ТЕОРЕТИЧЕСКИЙ И ПРИКЛАДНОЙ НАУЧНО-ТЕХНИЧЕСКИЙ ЖУРНАЛ

Издается с сентября 2010 г. DOI 10.17587/issn.2220-3397 ISSN 2220-3397

Журнал издается при поддержке Отделения математических наук РАН, Отделения нанотехнологий и информационных технологий РАН, МГУ имени М.В. Ломоносова, МГТУ имени Н.Э. Баумана

СОДЕРЖАНИЕ

Журнал зарегистрирован в Федеральной службе по надзору в сфере связи, информационных технологий и массовых коммуникаций. Свидетельство о регистрации ПИ № ФС77-38590 от 24 декабря 2009 г.

Журнал распространяется по подписке, которую можно оформить в любом почтовом отделении (индексы: по каталогу агентства "Роспечать" — **22765**, по Объединенному каталогу "Пресса России" — **39795**) или непосредственно в редакции.
Тел.: (499) 269-53-97. Факс: (499) 269-55-10.
Http://novtex.ru/prin/rus E-mail: prin@novtex.ru
Журнал включен в систему Российского индекса научного цитирования.
Журнал входит в Перечень научных журналов, в которых по рекомендации ВАК РФ должны быть опубликованы научные результаты диссертаций на соискание ученой степени доктора и кандидата наук.

THEORETICAL AND APPLIED SCIENTIFIC AND TECHNICAL JOURNAL

SOFTWARE ENGINEERING

PROGRAMMNAYA INGENERIA

Vol. 7
N 12
2016

Published since September 2010 **DOI 10.17587/issn.2220-3397** **ISSN 2220-3397**

CONTENTS

Information about the jornal is available online at:
http://novtex.ru/prin/eng e-mail:prin@novtex.ru

УДК 519.683 DOI: 10.17587/prin.7.531-538

В. И. Шелехов, канд. техн. наук, зав. лаб., e-mail: vshel@iis.nsk.su,
Институт систем информатики им. А. П. Ершова, г. Новосибирск

Классификация программ, ориентированная на технологию программирования

Рассматривается классификация программ по их внутренней организации, определяющей интерфейс с внешним окружением, форму спецификации программы и другие особенности. Классификация ориентирована на разработку адекватной технологии для каждого класса программ. Определены три класса программ: класс программ-функций, класс реактивных систем и класс языковых процессоров.

Ключевые слова: *программа, технология программирования, определение требований, спецификация программы, реактивная система, формальная семантика*

История развития программной инженерии демонстрирует калейдоскопическую смену большого числа моделей, методов и инструментальных средств. Разрабатываемые технологии программирования в основном ориентированы на вполне определенные ограниченные классы программ. Общеизвестны такие классы программ, как, например, реактивные системы, системы реального времени, вероятностные автоматы.

Даже универсальная технология программирования не является одинаково успешной для всех программ. Поэтому полезно определить класс программ, соответствующих каждой технологии. Актуальной задачей является построение полной системы классов программ, где каждый класс определяется своей особенной внутренней организацией программ, предопределяющей технологию программирования.

В мировой литературе не обнаружено попыток построить полную систему классов программ. В работах встречаются лишь общие поверхностные определения классов, как правило, без устоявшихся общепринятых математических моделей. Известны другие классификации: классификации программ по их назначению, классификация языков программирования и классификация парадигм программирования. Все они имеют мало общего с требуемой классификацией программ, учитывающей внутреннюю организацию программ.

Попытка создания общего теоретического базиса для разных классов и парадигм предпринята Хоаром и другими учеными в рамках серии работ по *унифицированным теориям программирования* (*Unifying theories of programming*, UTP) [1]. Для каждого класса программ определяется минимальное ядро языка программирования; далее разрабатываются формальная семантика ядра и методы формальной верификации. Эти работы ограничены созданием общей математической теории программирования и пока не получили продолжения в создании новых инструментов и технологий программирования.

В данном контексте следует также отметить инициативу SEMAT (Software Engineering Method and Theory) [2], объявленную в 2009 г. и получившую поддержку большого числа ученых, научных организаций и софтверных фирм. Цель инициативы — создание единой теории в программной инженерии, которая стала бы направляющей основой ее дальнейшего развития. Отмечался разрыв между научными исследованиями и практикой программирования. При отсутствии общего теоретического базиса современное развитие программной инженерии оказалось в зависимости от рекламного продавливания своих продуктов крупными софтверными корпорациями. Были сформулированы основные положения и цели инициативы, которые послужили основой для ее обсуждения на нескольких научных конференциях, в том числе с применением мозгового штурма. Через четыре года работы в рамках инициативы были приостановлены.

Предикатное программирование [3—6] возникло как отрицание императивного программирования. Класс предикатных программ определен как класс *программ-функций*, соответствующий моделям "программа есть функция" или "программа — это предикат", аналогично UTP [1]. Класс реактивных систем реализован в другой парадигме — в автоматном программировании [7]. Здесь используется автоматная модель программы в виде гиперграфа, являющаяся продолжением аппарата гиперфункций [3, 6] в предикатном программировании.

Интересен опыт распространения функционального программирования на класс реактивных систем, в частности, с использованием языков Haskell и Erlang, на базе акторской модели [8], допускающей взаимодействие между параллельными подпрограммами реактивной системы только через прием и посылку сообщений. Отметим, что акторская модель покрывает лишь часть класса реактивных систем — не допускаются разделяемые переменные, которые

обязательны, например, в серии протоколов взаимного исключения.

Работы по классификации программ начались с попытки найти точную границу между классом реактивных систем и классом программ-функций. Обнаружилось, что эти два класса не покрывают всего множества программ, в частности, программ трансляторов с языков программирования.

Настоящая работа по классификации программ — это одна из попыток построить единый фундамент в программной инженерии. Определены три класса программ: класс программ-функций, класс реактивных систем и класс языковых процессоров. Данная работа также может быть использована для обучения студентов и аспирантов, специализирующихся в области программной инженерии. Описанию трех классов программ предшествуют определения базисных понятий программы, требований и спецификации программы.

Общее понятие программы

В теоретическом и общеметодологическом плане понятие *программы* является предметом *информатики* (*Computer science*), в производственном — предметом *программной инженерии* (*Software engineering*) — научно-инженерной дисциплины, определяющей методы и инструментальные средства разработки программ. Технология программирования часто используется как эквивалент программной инженерии, хотя содержание этих понятий не совпадает.

Понятие программы появилось вместе с первыми компьютерами в 1940-х гг.[1] Развитие средств вычислений сопровождалось все большей степенью *автоматизации вычислений*. Потребность в проведении сложных и длительных расчетов привела человека к необходимости создания и совершенствования механических устройств и машин для облегчения процесса вычисления. С древних времен известно устройство, которое в России называют счеты. Счетные машины и арифмометры, создаваемые в XVII—XIX вв., автоматизировали выполнение четырех арифметических действий. В середине XX столетия актуальные потребности науки и практики привели к созданию новой отрасли техники — конструированию и производству счетно-решающих устройств, т. е. приборов и машин для решения математических задач [9]. Применение счетно-решающих устройств становится массовым: используют их для быстрых расчетов в боевой обстановке, в навигации, в автоматизированном управлении сложными агрегатами и т. д. Появление универсальных электронно-вычислительных машин (ЭВМ), умеющих автоматически выполнять программу, построенную для произвольного алгоритма, стало неизбежным.

[1] Первые программы писала математик Ада Лавлейс, дочь поэта лорда Байрона, для спроектированной, но не реализованной вычислительной машины Чарльза Беббиджа в 1830-х гг.

Промежуточное состояние автоматизации вычислений алгоритма реализуется современными калькуляторами, сравнимыми по своим возможностям со счетно-решающими устройствами 1940-х гг. На очередном шаге своей работы калькулятор либо принимает данные от пользователя, либо в автоматическом режиме вычисляет одну из операций. Более развитые калькуляторы позволяют сохранять промежуточные значения вычислений, именовать отдельные значения и массивы значений и использовать эти имена в дальнейших вычислениях. Таким образом, эти калькуляторы могут хранить в своей памяти значения нескольких переменных. Подчеркнем, что в сравнении с ручным счетом калькулятор автоматизирует вычисление только одной операции на каждом шаге, а между шагами требуются действия человека. С помощью калькулятора можно провести расчеты в принципе по любому алгоритму. Если мы захотим исключить действия человека между шагами и ограничить его действия лишь вводом начальных данных, то потребуется каким-то способом задать всю совокупность операций алгоритма и указать порядок вычисления операций. Необходимым инструментом для этого является программа.

Программа есть алгоритм, реализованный таким способом и в такой форме, что вычисление алгоритма проводится автоматически. По способу реализации различают аппаратно реализованные программы (например, в виде интегральной схемы) и программы на языках программирования. *Автоматическая вычислимость* является неотъемлемым свойством программы. Приведенное определение уточняет краткое определение в энциклопедиях: "программа есть описание алгоритма решения задачи, заданное на языке программирования" [10].

Понятия алгоритма и программы не тождественны. Понятие *алгоритма* известно давно и считается хорошо изученным. В математике существуют следующие формализации понятия алгоритма: машина Тьюринга [11], частично рекурсивные функции, нормальный алгорифм Маркова [12] и каноническая система Поста [13]. Используется также другая трактовка понятия алгоритма как набора алгоритмических предписаний, применяемых в различных сферах человеческой деятельности.

Окружение программы — программные и аппаратные компоненты, непосредственно взаимодействующие с программой при ее исполнении и реализующие входные и выходные информационные потоки программы.

Требования

Требование — утверждение относительно одного из свойств, которое должна иметь создаваемая *система*. В качестве системы может выступать все что угодно: токарный станок, лекарственный препарат, робот, космический аппарат, программная система и многое другое. Определение требований и методы работы с ними являются предметом *инженерии требований*, примыкающей к другой научно-инженерной

дисциплине — *системной инженерии* [14], определяющей методы проектирования сложных систем.

Инженерия требований для программных систем является независимой дисциплиной, учитывающей специфику программной инженерии. Эффективные методы построения требований для программных систем зафиксированы стандартом IEEE 830—1998 [15, 16]. В соответствии со стандартом требования должны быть: корректными, однозначными, полными, согласованными, ранжированными по значимости и обязательности, проверяемыми, модифицируемыми, хорошо организованными для анализа.

Виды требований. Требования делят на два класса: функциональные и нефункциональные. *Функциональные требования* определяют поведение программной системы. Есть разные подходы к описанию функциональных требований. Наиболее популярной формой являются *сценарии использования* (*use case*): на каждое событие в окружении программы определяется ее реакция с указанием всех вариантов. Виды *нефункциональных требований* следующие: требования к окружению программы, требования к оборудованию, ограничения реализации, требования баз данных, стандарты. К нефункциональным требованиям также относятся характеристики программы: надежность, доступность, защищенность, сопровождаемость, переносимость и др.

Языками спецификации требований являются: естественный язык, английский (80 % случаев), формализованное подмножество естественного языка с использованием аппарата онтологий (15 %) и формальный язык (5 %) [17]. Формальными языками являются: Statechart [18] SDL [19], UML и др. Большинство из них являются графическими. Формальными языками спецификации требований являются также общеизвестные универсальные языки спецификаций: VDM, Z, Event-B [20] и др. Семейство темпоральных языков спецификаций также используется для спецификации требований программных систем; наиболее популярным является язык LTL.

В мировой практике инженерия требований применяется преимущественно для больших информационных и телекоммуникационных систем. Разработку требований осуществляют специалисты — *инженеры требований*, составляющие иногда половину персонала.

Спецификация программы

Спецификация программы — это описание программы, в принципе любое, достаточно полное, чтобы на его основе можно было построить алгоритм программы. При таком определении код программы является ее спецификацией, что, разумеется, неверно. В действительности, программа и ее спецификация противопоставляются. Спецификация декларативна: она определяет, что вычисляет программа, а не как реализуется процесс исполнения. Более точное определение следующее. *Спецификация программы* — точное, полное и однозначное описание преобразования информации, реализуемого программой, т. е. описание зависимости результатов исполнения программы от исходных данных.

Спецификация определяется как результат намерения создать программу. Поэтому спецификация первична по отношению к программе. Однако она не всегда фиксируется в документальном виде, а если фиксируется, то может быть неполной, неточной, устаревшей.

Требование корректности программы. *Программа должна соответствовать спецификации*: набор утверждений, составляющих спецификацию программы, должен быть истинным для значений результатов исполнения программы и входных данных при любом исполнении программы. *Верификации программы* — проверка корректности программы относительно спецификации. Используются различные методы верификации программ: тестирование, экспертиза кода, статический анализ, формальные методы верификации [21].

Спецификации часто записывают на естественном языке, расширенном набором специальных обозначений предметной области. Спецификация, записанная на строгом формальном языке спецификации, является *формальной*. Она определяет математическое описание реализуемой программы.

Формальные методы (*Formal methods*) базируются на формальной спецификации программ. Формальные методы включают:

- тестирование на базе формальной спецификации;
- статическую верификацию (*software model checking*);
- проверку на моделях (*model checking*);
- дедуктивную верификацию;
- программный синтез.

Спецификация программы конструируется с учетом требований к программе и может содержать описание некоторых требований.

Задача классификации программ

Методы программной инженерии, доказавшие свою эффективность для многих программ, не всегда успешно применимы для всех программ. Причина здесь в различиях архитектур программ. Подобная ситуация часто возникает в контексте применения формальных методов. Это ставит задачу *классификации программ*, т. е. построения системы классов программ, и разработку адекватных методов для каждого класса.

Обычно рассматривают классификации по назначению программ. Здесь же определяется классификация программ по их внутренней организации. Цель классификации — разработка адекватной технологии программирования для каждого класса программ. Теория программ каждого класса должна определять методы спецификации, верификации (в широком смысле), моделирования и эффективной реализации программ. Базисом классификации являются:

- внешняя форма программы (интерфейс с окружением);
- базисные конструкции языка программирования;

- формы определения спецификации программы;
- виды условий корректности программы.

