УНИВЕРСИТЕТ ИТМО

НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
июль–август 2021 Том 21 № 4 http://ntv.ifmo.ru/
SCIENTIFIC AND TECHNICAL JOURNAL OF INFORMATION TECHNOLOGIES, MECHANICS AND OPTICS
July–August 2021 Vol. 21 No 4 http://ntv.ifmo.ru/en/
ISSN 2226-1494 (print) ISSN 2500-0373 (online)

НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК УНИВЕРСИТЕТ ИТМО
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

DOI: 10.17586/2226-1494-2021-21-4-525-534
УДК 50.41.17

Исполняющая машина автоматных программ

Дмитрий Викторович Дагаев

АО «Русатом — Автоматизированные системы управления», Москва, 115230, Российская Федерация
dvdagaev@oberon.org✉, https://orcid.org/0000-0003-0343-3912

Аннотация
Автоматное программирование основано на построении программных систем как конечных автоматов с явным выделением состояний. Рассмотрены подходы к автоматам как структурам данных и их реализации в разных парадигмах. Оценены требования в части применения автоматных подходов к решению реальных задач. Показано, что для реализации поведения автоматов необходимы подходы, выходящие за рамки объектно-ориентированного программирования. Рассмотрены принципы подстановки вместо механизма наследования объектно-ориентированных программ. Предложено использовать сепарацию кода и данных в рамках парадигмы программирования, управляемого данными. Описана структура данных и взаимодействие с кодом как результат этой сепарации. Рассмотрены механизмы динамической загрузки модулей и представления как данных, кода и схем. Предложена концепция исполняющей машины автоматных программ. Приведено описание ссылочных взаимосвязей в потенциально распределенных системах. Определены требования к реализации исполняющей машины: модульность, использование метаданных, доступ по чтению. Выбран язык программирования Оберон/Компонентный Паскаль и среда разработки BlackBox. Реализован прототип исполняющей машины в виде системы Abpe. Рассмотрено функционирования и построение примеров автоматов в Abpe.

Ключевые слова
автоматное программирование, программирование, управляемое данными, принцип подстановки, Оберон, Компонентный Паскаль

Благодарности
Работа реализована в рамках Международного общественного научно-образовательного проекта Информатика-21.

Ссылка для цитирования: Дагаев Д.В. Исполняющая машина автоматных программ // Научно-технический вестник информационных технологий, механики и оптики. 2021. Т. 21, № 4. С. 525–534. doi: 10.17586/2226-1494-2021-21-4-525-534

An automata-based programming engine

Dmitry V. Dagaev

Rusatom Automated Control Systems JSC, Moscow, 115230, Russian Federation
dvdagaev@oberon.org✉, https://orcid.org/0000-0003-0343-3912

Abstract
Automata-based programming considers program systems construction as finite state machines that demonstrate state-based behavior. This paper analyzes approaches to data structures and their realization in different programming paradigms. The requirements for automata style implementations are estimated for actual tasks. It is shown that automata-based algorithms need approaches beyond the standard object-oriented inheritance and polymorphism. The Liskov substitution principle is considered as an implementation base instead them. Data-oriented programming approach and in particular data and code separation form the backbone of the engine. The work describes the automata data structure and code-data interaction. The dynamically loaded modules and representations of data, code and schemes provide the main building blocks. Automata-based programming engine conception is introduced to clue all above. This engine supports distributed systems referencing. In order to implement an automata-based programming engine, the pilot project has to meet a set of requirements, including modular programming support, extended metadata availability and

code-free read-only data access. Oberon/Component Pascal programming language is therefore chosen, along with a BlackBox Component Builder graphical environment. Automata-based programming engine prototype is implemented as Abpe subsystem for BlackBox. Several example automata-based modules demonstrate functional interacting programs.

Keywords
automata-based programming, data-oriented programming, Liskov substitution principle, Oberon, Component Pascal

Acknowledgements
The study was implemented within the framework of "Informatika-21" (a non-commercial project to promulgate scientific rationality for IT education).

For citation: Dagaev D.V. An automata-based programming engine. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2021, vol. 21, no. 4, pp. 525–534 (in Russian). doi: 10.17586/2226-1494-2021-21-4-525-534

Введение

Парадигма автоматного программирования, описанная в [1], определяет основную его особенность — «программы предлагается создавать так же, как производится автоматизация технологических (и не только) процессов».

Существует множество программных реализаций систем с конечными автоматами на основе различных парадигм программирования, подобные обзоры представлены в статьях «Автоматное программирование»[1] и «Еще об автоматном программировании»[2]. Исторически Switch-технология реализовывалась средствами структурного программирования [2]. Использование общепринятых подходов [3] привело к объектно-ориентированной реализации. Известны работы по реализации конечных автоматов на функциональных языках программирования [4]. В работе [5] исследованы интеграции автоматного программирования с технологиями предикатного и объектно-ориентированного программирования. Графические средства представления в настоящей работе не рассматриваются.

Перечисленные работы демонстрируют множество реализаций систем с конечными автоматами и показывают, как их отобразить средствами структурного, объектно-ориентированного, функционального и предикатного программирования. Такое гибридное представление имеет свои практические применения.

Исходя из основной особенности [1], можно ожидать от автоматных программных модулей структурного подобия электрических и логических схем, которые создаются и функционируют независимо от того, как (последовательно/параллельно/асинхронно) выполняется программный код системы. Можно предположить возможности сборки систем из автоматных модулей и структурной основы для последующей реализации горячей замены и восстановления автоматов при сохранении состояния.

В настоящей работе рассмотрено программное представление данных в автоматной парадигме в сравнении с представлениями объектов в объектно-ориентированном программировании (ООП). Предложены концепция и прототип исполняющей машины автоматных программ.

