Ассоциация участников технологических кружков

Национальная киберфизическая платформа

Основы событийного программирования и программирования расширенных иерархических машин состояний (ПРИМС)

На основе Miro Samek, «A Crash Course in UML State Machines», Quantum Leaps, 2015,
https://www.state-machine.com/doc/AN_Crash_Course_in_UML_State_Machines.pdf

Астрахань – Москва – Уфа

Аннотация

Многие компьютерные системы управляются событиями, что означает, что они постоянно ожидают возникновения какого-либо внешнего или внутреннего события, такого как отсчет времени, поступление пакета данных, нажатие кнопки или щелчок мышью. К таким системам в первую очередь относятся так называемые **встраиваемые системы** – специализированные микропроцессорные системы управления, контроля и мониторинга, встроенные непосредственно в устройство, которым они управляют. После возникновения и распознавания события такие системы реагируют, выполняя соответствующие вычисления, которые могут включать управление устройствами или генерацию программных событий, которые запускают другие внутренние компоненты программного обеспечения, поэтому системы, управляемые событиями, также называются **реактивными системами**. Как только обработка событий завершена, система возвращается к ожиданию следующего события.

Подход к программированию систем, управляемых событиями, называется **событийно-ориентированным** или просто **событийным программированием**. Он применим во многих ситуациях, однако не очень хорошо работает, когда существует несколько возможных источников событий, время и порядок поступления которых невозможно предсказать, и где важно своевременно обрабатывать события. Проблема заключается в том, что, пока последовательная программа ожидает события одного типа, она не выполняет никакой другой работы и не реагирует на другие события.

Очевидно, что нужна программная структура, способная реагировать на множество возможных событий, любое из которых может произойти в непредсказуемое время и в непредсказуемой последовательности. Эта проблема очень распространена во встраиваемых системах, таких как бытовая техника, сотовые телефоны, промышленные контроллеры, медицинское оборудование и многих других, однако она также свойственна программам для настольных компьютеров, например, веб-браузерам, текстовым процессорам или электронным таблицам. Современный графический интерфейс пользователя этих программ способен обрабатывать множество событий. При разработке современных систем с графическим интерфейсом и многих встраиваемых приложений принято использовать общую структуру программы, которая решает проблему своевременной обработки многих **асинхронных событий**.

Программирование, управляемое событиями, требует совершенно иного подхода, чем разработка традиционных последовательных программ. Программа, управляемая событиями, не контролирует ситуацию в ожидании события, поскольку до возникновения события она даже не активна. Только когда событие приходит, программа вызывается для его обработки, а затем снова быстро возвращает управление. Такой механизм позволяет системе, управляемой событиями, одновременно ожидать множества событий, и поэтому система остается отзывчивой на все события, которые ей необходимо обработать.

Этот механизм имеет важные особенности. Во-первых, система, управляемая событиями, делится на приложение, которое фактически обрабатывает события (программа, управляемая событиями), и контролирующую подсистему, которая ожидает событий и отправляет их приложению. Во-вторых, программа, управляемая событиями, должна возвращать управление после обработки каждого события, поэтому контекст выполнения не может быть сохранен в переменных стека и программном счетчике, как в последовательной программе. Вместо этого программа, управляемая событиями, становится машиной состояний или набором взаимодействующих машин состояний, которые сохраняют контекст между обработкой событий в статических переменных.

Машины состояний – это самый известный способ описания систем, управляемых событиями, при этом наиболее современное состояние теории машин состояний представлено **расширенными иерархическими машинами состояний UML**.

В этой статье обсуждаются проблемы событийно-ориентированного программирования, и то, как могут помочь машины состояний при решении задач реального мира.

После знакомства с материалом статьи иерархические машины состояний и событийно-ориентированное программирование должны стать гораздо более эффективным способом решения задач управления, чем традиционные подходы, такие как глубоко вложенные структуры **if - else** для программной реализации поведения с сохранением состояния или передача событий через флаги традиционных **операционных систем реального времени** (RTOS).

Содержание

1. Основы машин состояний

В настоящей работе представлено краткое и доступное для начинающих введение в современную и перспективную технологию разработки систем управления. Данная технология основана на создании и применении расширенных иерархических машин состояний, использующих методы событийного программирования. Работа в значительной степени опирается на работы Miro Samek [1, 2], в том числе, на краткое руководство по машинам состояний UML – «A Crash Course in UML State Machines» [2], расширяя и дополняя его, а также адаптируя текст для восприятия начинающих. Читателю также будет интересен опыт отечественных специалистов, в течение последних лет занимающихся аналогичным подходом в программировании, практическим использованием его при создании систем управления. Машины состояний представляют собой разновидность конечных автоматов, математической концепции, нашедшей практическое применение в программировании. Одним из наиболее известных российских ученых, внесших большой вклад в развитие автоматного подхода в программировании, является А.А. Шалыто [3, 4, 5]. Его основной разработкой является SWITCH-технология, основанная на программной реализации граф-схем автоматов систем управления.

Занимаясь программированием, в традиционных последовательных программах практически невозможно встретить программный код, который выполняется построчно и не содержит ни условных операторов, ни циклов, ни обращений к процедурам и функциям. Очевидно, что причиной тому характер и сложность решаемых задач и реализуемых алгоритмов. Чем сложнее задача и сложнее алгоритм ее решения, тем больше упомянутых структур в программе. Зачастую наибольшую сложность как раз представляют условные операторы «если - то» (if - then), число которых неизменно растет не только по мере усложнения алгоритма, но и при добавлении проверок в попытке выявить и исключить ошибки, возникающие в ходе работы программы. Ситуация существенно усугубляется, если возникают вложенные условные операторы. Читать такой программный код, понимать и изменять его становится сложнее, особенно для человека, не являющегося его автором. И речь не только о «красоте» программного кода, когда блоки программы отделены друг от друга, а сам код написан с отступами, чтобы выделить логические структуры и их содержимое, что повышает его читаемость, улучшает восприятие. Речь идет о возможности по тексту программы понять, какой алгоритм в ней заложен и реализован в виде совокупности операторов, логических структур и вызовов процедур.

Заметим, что здесь мы будем рассматривать такие программы, которые а) выполняют действия в ответ на некоторые события (или бездействуют, находясь в ожидании таких событий), б) могут по-разному реагировать на события в зависимости от текущей ситуации, в) зачастую, ограничены во времени реакции на событие, г) могут допускать одновременное выполнение нескольких действий, д) должны реагировать на каждое событие, даже если они поступают (почти) одновременно, е) по возможности, не должны пропускать события, даже если заняты обработкой предыдущих. Это не обычные простые программы, которые позволяют получить результат в ответ на введенные данные и после этого завершают свою работу. Такими характеристиками обладают программы, которые заняты управлением какими-либо объектами. Зачастую такие программы используются в так называемых **встраиваемых системах** – системах управления, встроенных в управляемый объект (от кофеварки и микроволновой печи до самолета и марсохода). Такие программы могут работать бесконечно – до отключения системы управления, постоянно находясь в ожидании событий. Наиболее ярко это проявляется в робототехнических системах, а также в системах, моделирующих поведение роботов – например, в компьютерных играх.

Возникает вопрос, могут ли в принципе в системах управления использоваться традиционные программы, написанные в традиционном, последовательном стиле, с использованием вложенных конструкций «если - то» (if - then)? Конечно, могут, но разработка и поддержка программ с многократно вложенными конструкциями «если - то» резко усложняется при увеличении уровней вложенности (возникает так называемый «спагетти-код»), при этом сложность программы растет экспоненциально от числа событий и действий. Данный подход, называемый **императивным** (классический способ написания программ в виде последовательно выполняемых инструкций), возможен в системах управления, но не эффективен ввиду сложности понимания модели работы программы, трудности написания и особенно отладки и сопровождения написанного таким образом кода.

Для разработки таких управляющих программ придуманы другие способы программирования, главным образом с целью повышения наглядности, снижения сложности, избыточности кода, облегчения его разработки и отладки. И, поскольку речь идет, в конечном счете, о реакции на некоторые события, в данном случае существует концепция «событийно-ориентированного программирования» (event-driven programming), другими словами, программирования, управляемого событиями. Мы будем называть такой подход **событийным программированием**.

Концепция событийного программирования предполагает наличие в программе множества небольших, детализированных обработчиков событий – обособленных фрагментов программного кода, выполняющих действие (совокупность действий), ответное по отношению к событию, его инициирующему, т.е. вызываемых по мере возникновения событий. Фактически, программа делится на ограниченные, понятные фрагменты, которые можно рассматривать и анализировать по отдельности (в каждом фрагменте реализована реакция на отдельное событие). При этом выполнение программы в целом представляет собой переключение между вызовами отдельных обработчиков событий. При событийном программировании программа имеет более понятную и логичную структуру, однако механизм переключения между вызовами отдельных обработчиков событий также может включать глубоко вложенные условные операторы («если - то»).

Устранение хотя бы части ветвей условного оператора, снижение уровня вложенности, как в первом, так и во втором рассмотренных случаях, позволит существенно снизить количество путей выполнения программного кода, в значительной степени упростить понимание, тестирование и поддержку программы.

Методы решения данной проблемы существуют, и один из них – метод использования **машин состояний UML** [6, 7, 8] – рассматривается далее.

Основным подходом к разработке событийно-ориентированных программ является **модель и парадигма программирования «событие-действие»**, в которой события непосредственно привязаны к коду, который должен быть выполнен в ответ на событие [9]. Модель «событие-действие» является важной для понимания машин состояний, однако, обладая рядом преимуществ, не лишена и недостатков [10]. В частности, одно лишь событие само по себе не может полностью определить действие, которое должно быть выполнено в ответ на это событие. Большое значение для выбора требуемого действия будут иметь текущее состояние программы и значения переменных в ней. Иными словами, реакция на одно и то же событие может быть различной в различных ситуациях в программе. Именно из-за этого применение модели «событие-действие» в чистом виде зачастую сопряжено с написанием многоуровневых вложенных условных операторов, что также является существенным недостатком. По этим причинам модель «событие-действие» в чистом виде редко применяется на практике.

1.1. Основные концепции машин состояний

Поведение большинства событийно-ориентированных систем можно разделить на относительно небольшое количество блоков, при этом реакции на события в каждом отдельном блоке зависят только от *текущего* события, а не от последовательности *прошлых* событий. Здесь модель «событие-действие» реализуется в блоке и им определяется (ограничивается). В различных блоках могут быть различные реакции на одно и то же событие.

Распространенным и простым способом моделирования поведения на основе этой идеи является конечный автомат (машина с конечным числом состояний, **машина состояний**) (finite state machine, FSM). При этом «блоки поведения» называются **состояниями**, а смена поведения (изменение в ответ на какое-либо событие) соответствует изменению состояния и называется **переходом между состояниями**.

Машина состояний (МС) – это модель системы, у которой есть конечное количество состояний и переходов между этими состояниями, вызываемых событиями. Машина состояний – эффективный способ задать ограничения на общее поведение системы. Пребывание в состоянии означает, что система реагирует только на подмножество (часть) всех разрешенных событий, производит только подмножество (часть) возможных реакций и меняет состояние только на подмножество (часть) возможных состояний.

Концепция машины состояний так важна в программировании из-за того, что она делает обработку событий зависимой как от типа события, так и от контекста выполнения (от состояния) всей системы. При правильном использовании машина состояний позволяет в значительной степени сократить количество путей выполнения кода, упрощает условия, проверяемые в каждом разветвлении, и упрощает переходы между разными режимами выполнения.

1.2. Состояния

Состояние в общем смысле – понятие, обозначающее множество устойчивых значений переменных параметров объекта (в нашем случае, например, программы). Для нас важна способность состояния эффективно отражать историю системы.

Для примера можно рассмотреть обычную клавиатуру. Когда происходит нажатие клавиши на клавиатуре, генерируемый код символа будет кодом либо прописного, либо строчного символа, в зависимости от того, активен ли в настоящее время режим Caps Lock. Поэтому поведение клавиатуры можно разделить на два блока (состояния): состояние «по умолчанию» и состояние, когда был включен режим Caps Lock. Важно, что в этом случае поведение клавиатуры зависит только от того, нажата ли клавиша Caps Lock, а не от того, например, какие и в каком порядке были нажаты клавиши до этого. Состояние может абстрагироваться от всех возможных (но неактуальных) последовательностей событий и фиксировать только актуальные.

С точки зрения программирования такой подход выражается следующим образом: вместо записи истории событий в виде огромного количества переменных, флажков и запутанной логики, вводится переменная состояния, которая может принимать только ограниченное количество ранее установленных значений (например, в случае с клавиатурой, это 2 значения). Значение переменной состояния четко определяет текущее состояние системы в любой момент времени. Такой подход имеет два положительных аспекта:

- для определения текущего состояния системы необходимо проверить только значение переменной состояния вместо проверки множества переменных. Таким образом можно избавиться от большого числа операций (типа «если - то») по проверке множества условий, в том числе и глубоко вложенных;

- для переключения между состояниями необходимо изменить значение только переменной состояния вместо того, чтобы согласованно изменять значения множества переменных.

1.3. Диаграммы машин состояний

Машины состояний (конечные автоматы) графически можно представить в виде диаграмм состояний (не путать с блок-схемами).

Эти диаграммы представляют собой ориентированные графы, в которых вершины (узлы) обозначают состояния, а дуги (связи) – переходы между состояниями.

На рисунке 1 показана диаграмма состояний, соответствующая машине состояний компьютерной клавиатуры. Состояния представлены в виде прямоугольников с закругленными углами с подписанными названиями состояний. Переходы, представленные в виде стрелок, помечены событиями, запускающими эти переходы (переключающими событиями), за которыми после наклонной черты (/) может следовать список выполняемых действий. Начальный переход выходит из закрашенного кружка и определяет стартовое состояние системы при запуске. В каждой диаграмме машины состояний должен быть такой переход, который не должен быть помечен событием ввиду того, что он не вызывается событием. У начального перехода могут быть связанные с ним действия.

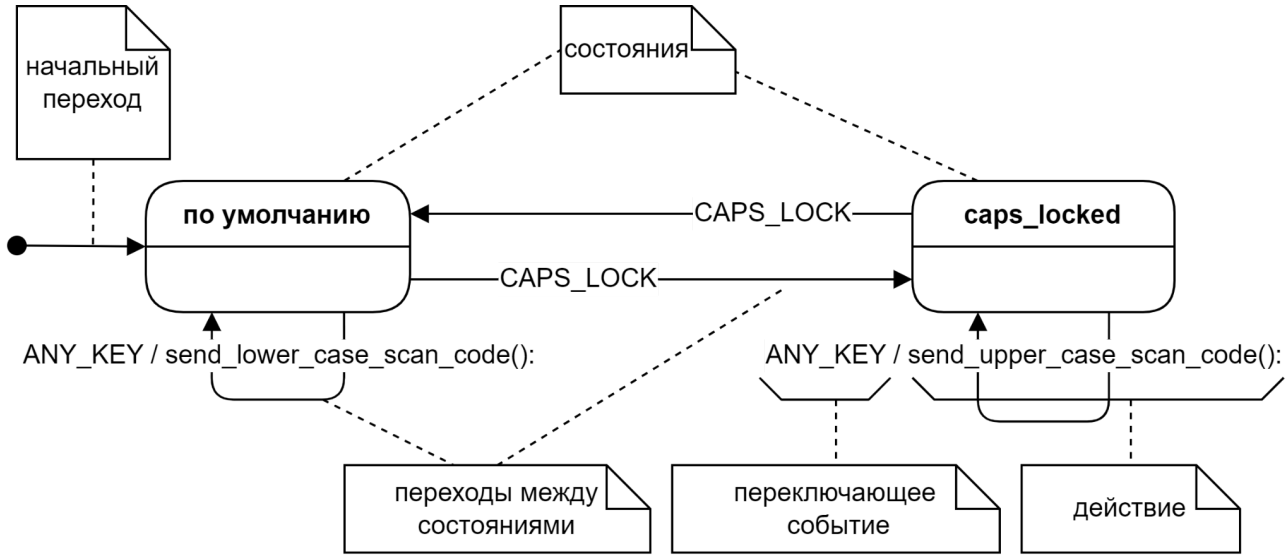


Рисунок 1. Диаграмма состояний для машины состояний компьютерной клавиатуры

1.4. Диаграммы машин состояний в сравнении с блок-схемами

При первом знакомстве с машинами состояний их визуальное представление (диаграммы состояний) можно спутать с блок-схемами.

На рисунке 2 показана разница между диаграммой машины состояний и блок-схемой. Машина состояний (а) выполняет действия в ответ на явные переключающие события. В отличие от нее, в блок-схеме (б) нет нужды в явных переключающих событиях, а переходы от узла к узлу происходят автоматически по завершении действий.

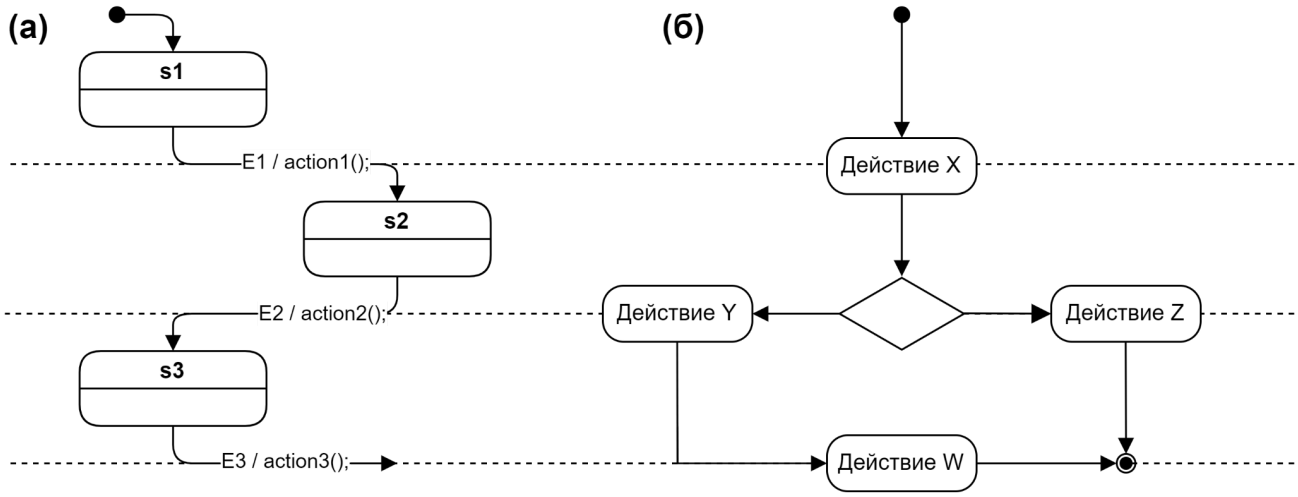


Рисунок 2. Сравнение диаграммы состояний (а) и блок-схемы (б)

По сравнению с диаграммами машин состояний, в блок-схемах роли узлов (вершин графа) и связей (дуг графа) противоположны по смыслу. На диаграмме машины состояний обработка информации связана с переходами («стрелками»), тогда как в блок-схеме она связана с узлами (операциями). В узлах (состояниях) в машине состояний ничего не происходит, пока не появится какое-либо событие. А в блок-схеме в узлах выполняются действия.

Из-за того, что блок-схема описывает последовательность выполнения задачи от начала до конца, ее можно сравнить с конвейером на производстве, где каждое следующее действие является очередным этапом на пути решения задачи. Машина состояний скорее больше похожа на лифт, для которого этаж, расположенный выше (или ниже) текущего этажа, не является следующим (или предыдущим) этапом решения задачи. Например, клавиатура в состоянии, когда был включен режим Caps Lock, по сравнению с состоянием «по умолчанию» не находится на более продвинутом этапе выполнения какой-то задачи, она просто по-разному реагирует на события. Состояние в машине состояний – это эффективный способ определения конкретного поведения, а не этапа обработки.

Разница между диаграммами состояний и блок-схемами очень важна, так как они представляют два противоположных подхода к программированию: событийно-ориентированное программирование (машина состояний) и процедурное программирование (блок-схема). Для разработки эффективной машины состояний нужно всегда учитывать возможные события. Напротив, в блок-схемах события играют второстепенную роль либо вообще не используются.