Генеральная классификация определяет два класса программ: невзаимодействующие программы (или *программы-функции*) и реактивные системы (или *программы-процессы*). Данные два класса составляют более 90 % всех программ. Существуют другие, более сложные классы, например, языковые процессоры и операционные системы; они находятся на метауровне по отношению к первым двум классам. Языковые процессоры — это интерпретаторы программ, компиляторы, оптимизаторы, трансформаторы и т. д. Приведенная классификация не покрывает всего спектра программ, в частности, определяемых разнообразного вида фреймворками, и различных особенностей, например, такой как тесная интеграция программы с данными.

Класс программ-функций

Программа принадлежит этому классу, если она не взаимодействует с внешним окружением; точнее, если возможно перестроить программу таким образом, чтобы все операторы ввода данных находились в начале программы, а весь вывод был собран в конце программы. Если подобная перестройка программы принципиально невозможна, ее следует пытаться определять в виде реактивной системы. Программа обязана всегда нормально завершаться с получением результата, поскольку бесконечно работающая и невзаимодействующая программа бесполезна. Программа определяет функцию, вычисляющую по набору входных данных (аргументов) некоторый набор результатов. Класс программ-функций, содержит, по меньшей мере, программы для задач дискретной и вычислительной математики.

Таким образом, окружение программы-функции имеет простую структуру и состоит из средств ввода аргументов и вывода результатов программы.

Программу-функцию U с набором аргументов x и набором результатов y будем записывать в виде U(x**:** y). *Спецификацией* программы-функции являются два предиката: *предусловие* P(x) и *постусловие* Q(x, y). Предусловие ограничивает допустимый набор аргументов, а постусловие определяет связь между значениями аргументов и результатов. Спецификацию программы будем записывать в виде [P(x), Q(x, y)].

Однозначность и тотальность спецификации [P(x), Q(x, y)] определяются, соответственно, формулами:

$$P(x) \,\&\, Q(x, y_1) \,\&\, Q(x, y_2) \Rightarrow y_1 = y_2;$$
$$P(x) \Rightarrow \exists y.\ Q(x, y).$$

Программа должна соответствовать спецификации. Это требование формулируется в виде условия *частичной корректности*: если перед исполнением программы U(x**:** y) истинно предусловие P(x), то в случае завершения программы должно быть истинно постусловие Q(x, y) в момент ее завершения. Обязательным является также *условие завершения программы*: если перед исполнением программы U(x**:** y) истинно предусловие P(x), то программа обязана завершаться. Объединение условий частичной корректности и завершения программы определяет условие *тотальной корректности*.

Адекватная формализация условий корректности программы возможна лишь при использовании формальной операционной семантики языка программирования. *Операционную семантику* программы U(x**:** y) определим в виде предиката:

$\mathcal{R}(U)(x, y) \cong$ для значения набора x исполнение программы U всегда завершается и существует исполнение программы, при котором результатом вычисления является значение набора y.

Данное определение исключает ситуацию, когда для некоторого значения набора x существуют два разных исполнения программы, одно из которых завершается, а другое — не завершается. В этом случае $\mathcal{R}(U)(x, y)$ будет ложным для любых y.

Однозначность и тотальность программы U(x**:** y) для некоторого значения x определяются, соответственно, формулами

$$\mathcal{R}(U)(x, y_1) \,\&\, \mathcal{R}(U)(x, y_2) \Rightarrow y_1 = y_2;$$
$$\exists y.\ \mathcal{R}(U)(x, y).$$

Отметим, что тотальность программы для некоторого значения x есть в точности условие завершения программы для этого значения x. Программа называется *однозначной*, если она однозначна для всех значений аргументов.

Операционная семантика $\mathcal{R}(U)$ является эквивалентом программы U. Доказательство некоторого свойства программы W(x, y) реализуется доказательством истинности формулы $\mathcal{R}(U)(x, y) \Rightarrow W(x, y)$. Кроме того, эту формулу достаточно доказать при истинном предусловии. С учетом этого, условие частичной корректности программы U(x**:** y) записывается в виде формулы

$$\forall x, y.\ P(x) \,\&\, \mathcal{R}(U)(x, y) \Rightarrow Q(x, y).$$

Условие завершения программы U(x**:** y) при истинном предусловии представляется формулой:

$$\forall x.\ P(x) \Rightarrow \exists y.\ \mathcal{R}(U)(x, y).$$

Формула для условия тотальной корректности программы получается объединением двух формул:

$$\forall x.\ P(x) \Rightarrow [\forall y.\ \mathcal{R}(U)(x, y) \Rightarrow Q(x, y)] \,\&\, \exists y.\ \mathcal{R}(U)(x, y).$$

Лемма 1. Если программа U(x**:** y) тотально корректна относительно спецификации [P(x), Q(x, y)], то спецификация тотальна.

Для конструирования программ-функций допустим любой язык императивного или функционального программирования. Минимальный язык P_0, из которого можно построить полный язык предикатного программирования, определен в работе [22]. Операторами языка P_0 являются: *оператор суперпозиции* B(x**:** z); C(z**:** y); *параллельный оператор*

B(x**:** y) || C(x**:** z); *условный оператор* **if** (e) B(x**:** y) **else** C(x**:** y), *вызов программы* и *оператор каррирования* D(y**:** z) {B(x, y**:** z)}.

Класс программ-процессов

Программа-процесс является *реактивной системой*, реагирующей на определенный набор событий (сообщений) во внешнем окружении программы. Взаимодействие программы с окружением реализуется через прием/посылку сообщений и разделяемые переменные, доступные в программе и окружении.

Сообщение — объект вида $m(x_1, x_2, ..., x_n)$, где m — *имя сообщения*, $x_1, x_2, ..., x_n$ ($n \geqslant 0$) — переменные, являющиеся *параметрами сообщения.* Сообщение, пришедшее из окружения программы-процесса, может быть получено *оператором приема сообщения*, которому становятся доступными значения переменных $x_1, x_2, ..., x_n$. Оператор **send** $m(e_1, e_2, ..., e_n)$ посылает сообщение m с параметрами — значениями выражений $e_1, e_2, ..., e_n$.

Разделяемая переменная является глобальной по отношению к программе-процессу. Она доступна по чтению и/или записи внутри программы, а также может быть модифицирована в окружении программы.

Программа-процесс является либо автоматной программой, либо она определяется в виде композиции нескольких автоматных программ, исполняемых параллельно и взаимодействующих между собой через сообщения и разделяемые переменные.

Автоматная программа состоит из одного или нескольких сегментов. *Сегмент* имеет один *вход*, помеченный меткой — *управляющим состоянием*. Сегмент имеет один или несколько выходов. *Выход* реализуется оператором перехода на начало другого сегмента (или того же самого), либо как выход из автоматной программы. Автоматная программа определяет конечный автомат в виде *гиперграфа* с набором управляющих состояний в качестве вершин и набором сегментов в качестве ориентированных гипердуг. Существует одно *начальное управляющее состояние*, с которого начинается исполнение автоматной программы. Автоматная программа может иметь одно или несколько *выходных управляющих состояний*, которыми завершается исполнение автоматной программы. Отметим, что гиперграфовая модель является наиболее общей среди других используемых моделей автоматных программ.

Состояние автоматной программы определяется значениями набора модифицируемых переменных, локальных по отношению к автоматной программе и глобальных по отношению к каждому ее сегменту.

Структура класса реактивных систем. Частью задачи классификации программ является определение подклассов класса программ-процессов, т. е. класса реактивных систем.

Подклассом реактивных систем являются *гибридные системы*, соединяющие дискретное и непрерывное поведение. Часть переменных состояния гибридной системы соответствует непрерывным параметрам (типа **real**), изменение которых реализуется независимо от программы гибридной системы (вне ее) по определенным законам, обычно формулируемым в виде дифференциальных уравнений. Важнейшими подклассами гибридных систем являются контроллеры систем управления и временные автоматы.

Система управления реализует взаимодействие с *объектом управления* для поддержания его функционирования в соответствии с поставленной целью. Системы управления используют в аэрокосмической отрасли, энергетике, медицине, робототехнике, массовом транспорте и других отраслях. На каждом шаге вычислительного цикла *контроллер системы управления* получает входную информацию из окружения и обрабатывает ее. Результаты вычисления используются для передачи управляющего сигнала для воздействия на объект управления. Большинство систем управления являются встроенными системами.

Временной автомат реализует функционирование процесса, используя показания времени. Пересчет времени проводится вне программы-процесса (временного автомата). Существуют различные модели автоматов с дискретным и непрерывным временем [23]. Временной автомат является *системой реального времени*, если взаимодействие с окружением должно удовлетворять временным ограничениям, что характерно для встроенных систем. В системах с жестким реальным временем непредоставление результатов вычислений к определенному сроку является фатальной ошибкой.

Автоматная программа является *детерминированной*, если каждое управляющее состояние метит не более одного сегмента. Для *недетерминированного автомата* одно управляющее состояние может метить несколько сегментов. При исполнении программы из данного управляющего состояния недетерминировано выбирается один из сегментов. Отметим, что недетерминизм неявно реализуется для параллельной композиции автоматных программ. Автомат является *вероятностным*, если для каждого сегмента определена вероятность его выбора, причем сумма вероятностей сегментов, исходящих из одного управляющего состояния, равна единице.

Программа может быть составлена из частей, принадлежащих разным подклассам реактивных систем. Например, возможно сочетание вероятностных и недетерминированных автоматов в рамках одной программы. Системы реального времени в большинстве случаев являются системами управления. В дополнении к этому автоматная программа может быть частью *распределенной системы.* Отметим также возможность интеграции с объектно-ориентированной технологией, когда состояние автоматной программы реализовано как объект класса.

Валидация и верификация реактивной системы. Разработка реактивной системы начинается с определения ее требований. *Валидация требований* заключается в проверке требований на соответствие потребностям пользователей. Обычно здесь приме-

няют моделирование. Результаты валидации оценивают совместно разработчик и заказчик.

Спецификация реактивной системы включает: инварианты управляющих состояний, свойства, формулируемые на языке темпоральной логики, и описание части требований.

Инвариант управляющего состояния — предикат, который должен быть истинным в начале сегмента, ассоциированного с данным управляющим состоянием. Инварианты реактивных систем принципиально отличаются от инвариантов циклов и инвариантов классов императивных программ.

Объектами верификации могут быть также *свойства* автоматной программы, обычно формулируемые на языке темпоральной логики. Они могут быть верифицированы с помощью инструментов проверки на модели (*model checking*).

Программа-функция — это автоматная программа с двумя управляющими состояниями: одно входное и одно выходное. Предусловие и постусловие полностью определяют функциональные требования к программе. Иногда формулируются отдельные нефункциональные требования, однако обычно эта часть требований отсутствует.

Класс языковых процессоров

Языковые процессоры — это интерпретаторы программ, трансляторы, оптимизаторы, трансформаторы, смешанные вычислители, конверторы и другие операции с программами.

Язык программирования — формальный язык, определяющий правила записи программы в виде текста в конечном *алфавите* символов. *Лексические правила* определяют правила кодирования символов алфавита (*лексем*) посредством алфавита компьютера, а также программы в целом в конкретной файловой системе.

Описание языка программирования определяет: типы данных, структуру памяти исполняемой программы, виды языковых конструкций программы, правила исполнения конструкций каждого вида и программы в целом.

Языковая конструкция — независимая часть программы с определенным для нее синтаксисом и семантикой. Язык программирования характеризуется набором *видов языковых конструкций*. Например, вид Оператор присваивания, синтаксически изображаемый композицией <Переменная>: = <Выражение>, с двумя *позициями* для подконструкций. Виды конструкций объединяются в *группы*. Примеры групп: оператор, выражение, переменная, операция.

Для всякого вида языковых конструкций определены синтаксис и семантика. *Синтаксис* определяет правила представления конструкции в виде последовательности символов алфавита. *Семантика* определяет правила *исполнения* произвольной конструкции данного вида. Частью семантики является *статическая семантика*: правила идентификации переменных и других объектов программы, совокупность ограничений на типы (и другие атрибуты) конструкций и система умолчаний. Вторая часть семантики является собственно *семантикой исполнения* программы.

Точная спецификация языка программирования, проводимая преимущественно для целей формальной верификации программ, реализуется в виде *формальной семантики*, представляющей математическое описание семантики языка программирования. Формальная семантика есть описание семантики исполнения, абстрагированное от синтаксиса и статической семантики. Различают следующие виды формальной семантики: операционную, денотационную и аксиоматическую [24, 25]; алгебраическая семантика [26] является разновидностью денотационной. *Операционная семантика* для всякого вида K определяет предикат $\mathcal{R}(K)$, истинный при нормальном завершении исполнения конструкции вида K. Описание формальной семантики реальных языков программирования оказывается сложным и громоздким.

Интерпретатор — программа, реализующая автоматическое исполнение произвольной программы на языке программирования. В содержательном пользовательском описании семантики исполнения языка программирования интерпретатор подразумевается неявно. Чтобы определить *спецификацию интерпретатора*, рассмотрим отображение S формальной операционной семантики на пользовательское представление языка программирования: конструкции формальной семантики кодируются в синтаксических структурах исходного языка. Необходимо будет явно определить интерпретатор исходного языка, в частности, формализовать структуру памяти исполняемой программы. Исполнение конструкции вида K реализуется независимой подпрограммой интерпретатора. Результаты ее исполнения должны удовлетворять предикату $S(\mathcal{R}(K))$.

Интерпретатор программы на исходном языке крайне неэффективен. Необходимым инструментом программирования является *транслятор*, преобразующий программу на другой язык, называемый *объектным*, для которого обеспечено эффективное исполнение программы. При построении транслятора необходимо определить объектное представление программы. Формально это можно представить как отображение O формальной операционной семантики на объектный язык. Тогда *спецификацию транслятора* можно было бы представить в виде функции, отображающей $S(\mathcal{R}(K))$ в $O(\mathcal{R}(K))$. Такая спецификация должна гарантировать, что любое исполнение исходной программы будет совпадать по всем промежуточным результатам с соответствующим исполнением объектной программы.

