Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра “Компьютерных технологий ”

М.И. Гуисов, А.Б. Кузнецов, А.А. Шалыто

Интеграция механизма обмена сообщениями в Switch-технологию

Санкт-Петербург
2003

Введение

При использовании Switch-технологии [1] для спецификации алгоритмов предлагается применять системы взаимосвязанных автоматов. Однако предложенные в этой технологии способы взаимодействия автоматов не позволяют эффективно решать некоторые задачи, например, приведенные в разд. 3.1.

1. Постановка задачи

Данная работа призвана устранить этот недостаток путем введения нового механизма взаимодействия автоматов – механизма обмена сообщениями (**МОС**). В настоящей работе он реализуется с помощью **библиотеки** «swmem» (SWitch Message Exchange Mechanism), приведенной в Приложении. В качестве **критериев оценки** механизмов взаимодействия будем учитывать естественность решения прикладной задачи и наглядность графов переходов, описывающих поведение автоматов.

В работе [1] предлагается использовать следующие типы **входных воздействий**:

- *входные переменные* x_i – в общем случае булевы формулы, вычисляющие значения заданных логических выражений;
- *внешние события* e_i - события, происходящие **вне** системы автоматов. Информация о произошедшем событии может, например, обрабатываться «налету» или помещаться в очередь, а затем поэлементно обрабатываться;
- *внутренние события* e_i – механизм взаимодействия автоматов, основанный на **вызываемости**;
- *внутренние переменные* Y_k (в объектно-ориентированном программировании доступ к этим переменным целесообразно реализовать через интерфейс) – механизм взаимодействия автоматов, основанный на том, что каждый автомат предоставляет другим автоматам информацию о собственном состоянии. Этот механизм наиболее естественным образом позволяет синхронизировать работу автоматов [2,3].

Применение механизма взаимодействия на основе внутренних событий в Switch-технологии ограничено в силу его одностороннего характера – вызванный автомат не может вызвать вызывающий автомат, так как при этом исходный (вызывающий) автомат запускается повторно, не окончив предыдущий цикл работы.

Обмена информацией о состояниях автоматов оказывается недостаточно для эффективного и наглядного решения некоторых классов задач, которые будут рассмотрены ниже.

2. Идеи, лежащие в основе механизма обмена сообщениями

Для устранения указанных выше недостатков в настоящей работе предлагается пятый тип входных воздействий – ***сообщения***. Необходимо отметить, что под термином «сообщение» понимается не вызов метода автоматного класса, а создание экземпляра объекта типа «сообщение». Кроме того, предполагается рассылка его **всем** автоматам, входящим в систему.

Механизм взаимодействия автоматов, основанный на обмене сообщениями, заключается в следующем:

- автомату разрешается сигнализировать об изменении состояния (совершении перехода) и других изменениях в среде, вызванных его выходными воздействиями, путем создания сообщений и помещения их в очередь;
- любой объект получает все сообщения из очереди, и если конкретное сообщение направлено объекту, обрабатывает его.

Обычно внешние события порождаются изменениями в окружающей среде. При этом предполагается, что скорость работы автомата больше скорости возникновения этих изменений. Поэтому для всех внешних событий имеется возможность их последовательной обработки – для каждого внешнего события может происходить переход автомата.

В случае с сообщениями нельзя полагаться на разность скоростей создания и обработки сообщений, так как на каждом шаге работы автомата может создаваться любое их количество. Эта ситуация

объясняет одно из свойств сообщений – все сообщения должны быть обработаны на следующем шаге после генерации.

Для идентификации сообщения используется два числа: **id** (тип сообщения) и **src** (уникальный идентификатор источника сообщения).

Необходимость использования переменной **src** связана, в основном, с созданием нескольких экземпляров одного автомата. Идентификация двумя числами носит также и логический смысл. Например, сообщение «нажата клавиша **w**» идентифицируется типом – «нажата клавиша» и источником – «клавиша **w**».

Ниже приведен шаблон для класса «Сообщение», учитывающий указанные выше факторы.

Шаблон 1. Класс «Сообщение»

```
class CSMessage {
public:
     long int    id;                       // Идентификатор сообщения (на графе переходов)
     CSAutomate  *src;                     // Источник сообщения (экземпляр автомата)
     CSMessage(long int    _id  = -1,      // Идентификатор сообщения на графе
               CSAutomate  *src = NULL);   // Адрес объекта-источника сообщения
};
```

Необходимо отметить, что как внешние, так и внутренние события (в обычном понимании) «доставляются» автоматам в виде сообщений. Кроме того, сообщения используются для организации взаимодействия автоматов. Поэтому предлагается сообщения по виду объекта – источника классифицировать на *автоматные* (сообщения от автоматов) и *внешние* (от остальных источников).

