info *event_id)
    {
      return (*event_id_ == *event_id) != 0;
    }
  };

  CONTEXT *context_;

  // формирует множество разрешенных переходов для следующего шага
  // если event_id == NULL, может быть выбраны только переходы по условию
  void process_event(const type_info *event_id, vector<BaseTransition*> &enabled)
  {
    for (vector<BaseTransition*>::iterator it = outgoing_.begin();
    it != outgoing_.end(); it++)
    {
      if ((*it)->enabled(context(), event_id))
      {
        assert(enabled.size() == 0 || is_fork());
        enabled.push_back(*it);
      }
    }
  }
};
```

BaseState является полноценным классом, который может быть использоваться для представления конкретных состояний. Он также может быть использован в качестве базового класса. Состояние, выводящее сообщения при своей активации и деактивации, может быть определено следующим образом:

```
class ReportingState : public hsm::BaseState<Context>
{
public:
  virtual void entry_action(const BaseEvent *event)
  {
    cout << "State activated\n";
  }

  virtual void exit_action(const BaseEvent *event)
  {
    cout << "State deactivated\n";
  }
};
```

**ПРИМЕЧАНИЕ**

Библиотечные классы помещены в пространство имен hsm.

Состояния могут быть связаны друг с другом триггерным переходом:

```
hsm::BaseState<Context> *s1, *s2;
...
s1->connect<EvReceivedCoin>(s2, &Context::action);
```

или условным переходом:

```
s1->connect(s2, mem_fun(&Context::pin_valid));
s1->connect(s3, not1(mem_fun(&Context::pin_valid)));
```

Для того чтобы определить, активируется ли триггерный переход поступившим событием, необходимо сравнивать тип событий друг с другом. Для этого используется структура type_info, возвращаемая оператором typeid. Таким образом, каждому классу событий автоматически присваивается уникальный идентификатор, что избавляет пользователей от необходимости создавать идентификаторы самостоятельно.

В качестве сторожевых условий используются объекты-функторы (вместо указателей на методы контекста). Это дает возможность легко конструировать новые условия на базе уже определенных, как показано в предыдущем примере. Из состояния s1 ведут два перехода, один их которых активируется при условии pin_valid, второй – в противном случае. Для создания безусловных переходов используется предикат

```
struct Unconditional
{
  bool operator()(...) {  return true; }
};
```

Псевдосостояния, обладающие специфическим поведением, также наследованы от класса BaseState. В качестве примера рассмотрим реализацию тесно связанных состояний fork/join.

```
template<class CONTEXT>
class ForkState : public BaseState<CONTEXT>
{
public:
  ForkState(const string &name = "ForkState") : BaseState<CONTEXT>(name) {}

  virtual bool is_fork() {  return true; }
};

template<class CONTEXT>
class JoinState : public BaseState<CONTEXT>
{
  typename vector<BaseTransition*>::size_type num_joined_;

public:
  JoinState(const string &name = "JoinState") : BaseState<CONTEXT>(name),
    num_joined_(0) {}

protected:
  virtual bool can_activate()
  {
    if (++num_joined_ >= incoming_.size())
    {
        num_joined_ = 0;
        return true;
    }
    return false;
  }
};
```

Все, что требуется для реализации псевдосостояния fork – переопределить виртуальную функцию is_fork. Это разрешит функции BaseState::process_event возвращать более одного разрешенного перехода (условие assert(enabled.size() == 0 || is_fork()) всегда выполняется для состояний fork). Вообще говоря, такая зависимость базового класса от производного не очень желательна, но в данном случае мы

не ожидаем расширения иерархии производных классов BaseState и поэтому можем проигнорировать это соображение.

Также несложно реализовать поведение класса JoinState. Функция can_activate (она вызывается каждый раз, когда одно из присоединенных конкурентных состояний переходит в псевдосостояние join) подсчитывает число сработавших входящих переходов. Когда это число совпадает с общим числом входящих переходов, can_activate сигнализирует о том, что состояние может стать активным.

Все компоненты конечного автомата связывает вместе класс StateMachine:

```
template<class CONTEXT>
class StateMachine
{
  typedef BaseState<CONTEXT> State;