Требования эргодичности для автоматных программ

Для систем длительной работы эргодичность означает стабильность характеристик со временем [6]. Отсутствие означает деградацию, при которой со временем характеристики системы ухудшаются, что приводит к необходимости вмешательства извне путем перезагрузок, включений резервов и т. д. Предложенная в [6] презумпция неэргодичности означает, что рассматриваемая система предполагается как заведомо не обладающая свойствами, обеспечивающими предсказуемость в части:

— управления стековой и динамической памятью;
— обеспечения реального времени;
— механизма обработки исключительных ситуаций;
— сетевого обмена сообщениями.

В связи с этим требуется доказательство эргодичности, в том числе и путем рассмотрения архитектурных особенностей системы.

Для автоматных программ свойство эргодичности может быть точно доказано. В определениях [2] детерминированный конечный автомат — это семерка: X, Y, Z, δ, ϕ, $y0$, F, где X — конечный алфавит входных символов; Y — конечное множество состояний; Z — конечный алфавит выходных символов; $\delta: X \times Y \to Y$ — функция переходов; $\phi: X \times Y \to Z$ — функция выхода; $y0 \in Y$ — начальное (стартовое) состояние; $F \subset Y$ — множество допускающих состояний. Следовательно, семерка X, Y, Z, δ, ϕ, $y0$, F однозначно определяет динамику изменения состояний системы. Для доказательства эргодичности требуется обеспечить сохранение состояний в памяти автомата и обеспечение подачи значений входов некоторой исполняющей машиной.

Исполняющая машина должна быть рассмотрена однократно, вне зависимости от конкретного автомата, реализующего предписанную функцию.

Для обеспечения сохранения состояний в памяти должна быть определена структура данных автомата как программы.

Попытки реализации автомата средствами ООП [2] не предложили удачной структуры данных, например, был выбран паттерн State [3] с необязательным для

[1] Шалыто А.А. Автоматное программирование [Электронный ресурс]. Режим доступа: https://vk.com/@1077823-vtomatnoe-programmirovanie, свободн., яз. рус. (дата обращения: 03.06.2021).

[2] Шалыто А.А. Еще об автоматном программировании [Электронный ресурс]. Режим доступа: https://vk.com/@1077823-esche-ob-avtomatnom-programmirovanii, свободн., яз. рус. (дата обращения: 03.06.2021).

автоматного программирования использованием динамической памяти. При изменении состояний применено аллокирование динамической памяти, что может привести к эффектам неэргодичности. Это не гарантирует, что память будет выделена (в методе next_state) или корректно удалена в методе delete. При этом распределение памяти представляет собой механизм операционной системы с блокировками и обработками ошибок.

Представления автоматов и объектов

ООП рассматривает объекты, содержащие данные и таблицы виртуальных методов. Каждый метод представляет собой функцию, соответствующую позиции в интерфейсе, и содержит тип и сигнатуру, представляющую численное значение, соответствующее типу. Методы с одинаковыми типами имеют равные сигнатуры. Вызов метода в ООП означает вызов программного кода процедуры, приведенной в соответствие данному (возможно виртуальному) методу.

Известны два подхода для реализации автоматного программирования средствами ООП, при этом в [7] осуществлена в основном надстройка над базовой С-реализацией, а в [8] большая часть программного обеспечения (ПО) описывает логику внутреннего взаимодействия состояний, классов, и т. д. Оба подхода интенсивно используют динамическую память, что не рекомендуется делать для критически важных систем.

Для автоматов [1] принципиальным отличием является выделение управляющих состояний, для каждого из которых определяется программный код, определяющий переход состояний δ и, по возможности, выдачу выходных воздействий ϕ. Для предлагаемой исполняющей машины каждому состоянию автомата соответствуют прототипы:

```
  PROCEDURE StateProg1* (VAR t:
D.Hello; IN e: A.Event; VAR y: A.State);
  PROCEDURE OutputProg1* (VAR t:
D.Hello; y: A.State);
```

В программном коде автомата для каждого управляющего состояния 1 выполняются одна или две из вышеприведенных функций. В данном случае для некоторого автомата Hello, наследуемого от встроенного типа Auto, и состояния 1 определены функции перехода состояний State и выдачи выходных воздействий Output. В целом для автомата Hello возможны два типа функций и две сигнатуры, — количество функций определяется состояниями.

Рис. 1 иллюстрирует разницу в представлении данных для объектов и автоматов. Методы объекта фиксированы по их именам. Для автоматов в зависимости от управляющего состояния State визуально происходит перемещение указателей вверх/вниз. На рисунке для состояния 1 выбраны методы автомата типа Auto StateProg1 и OutputProg1. Обозначение [1] предполагает индексный доступ в таблице методов по номеру состояния. Адреса функций перехода и выхода расположены в этих таблицах.

Следовательно, взаимосвязь данных и методов, предложенная в ООП, не является единственно возможной. Более того, для представления автоматных

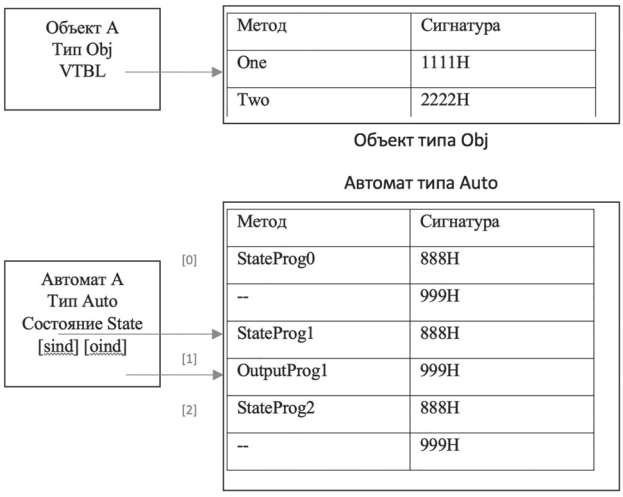

Рис. 1. Представление методов объекта и автомата
Fig. 1. Object and finite-state machine methods representation

программ она явно не оптимальна, так как порождает необходимость внесения в сами методы алгоритмов перехода состояний, что приводит к смешиванию уровней представления системы.

