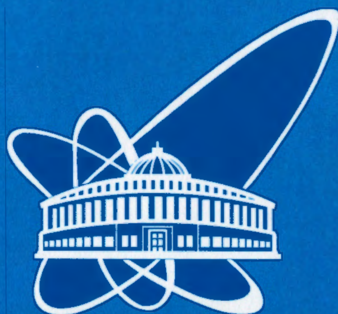
00-188



ОБЪЕДИНЕННЫЙ
ИНСТИТУТ
ЯДЕРНЫХ
ИССЛЕДОВАНИЙ

Дубна

00-188

Р10-2000-188

Е.А.Горская, В.Н.Самойлов

МОДЕЛИРОВАНИЕ СЛОЖНЫХ ИНФОРМАЦИОННЫХ СИСТЕМ НА ОСНОВЕ КОНЕЧНЫХ АВТОМАТОВ

Направлено в журнал «Автоматизация проектирования»

2000

Введение

Идея использования вычислительной техники для управления сложными технологическими процессами возникла почти одновременно с появлением первых ЭВМ. Созданные для сложных математических, физических и инженерных расчетов вычислительные машины, дополненные периферийным оборудованием, стали использоваться для решения задач управления объектами, сбором и цифровой обработкой данных. Сначала для этих целей использовались универсальные вычислительные машины с операционными системами общего назначения, затем появились операционные системы реального времени, далее стали разрабатываться управляющие, измерительные и вычислительные комплексы.

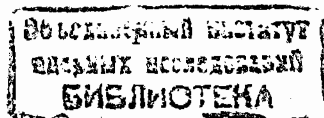
Показательна в этом смысле история развития исключительно удачной и оригинальной разработки Института точной механики и вычислительной техники в Москве – универсальной вычислительной машины БЭСМ-6: в 1967 году была создана первая операционная система для этой машины, названная "Диспетчер-68", в 1970 г. – операционная система реального времени "ОС НД –70", с развитыми средствами организации параллельных вычислений и возможностью работы в составе многомашинного комплекса, а в 1971–1973 гг. была создана распределенная ОС многомашинного комплекса, обеспечившего обработку в реальном времени больших потоков информации о полетах космических аппаратов[19].

Уровень сложности современного программного обеспечения приближается к тому уровню, который характеризует многие из объектов и явлений нашего мира, традиционно признаваемых в качестве предметов научного исследования [5].

В данной статье дан краткий анализ истории развития методологии программирования вообще и создания сложных информационных систем реального времени в частности, выявлены критерии, определяющие степень сложности программного обеспечения, показана важность создания формальной алгоритмической модели и обоснован выбор модели конечных автоматов в качестве базовой алгоритмической модели, которая была использована при создании ряда реальных систем. В первом разделе анализируются этапы развития методологии построения сложных программных систем. Во втором разделе рассматриваются основные направления в развитии программирования. В третьем разделе обсуждаются особенности построения сложных информационных систем реального времени, анализируются модели алгоритмизации программного обеспечения, обосновывается выбор модели на основе конечных автоматов.

1. Основные этапы развития методологии построения сложных программных систем

На протяжении всего периода истории развития вычислительной техники методология построения систем программного обеспечения несколько раз ко-


БИБЛИОТЕКА

ренным образом менялась. В 50-е годы, на первом этапе развития вычислительной техники, программы разрабатывались методом ad hoc (специально для данного случая): каждая система программного обеспечения была уникальным, индивидуально созданным интеллектуальным продуктом. Не существовало понятий повторного использования и взаимозаменяемости отдельных частей или формального проектирования.

В 60-х годах усилия по созданию более сложного, но более удобного в управлении и эффективного в работе программного обеспечения привели к первому существенному пересмотру аспектов его развития. Метод ad hoc уступил место методу структурного проектирования «сверху вниз». Такой подход требовал, чтобы процесс создания системы программного обеспечения состоял из некоторого числа последовательных формально законченных фаз – этапов разработки. С каждым этапом проект системы все более и более детализировался. Однако, несмотря на такую формализацию, разрабатываемые системы по-прежнему не удовлетворяли запросам пользователей[1, 2].

Второе коренное изменение в подходе к разработке программного обеспечения приходится на 70-е годы. Сложные системы программного обеспечения стали проектироваться так же, как и любые другие большие и сложные технические системы. Прежде чем приступать к реализации системы, создавалась ее алгоритмическая модель на бумаге [3]. Все современные технологии разработки программного обеспечения используют этот принцип.

Новый коренной пересмотр методологии развития программного обеспечения начался с возникновения объектно-ориентированного подхода, суть которого сводится к тому, что при моделировании сложных систем процедурно-ориентированная декомпозиция задачи заменяется объектно-ориентированной декомпозицией. При процедурно-ориентированной декомпозиции система разбивается на ряд параллельных задач, внутри которых события выстраиваются в последовательную цепочку, и синхронизация событий осуществляется на межзадачном уровне. При объектно-ориентированной декомпозиции задачи-объекты взаимодействуют между собой хаотично, непредсказуемо и одновременно, поэтому проблема распараллеливания и синхронизации событий решается внутри задачи.