1.5. Расширенные машины состояний

Машины состояний окружают нас. Компьютер, смартфон, контроллер, калькулятор и т.д. представляют собой весьма сложные, но все же вполне поддающиеся нашему контролю автоматы. Количество состояний в таких автоматах огромно, а само поведение определяется не только структурой вычислителя (процессора), но и программой, управляющей вычислителем, а также данными, подаваемыми на вход. Самый известный в теории автомат – машина Тьюринга – «универсальная машина», способная выполнять функции любой другой машины, или, другими словами, вычислить все, что можно в принципе вычислить, стала основой для создания концепции современного компьютера. Машину Тьюринга с бесконечной лентой невозможно без ограничений реализовать на практике, однако стало

возможным создать два различных автомата, расширяемых с помощью аналогов «ленты» – запоминающих устройств. Эти два вида автомата – вычислительные машины с архитектурой фон Неймана и с гарвардской архитектурой. Как в случае теоретической машины Тьюринга, так в практических реализациях вычислительных машин, число возможных состояний управляющего устройства конечно и точно задано (это и есть обычный конечный автомат, или машина состояний). Однако хранимая в памяти последовательность инструкций (программа), а также находящиеся там же (фон-неймановская архитектура) или в другом запоминающем устройстве (гарвардская архитектура) данные (переменные) дополняют (расширяют) автомат и определяют его поведение и результаты его работы.

Простейший (и одновременно весьма неэффективный в практической реализации) подход к представлению подобных машин состояний в программах – ситуация, когда каждое состояние представляет собой один из возможных наборов допустимых значений всей используемой памяти. Напомним, что один байт – 8 бит информации – представляет $2^8 = 256$ различных состояний. Даже в простых программах с небольшим количеством элементарных переменных такой подход ведет к колоссальному количеству состояний. Например, даже одно 32-битное целое значение создает более чем 4 миллиарда различных состояний ($2^{32} = 4\ 294\ 967\ 296$). Такой подход непрактичен, поэтому **переменные программы принято отделять от состояний**. Если представить, что структура машины состояний задана программой, хранящейся в постоянной памяти (ПЗУ, ROM), в так называемой **памяти программ**, что естественно для встраиваемых систем (программируемых логических контроллеров, ПЛК), а **переменные** этой машины состояний находятся в оперативной памяти (ОЗУ, RAM), в так называемой **памяти данных**, то таким образом машина состояний наиболее естественным образом реализуется с помощью вычислительной системы с гарвардской архитектурой, в рамках которой построено большинство встраиваемых систем (например, ПЛК).

Состояния (качественное представление системы) и переменные расширенного состояния (количественное представление) объединяются в **расширенное состояние** системы. В таком случае изменение переменной не всегда означает качественное изменение поведения системы и поэтому не ведет к изменению состояния.

Машины состояний, дополненные переменными, называются **расширенными машинами состояний**. Расширенные машины состояний позволяют решать гораздо более сложные задачи, чем это практически возможно без использования переменных расширенного состояния. Предположим, что поведение клавиатуры зависит от количества введенных до этого знаков, и что после, скажем, 1000 знаков клавиатура ломается и переходит в свое конечное состояние (назовем ее «дешевой клавиатурой»). Чтобы смоделировать такое поведение в машине состояний без памяти, пришлось бы ввести 1000 состояний (например, нажатие клавиши в состоянии N приведет к переходу в состояние N+1 и т.д.), что является непрактичным решением. В качестве альтернативы можно разработать машину состояния с переменной счетчика нажатых клавиш. Счетчик инициализируется числом 1000 и будет уменьшаться при каждом нажатии клавиши без изменения состояния. Когда счетчик достигнет нуля, машина состояний перейдет в конечное состояние.

Машина состояний на рисунке 3 является примером расширенной машины состояний, в которой полное состояние системы (расширенное состояние) представляет собой сочетание качественного фактора – «состояния», и количественных факторов – переменных расширенного состояния (таких как счетчик key_count). В расширенных машинах состояний изменение переменной не обязательно ведет к качественному изменению поведения системы и поэтому не всегда ведет к изменению состояния.

Очевидное преимущество расширенных машин состояний – это их **гибкость**. Например, увеличение продолжительности жизни «дешевой клавиатуры» с 1000 знаков до

10000 никак не усложнит код расширенной машины состояний. Единственное, что нужно будет сделать – это изменить первоначальное значение счетчика key_count в начальном переходе.

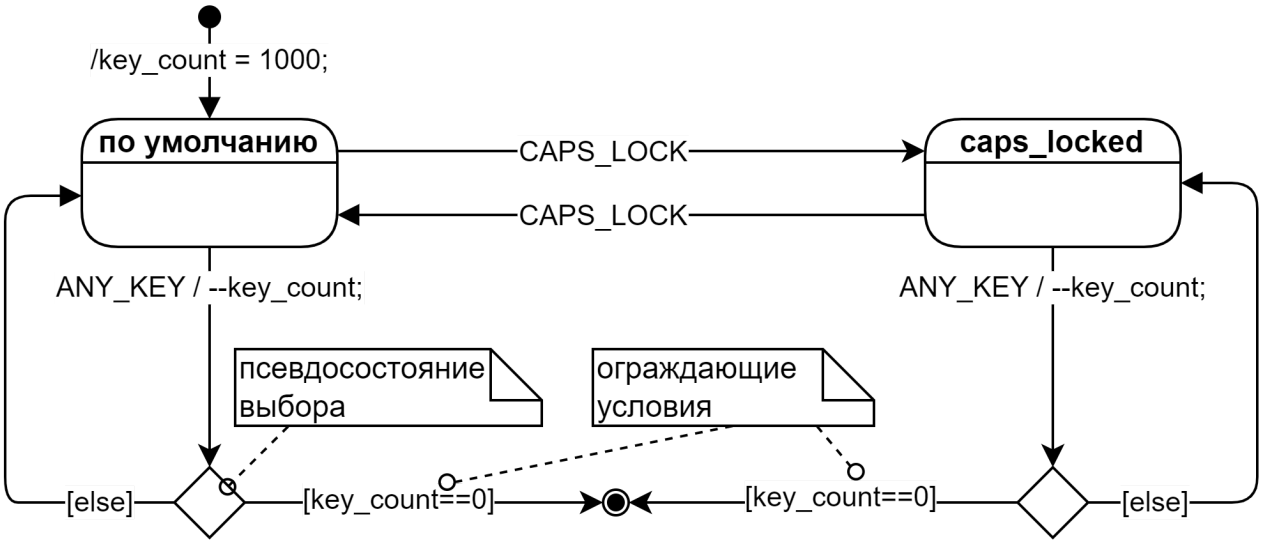


Рисунок 3. Расширенная машина состояний «дешевой клавиатуры» с переменной расширенного состояния key_count (счетчик нажатых клавиш) с различными ограждающими условиями

1.6. Ограждающие условия

Ограждающие условия – это логические выражения, вычисляемые на основе текущих значений переменных расширенного состояния и параметров событий. Ограждающие условия влияют на поведение машины состояний следующим образом: разрешают выполнение действий или переходы только тогда, когда их значение – истина (TRUE) и запрещают их, когда значение – ложь (FALSE). На диаграммах состояний ограждающие условия обозначаются квадратными скобками (например, [key_count==0]) (рис. 3).

Ограждающие условия нужны только в случае использования переменных расширенного состояния.

При экономном использовании переменные расширенного состояния и ограждающие условия представляют собой невероятно мощный механизм, который может значительно упростить код.

Важно отметить, что при разработке программного кода, реализующего расширенную машину состояний, ограждающие условия становятся такими же условными операторами «если - то» (if - then), как те, которые приводят обычно к чрезмерному усложнению кода.

Если таких условных операторов будет слишком много, то описанные выше преимущества машины состояний исчезнут, возникнет ситуация, когда ограждающие условия осуществляют обработку всех актуальных условий в системе – аналогично сложному для восприятия коду со множеством вложенных if. Чрезмерное использование переменных

расширенного состояния и ограждающих условий – это прямой путь к разрушению архитектуры кода, основанного на машинах состояний.

В связи с этим основная сложность при разработке расширенных машин состояний – это необходимость решить, какие части поведения системы следует реализовать качественно (как состояния), а какие элементы лучше оставить количественными (в виде переменных расширенного состояния). Необходимо активно искать возможности превратить историю событий («что произошло») в состояния системы вместо того, чтобы хранить информацию в переменных.

1.7. События

Событие – это мгновенное явление, произошедшее во времени и пространстве, и имеющее значение для системы. Понятие «событие» применительно к машинам состояний относится к типу события, а не к конкретному экземпляру этого события. Например, «нажатие клавиши» – это событие для клавиатуры, но каждое отдельное нажатие клавиши – это не событие, а конкретный экземпляр этого события.

У события могут быть связанные с ним параметры, позволяющие экземпляру события передать не только факт наступления какого-либо значимого события, но и количественную информацию о нем. Например, у события «нажатие клавиши», возникающего при нажатии на клавишу на клавиатуре, есть связанные параметры, которые содержат в себе информацию о скан-коде символа, а также статус клавиш Shift, Ctrl и Alt (Option, Command, Fn и т.п.).

Экземпляр события живет дольше, чем мгновенное явление, которое его породило, и может передать информацию об этом явлении одной или более машин состояний. После создания экземпляр события проходит через цикл обработки, который может состоять из трех этапов. Сначала экземпляр принимается и ожидает обработки (например, помещается в очередь событий). Затем он отправляется в машину состояний, и в этот момент он становится текущим событием. И, наконец, когда машина состояния завершает обработку экземпляра события, он поглощается. Поглощенный экземпляр события больше нельзя обработать.

1.8. Действия и переходы

Когда экземпляр события отправлен, машина состояния отвечает на него, выполняя **действия**, например: изменить переменную, выполнить ввод/вывод, вызвать функцию, сгенерировать другой экземпляр события или изменить состояние. Любые значения параметров, связанные с текущим событием, доступны для всех действий, непосредственно вызванных данным событием.

Переключение от одного состояния к другому называется **переходом между состояниями**, а событие, которое его вызывает, называется **переключающим событием** (событием-триггером, или просто триггером). В примере с клавиатурой, при нажатии клавиши Caps Lock состояние клавиатуры «по умолчанию» переходит в состояние «Нажата Caps Lock». Однако, если клавиатура уже находится в состоянии «Нажата Caps Lock», нажатие клавиши Caps Lock вызовет другой переход: из состояния «Нажата Caps Lock» в состояние «по умолчанию». В обоих случаях нажатие клавиши Caps Lock является переключающим событием (событием-триггером).

В расширенных машинах состояний у переходов могут быть ограждающие условия, при этом переход «срабатывает», только если значение ограждающего условия – истина (TRUE).

У состояния допустимо наличие нескольких переходов в ответ на одно и то же переключающее событие, но при этом у таких переходов должны быть непересекающиеся ограждающие условия (например, можно использовать конструкцию «если-иначе», if - else,

на рис. 3 реализованную с помощью **псевдосостояния** выбора). При срабатывании общего события такая ситуация может привести к проблемам при проверке ограждающих условий в определенном порядке. Выход из этой ситуации – проектировать ограждающие условия так, чтобы порядок их проверки не имел значения. Практически, это значит, что у ограждающих условий не должно быть побочных эффектов (действий), по крайней мере таких, которые могли бы повлиять на вычисления истинности других ограждающих условий, разделяющих между собой одно переключающее событие.

1.9. Модель «выполнение до завершения» (RTC)

Диаграммы состояний предполагают, что машина состояний завершает обработку каждого события, перед тем как начать обрабатывать следующее. Такая модель выполнения называется «**выполнение до завершения**» или **RTC** (run to completion).

В такой модели система обрабатывает события в рамках неделимых RTC-этапов. Новые входящие события не могут прерывать обработку текущего события и должны храниться (обычно, в очереди событий) до тех пор, пока машина состояний не освободится. Благодаря такому подходу в рамках одной машины состояний можно избежать любых внутренних проблем, связанных с параллельной обработкой. Модель RTC также позволяет обойти проблему обработки действий, связанных с переходами, когда машина состояний не находится в четко определенном состоянии (находится между двумя состояниями) в течение периода выполнения действия. Во время обработки события система ни на что не реагирует, поэтому нечетко определенное состояние в это время не имеет значения.

Ключевым преимуществом модели «выполнение до завершения» является ее простота. Но ее главный недостаток заключается в том, что **скорость реакции** машины состояний на события (это очень важная характеристика любой системы управления, работающей в реальном времени, встраиваемой системы) определяется ее самым длинным (долгим) RTC-этапом. Если в **системе реального времени** нужно достичь высокой **реактивности** (кратчайшего времени реакции), потребуется реализовать как можно более короткие этапы RTC, что зачастую приводит к значительному усложнению машины состояний.

2. Расширенные иерархические машины состояний

Традиционные машины состояний, описанные в главе 1, подходят для решения небольших задач, однако из-за явления, известного как «**комбинаторный взрыв**»[1], сложность машины состояний растет намного быстрее, чем сложность системы, которую она описывает.

Под комбинаторным взрывом понимается экспоненциальный рост количества вычислительных операций, вариантов (состояний) или требуемых для решения задачи ресурсов при линейном повышении ее размерности. Например, время, необходимое для поиска выхода из лабиринта, экспоненциально возрастает с увеличением числа разветвлений. В аналитических технологиях комбинаторные взрывы типичны для алгоритмов, использующих методы перебора. Примером является поиск ассоциативных правил, где количество рассматриваемых комбинаций экспоненциально растет с увеличением числа предметов [11]. Примерами комбинаторного взрыва являются, например, функция Аккермана или задача коммивояжера.

В данном случае увеличение сложности машины состояний связано с увеличением числа взаимодействий между ее элементами, например, с увеличением числа *комбинаций* состояний и переходов между ними, или увеличением числа сочетаний одновременно

[1]**Комбинаторный взрыв** - термин, используемый для описания эффекта резкого («взрывного») роста сложности алгоритма при увеличении размера входных данных задачи

активных состояний, что в комбинаторике определяется *перекрестным произведением* множеств элементов машины состояний. Мощность такого множества пропорциональна не первой, а второй степени числа элементов исходных множество, например, при 2 состояниях и 2 переходах возможны 4 комбинации, при 4 состояниях и 4 переходах комбинаций уже 16, и т.д. **Диаграмма машины состояний** всегда представляет собой *ориентированный граф* с вершинами – состояниями и дугами – переходами, для полного ориентированного графа число дуг определяется как $n \cdot (n - 1)$, где n – число вершин. Таким образом, в общем случае сложность традиционной машины состояний определяется как $O(n^2)$, где n – число ее элементов.

Комбинаторный взрыв при построении традиционных машин состояний связан с *повторениями* в ситуации, когда множество событий во многих состояниях обрабатывается одинаково. Традиционная машина состояний не позволяет сделать обобщение и требует частого повторения одних и тех же действий и переходов. В традиционных машинах состояний не хватает способа описания общего поведения, чтобы впоследствии применять его сразу к нескольким состояниям.

Во второй главе описывается способ решения данной проблемы с помощью иерархических машин состояний, которые обеспечивают эффективный способ разделения (повторного использования) поведения [12]. При этом сложность диаграммы машины состояний не увеличивается резко, а остается сравнимой со сложностью системы, которую она описывает. Этот подход эффективен при разработке любых событийно-ориентированных систем (в частности, встраиваемых систем), поскольку он делает использование машин состояний действительно применимым к реальным задачам практически любой сложности.

2.1. Повторное использование поведения в машинах состояний

Повторное использование поведения – мощный механизм, который часто используется в системах, устроенных наподобие матрешки (иерархически). Мы постоянно используем этот механизм, например, когда работаем с графическими интерфейсами пользователя в любой операционной системе. Рассмотрим ситуацию – в операционной системе запущена программа, которая отображает окно, пусть пользователь нажал кнопку на клавиатуре, тогда система графического интерфейса, зафиксировав событие нажатия, сначала отправляет событие в программу. Возможно, что программа не предусматривает обработки данного события, тогда событие возвращается обратно в систему графического интерфейса и обрабатывается там, обычно, в соответствии с общими системными требованиями. Получается, что, во-первых, программе, находящейся ниже по иерархии (вложенной) дается возможность первой отреагировать на событие (возможно, каким-то специфическим образом), но, если она этого не делает, события отправляются на уровень выше; и, во-вторых, события обрабатываются некоторым общим способом даже в том случае, если реакция на них не предусмотрена во вложенной системе (программе). То есть все вложенные в систему графического интерфейса программы имеют возможность повторного использования некоторого поведения, которое определено в системе (например, это может быть изменение размеров окна или обновление его содержимое), и, поскольку такие возможности нужны всем окнам, и они стандартны, легко представить, какой объем работ программиста при этом экономится. При этом программисту при необходимости создания какой-либо специфической обработки в окне достаточно лишь реализовать отличия в поведении от стандартного, а в остальном повторно использовать предопределенное в системе верхнего уровня поведение. Можно сказать, что все дочерние программы (окна) в системе разделяют одно и то же стандартное системное поведение в части, касающейся управлением внешним видом графического интерфейса.

2.1.1. Иерархически вложенные состояния

Механизм повторного использования поведения может быть объединен с механизмом машины состояний при использовании вложенности. При этом возникает мощный инструмент – **иерархические машины состояний** (Hierarchical State Machines, HSM).

Смысл вложенности состояний заключается в следующем (см. рис. 4 (а)): если система находится во вложенном состоянии «s11» (т.н. **подсостояние** или **дочернее состояние**), она в то же время (косвенно) находится в содержащем его состоянии «s1» (т.н. **суперсостояние** или **родительское состояние**).

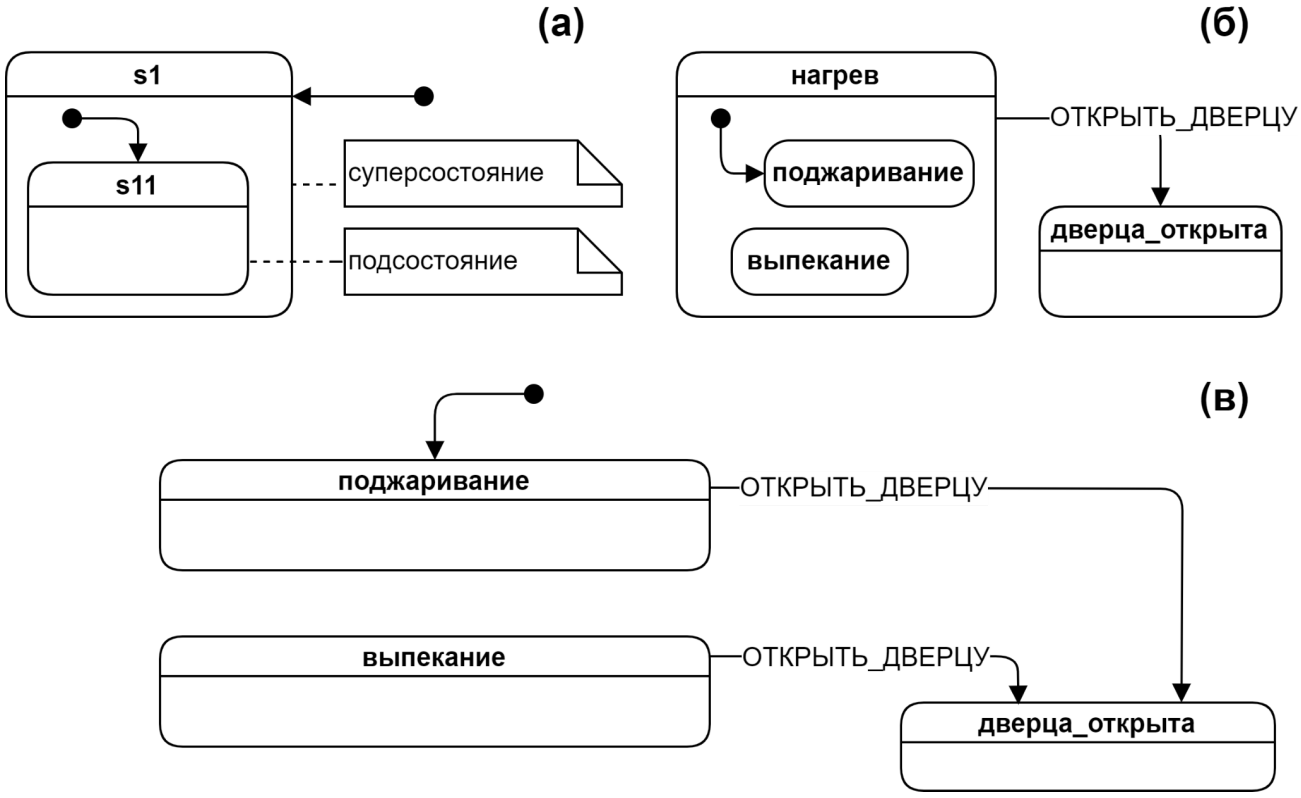


Рисунок 4. Диаграмма иерархически вложенных состояний (а) и диаграмма состояний печи (б), в которой состояния «поджаривание» и «выпекание» имеют общий переход от состояния «нагрев» к состоянию «дверца_открыта», что в традиционной («плоской») машине состояний (в) представлено двумя отдельными переходами

Аналогично рассмотренной выше системе графического интерфейса, такая машина состояний (рис. 4 (а)) сначала отправляет любое событие состоянию «s11», которое находится на более низком уровне иерархии. Однако, если в состоянии «s11» не прописано, как обрабатывать событие, оно не отбрасывается, как в традиционной «плоской» машине состояний, а автоматически отправляется на обработку в более высокое по иерархии суперсостояние «s1». Это и означает, что система одновременно находится и в состоянии «s11», и в состоянии «s1». Конечно, вложенность состояний не ограничивается только одним уровнем, и правило обработки событий применяется к любому уровню вложенности.