Принцип подстановки вместо наследования

В ООП понятие наследования определяет условие корректности соответствия кода реализации методам интерфейса. Для автоматов точного определения наследования нет. Тем не менее необходимо определить условие корректности соответствия методов автомата состояниям. Принцип подстановки Лисков (Liskov Substitution Principle, LSP) определяет такое соответствие [9]. В нашем случае существенно, что функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

Идея Лисков о «подтипе» базового типа дает определение понятия замещения, когда в программе объекты одного типа могут быть замещены объектами другого типа без каких-либо изменений желательных свойств этой программы. Принцип LSP представляет более жесткую схему, чем наследование, не давая возможности изменения поведения алгоритмов при подстановке. Такое ограничение необходимо при безопасной подстановке на работающих системах, но является обременительным при отладке алгоритмов. В связи с этим будем говорить о принципе минимальной (для ООП — это наследование) и безопасной подстановки.

Для автоматов данные принципы сохраняются, также как и требование создания условий для горячей замены. Принцип безопасной подстановки определяется структурной схемой автомата. Все связи и переходы должны сохраниться, чтобы гарантировать работу нового автомата в условиях старого. Инвариант, сохраняющий

$$X, Y, Z, \delta, \phi, y0, F,$$

— это принцип безопасной подстановки, так как однозначно определяет динамику изменения состояний системы.

Принцип минимальной подстановки определяет условия, при которых подстановка возможна, но не гарантируется динамика функционирования. Принцип минимальной подстановки сохраняет инвариант вида

$$X, Y, Z, I\delta, I\phi, y0, F,$$

где $I\delta$ — набор функций переходов; $I\phi$ — набор функций выходов.

Наборы означают множества индексов функций переходов и выходов, определенных для каждого состояния. При этом ограничений на алгоритмы не накладывается.

Принцип безопасной подстановки и горячая замена в настоящей работе не рассматриваются. Принцип минимальной подстановки требуется и используется ниже для сопоставления методов автомата кодам реализации.

Принципы DOP — программирование, управляемое данными

Автоматное программирование является не единственным случаем, идущим вразрез с практикой ООП. Для автоматных программ характерна первичность данных и состояния, автоматы относятся к технологии программирования, управляемого данными (Data-Oriented Programming, DOP), основные принципы которой изложены в [10].

Программы, управляемые данными, предназначены для построения слабосвязанных (loose-coupling) систем, использующих механизмы восстановления. В [10] представлен список дата-центричных технологий: Application Servers, Enterprise Service Bus, Java Business Integration, Data Distribution Service.

Принципы программирования, управляемого данными, диктуют следующие решения.

1. Предоставление данных и метаданных. Прямо противоположно инкапсуляции в ООП — данные о состоянии ПО предоставляются в рамках прав доступа программных компонентов. Метаданные несут информацию об имени, типе и далее о составе входящих в данный тип вложенных типов. Такое представление должно содержать достаточно информации для чтения и выполнения представленных данных формальной машиной абстрактного исполнителя.
2. Скрытие подробностей кода. DOP скрывает программный код и основывается на сообщениях между компонентами. Интерфейс является некритическим свойством системы. Известна информация о слабосвязанных системах в распределенной конфигурации. В отличие от ООП зависимости между объектами и, как следствие, взаимное влияние отсутствует.
3. Сепарация кода от данных является также принципом DOP.
4. Обработка данных должна осуществляться на основе метаданных реализующей платформы.

Программы, управляемые данными, лучше масштабируются, так как не зависят от интерфейсов, сетевых протоколов, и концепция сети как таковая находится за пределами абстракции данных.

Механизмы метапрограммирования DOP получают гибкий, свойственный интерпретаторам доступ к данным и эффективный вызов откомпилированных функций по номерам. Таким образом, создается гибридный датацентричный подход, имеющий преимущества как интерпретатора, так и компилятора.

На следующем этапе осуществляется детализация представления метаданных и данных автоматной модели.

Автомат как программная единица

Сепарация кода от данных — естественный способ представления ПО. Например, с точки зрения компилятора модуль (файл) программного кода должен иметь информацию, какая его часть является кодом, данными и таблицей внешних символов. Например, структура объектного и исполняемого файла ELF (формат исполняемых и компонуемых файлов, Executable and Linkable Format) [11] имеет соответствующие секции .text, .bss, .symtab. При загрузке .symtab используется для привязки, а секция .bss хранит состояние глобальных переменных программы.

Аналогично предлагаемый автомат как программная единица должен представлять собой структуры данных, метаданные и алгоритмы выполнения. Экземпляры АвтоДата представляют собой модули, содержащие только структуры данных и метаданных. Экземпляры АвтоКод – модули, содержащие только алгоритмы.

Структурное представление в части параметров на рис. 2 — математическое отражение инварианта X, Y, Z, δ, ϕ, $y0$, F. АвтоДата — область данных и представляет собой данные X, Y, Z в виде некоторой четко определенной для данного типа автомата записи (RECORD или struct в нотации языка С). АвтоДата имеет состояние Y и поля X, Z конечного автомата, а также параметры, если для данного типа автомата предполагается параметризация [12].

Также АвтоДата должна иметь статическую метаинформацию для конкретного экземпляра автомата. Чтобы однозначно его идентифицировать требуется кортеж из номеров #Модуль.#Тип.#Элемент. Номера необходимы для прямого доступа к таблицам модулей, типов и элементов. Таким образом, ссылка на экземпляр автомата определяется в виде #Модуль.#Тип.#Элемент. Выходные контакты Z автомата имеют формат ссылки #Контакт. Входные элементы автомата должны быть привязаны к выходным контактам другого автомата, формат ссылок #Элемент.#Контакт.