2. Основные направления в развитии общей теории программирования

Исторически сложилось так, что развитие программирования как дисциплины происходило в двух направлениях. Основное, практическое, направление было связано с разработкой машинно-ориентированных языков и с построением формализованных алгоритмических языков более высокого уровня. Другое, теоретическое, направление было связано с математической логикой и теорией алгоритмов [6].

Традиционное (процедурное) программирование

Зарождение основного, практического, направления программирования следует отнести к тому моменту, когда американский математик фон Нейман (в 1946 г.) сформулировал концепцию построения ЭВМ с запоминаемой программой.

С тех пор началось развитие архитектуры ЭВМ, которая составляет основу и современного компьютера, с одной стороны, и зарождение и развитие программирования – с другой. Первым языком программирования был двоичный язык самой ЭВМ – машинный код. Затем появился ассемблер, представляющий собой символьное представление машинного кода. Далее начали появляться языки с более высоким уровнем абстракции, так называемые «языки высокого уровня» [7,8].

Теоретической основой для разработки алгоритмических языков «высокого уровня» явились лингвистико-математические понятия формального языка, формальной грамматики и формальной семантики [7].

Формальная грамматика порождающего типа, которую создал Джон Бэкус (1959), была использована в сообщении по языку Алгол, изданном Науром (1960). Эта грамматика и по сей день используется в качестве инструмента для создания и описания алгоритмических языков [7,8]. Одновременно с развитием алгоритмических языков программирования формировалась и развивалась теория программирования, которая позволяла объяснить, скоординировать, узаконить и проверить многое из накопленного эмпирического материала.

Нетрадиционное (декларативное) программирование

Истоки другого, более изысканного и более теоретизированного направления в программировании лежат в математической логике. Математическая логика зародилась в начале XX века и своим бурным развитием была обязана кризису в основаниях математики, который вызвали открытые Б. Расселом парадоксы наивной теории множеств [6]. Д. Гильберт видел выход из кризиса в идее представления математических теорий в виде формальных логико-математических исчислений, а математических рассуждений, т.е. доказательств, – в виде выводов в этих исчислениях. Такие фундаментальные понятия, как функция, отношение, аксиома, доказательство, в математической логике представляются с помощью построенных для этой цели формальных языков. Именно в недрах математической логики, еще задолго до появления первых вычислительных машин, были найдены точные понятия алгоритма и вычисляемой функции, развита семантика формальных языков и теорий. Эти понятия стали впоследствии использоваться в теории программирования. Глубокому анализу были подвергнуты выводы в логических исчислениях, что позволило с появлением компьютеров выдвинуть идею автоматического доказательства теорем, вылившуюся потом в концепцию логического и функционального программирования. Таким образом, параллельно с традиционными методами программирования, отражающими архитектуру неймановского компьютера, с самого зарождения теории программирования развивались логические и функциональные методы программирования.

Итоги кризиса 60-70-х годов

В 60-е годы ситуация, заставившая математиков обратиться к логике, в какой-то степени повторилась в программировании – произошел кризис программного обеспечения. «Языки программирования оказались в трудном положении», – утверждал Дж. Бакус в своей знаменитой Тьюринговской лекции [9]. Сфера применения компьютеров к этому времени значительно расширилась и, соот-

ветственно, выросла сложность и масштабность решаемых на них задач. Кроме того, наблюдался быстрый прогресс в развитии аппаратных средств вычислительной техники, что привело к резкому возрастанию их производительности и значительному удешевлению. Все это стало приходить в противоречие с имевшимися средствами программирования.

Кризис программного обеспечения заставил программистов-теоретиков заняться анализом основ своей науки. Выход из создавшегося положения одни видели в разработке соответствующей методологии программирования на классических языках. В качестве примера можно привести модульное программирование и структурное проектирование Дейкстры: «Как только программирование вышло за пределы допустимой сложности, произошел поворот к дисциплине... Эта дисциплина, всем нам более или менее знакомая, называется математикой. Если мы согласимся с правильностью суждений о том, что математические методы являются наиболее эффективным средством преодоления сложности, у нас не остается другого выбора, как только перестроить область программирования таким образом, чтобы стало возможным применять эти методы, ибо иных средств не существует» [2,10].

Другие выдвигали идеи, которые, в конечном счете, тоже сводились к формализации основных программистских понятий и разработке формальных способов рассуждений о программах с целью доказательства их характерных свойств. Третьи находили истоки кризиса в императивной сущности и машинной ориентированности традиционных языков и предлагали перейти к декларативным языкам и, быть может, к новой машинной архитектуре [11].

Важное значение приобрела и теория логического вывода, причем не только в задачах искусственного интеллекта, но и как средство рассуждения о программах и доказательств их свойств. Известный академик, программист и математик А. П. Ершов писал в то время: «Можно только удивляться, насколько... логика и программирование подготовлены к тому, чтобы вступить в более тесное взаимодействие. Программирование может дать абстрактной рекурсии цель, взяв у нее метод» [8].