3. Механизм обмена сообщениями на этапе проектирования

Предположим, что у нас есть система автоматов, нарисованы графы переходов. Остается надписать условия. Благодаря наличию у сообщений двух полей идентификации, некоторые условия могут иметь сложный вид (например, id=10 && (src=3 || src=7)). Явное указание этих условий уменьшает наглядность графов переходов. Чтобы не загромождать графы, выписываем все условия по сообщениям в столбик. При этом будет удобно сначала «регистрировать» внешние, затем автоматные сообщения. После этого все строки нумеруются. Затем на графе вместо сложных условий, пишем m_i, где **i** – определенный ранее номер условия. При желании можно явно указывать тип сообщения. Например, автоматные сообщения можно обозначать в виде a_i, а внешние – в виде b_i. При этом нумерация должна быть «сквозной» (общей) для обоих типов сообщений. Заметим, что на уровне реализации это указание не вызовет изменений – при реализации смотрим на граф, затем находим полное условие в таблице.

3.1. Задача идентификации дуг

Покажем простоту и наглядность применения механизма обмена сообщениями на примере *обобщенной задачи идентификации дуг в графе переходов*. Эта задача заключается в отслеживании переходов в конкретном графе. Иными словами, необходимо знать не только состояние автомата в данный момент времени, но и последний совершенный им переход или предыдущее состояние (в случае только одного перехода между рассматриваемыми состояниями). Использование флага (внешней переменной) считается запрещенным. Поведение автомата должно оставаться неизменным независимо от необходимости наблюдения за ним.

Рассмотрим четыре возможных решения этой задачи:

- прямая нумерация дуг на графе переходов;
- формальное решение (преобразование графа «под» Switch-технологию);
- применение механизма обмена сообщениями (МОС);
- использование истории состояний**.** При этом требуется изменение шаблона автомата, предложенного в работе [1].

3.1.1. Прямая нумерация дуг на графе переходов

Прямая нумерация дуг кажется наиболее очевидным методом, однако ее применение значительно ухудшает наглядность графа, так как необходимо ставить номера дуг перед условием переходов и явно выделять петли во всех состояниях. Поэтому этот метод неприемлем по критерию наглядности графа.

3.1.2. Формальное решение

Формальное решение – преобразование графа переходов по специальному правилу. Оно позволяет другим автоматам фиксировать событие выхода автомата из некоторого состояния, не требующее изменения шаблона автоматного класса. При этом решении не требуется передача сообщений.

Рассмотрим два крайних по сложности варианта постановки сформулированной выше задачи и для каждой из них предложим формальное решение.

3.1.2.1. Отслеживание выхода автомата из некоторого состояния

Пусть требуется отслеживать выход автомата **A только** из состояния **N**.

Для решения этой задачи необходимо выполнить следующее. Рядом со всеми вершинами в графе переходов, в которые есть дуги из вершины **N** (петля также является переходом), создаются **вершины – дубликаты**, и к их номеру добавляется «штрих». Все дуги, ведущие из вершины **N**, перенаправляются в соответствующие «штрихованные» вершины. Все дуги из вершин, для которых есть «штрихованные», дублируются, и **начало дуги – дубликата** «привязывается» к «штрихованной» вершине. Если некоторому объекту (не обязательно автомату) необходимо узнать о выходе автомата **A** из состояния **N**, то он проверяет, является ли его текущее состояние «штрихованным».

Введенный термин *штрихованная вершина* не означает использования «штрихов» в формальной записи номера состояния. Для идентификации этих вершин выделяются дополнительные, неиспользованные при нумерации исходного графа, номера. Ввиду того, что на графе переходов «штрихованные» вершины располагаются рядом с соответствующими «нештрихованными» вершинами, путаницы или непонимания не возникает. Заметим, что если некоторый автомат – наблюдатель проверял нахождение автомата **A** до его преобразования в состоянии **Y**, а после преобразования появилась вершина **Y'**, то проверка должна осуществляться на нахождение автомата **A** в обоих состояниях (**Y** и **Y'**).

3.1.2.2. Отслеживание всевозможных выходов автомата из различных состояний

Предположим, что необходимо **по отдельности** отслеживать выходы автомата **A** из **всех** его состояний. В этом случае требуется кардинальное изменение графа, которое может сильно затруднить понимание логики работы автомата. Это преобразование представляет, видимо, чисто математический интерес, и может быть использовано при «постпроцессинге» – автоматическом преобразовании уже отлаженного графа, если никакой другой способ регистрации выхода автомата из состояния неосуществим.

Для решения этой задачи обозначим все вершины нового автомата **A*** буквенными строками вида «**N-K**». Каждая такая вершина будет функционировать как вершина с номером **N** из исходного графа. Она является дубликатом вершины **N** «с точностью до номера». Если автомат **A*** находится в состоянии, соответствующем вершине «**N-K**», то можно утверждать, что на предыдущем шаге автомат находился в состоянии, имеющем в исходном графе номер **K**.

Рассмотрим все пары вершин **X** и **Y** графа **A**, соединенные дугами. Для каждой дуги создадим дубликаты, ведущие из вершин вида «**X-Z**» (**Z** «пробегает» все номера вершин исходного графа) в

вершину «**Y-X**». В результате этой операции в новом графе может оказаться некоторое количество «висящих» (недостижимых) вершин. Вершина, соответствующая **исходному** состоянию, «висящей» не считается. Удалив «висящие» вершины, получим конечный граф **A***. Информации о состояниях автомата **A*** будет достаточно для идентификации состояния и последнего перехода автомата **A**.