Состояния, которые содержат другие состояния, называются **составными состояниями**, и наоборот, состояния без внутренней структуры называются **простыми состояниями**. Когда вложенное состояние не содержится ни в каком другом состоянии, оно

называется **прямым подсостоянием**, в ином случае оно называется **транзитивно вложенным подсостоянием**.

Поскольку внутренняя структура составного состояния может быть сколь угодно сложной, любую иерархическую систему состояний можно рассматривать как внутреннюю структуру некоторого (более высокого по иерархии) составного состояния. Одно составное состояние удобно определять как корень иерархии машины состояний. Для каждой машины состояний можно определить **верхнеуровневое состояние** (абстрактный корень иерархии машины состояний), которое содержит в себе все остальные элементы всей машины состояний. Верхнеуровневое состояние не обязательно изображать графически.

Иерархическая декомпозиция состояний способствует **наследованию поведения** аналогично тому, как это происходит в оконных графических интерфейсах. Подсостояния (вложенные состояния) должны определять лишь отличия от суперсостояний (родительских состояний). Подсостояние может наследовать общее поведение своего суперсостояния(ий), просто игнорируя совместно обрабатываемые события, которые затем автоматически отправляются на обработку состояниям более высокого уровня. Таким образом, подсостояния могут разделять со своими суперсостояниями все особенности поведения. Например, в модели машины состояния печи, показанной на рисунке 4 (б), состояния **«поджаривание»** (тоста) и **«выпекание»** (хлеба) имеют общий переход **«ОТКРЫТЬ_ДВЕРЦУ»** к состоянию **«дверца_открыта»**, определенный в их общем суперсостоянии **«нагрев»**.

Мощный прием, который позволяет снизить сложность системы – это абстрагирование, состоящее в отвлечении от несущественных сторон, свойств, связей объекта с целью выделения его наиболее существенных признаков. Вместо того, чтобы работать со всеми частями сложной системы одновременно, зачастую можно абстрагироваться от некоторых ее частей (игнорировать их). С помощью иерархии состояний можно легко скрыть внутренние детали, так как программист может легко увеличить или уменьшить масштаб, чтобы скрыть или показать вложенные состояния. Хотя абстрагирование само по себе не снижает общую сложность системы, оно ценно тем, что позволяет иметь дело с меньшим количеством элементов одновременно [13].

Абстракция не позволяет полностью избавиться от сложности кода, спрятав ее внутри составных состояний. Однако составные состояния активно сокращают сложность благодаря мощному механизму повторного использования поведения. Без такого механизма даже небольшое усложнение системы часто приводит к взрывному росту количества состояний и переходов.

Например, если переделать диаграмму машины состояний из рисунка 4 (б) в классическую «плоскую» машину состояний (рисунок 4 (в)), то один и тот же переход (из состояния **«нагрев»** в состояние **«дверца_открыта»**) придется повторить в двух местах: как переход из состояния **«поджаривание»** в состояние **«дверца_открыта»** и как переход из состояния **«выпекание»** в состояние **«дверца открыта»**. Возможность избегать таких повторений позволяет иерархическим машинам состояний расти (усложняться) пропорционально росту сложности самой системы («плоская» машина состояний растет гораздо быстрее). По мере роста моделируемой системы увеличивается и количество возможностей повторного использования поведения. Это препятствует взрывному росту количества состояний и переходов, характерных для традиционных («плоских») машин состояний.

2.1.2. Наследование поведения

Иерархии состояний представляют из себя определенные группировки вложенных машин состояний, и их очевидное преимущество – значительное упрощение диаграммы,

когда набор событий применяется к нескольким подсостояниям. Однако истинный смысл и значение механизма иерархической вложенности состояний намного глубже. Экономия состояний и переходов вполне реальна и приводит к упрощению диаграмм состояний, но упрощенные диаграммы – это лишь побочный эффект повторного использования поведения за счет использования вложенности состояний.

Фундаментальный характер вложенности состояний обусловлен сочетанием абстракции и иерархии, что является традиционным подходом к уменьшению сложности, известном в программировании как **наследование**.

В объектно-ориентированном программировании концепция наследования классов описывает отношения между классами объектов. Наследование классов описывает отношение типа «IS-A …» (есть, является) между классами. Например, класс Bird («Птица») может быть производным от класса Animal («Животное») (подклассом, классом-потомком). Если объект – птица (экземпляр класса Bird), он автоматически становится животным, так как все действия, применимые к животным (например: питаться, жить, размножаться), также применимы и к птицам. Но у птиц есть свои особенности, так как у них есть действия, которые нельзя применить ко всем животным. Например, действие «летать» применимо к птицам, но не к рыбам.

Наследование помогает быстро определить новый класс на основе старого, повторно используя функциональность родительских классов. Это позволяет определять новые классы с помощью отличий от родительских, а не создавать их каждый раз с нуля. Это дает возможность получать новые реализации классов практически без усилий, наследуя большую часть того, что является общим, от классов-предков [14].

Все основные характеристики наследования классов применяются и ко вложенным состояниям (при замене слова «класс» на «состояние»). Вложенность состояний основана на той же фундаментальной классификации IS-A, что и объектно-ориентированное наследование классов.

Например, в модели состояний печи состояние **«поджаривание»** вложено в состояние **«нагрев»**. Если печь находится в состоянии **«поджаривание»**, она одновременно находится и в состоянии **«нагрев»**, так как поведение, относящееся к состоянию **«нагрев»**, применимо и к состоянию **«поджаривание»** (например, должен быть включен нагревательный элемент). Но состояние **«поджаривание»** более специализировано, потому что поведение в этом состоянии не может быть применимо к состоянию **«нагрев»** в целом. Например, настройка степени прожарки тоста (светлый или темный) применяется к состоянию **«поджаривание»**, а не к **«выпеканию»**.

В случае вложенных состояний, отношение IS-A (is-a-kind-of, один класс входит в другой класс) нужно заменить на отношение IS-IN (is-in-a-state, одно состояние находится в другом состоянии). В остальном, это одна и та же фундаментальная классификация. Вложенность состояний позволяет подсостояниям наследовать поведение у своих предков (суперсостояний), поэтому это называется наследованием поведения. Концепция наследования является основополагающей в разработке программного обеспечения. Наследование классов необходимо, чтобы лучше организовать программное обеспечение, а также чтобы повторно использовать код, что делает его основой объектно-ориентированного программирования. Точно так же наследование поведения необходимо для эффективного использования иерархических машин состояний и для повторного использования поведения, что делает его основой событийно-ориентированного программирования.

2.2. Ортогональные области

В рассмотренной ранее иерархической машине состояний печи (рисунок 4 (б)), если система находится в суперсостоянии **«нагрев»**, это значит, что она ИЛИ в состоянии **«поджаривание»**, ИЛИ в состоянии **«выпекание»**. Именно поэтому суперсостояние **«нагрев»** называется **состоянием-ИЛИ.**

Машины состояний также могут иметь дополнительную **декомпозицию-И.** При такой декомпозиции составное состояние может содержать две или более **ортогональные области** (в данном случае, ортогональная означает «независимая»), и пребывание в таком составном состоянии влечет за собой пребывание во всех его ортогональных областях одновременно [15]. В каждой такой области может находиться отдельное состояние или целая машина состояний.

Ортогональные области решают часто возникающую проблему комбинаторного увеличения количества состояний, когда поведение системы делится на независимые, одновременно активные части. Например, помимо основной клавиатуры, клавиатура компьютера имеет независимую цифровую клавиатуру (numpad). В прошлом разделе было описано, что у основной клавиатуры может быть два состояния: «по умолчанию» и «caps_locked» (Рисунок 1).

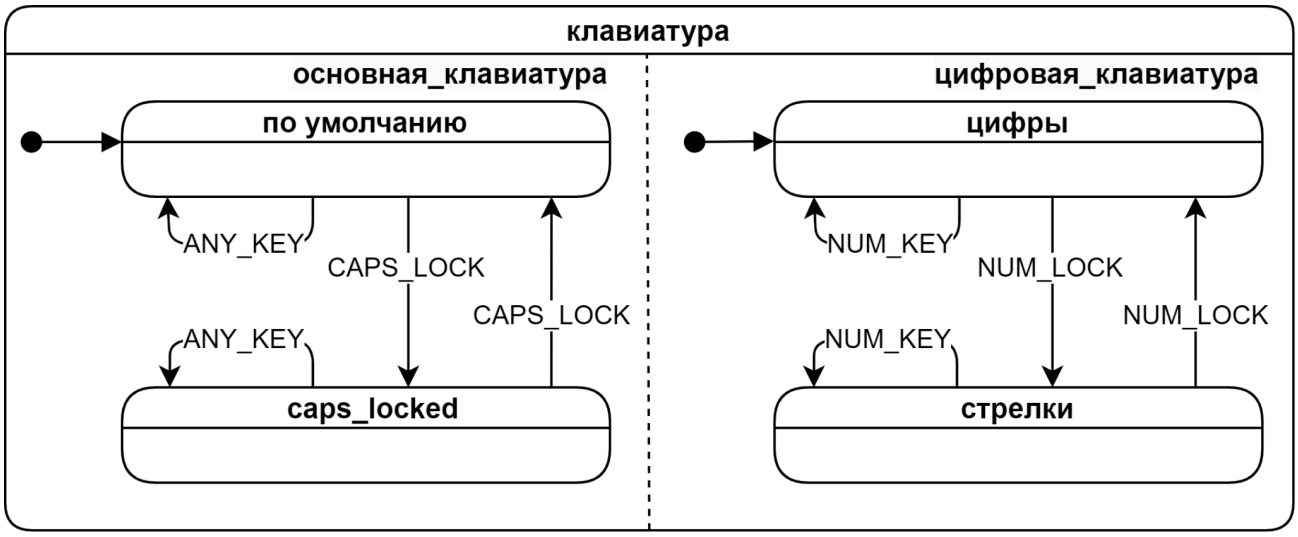


Рисунок 5. Две ортогональные области машины состояний клавиатуры компьютера (основной и цифровой клавиатуры)

Цифровая клавиатура тоже может быть в двух состояниях: «цифры» и «стрелки», в зависимости от того, активен ли Num Lock. Полное пространство состояний компьютерной клавиатуры в стандартной декомпозиции – это *перекрестное произведение* множеств состояний двух компонентов (основной и цифровой клавиатур) и включает в себя 4 состояния: «по умолчанию-цифры», «по умолчанию-стрелки», «caps_locked-цифры» и «caps_locked-стрелки». Однако, такое представление неестественно, так как поведение цифровой клавиатуры не зависит от состояния основной и наоборот. Ортогональные области позволяют избежать смешивания независимых поведений в виде их перекрестного произведения и, вместо этого, сохранить их отдельными, как показано на рисунке 5.

Если ортогональные области полностью независимы друг от друга, то их общая сложность всего лишь *аддитивна*, что означает, что количество независимых состояний, необходимых для моделирования системы, сводится к *сумме* k+l+m+…, где k, l, m, … обозначают количество **состояний-ИЛИ** в каждой ортогональной области. Зависимость же

поведений друг от друга приводит к *умножению* числа состояний, то есть количество необходимых состояний равно произведению k*l*m*…

Однако в большинстве реальных ситуаций ортогональные области лишь приблизительно ортогональны (т.е. не являются полностью независимыми). Поэтому диаграммы машин состояний предоставляют ряд способов для связи ортогональных областей и синхронизации их поведений. Из всего набора (порой сложных) механизмов, возможно, наиболее важной является возможность координировать поведение ортогональных областей путем отправки экземпляров событий от одной области к другой.

Несмотря на то, что ортогональные области подразумевают независимость выполнения (т.е. своего рода параллелизм), при программной реализации машины состояний обычно не требуется, чтобы для каждой ортогональной области был выделен отдельный поток выполнения (хотя это можно реализовать и таким образом). На самом деле, чаще всего ортогональные области выполняются в рамках одного потока. Требуется только не полагаться на какой-либо конкретный (специфический) порядок, в котором экземпляр события будет отправляться задействованным ортогональным областям (потому что этот порядок обычно заранее не известен и не гарантирован, если не реализован истинный параллелизм, и фактически ортогональные области выполняются последовательно).

2.3. Входные и выходные действия

В каждой диаграмме машины состояний могут быть необязательные входные действия, которые выполняются при входе в состояние, а также необязательные выходные действия, которые выполняются при выходе из состояния. Входные и выходные действия связаны с состояниями, а не с переходами. Все входные и выходные действия состояния будут выполняться, независимо от того, как именно произошел выход или вход в состояние. Входные и выходные действия обозначаются словом **«вход»** и **«выход»**, за которым следует косая черта и список действий. Они указываются внутри состояния сразу под его названием. (см. рисунок 6).

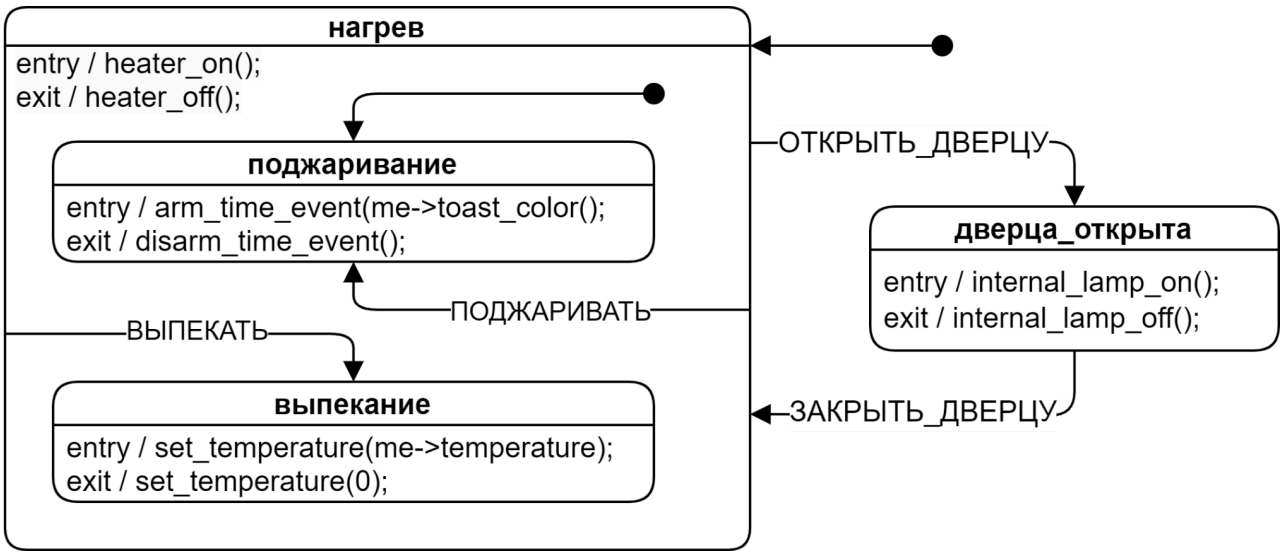


Рисунок 6. Машина состояний печи с входными и выходными действиями.

Входные и выходные действия – это способ гарантированно выполнить начальную установку (например, инициализацию переменных) и завершающую обработку (например,

очистку памяти). Этим они напоминают конструкторы и деструкторы классов в объектно-ориентированном программировании. Например, рассмотрим состояние **«дверца_открыта»** (рисунок 4 (б)); оно соответствует поведению печи при открытой дверце. У этого состояния есть одно очень важное требование по безопасности: нагревательный элемент всегда (гарантированно) должен быть выключен, когда открыта дверца. Кроме того, при открытой дверце должна гореть внутренняя лампа печи.

Такое поведение можно смоделировать, добавив соответствующие действия (выключение нагревательного элемента и включение лампы) к каждому переходу, ведущему к состоянию **«дверца_открыта»** (пользователь может открыть дверцу когда угодно во время поджаривания и выпекания, или когда печь вообще не используется). Кроме того, необходимо гасить внутреннюю лампу печи вместе с каждым переходом из состояния **«дверца_открыта»**. Такой подход может привести к повторению действий во многих переходах. Это также может привести к ошибкам в связи с последующими изменениями в машине состояний (например, другой программист, работающий над новой функцией, такой как подрумянивание сверху, может просто забыть выключить нагревательный элемент при переходе в состояние **«дверца_открыта»**).

Входные и выходные действия позволяют реализовать желаемое поведение гораздо более безопасным, простым и интуитивно понятным способом. Как показано на рисунке 6, можно указать, что выходное действие состояния **«нагрев»** должно выключать нагревательный элемент, входное действие состояния **«дверца_открыта»** должно включать лампу печи, а выходное действие состояния **«дверца_открыта»** должно гасить эту лампу. Использование входных и выходных действий работает лучше, чем размещение действий на переходах, так как позволяет избежать повторения этих действий на многих переходах и устраняет угрозу безопасности, связанную с включенным нагревательным элементом при открытой дверце. Логика выходных действий гарантирует, что независимо от пути перехода, нагревательный элемент будет выключен всегда, когда печь не находится в состоянии **«нагрев»**.

Поскольку входные действия выполняются автоматически при входе в соответствующее состояние, они часто определяют характеристики состояния, что схоже с тем, как конструктор класса определяет характеристики конструируемого объекта. Например, характеристика состояния **«нагрев»** определяется тем, что включен нагревательный элемент. Это действие (включение нагревательного элемента) должно быть выполнено до входа в любое подсостояние состояния **«нагрев»**, так как входные действия подсостояний, например, **«поджаривание»** или **«выпекание»**, опираются на то, что нагревательный элемент включен, и могут включать лишь дополнения к входному действию суперсостояния **«нагрев»**.

Следовательно, порядок выполнения входных действий должен идти от самого внешнего состояния к самому внутреннему. Неудивительно, что этот порядок аналогичен порядку, в котором вызываются конструкторы классов. Создание класса всегда начинается с корня иерархии (базового класса) и проходит через все уровни наследования вплоть до места, где создается объект/экземпляр.

Выполнение выходных действий происходит в обратном порядке, начиная с внутреннего состояния. Это похоже на порядок вызова деструкторов классов, начиная с самого дальнего класса-потомка.

2.4. Внутренние переходы

Часто событие вызывает выполнение только некоторых действий, но не приводит к изменению состояний (переходу между состояниями). В таком случае будем считать, что выполняемые действия являются частью внутреннего перехода. Например, когда вы

набираете текст на клавиатуре, она реагирует, генерируя различные коды символов. Однако, пока вы не нажмете клавишу Caps Lock, состояние клавиатуры не изменится (не произойдет перехода между состояниями). Такая ситуация моделируется с помощью внутренних переходов, как показано на рисунке 7.