Данные в модулях АвтоДата должны быть доступны только на чтение из внешних систем. При этом АвтоКод должен иметь полный доступ к чтению и записи относящихся к нему данных.

Программный код модуля АвтоКод представляет собой исполняемый код, полученный из алгоритмов конечного автомата. Входы, выходы и состояния существуют отдельно от программного кода в модуле АвтоДата и даже при его отсутствии. Например, при установке начального (стартового) состояния АвтоКод не используется и может стартовать позднее. Но программный код не может существовать без программных

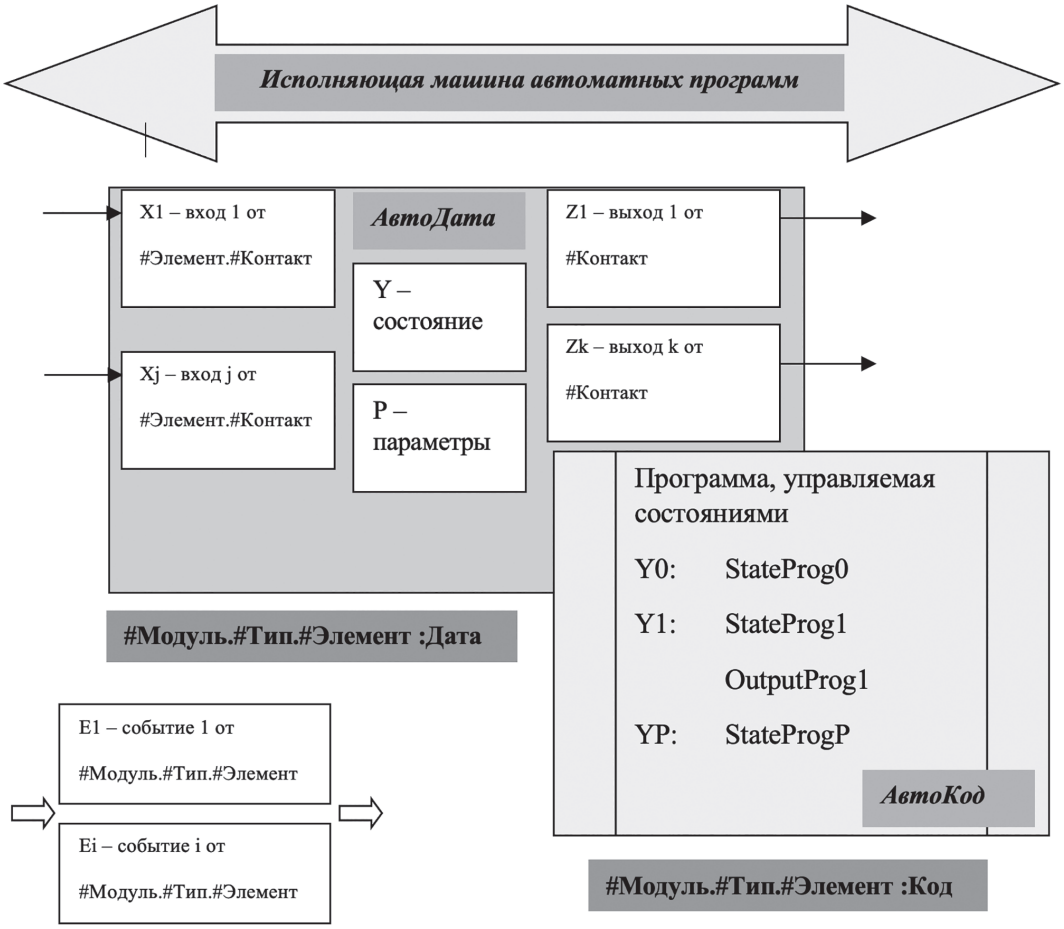

Рис. 2. Структурное представление программы конечного автомата
Fig. 2. Finite-state machine program data structure

данных. Важно, что поведение АвтоКода основано на «явном выделении состояний» [1], которые размещены в структуре АвтоДата. Для каждого состояния вызывается предписанный этому состоянию программный блок в АвтоКоде. На рис. 2 состоянию Y1 соответствуют программные блоки StateProg1, OutputProg1 и далее. АвтоКод выполняется согласно алгоритмам δ-функции переходов в первом и ϕ-функции выхода во втором случае.

В итоге АвтоКод представляет собой набор функций перехода и выхода, а управляющая конструкция switch [2] перенесена в исполняющую машину.

Сепарация АвтоКода от АвтоДата соответствует принципу разделения ответственности Дейкстры [13]. Для автомата характерна первичность данных и состояния, а не первичность кода и интерфейса. В автоматном подходе данные доступны на чтение, при этом обеспечивается их обновление только от одного, активного в текущий момент времени, элемента кода. В автоматном программировании АвтоДата является спецификацией, определяющей точки внешнего взаимодействия. Как упоминалось выше, данный подход идет вразрез с существующей практикой ООП, когда внешние взаимодействия задаются интерфейсом, определяющим поведение:

— используем АвтоДата <X, Y, Z, E, F>;
— ~~не используем Интерфейс~~ (АвтоКод).

Принцип минимальной подстановки должен быть обеспечен установкой в АвтоДата множества допускающих состояний *F* (для функций перехода и выхода) и проверкой полного соответствия множества *F* процедурам АвтоКод средствами рантайма.

АвтоДата и АвтоКод должны быть привязаны к внешней среде и абстрактному исполнителю. Привязка к внешней среде означает соединение <X, Y, Z, E, F> автомата с внешними данными. В свою очередь, привязка АвтоКода определяет, как будет исполняться алгоритм конечного автомата.

Ссылка на экземпляр процедуры АвтоКода также представляет собой кортеж из номеров #Модуль.#Тип.#Элемент. При этом номер элемента относится к номеру процедуры, а тип соответствует типу процедуры. Ссылка на процедуру обеспечивает индексный доступ к коду процедуры загруженного модуля АвтоКода.