3.1.3. Механизм обмена сообщениями

Перейдем к решению задачи идентификации с применением МОС. У всех переходов, ведущих **из** интересующего нас состояния, в выходных воздействиях формируется сообщение с полем *src*, указывающим номер автомата и идентификатор дуги (**A**W: **05**11). Получив это сообщение, наблюдатель узнает, какой переход совершил интересующий его автомат. В случае нескольких автоматов при создании сообщения изменяется поле источника, а номер сообщения во всех автоматах один и тот же.

3.1.4. История состояний

Основная идея этого подхода состоит во введении в шаблон автомата дополнительной переменной, запоминающей последнее состояние, в котором находился автомат. Каждый раз, совершая переход, автоматная процедура обновления помещает предыдущее состояние в определенную переменную. По аналогии с переменными состояния **y_o** (old) и **y_n** (new), используемыми в шаблоне автомата [1], новую переменную логично назвать **y_h** (history).

Любой внешний объект, имея доступ к этим переменным, сможет определить последний совершенный автоматом переход. Единственный недостаток у этого решения задачи идентификации перехода заключается в следующем: в тех случаях, когда из вершины графа **N** в вершину **M** идет более одной дуги, идентификация этих дуг их концами невозможна.

Перейдем к рассмотрению примеров, в которых используется каждый из указанных выше походов.

3.2. Примеры задачи идентификации дуг

3.2.1. Циклические процессы

Пример 1. Предположим, что автомат из исходного состояния может запустить несколько циклических процессов. На следующем шаге наблюдателю необходимо узнать номер запущенного процесса.

Уточним постановку задачи**.** Пусть задан автомат, реализующий три циклических процесса (рис.1). Кроме того, задано начальное состояние, не относящееся ни к одному из процессов. Предположим, что из этого состояния переходы ведут в вершины, соответствующие начальным конфигурациям циклических процессов. В некотором объекте-наблюдателе необходимо зафиксировать момент начала выполнения одного из процессов (выход из начального состояния) с учетом процесса, который был запущен. Ввиду того, что процессы циклические, проверка на нахождение автомата в состоянии, соответствующем начальной конфигурации какого-либо процесса (состояния 10, 20 или 30 на рис.1), ничего не даст. Действительно, автомат может вернуться в это состояние после одной итерации циклического процесса.

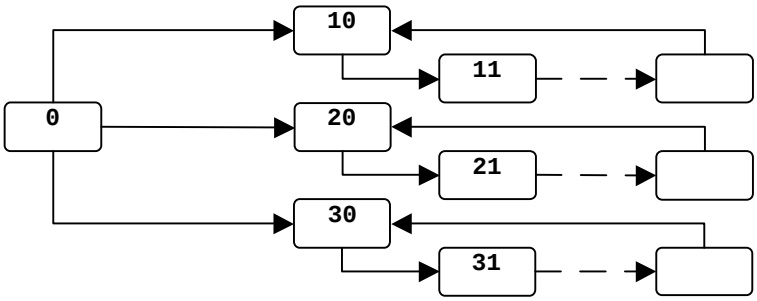

Рис. 1. Исходный граф переходов автомата

3.2.1.1. Формальное решение

Формальное решение (добавление в граф «сигнальных» вершин **10-0**, **20-0** и **30-0**) изменяет граф и делает его менее наглядным (рис.2).

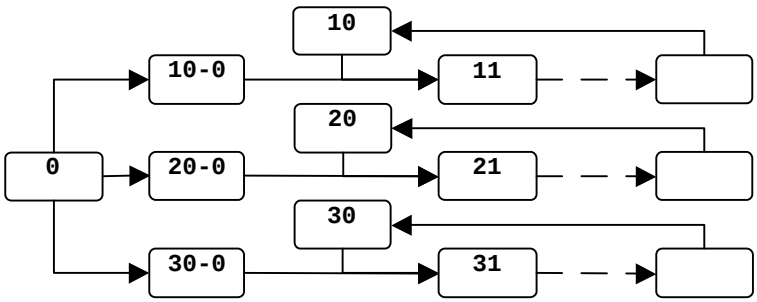

Рис. 2. Граф переходов автомата, соответствующий формальному решению

3.2.1.2. Механизм обмена сообщениями

Приведем решение с использованием МОС. При этом создается сообщение в выходных воздействиях **z1**, **z2** и **z3**. На следующем шаге любой внешний объект, получив сообщение, «**узнает**» о начале процесса.

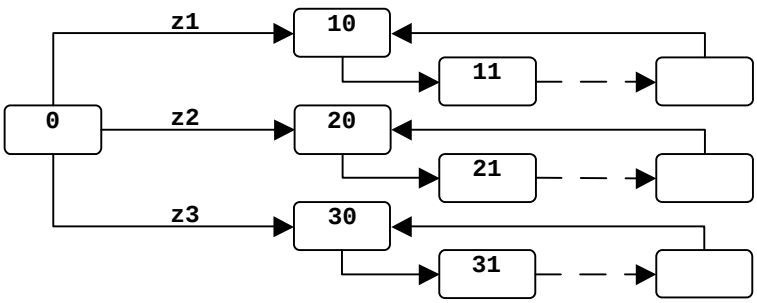

Рис. 3. Граф переходов автомата, соответствующий решению с использованием механизма обмена сообщениями

3.2.1.3. История состояний

Решение с применением истории состояний не изменяет исходного графа автомата.