С другой стороны, языки высокого уровня традиционного направления нуждались в детально проработанной семантике, которая имеет важнейшее значение при написании трансляторов. И здесь идеи и аппарат математической логики, уже занимавшейся проблемами семантики, оказались весьма кстати. Поначалу для формализации смысла программ использовалось бестиповое λ-исчисление А. Черча, а затем абстрактным базисом денотационной семантики стала мощная теория областей, развитая Д. Скоттом [6, с. 56 -118].

На пути поиска новых подходов к методологии программирования в 70–80-е годы получили новое развитие такие парадигмы программирования, как модульное, структурное, логическое, функциональное программирование.

Модульное программирование. Рациональным зерном модульного программирования является представление программы в виде отдельных блоков, так называемых программных модулей, которые имеют стандартный внешний интерфейс [12]. Из этих модулей, сцементированных внешним интерфейсом, как здание из кирпичей, строится программа. Эта сборка программ из модулей и соответствующая настройка выполняются автоматически программой-компоновщиком. Модульность в программировании явилась первым шагом перехода от эвристических методов разработки программ к строгим методам.

Структурное программирование. Структурное программирование – метод модульного программирования, при котором процесс конструирования программы представляется в виде такой иерархии уровней, в которой каждый определенный уровень полностью изолирован от деталей более низкого уровня [13]. Основной идеей структурного программирования является «программирование сверху вниз». Второй важной идеей структурного программирования является использование простых и наглядных структур управления ходом программы. Структурное программирование до сих пор служит важной цели систематизации программирования и познаваемости программ.

Логическое программирование. Суть метода логического программирования состоит в том, что в качестве программы решения задачи задается логическая спецификация задачи, а процесс решения задачи выполняется с использованием логического вывода и, возможно, автоматического синтеза программы в подходящем логическом исчислении [14].

В процессе составления программ на языках логического программирования не требуется указывать последовательность действий над данными, как в программах на процедурных языках; не требуется также конструировать программу в виде композиции вычисляемых функций, как при функциональном программировании. Вместо этого программист описывает в некоторой формальной системе только саму задачу: формулирует утверждения о свойствах объектов, об отношениях между объектами рассматриваемой предметной области, о вытекании одних свойств и отношений из других [15].

Функциональное программирование. В функциональном программировании программы представляются в виде функций, задаваемых с помощью композиции базовых и вновь построенных функций [16, 17].

Значительным толчком к развитию функционального программирования, разработок его технологических принципов и инструментальных средств явилась работа (1978) американского математика Дж. Бэкуса [9], подвергшего критике принципиальные недостатки традиционной (неймановской) архитектуры ЭВМ и соответствующих языков программирования и предложившего варианты языков функционального программирования.

Функциональные программы, в отличие от процедурных, не используют принцип текущего состояния памяти, не имеют операторов присваивания и переходов. Языки функционального программирования предоставляют средства более высокого уровня, не требующие описания протекания вычислительного процесса в памяти ЭВМ и во времени [18].

Итоги кризиса 90-х годов

В начале текущего десятилетия в программно-информационных технологиях начались изменения, продолжающиеся и по сей день. На смену традиционному методу пакетной обработки информации пришла парадигма «клиент-сервер». Осуществился тотальный переход с алфавитно-цифрового интерфейса на графический интерфейс (GUI). В области разработок программного обеспечения тоже произошли изменения – новая парадигма – «объектно-ориентированный» подход стал конкурировать со структурным программированием, структурным проектированием и структурным анализом. Переход к новой парадигме поиска, создания, обновления и реализации «объектов» оказался таким же трудным и знаменательным, какими были для разработчиков программ-

ного обеспечения первые опыты по структурной методологии в начале 70-х годов.

Объектно-ориентированное программирование. При классических методах программирования декомпозиция задачи производится по функциям, которые требуется выполнить в процессе ее решения, то есть проблемная область отображается на множество операторов (функций) и операндов (данных). В отличие от этого при объектно-ориентированном программировании функции (операторы) и данные (операнды) рассматриваются как два разных аспекта одного и того же объекта. Проектирование приложения для некоторой проблемной области в этом случае состоит из идентификации объектов, их поведения и реализации этого поведения на компьютере. Для объектов определяются переменные, отражающие их состояние, операции, которые они могут выполнять, и сообщения, которые они могут получать. Выполнение операций осуществляется только при получении соответствующих сообщений. Объект представляет собой модуль системы, в котором локализована определенная часть знаний о проблемной области; объект модифицируется при модификации этих знаний. Последовательность выполняемых программой функций (операторов) определяется динамически, в процессе выполнения программы, проверка соответствия операндов операторам реализуется самим объектом. Разделение задачи по алгоритмам концентрирует внимание на порядке происходящих событий, а разделение по объектам придает особое значение факторам, либо вызывающим действия, либо являющимся объектами этих действий. В связи с этим новое звучание приобретает проблема параллелизма, так как при построении сложных систем, как правило, всегда возникает необходимость в обработке многих событий, происходящих одновременно [4,5].