Описание внутренних переходов похоже на описание входных и выходных действий, за исключением того, что вместо слов «вход» («entry») или «выход» («exit») внутренний переход обозначается переключающим событием (например, внутренний переход, вызванный событием ANY_KEY (нажатие любой клавиши) (см. рисунок 7)).

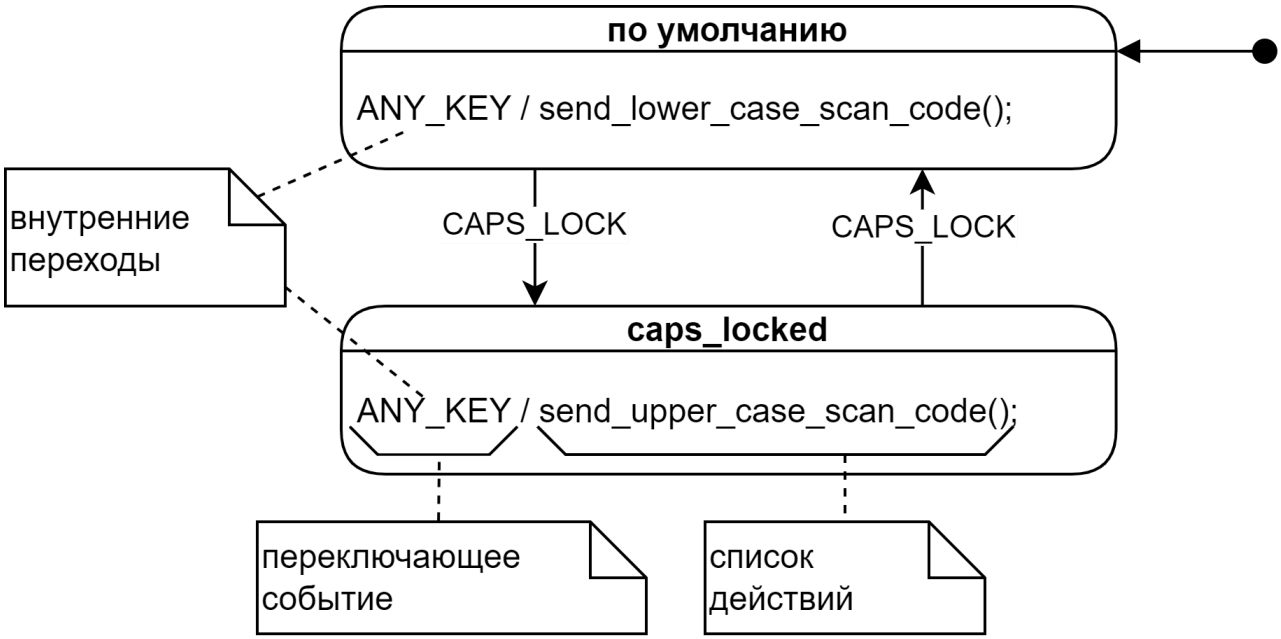


Рисунок 7. Диаграмма машины состояний клавиатуры с внутренними переходами

При отсутствии входных или выходных действий, описанных в состоянии, внутренние переходы (internal transitions) идентичны переходам в себя (self-transitions) (переходам, в которых целевое состояние (target state) совпадает с исходным (source state)). Переход в себя считается изменением состояния. При его запуске осуществляется выход из исходного состояния и вход в целевое, (то есть в него же) – а значит, запускаются действия при входе и выходе. Примеры переходов в себя приведены на рисунке 1. Переход в себя отличается от внутреннего перехода, который не приводит к изменению состояния.

В классической машине состояний действия связаны исключительно с переходами между состояниями, так что единственный способ выполнить действие, не меняя при этом состояния, это использовать переходы в себя (как показано на рисунке 1 в случае перехода по событию ANY_KEY). Однако, когда в состоянии есть входные или выходные действия, переходы в себя вызывают выполнение этих действий, и поэтому заметно отличаются от внутренних переходов.

При внутреннем переходе, в отличие от перехода в себя, входные и выходные действия никогда не выполняются, даже если внутренний переход унаследован из более высокого уровня иерархии. Внутренние переходы, унаследованные от суперсостояний любого уровня вложенности, работают так же, как если бы они были заданы непосредственно в текущем состоянии.

2.5. Последовательность выполнения переходов

Вложенные состояния в сочетании с входными и выходными действиями значительно усложняют понимание переходов между состояниями в иерархических машинах состояний по сравнению с традиционными «плоскими» машинами состояний.

В ситуации с иерархически вложенными состояниями и ортогональными областями, трудно определить, что такое «текущее состояние». В иерархических машинах состояний активными могут быть одновременно несколько состояний. Если машина состояний в текущий момент находится в простом состоянии, которое содержится в составном состоянии (которое, возможно, содержится в составном состоянии более высокого уровня и так далее), то все составные состояния, которые непосредственно или транзитивно содержат в себе данное активное простое состояние, также активны. Более того, из-за того, что у некоторых составных состояний в такой иерархии могут быть ортогональные области, текущее активное состояние фактически представлено деревом состояний, в котором одно верхнеуровневое состояние представлено корнем дерева, а отдельные простые состояния – его листьями. Такое дерево состояний называется конфигурацией состояний.

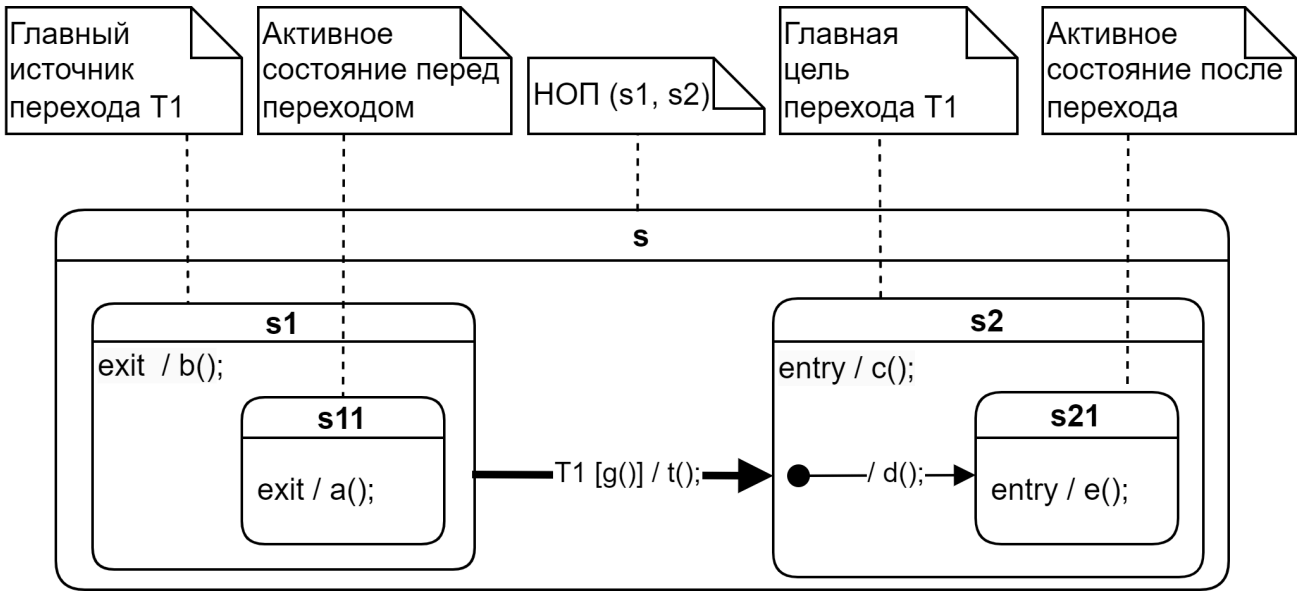


Рисунок 8. Роли состояний в переходе между состояниями

Переход между состояниями может напрямую связывать любые два состояния. Эти два состояния (они могут быть составными) обозначаются как главный источник и как главная цель перехода. На рисунке 8 можно увидеть пример простого перехода, а также пояснение ролей состояния в этом переходе. Переход в состояние включает в себя выполнение следующих действий в следующей последовательности:

1. Вычисление истинности связанного с переходом ограждающего условия и выполнение следующих шагов, только если значение ограждающего условия равно «истина» (TRUE).
2. Выход из конфигурации исходного состояния.
3. Выполнение связанных с переходом действий.
4. Вход в конфигурацию целевого состояния.

Последовательность переходов легко определить в простом случае, когда и главный источник, и главная цель перехода находятся на одном уровне вложенности. Например, на рисунке 8 переход T1 вызывает проверку истинности ограждающего условия g(); за ним

следует последовательность действий a(); b(); t(); c(); d(); e();, при условии, что значение ограждающего условия равно «истина» (TRUE).

Однако, в случае, когда главный источник и главная цель перехода вложены на разных уровнях иерархии состояний, не всегда очевидно, сколько уровней вложенности необходимо покинуть.

Переход между состояниями включает в себя выход из всех вложенных состояний, начиная с текущего активного состояния (может быть прямым или транзитивным подсостоянием главного исходного состояния) и заканчивая (но не включая) состоянием наименьшего общего предка (НОП) (также Least Common Ancestor, LCA) главного исходного состояния и главного целевого состояния. Как видно из названия, НОП – это наименьшее составное состояние, которое является суперсостоянием (предком) одновременно и исходного, и целевого состояний.

Как было описано выше, порядок выполнения выходных действий всегда начинается с самого глубоко вложенного состояния (текущего активного состояния) вверх по иерархии до наименьшего общего предка (но без выхода из него). Например, как показано на рисунке 8, наименьшим общим предком (s1, s2) состояний «s1» «s2» является состояние «s».

Вход в конфигурацию целевого состояния начинается с того уровня, где закончились выходные действия (т.е. внутри НОП). Как было описано ранее, входные действия должны выполняться, начиная с состояния самого высокого уровня вниз по иерархии состояний до главного целевого состояния. Если целевое состояние является составным, то следует рекурсивно входить во вложенную в него машину состояний, используя локальные начальные переходы. Машина полностью входит в целевое состояние только, когда встречает простое состояние, в котором нет начальных переходов.

Существует проблема с описанной выше последовательностью переходов: требуется выполнить действия, связанные с переходом (на рисунке 8 – действия t();), после уничтожения конфигурации исходного состояния, но до создания конфигурации целевого состояния. По аналогии между выходными действиями в машинах состояний и деструкторами в ООП, эта ситуация соответствует выполнению метода класса после частичного уничтожения объекта. Конечно, в ООП такое делать нельзя. Как оказывается, в машинах состояний реализовывать подобное также очень неудобно.

Намного более разумно выполнять связанные с переходом действия в контексте начального состояния – в том же контексте, где вычисляется истинность ограждающего условия (на рисунке 8 – условия g()). Только после вычисления ограждающего условия (здесь – g()) и выполнения действий, связанных с переходом (здесь – t();), происходит атомарный выход из конфигурации исходного состояния и вход в конфигурацию целевого состояния. Таким образом машина состояний наблюдаема только в стабильной конфигурации состояния, либо до, либо после перехода, но не посередине (в процессе перехода и выполнения выходных и входных действий). Под атомарностью здесь понимается неделимость процесса, он может быть выполнен только целиком или не выполнен совсем. Под наблюдаемостью машины состояний понимается возможность определения ее текущего состояния, при этом в процессе перехода текущее состояние определить невозможно, пока не завершился переход, а также выполняемые при этом входные и выходные действия конфигураций исходного и целевого состояний, соответственно.

В этом случае при реализации последовательности выполнения переходов сохраняется необходимый порядок выхода из исходной конфигурации с последующим входом в конфигурацию целевого состояния, однако действия, связанные с переходом, выполняются полностью в контексте исходного состояния, то есть, перед выходом из конфигурации исходного состояния. В частности, последовательность переходов выглядит так:

1. Вычисление истинности ограждающего условия, связанного с переходом, и выполнение следующих шагов, только если значение ограждающего условия равно «истина» (TRUE).

2. Выполнение действий, связанных с переходом.

3. Атомарный выход из конфигурации исходного состояния и вход в конфигурацию целевого состояния.

Например, как показано на рисунке 8, переход T1 вызовет вычисление ограждающего условия g() и последующие за этим действия t(); a(); b(); c(); d(); e(), при условии, что значение ограждающего условия g() равно «истина» (TRUE).

2.6. Локальные переходы в сравнении с внешними

В иерархических машинах состояний выделяют два вида переходов между состояниями. Наиболее часто используются **внешние переходы**, в которых из исходного состояния – главного *источника* перехода – всегда происходит выход, а в состояние – главную *цель* перехода – происходит вход. Другой тип перехода называется **локальный переход**. Во многих случаях внешние и локальные переходы фактически идентичны. Однако, локальный переход не приводит к выходу из главного исходного состояния, если главное целевое состояние является подсостоянием главного исходного состояния.

На рисунке 9 противопоставлены локальные (а) и внешние (б) переходы. В верхнем ряду показан случай, когда в главном источнике находится главная цель (то есть, когда целевое состояние вложено в исходное). Локальный переход не приводит к выходу из источника (и к выполнению выходных и входных действий исходного состояния), тогда как внешний переход вызывает выход из источника и повторный вход в него же (с соответствующим выполнением выходных и входных действий источника). В нижнем ряду показан случай, когда, наоборот, исходное состояние вложено в целевое. Локальный переход не приводит к выходу из целевого состояния (соответственно, при этом не выполняются выходные и входные действия целевого состояния), тогда как внешний переход приводит к выходу из цели и повторный вход в нее же (с выполнением выходных и входных действий целевого состояния).

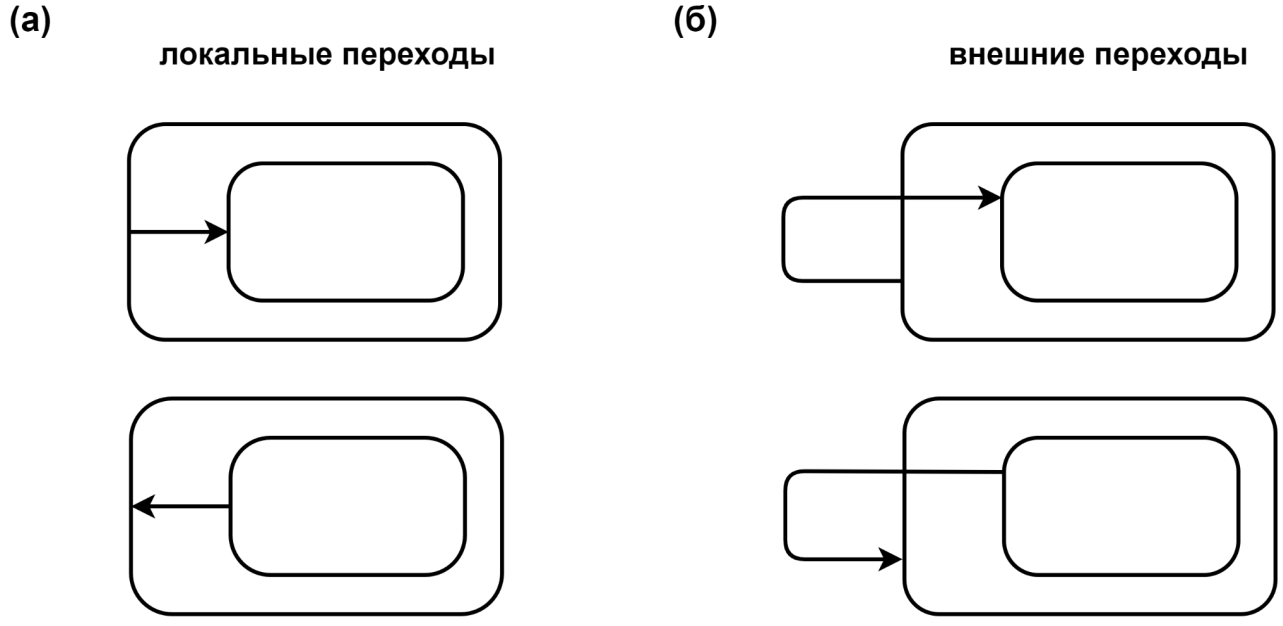


Рисунок 9. Локальные переходы в сравнении с внешними

2.7. Диаграммы машин состояний и автоматический синтез кода

Диаграммы машин состояний – это эффективный инструмент для построения выполнимых машин состояний. Существующие универсальные системы автоматической генерации кода по графическим моделям (например, Umple) поддерживают разные варианты диаграмм машин состояний. Доступные средства автоматизации обычно не только автоматически генерируют код на основе диаграмм машин состояний, но и позволяют устранять ошибки и тестировать модели состояний на графическом уровне.

В рассматриваемых системах разработки диаграммы машин состояний используются как основа графических языков программирования, обеспечивая визуальное (графическое) представление разрабатываемого алгоритма, как правило, значительно более наглядное, чем текстовое, облегчающее контроль программиста над создаваемым алгоритмом, повышающее стабильность и надежность работы программы. Разработанные диаграммы машин состояний в дальнейшем обрабатываются генератором программного (или машинного) кода для формирования непосредственно исполняемых управляющих программ (синтеза кода).

Зачастую под автоматическим синтезом кода понимают генерацию программы для решения задачи на основе формулировки *условий* задачи (постановки задачи). Основанные на диаграммах инструменты не могут этого обеспечить, потому что диаграмма машины состояний – это просто высокоуровневое (в основном, визуальное) описание *решения* задачи, а не формулировка условий этой задачи.

Кроме того, основанные на диаграммах универсальные инструменты (такие, как Umple) могут автоматически сгенерировать лишь базовый, «скелетный» код. Программист должен самостоятельно разработать весь конкретный прикладной код, например, действия и ограждающие условия. Роль базового кода заключается в объединении различных выражений действий и ограждающих условий, чтобы обеспечить правильную работу машины состояний в соответствии с логикой диаграммы машины состояний. Например, сгенерированный базовый код обычно управляет очередью событий, отправкой событий, вычислением истинности ограждающих условий или порядком выполнения переходов (включая вход и выход в/из соответствующие/их состояния/ий). Часто системы автоматической генерации кода также включают в себя среду исполнения реального времени, которая тесно интегрируется с операционной системой.

На данный момент в России разработаны средства моделирования и выполнения машин состояний, тесно связанные со средой исполнения таким образом, что получаемый код является полным и завершенным, несет в себе не только базовые, но и прикладные возможности [3]. Это возможно за счет определенной специализации инструментов, рассчитанных на использование в конкретных средах. Так, инженерный симулятор ОРБИТА содержит инструмент для автоматического преобразования диаграмм машин состояний из графического формата GraphML в исполняемый код на языке программирования Python с подключением всех необходимых библиотек для имитационного моделирования процессов управления полетом космического корабля. При этом код ограждающих условий, действий и других элементов модели (например, код инициализации) задается прямо на диаграмме, а затем включается в итоговую программу [16].

Примерами замкнутых систем моделирования и выполнения машин состояний являются видеоигры Национальной киберфизической платформы, обладающие полезным образовательным эффектом, такие, как полезные игры «Защита пасеки» и «Академия» игровой платформы «Берлога» [17]. В них поведение игровых юнитов полностью описывается с помощью диаграмм машин состояний, использующих возможности так называемых модулей (программных моделей подсистем роботов) таким образом, что диаграмма машины состояний непосредственно выполняется в системе, управляя поведением

юнита. В данном случае пользователь не имеет возможности получать внутренний программный код, управляющий игровым юнитом, а только лишь наблюдать за результатами выполнения разработанной машины состояний.