Исполняющая машина автоматных программ

Исполняющая машина автоматных программ (ИМАП) представляет собой интерфейс абстрактного исполнителя и реализацию работы на основе представленных данных и метаданных. Реализации ИМАП могут различаться в зависимости от предъявляемых требований: распределенность или нет, многопоточность или нет, реальное время или нет. Принцип разделения

ответственности предписывает работу ИМАП через интерфейс, скрывая детали реализации.

Взаимодействие автомата с внешней средой постулирует наличие следующих условий:

— обмен сообщениями осуществляется по ссылкам на абонентов, которые могут находиться вне зоны прямого контакта, т. е. быть слабосвязанными;
— привязка входов к выходам осуществляется в рамках одной схемы/модуля, включающей в себя несколько элементов;
— взаимодействие с внешним миром осуществляется через интерфейс ИМАП.

Основные методы интерфейса ИМАП представлены в таблице.

Элементы имеют доступ к ссылочной информации, но при этом им не доступны объекты, с которыми осуществляется взаимодействие. Такой подход позволяет строить слабосвязанные и распределенные системы.

С точки зрения отдельного автомата доступна следующая функциональность.

1. Выполнение циклической функции перехода **StateProg1*** (VAR t: D.Hello; IN e: A.Event; VAR y: A.State) для данного состояния. На вход процедуры поступает событие (возможно, непустое). Процедура обновляет данные автомата Hello и возвращает новое значение состояния.
2. Выполнение циклической функции выхода **OutputProg1*** (VAR t: D.Hello; y: A.State) при смене состояния. На вход процедуры подается значение нового состояния перед обновлением.
3. Посылка сообщений другим автоматам внутри пп. 1 и 2 посредством интерфейса ИМАП.
4. Установка выходных значений переменных Z.
5. Обновление данных автомата Hello, не являющихся управляющими состояниями (например, получение и запоминание значений ссылок на другие автоматы в отдельных переменных с последующей посылкой сообщений по этим ссылкам).

В свою очередь ИМАП имеет доступ к внутренним таблицам модулей, типов и элементов, а также, при необходимости, к средствам взаимодействия с другими, потенциально удаленными процессами. ИМАП осуществляет получение по ссылке адреса конкретного объекта или его типа.

Реализация ИМАП должна иметь возможность работы с динамически загружаемыми и выгружаемыми модулями и с содержательной метаинформацией о типах и объектах программного модуля. Некоторые языки программирования имеют поддержку формирования подобной информации на уровне компилятора.

Таблица. Интерфейс взаимодействия исполняющей машины автоматных программ

Table. Automata-based programming engine interface

Функциональное назначение	Методы интерфейса
Доступ к ссылочной информации	GetRef
Связь входов с выходами	Link
Передача различных сообщений	SendSimpleMsg SendParamMsg SendErrorMsg SendOnTouch

Получение адресов по ссылкам осуществляется только при наличии метаинформации, например:

«ModuleAddress := M.modules[#Module] – массив загруженных динамически модулей выбирает по индексу #Module;

ElementAddress := ModuleAddress.obj[#Element]. Addr – метамассив элементов выбирает по индексу #Element.»

Реализация ИМАП основана на привязке и периодическом обходе обозримых модулей автоматных схем, содержащих элементы АвтоДата. Модули АвтоКода привязываются к модулям АвтоДата установкой массива индексов функций для каждого состояния. Привязка также устанавливает индексы модулей, типов и элементов данных.

Периодический обход элементов автоматных схем осуществляется по ранее установленным метаданным, как показано на рис. 3.

Для каждого элемента вычисляется индекс процедуры, соответствующий состоянию s := sind[y]. Массив индексов функций создан на этапе привязки. Вызов функции перехода δ[s] обращается к соответствующему коду функции StateProg. Если новое состояние изменяется ynew != y, то вычисляется индекс o := oind[y], после чего вызывается функция выхода ϕ[o], и вызов обращается к соответствующему коду функции OutputProg. По окончании любого из вариантов происходит смена состояния y := ynew.

Далее ИМАП переходит к обработке следующего элемента схемы. Когда все элементы схемы обработаны, ИМАП переходит к обработке следующей зарегистрированной схемы.

Сообщения для других автоматов, поступающие в ИМАП, имеют адрес отправителя и адрес получателя. На усмотрение ИМАП эти сообщения могут отправляться сразу адресату с вызовом функций δ[s] и ϕ[o], либо буферизироваться для дальнейшей передачи локальному или удаленному автомату. Это вопрос к конфигурации конкретной реализации ИМАП.

Получение ссылок на символьные имена и привязка входов к выходам обрабатываются ИМАП без буферизаций и задержек.

Программные средства реализации исполняющей машины автоматных программ

Для реализации ИМАП как программной системы программные средства должны обеспечить:

— модульное программирование с поддержкой компиляции, загрузки, выполнения;
— наличие метаданных на модули, типы, элементы, функции;
— механизмы доступа к переменным только на чтение;
— средства реализации исполняемого кода как в графических, так и во встроенных системах с возможностью работы в системах реального времени.