Концепция распараллеливания и синхронизации процессов. Концепция параллельного функционирования двух или более модулей системы настолько естественна в мире технических средств, что уже не допускает какого-либо иного образа мышления в этом плане. Однако в области программного обеспечения исторически ситуация сложилась по-другому. Получившие широкое распространение традиционные языки программирования не содержали ни средств обеспечения повторной входимости, ни средств реализации параллелизма. А между тем жизнь диктовала необходимость выполнения распараллеливания вычислительных процессов. К тому же появление векторно-конвейерных ЭВМ поставило перед программистами проблему автоматического распараллеливания программ, написанных на обычных (последовательных) языках программирования, при их компиляции. Распараллеливание вычислений выполнялось на микроуровне и достигалось в основном за счет замены итерационных циклов векторными операциями (векторизация циклов) и за счет такого упорядочения последовательности операций, при котором сводятся к минимуму задержки в выдаче команд в связи с неготовностью операндов (планирование вычислений) [20].

Идея параллелизма, несмотря на кажущуюся простоту, не так тривиальна. Более того, она настолько сложна, что для ее полной реализации требуется разработка общей теории и модели параллельных процессов. Большой интерес в этой области представляет разработанная Вл. В. Воеводиным теория исследования информационной структуры алгоритмов, записанных в форме последовательных программ, основной целью которой является распознавание и использование параллелизма в программах как на макро-, так и на микроуровне

[24]. Эта теория эффективно применяется для статического распараллеливания алгоритмов и последовательных программ.

3. Проблемы построения программного обеспечения для сложных информационных систем реального времени

Для управления физическими и технологическими процессами вычислительная система непосредственно включается в контур сбора информации, ее переработки и выдачи управляющих воздействий на реально происходящие технологические процессы. [13]. Из этого факта следует:

1. Любая автоматизированная система управления технологическим процессом распадается на две задачи. Первая – это управление реальным процессом и накопление информации. Вторая – интерфейс с человеком-оператором, в котором визуализируется состояние объекта управления и содержатся средства для ручного управления.
2. Любая система управления технологическим процессом является системой реального времени. Время реакции системы на изменения параметров объекта управления определяется внешними реальными процессами и должно быть гарантировано. Во многих системах требуется практически мгновенная реакция.
3. Широкий спектр периферийных устройств, соединяющих реальные процессы с процессами вычислительными, требует сложной и надежной системы ввода/вывода.

Необходимость распараллеливания вычислительных процессов в системах реального времени привела к разработке программных средств, позволяющих выполнять это распараллеливание [21]. Здесь развитие шло в трех направлениях:

– создание операционных систем реального времени, позволяющих приоритетно выполнять задачи и быстро (в заданный промежуток времени) реагировать на события во внешней среде;

– создание языков параллельного программирования, которые имели бы встроенные средства для задания параллельного выполнения отдельных модулей задачи;

– создание интегрированных сред разработки программных систем реального времени, в которых распараллеливание и синхронизация обеспечиваются самой средой.

Критерии сложности программных систем

Провести строгое научное исследование и выявить законы, по которым происходят изменения в области программного обеспечения, определить, по каким причинам одно изменение интегрируется и становится классикой, а другое исчезает бесследно, в настоящее время невозможно. Очевидно, что степень сложности программных систем, если их рассматривать как информационное пространство, можно определить размерностью этого пространства. Тогда в качестве главного критерия, определяющего сложность архитектурного решения системы, будет служить число измерений среды программирования [22,23].

Первое измерение. Одномерной программой будем считать линейную последовательность команд (не включающих команды перехода).

Второе измерение. Второе измерение появляется у программы, когда вводится (явно или неявно) возможность управления последовательностью выполнения команд. В процедурном программировании для такого управления служат операторы управления (предикаты), а в непроцедурном часто используются системы правил и соглашений. Вообще говоря, механизмы управления позволяют вносить «управляемый хаос» в поток последовательности выполнения команд программы. Большинство широко применяемых языков программирования в предложенной классификации должны быть отнесены к двумерным.

Третье измерение. Под трехмерной будем понимать программу, отдельные части которой могут выполняться параллельно. В программировании также есть средства, позволяющие при наличии всего лишь двух измерений программы практически реализовать ее выполнение как трехмерной программы.

Четвертое измерение. Под четвертым измерением фактически подразумевается время: подобно тому, как мы часто синхронизируем свои действия по часам с нужной нам точностью, так и параллельные вычислительные процессы функционируют в едином дискретном машинном времени. Четвертое измерение предполагает наличие единого времени у всех параллельных процессов. Если это условие выполнено, то снимаются многие проблемы, которые приходилось решать, например, синхронизация процессов.

Формальные методы моделирования алгоритмов для систем реального времени

Очевидно, что операционные системы (сетевые) реального времени с единым системным временем и интегрированные среды разработки систем реального времени можно рассматривать как четырехмерную среду программирования. Программные средства для реализации механизмов распараллеливания и синхронизации вычислительных процессов в той или иной степени там заложены.