Введение **истории состояний** также, как и использование МОС, требует изменения шаблона автомата. В этом методе всю работу берет на себя новый шаблон автомата, а не разработчик автомата. Метод

обладает обратной совместимостью – если какому-то объекту понадобится зафиксировать выход автомата **A** из состояния **N,** то это не потребует изменений в этом автомате.

Суть метода заключается в добавлении в шаблон переменной **y_h**, в которой хранится номер состояния, предшествующего текущему. В процедуре обновления в переменную **y_h** заносится значение переменной **y_o** и только после этого переменной **y_o** присваивается значение переменной **y_n**. Также в шаблон добавляется функция, сравнивающая значения этой переменной с заданным числом. Таким образом, наблюдатель может зафиксировать выход автомата из некоторого состояния.

Последний метод кажется наиболее естественным при решении этой задачи, однако он не так универсален, как МОС, так как даже небольшое изменение требований наблюдения не позволяет его использовать.

3.2.2. Состояние ошибки

Пример 2. Групповой переход из некоторого множества состояний в общее состояние ошибки позволяет сигнализировать о том, что ошибка **произошла**, однако **не несет** в общем случае информации о том, **в каком состоянии** она **возникла**.

Уточнение постановки задачи. Пусть задан автомат, в ряде вершин которого при некоторых входных данных возникает ошибка. Об этом сигнализирует переход в вершину или группу вершин, которые обозначают ошибку. Предположим, что существуют несколько типов ошибок, причем в каждой вершине могут происходить ошибки разных типов. Определение типа произошедшей ошибки и места ее происхождения является рассматриваемой задачей.

Исходный граф (рис. 4) содержит **одно** состояние **ERROR** для завершения работы в случае возникновения критической ситуации. Необходимо узнать «**со стороны**» в каком из **состояний** произошла ошибка и какого **типа** она была. Отметим, что в силу того, что в состояниях могут выполняться различные действия, то в нем возможны и **различные** ошибки. Пусть возможны три типа ошибок. В первом состоянии могут возникнуть ошибки типов «один» и «два», во втором – типов «два» и «три», а в третьем – только третьего типа.

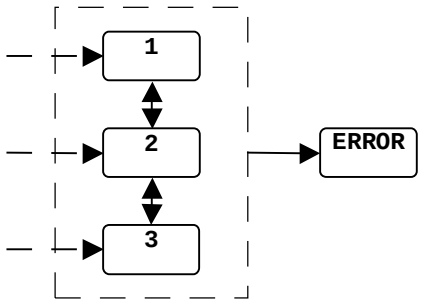

Рис. 4. Исходный граф переходов автомата

3.2.2.1. Формальное решение

Заменим исходное состояние ошибки **ERROR** пятью состояниями **ERR-ST** (рис. 5), каждое из которых сигнализирует о месте **S** и типе **T** возникновения ошибки. Создание обработчиков других ошибок требует, соответственно, введения новых состояний в граф переходов. Граф остается наглядным и логичным, однако сильно увеличивается.

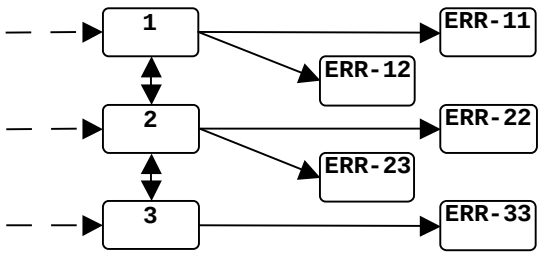

Рис. 5. Граф переходов автомата, соответствующий формальному решению

3.2.2.2. Механизм обмена сообщениями

Применение **МОС не требует** изменения графа переходов. Вместо этого, выходные воздействия на переходах из конкретных состояний в состояние ошибки дополняются созданием соответствующих **сообщений**.

3.2.2.3. История состояний

Использование **истории состояний** также требует изменения графа – введения состояний **ERR-T** для сигнализации типа **T** возникшей ошибки (рис.6). В переменной **y_h** «**запоминается**» номер предыдущего состояния. Поэтому, если в различных состояниях автомата возможны ошибки одного типа, количество **новых** состояний уменьшается по сравнению с **формальным решением**.

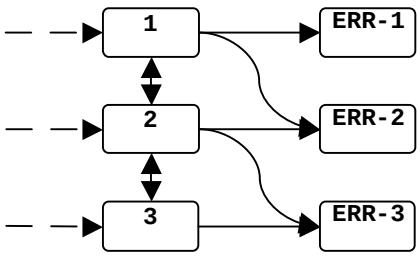

Рис. 6. Граф переходов автомата, соответствующий решению, использующему историю состояний

4. Механизм обмена сообщениями на этапе реализации

Перечислим «последствия» применения МОС:

- требование синхронности автоматов приводит к необходимости разделения логики работы автомата;
- сохранение механизма приоритетов приводит к появлению переменной **priority**;
- различение парных дуг приводит к появлению переменной **action**;
- сохранение отношения вложенности определяет порядок вызова вложенных автоматов;
- изображение сообщений на графе требует введения процедуры **Translate**.