Однако любой транслятор, иногда даже простой конвертор, дополнительно проводит эквивалентные оптимизирующие преобразования программы, например, константные вычисления. Поэтому в действительности спецификация транслятора значительно сложнее: это перевод программы на объ-

ектный язык с выполнением набора эквивалентных оптимизирующих преобразований, гарантирующих совпадение конечных результатов вычисления.

Класс языковых процессоров существенно сложнее класса программ-функций. Спецификация языкового процессора базируется на формальной семантике, оперирующей множеством исполнений произвольной программы.

Заключение

В настоящей работе рассматривается классификация программ по их внутренней организации, определяющей интерфейс с внешним окружением, форму спецификации программы, базисные языковые конструкции и другие особенности. Классификация ориентирована на разработку адекватной технологии для каждого класса программ. На базе общей системы понятий, используемых в программной инженерии, определены три класса программ: класс программ-функций, класс реактивных систем и класс языковых процессоров. В дальнейшем предстоит идентифицировать и определить класс операционных систем, являющийся метауровневым по отношению к первым двум классам. Приведенная система классов заведомо неполна. Для продолжения классификации программ необходимо проанализировать самые разные виды приложений, в частности, связанные с базами данных и фреймворками.

Одной из задач классификации программ является дальнейшая детализация подклассов внутри класса реактивных систем. Задача следующего уровня — построение моделей подклассов с ориентацией на технологию программирования. В работе [27] на примере программы управления беспилотным летательным аппаратом описана типовая модель систем управления с включением трех дополнительных слоев: интеграции автоматического и ручного управления, мониторинга и защиты от несанкционированного доступа. Дальнейшей задачей является определение модели движения объекта в трехмерном пространстве с ориентацией на системы робототехники.

Работа выполнена при поддержке РФФИ, грант № 16-01-00498.

Список литературы

1. **Hoare C. A. R., Jifeng He.** Unifying Theories of Programming. Prentice Hall Series in Computer Science. Prentice Hall Europe. 1998.

2. **Software** Engineering Method and Theory. URL: http://www.semat.org/

3. **Карнаухов Н. С., Першин Д. Ю., Шелехов В. И.** Язык предикатного программирования Р // Препринт № 153. Новосибирск: ИСИ СО РАН, 2010. 42 с.

4. **Shelekhov V. I.** Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements // Automatic Control and Computer Sciences. 2011. Vol. 45, N. 7. P. 421—427.

5. **Шелехов В. И.** Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия. 2011. № 2. С. 14—21.

6. **Шелехов В. И.** Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования // Препринт № 164. Новосибирск: ИСИ СО РАН, 2012. 30 с.

7. **Шелехов В. И.** Язык и технология автоматного программирования // Программная инженерия. 2014. № 4. С. 3—15.

8. **Hewitt C., Bisshop P., Steiger R.** A Universal Modular Actor Formalism for Artificial Intelligence // 3rd International Joint Conference on Artificial Intelligence, Stanford, CA, 1973. P. 235—245.

9. **Ершов А. П., Шура-Бура М. Р.** Пути развития программирования в СССР. Препринт № 12. Новосибирск: ВЦ СО АН СССР, 1976.

10. **Программа.** Краткая российская энциклопедия. М.: Большая российская энциклопедия ОНИКС 21 век, 2003.

11. **Клини С. К.** Введение в метаматематику: Пер. с англ., М.: Издательство иностранной литературы, 1957.

12. **Марков А. А., Нагорный Н. М.** Теория алгорифмов. М.: Наука, Физматлит, 1984. 432 с.

13. **Минский М.** Вычисления и автоматы: Пер. с англ. М.: Мир, 1971. 364 с.

14. **Левенчук А. В.** Системноинженерное мышление. М.: МФТИ, 2015. 305 с.

15. **IEEE** Recommended Practice for Software Requirements Specifications. Revision: 29/Dec/11.

16. **Методика** составления спецификаций требований к программному обеспечению (IEEE-830-1998). URL: http://www.webisgroup.ru/services/programming/srs/ieee-830-1998/

17. **Mich L., Franch M., Novi Inverardi P.** Market research for requirements analysis using linguistic tools // Requirements Engineering 2004. Vol. 9, N. 1. P. 40—56.

18. **Zhang W., Beaubouef T., Ye H.** Statechart: A Visual Language for Software Requirement Specification // International Journal of Machine Learning and Computing. 2012. Vol. 2. No. 1. P. 52—61.

19. **Specification** and description language (SDL). ITU-T Recommendation Z.100 (03/93). URL: http://www.itu.int/ITU-T/studygroups/com17/languages/Z100.pdf

20. **Abrial J.-R.** Modelling in Event-B: System and Software Engineering, Cambridge Univ. Press, 2010.

21. **Кулямин В. В.** Методы верификации программного обеспечения. М.: Институт Системного Программирования РАН, 2008. URL:http://www.ict.edu.ru/ft/005645/62322e1-st09.pdf

22. **Шелехов В. И.** Семантика языка предикатного программирования // Знания-Онтологии-Теории (ЗОНТ-15). Новосибирск, 2015. С. 13. URL: http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf

23. **Alur R., Dill D. L.** A theory of timed automata // Theor. Comput. Sci. 1994. Vol. 126. N. 2. P. 183—235.

24. **Лавров С. С.** Программирование. Математические основы, средства, теория. СПб.: БХВ-Петербург, 2001. 220 с.

25. **Meyer B.** Introduction to the Theory of Programming Languages. Prentice Hall, 1990. 448 p.

26. **Замулин А. В.** Алгебраическая семантика императивного языка программирования // Программирование. 2003. № 6. С. 1—14.

27. **Тумуров Э. Г., Шелехов В. И.** Требования к системе управления квадрокоптером // Системная информатика. 2015. № 5. С. 39—54. URL: http://persons.iis.nsk.su/files/persons/pages/QuadReq.pdf

Program Classification in Software Engineering

V. I. Shelekhov, vshel@iis.nsk.su, A. P. Ershov Institute of Informatics Systems, Novosibirsk, 630090, Russian Federation

Corresponding author:

Shelekhov Vladimir I., Head of Laboratory, A. P. Ershov Institute of Informatics Systems, Novosibirsk, 630090, Russian Federation
E-mail: vshel@iis.nsk.su

Received on August 17, 2016
Accepted on August 31, 2016

Program classification based on kinds of an environment interface, program specification, language kernel, etc. is analyzed. The classification is intended to be a scientific basis for creating proper development methods for each program class. The following three program classes were defined: programs implementing functions (program-functions), reactive systems, and programming language processors. Definitions of program classes are based on the basic notions of a program, a program specification, requirements, etc. which were previously systematically defined. The paper may be used for teaching university students specialized in software engineering.

The program-function and reactive systems classes include more than 90 % of all programs. There are also the following two meta-level classes: programming language processors and operating systems which are considerably more complex. The four mentioned program classes are not the full collection of program classes. Other program classes exist, for example, programs connected with data bases.

Keywords: *software engineering, requirement specification, program specification, reactive system, formal semantics*

Acknowledgements: *This work was supported by the Russian Foundation for Basic Research, project nos. 16-01-00498*

For citation:

Shelekhov V. I. Program Classification in Software Engineering, *Programmnaya Ingeneria*, 2016, vol. 7, no. 12, pp. 531—538.

DOI: 10.17587/prin.7.531-538

References

1. **Hoare C. A. R., Jifeng He**. *Unifying Theories of Programming*, Prentice Hall Series in Computer Science, Prentice Hall Europe, 1998.
2. **Software** Engineering Method and Theory, available at: http://www.semat.org/
3. **Kharnaukhov N. S., Perchine D. Ju., Shelekhov V. I.** *Yazyk predikatnogo programmirovaniya P* (The predicate programming language P), Preprint no. 153. Novosibirsk, ISI SB RAN, 2010, 42 p. (in Russian).
4. **Shelekhov V. I.** Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements, *Automatic Control and Computer Sciences*, 2011, vol. 45, no. 7, pp. 421—427.
5. **Shelekhov V. I.** Verifikatsiya i sintez effektivnykh programm standartnykh funktsiy v tekhnologii predikatnogo programmirovaniya (Verification and synthesis of effective programs for standard functions in predicate software engineering), *Programmnaya Ingeneria*, 2011, no. 2, pp. 14—21 (in Russian).
6. **Shelekhov V. I.** *Razrabotka i verifikatsiya algoritmov piramidalnoy sortirovki v tekhnologii predikatnogo programmirovaniya* (Development and verification of the heapsort algorithms in predicate software engineering), Preprint no. 164. Novosibirsk, ISI SB RAN, 2012, 30 p. (in Russian).
7. **Shelekhov V. I.** Yazyk i tekhnologiya avtomatnogo programmirovaniya (Automata-based software engineering: the language and development methods), *Programmnaya Ingeneria*, 2014, no. 4, pp. 3—15 (in Russian).
8. **Hewitt C., Bisshop P., Steiger R.** A Universal Modular Actor Formalism for Artificial Intelligence, *3rd International Joint Conference on Artificial Intelligence*, Stanford, CA, 1973, pp. 235—245.
9. **Ershov A. P., Shura-Bura M. R.** *Puti razvitija programmirovanija v SSSR* (The history of software engineering in the USSR), Preprint № 12. Novosibirsk, VC SO AN SSSR, 1976 (in Russian).
10. **Programma,** *Kratkaja rossijskaja jenciklopedija* (The short Russian encyclopedia), Moscow, Bol'shaja rossijskaja jenciklopedija ONIKS 21 vek, 2003 (in Russian).
11. **Klini S. K.** *Vvedenie v metamatematiku* (Introduction in Metamathematics), Per. s angl., Moscow, Izdatel'stvo inostrannoj literatury, 1957 (in Russian).
12. **Markov A. A., Nagornyj N. M.** *Teorija algorifmov* (Algorithms Theory). Moscow: Nauka, Fizmatlit, 1984, 432 p. (in Russian).
13. **Minskij M.** *Vychislenija i avtomaty* (Computations and Automata): Per. s angl., Moscow, Mir, 1971, 364 p. (in Russian).
14. **Levenchuk A. V.** *Sistemnoinzhenernoe myshlenie* (System Engineering Thinking), Moscow, MFTI, 2015, 305 p. (in Russian).
15. **IEEE** Recommended Practice for Software Requirements Specifications. Revision: 29/Dec/11.
16. ***Metodika*** *sostavlenija specifikacij trebovanij k programmnomu obespecheniju* (IEEE Recommended Practice for Software Requirements Specifications) (IEEE-830-1998), available at: http://www.webisgroup.ru/services/programming/srs/ieee-830-1998/ (in Russian).
17. **Mich L., Franch M., Novi Inverardi P.** Market research for requirements analysis using linguistic tools, *Requirements Engineering*, 2004, vol. 9, no. 1, pp. 40—56.
18. **Zhang W., Beaubouef T., Ye H.** Statechart: A Visual Language for Software Requirement Specification, *International Journal of Machine Learning and Computing*, 2012, vol. 2, no. 1, pp. 52—61.
19. **Specification** and description language (SDL). ITU-T Recommendation Z.100 (03/93), available at: http://www.itu.int/ITU-T/studygroups/com17/languages/Z100.pdf
20. **Abrial J.-R.** *Modelling in Event-B: System and Software Engineering*, Cambridge Univ. Press, 2010.
21. **Kuljamin V. V.** *Metody verifikacii programmnogo obespechenija* (Methods of Software Verification), Moscow, Institut Sistemnogo Programmirovanija RAN, 2008 (in Russian).
22. **Shelekhov V. I.** Semantika jazyka predikatnogo programmirovanija (Formal Semantics of the P Predicate Language), *ZONT-15*, Novosibirsk, 2015, 13 p. (in Russian).
23. **Alur R., Dill D. L.** A theory of timed automata, *Theor. Comput. Sci.*, 1994, vol., 126, no. 2, pp. 183—235.
24. **Lavrov S. S.** *Programmirovanie. Matematicheskie osnovy, sredstva, teorija* (Software Engineering: mathematics, theory, tools). Saint Petersburg, BHV-Peterburg, 2001, 320 p. (in Russian).
25. **Meyer B.** *Introduction to the Theory of Programming Languages*, Prentice Hall, 1990, 448 p.
26. **Zamulin A. V.** Algebraicheskaja semantika imperativnogo jazyka programmirovanija (Algebraic Semantics of Imperative Programming Languages), *Programmirovanie.* 2003, no. 6, pp. 1—14 (in Russian).
27. **Tumurov E. G., Shelekhov V. I.** Trebovaniy k sisteme upravleniya kvadrokopterom (Requirement specification of quad_rotor flight control system), *Sistemnaja informatika*, 2015, no. 5, pp. 39—54 (in Russian).

УДК 004.4’2 DOI: 10.17587/prin.7.537-546

П. А. Ченцов, канд. физ.-мат. наук, ст. науч. сотр., e-mail: chentsov.p@mail.ru,
Институт математики и механики им. Н. Н. Красовского УрО РАН, Екатеринбург

Об одном подходе к построению интерфейсов консольных приложений: технология TextControlPages

Конфигурирование консольных приложений, работающих в текстовом режиме, и управление ими — актуальная задача. Как правило, для ее решения используют текстовые команды или конфигурационные файлы. В данной работе предложены новый, интерактивный подход к взаимодействию пользователя с консольными приложениями, обеспечивающий высокую переносимость кода, а также его реализация в виде библиотеки классов C++.

Ключевые слова: *интерфейс, меню, модальный диалог, переносимость кода, консольное приложение*

Введение

Наиболее распространенным и удобным механизмом взаимодействия пользователя с программами в настоящее время является графический интерфейс. При этом следует иметь в виду, что графика свойственна клиентским системам. Большинство серверных программ работают в консоли и не имеют графических интерфейсов. Достаточно представить, какой процент серверов в мире работает под управлением операционных систем Unix и Linux. Кроме того, существует большое число различного назначения утилит и вычислительных программ, которые также работают в консоли. Иными словами, несмотря на прогресс в области графических интерфейсов, консольные приложения остаются востребованными.