Но предтечей любого программирования, а тем более программирования сложных систем управления, является формальная алгоритмическая модель[8]. И для создания ее нужны такие средства, которые позволили бы представить проектируемую систему во всей ее сложности и многообразии.

Алгоритмическая модель – на основе блок-схемы. В основе практически всех алгоритмических моделей лежат блок-схемы (БС). Может быть, программисты не всегда это осознают, но это понимание всегда присутствует на уровне подсознания. В «плоском» программировании, где система имеет два измерения (функции и предикаты), и применяется подход, предусматривающий точное и полное знание алгоритма работы всей системы, БС позволяет полностью формализовать этот алгоритм. Для построения алгоритма сложных систем БС недостаточно. Единственно возможный путь, который позволяет справиться с возрастающей структурной и алгоритмической сложностью программных систем,– это разбиение монолитной системы на отдельные составляющие (процессы), алгоритмы которых можно описать с помощью БС, а для описания их взаимодействия и отношения друг к другу применять какие-то другие средства.

Алгоритмическая модель на основе конечных автоматов. Предлагаемая схема построения алгоритма на основе конечных автоматов состоит в том, что вместо БС строится модель конечного автомата. Построение модели заключается в следующих шагах:

- выделение самостоятельных параллельно работающих компонентов-процессов,
- определение алгоритмов их функционирования,
- определение и реализация связей между компонентами,
- их запуск в среде функционирования.

Предлагаемая схема не требует описания общего алгоритма системы. Достаточно определить алгоритм работы каждого элемента и необходимые связи между компонентами системы.

Достоинства автоматной модели

Автомат изначально имеет множество входных и выходных каналов и потому полностью соответствует структурной модели «черного ящика». У автомата есть состояния, которых нет у блок-схем. То, что для блок-схем приходится придумывать (метод программирования с переменной состояния, таблицы решений, в конце концов, сами автоматы), для автомата часто является естественным.

Параллелизм автоматов несравненно выше, чем у блок-схем. Можно организовать параллельную работу входных и выходных каналов и выполнять параллельный анализ дуг текущего состояния для перехода в следующее состояние. Автоматы самостоятельно решают, какую дугу перехода выбрать и каким будет следующее состояние автомата. Программист только задает условия таких переходов, множество следующих состояний и определяет перечень действий, которые нужно выполнить, когда возникла ситуация перехода. Математическая теория автоматов более разнообразна и обширна, чем теория блок-схем [25– 27]. Например, автоматы можно делить, умножать и т. п. Над блок-схемами такие операции немыслимы.

Модель БС эффективна благодаря тому, что воспроизводит современную архитектуру процессоров. Заменяя блок-схемы автоматами, мы создаем свою виртуальную машину, и она достаточно эффективна, потому что воспроизводит модель реальных взаимодействующих процессов. Любая операционная система – тоже виртуальная машина. И если каждому измерению выделить свою операционную систему, то алгоритм разрабатываемой k-мерной программной системы может быть разложен на более простые его составляющие, функционирующие в разных операционных системах, т.е. в разных плоскостях.

Процессы и подпрограммы в КА - технологии

При построении автоматной модели будем исходить из того, что один и тот же объект с некоторыми отличиями может выступать как в качестве подпрограммы, так и в качестве процесса. Процесс характеризуется временем запуска, приоритетом, средой функционирования и другими параметрами, его можно запускать, останавливать и завершать, анализировать на активность и т. д. Как

процесс, подпрограмма имеет обычно те же характеристики, что и процесс, который ее запустил. Правда, возможности управлять выполнением ограничиваются только запуском. Подпрограмма должна обязательно завершиться, чтобы вернуть управление вызвавшему ее процессу или процессу, к которому она была подключена (подключение подпрограммы к «чужому» процессу вполне возможно). Таковы в общих чертах основные характеристики, существенные при рассмотрении процессов и подпрограмм.

Для организации КА-подпрограмм введем понятие вложенного автомата (подавтомата) и определим, когда становятся истинными значения, которые являются результатом работы такого автомата. Для сохранения основ теории базовой модели достаточно, чтобы автомат, вызвавший или запустивший вложенный автомат, не переходил бы в новое состояние до тех пор, пока не завершится работа вложенного автомата. Состояния подавтомата становятся новыми текущими состояниями автомата верхнего уровня. Естественно, ни глубина вложения автоматов, ни число одновременно вызванных подавтоматов ограничиваться не должны. Нужно учитывать различия в работе подавтоматов, вызванных в пределах одного действия и вызванных в пределах разных действий одного перехода. Автоматы, вызванные из разных действий, выполняются в том же порядке, что и сами эти действия. Поскольку мы условились, что действия, запущенные на одном переходе, могут выполняться параллельно, ничто не мешает задать, в частности, параллельную работу подпрограмм, вызывая их из таких действий.