В дополнение и в развитие к данным играм разрабатывается интегрированная среда разработки Lapki IDE [18] – графическая среда разработки киберфизических систем на основе расширенных иерархических машин состояний. Среда Lapki IDE полностью совместима с игровой платформой «Берлога», а также предоставляет возможность разработки программ для микроконтроллерной аппаратно-программной платформы Arduino. При наличии единого графического интерфейса для представления машин состояний, среда разработки использует различные прикладные элементы отдельно для создания машин состояний для игровой платформы «Берлога», отдельно для формирования машин состояний для аппаратно-программной платформы Arduino. Результаты преобразования диаграмм также различны – в одном случае это файл формата GraphML, являющийся формой представления диаграммы машины состояний в полезной игре игровой платформы «Берлога». В другом – законченный исполняемый код для аппаратно-программной платформы Arduino с подключением всех необходимых библиотек поддержки функционирования всех имеющихся в среде Lapki IDE элементов.

2.8. Ограничения диаграмм машин состояний

Машины состояний были придуманы как «визуальный формализм сложных систем», поэтому с самого начала они были неотрывно связаны с графическим представлением в виде диаграмм. Однако важно понимать, что концепция иерархических машин состояний выходит за рамки какой-либо конкретной нотации, графической или текстовой. Нотация диаграмм машины состояний не только визуальная. Любая нетривиальная машина состояний (тот есть машина состояний, разрабатываемая для решения реальной задачи) требует значительного количества текстовой информации (например, описания специфики или программного кода действий и ограждающих условий). Точный синтаксис выражений действий и ограждающих условий обычно не определен, поэтому многие используют либо структурированный русский/английский язык, либо, более формально, выражения из языка, на котором реализуется программа, например, C, C++ или Java. На практике это означает, что нотация диаграмм машины состояний сильно зависит от конкретного языка программирования.

Большинство диаграмм машин состояний представлены, главным образом, графически. В виду этого диаграммы машины состояний недостаточно четко описывают последовательность обработки, будь это порядок вычисления истинности ограждающих условий или порядок отправки событий в ортогональные области. Чтобы обойти эту проблему, существующие требования стандарта OMG UML [1] к диаграммам вынуждают разработчика не полагаться на какую-либо конкретную последовательность. Но при практической работе с машинами состояний у разработчика всегда есть полный контроль за порядком выполнения, поэтому ограничения, которые накладывают требования стандарта UML, могут быть излишне жесткими. Аналогично, для графического представления потока управления диаграммы машины состояний требуют большого количества служебных элементов (**псевдосостояний**, таких как **слияния** (joins), **разветвления** (forks), **переходные состояния** (junctions), **точки выбора** (choicepoints) и т.д.). Эти элементы представляют собой замаскированную блок-схему, методы структурного программирования которой уже давно признали не очень эффективными. Такие элементы графической нотации имеют не больше ценности в представлении потока управления, чем обычный структурный код.

Использование диаграмм машин состояний ориентировано на компьютерные инструменты автоматизации программирования. Однако машина состояний, представленная в

таких инструментах, – это не только диаграмма, но и текстовое представление, которые совместно описывают не только логику состояний и переходов, но и действия. Пользователи этих инструментов могут получить несколько взаимодополняющих представлений одной и той же машины состояний, как визуальных, так и текстовых, в то время как сгенерированный код является лишь одним из множества возможных представлений.

2.9. Пример работы машины состояний

Сложная логика машины состояний может оказаться довольно запутанной и для новичков, и даже для достаточно опытных программистов. Всегда желательно иметь возможность сгенерировать точную последовательность действий для каждого возможного перехода, чтобы точно знать, какие и в каком порядке выполняются действия.

В этом разделе представлен рабочий пример иерархической машины состояний (см. рисунок 10), которая содержит все возможные типы переходов вплоть до четырех уровней вложенности состояний. Эта машина состояний содержит шесть состояний: «s», «s1», «s11», «s2», «s21» и «s211». Данная машина состояний распознает девять событий от A до I, которые можно сгенерировать, набирая на клавиатуре заглавные и строчные буквы. Все действия этой машины состояний состоят только из операторов printf, которые выводят состояние машины на экран.

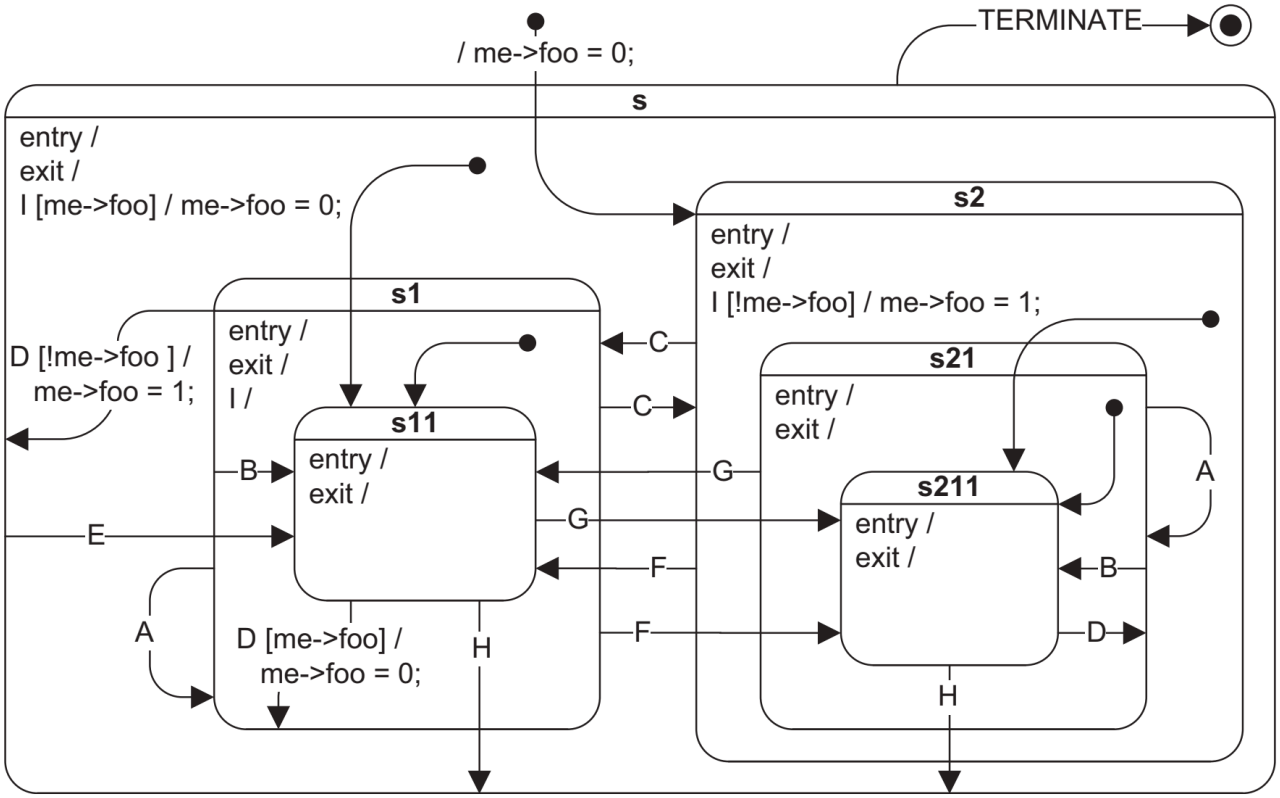


Рисунок 10. Гипотетическая машина состояний, содержащая все возможные типы переходов между состояниями вплоть до четырех уровней вложенности состояний

На рисунке 11 показан пример запуска приложения, которое выполняет данную машину состояний. Номера строк в скобках слева от окна приложения указаны для справки. На строке (1) показан результат самого верхнего начального перехода. Обратите внимание на последовательность входных действий и начальных переходов, заканчивающуюся

выведенной на экран надписью «s211-ENTRY». Ввод событий в машину состояний начинается со строки (2). За каждым сгенерированным событием следует последовательность выходных действий из конфигурации исходного состояния, за которыми следуют входные действия и начальные переходы, входящие в конфигурацию целевого состояния. По выведенным на экран надписям всегда можно определить порядок обработки переходов, а также **активное состояние** машины (последнее состояние, в которое вошла машина). Например, в строке (2) активным состоянием перед введением события G является состояние «s211», так как это последнее состояние, в которое вошла машина на предыдущей строке.

Номер строки	События	Последовательность действий
(1)		top-INIT;s-ENTRY;s2-ENTRY;s2-INIT;s21-ENTRY;s211-ENTRY;
(2)	G:	s21-G;s211-EXIT;s21-EXIT;s2-EXIT;s1-ENTRY;s1-INIT;s11-ENTRY;
(3)	I:	s1-I;
(4)	A:	s1-A;s11-EXIT;s1-EXIT;s1-ENTRY;s1-INIT;s11-ENTRY;
(5)	D:	s1-D;s11-EXIT;s1-EXIT;s-INIT;s1-ENTRY;s11-ENTRY;
(6)	D:	s11-D;s11-EXIT;s1-INIT;s11-ENTRY;
(7)	C:	s1-C;s11-EXIT;s1-EXIT;s2-ENTRY;s2-INIT;s21-ENTRY;s211-ENTRY;
(8)	E:	s-E;s211-EXIT;s21-EXIT;s2-EXIT;s1-ENTRY;s11-ENTRY;
(9)	E:	s-E;s11-EXIT;s1-EXIT;s1-ENTRY;s11-ENTRY;
(10)	G:	s11-G;s11-EXIT;s1-EXIT;s2-ENTRY;s21-ENTRY;s211-ENTRY;
(11)	I:	s2-I;
(12)	I:	s-I;

Рисунок 11. Пример выполнения машины состояний при различных событиях

Согласно логике машины состояний, введенное в строке (2) событие G обрабатывается следующим образом. Сначала активное состояние («s211») пытается обработать событие. Однако, как видно из диаграммы машины состояний на рисунке 10, в состоянии «s211» не указано, как обрабатывать событие G. Поэтому событие передается следующему состоянию более высокого по иерархии уровня, т.е. состоянию «s21». В суперсостоянии «s21» указано, как обработать событие G, так как из него есть переход между состояниями, вызывающийся событием G. Состояние «s21» выполняет действия, связанные с переходом («s21-G» в строке (2) на рис. 7). Далее состояние выполняет цепочку переходов, в которой происходит выход из конфигурации исходного состояния и вход в конфигурацию целевого состояния. Цепочка переходов начинается с выполнения выходных действий из активного состояния через все более высокие уровни иерархии. Затем происходит вход в конфигурацию целевого состояния

в противоположном порядке, а именно, начиная с самых высоких уровней иерархии к самым нижним. Переход G из состояния «s21» завершается в состоянии «s1». Однако, цепочка переходов не завершается на непосредственной цели перехода, а продолжается через начальный переход, определенный в непосредственном целевом состоянии «s1» («s1-INIT»). Наконец, путем входа в состояние «s11» («s11-ENTRY»), у которого нет начального перехода, устанавливается новое активное состояние.

В строке (3) на рисунке 11 можно увидеть, как диаграмма машины состояний обрабатывает внутренний переход. Событие I вводится в машину, когда она находится в состоянии «s11». И снова, в активном состоянии не указано, как обработать событие I, поэтому оно передает его следующему по иерархии состоянию, т.е. состоянию «s1». В состоянии «s1» содержится внутренний переход, вызывающийся событием I, что определено в разделе внутренних переходов; следовательно, событие обрабатывает состояние «s1» («s1-I» в строке (3)). На этом моменте обработка завершается. Во внутреннем переходе не происходит смена состояния, даже если такой переход унаследован от состояний из более высоких уровней иерархии.

В машине состояний внутренние переходы отличаются от переходов в себя. Строка (4) на рисунке 11 демонстрирует это различие. Когда вводится событие A, машина находится в состоянии «s11». Как и в случае с внутренними переходами, в активном состоянии «s11» не указано, как обрабатывать событие A, так что событие передается суперсостоянию «s1». В суперсостоянии есть переход в себя, вызывающийся событием A, поэтому оно выполняет действия, связанные с переходом («s1-A» в строке (4)). В этот раз, однако, осуществляется обычный переход, который требует от системы выход из конфигурации исходного состояния и вход в конфигурацию целевого состояния.

Рассматриваемые диаграммы машин состояний описывают расширенные машины состояний, что означает, что действия, выполненные машиной, могут зависеть также от значений переменных расширенного состояния. Рассмотрим, например, событие D, которое вводится в строке (5). В активном состоянии «s11» есть переход D, но у этого перехода есть ограждающее условие [me->foo]. Переменная me->foo – это переменная расширенного состояния машины состояния на рисунке 10. Можно видеть, что me->foo инициализируется в 0 в самом верхнем начальном переходе. Поэтому ограждающее условие [me->foo], которое проверяет, что значение условия не равно 0, оценивается как «ложь» (FALSE). Ограждающее условие на время отменяет переход D, содержащийся в состоянии «s11», который обрабатывается так, как если бы состояние «s11» не распознало переход вообще.

Поэтому событие D передается на следующий, более высокий по иерархии уровень, то есть в состояние «s1». В состоянии «s1» есть переход D с ограждающим условием [!me->foo]. В этот раз ограждающее условие оценивается как «истина» (TRUE) и осуществляется локальный переход D из состояния «s1» в состояние «s». Как показано на диаграмме, действие перехода меняет значение переменной расширенного состояния me->foo на 1. Поэтому в строке (6), когда снова вводится событие D, ограждающее условие [me->foo] на переходе D в состоянии «s11» оценивается как «истина» (TRUE) и происходит локальный переход из состояния «s11» в состояние «s1» (строка (6) на рисунке 11).

В строке (7) на рисунке 11 демонстрируется, что, независимо от маршрута входа и выхода в/из состояние/я, все входные и выходные действия в состояниях всегда выполняются (за исключением суперсостояния при локальном переходе, как рассмотрено выше). Главная цель перехода C из состояния «s1» – это состояние «s2». Начальный переход в главном целевом состоянии «s2» осуществляется «через» подсостояние «s21» вплоть до подсостояния «s211». Однако при этом выполняются входные действия в состоянии «s21». Этот пример демонстрирует силу концепции гарантированной завершающей обработки конфигурации

исходного состояния и гарантированной инициализации конфигурации целевого состояния, независимо от сложности маршрута входа или выхода.

Интересно, что в иерархических машинах состояний одинаковые переходы могут вызывать разные последовательности выходных действий, в зависимости от того, какое состояние наследует этот переход. Например, в строках (8) и (9) на рисунке 11 событие E вызывает точно такой же переход между состояниями, как тот, что определен в состоянии «s» (все вложенные состояния в состояние «s» наследуют от него переход по событию E). Однако, реакции в строках (8) и (9) разные, так как переход E срабатывает из разных конфигураций состояний: один раз, когда активно состояние «s211» (строка (8)), а другой раз – когда активно состояние «s11» (строка (9)).

Наконец, строки (11) и (12) демонстрируют, что ограждающие условия можно использовать и на внутренних переходах. И в состоянии «s2» и в состоянии «s» определен внутренний переход I с сопутствующими ему ограждающими условиями. В строке (11) ограждающее условие [!me->foo] (при текущем значении me->foo, равном 0) разрешает внутренний переход в состоянии «s2» (при этом также выполняется действие me->foo = 1). В строке (12) то же ограждающее условие отменяет внутренний переход в состоянии «s2» (поскольку значение me->foo изменилось, и условие не выполняется), и поэтому осуществляется внутренний переход, определенный в суперсостоянии «s».

3. Проектирование машины состояний для реального устройства

Проектирование машины состояний, как и любое другое проектирование, не является строгой наукой. Как правило, это итеративный и поэтапный процесс, состоящий из циклически повторяющихся стадий проектирования, программирования, отладки. Таким образом можно приблизиться к конечному результату разными способами, и, обычно, более, чем один вариант разработанной иерархической машины состояний удовлетворяет условиям задачи. Далее будет рассмотрено проектирование машины состояний, корректно реализующей поведение простого калькулятора. Очевидно, что представленное здесь решение является лишь одним из многих возможных.

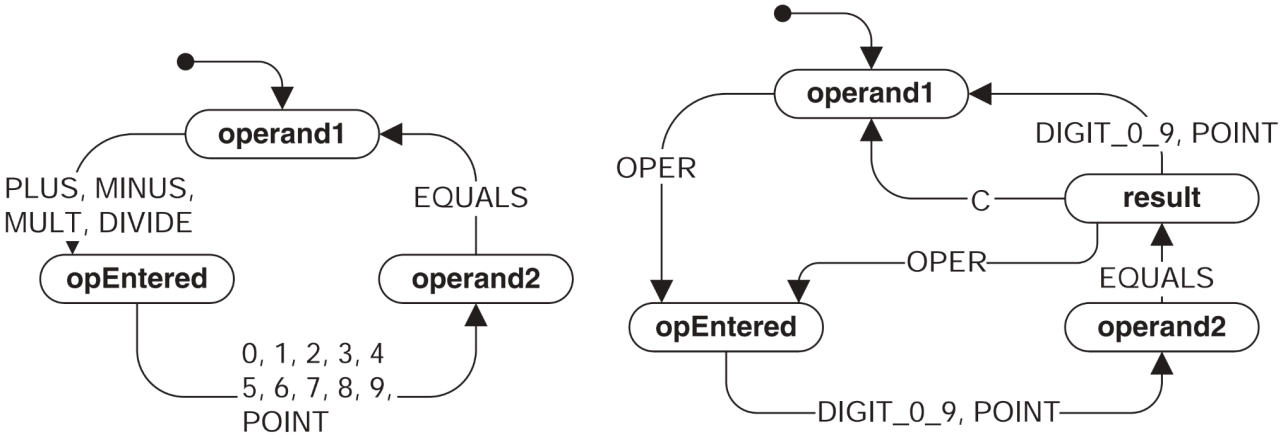
Рисунок 12. Простой электронный калькулятор, используемый в качестве примера устройства для создания диаграммы машины состояний

3.1. Условия задачи

Калькулятор (см. рисунок 12) работает следующим образом: пользователь вводит операнд, затем оператор, а затем еще один операнд, и, наконец, нажимает кнопку «равно», чтобы получить результат. С точки зрения программирования это означает, что калькулятор должен разбирать различные числовые выражения. Проблема заключается не только в том, чтобы правильно разобрать числовые выражения, но и в том, чтобы сделать это интерактивно («на ходу»). Пользователь может предоставить любой символ в любое время, и не обязательно только те символы, которые разрешены в данной ситуации. Приложение должно само игнорировать такие символы. (Данное конкретное приложение игнорирует некорректные входные данные. Часто еще более удачный подход – это активное предотвращение генерации некорректных входных данных, например, путем отключения недопустимых вариантов выбора). Кроме этого, приложение должно обрабатывать входные данные, не связанные с разбором выражений, например, Cancel (C) (общий сброс) или Cancel Entry (CE) (отмена последнего ввода информации). Все это создает проблему, которую трудно решить с помощью традиционной парадигмы программирования «событие-действие» (см. часть 1) или даже с помощью традиционной (неиерархической) машины состояний.

3.2. Обобщенное (высокоуровневое) решение

На рисунке 13 показаны первые шаги по разработке диаграммы машины состояний калькулятора. На самом первом этапе (рис. 13 (а)), машина состояний пытается реализовать основное назначение системы (основной вариант использования), которое заключается в вычислении выражений: операнд1 оператор операнд2 равно… Машина состояний начинает работу в состоянии «operand1», функция которого состоит в том, чтобы гарантировать, что пользователь может ввести только допустимый операнд. Для достижения цели это состояние нуждается во вложенной машине состояний, но на данном этапе это пока не рассматривается. Критерием перехода из состояния «operand1» является ввод оператора: «+» (PLUS), «-» (MINUS), «*» (MULT) или «/» (DIVIDE). Диаграмма машины состояния входит в состояние «opEntered», в котором калькулятор ожидает ввода второго операнда. Когда пользователь вводит цифру (0…9) или десятичную запятую, машина состояния переходит в состояние «operand2», аналогичное состоянию «operand1». Когда пользователь нажимает «=» (EQUALS), калькулятор вычисляет выражение и выводит на экран результат. Затем он возвращается в состояние «operand1», чтобы подготовиться к новому вычислению.