Выбор средств приводит к модульным языкам программирования, таким как Оберон, Модула [14], где модуль представляет собой единицу программного

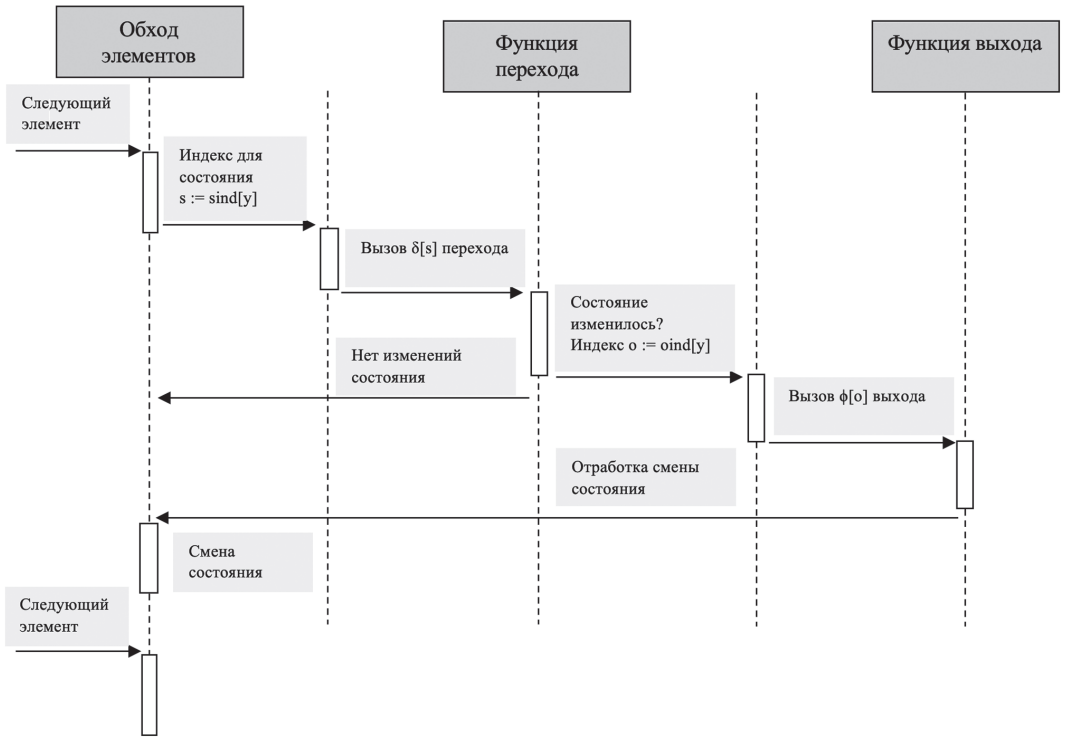

Рис. 3. Реализация периодического обхода исполняющей машиной автоматных программ

Fig. 3. Periodic traversal algorithm implemented by the automata-based programming engine

кода, компиляции и динамической загрузки. Системы на модульных языках используют в качестве компактных приложений повышенной надежности, вплоть до критически важных систем.

Для реализации ИМАП была выбрана версия Оберона — Компонентный Паскаль (КП) со средой программирования BlackBox 1.7.2[1]. Язык Оберон позволяет строить как структурные, так и объектно-ориентированные программы. Автор располагает большим опытом разработки в обеих парадигмах. Данная среда соответствует всем вышеуказанным требованиям к ПО.

ИМАП реализована в качестве прототипа (Automata-based programming engine, Abpe) и добавлена в коллекцию Helmut Zinn CPC[2] для BlackBox. И среда BlackBox, и Abpe являются свободно-распространяемым ПО, доступным в Интернет. Как принято в среде BlackBox, пакет должен быть разархивирован средствами предварительно установленной среды BlackBox, скомпилирован и запущен по указаниям инструкции Quick-Start.

Здесь и далее пакет Abpe означает программную реализацию ИМАП в указанном прототипе.

[1] BlackBox Framework Center, Среда программирования BlackBox [Электронный ресурс]. Режим доступа: https://blackboxframework.org/index.php?cID=home-ru,ru, свободный, яз. рус. (дата обращения: 03.06.2021).

[2] Helmut Zinn Component Pascal Collection, Abpe – Automata-Based Programming engine [Электронный ресурс]. Режим доступа: http://zinnamturm.eu/downloadsAC.htm#Abpe, свободный, яз. англ. (дата обращения: 03.06.2021).

Пример Hello для Abpe

Рассмотрим использование наиболее простого примера — программы Hello (Hello World для автоматного программирования). Программа Hello запускается нажатием нижеприведенных команд в среде BlackBox.

```
AbpeRunner.AddPtr('AbpeObxHelloData',
'Hello', 'Greetings', 'AbpeObxHelloCode')
```

— добавить схему Greetings из одного элемента Hello модуля AbpeObxHelloData, привязать к коду AbpeObxHelloCode.

```
AbpeRunner.InitSchema('Greetings',
'en')
```

— инициализировать схему Greetings в состояние 0 и стартовать с посылкой начального сообщения Begin('en') (английский язык).

После этого автомат начинает печатать приветствие «Use it and enjoy» по буквам с паузой.

При старте можно запустить русское приветствие (также есть немецкое, французское, польское, испанское):

```
AbpeRunner.InitSchema('Greetings',
'ru')
```

Автомат напечатает фразу «Приятного использования» по буквам с паузой.

Граф переходов автомата Hello приведен на рис. 4.

На сером фоне отмечены события. Событие Begin устанавливает строку приветствия сообразно выбранному языку. В состоянии Put1 происходит посылка сообщений с текущей буквой msg[ind]. Если текущая буква — нулевой символ конца строки, автомат переходит в состояние Finish2.