Последовательность работы подавтоматов, вызванных из одного действия, определить сложно, и если она важна, лучше вызывать подавтоматы из разных действий: это позволит запускать их как параллельно, так и в нужном порядке. Программа будет чуть сложнее, но такой подход является и формально более правильным, и более гибким. Необходимо учитывать, что значения, возвращаемые вложенными автоматами, можно считать истинными лишь после перехода в новое текущее состояние автомата верхнего уровня. Поэтому анализировать возвращаемые значения нужно после перехода, когда все подавтоматы, вызванные на данном переходе, уже завершили свою работу и возвращаемые ими значения заведомо определены.

Проблемы реентерабельности подпрограмм

Для параллельного программирования огромное значение имеет реентерабельность модулей и подпрограмм, так как они должны одновременно использоваться многими процессами. В ООП реентерабельные программы легко строить с помощью средств динамического создания объектов и локальных переменных. В полной мере это распространяется и на КА-объекты. Более того, КА-модель позволяет организовать разделение не только исполняемого кода, но и логики программы: ничто не запрещает использовать сразу в нескольких программах одну и ту же таблицу переходов.

Для отображения данных можно использовать переходы из состояния в состояние, однако, с одной стороны, удобнее отделить код самой модели от кода, отвечающего за отображение данных, с другой же – такое решение не вполне корректно по отношению к модели. Например, мы ввели действие, отображающее на том или ином переходе новое состояние модели. Но ведь модель в этот момент еще только выполняет переход в данное состояние. Конечно, на

практике такие тонкости часто несущественны: микросекундой раньше или позже автомат все равно переходит в новое состояние. Но эти микросекунды важны принципиально: из них вырастает понятие нового измерения.

В сетевой среде автоматов процессы имеют формально одинаковое «автоматное» время – сетевое дискретное время. Это означает, что переходы, одновременно запущенные на исполнение, одновременно завершат свою работу. В реальной ситуации среда не запустит следующие переходы на исполнение до тех пор, пока не завершат работу текущие. Именно поэтому задача отображения данных одного процесса другим получает столь простое решение в сетевой среде автоматов. Среда гарантирует, что скорость работы автоматов будет одинаковой. Здесь не может быть ситуации, когда модель работает, а система отображения подводит.

Модель конечных автоматов как базовая модель проектирования программных систем

Описанная выше модель конечных автоматов использовалась (отрабатывалась и развивалась) как базовая алгоритмическая модель в НЦеПИ ОИЯИ с 1993г. Формальное описание алгоритмов в виде конечных автоматов – диаграмм состояний и таблиц переходов включалось в технологическую цепочку «проектирование – реализация» при создании следующих систем реального времени:

- системы дистанционного автоматического контроля работы устройств гарантированного питания [28,29]. Ниже, в качестве примера, приведен алгоритм работы программ подсистемы измерения, реализованной в среде операционной системы реального времени OS-9;
- автоматизированной системы управления ECR-источником ионов [30–32];
- автоматизированного рабочего места оператора “контрольной точки измерения” для Отдела радиоактивных и делящихся веществ ОИЯИ, в рамках проекта « Material Tracking Information System» [33].

Использование метода позволило формализовать задачи программирования взаимодействующих объектов на более ранней стадии, чем само программирование, тем самым ускорило процесс реализации проектов.

Автоматная модель алгоритма работы подсистемы измерения системы автоматического контроля параметров энергоснабжения

Алгоритм работы подсистемы измерения составляют алгоритмы работы управляющей подпрограммы PARENT и обрабатывающих подпрограмм (процессов), их взаимодействий с аппаратно-техническими средствами измерения и с внешней системой отображения и управления.

Получив очередную команду извне, программа PARENT выполняет соответствующие этой команде действия. Алгоритм работы программы PARENT, представленный в виде диаграммы состояний конечного автомата, изображен на рисунке 1 , а в табл. 1 приведены переходы и действия для этого автомата.

Обрабатывающие подпрограммы (процессы) запускаются и контролируются программой PARENT. Обрабатывающие подпрограммы обладают свойст-

вом реентерабельности, поэтому процессов может быть гораздо больше, чем подпрограмм. Обрабатывающие процессы, используя механизмы приоритетов и прерываний, параллельно друг с другом выполняют ввод, обработку и передачу данных. Обобщенный типовой алгоритм работы обрабатывающих подпрограмм в виде диаграммы состояний конечного автомата представлен на рис. 2 , а таблица переходов и действий для этого автомата в табл. 2.