(а) (б)

Рисунок 13. Первые два шага в проектировании диаграммы машины состояний калькулятора

Однако у простой диаграммы машины состояний на рисунке 13 (а) есть одна серьезная проблема. Когда на последнем шаге пользователь нажимает «=», машина состояний не может перейти непосредственно в состояние «operand1», так как приложение сразу сотрет результат с экрана (чтобы подготовиться к новому операнду 1). Необходимо добавить еще одно состояние – «результат» («result»), в котором калькулятор делает паузу для отображения результата (рис. 13 (б)). В состоянии «результат» возможны три различные ситуации:

1) пользователь может нажать на кнопку оператора, чтобы использовать результат в качестве первого операнда для нового вычисления;

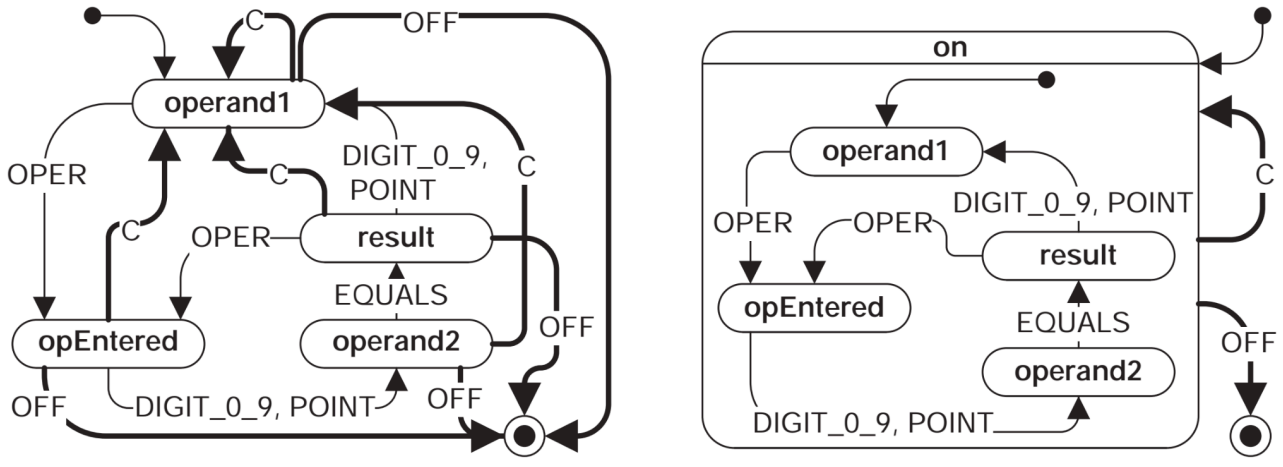
2) пользователь может нажать Cancel (C) (общий сброс), чтобы начать совершенно новое вычисление;

3) пользователь может ввести цифру или десятичную запятую, чтобы начать ввод нового первого операнда.

Совет: на рисунке 13 (б) показан прием, который стоит запомнить: объединение сигналов PLUS (умножение), MINUS (вычитание), MULTIPLY (умножение) и DIVIDE (деление) в сигнал более высокого уровня – OPER (операнд). Это преобразование помогает избежать повторения одной и той же группы переключающих событий на двух переходах (из «operand1» в «opEntered» и из «result» в «opEntered»). Такие преобразования часто упрощают проектирование более существенно, чем самые изощренные схемы состояний и переходов.

3.3. Повторное использование поведения

Машина состояний на рисунке 13 (б) принимает команду C (общий сброс), только находясь в состоянии «результат». Однако пользователь ожидает, что он в любой момент сможет отменить действие и начать работу сначала. Аналогично, он ожидает, что он в любое время сможет выключить калькулятор. На диаграмме машины состояний на рисунке 14 (а) эти функции реализованы непосредственно. Эффективнее будет выделить общий переход в состояние более высокого уровня под названием «on» (включено) и позволить всем подсостояниям повторно использовать переходы C (общий сброс) и OFF (выключение) путем наследования поведения (рис. 14 (б)).



(а) (б)

Рисунок 14. Применение вложенности состояний для повторного использования общих переходов

3.4. Разработка составных состояний

Состояния «operand1» и «operand2» нуждаются во вложенных машинах состояний для разбора чисел с целой и дробной частями. На рисунке 15 для этих состояний используется наименование «operandX».

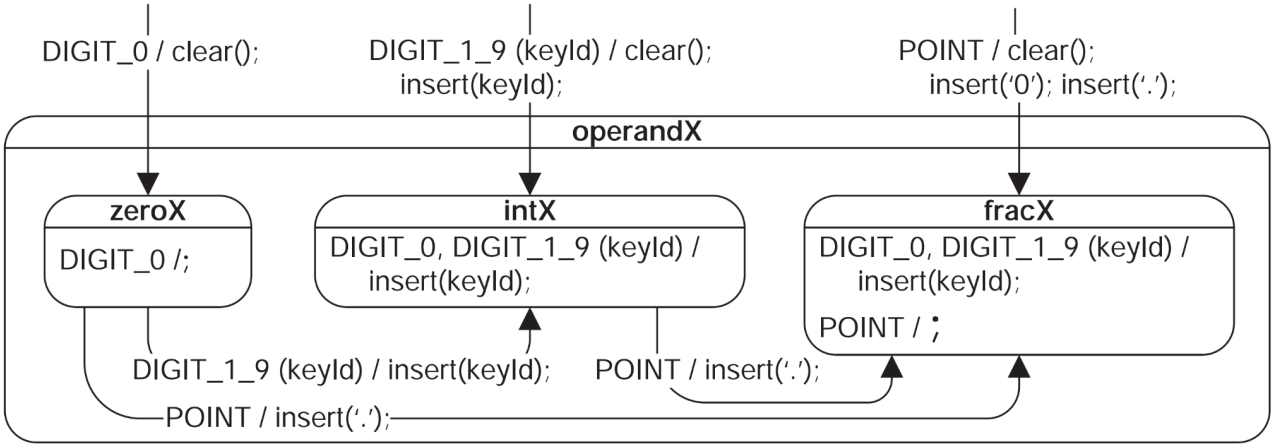


Рисунок 15. Внутренняя вложенная машина состояний для состояний «operand1» и «operand2»

Эти вложенные машины состояний включают три подсостояния. Машина входит в подсостояние «zero», когда пользователь нажимает клавишу «0». Задача подсостояния «zero» заключается в том, чтобы игнорировать дополнительные нули, которые пользователь может попытаться ввести (чтобы калькулятор отображал только один 0). Здесь использована нотация для явного игнорирования события. Используется внутренний переход (в данном случае, DIGIT_0), за которым следует явно пустой список функций (точка с запятой в языке C).

Функция подсостояния «int» заключается в разборе целой части числа. В это состояние машина входит либо извне, либо из соседнего подсостояния «zero» (когда пользователь нажимает клавиши от «1» до «9»). Наконец, подсостояние «frac» разбирает дробную часть числа. В него машина входит либо извне, либо из обоих соседних подсостояний, когда пользователь нажимает на десятичную запятую («,»). Подсостояние «frac» явно игнорирует событие ввода десятичной запятой POINT, так что пользователь не может набрать несколько десятичных запятых в дробной части числа.

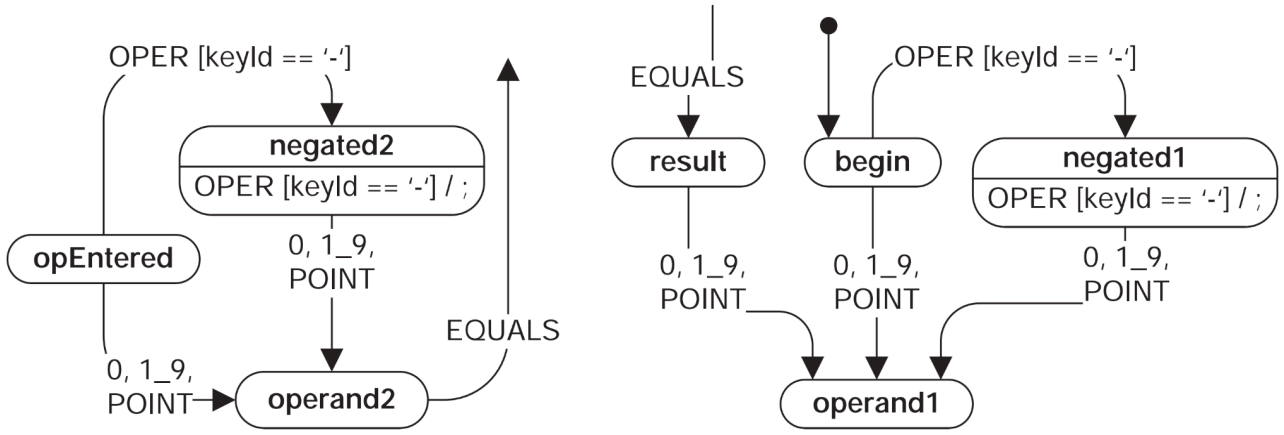
3.5. Совершенствование поведения

Последний шаг подводит диаграмму машины состояний калькулятора к этапу, в котором он действительно может вычислять выражения. Однако, он может работать только с положительными числами. На следующем этапе добавляется обработка отрицательных чисел. Это, пожалуй, самая сложная проблема в данной разработке, так как одна и та же кнопка «-» (минус) в некоторых ситуациях представляет собой бинарный оператор вычитания, а в некоторых – унарный оператор изменения знака числа.

Существует только две возможных ситуации, в которых «-» (минус) может однозначно представлять смену знака, а не вычитание:

1) в состоянии «opEntered» (как в выражении 2 * -2 =);

2) в начале нового вычисления (как в выражении -2 * 2 =).



(а) (б)

Рисунок 16. Два случая обработки отрицательных чисел

В первом случае проблему решить проще (рис. 16 (а)). Необходимо добавить еще одно состояние – «negated2», в которое входит машина, когда введен оператор «-» (минус, MINUS) (в данном случае в переходе OPER используется ограждающее условие, выделяющее конкретный оператор). При входе в состояние «negated2» на дисплее отображается «-0», и не пропадает с дисплея, когда машина переходит в состояние «operand2». Такое поведение отличается от «opEntered», так как в состоянии «opEntered» машина должна очистить дисплей, чтобы подготовиться к вводу второго операнда.

Второй случай, в котором «-» означает смену знака, более запутанный, поскольку условие «начало нового вычисления» более сложное. Здесь оно отражает ситуацию, возникающую сразу после запуска программы, или после того, как пользователь нажимает на кнопку C (общий сброс), но не тогда, когда калькулятор отображает результат прошлого вычисления. На рисунке 16 (б) можно увидеть решение для такого случая. Чтобы зафиксировать поведение, характерное для «начала нового вычисления», создается новое состояние – «begin» («начало») (начальный переход теперь направлен на «begin», а не на «operand1»). Остальная часть решения аналогична первому случаю, только теперь роль состояния «opEntered» играет состояние «begin».

3.6. Последние штрихи

Калькулятор почти готов. Последние штрихи включают в себя добавление переходов Cancel-Entry в соответствующих ситуациях, а также добавление состояния «error» (ошибка) для выявления ситуаций, когда пользователь пытается поделить на 0 или вводит слишком большое число. На рисунке 17 показан окончательный вид диаграммы машины состояний калькулятора.

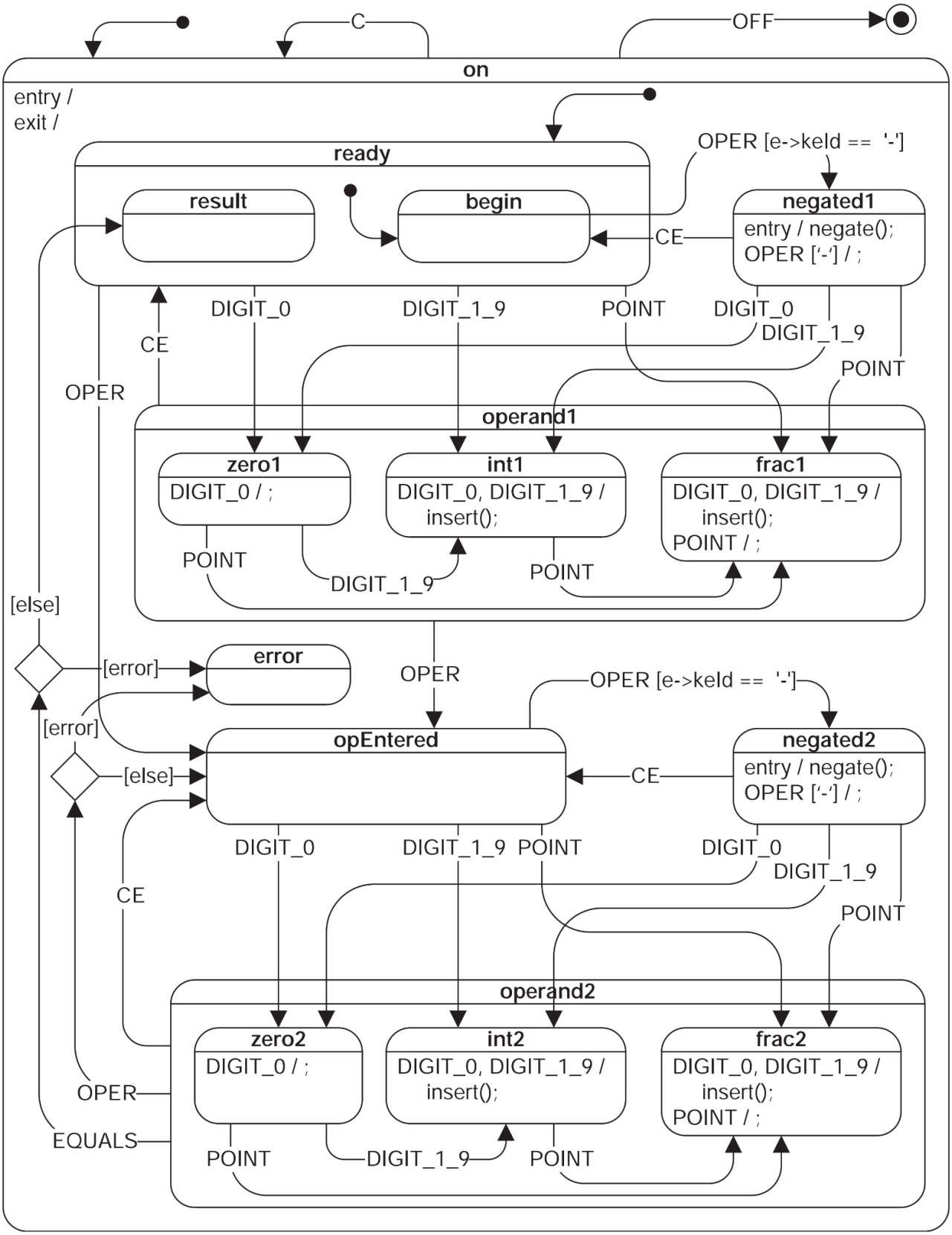


Рисунок 17. Окончательный вид диаграммы машины состояний калькулятора

Выводы

Главная проблема при программировании событийно-ориентированных систем – это определение, какие действия система должна выполнить в ответ на данное ей событие. В основном, действия определяются природой события или текущим контекстом (т.е. последовательностью прошлых событий, в которых система была задействована). Традиционные методы, такие как парадигма программирования «событие-действие», не учитывают контекст и, как результат, появляется код с непропорционально большим количеством запутанной условной логики, который программисты называют «спагетти-код».

Методы, основанные на машинах состояний, позволяют значительно сократить количество различных путей выполнения кода и упростить условия, которые проверяются в каждой точке разветвления. В машине состояний обработка событий зависит как от природы события, так и от контекста (состояния) системы. Состояния – это «блоки поведения», а парадигма программирования «событие-действие» применяется локально в рамках каждого состояния. Понятие состояния – это очень полезная абстракция истории системы, способная фиксировать только актуальную последовательность событий (и игнорировать все неактуальные). В расширенных машинах состояний (машинах состояний с «памятью») состояние соответствует качественному представлению поведения системы, тогда как переменные расширенного состояния (память программы) соответствуют количественному.

Событие – это тип мгновенного стимула, который может заставить машину выполнять действия. У событий могут быть параметры, которые передают количественную информацию об этом событии. Получив экземпляр события, машина состояний реагирует на него, выполняя действия (исполняя код). Реакция может включать в себя изменение состояния, т.н. переход между состояниями.

Во всех способах описания машин состояний предполагается использование модели «выполнение до завершения» (RTC). В этой модели все действия, вызванные экземпляром события, должны завершиться до того, как следующий экземпляр события может быть отправлен в машину состояний, что означает, что машина состояний выполняет неделимые этапы RTC и начинает обработку каждого события в стабильной конфигурации состояния (то есть когда переход между состояниями полностью завершен).

Самым важным новшеством по сравнению с классическими («плоскими») машинами является внедрение иерархически вложенных состояний. Ценность вложенных состояний заключается в том, что они позволяют избежать повторений, которые неизбежны в традиционной «плоской» машине состояний. Логика вложенности состояний позволяет подсостояниям определять только отличия от поведения суперсостояний, что способствует совместному и повторному использованию поведения. Отношение между подсостоянием и его суперсостоянием называется наследованием поведения. Наследование поведения так же фундаментально, как и наследование классов в объектно-ориентированном программировании, и позволяет выстроить целые иерархии состояний, которые соответствуют иерархиям классов в ООП. Правильно спроектированные иерархии состояний соответствуют принципу подстановки Лисков (LSP), определенному для состояний.

Машины состояний поддерживают входные и выходные действия для состояний. Они предоставляют средства для гарантированных начальной и завершающей обработок, подобно тому, как это делают конструкторы и деструкторы классов. Входные действия всегда выполняются, начиная с самого внешнего состояния, что аналогично конструкторам классов, выполняющихся с самого общего класса. Выходные действия, аналогично деструкторам, всегда выполняются в обратном порядке.

Изначально диаграммы машины состояний были разработаны как универсальный способ визуализации поведения систем любой сложности, поэтому они в значительной

степени ориентированы на графическое представление. Однако важно отличать основополагающую концепцию иерархической машины состояний от ее графического изображения. Также важно различать диаграммы машин состояний и блок-схемы.

Проектирование эффективных машин состояний не является простой задачей и, как обычно бывает в программировании, является поэтапным, итеративным процессом. Повторное использование поведения не появляется автоматически, необходимо активно искать возможности его внедрения.

Приложение А.
Материал для углубленного изучения.

А.1. Применение принципа подстановки Лисков для состояний

Определение отношений между подсостояниями и суперсостояниями как наследования имеет множество практических последствий. Возможно, наиболее важным в применении к иерархии состояний является принцип подстановки Лисков (LSP). В своей классической формулировке для классов принцип подстановки подразумевает, что подкласс может свободно заменять свой суперкласс. Это означает, что каждый экземпляр подкласса должен быть совместим с каждым экземпляром суперкласса и что любой код, разработанный для работы с экземпляром суперкласса, должен правильно работать, даже если вместо экземпляра суперкласса используется экземпляр подкласса.

Так как наследование поведения – это один из видов наследования, принцип подстановки можно применить к состояниям так же, как и к классам. Принцип подстановки, применяющийся к состояниям, подразумевает, что поведение подсостояния должно иметь те же характеристики, что и поведение суперсостояния. Например, все состояния, вложенные в состояние **«нагрев»** печи (в том числе **«поджаривание»** и **«выпекание»**) должны обладать теми же базовыми характеристиками, что и состояние **«нагрев»**. В частности, если нахождение в состоянии **«нагрев»** означает, что включен нагревательный элемент, то никакое подсостояние в этом суперсостоянии не может его выключить (без выхода из состояния **«нагрев»**). Выключение нагревателя при нахождении в состоянии **«поджаривание»** или **«выпекание»** будет несовместимо с нахождением в состоянии **«нагрев»** и будет указывать на плохую конструкцию машины состояний (нарушение принципа подстановки).

Соблюдение принципа подстановки Лисков позволяет строить более правильные иерархии состояний и наиболее эффективно использовать абстрагирование. Например, в соответствующей этому принципу иерархии состояний вы можете безопасно уменьшить масштаб и работать на более высоком уровне состояния **«нагрев»** (таким образом, абстрагируясь от конкретных аспектов, таких как **«поджаривание»** и **«выпекание»**). Пока все подсостояния совместимы со своим суперсостоянием, в таком абстрагировании есть смысл. С другой стороны, если в подсостоянии нарушаются основные принципы принадлежности к суперсостоянию, уменьшение масштаба и игнорирование особенностей подсостояний будет неверным.