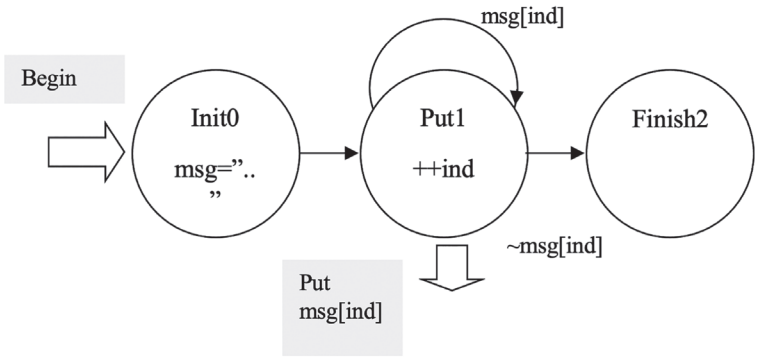

Рис. 4. Граф переходов Hello
Fig. 4. State transition graph for Hello

АвтоДата примера Hello для Abpe

Модуль AbpeObxHelloData представляет собой объявления АвтоДата в файле следующего вида.

```
MODULE AbpeObxHelloData;
IMPORT A := AbpeAuto;
TYPE
    Hello* = RECORD (A.Data)
        msg*: ARRAY A.N_STR OF CHAR;
        ind*: INTEGER;
    END;

    HelloStates_0_1* = RECORD (A.OnlyStates) END;
    HelloOutputs_1_2* = RECORD (A.OnlyOutputs) END;
END AbpeObxHelloData.
```

Тип Hello является автоматом, т. е. расширением A.Data, в котором определены параметры msg (текст приветствия) и индекс ind. Тип HelloStates_0_1 определяет множество допускающих состояний F для функции перехода. Принцип минимальной подстановки требует в реализации АвтоКода наличия только функций перехода для состояний 0 и 1. Тип HelloOutputs определяет множество допускающих состояний для функции выхода. Принцип минимальной подстановки требует в реализации АвтоКода наличия только функций выхода для состояний 1 и 2. В Abpe используется до 32 состояний, из них 0 — начальное, а 2 — конечное. Для каждого состояния выхода должна быть также функция перехода, исключение составляет конечное состояние 2, из которого переход не требуется.

При создании схемы Greetings модуль AbpeObxHelloData загружается в случае, если не был загружен ранее.

АвтоКод примера Hello для Abpe

Модуль AbpeObxHelloCode представляет собой процедуры реализации АвтоКода в файле следующего вида.

```
MODULE AbpeObxHelloCode;
IMPORT T := AbpeTypes, A := AbpeAuto, D :=
AbpeObxHelloData;

(* ========== Init Hello message and Put char-by-char
========== *)
    (* ST_BEGIN = 0 *)
    PROCEDURE Init0* (VAR t: D.Hello; IN e: A.Event; VAR
y: A.State);
    BEGIN
        WITH e: A.Begin DO
            IF e.msg = “en” THEN
                t.msg := “Use it and enjoy”;
            ELSIF e.msg = “sp” THEN
                t.msg := “Ўъsalos y disfrъtalos”;
            ELSIF e.msg = “fr” THEN
                t.msg := “Bonne utilisation”;
            ELSIF e.msg = “ru” THEN
                t.msg := «Приятного использования»;
            ELSIF e.msg = «pl» THEN
                t.msg := “Powodzenia”;
            ELSIF e.msg = “ge” THEN
                t.msg := “Viel SpaЯ”;
            ELSE
                t.msg := “”;
                t.SendErrorMsg(e.from, “wrong begin init ‘”
+ t.msg + “’”, A.E_ERROR);
            END;
            t.ind := 0;
            y := 1 (* RUN *)
        ELSE
        END
    END Init0;

    (* ST_RUN = 1 *)
    PROCEDURE Put1* (VAR t: D.Hello; IN e: A.Event; VAR
y: A.State);
        VAR m2: ARRAY 2 OF CHAR;
    BEGIN
        WITH e: A.End DO
            y := 2;          (* END *)
        ELSE
            IF t.msg[t.ind] # 0X THEN
                m2[0] := t.msg[t.ind];
                m2[1] := 0X;
                t.SendSimpleMsg(e.from, m2);
                INC(t.ind);
            ELSE
                t.SendSimpleMsg(e.from, “\n”);
                y := 2;      (* END *)
            END;
        END
    END Put1;

    (* ========== Started/Finished outputs ========== *)
    (* ST_RUN = 1 *)
    PROCEDURE Started1* (VAR t: D.Hello; y: A.State);
    BEGIN
        IF A.traceLevel >= A.TRACE_LEVEL_NORMAL THEN
            t.SendParamMsg(T.sysRef, “%r started\n”, t.ref.
rec.typfp, t.ref.rec.ind, 0, 0)
        END
    END Started1;

    (* ST_END = 2 *)
    PROCEDURE Finished2* (VAR t: D.Hello; y: A.State);
    BEGIN
        IF A.traceLevel >= A.TRACE_LEVEL_NORMAL THEN
            t.SendParamMsg(T.sysRef, “%r finished\n”, t.ref.
rec.typfp, t.ref.rec.ind, 0, 0)
        END
    END Finished2;

END AbpeObxHelloCode.
```

АвтоКод Hello использует типы объектов из AbpeObxHelloData, которым сопоставляются функции перехода и выхода. Тип D.HelloStates_0_1 предписывает наличие функций перехода для состояний 0 и 1. Это — функции Init0 и Put1 соответственно. Имена функций перехода и выхода в Abpe должны оканчиваться номером состояния. Тип D.HelloOutputs_1_2 предписывает наличие функций выхода для состояний 1 и 2. Это — функции Started1 и Finished2.

Функция Init0 вызывается для события Begin, устанавливает текст приветствия и возвращает следующее состояние 1. Функция Put1 посылает сообщения для ненулевого символа и предлагает переход в состояние 1 для нулевого. Функции Started1 и Finished2 в данном примере используются только для печати служебных сообщений в режиме трассировки.

Помимо рассмотренного примера, автором также реализованы приемник и передатчик азбукой Морзе, и счетчик событий, с автоматом скользящего усреднения времени между событиями.

Заключение

В работе рассмотрены способы реализации программ в парадигме автоматного программирования, где во главу угла поставлены управляющие состояния. Показано, что в автоматном программировании имеет смысл использовать новые подходы вместо стандартного минимального набора средств объектно-ориентированного программирования.

На основе подхода «с явным выделением состояний» автором предложена концепция исполняющей машины автоматных программ, которая использует принцип программирования, управляемого данными.