Определим **шаг**, как единицу времени, в течение которой все автоматы совершают не более одного перехода. **Деревом вызовов** будем называть последовательность запуска всех автоматов на каждом шаге, соответствующую вложенности автоматов. Естественно потребовать, чтобы все сообщения, помещенные в очередь на некотором шаге, были рассмотрены на следующем шаге. Назовем это требование **условием синхронизации**.

Это условие не выполняется в Switch-технологии – формальная реализация автомата совмещает выбор перехода, собственно переход и выходные воздействия в одной процедуре. Следовательно, запущенный с некоторым сообщением (событием) автомат, **обязательно** совершит переход, быть может, по пустой (без действий и с умалчиваемыми условиями) петле. Ввиду того, что сообщения рассылаются **всем** автоматам, обработка **одного** сообщения является шагом.

По этой же причине невозможно взаимодействие **независимых** автоматов на основе внутренних событий, так как непосредственный запуск **не вложенного** автомата с неким событием «провоцирует» **рассинхронизацию**, поскольку вызываемый автомат совершает **два** перехода на **одном** общем шаге – второй запуск происходит в дереве вызовов.

Для выполнения условия синхронизации автоматная процедура разбивается на **две части**: процедуру выбора перехода в автоматном графе (**A**-процедура) и процедуру, выполняющую переход и действия на нем (**S**-процедура). Это необходимо для обработки нескольких сообщений на одном шаге и возможности непосредственного вызова автомата из любой части программы, а не только из дерева вызовов.

К сожалению, при разделении процедуры исчезает информация о **приоритетах** возможных переходов. Порядок следования условий переходов в каждом состоянии теряет свой первоначальный смысл, потому что процедура выбора перехода запускается несколько раз за шаг. Для сохранения механизма приоритетов в шаблон вводится переменная **priority**. Условия на дугах дополняются сравнением этой переменной с заданными значениями приоритетов. В процедуре обновления она обнуляется.

Также при разделении возникает «проблема множественных дуг», связанная с наличием нескольких дуг между двумя вершинами. Если это не так, то любая дуга однозначно идентифицируется своим началом и концом. На этом основывается идея разделения шага автомата на две процедуры – выбор перехода и выполнение действий на переходе. При этом во второй процедуре выбор выходного воздействия производится на основе номера вершины, куда осуществляется переход. Если же есть две или более дуги в эту вершину, то такой подход неприменим. Требуется либо запоминать номер дуги, по которой будет осуществляться переход, либо выполнять действия на переходе сразу после выбора этого перехода.

Предложим два метода запоминания дуги для осуществления перехода.

1. Нумеруются все дуги в графе, и при выборе перехода запоминается его номер. В **S**-процедуре воздействие выбирается в соответствии с этим номером.

2. Нумеруются не дуги, а выходные воздействия. Так как в упомянутой выше процедуре необходимо знать только выходные воздействия, связанные с выбранным переходом, то после его выбора запоминается номер соответствующего воздействия. Этот механизм предполагает, что на каждой дуге осуществляется только одно выходное воздействие. При этом, если раньше можно было использовать конструкцию вида «**z1**;**z2**», то теперь необходимо ввести новое обозначение «**zN**» **::=** «**z1**;**z2**» и указать на графе переходов действие **zN**. В **A**-процедуре, при выполнении условий некоторого перехода, в переменной **action** запоминается номер выходного воздействия, связанного с этими условиями. **S**-процедура содержит оператор *switch* по номерам выходных воздействий.

Второй метод предпочтительнее, так как при его использовании сохраняется большая наглядность графа. Поэтому он в дальнейшем и будет применяться.

Рассмотрим еще один вопрос, возникающий при разделении автоматной процедуры на две части – речь идет о вызове вложенных автоматов. В *Switch-технологии* существует тезис о вложенности автомата в состояние, означающий следующее. Поведение данного автомата в некотором состоянии может описываться функцией, реализованной системой автоматов. Эта система является вложенной в состояние данного автомата. Для вложенных автоматов **A**-процедура вызывается из аналогичной процедуры автомата-владельца в состоянии, в которое эти автоматы вложены. В **S**-процедуре создается *switch*-конструкция по значению состояния, в которой происходит вызов **S**-процедур вложенных автоматов. Инициализация каждого вложенного автомата осуществляется как и в *Switch-технологии* – вызовом автомата (**A**-процедуры) с сообщением инициализации $\mathbf{m_0}$.

В связи со способом изображения сообщений (разд. 3) на графе переходов (особенно для сложных условий, содержащих входные переменные и события) для формального соответствия графа и его реализации необходимо ввести процедуру-транслятор в шаблон автомата. Она должна полностью соответствовать таблице условий переходов данного автомата – получая на вход сообщение, выдавать в качестве результата номер выполненного условия. Назовем эту процедуру **Translate**.

В силу того, что все взаимодействия автоматов ограничиваются **A**-процедурой, обновление внутренних переменных можно производить в **S**-процедуре. Кроме того, необходимо выполнять начальную инициализацию внутренних переменных. Оптимальный способ – инициализация в конструкторе автоматного класса.