  CONTEXT context_;
  vector<State*> states_;
  vector<State*> active_states_;

public:
  void add(BaseState<CONTEXT> *state)
  {
    if (state->is_initial())
    {
      assert(active_states_.size() == 0);
      active_states_.push_back(state);
    }
    state->context_ = &context_;
    states_.push_back(state);
  }


  void post_event(BaseEvent *event);

  bool stopped();
};
```

Рассмотрим подробнее, как реализован метод StateMachine::post_event. Для каждого активного состояния мы вычисляем множество активных состояний, полученных из оригинального для следующего шага. Итерации повторяются до тех пор, пока автомат не попадет в так называемую стабильную конфигурацию, не изменяющуюся на очередном шаге.

Для активации только условных переходов используется внутреннее событие NullEvent. Поскольку триггерные переходы по этому событию не могут быть созданы пользовательским кодом, NullEvent никогда не активирует триггерные переходы.

```
void post_event(BaseEvent *event)
{
  struct NullEvent : public BaseEvent {};
  static const type_info *null_event = &typeid(NullEvent);

  const type_info *ti = event ? &typeid(*event) : null_event;
  bool configuration_changed;
  do
  {
    vector<State*> new_active;
    configuration_changed = false;
    for (vector<State*>::iterator s = active_states_.begin();
    s != active_states_.end(); s++)
    {
      vector<State::BaseTransition*> transitions;
      (*s)->process_event(ti, transitions);
      if (!transitions.empty())
      {
        (*s)->exit_action(event);
        for (vector<State::BaseTransition*>::iterator t = transitions.begin();
```