Существуют три основных подхода к взаимодействию пользователя с консольной программой. Первый и, безусловно, основной — использование текстовых команд. Его суть состоит в том, что пользователь запускает приложение с набором операндов, из которых часть описывает команды, а часть — параметры этих команд. Примеры таких утилит можно найти в работе [1]. Кроме того, при запуске приложения может быть выведена внутренняя консоль программы (см. [1], утилита mysql). В таком случае пользователь вводит команды прямо в процессе выполнения программы. Отрицательной стороной такого подхода является то обстоятельство, что необходимо помнить большое число текстовых команд, типы и последовательность параметров этих команд. Безусловно, программы, использующие такой подход, содержат либо файл с инструкцией, либо описание команд, которое выводится прямо в консоль (обычно в ответ на символ знака вопроса). Когда число команд исчисляется одним или двумя десятками, то их еще можно помнить. Если их сотни, и каждая имеет значительное число операндов, процесс работы с приложением становится не столь комфортным и наглядным. Еще одним отрицательным фактором такого подхода является время, которое затрачивается на набор команд и исправление синтаксических ошибок.

Второй подход состоит в использовании конфигурационных файлов (см. [2], конфигурационные файлы Apache). В файлах содержатся строки с параметрами и их значениями, а также тексты комментариев. На первый взгляд этот подход кажется удобным и прогрессивным. Но и здесь не обходится без затруднений.

Во-первых, как правило, такие файлы имеют большой размер (например, php.ini). В таком случае найти нужный параметр без использования функции поиска в текстовом редакторе сложно. Однако и такой поиск может не помочь, если не помнить точное написание названия параметра.

Во-вторых, некоторые программы используют конфигурационный файл для описания параметров какой-либо одной функции из набора. Например, для описания параметров одного из вычислительных алгоритмов. У разных алгоритмов разные параметры. Если какой-либо параметр пропустить, то будет использовано значение по умолчанию, и серьезных затруднений не возникнет. Вместе с тем для правильной работы нужно помнить все параметры, связанные с данной функцией, так как они влияют на ее выполнение. Может возникнуть еще более сложная ситуация, когда внутри какой-либо функции может потребоваться конфигурирование одной или нескольких вложенных функций. Один из возможных способов преодоления трудности — перечислить все возможные параметры всех функций, после чего откомментировать их. Когда пользователю нужно выполнить одну из функций, он снимает символы комментирования с соответствующих параметров. Однако этот процесс трудоемкий, и к тому же не позволяющий преодолеть трудности, описанные далее.

В-третьих, при переходе к новой версии программы, некоторые из отмеченных выше параметров исключаются и добавляются новые. Адаптация старого конфигурационного файла к новой версии может вызвать серьезные затруднения. Нужно удалить вручную не используемые параметры (этого можно не делать, но тогда есть риск получить через несколько версий файл с большим количеством информационного мусора). Кроме того, появляется необходимость изучить особенности новой версии, и вручную добавить появившиеся вновь параметры. В противном случае эти параметры останутся неизвестными. Для больших конфигурационных файлов решение такой задачи очень нетривиально.

Третий подход состоит в использовании таких библиотек, как TurboVision [3], Curses [4] и др., которые предлагают интерфейс в оконном стиле с использованием цветового оформления и символов псевдографики. Подобный интерфейс можно увидеть в широко известном файловом менеджере FAR. Данные разработки базируются на системных функциях работы с консолью, которые различны в разных операционных системах (ОС). Безусловно, разработчики предлагают реализации для разных ОС, но полной переносимости нет. Кроме того, с развитием графических систем поддержка подобных библиотек прекращается.

Принимая во внимание представленные выше соображения, хотелось бы сформировать некий универсальный принцип построения консольных интерфейсов. С одной стороны, этот принцип должен гарантировать корректную работу при изменении размеров консоли, кодировки текса, он должен опираться на простейшие функции вывода строки в консоль и ввода пользователем символа и строки с клавиатуры (такие функции обязательно должны быть в любой системе разработки программного обеспечения, работающей с консолью). С другой стороны, он должен обеспечивать достаточно высокий уровень интерактивности, наглядности.

Вторая цель настоящей работы — разработка интерфейсной библиотеки на языке C++ с использованием предложенного подхода. Должны быть использованы всего лишь две стандартные библиотеки, а именно *stdio* и *conio.* Наряду с другими функциями в состав этих библиотек входят функции *printf*, *getch* и *gets*, на которые и будет опираться разрабатываемая система. Сегодня трудно себе представить компилятор C++ без этих библиотек и уж тем более без отмеченных выше функций в их составе. Наличие таких функций сделало бы подобную разработку стопроцентно переносимой без подключения различных библиотек в разных ОС. Такая интерфейсная библиотека гарантированно работала бы с любыми компиляторами и ОС, не создавая никаких затруднений с совместимостью. Таким образом, пользователи могут либо самостоятельно разработать программный интерфейс, основанный на предлагаемых принципах, либо использовать указанную библиотеку.

Постановка задачи

Наиболее удобным, наглядным и популярным у пользователей в настоящее время является графический пользовательский интерфейс на основе окон (речь идет о компьютерном программном обеспечении). В качестве примеров можно привести: Windows Forms в платформе Microsoft.NET [5], Microsoft Visual C++ [6], Borland C++ Builder [7]. При детальном рассмотрении видно, что за внешне красивым графическим интерфейсом скрывается реализация некоторых абстрактных приемов работы с данными и программными функциями. Требуется выделить наиболее востребованные из этих приемов и постараться перенести их в текстовую консоль с использованием минимума основных функций работы с консолью, сформировав принцип построения интерактивных переносимых консольных интерфейсов. Далее требуется реализовать этот принцип в виде библиотеки классов C++.

Пользователь должен иметь возможность доступа к данным и функциям программы. Для этих целей следует использовать многоуровневое меню с возможностью запрещать доступ к отдельным элементам и отмечать требуемые элементы. Нужно предусмотреть возможность вывода расширенной подсказки по каждому элементу меню.

Для ввода данных следует использовать модальные диалоги, а именно стек модальных диалогов. Должны быть предусмотрены следующие наиболее распространенные элементы диалогов: поле ввода с возможностью валидации, флажок (CheckBox), список строковых значений с возможностью выбора, статический текст и кнопка. Для всех перечисленных элементов диалогов должна присутствовать возможность вывода развернутой подсказки, а также возможность запрещать доступ к элементу (ReadOnly).

Для взаимодействия с пользователем доступной предполагается только клавиатура. Следовательно, для активации пунктов меню и взаимодействия с элементами диалогов доступны лишь клавиши на клавиатуре. В связи с этим предлагается использовать клавиши с цифрами и латинскими буквами. Такой подход позволит абстрагироваться от набора клавиш конкретных клавиатур — латинские буквы и цифры, как правило, присутствуют на любой клавиатуре. Перед каждым элементом интерфейса в скобках должен располагаться соответствующий данному элементу символ. Такие символы далее будем называть управляющими символами.

Другой важный момент — расположение элементов интерфейса. Предполагается только последовательный вывод символов без выделения цветом. Наиболее целесообразным представляется вывод элементов интерфейса в виде вертикального списка. Обязательно должна быть предусмотрена возможность работы с большим числом элементов, превышающим число строк консоли (как для меню, так и для диалогов). Реализация должна иметь в своей основе принцип скроллинга. Должен быть также при-

нят во внимание тот факт, что на разных консолях один и тот же элемент может быть выведен в одну строку или занять несколько строк, если длина его текста превышает ширину консоли. Переносы не должны разрывать слова и числа.

Требуется предусмотреть возможность смены кодировки текста без перекомпиляции непосредственно в процессе работы программы. Возможно, следующее замечание покажется многим читателям неактуальным, однако такой интерфейс вполне сможет функционировать на компьютерах без дисплея, оснащенных для вывода, например, принтером.

Концепция TextControlPages

Концепция TextControlPages представляет собой способ построения интерфейсов, основанный на нескольких стандартных функциях работы с консолью, который обеспечивает высокий уровень интерактивности, легкий и удобный доступ к данным и функциям программы. Иными словами, с ее использованием получаем максимальную совместимость с различными компиляторами и ОС (устаревшими, современными и теми, которые должны появиться в будущем) с одной стороны и возможность визуально наглядного взаимодействия с программой в стиле оконных приложений (пусть интерфейс и не является красивым и цветным, а дизайн простой), с другой стороны.

Существуют программы с несколькими пунктами меню и парой коротких диалогов. Однако кроме них существуют и другие программы — с большими меню, число элементов которых на одном уровне не способно уместиться на экран даже с учетом хорошей структуризации, а также с большими диалогами и списками выбора значений, содержащими сотни строк. В рамках концепции TextControlPages предусматривается отображение конструкций, не помещающихся полностью на экран. Обязательно следует учитывать размер консоли, чтобы правильно рассчитывать объем выводимой информации. Учет размера консоли также нужен, чтобы контролировать символы новой строки в позициях автоматического переноса (если каретка достигла крайнего правого положения, то осуществляется автоматический перенос; если следующий символ — символ переноса строки, то возникнет нежелательная дополнительная пустая строка, которая не должна отображаться). Также известная ширина консоли позволяет исключить ухудшающие восприятие разрывы слов и чисел. Для вычисления размера консоли предлагается простой и наглядный режим калибровки. Такой режим должен быть выполнен один раз, он позволяет избежать использования системных функций, специфических для той или иной системы разработки.

Другая важная особенность TextControlPages — поддержка различных кодировок. Смена кодировки проводится непосредственно в процессе работы программы, что позволяет использовать один и тот же код с кириллицей в разных ОС.

Как уже отмечалось ранее, мышь для пользователя предполагается недоступной, следовательно, он будет взаимодействовать с системой только через клавиатуру, а именно — через управляющие символы. На экране перед каждым интерактивным элементом интерфейса должен быть расположен в скобках соответствующий управляющий символ. Безусловно, работать с мышью значительно удобнее для современного пользователя. Однако на практике взаимодействие через кнопки оказывается вполне комфортным, так как вся информация выведена на экран и достаточно беглого взгляда и быстрого нажатия на кнопку с соответствующим управляющим символом.

Как правило, современные интерфейсы имеют развернутые всплывающие подсказки к интерактивным элементам интерфейса — достаточно задержать на пару секунд мышь над соответствующим пунктом меню или кнопкой. Такая функция обязательна — ведь в коротком тексте элемента меню или комментария к управляющей конструкции в большинстве случаев нельзя уместить текст, подробно разъясняющий назначение. В TextControlPages также реализован механизм всплывающих подсказок. Если для управляющей конструкции задана всплывающая подсказка, то между управляющим символом и текстом конструкции отображается знак вопроса. Ввод с консоли символа "?" приводит к активации режима отображения всплывающих подсказок и в таком режиме ввод управляющего символа не вызывает активацию какого-либо действия, а приводит к отображению текста подсказки.

Содержимое интерфейса и меню

Графический интерфейс меняет свой внешний вид в ходе взаимодействия с пользователем либо мгновенно, либо с анимацией. В консоли нет возможности обеспечить такой способ поведения. Пользователю выдается блок текста с текущим состоянием интерфейса, причем число строк этого блока не может превышать число строк консоли. Данный блок текста отображает текущее состояние интерфейса. Далее подобный текст, который выводится в консоль и отображает текущее состояние программы, будем называть содержимым интерфейса. При нажатии какого-либо управляющего символа интерфейс должен изменить свое состояние — в консоль выводится пустая строка, а за ней новое содержимое интерфейса (рис. 1).

Приведенный на рис. 1 текст демонстрирует интерфейс некоторой программы. В каждый момент времени на экран выводится подобный текст, и от пользователя ожидается ввод управляющего символа (в данном случае это "a", "b" или "c"). Если какой-либо элемент интерфейса неактивен (включен режим ReadOnly), то вместо управляющего символа в квадратных скобках указан символ "–" (на рис. 1 неактивен пункт Руководство). Активизировать такой элемент невозможно. После ввода

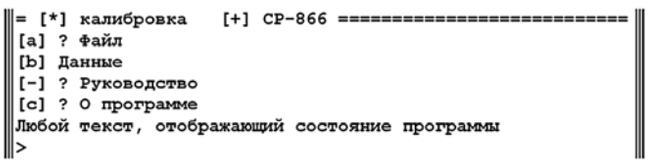

Рис. 1. Содержимое интерфейса

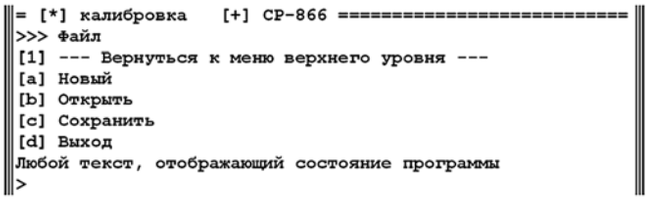

Рис. 2. Отображение вложенных элементов меню

символа происходит вызов обработчика меню. Если элементу меню соответствует какое-либо действие, то это действие выполняется и вновь выводится содержимое интерфейса. Если этот элемент меню используется для группировки и содержит другие элементы, то, кроме вызова обработчика, происходит переход к вложенным элементам (на уровень ниже), как показано на рис. 2.

Управляющий символ "1" используется для возврата к стоящему выше в иерархии элементу меню. При этом во второй строке содержимого интерфейса отображается последовательность вышестоящих элементов меню (в данном случае только "Файл").

Ниже элементов меню отображается строка "Любой текст, отображающий состояние программы". Сюда может выводиться самая различная информация. Например, данный текст может выполнять функции области контента или строки статуса. Далее будем называть этот текст информационным текстом. Информационный текст может содержать несколько строк.

Если содержимое интерфейса насчитывает большее число строк, чем число строк консоли, то включается так называемый режим скроллинга, позволяющий уменьшить число строк до значения этого параметра у консоли. На рис. 3 отображена ситуация, когда не все пункты меню помещаются на экране.