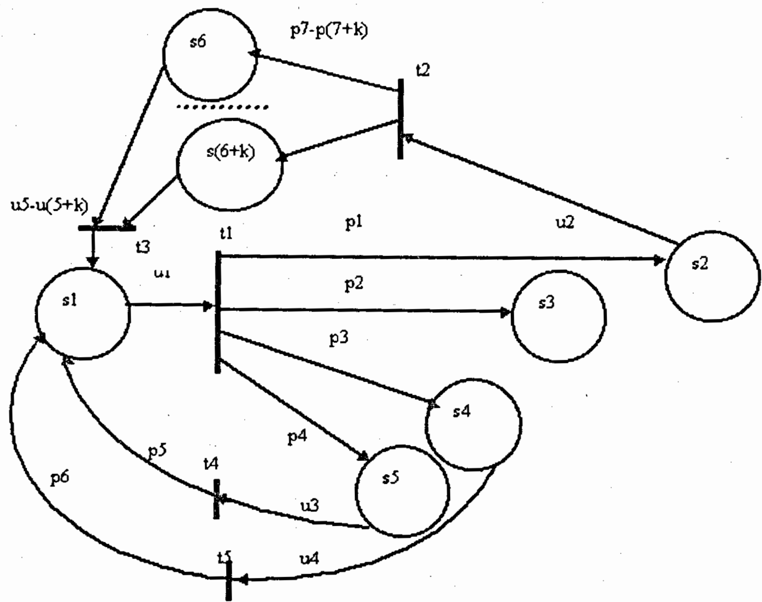


Рис.1. Алгоритм работы программы PARENT, представленный моделью конечного асинхронного автомата

Таблица 1

Состояние (s) /Позиция (t)	Условие (u) /Постусловие (p)	Действие
s1– состояние ожидания команды извне	u1–поступила команда	Разборка команды
t1– после разборки команды	p1– команда “Начать работу»	Разборка стартового файла
	p2 – команда “Завершить работу”	Уничтожаются все запущенные процессы
	p3 – команда “Изменить режим контроля точки b”	Процессу, контролирующему точку b, пересылается команда “Изменить режим контроля”
	p4 – команда “Изменить период скважности контроля в точке b”	Процессу, контролирующему точку b, пересылается команда “Изменить период скважности”
s2– состояние ожидания завершения разборки стартового файла	u2 – конец файла	Запуск (k+1) обрабатывающих процессов
t2 –после запуска (k+1) процессов	p7-p(7+k+1) – переходы на ожидания завершения запуска k процессов	Формирование запросов на сообщения от запущенных процессов
s6 – s(6+k) – ожидание подтверждений от запущенных процессов	u5-u(5+k+1) – получены подтверждения от запущенных процессов	Обработка подтверждений
t3–после обработки подтверждений	Формирование общего состояния обрабатывающих процессов	Переход в состояние ожидания сообщений и управляющих команд
s3 – состояние “останов”		
s4 – состояние ожидания сообщения о выполнении изменения режима контроля	u3 – получено подтверждение об изменении режима контроля	
s5 – состояние ожидания сообщения о выполнении изменения периода скважности	u4 – получено подтверждение об изменении периода скважности	
t4 – после обработки подтверждения u3	p5 – режим контроля точки b изменился	Возврат в состояние s1
t5 – после обработки подтверждения u4	p6 – период скважности в точке b изменился	Возврат в состояние s1

Таблица 2

Состояние (s) /Позиция (t)	Условие (u) /Постусловие (p)	Действие
1 - ожидание управляющих команд от программы PARENT и данных от устройства	u1 – команда "Начать работу"	Инициализация устройства
	u2 – управляющая команда от PARENT	Разборка команды
	u3 – данные от драйвера	Обработка полученных данных
t1 – ожидание готовности устройства	p1– готовность есть	Подготовка к вводу новой порции данных
t2 – после разборки команды	p2 – команда "Завершить работу"	Сброс операций ввода, деинициализация устройства
	p3 – команда "Изменить режим контроля "	Изменяется режим ввода и обработки
	p4 – команда "Изменить период скважности контроля "	Изменяется частота опроса и обработки данных
s2 – ожидание завершения сброса	u4 – сброс выполнен	Переход в исходное состояние
t3 – после изменения контроля	u6 – режим изменился	Работа в новом режиме контроля
t4 –после изменения периода скважности	u5 – частота опроса и обработки изменена	Работа с новым периодом скважности
t5 – обработка данных и запуск нового ввода	p7 – обработка завершена	Передача результатов в подсистему отображения
s3 – ожидание завершения передачи данных	u7 – передача завершена	Переход на ожидание следующих событий

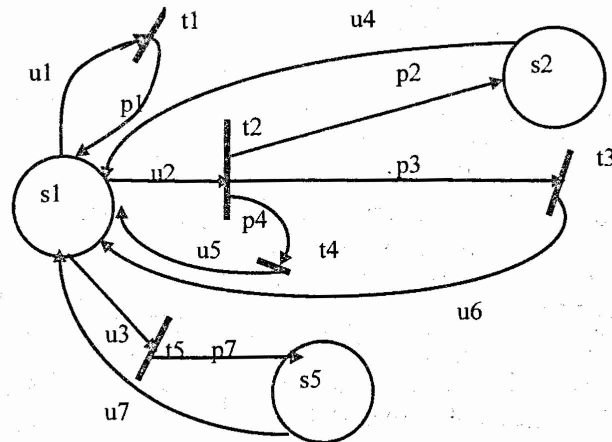


Рис. 2. Модель алгоритма типовой обрабатывающей программы в виде асинхронного конечного автомата