```
        t != transitions.end(); t++)
        {
          if ((*t)->action_)
          {
            (context_.*(*t)->action_)();
          }
          if ((*t)->target_->can_activate())
          {
            (*t)->target_->entry_action(event);
            new_active.push_back((*t)->target_);
          }
        }
        configuration_changed = true;
      }
      else
      {
        new_active.push_back(*s);
      }
    }
    active_states_ = new_active;
    event = 0;
    ti = null_event;
  }
  while (configuration_changed);
}
```

Все события и состояния обрабатываются в контексте одного потока. Это упрощает реализацию, а также позволяет получить полностью воспроизводимое поведение автомата при одинаковых начальных условиях. Недостатком однопоточной модели является отсутствие поддержки do activity – выполнение этих действий не может быть завершено при получении внешнего события.

## Автоматическое генерирование кода

Начав использовать конечные автоматы, мы быстро столкнулись с проблемой корректного создания машины состояний. Очевидно, что машина, конструируемая во время выполнения, должна полностью соответствовать своему описанию на диаграмме UML. Это значит, что объекты C++, представляющие состояния и переходы между ними должны быть созданы в строгом соответствии с элементами диаграммы.

Практика показала, что уже для автоматов среднего размера синхронизация UML-диаграмм и C++-кода становится достаточно сложной задачей. При этом возникает большое число ошибок, на поиск и исправление которых приходится тратить все больше времени. Естественным выходом стало автоматическое генерирование кода C++ из диаграмм UML. Мы сразу же отказались от идеи полного создания кода по диаграмме, поскольку в этом случае элементы диаграммы оказались бы перегружены информацией, необходимой для создания объявлений классов и определения их методов. Весь этот код по-прежнему должен быть написан разработчиком. Вместо этого автоматически создается относительно небольшой фрагмент кода, определяющий структуру машины состояний. В этом коде, тем не менее, было сосредоточено большинство ошибок, связанных с созданием машины состояний.

В качестве средства моделирования и генерирования кода был выбран бесплатный продукт с открытым исходным кодом ArgoUML. По всей видимости, из бесплатных средств UML моделирования именно ArgoUML обладает наиболее богатой функциональностью. При этом доступность исходного кода позволяет создавать расширения, возможности которых не ограничены набором API и точек расширения, предусмотренными разработчиками продукта.

Принцип генерирования кода, конструирующего машину состояния, весьма прост. Диаграмма состояний, созданная в ArgoUML, имеет разделенное на две части внутреннее представление. Это собственно модель UML, хранящая элементы конечного автомата и их свойства, а также информация для визуального представления элементов на диаграмме (размеры графических элементов, их расположение на диаграмме и т.п.) В данном случае нас интересует только модель UML, по которой генерируются фрагменты кода, выполняющие следующие задачи:

Создание объектов, соответствующих состояниям на диаграмме;

Установление композиционных связей между созданными объектами (состояния, содержащиеся в композитном состоянии, должны «знать» о своем контейнере);

Установление переходов между состояниями.

Одновременно с генерированием кода производится проверка UML модели. Сообщения об обнаруженных ошибках, таких как переход из состояния в само себя, записываются в создаваемый файл C++ в виде комментариев.