Листание вверх осуществляется с помощью управляющего символа "2", вниз — с помощью символа "3". При достижении начала или конца списка

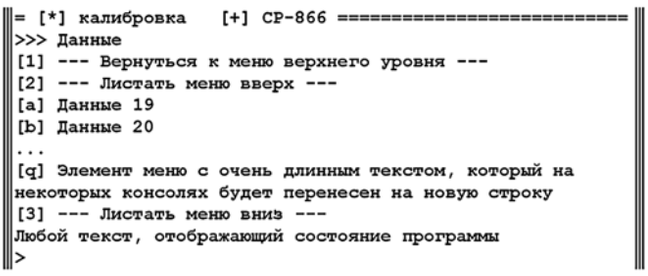

Рис. 3. Большое число элементов меню

элементов меню, соответствующий элемент скроллинга с управляющим символом "2" или "3" пропадает. Этот же принцип используется и в случае с диалогами. Кроме того, текст некоторых элементов интерфейса может превышать ширину консоли. В этом случае будет проведен автоматический перенос строки без разрыва слова (на рис. 3 это элемент меню с управляющим символом "q"). Число строк содержимого интерфейса рассчитывается с учетом этого обстоятельства.

Данная функция имеет большое значение. С одной стороны, она позволяет полностью абстрагироваться от размерности консоли и не опасаться, что созданный интерфейс не прекратит правильно функционировать на консоли с меньшим числом строк или столбцов. С другой стороны, такая функция меню позволяет создавать большие и сложные интерфейсы, не ограничивающие разработчика размером консоли.

Если элемент меню содержит текст описания, то правее управляющего символа располагается знак вопроса. Ввод пользователем управляющего символа "?" активирует режим вывода описаний элементов интерфейса. При этом строка приглашения ввода управляющего символа меняет свой вид с ">" на "?>" (рис. 4).

Если в этом режиме ввести управляющий символ одного из элементов интерфейса с описанием, то в нижней части содержимого интерфейса будет выведено имя элемента и соответствующее описание. Режим вывода описаний при этом выключится автоматически (рис. 5). Отключение данного режима также проводится повторным вводом управляющего символа "?".

Следует отметить, что в первой строке каждого содержимого интерфейса при работе с меню содержится информация о командах калибровки и смены кодировки. Для правильного функционирования интерфейса обязательно нужно знать число строк и столбцов консоли. Это нужно по следующим причинам. Если содержимое интерфейса содержит большее число строк, чем число строк консоли, нужно включить режим скроллинга интерфейса. Число

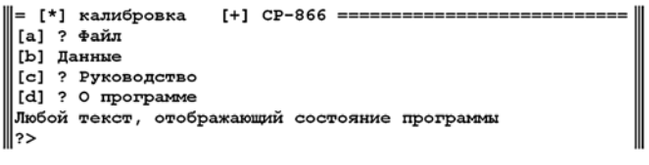

Рис. 4. Режим вывода описаний элементов интерфейса

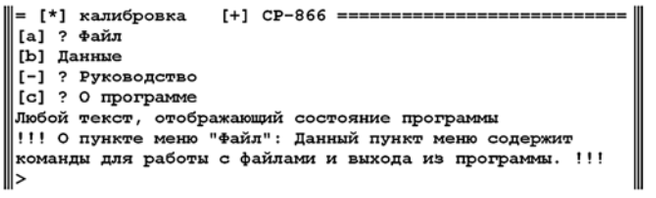

Рис. 5. Описание элемента интерфейса

выводимых на экран строк необходимо рассчитать правильно, принимая во внимание, что некоторые элементы интерфейса с учетом переноса строки могут занять несколько строк, а также то обстоятельство, что информационный текст (текст, располагаемый ниже меню) может содержать несколько строк. Кроме того, число столбцов необходимо знать для того, чтобы избежать разрывов слов и чисел, которые снижают уровень воспринимаемости информации при чтении, и не очень хорошо выглядят с позиций дизайна. Другая причина — появление лишних пустых строк. Как известно, при достижении последнего столбца консоли при выводе текста каретка переходит на новую строку. Если следующий символ — символ переноса строки, по тем или иным причинам расположенный в тексте элемента интерфейса или информационного текста, то добавляется еще одна незапланированная пустая строка, так как перенос уже один раз был сделан автоматически. Знание размеров консоли позволит исключить вывод данного символа переноса строки.

Для того чтобы получить размеры консоли, можно было бы воспользоваться системными функциями. Однако это противоречит концепции максимальной переносимости — такие функции обычно зависят от ОС и библиотек компилятора. По этой причине задача определения размеров консоли возлагается на пользователя, но считать строки и столбцы ему все-таки не придется. Кроме того, выполняется данная функция только один раз (подразумевается, что разработчик должен сохранить и восстановить данные параметры). При вводе управляющего символа "*" пользователю предлагается выполнить два следующих действия. Первое действие — нажимать клавишу пробела до тех пор, пока строчка с символами "*" не заполнит по ширине весь экран, после чего нажать любую клавишу. Второе действие — нажимать клавишу пробела до тех пор, пока столбец с символами "*" в левой части экрана не заполнит по высоте весь экран, после чего нажать любую клавишу. Число нажатий будет определять размеры консоли. Выполняется вся процедура очень быстро, она не является обременительной или трудоемкой.

Вторая функция, на которую указывается в первой строке содержимого интерфейса, позволяет менять кодировку выводимых на консоль и принимаемых от пользователя символов прямо в ходе работы. Дело в том, что программа обычно разрабатывается в какой-либо одной ОС с использованием некоторого компилятора. Вместе с тем следует заметить, что интерфейсная часть не препятствует компиляции ее под другой ОС. В этой, другой ОС может быть использована иная кодировка. Более того, разные кодировки могут быть даже в пределах одной ОС (например, в Windows широко распространена кодировка CP1251, но в консоли используется кодировка CP866). Один из способов решения такой задачи — перекодировать всю программу перед компиляцией, что не является хорошим подходом. Во-первых, программа может работать с файлами, данные из которых имеют кодировку основной ОС. Работа с этими файлами напрямую с использованием новой кодировки вызовет ошибку. Во-вторых, не вполне удобно всякий раз перекодировать множество файлов.

Нажатие клавиши с символом "+" осуществляет циклически смену трех кодировок — CP866, CP1251 и KOI8. Подразумевается, что разработчик должен обеспечить сохранение информации о выбранной кодировке при выходе из программы для последующего восстановления при следующем запуске.

Диалоговые формы

В предыдущем разделе было представлено, как реализуется работа с меню в системе TextControlPages. Одно из назначений меню — активировать диалоги. Диалоги используются для ввода и редактирования данных программы. Все диалоги в TextControlPages модальные. Из одного диалога можно активировать другой (тоже в модальном режиме). На рис. 6 приведен пример диалога в TextControlPages.

В верхней части после строки символов "=", используемой для визуального разделения блоков содержимого интерфейса, выводится строка с элементом меню, из которого был активирован диалог (включая иерархию в меню). Далее идет заголовок формы, обрамленный с обеих сторон комбинацией из трех звездочек для выделения из остального текста. Ниже представлены управляющие элементы. Каждому доступному элементу (если для него отключен режим ReadOnly) ставится в соответствие управляющий символ (буква латинского алфавита), как и в случае с меню. Исключение — статический текст, который можно размещать между активными элементами диалога. Текст каждого элемента формы может превышать длину строки. В этом случае будет проведен автоматический перенос строки. Рассмотрим основные элементы формы.

Первый из них — поле ввода. Данный элемент ввода представляет возможность валидации значений. При вводе управляющего символа, соответствующего полю ввода, в нижней части содержимого интерфейса появится предложение ввести значение. На рис. 7 показан процесс ввода значения "Числового параметра" из примера с рис. 6.

```
============================================================
>>> Данные >>> Данные 1
      *** Заголовок диалога ***
{a} ? Числовой параметр: 17.5
{b} ? Строковый параметр: Текстовое значение параметра
{c} [x] Чекбокс
{d} ? Выбор строки из списка: >> Значение 3 << [v]
Статический текст.
{e} [Сохранить]
{f} [Отмена]
>
```

Рис. 6. Диалоговая форма

```
...
{f} [Отмена]
Числовой параметр (17.5):
```

Рис. 7. Ввод значения параметра

```
...
{f} [Отмена]
Числовой параметр. Укажите число с плавающей точкой в
диапазоне от -10.0 до 1000.0  (17.5):
```

Рис. 8. Повторный ввод значения параметра

```
=========================================================
>>> Данные >>> Данные 1
      *** Заголовок диалога ***
Выбор строки из списка:
Текущее значение: Значение 3
[1] --- Отмена ---
>a> Значение 1
>b> Значение 2
>c> Значение 3
>d> Значение 4
>
```

Рис. 9. Выбор текстовой строки

```
...
{f} [Отмена]
...
```

Рис. 10. Кнопка

Сначала представлено название поля ввода, затем в скобках текущее значение, потом двоеточие, за которым находится курсор ввода. Пользователь должен ввести число и нажать клавишу Enter. В данном случае проводится валидация, введенное число должно находиться в диапазоне от −10 до 1000. При вводе неверного значения (например, "1500" или "Сто") пользователю предлагается повторить ввод (рис. 8).

Следующий элемент ввода — флажок (checkbox). При вводе соответствующего управляющего символа флажок меняет свое значение, что отображается в квадратных скобках перед его описанием.

Для выбора строки из списка используется соответствующий элемент ввода. При вводе нужного управляющего символа содержимое интерфейса изменяет свой вид так, как показано на рис. 9.

Элементы формы скрываются чтобы освободить пространство для списка строк. Если строк слишком много и они не помещаются на экран, то включается функция скроллинга. В этом случае управляющие символы "2" и "3" позволяют листать список для поиска требуемой строки. Чтобы осуществить выбор соответствующей строки, необходимо ввести управляющий символ, предваряющий значение. Если нужно отказаться от выбора, оставив прежнее значение, то следует ввести управляющий символ "1".

Еще один интерфейсный элемент — статический текст. Несмотря на то что данный элемент не используется напрямую для ввода, он позволяет разделять и группировать элементы ввода, либо выводить какую-либо вспомогательную информацию. Элемент статического текста не имеет управляющего символа.

Для выполнения какого-либо действия используются кнопки. Кнопка отображается как текст в квадратных скобках (рис. 10).

Нажатие на кнопку активирует вызов соответствующего обработчика, в котором могут быть прописаны какие-либо пользовательские действия. Например, подтверждение введенных данных и закрытие диалога.

Перечисленных элементов форм вполне достаточно для большинства случаев, хотя, при надобности, можно добавлять и другие элементы, расширяя библиотеку.

Реализация

В настоящем разделе представлено краткое описание библиотеки классов. Однако следует заметить, что это всего лишь один из вариантов реализации. Основная цель статьи — сформулировать базовые положения работы с приложениями в консоли с минимальной зависимостью от ОС, компилятора и оборудования. Библиотека TextControlPages с примером использования доступна для загрузки по адресу: http://tcp.uran.ru. На рис. 11 приведена иерархия классов библиотеки.

Прежде чем комментировать иерархию классов, следует отметить класс CApplication. Это основной класс, описывающий интерфейс. Для создания конкретного интерфейса следует создать класс-наследник от CApplication, наполнить его содержимым (элементы меню, диалоги, обработчики событий),

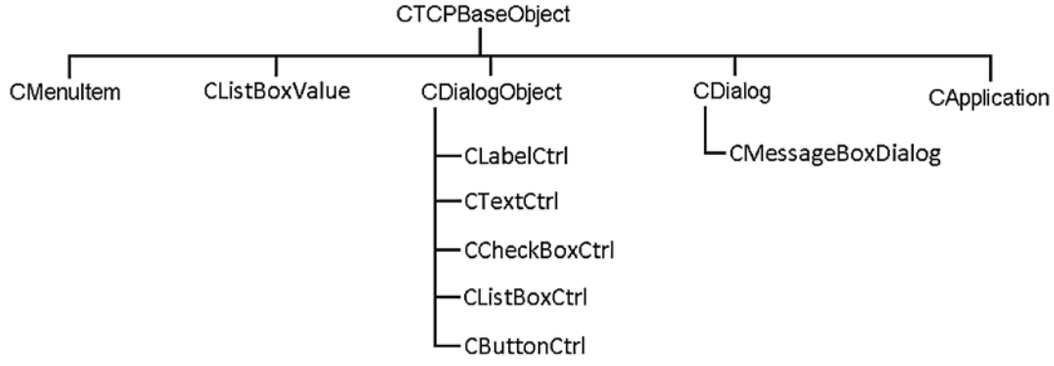

Рис. 11. Иерархия классов

создать объект данного класса и вызвать для него метод Run. Таким образом, внешний вид и меню приложения, а также объекты диалоговых форм создаются именно в классе-потомке от CApplication. Далее такой класс будем называть классом приложения. Допускается создание только одного объекта класса приложения (объект приложения).

Базовым для всех классов библиотеки является класс CTCPBaseObject. Данный класс реализует все функции по взаимодействию с консолью, а именно, он реализует функции ввода и вывода данных с учетом размеров консоли и выбранной кодировки.

Класс CMenuItem используется для создания элементов меню. Каждый элемент меню имеет свойства Visible (элемент виден) и ReadOnly (элемент недоступен), а также Checked (элемент отмечен звездочкой). Элементы меню верхнего уровня вкладываются в объект класса CApplication, который используется для описания интерфейса. Каждый элемент меню может иметь вложенные элементы, образуя, тем самым, иерархическое меню. При выборе элемента меню в объекте приложения проводится вызов обработчика OnMenuItemClick. Это абстрактный метод класса CApplication, который должен быть переопределен в классе приложения. Данный обработчик принимает следующие указатели: на выбранный элемент меню, позволяющий идентифицировать требуемое действие; на булевскую переменную, которая используется для программного запрета доступа к вложенным пунктам меню.