Литература

1. Кнут Д. Искусство программирования для ЭВМ. Том 2. Получисленные алгоритмы. М.: Мир, 1977. Том 3. Сортировка и поиск. М.: Мир 1978.
2. Дейкстра Э. Дисциплина программирования. М.: Мир, 1984.
3. DeMarco T. Structured Analysis and System Specification. Englewood Cliffs, NJ: Prentice Hall, 1979.
4. Буч Г. Объектно-ориентированное проектирование с примерами применения: Пер. с англ. М.: Конкорд, 1992.
5. Йордан Э., Аргила К. Структурные модели в объектно-ориентированном анализе и проектировании. М.: ЛОРИ, 1999.
6. Сборник статей. Математическая логика в программировании. М: Мир, 1991.
7. Хигман Б. Сравнительное изучение языков программирования. М.: Мир, 1974.
8. Ершов А.П. Избранные труды. Новосибирск: Наука, 1994.
9. Backus J. Can programming de liberated from the von Neumann style? A functional style and its algebra of programs -Comm. ACM, 1978, v.21, N 8, p. 613-641.
10. Дал У., Дейкстра Э., Хоар К. Структурное программирование. М.: Мир, 1975.
11. Knuth D.E. Structured Programming with GO TO Statements. Computer Servers, 6(40),1974.
12. Лингер Р., Миллс Х., Уитт Б. Теория и практика структурного программирования. М.: Мир.1982.
13. Словарь по кибернетике. Под ред. В. С. Михалевича. Киев: Гл. ред. УСЭ им. М.П. Бажана, 1989.
14. Братко И. Программирование на языке ПРОЛОГ для искусственного интеллекта. М.: Мир, 1990.
15. . Кларк К., Маккейб Ф. Введение в логическое программирование на микропрологе. М.:Радио и связь,1987.
16. Хювунен Э., Сеппянен Й. Мир Лиспа, том 1. Введение в язык Лисп и функциональное программирование. М.:Мир,1990.
17. Лавров С.С., Силагадзе Г.С. Автоматическая обработка данных. Язык Лисп и его реализация. М.:Наука, 1978.
18. Хендерсон П. Функциональное программирование. Применение и реализация. М.: Мир, 1983.
19. Королев Л.Н. Структуры ЭВМ и их математическое обеспечение. Главная редакция физико-математической литературы, М.: Наука, 1978.
20. Иванников В.П. и др. Проблемы построения системы программирования для векторно-конвейерных ЭВМ. – Вопросы кибернетики. Проблемы создания высокопроизводительных ЭВМ. М., 1984,ст. 111–117.
21. Вайс Р. IDE-среды приходят в реальное время. Мир компьютерной автоматизации №2/99, М.1999.
22. Любченко В.С. Мир ПК № 6/98, М.: Открытые системы,1998, ст.114–119.
23. Любченко В.С. Мир ПК №10/97, М.: Открытые системы,1997, ст.116 – 119.

24. Воеводин В.В. Математические модели и методы в параллельных процессах. М.: Наука, 1986.
25. Кудрявцев В.Б., Алешин С. В., Подколзин А. С. Введение в теорию автоматов. М.: Наука, 1985.
26. Глушков В.М. Синтез цифровых автоматов М.: 1962.
27. Брауэр В. Введение в теорию конечных автоматов М.: Радио и связь, 1987.
28. Голованова Э.З., Горская Е.А., Добрянский В.М., Маканькин А.М., Пузынин В.И., Самойлов В.Н., Чекер А.В. Комплекс программ контроля системы Энергоснабжения сложного физического эксперимента. ОИЯИ Р10-99-63, Дубна, 1999.
29. Алпатов С.В., Голованова Э.З, Горская Е.А., Добрянский В.М., Маканькин А.М., Пузынин В.И., Самойлов В.Н., Чекер А.В. Программно-технический комплекс сбора, обработки и архивирования физической информации о цепной ядерной реакции на базе шины VMEbus. ОИЯИ Р10-96-313, Дубна, 1996.
30. Горская Е.А., Самойлов В. Н. Описание комплекса программ для автоматизированной системы управления ECR-источником ионов, реализованного в интегрированной программной среде LabVIEW. ОИЯИ Р10-99-64, Дубна, 1999.
31. Горская Е. А., Самойлов Е. А. Метод построения программного обеспечения многоканальной системы автоматизированного управления физическими экспериментами на базе инструментального пакета National Instruments LabVIEW. ОИЯИ Р10-99-65, Дубна,1999.
32. Горская Е.А., Логинов В.Н., Самойлов В. Н. Описание библиотеки программ для работы с модулями КАМАК через последовательный контроллер крейта КК011 и последовательный интерфейс ПИ021 (на базе инструментального пакета LabVIEW). ОИЯИ Р10-99-66, Дубна,1999.
33. Samoilov V.N., Checker A.V., Shestakov B.A., Dobrianskii V.M., Koltin G.P., Heinberg C.L., Cowley P.J., Stottlemyre A.J. «Materials Tracking Information System: A Computerized Nuclear Materials Accounting System for Tracking Items at Russian Nuclear Facilities». 39th Annual Meeting Proceedings of the Institute of Nuclear Materials Management, Vol. XXVII, Naples, Florida, July 26-30, 1998, pp. 122-131.

Рукопись поступила в издательский отдел
3 августа 2000 года.