# Пример: сценарий авторизации пользователя

### Шаг первый: описание конечного автомата на диаграмме состояний

Приведем в качестве примера реализацию авторизации клиентов в некотором коммуникационном протоколе. В простейшем случае система ожидает ввода логина и пароля пользователя, затем проверяет их соответствие разрешенным значениям и в зависимости от результата проверки авторизует пользователя или возвращается к ожиданию нового логина. Соответствующая диаграмма состояний UML может выглядеть подобным образом:

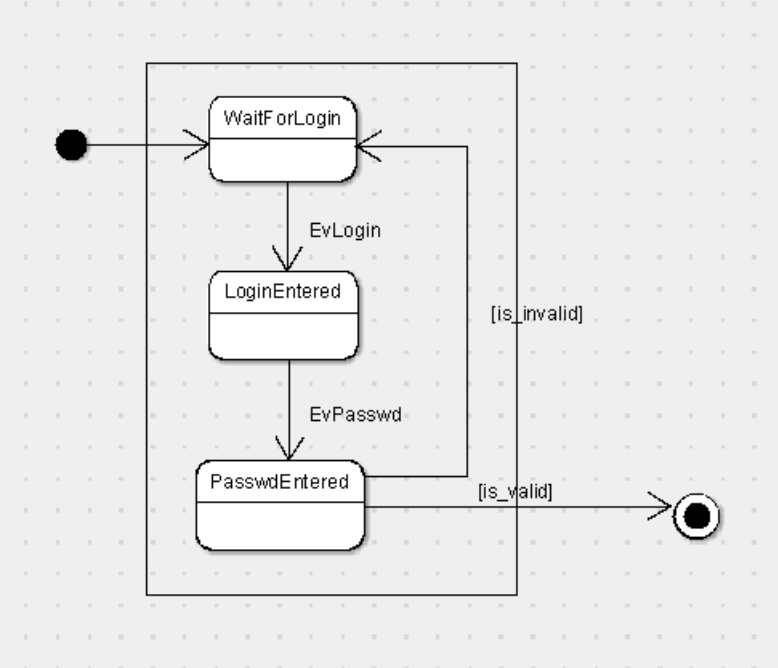


*Диаграмма 3.*

Из исходного состояния система сразу попадает в состояние WaitForLogin. Оно остается активным до поступления события EvLogin, которое возникает после получения от пользователя логина. События EvLogin активирует триггерный переход в состояние LoginEntered (ожидание пароля). После получения события EvPasswd система перейдет в состояние PasswdEntered; теперь системе известны и логин и пароль.

Исходящие переходы из состояния PasswdEntered активируются сторожевыми условиями is_valid и is_invalid, изображенными на диаграмме в квадратных скобках. В зависимости того, какое из условий принимает значение true, система возвращается к ожиданию логина или переходит в финальное состояние LoggedIn, обозначающее успешное завершение авторизации. Следует заметить, что поскольку одно из сторожевых условий всегда выполняется (пароль либо верен, либо не верен), система проводит в состоянии PasswdEntered нулевое время.

Усложним пример, приблизив его к реальности. Прежде всего, введем ограничение на число попыток ввода пароля. Это улучшит защиту системы от взлома при помощи подбора паролей. Пусть в нашей

системе после трех неудачных попыток ввода пароля учетная запись пользователя блокируется, делая невозможным успешную авторизацию пользователя. Кроме того, добавим возможность возврата к выбору логина, если пользователь ошибся при его вводе.

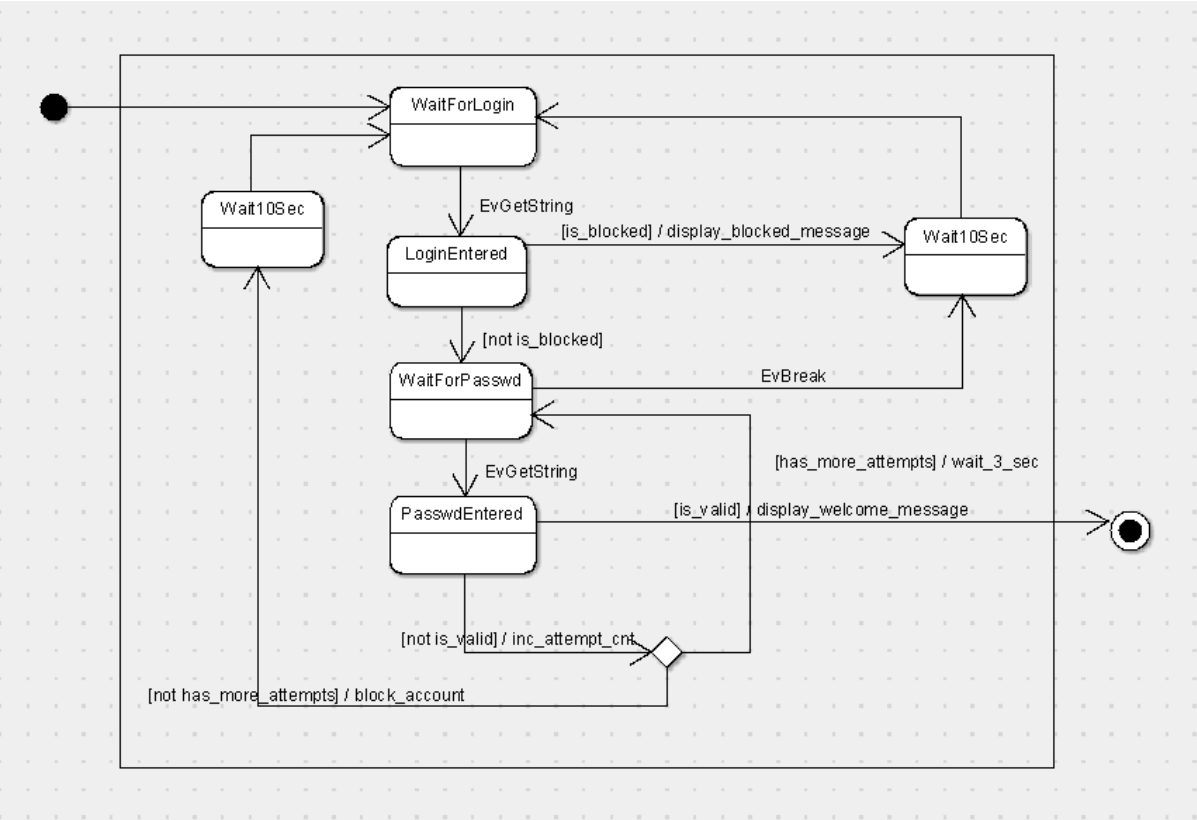


*Диаграмма 4.*

Возврат к вводу логина реализуется просто – мы добавляем исходящий переход из состояния WaitForPasswd, активируемый событием EvBreak (например, пользователь нажал на клавиатуре Ctrl-C). Чтобы защитить систему от подбора пароля, сделаем следующим активным состоянием Wait10Sec. Entry-действие этого состояния заключается в десятисекундном ожидании, после завершения которого происходит переход к ожиданию ввода нового логина.

Добавление счетчика попыток ввода пароля является более сложной задачей. Наиболее очевидное решение – создание трех состояний PasswdEntered2, PasswdEntered1 и PasswdEntered0, где цифра в конце названия состояния показывает число оставшихся попыток ввода пароля. Однако этот подход вряд ли устроит нас, если потребуется разрешить 100 попыток ввода, и вообще неприменим в ситуации, когда число попыток может быть задано во время выполнения.

Универсальным решением является запись текущего числа попыток в контексте, связанном с конечным автоматом. Этот счетчик обнуляется, когда становится активным состояние WaitForLogin, увеличивается при выполнении действия inc_attemp_cnt и проверяется условием has_more_attempts. За универсальность этого подхода нам, однако, приходится платить менее прозрачным отображением поведения системы на диаграмме.

## Также в этом варианте вместо условия is_invalid мы используем отрицание условия is_valid (для исключения дублирования кода).

## Шаг второй: реализация классов C++

Начнем с реализации контекста. В него мы включаем необходимые данные, определяем условия и действия, связанные с переходами.