Для работы с диалогами используется класс CDialog. Чтобы построить требуемый диалог следует определить класс-потомок от CDialog и в его конструкторе создать нужные элементы управления, а также переопределить обработчики событий. В классе CDialog можно переопределить следующие обработчики:

- OnTextValidation — нестандартный вариант проверки введенного пользователем значения;
- OnTextEdit — изменен текст в поле ввода;
- OnCheckBoxClick — изменено состояние флажка;
- OnListBoxBoxValueChange — проведен выбор строки из предложенного набора;
- OnButtonClick — проведено нажатие на кнопку.

Это минимальный набор обработчиков, однако для стандартных ситуаций его вполне достаточно.

Далее, объект класса диалога создается в конструкторе или в обработчике объекта приложения. Для вывода на экран диалога (обычно в обработчике меню) объект диалога должен быть переведен в модальный режим вызовом его метода DoModal. Внутри обработчиков диалога можно активировать другой диалог, что соответствует модели модальных форм. Результат работы диалога возвращается на выходе DoModal.

Класс CDialogObject является базовым для всех элементов диалога. Он содержит общие для всех элементов функциональные возможности (например, свойства Visible, ReadOnly и HelpText).

Класс CLabelCtrl используется для вывода статического текста.

Класс CTextCtrl позволяет вводить текстовые и числовые значения. Есть возможность валидации значений (контроль пределов для целых чисел и чисел с плавающей запятой, а также длины строки). Если предложенные варианты не устраивают, разработчик может реализовать свой вариант. В классе диалога присутствует обработчик события OnTextValidation, вызов которого предваряет стандартную валидацию. Внутри данного обработчика можно реализовать специализированный вариант проверки введенного значения.

После ввода пользователем значения в объекте диалога формируется вызов обработчика OnTextEdit, в котором можно реализовать какие-либо действия, связанные с данным значением.

Для работы с флажками предназначен класс CCheckBoxCtrl. При изменении состояния флажка происходит вызов обработчика OnCheckBoxClick.

Класс CListBoxCtrl используется для создания объектов, позволяющих пользователю выбирать строчку из предложенного списка. Внутри методов этого класса используются объекты класса CListBoxValue, скрытые от разработчика. При выборе строки формируется вызов обработчика OnListBoxBoxValueChange в объекте диалога.

Работа с кнопками осуществляется с использованием класса CButtonCtrl. При нажатии на кнопку происходит вызов обработчика OnButtonClick в объекте диалога.

Зачастую возникает необходимость вывода на экран сообщения с предупреждением или запроса подтверждения. Для этих целей используется класс CMessageBoxDialog, унаследованный от класса CDialog и реализующий необходимые функциональные возможности. Данный диалог содержит четыре кнопки: "Да", "Нет", "OK", "Отмена". В параметрах конструктора при создании объекта можно указать, какие кнопки будут использоваться. Метод DoModal возвращает константу, соответствующую нажатию одной из кнопок.

Заключение

Сформулированы и реализованы базовые (концептуальные) положения технологии TextControlPages, которая может быть использована при разработке интерфейсов для консольных приложений, не зависящих от компилятора и операционной системы. Эта технология является некоторым компромиссом между современными интерфейсными системами и полной независимостью кода от компилятора, операционной системы и оборудования.

Разработана одноименная библиотека классов на языке C++, реализующая описанные в статье принципы построения консольного интерфейса. В ходе эксплуатации библиотеки эффективность подхода была полностью подтверждена.

Список литературы

1. **Кузнецов М. В., Симдянов И. В.** MySQL 5. Наиболее полное руководство. СПб.: БХВ-Петербург, 2010. 1024 с.
2. **Хокинс С.** Администрирование Web-сервера Apache. М.: Вильямс, 2001. 336 с.
3. **Фаронов В. В.** Turbo Pascal. Наиболее полное руководство. СПб.: БХВ-Петербург, 2003. 1056 с.
4. **Strang J.** Programming with Curses. O'Reilly Media, 1986. 80 p.
5. **Троельсен Э.** Язык программирования C# 5.0 и платформа.NET 4.5. М.: Вильямс, 2013. 1312 с.
6. **Сидорина Т.** MFC и MS Visual Studio C++. Разработка приложений. СПб.: БХВ-Петербург, 2009. 848 с.
7. **Холингворт Дж., Сворт Б., Кэшмэн М., Густавсон П.** Borland C++ Builder 6. Руководство разработчика. М.: Вильямс, 2003. 976 с.

New Way to Construct Console Application Interfaces: Technology TextControlPages

P. A. Chentsov, chentsov.p@mail.ru, Institute of Mathematics and Mechanics Ural Branch of RAS, Yekaterinburg, 620219, Russian Federation

Corresponding author:

Chentsov Pavel A., Senior Researcher, Institute of Mathematics and Mechanics Ural Branch of RAS, 620219, Yekaterinburg, Russian Federation,
E-mail: chentsov.p@mail.ru

Received on August 23, 2016
Accepted on September 22, 2016

Console applications has a significant place in the world of information technology. This applications work in text mode. The main ways of its interaction with user are text commands or configuration files. This ways are not so convenient in comparison with the modern graphic interfaces. This paper presents a concept of console interfaces creation with high level of interactivity and usability. It based on the very common constructions - menu and dialogs. But this concept uses only few standard input and output functions. It provides high code portability. Moreover, was created C++ library, based on the stdio and conio standard libraries. Speaking strictly, it uses only three functions: printf, getc and gets. Every C++ compiler provides these functions. TextControlPages library allows to change text encoding in a program run, and works with large text constructions exceeding the dimensions of a console.

Keywords: *interface, menu, modal dialog, code portability, console application*

For citation:

Chentsov P. A. New Way to Construct Console Application Interfaces: Technology TextControlPages, *Programmnaya Ingeneria,* 2016, vol. 7, no. 12, pp. 539—546.

DOI: 10.17587/prin.7.539-546

References

1. **Kuznecov M. V., Simdjanov I. V.** *MySQL 5. Naibolee polnoe rukovodstvo* (MySQL 5. The most comprehensive guide), Saint-Petersburg, BHV-Peterburg, 2010, 1024 p. (in Russian).
2. **Howkins S.** *Apache Web server administration & e-commerce handbook*, Upper Saddle River, NJ, Prentice Hall, 2001, 359 p.
3. **Faronov V. V.** *Turbo Pascal. Naibolee polnoe rukovodstvo* (Turbo Pascal. The most comprehensive guide), Saint-Petersburg, BHV-Peterburg, 2003, 1056 p. (in Russian).
4. **Strang J.** *Programming with Curses,* O'Reilly Media, 1986, 80 p.
5. **Troelsen A.** *Pro C# 5.0 and the.NET 4.5 framework*, New York, NY Apress, 2012, 1487 p.
6. **Sidorina T.** *MFC i MS Visual Studio C++. Razrabotka prilozhenij* (MFC and MS Visual Studio C++. Applications development), Saint-Petersburg, BHV-Peterburg, 2009, 848 p. (in Russian).
7. **Hollingworth J., Swart B., Cashman M., Gustavson P.** *Borland C++ Builder 6 developer's guide,* Indianapolis, Sams, 2003, 1097 p.

УДК 004 DOI: 10.17587/prin.7.547-558

В. А. Васенин,[1, 2] д-р физ.-мат. наук, проф., e-mail: vasenin@msu.ru,
А. А. Иткес,[2] канд. физ.-мат. наук, науч. сотр., e-mail: itkes@imec.msu.ru,
В. Ю. Бухонов,[1] аспирант, e-mail: bukhonovvyu@gmail.com,
А. В. Галатенко,[1] канд. физ.-мат. наук, ст. науч. сотр., e-mail: agalat@msu.ru,
[1] Механико-математический факультет МГУ имени М. В. Ломоносова,
[2] НИИ механики МГУ имени М. В. Ломоносова

Модели логического разграничения доступа в многопользовательских системах управления наукометрическим контентом

Приведены результаты сравнительного анализа трех современных моделей логического разграничения доступа, а именно атрибутной модели (ABAC), сущностной модели (EBAC) и реляционной модели на основе цепочек отношений (ChRelBAC). Такие модели предназначены для многопользовательских информационных систем управления контентом, построенных на основе отношений "пользователь—пользователь", "пользователь—ресурс" и "ресурс—ресурс". Дана оценка возможности практического использования этих моделей в области наукометрии применительно к информационно-аналитической системе "ИСТИНА", которая разработана и эксплуатируется в МГУ им. М. В. Ломоносова. Эта система предназначена для сбора, систематизации и анализа результатов деятельности в крупных организациях науки и образования в целях подготовки и принятия управленческих решений.

Ключевые слова: *разграничение доступа, сущность, отношения, атрибуты, цепочки отношений, реляционная модель разграничения доступа на основе цепочек отношений, атрибутная модель разграничения доступа, сущностная модель разграничения доступа*

Введение

В современных многопользовательских информационных системах зачастую появляется необходимость ограничить доступ к определенным действиям с данными или к информации, полученной с помощью этих действий (далее для краткости изложения — доступов). Отношения, которые определяют такие ограничения в процессах контроля доступа, задаются определенным набором правил. В системах с большим числом пользователей, в условиях частого изменения представляющих ее атрибутов и связей между сущностями процесс формулирования и внедрения таких правил становится отдельной сложной наукоемкой задачей. К подобным системам могут быть отнесены, например, такие системы, которые обладают теми или иными характеристиками социальных сетей, в том числе — отдельные классы многопользовательских систем управления контентом [1].

В настоящее время для решения задач по контролю доступа уже разработаны некоторые модели логического разграничения доступа (ЛРД). К числу таких, хорошо описанных и широко используемых моделей, относятся традиционные модели — дискреционная [2], мандатная [2] и ролевая [3]. Нашли также применение на практике такие модели, как модель "песочницы" [4], модель фильтрации межсетевых экранов [5], объектная модель [6]. Тем не менее, ввиду постоянно расширяющегося спектра приложений и роста как числа, так и сложности информационных систем, увеличения объемов данных, с которыми эти системы должны работать, возникла необходимость в новых подходах к организации контроля доступа.

В ноябре 2009 г. американская организация Federal Chief Information Officers Council (Federal CIO Council) опубликовала документ под названием "Federal Identity, Credential, and Access Management (FICAM) Roadmap and Implementation Plan v1.0" [7], в котором содержались рекомендации для федеральных организаций по развитию архитектур систем разграничения доступа таким образом, чтобы включить оценку атрибутов в качестве способа управления доступом как внутри организаций, так и между ними во всех федеральных структурах США. В декабре 2011 г. был опубликован уже "FICAM Roadmap and Implementation Plan v2.0" [8], что ознаменовало собой следующий шаг в продвижении атрибутной модели (ABAC) в качестве рекомендованного на

государственном уровне средства контроля доступа при обмене информацией между различными организациями. В 2014 г. вышла специальная публикация NIST 800-162 "Guide to Attribute Based Access Control (ABAC) Definition and Considerations" [9], которая содержит рекомендации и разъяснения основных принципов ABAC.

В связи с появлением таких относительно новых информационных систем, как социальные сети, в которых контроль доступа в значительной степени зависит от отношений между различными информационными сущностями, активно начались разработки в области моделей ЛРД, основанных на методологии учета таких отношений. Одной из моделей, построенных на положениях подобной концепции, является сущностная модель EBAC [10] — Entity Based Access Control. Она была разработана коллективом ученых из Бельгии (Богаэртс, Декат, Лагайссе и Джусен) в 2015 г. Их подход основывается на дальнейшем развитии ABAC в сторону добавления отношений между субъектами и объектами.

Следует отметить, что кроме подхода EBAC существует и большое число иных моделей ЛРД, тоже основанных на концепции отношений. К их числу относят, например, семейство моделей ReBAC, разрабатываем Фонгом [11] и Карминати [12], RelBAC Жанга [13], а также разработанную и развиваемую авторами настоящей статьи реляционную модель логического разграничения доступа на основе цепочек отношений [14] — Chain-Relation Based Access Control (ChRelBAC). В связи с этим актуальным является сравнение функциональных возможностей подобных моделей с точки зрения их использования в тех или иных предметных областях.

В настоящей статье рассмотрены представленные в публикациях и имеющие результаты практического применения модели и программные механизмы ЛРД в системах с большими базами данных, где важное значение имеют отношения и связи между различными сущностями этих баз. К числу таких моделей относят атрибутную (ABAC), сущностную (EBAC) и реляционную (ChRelBAC) модели. В настоящей статье дан сравнительный анализ этих моделей и реализуемых на их основе программных механизмов. Целью такого анализа является сопоставление их возможностей при использовании в больших многопользовательских системах управления наукометрической информацией (далее, для краткости — предметная область).

Настоящая статья является продолжением публикации [1], в которой авторы выполнили анализ известных моделей ЛРД для социальных сетей на предмет их использования в наукометрических системах и сформировали требования к более подходящей модели ЛРД на примере одной из целевых систем, а также публикации [14], где авторы привели подробное описание модели ChRelBAC.

1. Логическое разграничение доступа в рассматриваемых системах

Определим особенности многопользовательских систем управления контентом в наукометрии на примере Интеллектуальной Системы Тематического Исследования НАучно-технической информации [15] (далее — целевой системы ИСТИНА) в вопросах логического разграничения доступа. Как было показано в работе [1], эта система имеет перечисленные далее общие с социальными сетями характеристики.

- Большое число пользователей в системе. В системе ИСТИНА, например, на настоящее время насчитывается уже несколько десятков тысяч пользователей и по ее назначению и степени востребованности ожидается дальнейшее увеличение их числа.
- Добавление новых пользователей возможно в автоматическом режиме, без участия администратора системы. Каждый пользователь системы при этом может добавлять объекты, связанные с другими объектами, как следствие, могут изменяться права доступа к объектам других пользователей. Например, при добавлении одним из пользователей в систему ИСТИНА новой публикации, имеющей других соавторов, эти соавторы также получают некоторые права доступа к информации об этой новой публикации, и, возможно, к информации о журнале, в котором она опубликована. Меняются также права доступа к объектам для пользователей, ответственных за подготовку и сопровождение информации по отдельным подразделениям организации, в которых работают авторы новой публикации.
- Отсутствие одного пользователя — владельца объекта, который должен иметь больше прав доступа к этому объекту, чем остальные пользователи. Например, если публикация имеет несколько авторов, то с точки зрения прав доступа к этой публикации все авторы должны быть равноправны.
- Важное значение с позиции процедуры разграничения доступа имеют не только объекты системы (пользователи и ресурсы), но и отношения между ними. В системе ИСТИНА для разграничения доступа используются не только значения атрибутов объекта (например, год выпуска статьи), но и отношения с другими объектами (например, отношения авторства между статьей и пользователем).