Шаблон 2. «Скелет» класса «автомат»

```
class CSAutomate {
protected:
     int y_o, y_n;                          // Переменные состояния в начале и в конце шага
     int priority;                          // Переменная, разрешающая механизм приоритетов
     int action;                            // Переменная, запоминающая действие

public:
     virtual void A(CSMessage &msg) = 0;  // Выбор перехода
     virtual void S() = 0;                // Осуществление перехода
     int GetState();                      // Функция доступа к переменной состояния
};
```

Пример 3. Реализовать фрагмент графа переходов, приведенный на рис. 7.

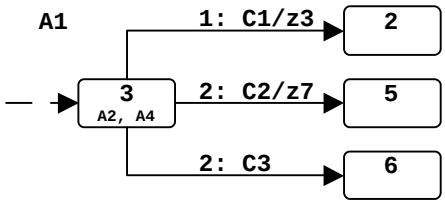

Рис. 7. Фрагмент графа переходов

На этом рисунке использованы следующие обозначения:

```
C1 = X1 && X2 && m71;     C2 = (Y3==2) && m72;      C3 = m35 || m45;
m71: id = 7, src = 1;     m72: id = 7, src > 1;
m35: id = 3, src = 5;     m45: id = 4, src = 5;
z1 – что-то сделать;      z2 – сделать что-то еще;
z3 = z1&z2;               z7 – и еще что-нибудь.
```

Этот фрагмент графа переходов с помощью предлагаемого подхода реализуется следующим образом:

```
void CAuto01::A(CMsg *msg) {
   long m = Translate(msg);
   switch(y_o) {
   ...
   case 3:
      if ((priority < 1) && x1() && x2() && m == 71)
            { y_n = 2; priority = 1; action = 3; }
      if ((priority < 2) && (A3->GetState() == 2) && m == 72)
            { y_n = 5; priority = 2; action = 5; }
      if ((priority < 2) && (m == 35 || m == 45))
            { y_n = 6; priority = 2; }
      A2->A(msg); A4->A(msg);
      break;
      ...
   }  // switch(y_o)
}

void CAuto01::S() {
   switch(action) {
   case 1: z1(); break;
   case 2: z2(); break;
   case 3: z1(); z2(); break;
   ...
   case 7: z7(); break;
   ...
   }  // switch(action)

   switch (y_o) {
```

```
    case 3: A2->S(); A4->S(); break;
    }
    y_o = y_n;
    priority = 0;
    action = 0;
}

long CAuto01::Translate(CMsg *msg) {
    ...
    if (msg->id == 3 && msg->src == 5) return 35;
    if (msg->id == 4 && msg->src == 5) return 45;
    if (msg->id == 7 && msg->src == 1) return 71;
    if (msg->id == 7 && msg->src >  2) return 72;
    ...
    return 0;
}
```

Следует отметить, что не все человечески понятные характеристики сообщений присущи внутренним сообщениям, реализуемым в МОС. Важнейшей особенностью является невозможность установки условия перехода, содержащего в себе одновременный приход двух сообщений. Иными словами, условие «**m1**&**m2**» не будет выполняться логически правильно в МОС. Это ограничение является следствием требования, предъявляемого к МОС, состоящего в минимальности изменений, вносимых в шаблон [3].

На практике это ограничение обычно выглядит следующим образом. Условие проверки нахождения двух автоматов в двух конкретных состояниях нельзя реализовать в МОС без добавления новых состояний или переходов. Например, в некоторых случаях проверку нахождения автомата **A1** в состоянии **C** можно заменить на проверку получения некоторого сообщения **m1**, добавив создание этого сообщения в выходные воздействия на дугах, входящих в состояние **C.** Для условия вида **(Y5=C)&(Y7=D)**, соответствующего автоматам **A5** и **A7** и их состояниям **C** и **D**, такая замена невозможна.

Далее рассмотрим хранение сообщений в течение шага и порядок вызова автоматных процедур на каждом шаге.

Для хранения *автоматных сообщений* (сообщений, порождаемых автоматами) создается *очередь автоматных сообщений*. Выполняя переход, любой автомат может добавить в эту очередь одно и более новых сообщений. Разбор очереди производится **полностью** на **каждом** шаге.

Информация о возникшем внешнем событии (*внешнее сообщение*) помещается в другую (*внешнюю*) очередь. Одна итерация цикла разбора этой очереди соответствует одному шагу работы системы автоматов. На каждом шаге из этой очереди извлекается **одно** сообщение (если очередь пуста, то создается «пустое» сообщение), которое после этого помещается в очередь автоматных сообщений.