А.2. Отложенные события

Иногда событие происходит в период, когда машина состояний находится в состоянии, которое не может обработать это событие. Во многих случаях природа события такова, что его можно отложить (в пределах допустимого времени) до того, как система войдет в другое состояние, в котором она сможет обработать это событие.

В машинах состояний существует специальный механизм для отложенной обработки событий в состояниях. В каждое состояние можно добавить раздел «deferred/ [event list]». Если произойдет событие, которое указано в списке отложенных событий, оно будет сохранено (отложено) для будущей обработки до того, как машина перейдет в состояние, в списке отложенных событий которого нет данного события. При переходе в такое состояние машина состояний автоматически вызовет все более не отложенные сохраненные события и обработает их так, как будто они только что поступили.

Реализации иерархических машин состояний могут напрямую не поддерживать отсрочку обработки событий. Однако, представленный в книге [1] шаблон состояния «Отложенное событие» показывает, как можно примерно повторить эту функцию, явно откладывая и вызывая события.

Приложение Б.
Средства автоматизации проектирования, поддерживающие диаграммы машин состояний

Некоторые автоматизированные программно-инженерные средства (computer-aided software engineering, **CASE**) с поддержкой диаграмм состояний:

- Lapki IDE – интегрированная среда разработки киберфизических систем с поддержкой графического языка программирования на основе расширенных иерархических машин состояний, с генератором программного и машинного кода, обеспечивает адаптированный интерфейс для рабочего стола, полную совместимость с игровой платформой «Берлога», возможность разработки программного обеспечения для микроконтроллерной аппаратно-программной платформы Arduino, имеет модульно-облачную архитектуру, https://github.com/kruzhok-team/lapki-client.

- QM (Quantum Platform) Model-Based Design Tool (QP Modeler) – бесплатный инструмент для проектирования на основе моделей (MBD) и автоматической генерации кода для разработки программного обеспечения на основе современных машин состояний (диаграмм состояний UML) и QP Real-Time Embedded Frameworks, Quantum Leaps, LLC, https://www.state-machine.com/products/qm .

- IBM Engineering Systems Design Rhapsody (Rational Rhapsody) (ранее – Telelogic Rhapsody) https://www.ibm.com/products/systems-design-rhapsody .

- Stateflow, Mathworks, https://www.mathworks.com/products/stateflow.html .

- IAR Visual State, IAR Systems, https://www.iar.com/products/iar-visual-state/ .

- AnyLogic simulation software, AnyLogic Company, https://www.anylogic.com/ .

- Stately Studio и XState, Stately, https://stately.ai [19].

- Umple, https://cruise.umple.org/umple/ .

Приложение В.
Разработка машины состояний реального устройства – от простого к сложному

Рассмотрим пример итеративного и инкрементного процесса разработки машины состояний для светофора с различными режимами работы сигналов с помощью диаграмм. Будем постепенно увеличивать сложность машины состояний, совершенствуя поведение и расширяя функциональность светофора. Принимаем, что для фактической реализации работы светофора используется программная библиотека на языке программирования Python, содержащая функции управления оборудованием светофора:

- allLightsOff() – функция выключения всех сигналов светофора;
- redLightOn() – включение красного сигнала светофора;
- yellowLightOn() – включение желтого сигнала светофора;
- greenLightOn() – включение зеленого сигнала светофора;
- blinkingOn(color) – включение мигания сигнала заданного цвета;
- blinkingOff – выключение мигания сигнала любого цвета;
- timer.start() – запуск таймера;
- timer.stop() – останов таймера;
- timer.read() – получение информации о текущем значении таймера;
- debug(' ') – отладочный вывод информации;
- turnOn() – включение оборудования светофора;
- turnOff() – выключение оборудования светофора.

В системе предусмотрены события таймера TIME_TICK, возникающие на каждом такте работы таймера; события ВЫКЛЮЧИТЬ, ВКЛЮЧИТЬ, АКТИВИРОВАТЬ, ДЕАКТИВИРОВАТЬ, возникающие при подаче соответствующей команды управления.

Функции библиотеки реализуют управление светофором путем отправки неблокирующих сообщений, таким образом, не внося задержек и блокировок в работу машины состояний.

1. Простейшая машина состояний для двухсекционного светофора (с зеленым и красным сигналами) (рис. 18).

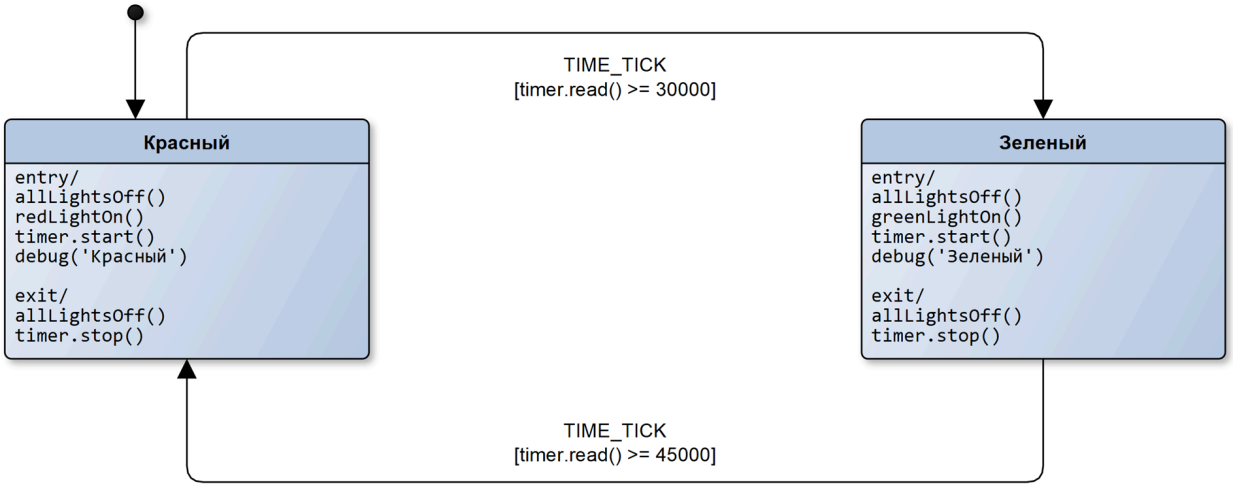


Рисунок 18. Машина состояний для двухсекционного светофора

2. Машина состояний для трехсекционного светофора (с зеленым, желтым и красным сигналами и сочетанием красного и желтого сигналов) (рис. 19).

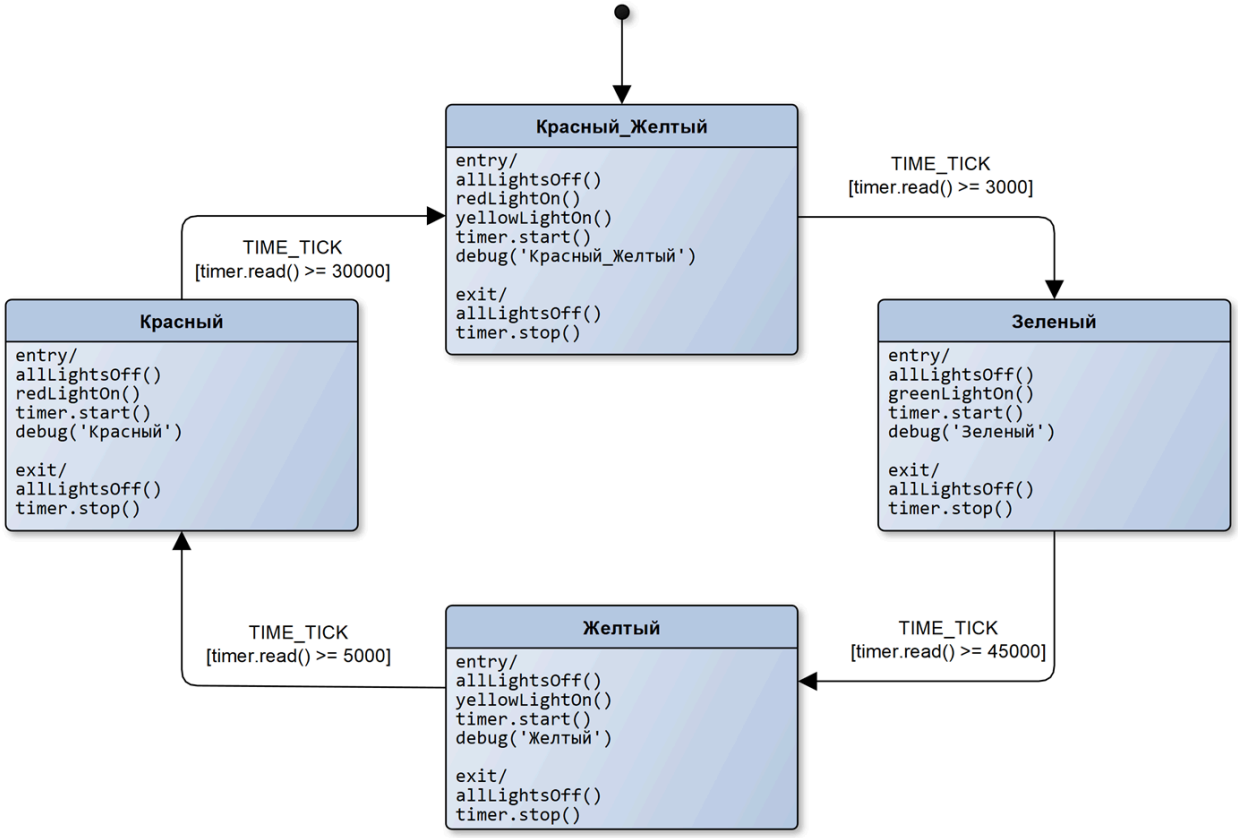


Рисунок 19. Машина состояний для трехсекционного светофора

3. Машина состояний для трехсекционного светофора с возможностью включения и выключения (рис. 20). Идентичные переходы по событию ВЫКЛЮЧИТЬ реализованы из всех основных состояний.

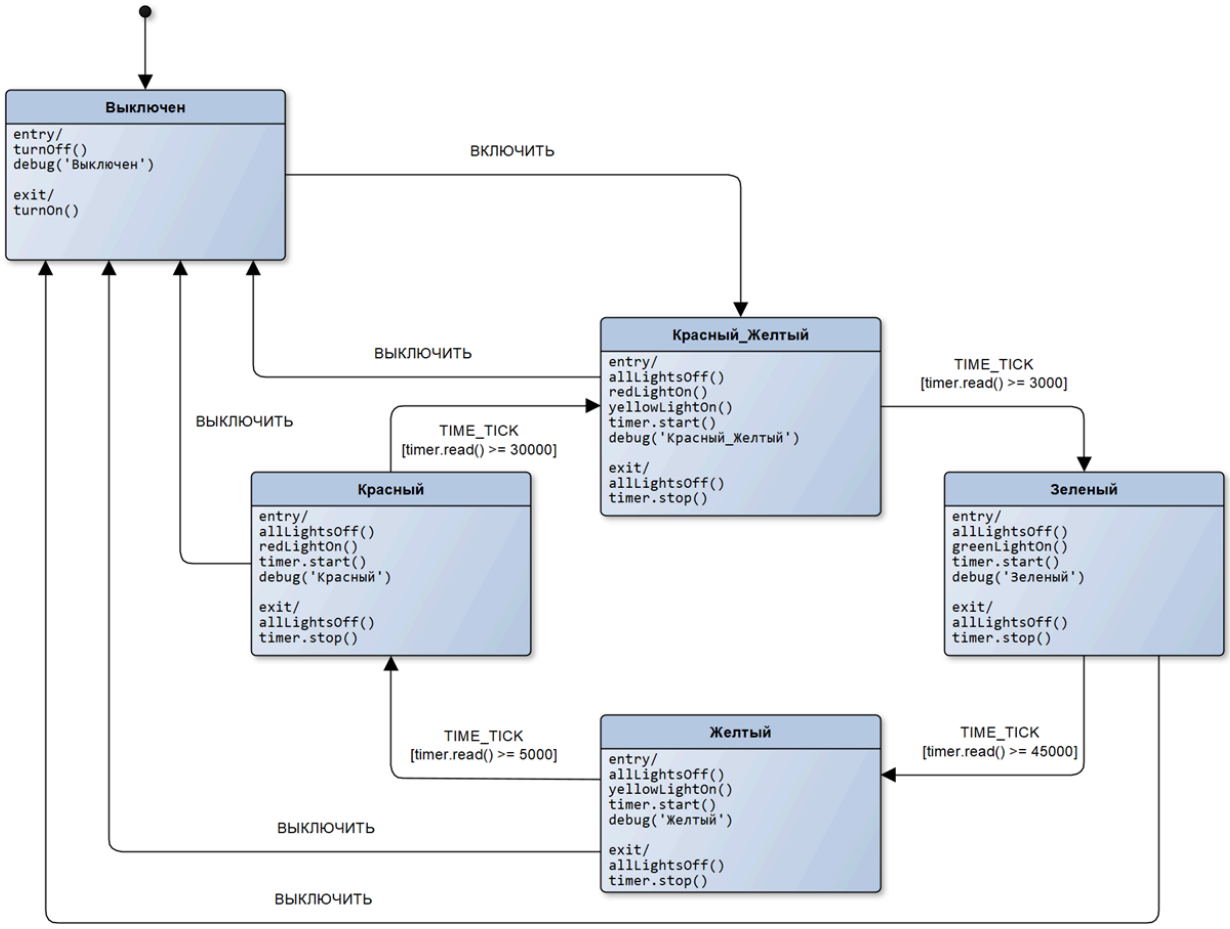


Рисунок 20. Машина состояний для светофора с возможностью включения и выключения

4. Машина состояний для трехсекционного светофора с возможностью включения и выключения, а также активации и деактивации основного режима работы с переходом в режим нерегулируемого перекрестка (с желтым мигающим сигналом) (рис. 21). Идентичные переходы по событиям ВЫКЛЮЧИТЬ и ДЕАКТИВИРОВАТЬ реализованы из всех основных состояний, что демонстрирует явление «комбинаторного взрыва» сложности традиционной («плоской») машины состояний.

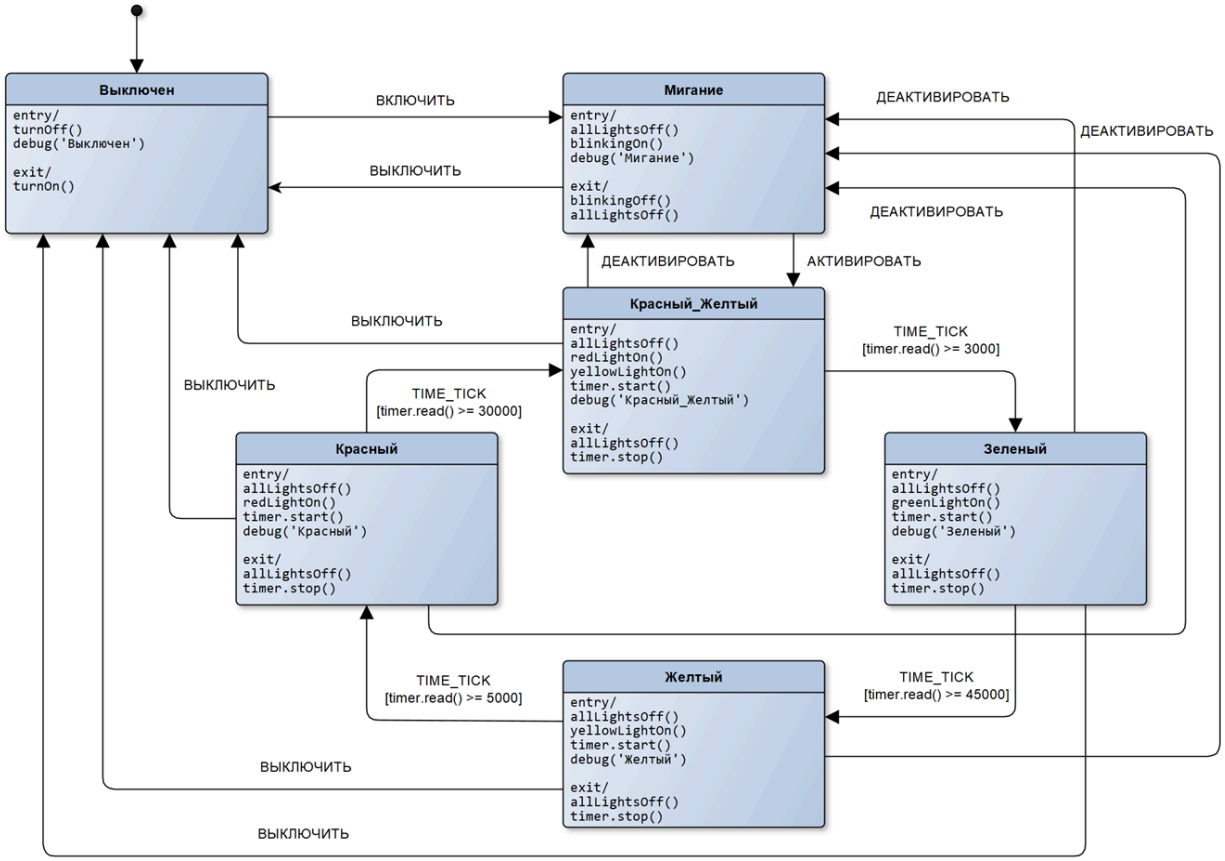


Рисунок 21. Машина состояний для светофора с возможностью перехода в режим нерегулируемого перекрестка

5. Иерархическая машина состояний для светофора с возможностью перехода в режим нерегулируемого перекрестка (рис. 22). Преобразование плоской машины состояний (рис. 21) в иерархическую не меняет поведения системы, но существенно сокращает сложность описания и количество переходов за счет повторного использования поведения при возникновении событий ВЫКЛЮЧИТЬ и ДЕАКТИВИРОВАТЬ.

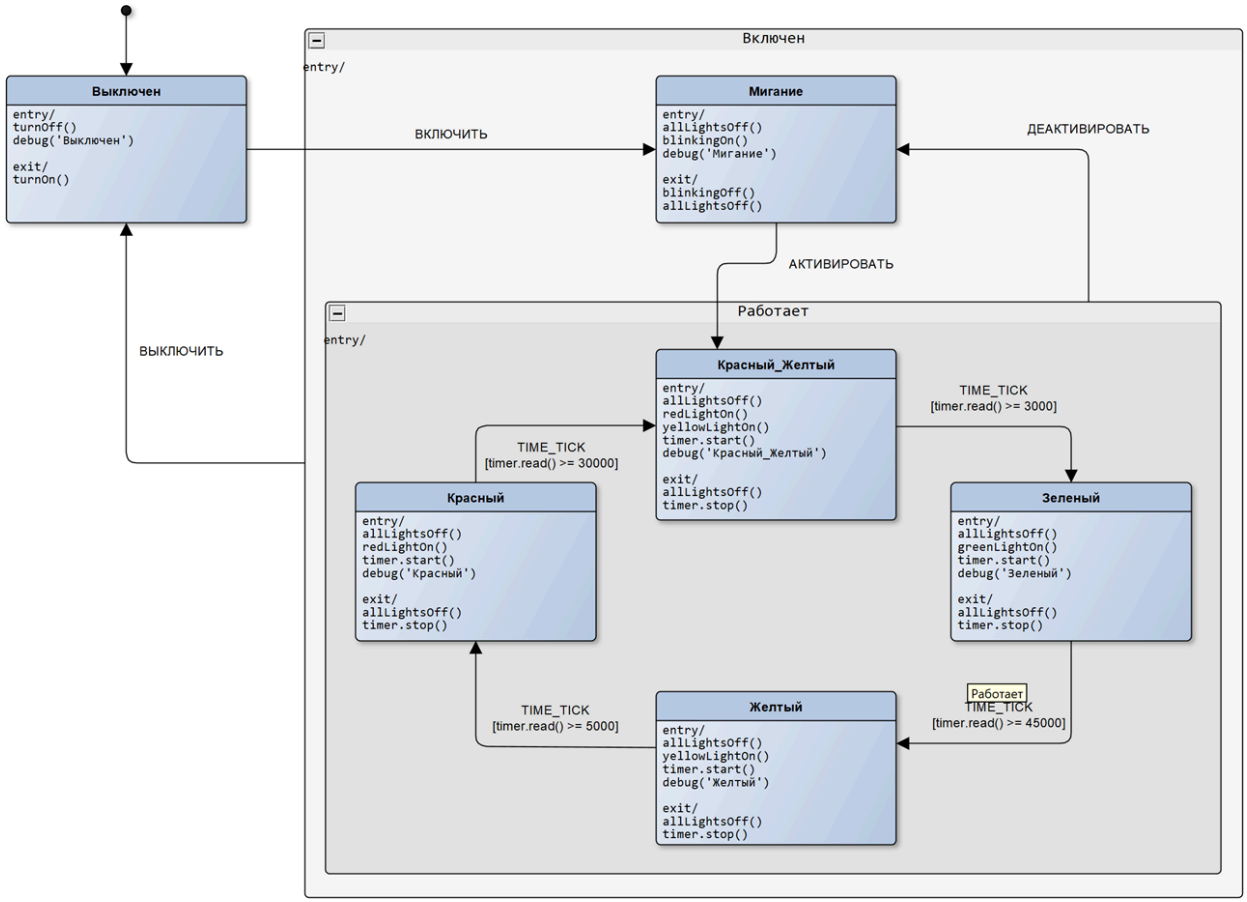


Рисунок 22. Иерархическая машина состояний для светофора с возможностью перехода в режим нерегулируемого перекрестка

6. Иерархическая машина состояний для светофора с зеленым мигающим сигналом (рис. 23). Зеленый и желтый мигающие сигналы реализованы с помощью библиотечной функции blinkingOn(green) непрозрачным для разработчика машины состояний образом. Данная функция должна работать асинхронно по отношению к работе основной машины состояния.