С точки зрения перечисленных позиций ИСТИНА адекватно отражает все основные особенности мировых наукометрических систем [15] и может быть использована в качестве примера, иллюстрирующего эти свойства. Для оценки и сравнения функциональных возможностей и выразительных свойств моделей ЛРД используются критерии, сгруппированные по следующим категориям.

1. Функциональные требования.

Требования, которые охватывают предполагаемое поведение системы (в рассматриваемом случае — системы контроля доступа, построенной на основе принципов выбранной модели ЛРД), определяя действия, которые эта система способна выполнять. Объектами системы при этом считаются как пользователи, так и ресурсы, субъектами — пользователи.

- Допускаются следующие виды отношений:
  - отношения между пользователями (*пользователь_1-[начальник]->пользователь_2; пользователь_3-[коллега]->пользователь_4; пользователь_5-[соавтор]->пользователь_6* и т. д.);

- отношения между пользователями и ресурсами (*пользователь_1-[является автором]->статья_1; пользователь_2-[работает в]->организация_1; пользователь_3-[создал]->отчет_1* и т. д.);
- отношения между ресурсами (*подразделение_1-[входит_в]->подразделение_2; подразделение_1-[входит в]->организация_1; статья_1-[опубликована в]->журнал_1* и т. д.);

♦ Отношения не обязаны быть симметричными (в системе ИСТИНА большинство отношений не являются симметричными —"является автором", "работает в", "входит в" и др.).

♦ При задании правил разграничения доступа должна предоставляться возможность учитывать атрибуты объектов — как пользователей, так и ресурсов, а также атрибуты отношений между ними. В системе ИСТИНА, например, может быть задано правило "пользователю (*U*), работающему в (*Rel*) подразделении (*D*), запрещено редактировать (*O*) ресурс (*R*)". Тогда для обработки запроса на доступ система должна определить значения следующих атрибутов объектов и отношений между ними: тип операции; имя пользователя; название подразделения; идентификатор ресурса и дату окончания работы пользователя в подразделении.

♦ При задании правил разграничения доступа должна предоставляться возможность учитывать параметры внешней среды (например, текущее время, местоположение, IP-адрес и т. д.).

♦ Должна предоставляться возможность ассоциировать с объектом не только разрешение, но и запрет на доступ к некоторым ресурсам.

♦ Пользователь должен иметь возможность управлять доступом к "своим" ресурсам. В наукометрических системах многие ресурсы имеют некоторого пользователя-владельца, а иногда даже нескольких пользователей-владельцев. Механизмы системы должны определять, кому разрешено или запрещено выполнять те или иные действия с ресурсом. В системе ИСТИНА, например, пользователь, который внес информацию о своем научном результате (публикация, участие в конференциях, участие в НИР и т. п.) считается владельцем этого объекта.

2. Формализм.

♦ Формальное описание модели должно быть "замкнуто" для использования математического аппарата при исследовании ее свойств и обозримо для визуального контроля модификаций.

♦ Процедуры по модификации и верификации программных механизмов, построенных на основе модели, при добавлении новых объектов, отношений, правил и т. п. не должны быть ресурсозатратными.

3. Эргономичность.

♦ Пользовательские характеристики (*usability*) модели должны обеспечивать легкость восприятия основных постулатов и правил модели конечным пользователем.

♦ Требования к формальному языку описания правил, реализующих политики информационной безопасности подлежащих защите систем (далее для краткости изложения — политики безопасности или просто политики) должны соответствовать положениям требований, представленных в работе [16], к числу которых относятся перечисленные далее.

- Язык должен поддерживать задание политик безопасности для разграничения доступа и делегирования полномочий на временную передачу прав доступа агенту, действующему от имени клиента, а также политик, реализующих элементы администрирования.
- Язык должен предоставлять способы структурирования требований к политикам, относящимся к большим системам с миллионами объектов. Такая "структурируемость" позволяет реализовывать политики, относящиеся не только к отдельным объектам, но и к группам объектов.
- Язык должен предоставлять возможности по описанию правил, реализующих составные политики безопасности. Составные политики создаются посредством группировки базовых политик безопасности (т. е. отвечающих требованиям на нижележащих уровнях структурной иерархии), относящихся к ролям, структурным подразделениям и к конкретным приложениям. Составные политики имеют важное значение для упрощения процедур администрирования в крупных информационных системах.
- Пользователю должна быть предоставлена возможность анализа правил, реализующих политики безопасности, на предмет наличия конфликтов и противоречий в требованиях к ним. Более того, должна присутствовать возможность определения того, какие политики применимы к некоторому объекту или к каким объектам применима выбранная политика. Следует отметить, что декларативные языки облегчают подобного рода анализ.
- Язык должен быть расширяем и способен поддерживать новые типы политик, потребность в которых может появиться в будущем. Это свойство, как правило, поддерживается объектно-ориентированными языками.
- Язык должен быть понятным и относительно легким для освоения конечными пользователями.
- Язык должен поддерживать задание политик безопасности для разграничения доступа и делегирования для временной передачи прав доступа агенту, действующему от имени клиента, а также задание политик, отражающих аспекты управленческой деятельности.

4. Доказательные возможности.

К доказательным относятся возможности использования моделей для математически строгих исследований свойств, построенных на их основе приложений. К числу таких исследований в том числе относятся механизмы проверки непротиворечивости правил, реализующих политики ЛРД. Один из способов, которым можно проводить подобные проверки, описан в работе [17], где авторы предложили способ верификации программной реализации модели ЛРД с помощью технологии model-checking [18]. Для этого сначала описывается структура модели разграничения доступа к ресурсам системы с помощью псевдо-

кода, формальных утверждений и правил. Затем на основе этого описания строится конечный автомат, который изменяет свое состояние при получении запроса на доступ от пользователя. С помощью этого автомата и набора тестов проводится верификация модели на предмет отсутствия противоречий.

К числу доказательных относятся возможности математически строгих способов описания моделей ЛРД, корректного их объединения и анализа на предмет выполнения ими требований политики безопасности, принятой в подконтрольной системе. Подходы к реализации таких возможностей описаны в работе [19].

Для некоторых реализаций ЛРД можно представить домен их модели в виде графа, что позволяет использовать инструментальные средства теории графов для проведения различного рода исследований. К числу подобных исследований, например, относятся поиск путей вычисления запросов на доступ, анализ их быстродействия (если добавить дугам графа веса), поиск неявных отношений и др.

5. Наличие свидетельств, подтверждающих общественное одобрение на разных уровнях.

Такими свидетельствами могут быть: апробация модели на различного рода форумах и конференциях, публикации в открытой печати, законодательные и нормативные документы, рекомендации разных организаций и сообществ.

2. Атрибутная модель разграничения доступа (Attribute Based Access Control, ABAC)

В настоящем разделе на основе материалов работы [9] представим определение атрибутной модели разграничения доступа и рассмотрим ее базовые концептуальные положения.

2.1. Определение ABAC

Модель ABAC — это модель логического разграничения доступа, согласно которой запросы субъекта на выполнение каких-либо операций с объектом разрешаются или отклоняются на основе результатов анализа: значений атрибутов субъекта; значений атрибутов объекта; условий среды выполнения запросов и анализа наборов политик, которые определяются правилами, сформулированными в терминах этих атрибутов и условий.

Далее в кратком изложении перечислены основные положения модели ABAC.

- Под *Attributes* понимают конечное множество атрибутов, *Attributes* = $\{attr_i\}$. **Атрибуты** — это характеристики субъекта, объекта или среды выполнения. Атрибуты содержат информацию, представленную в виде пары "имя—значение". Примеры атрибутов: *user.name* — имя пользователя; *magazine.article_count* — число статей в журнале; *article.publication_date* — дата публикации статьи и т. д.
- Под *Subjects* понимают конечное множество субъектов, *Subjects* = $\{subj_i\}$. **Субъект** — это пользователь (человек или NPE (*Non-Person Entity*) — устройство или процесс, который создает запросы на выполнение операций с объектами). Каждому субъекту поставлены в соответствие один или несколько атрибутов. В дальнейшем в рамках описания ABAC будем считать, что субъект и пользователь — синонимичные понятия.
- Под *Objects* понимают конечное множество объектов, *Objects* = $\{obj_i\}$. **Объект** — это ресурс подконтрольной системы, управление доступом к которому осуществляется с использованием ABAC-модели. К таким ресурсам относят устройства, файлы, записи, таблицы, процессы, программы, сети, а также домены, хранящие или получающие информацию. В предметной области (наукометрии) примеры объектов — статьи, журналы, организации и т. п.
- Под *Operations* понимают конечное множество операций, *Operations* = $\{op_i\}$. **Операция** — это выполнение субъектом по запросу некоторого действия над объектом. Множество операций в наукометрии включает в себя такие действия, как чтение (*read*), запись (*write*), редактирование (*edit*), удаление (*delete*), копирование (*copy*), выполнение (*execute*) и модификацию (*modify*).
- Под *Policies* понимают конечное множество политик, *Policies* = $\{pol_i\}$. **Политика** — это формальное представление требований в виде **правил** и/или **математических отношений**, которые определяют условия разрешения или отказа на запрашиваемый доступ, основываясь на оценке значений атрибутов субъекта, объекта и, возможно, на анализе условий внешней среды.
- Под *EnvironmentConditions* понимают конечное множество условий среды, *EnvironmentConditions* = = $\{env\_cond_i\}$. **Условия среды (или переменные среды)** — это операционный или ситуационный контекст, в котором выполняются запросы на доступ. Такой контекст определяется характеристиками внешней среды. Они независимы от субъекта или объекта и могут включать в себя текущее время, день недели, месторасположение пользователя или текущий уровень опасности.

Схема функционирования ABAC представлена на рисунке (см. третью сторону обложки). Здесь программный механизм контроля доступа (ACM) ABAC получает запрос субъекта на доступ, анализирует атрибуты субъекта и объекта по правилам некоторой заранее определенной политики. После этого ACM определяет, какие действия субъект может выполнять с запрашиваемым объектом.

2.2. Основные концептуальные положения ABAC

В самом общем представлении ABAC основывается на оценке атрибутов субъектов, атрибутов объектов, условий внешней среды, а также на анализе формальных отношений и анализе правил, задающих политики разграничения доступа. Все ABAC-решения в той или иной форме реализуют эти ключевые базовые возможности, позволяющие оценивать атрибуты и реализовывать обязательное выполнение принятых правил или отношений между этими атрибутами.

Даже в рамках небольшой изолированной системы ABAC основывается на привязке атрибутов к субъектам и объектам, а также на использовании политики, содержащей формально представленные правила

доступа. Каждому объекту в системе должны быть поставлены в соответствие специальные атрибуты, характеризующие этот объект. Некоторые атрибуты соответствуют всем экземплярам объекта целиком, например, такие как владелец. Другие атрибуты могут относиться только к отдельным частям объекта. Например, объект "Документ" (книга, сборник, статья и т. д.) может принадлежать организации *A*, содержать раздел с интеллектуальной собственностью *B* и быть частью реализации некоторого проекта организации *C*.

Каждому субъекту системы также должны соответствовать отражающие его специфику атрибуты. Рассмотрим пример, когда пользователь входит в систему. Этот пользователь определяется администратором безопасности (далее — администратор) как субъект системы, а характеристики этого пользователя записываются как атрибуты субъекта. Такими атрибутами могут быть имя, роль, должность, принадлежность к определенной организации, а также национальность, гражданство, уровень доверия. Эти атрибуты субъекта привязываются и управляются администраторами, которые предоставляют идентифицирующую информацию о субъекте в системе. С приходом новых и уходом старых пользователей меняются характеристики субъектов. Как следствие, появляется потребность в обновлении атрибутов этих субъектов.

Для каждого объекта в системе должна существовать хотя бы одна политика разграничения доступа к нему для определенных (допустимых) субъектов. Политика в ее неформализованном представлении обычно создается на основе правил, описывающих бизнес-процессы и разрешенные действия в организации. Например, в некотором университете такое правило может констатировать, что только его авторизованные сотрудники могут получить доступ к детальным данным по научно-исследовательским работам (НИР) своего университета. Тогда, если в системе выбирается поле "Полученное финансирование" для документа "НИР "Новые ИТ-разработки", привязанного к организации "МГУ", то в этом случае будет выбрано и применено правило "Правило для НИРов". В этом случае субъектам со значением "ВГУ" атрибута "Текущее место работы" при попытке осуществить операцию "чтение" с указанным НИР будет отказано в доступе и операция будет запрещена. Следует отметить, что это лишь один из способов создания связей между атрибутами и правилами.

Правила, которые связывают между собой атрибуты субъектов и объектов, неявным образом определяют **привилегии** (какой субъект, какие операции с каким объектом может выполнять). Правила о допустимости операций могут быть выражены различными способами их описания, например, такими как

- булево выражение, составленное из атрибутов и условий, которое удовлетворяет требованию авторизации для конкретной операции;
- конечное множество отношений, ассоциирующих атрибуты субъектов, объектов и допустимые операции.

После того как установлены атрибуты объектов, атрибуты субъектов и политики, ABAC можно применять для защиты объектов. Механизмы контроля доступа (Access Control Mechanisms, ACMs) выступают промежуточным звеном управления доступом к объектам, ограничивая доступ к разрешенным операциям для допустимых субъектов. Такие механизмы объединяют данные о политике, об атрибутах субъекта и об атрибутах объекта, затем эти данные обрабатываются, и выносится решение, основанное на принятой политике. У ACM должны быть возможности по управлению процессом, необходимым для формулирования и приведения в действие решения, включая также и механизмы определения: какую политику использовать; какие атрибуты задействовать; в каком порядке и откуда их получать. После этого ACM выполняет вычисления, необходимые для принятия решения о запросе.