```
class Context
{
```

```
  struct AccountInfo
{
    ...
  };

  std::vector <AccountInfo> accounts_;

  AccountInfo *find_account(const string &login);

public:
  string login_;     // current login.
  string password_;   // current password.
  int  attempt_cnt_;  // attempt counter

  Context();

  //========================================================================
  // Guards
  //========================================================================
  bool is_valid();
  bool is_blocked();
  bool has_more_attempts();

  //========================================================================
  // Actions
  //========================================================================
  void inc_attempt_cnt();
  void block_account();
  void display_welcome_message();
  void display_blocked_message();
  void wait_3_sec();
};
```

Определим внешние события. В нашем примере их всего два: ввод строки и сигнал прерывания.

```
struct EvGetString : public hsm::BaseEvent
{
  string value_;
};

struct EvBreak : public hsm::BaseEvent {};
```

Теперь необходимо определить классы состояний. В качестве примера приведем состояние LoginEntered:

```
class LoginEntered : public hsm::BaseState<Context>
{
public:
  virtual void entry_action(const hsm::BaseEvent *event)
{
    context()->login_ = static_cast<const EvGetString*>(event)->value_;
  }
};
```

Создадим машину состояний в строгом соответствии со спецификацией (как правило, следующий фрагмент кода автоматически генерируется по диаграмме).

```
hsm::StateMachine<Context> sm;

hsm::BaseState<Context> *s_0 = new hsm::InitialState<Context>;
hsm::BaseState<Context> *s_1 = new WaitForLogin;
hsm::BaseState<Context> *s_2 = new LoginEntered;
...
sm.add(s_0);
sm.add(s_1);
sm.add(s_2);
...
s_0->connect(s_1, hsm::Unconditional());
```

```
s_1->connect<EvGetString>(s_2);
s_2->connect(s_5, mem_fun(&Context::is_blocked),
  &Context::display_blocked_message);
s_2->connect(s_3, not1(mem_fun(&Context::is_blocked)));
...
```

Теперь у нас есть все необходимое для запуска машины состояний. Для этого в цикле будем получать внешние события и при помощи метода post_event передавать их в машину. Цикл обработки событий заканчивается при достижении автоматом финального состояния.

```
EvBreak ev_break;
EvGetString ev_string;

sm.init();
while (!sm.stopped())
{
  std::getline(cin, ev_string.value_);
  if (ev_string.value_.empty())
  {
    sm.post_event(&ev_break);
  }
  else
  {
    sm.post_event(&ev_string);
  }
}
```

# Выводы и перспективы

В реальных проектах нами были разработаны достаточно сложные конечные автоматы, включающие несколько сотен состояний. Практика показала, что для эффективного использования этого подхода ключевыми являются следующие факторы:

Как уже упоминалось, автоматическое генерирование кода по UML диаграммам.

Поддержка псевдосостояний fork/join и, соответственно, конкурентных активных состояний.

Возможность использования вложенных машин состояний.

Последний пункт означает, что мы можем использовать созданные и отлаженные конечные автоматы в автоматах более высокого уровня в качестве компонентов. Для этой цели в нотации UML используются элементы *submachine state*, которые позволяют ссылаться на другие автоматы в UML-модели. При этом в контексте содержащего автомата вложенная машина состояний обладает поведением обычного композитного состояния.

Эффективному использованию конечных автоматов мешает, прежде всего, отсутствие удобных средств отладки. Поэтому представляется перспективным разработка расширения ArgoUML, которое позволит подключится к отлаживаемому приложению и, например, автоматически подсвечивать активные состояния.