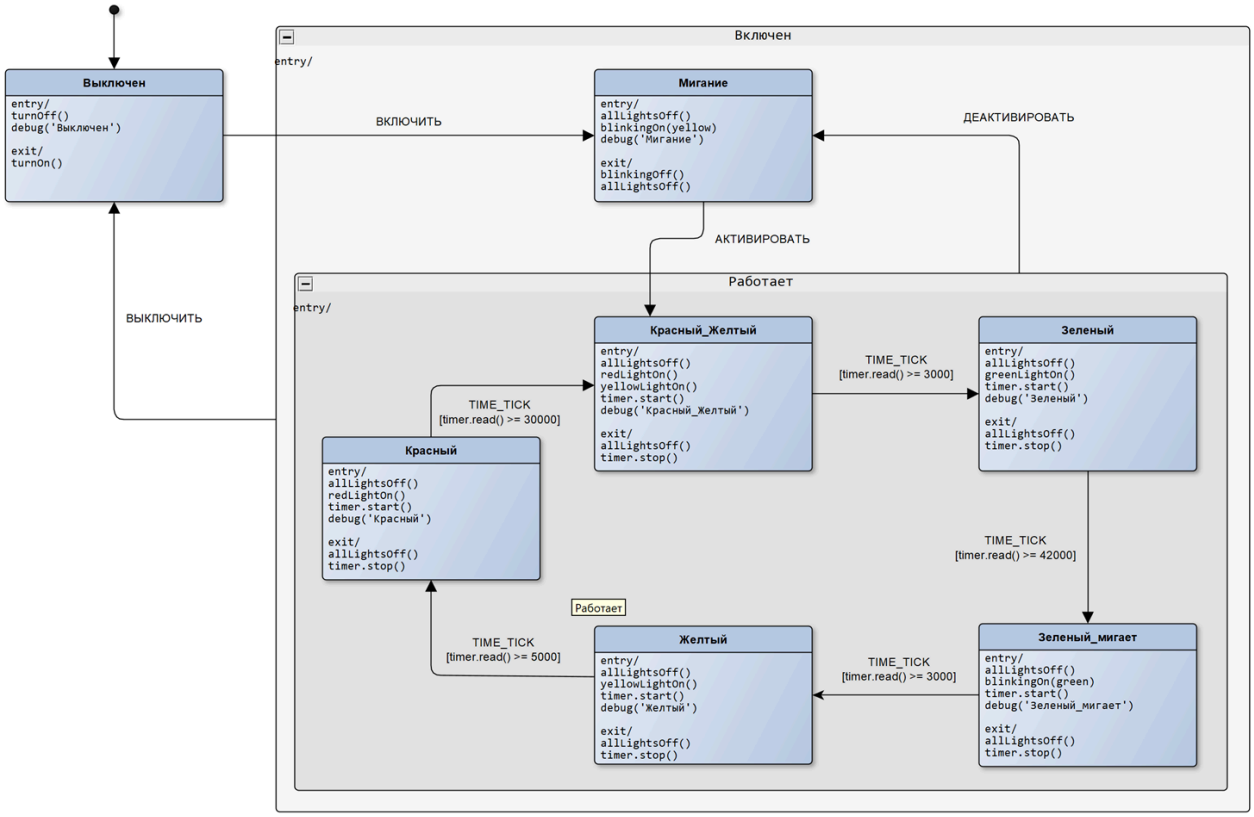


Рисунок 23. Иерархическая машина состояний для светофора с мигающим зеленым сигналом

7. Иерархическая машина состояний для светофора с зеленым мигающим сигналом (рис. 24). Зеленый мигающий сигнал реализован с помощью вложенной машины состояний **Зеленый_мигающий** под полным контролем разработчика машины состояний.

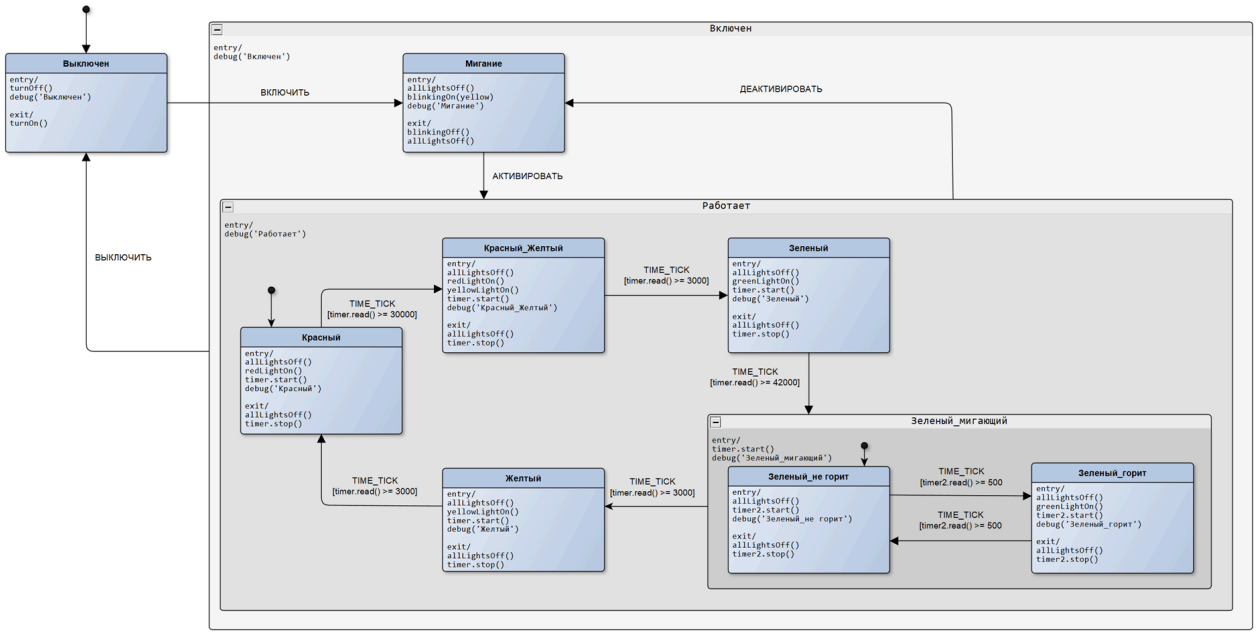


Рисунок 24. Машина состояний для светофора с вложенной машиной состояний для реализации зеленого мигающего сигнала

8. Иерархическая машина состояний для светофора с зеленым мигающим сигналом (рис. 25). Зеленый и желтые мигающие сигналы реализованы с помощью вложенных машин состояний под полным контролем разработчика машины состояний.

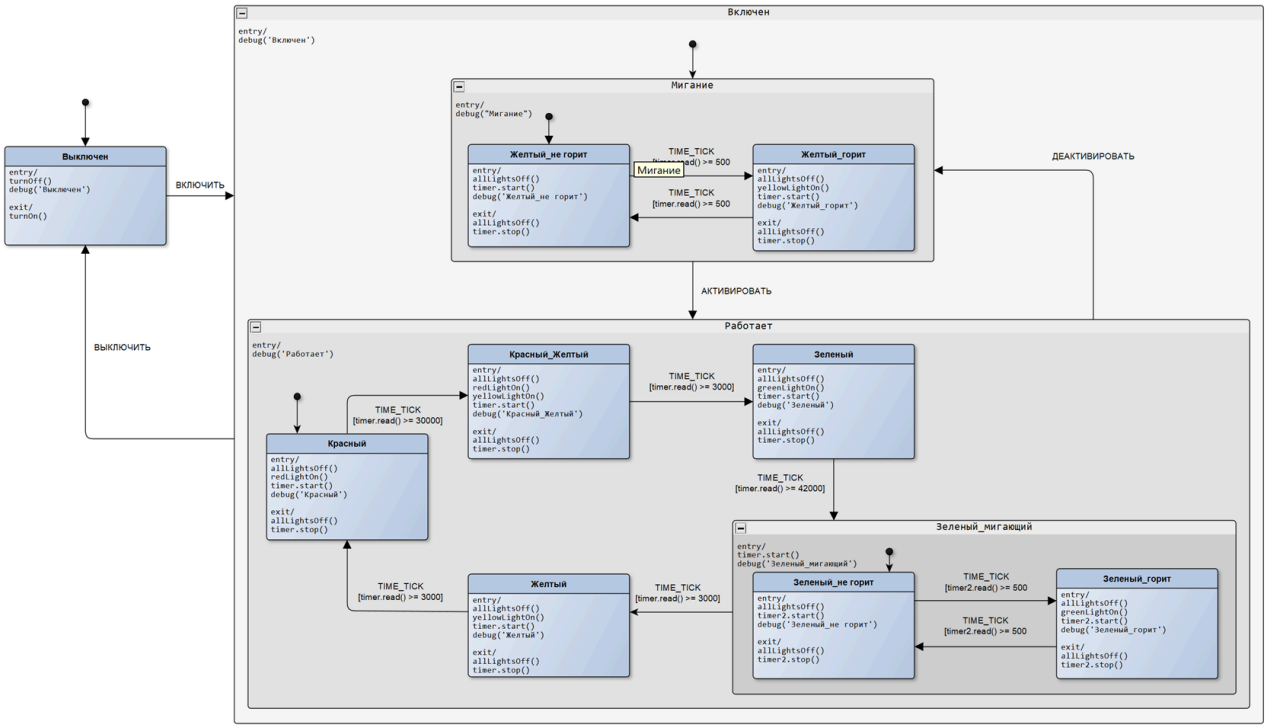


Рисунок 25. Машина состояний для светофора с вложенными машинами состояний для реализации зеленого и желтого мигающих сигналов

Глоссарий

Абстрактный (дискретный) автомат (в теории алгоритмов) – математическая абстракция, модель дискретного устройства, имеющего один вход, один выход и в каждый момент времени находящегося в одном состоянии из множества возможных. На вход этому устройству поступают символы одного алфавита, на выходе оно выдает символы (в общем случае) другого алфавита [https://ru.wikipedia.org/wiki/Абстрактный_автомат].

Аппаратно-программная платформа (hardware and software platform) – совокупность аппаратных и программных средств, обеспечивающих функционирование киберфизических систем в рамках НКФП.

Генератор программного и машинного кода (software and machine code generator) – инструмент в рамках НКФП, который преобразует программу на графическом языке программирования (например, язык расширенных иерархических машин состояний) в язык программирования высокого уровня или машинный код.

Графический язык программирования (visual programming language) – человекочитаемый язык, предназначенный для написания программы для компьютера или вычислительного устройства, благодаря которому текстовое описание алгоритма дополняется визуальным (графическим).

Декомпозиция абстрактных автоматов – представление абстрактного автомата совокупностью нескольких, более простых автоматов. Декомпозиция обычно соответствует параллельной, последовательной или смешанной работе более простых автоматов [20].

Диаграмма машины состояний UML – расширение математической концепции конечного автомата (машины состояний) в приложениях информатики, выраженной в нотации унифицированного языка моделирования (UML). Диаграмма машины состояний UML – это объектно-ориентированный вариант диаграммы машины состояний, адаптированный и расширенный с помощью UML. Целью использования диаграмм машин состояний UML является преодоление основных ограничений традиционных машин состояний при сохранении их основных преимуществ. Диаграммы машин состояний UML вводят новые концепции иерархически вложенных состояний и ортогональных областей, расширяя при этом понятие действий.

Диаграмма состояний – ориентированный граф для машины состояний (конечного автомата), в котором вершины обозначают состояния, дуги показывают переходы между двумя состояниями [https://ru.wikipedia.org/wiki/Диаграмма_состояний_(теория_автоматов)].

Игровая платформа «Берлога» (gaming platform «Berloga») – платформа для создания полезных игр, включенная в проект НКФП и направленная на стимулирование интереса к технологиям и программированию.

Иерархические машины состояний (hierarchical state machines) – машины состояний (конечные автоматы), в которых используется иерархия/вложенность состояний.

Интегрированная среда разработки (integrated development environment; IDE) – комплекс программных инструментов, предназначенных для коллективного дизайна систем и написания программ.

Киберфизическая система (cyberphysical system) – интеллектуальная система, объединяющая элементы цифрового (информационного) и физического (аппаратного) миров,

состоящая из программной и аппаратной составляющих, которая характеризуется высокой степенью технологической связанности

Машина (с конечным числом) **состояний** (**конечный автомат,** КА) в теории алгоритмов – математическая абстракция, модель дискретного устройства, имеющего один вход, один выход и в каждый момент времени находящегося в одном состоянии из множества возможных. Является частным случаем абстрактного дискретного автомата, число возможных внутренних состояний которого конечно. При работе на вход последовательно поступают входные воздействия, а на выходе КА формирует выходные сигналы. Обычно под входными воздействиями принимают подачу на вход автомата символов одного алфавита, а на выход КА в процессе работы выдает символы в общем случае другого, возможно даже не пересекающегося со входным, алфавита. Переход из одного внутреннего состояния КА в другое может происходить не только от внешнего воздействия, но и самопроизвольно [https://ru.wikipedia.org/wiki/Конечный_автомат].

Начальный переход – переход от начального псевдосостояния, не имеющий переключающего события, но могущий иметь связанные с переходом действия.

Национальная киберфизическая платформа (the National Cyberphysical Platform) – российская аппаратно-программная платформа, включающая новую методологию и средства разработки и нацеленная на создание линейки киберфизических устройств и киберфизических систем для различных сфер применения, а также на сквозное применение в сфере образования, включая общее, дополнительное и профессиональное образование.

Парадигма программирования (programming paradigm) – совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию), набор концептов, определяющий организацию вычислений и структурирование работы, выполняемой компьютером.

Полезные игры (serious games) – игры, в т.ч. видеоигры, обладающие полезным социальным эффектом, например, образовательным.

Расширенные иерархические машины состояний (extended hierarchical state machines) – расширенная версия конечных автоматов, включающая вложенность состояний и использование переменных, используемая для описания поведения сложных систем и программирования их реакции на внутренние и внешние события с использованием графического представления. Применение данного подхода представлено в стандарте UML 2.0: Statecharts

Расширенные машины состояний (extended state machines) – машины состояний, дополненные переменными.

Реактивная система – система, изменяющая свое поведение в ответ на конкретные ситуации; система, движимая событиями.

Событийное или событийно-ориентированное программирование (англ. event-driven programming; в дальнейшем СОП) – парадигма программирования, в которой выполнение программы определяется событиями – действиями пользователя (клавиатура, мышь, сенсорный экран), сообщениями других программ и потоков, событиями операционной системы (например, поступлением сетевого пакета). Также события могут быть обусловлены предустановленными таймерами, коллизиями (от лат. collisio — столкновение) объектов игрового мира и т.п.

СОП можно также определить как способ построения компьютерной программы, при котором в коде (как правило, в головной функции программы) явным образом выделяется главный цикл приложения, тело которого состоит из двух частей: выборки события и обработки события [https://ru.wikipedia.org/wiki/Событийно-ориентированное_программирование].

Литература

1. Samek, M. (2008). Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems (2nd ed.). CRC Press. https://www.state-machine.com/doc/PSiCC2.pdf. https://doi.org/10.1201/b16463.

2. Miro Samek, «A Crash Course in UML State Machines», Quantum Leaps, 2015, https://www.state-machine.com/doc/AN_Crash_Course_in_UML_State_Machines.pdf

3. Канжелев С. Ю., Шалыто А. А. Автоматическая генерация автоматного кода // Информационно-управляющие системы. 2006. № 6.

4. Шалыто А.А. Switch-технология. Алгоритмизация и программирование задач логического управления. – СПб.: Наука, 1998. – 628 с.

5. Шалыто А.А. Логическое управление. Методы аппаратной и программной реализации. – СПб.: Наука, 2000. – 780 с.

6. ISO/IEC 19505-2:2012(E), Date: April 2012, Information technology – Object Management Group. Unified Modeling Language (OMG UML), Superstructure, formal/2012-05-07, https://www.omg.org/spec/UML/ISO/19505-2/PDF.

7. Фаулер М. UML. Основы, 3-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2004. – 192 с.

8. Буч, Г. Язык UML. Руководство пользователя / Г. Буч, Дж. Рамбо, И. Якобсон; перевод Н. Мухина. – 3-е изд. – Москва : Академия АйТи, ДМК Пресс, 2022. – 494 с.

9. Wieringa R.J., Design Methods for Reactive Systems. Yourdon, Statemate, and the UML, 2003.

10. Ian Horrocks, Constructing the User Interface with Statecharts, 1999.

11. Комбинаторный взрыв (Combinatorial explosion). https://wiki.loginom.ru/articles/combinatorial-explosion.html.

12. Harel, David, Statecharts: A Visual Formalism for Complex Systems, Science of Computer Programming, 8, 1987, pp. 231-274, http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf.

13. Буч, Гради, и др. Объектно-ориентированный анализ и проектирование с примерами приложений, 3-е изд.: Пер. с англ. – М.: ООО «И.Д. Вильямс», 2008. – 720 с.

14. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995. Русский перевод: Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Паттерны объектно-ориентированного проектирования. – СПб.: Питер, 2020. – 448 с.

15. Harel, David, and Michal Politi, Modeling Reactive Systems with Statecharts, The STATEMATE Approach, McGraw-Hill, 1998, https://www.wisdom.weizmann.ac.il/~harel/STM.Book.

16. Федосеев А.И. Иерархические машины состояний для программ полета космических аппаратов. Руководство по программированию для инженерного симулятора ОРБИТА 2.0, 2023, https://github.com/dralex/orbita-simulator.

17. Национальная киберфизическая платформа. От мобильных игр к технологическому образованию. Игровая платформа «Берлога». https://platform.kruzhok.org/

18. Графическая IDE для киберфизических систем (клиентская часть). Клиентская часть Lapki IDE. https://github.com/kruzhok-team/lapki-client

19. Введение в конечные автоматы и диаграммы состояний, Справочник React разработчика с примерами кода.
https://reactdev.ru/libs/xstate/guides/introduction-to-state-machines-and-statecharts/.

20. Прикладная теория цифровых автоматов / К. Г. Самофалов, А. М. Ромлинкевич, В. Н. Валуйский, Ю. С. Каневский, М. М. Пиневич. – К.: Вища шк. Головное изд-во, 1987. – 375 с.