Предложена реализация на языке Оберон/Компонентный Паскаль в системе BlackBox.

Объяснены механизмы работы примеров реализации.

Преимуществами данного подхода являются следующие возможности:

— условия реализации горячей замены в режиме 24 × 7;
— сборка схем из элементов автоданных;
— работа в распределенных, слабосвязанных и многопроцессных средах.

Литература

1. Шалыто А.А. Парадигма автоматного программирования // Научно-технический вестник Санкт-Петербургского государственного университета информационных технологий, механики и оптики. 2008. № 53. С. 3–23.
2. Поликарпова Н.И., Шалыто А.А. Автоматное программирование. СПб., 2008. 167 с.
3. Гамма Э., Хэлм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001. 366 р.
4. Малаховски Я.М. Реализация конечных автоматов на функциональных языках программирования: дипломная работа / Санкт-Петербургский государственный университет информационных технологий, механики и оптики, Кафедра компьютерных технологий СПб., 2009. 68 с.
5. Шелехов В.И. Язык и технология автоматного программирования // Программная инженерия. 2014. № 4. С. 3–15.
6. Дагаев Д.В. О разработке Оберон-системы с заданными свойствами эргодичности // Труды Института системного программирования РАН. 2020. Т. 32. № 6. С. 67–78. https://doi.org/10.15514/ISPRAS-2020-32(6)-5
7. Шопырин Д.Г., Шалыто А.А. Объектно-ориентированные подход к автоматному программированию // Информационно-управляющие системы. 2003. № 5. С. 29–39.
8. Шамгунов Н.Н., Корнеев Г.А., Шалыто А.А. State Machine – новый паттерн объектно-ориентированного проектирования // Информационно-управляющие системы. 2004. № 5. С. 13–25.
9. Мартин Р. Чистая архитектура: искусство разработки программного обеспечения. СПб.: Питер, 2018. 351 р.
10. Joshi R. Data-Oriented Architecture: A Loosely-Coupled Real-Time SOA. Real-Time Innovations, Inc., 2007 August. 54 p.
11. Робинс А. Linux: программирование в примерах. М.: Кудиц-Пресс, 2008. 655 р.
12. Harel D., Pnueli A. On the development of reactive systems // Logic and Models of Concurrent Systems. Springer Verlag, 1985. P. 477–498. (NATO ASI Series, Series F: Computer and Systems Sciences, vol. 13). https://doi.org/10.1007/978-3-642-82453-1_17
13. Dijkstra E.W. On the role of scientific thought // Selected Writings on Computing: A Personal Perspective. New York, NY, USA: Springer-Verlag, 1982. P. 60–66. https://doi.org/10.1007/978-1-4612-5695-3_12
14. Reiser M., Wirth N. Programming in Oberon: Steps Beyond Pascal and Modula. ACM Press, 1992. 320 p.

References

1. Shalyto A. Automata-based programming paradigm. *Scientific and Technical Bulletin of St. Petersburg State University of Information Technologies, Mechanics and Optics*, 2008, no. 53, pp. 3–23. (in Russian)
2. Polikarpova N.I., Shalyto A.A. *Automata-Based Programming*. St. Petersburg, 2008, 167 p. (in Russian)
3. Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns. Elements of Reusable Object-Oriented Software*. Pearson Education, 1994, 395 p.
4. Malakhovskyi Ya.M. *Automata Implementation in Functional Programming Languages. Thesis*. St. Petersburg, ITMO University, 2009, 68 p. (in Russian)
5. Shelekhov V.I. Automata-based software engineering: the language and development methods. *Software Engineering*, 2014, № 4. C. 3–15. (in Russian)
6. Dagaev D.V. Towards developing of Oberon system with specific requirements of ergodicity. *Proceedings of ISP RAS*, 2020, vol. 32, no. 6, pp. 67–78. (in Russian). https://doi.org/10.15514/ISPRAS-2020-32(6)-5
7. Shalyto A.A., Shopyrin D.G. Object-oriented approach to a automata programming. *Information and Control Systems*, 2003, no. 5, pp. 29–39. (in Russian)
8. Shamgunov N.N., Korneev G.A., Shalyto A.A. State machine - a new design pattern. *Information and Control Systems*, 2004, no. 5, pp. 13–25. (in Russian)
9. Martin R.C. Clean Architecture: *A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017, 432 p.
10. Joshi R. *Data-Oriented Architecture: A Loosely-Coupled Real-Time SOA*. Real-Time Innovations, Inc., 2007 August, 54 p.
11. Robbins A. *Linux Programming by Example*. Prentice Hall Professional Technical Reference, 2004, 687 p.
12. Harel D., Pnueli A. On the development of reactive systems. *Logic and Models of Concurrent Systems*. Springer Verlag, 1985, pp. 477–498. NATO ASI Series, Series F: Computer and Systems Sciences, vol. 13. https://doi.org/10.1007/978-3-642-82453-1_17
13. Dijkstra E.W. On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*. New York, NY, USA, Springer-Verlag, 1982, pp. 60–66. https://doi.org/10.1007/978-1-4612-5695-3_12
14. Reiser M., Wirth N. *Programming in Oberon: Steps Beyond Pascal and Modula*. ACM Press, 1992, 320 p.

Автор

Дагаев Дмитрий Викторович — главный эксперт, АО «Русатом — Автоматизированные системы управления», Москва, 115230, Российская Федерация, https://orcid.org/0000-0003-0343-3912, dvdagaev@oberon.org

Author

Dmitry V. Dagaev — Chief Expert, Rusatom Automated Control Systems JSC, Moscow, 115230, Russian Federation, https://orcid.org/0000-0003-0343-3912, dvdagaev@oberon.org

Статья поступила в редакцию 17.05.2021
Одобрена после рецензирования 03.06.2021
Принята к печати 25.07.2021

Received 17.05.2021
Approved after reviewing 03.06.2021
Accepted 25.07.2021