Шаблон 3. Предлагаемый класс «очередь».

```
class CSQueue {                                 // Класс «очередь сообщений»
     class CSQueueItem {                        // Подкласс «элемент очереди»
     public:
          CSMessage    *msg;                    // Сообщение, хранящееся в очереди
          CSQueueItem  *next;                   // Следующий элемент очереди
          CSQueueItem(CSMessage *_msg);         // Конструктор
     };

protected:
     CSQueueItem *first;                        // «Голова» очереди
     CSQueueItem *last;                         // «Хвост» очереди

public:
     CSQueue();                                 // Создание пустой очереди
     ~CSQueue();                                // Уничтожение очереди
     bool IsEmpty();                            // Проверка на пустоту
     void AddMessage(CSMessage *msg);           // Добавление сообщения в очередь
     CSMessage *GetMessage();                   // Извлечение сообщения из очереди
};
```

Опишем алгоритм запуска автоматов:

- извлечь *сообщение* из «*внешней*» очереди и поместить в очередь *автоматных* сообщений. Если «*внешняя*» очередь пуста, то поместить «пустое» сообщение;
- пока очередь автоматных сообщений не пуста, в цикле извлечь сообщение и откорректировать выбранный переход в каждом автомате, используя это сообщение в качестве параметра функции выбора перехода;
- после того, как в очереди закончатся сообщения, каждый автомат перевести по выбранному переходу в новое состояние, выполняя соответствующее действие;
- произвести обновление (переобозначение) переменных состояния у всех автоматов.

Пример 4. Иллюстрация алгоритма (цикл разбора очереди внешних сообщений).

```
for(;;) {
    msg = ExtQueue.GetMessage();    // Извлекаем внешнее сообщение
    AQueue.AddMessage(msg);         // Помещаем его в очередь автоматных сообщений
    while(!AQueue.IsEmpty()) {      // Пока очередь автоматных сообщений не пуста...
        msg = AQueue.GetMessage();  // Извлекаем сообщение
        tree_A(msg);                // Запускаем с ним дерево процедур выбора перехода
        delete msg;                 // Уничтожаем экземпляр сообщения
    }
    tree_S();           // Осуществляем перевод всех автоматов в новые состояния
    tree_U();           // Обновляем переменные состояния у всех автоматов
}
```

5. Изменения, вносимые в шаблон для реализации автомата при интеграции механизма обмена сообщениями в Switch-технологию

- Разделение шага работы автомата на три этапа: выбор перехода, совершение действий на переходе и обновление переменной состояния.
- Введение переменной **priority** для использования механизма приоритетов при расстановке условий на дугах автоматного графа.
- Введение переменной **action** для запоминания выбранного действия и последующего его выполнения.

6. Пример проекта

В качестве примера использования МОС и предложенной библиотеки в работе [4] рассмотрена задача о синхронизации цепи стрелков.

7. Выводы

Из изложенного выше следует, что интеграция МОС в Switch-технологию не только возможна, но и целесообразна, так как позволяет эффективно реализовывать параллельные процессы. Использование сообщений позволяет сделать более наглядным взаимодействие автоматов.

Литература

1. *Шалыто А.А.*, *Туккель Н.И.* SWITCH-технология – автоматный подход к созданию программного обеспечения "реактивных" систем // Программирование. 2001. №5. http://is.ifmo.ru/ раздел “Статьи”.
2. *Дейкстра Э.* Взаимодействие последовательных процессов //Языки программирования. М.: Мир, 1972.
3. *Шалыто А.А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.
4. *Гуисов М.И.*, *Кузнецов А.Б.*, *Шалыто А.А.* Задача Д. Майхилла "Синхронизация цепи стрелков”. Вариант 2. http://is.ifmo.ru/ раздел “Проекты”.

Приложение. Библиотека «swmem»

Файл «swmem.h»

```
#ifndef _SWITCH_MEM_INCLUDED
#define _SWITCH_MEM_INCLUDED

#include <stdio.h>

#ifndef NULL
#define NULL 0
#endif

class CSMessage {
public:
	long int     id;                // Идентификатор сообщения (на графе переходов)
	long int     src;               // Указатель на источник сообщения (автомат)

	CSMessage(long int  id  = -1,    // Идентификатор сообщения на графе
	          long int  src = -1);   // Адрес объекта-источника сообщения
};

class CSQueue {                         // Очередь указателей на сообщения

	class CSQueueItem {             // Элемент очереди
	public:
		CSMessage    *msg;      // Экземпляр сообщения
		CSQueueItem *next;      // Следующий элемент очереди
		CSQueueItem(CSMessage *msg);    // Конструктор элемента очереди
	};

public:
	CSQueue();                      // Создание пустой очереди
	~CSQueue();                     // Уничтожение очереди
	bool IsEmpty();                 // Проверка на пустоту
	bool AddMessage(CSMessage *msg); // Добавление сообщения в очередь
	CSMessage *GetMessage();        // Извлечение сообщения из очереди
protected:
	CSQueueItem *first;             // Первый элемент очереди
	CSQueueItem *last;              // Последний элемент очереди
};

struct CSAutoState {
public:
	int number;
	char *caption;
};

class CSStateArray {
public:
	CSStateArray(int size,          // Размер массива
		int num_arr [],         // Вектор номеров состояний
		char *cap_arr []);      // Вектор названий состояний
	~CSStateArray();                // Деструктор
	char *Caption(int state);       // Возврат названия состояния по номеру
private:
	int size;                       // Количество состояний
	CSAutoState *data;              // Собственно массив
	int BinSearch(int state);       // Поиск названия состояния по номеру
};

class CSAutomate {
public:

	CSAutomate(CSQueue *queue,              // Указатель на очередь сообщений
		char *automate_id,              // Идентификатор автомата
		CSStateArray *st_arr = NULL,    // Массив идентификаторов состояний
		FILE *f_rec = NULL,       // Указатель на ОТКРЫТЫЙ файл для записи протокола
		bool recording = false);  // Начальное состояние протоколирования
	~CSAutomate();
```

```
	virtual void A(CSMessage &msg) = 0; // Выбор пути (сообщение передается по ссылке)
	virtual void S() = 0;		    // Осуществление перехода

	void StartRec();		    // Начать ведение протокола
	void PrintRec();		    // Вывод протокола в файл
	void StopRec();			    // Остановить ведение протокола
	void Update();			    // Обновление внутренних переменных автомата
	int  GetState();		    // Интерфейс переменной состояния

protected:
	char *AutomateID;		    // Идентификатор автомата
	CSQueue *Queue;			    // Указатель на очередь сообщений

	int y_o, y_n;			    // Переменные состояния в начале и в конце шага
	int priority;			    // Переменная, разрешающая механизм приоритетов
	int action;			    // Переменная выбора действия

	CSStateArray *st_arr;		    // Массив идентификаторов состояний
	FILE *f_rec;			    // Файл для вывода протокола
	bool recording;			    // Ведение протокола
};

#endif
```

Файл «swmem.cpp»

```
#include "swmem.h"

// Конструктор сообщения
CSMessage::CSMessage(long int id, long int src)
{
	this->id     = id;
	this->src    = src;
}

// Конструктор элемента очереди сообщений
CSQueue::CSQueueItem::CSQueueItem(CSMessage *msg)
{
	this->msg = msg;
	next = NULL;
}

// Создание пустой очереди
CSQueue::CSQueue()
{
	first = NULL;
	last  = NULL;
}

// Уничтожение очереди
CSQueue::~CSQueue()
{
	CSQueueItem *item;
	while(!IsEmpty()) {
		item = first;
		first = item->next;
		delete item;
	}
}

// Проверка на пустоту
bool CSQueue::IsEmpty()
{
	return (first == NULL);
}

// Добавление указателя сообщения в очередь
bool CSQueue::AddMessage(CSMessage *msg)
{
	CSQueueItem *item = new CSQueueItem(msg);
```

```
	if (IsEmpty()) {		// Если очередь пуста, создаем первый элемент
		first = last = item;
	} else {			// Иначе добавляем элемент в конец
		last->next = item;
		last = item;
	}
	return item == NULL;		// Если элемент не создался, сообщим об ошибке
}

// Извлечение указателя сообщения из очереди
CSMessage *CSQueue::GetMessage()
{
	if (IsEmpty()) return NULL;		// Из пустой очереди нечего взять

	CSQueueItem *item = first;
	CSMessage *msg = item->msg;

	if (!first->next) {			// Если один элемент, берем его
		first = last = NULL;
	} else {				// Если больше одного, изменяем указатель first
		first = first->next;
	}

	delete item;
	return msg;
}

CSStateArray::CSStateArray(int size, int num_arr[], char *cap_arr[])
{
	this->size = size;
	data = new CSAutoState [size];

	for (int i=0; i<size; i++) {
		data[i].number	= num_arr[i];
		data[i].caption	= cap_arr[i];
	}
}

CSStateArray::~CSStateArray()
{
	delete [] data;
}

char *CSStateArray::Caption(int state)
{
	return data[BinSearch(state)].caption;
}

// Поиск названия состояния по номеру
int CSStateArray::BinSearch(int key)
{
	int low = 0;
	int high = size - 1;
	int mid;

	while (low <= high) {
		mid = (low + high) / 2;
		if (key < data[mid].number) high = mid - 1;
		else if (key > data[mid].number) low = mid+1;
		else return mid;

	}

	return -1;
}

CSAutomate::CSAutomate(CSQueue *queue, char *automate_id,
				CSStateArray *st_arr, FILE *f_rec, bool recording)
{
	this->Queue		= queue;
	this->AutomateID	= automate_id;
	this->st_arr		= st_arr;
	this->f_rec		= f_rec;
```

```
	this->recording	= recording;
	y_n = 0; Update();
}


CSAutomate::~CSAutomate()
{
	delete st_arr;
}


// Начать ведение протокола
void CSAutomate::StartRec()
{
	recording = true;
}

// Вывод протокола в файл
void CSAutomate::PrintRec()
{
	if (!recording) return;

	char *s_old = st_arr->Caption(y_o);
	char *s_new = st_arr->Caption(y_n);

	fprintf(f_rec, "\n{ %s: %2.1d '%s' -> %2.1d '%s', с действием %2.1d }",
		AutomateID, y_o, s_old, y_n, s_new, action);
}

// Остановить ведение протокола
void CSAutomate::StopRec()
{
	recording = false;
}

// Обновление внутренних переменных автомата
void CSAutomate::Update()
{
	y_o = y_n;
	priority = -1;
	action = -1;
}

// Интерфейс переменной состояния
int CSAutomate::GetState()
{
	return y_o;
}
```