Политики, которые могут быть выражены в модели ABAC, ограничены только степенью выразительности языка вычислений и числом доступных атрибутов. Такая гибкость позволяет огромному числу субъектов получать доступ к огромному множеству объектов без необходимости определять отдельные отношения между каждым субъектом и каждым объектом. Она достигается также за счет того обстоятельства, что атрибуты и их значения могут быть модифицированы в течение жизненного цикла субъектов, объектов и самих атрибутов.

Отметим, что составителю правил или владельцу объекта не требуется предварительная информация о каждом конкретном субъекте. При создании нового субъекта не появляется необходимость модифицировать уже существующие объекты и правила доступа. До тех пор пока субъекту поставлены в соответствие необходимые для доступа к объектам атрибуты, никаких модификаций существующих правил или атрибутов объектов не требуется. Это достоинство модели часто называют "размещение нового пользователя" и оно считается одним из главных положительных аспектов внедрения ABAC.

В отличие от некоторых иных схем, в модели ABAC у операций нет атрибутов. Согласно определению «Атрибуты хранят информацию в виде пар "ключ—значение"». Например, конструкция "read = all" (или "all = read") недопустима. Операции могут иметь много разных типов или классов, которые, тем не менее, не представляют собой атрибуты, а являются фиксированными множествами значений. В целом можно сделать операцию "именем атрибута", как, например "operation = read". Однако в этом случае имя будет единственным атрибутом отдельной сущности "операция", что является избыточным.

Для того чтобы модель соответствовала требованиям, предъявляемым к возможностям отчетности и протоколирования, должна существовать возможность отслеживания доступов к объектам конкретными субъектами, закрепленными за конкретными пользователями. Возможности отчетности могут потеряться, если решения о доступе выносятся на основе атрибутов, и при этом идентификаторы субъекта или пользователя не отслеживаются для определенных запросов и принятых решений.

3. Сущностная модель логического разграничения доступа (Entity Based Access Control, EBAC)

Сведения, представленные в настоящем разделе, основаны на материалах работы [10] и посвящены краткому описанию сущностной модели логического разграничения доступа. Приведены ее определение и основные концептуальные положения.

3.1. Определение EBAC

Модель логического разграничения доступа EBAC можно рассматривать как попытку развития модели ABAC путем добавления отношений "сущность-связь" (*ER*-отношений) в выражения ее политик разграничения доступа. С учетом этого обстоятельства можно сформулировать следующее ее определение.

Модель EBAC — это модель логического разграничения доступа, которая как и ABAC использует атрибуты субъектов и объектов, переменные среды исполнения, множество правил, которые задают набор политик для определения возможности осуществления запрашиваемой субъектом операции. При этом EBAC включает в себя элементы *ER*-отношений. Этот факт означает, что при составлении логических выражений (в том числе и кванторных) для правил и политик системы, в которой используется EBAC, учитываются как сущности — субъекты и объекты, так и отношения между ними.

Учет отношений между сущностями также позволяет утверждать, что EBAC имеет общие черты с моделями ЛРД на основе отношений (*Relation-Based Access Control*, ReBAC). Модели ReBAC [11, 12] представляют семейство моделей ЛРД, которые были разработаны специально для решения задач контроля доступа к данным конечных пользователей социальных сетей. Модели ReBAC используют отношения между такими пользователями, чтобы на основе их анализа принимать решения о возможности выполнения некоторой операции. Однако следует отметить, что в этом случае не поддерживается (или только частично поддерживается) использование свойств конечных пользователей и отношений между ними. Как следствие, это может вызвать ряд затруднений, причем довольно серьезных, если возникает необходимость разграничивать доступ на основе значений некоторых конкретных свойств (например, временные атрибуты). Более того, большинство представленных ReBAC-моделей не учитывают иных сущностей, кроме субъектов и объектов при определении правил контроля доступа. Таким образом, можно сделать вывод, что в EBAC обобщаются подходы и ABAC, и ReBAC.

3.2. Основные концептуальные положения EBAC

Граф сущностей: типы сущностей и отношений в EBAC могут быть представлены в виде направленного помеченного графа $G(V, E)$, в котором вершины являются типами сущностей, а дуги — типами отношений.

Выражения позволяют правилам EBAC поддерживать сравнения между атрибутами и значениями. Выражения могут возвращать только *true* или *false*. Модель EBAC поддерживает следующие три типа выражений:

- *простые выражения*, которые отображают отношение между двумя атрибутами;
- *составные выражения*, в которых выражения комбинируются с помощью логических связок (*and*, *or*, *not*);
- *кванторные выражения*, для которых вводятся кванторные функции, вычисляемые над свойствами и путями отношений произвольной длины. Тогда необходимо определить *селекторы путей* для того, чтобы обращаться к атрибутам в этих выражениях. Запросы на доступ включают в себя данные *субъекта, объекта, действия* и контекста, в котором запрос выполняется (представлен как *среда выполнения*). Следовательно, чтобы получить значения атрибутов, к ним нужно обращаться через пути, начинающиеся с одной из этих четырех перечисленных сущностей. Однако, в отличие от ABAC, EBAC поддерживает просмотр путей отношений произвольной длины, используя селекторы путей.

С неформальной точки зрения селектор пути ссылается на атрибут посредством задания пути из типов отношений, заканчивающегося атрибутом финальной сущности. Применительно к наукометрии, например, *"article.author.id"* обращается к идентификатору *автора статьи*.

Более формально, пусть S — конечное множество экземпляров сущностей, тип которых — один из элементов множества {*subject, object, action, environment*}. *Селектор пути* — это кортеж (s, P, k), в котором:

- экземпляр сущности $s \in S$ — это начальный узел, с которого начинается обход пути;
- конечный упорядоченный список $P = (p_1, p_2, \ldots p_n)$ содержит путь, составленный из типов отношений (каждый встречается только раз), который обходится для того чтобы получить доступ к атрибуту; элементы пути при этом должны соответствовать меткам дуг в указанном порядке;
- ключ атрибута k, который извлекается из адресуемой сущности в конце пути P.

Селекторы путей позволяют задавать требования для получения выборок нужных атрибутов с использованием путей отношений.

После определения селекторов путей рассмотрим типы кванторных выражений, использующихся в модели EBAC.

♦ *Горизонтальные кванторные выражения*. Типы сущностей могут определять отношения, которые встречаются более чем один раз. Аналогично, сущности могут иметь многозначные атрибуты. Модель EBAC поддерживает кванторы, чтобы осуществлять вычисления с сущностями, которые адресуются подобного рода атрибутами. Поддерживаются также и вычисления с сущностями, которые адресуются отношениями с арностью больше единицы.

Для того чтобы поддерживать отмеченные выше свойства, в EBAC используются *частичные выражения*, ссылки на которые осуществляются внутри кванторов ($\forall$ или $\exists$). Частичные выражения — это выражения, которые содержат *анонимные* селекто-

ры путей. Если делать вызовы в обычном режиме, то каждый селектор пути, по которому выбирается атрибут, содержит стартовый узел. Этот стартовый узел является элементом множества S (множество всех экземпляров типов сущностей, которые представляют *субъект, объект, действие* или *внешнюю среду*). Вместе с тем анонимные селекторы путей могут иметь любой экземпляр в качестве стартового узла. Это позволяет параметризовывать частичные выражения путем комбинирования анонимных селекторов путей с путем, начинающимся с элемента $i_v \in S$.

Более формально, *горизонтальные кванторные выражения* определяют как кортежи (*attr, pexpr,* $f \in \{\forall, \exists\}$).

- Элемент *attr* представляет собой последовательность значений *Seq* или сущности, которые адресуются через отношение с арностью больше 1.
- Элемент *pexpr* — это частичное выражение, которое параметризуется каждым элементом $e \in Seq$. Заметим, что анонимные селекторы путей должны строиться на основе сущностей того же типа, что и элементы $e \in Seq$.
- Функция $f \in \{\forall, \exists\}$ — это тип квантора, используемого в выражении.

♦ *Вертикальные кванторные выражения*. Горизонтальные кванторные выражения позволяют обрабатывать отношения для некоторой конкретной сущности. При этом они не поддерживают обработку рекурсивных отношений. Чтобы обеспечить такую обработку, EBAC поддерживает *вертикальные* кванторные выражения. Они вычисляются для каждой сущности, встречающейся в пути, который формируется посредством рекурсивного обхода для всех типов каждой из этих сущностей.

Более формально вертикальные кванторные выражения можно определить как кортежи (*es, P, pexpr,* $f \in \{\forall_\rho, \exists_\rho\}$).

- Элемент *es* определяет начальный селектор сущностей. Селекторы сущностей, как и селекторы путей, определяют путь, по которому можно обращаться к сущности. Однако в отличие от селекторов путей, селекторы сущностей возвращают в конце пути не атрибут, а всю сущность целиком.
- Путь отношений $P = (p_1, p_2, ..., p_n)$ — это конечный непустой упорядоченный список, определяющий путь, который нужно просмотреть на каждом шаге, чтобы задать значения параметров частичного выражения *pexpr*. Последний элемент p_n всегда оканчивается на одной или более сущности того же типа, что и *es*.
- Частичное выражение *pexpr*, значения параметров которого задаются каждой сущностью из пути *P*.
- Функция $f \in \{\forall_\rho, \exists_\rho\}$, где $\forall_\rho$ — квантор всеобщности и $\exists_\rho$ — квантор существования.

Опционально вертикальные кванторные выражения могут быть расширены переменными min, max $\in N$, min < max, которые отображают минимальную и максимальную глубину используемого пути.

4. Реляционная модель логического разграничения доступа на основе цепочек отношений (Chain-Relation Based Access Control, ChRelBAC)

Модель Chain-Relation Based Access Control (ChRelBAC) — это модель логического разграничения доступа, согласно которой запросы субъекта на выполнение каких-либо операций с объектом разрешаются или отклоняются на основе правил и политик, составленных из утверждений об объектах (субъектах и ресурсах) и их атрибутах, об условиях внешней среды, а также об отношениях между объектами, атрибутами этих отношений и о цепочках отношений.

Далее, в кратком изложении представим основные сущности модели ChRelBAC.

- **Атрибуты** — это характеристики *объектов* или *среды выполнения*. Атрибуты представляют собой пары "ключ—значение".
- **Объекты** — это информационные сущности, связанные между собой некоторыми отношениями. Объектами в модели ChRelBAC могут быть *ресурсы* и *субъекты*. Объекты всегда принадлежат определенному *классу*.
- **Класс** — это абстрактная информационная сущность, описывающая *объект* в общем, а именно — какими характеристиками и методами он может обладать. Отдельно подчеркнем, что принадлежность объекта к классу определяет и то, в каких *отношениях* он может быть с другими объектами. Это обстоятельство имеет высокую значимость в рамках рассматриваемой модели ChRelBAC. Классы, как и в объектно-ориентированных языках программирования, организованы в *иерархию наследования*. Имена классов используются в описаниях отношений и их цепочек, правил и политик.
- **Субъект —** это активный агент рассматриваемой системы, т. е. та сущность, которая выполняет действия, предполагающие осуществление доступа к некоторым *ресурсам*. Субъектами могут быть пользователи, устройства или процессы, выступающие от имени этих пользователей, и другие сущности.
- **Ресурс** — это пассивная информационная сущность, хранящая некоторую содержательную информацию. Примеры ресурсов: статья, книга, журнал, научный отчет и т. д.
- **Отношение** — это определенная связь между двумя *объектами (субъектами* и *ресурсами)*. В рамках модели ChRelBAC рассматривают *параметризованные именованные несимметричные* отношения, часть которых являются *транзитивными*. **Тип отношения** определяется его именем, а также тем, объекты каких *классов* могут быть на *"правом конце"* и *"на левом конце"*. Например:

сотрудник — [работает_в]-> подразделение.

У отношения всегда есть как минимум один атрибут, а именно — имя отношения, однако могут быть также и другие атрибуты. Например, у отмеченного выше отношения "*работает в*" между объектами *"сотрудник"* и *"подразделение"* могут быть атрибуты

"дата начала работы", "дата увольнения", "должность" и т. п.

• **Цепочка отношений** — это упорядоченная структура, составленная из отношений и описываемая следующим образом:

cls_1 —[rel_1_2]—> cls_2 <—[rel_2_3:cond_2_3]— cls_3 ... —[rel_n-1_n]—> cls_n = >cls_1 —[rel_1_n]—> cls_n,

где *cls_1, cls_2, ... cls_n* — имена классов объектов; *rel_1_2, rel_2_3, ... rel_n-1_n* — отношения; часть до знака => называется *телом цепочки,* а отношение *rel_1_n*, стоящее после знака =>, — *порожденным отношением*. Смысловая нагрузка этой структуры заключается в следующем: если для некоторого объекта *obj_1* класса *cls_1* существует объект *obj_2* класса *cls_2,* связанный с ним отношением *rel_1_2*, а для объекта *obj_2* существует объект *obj_3* класса *cls_3*, связанный с ним отношением *rel_2_3*, и для этих объектов и отношения выполняется условие *cond_2_3*, и если продолжать проход по цепочке аналогичным образом для всех указанных в ней классов, отношений и подходящих объектов вплоть до объекта *obj_n* класса *cls_n,* то объекты *obj_1* класса *cls_1* и *obj_n* класса *cls_n* считаются связанными отношением *rel_1_n*. Следует отметить, что в качестве отношения в звене цепочки может использоваться порожденное другой цепочкой отношение.

• **Политика** — это формальное представление требований в виде правил, которые определяют условия разрешения или запрета на выполнение запрашиваемой операции.

Таким образом можно сформулировать следующее формальное определение модели ChRelBAC.

Определение. Пусть в информационной системе заданы следующие конечные множества:

- *Objects* — множество объектов;
- *Subjects* ⊂ *Objects* — множество субъектов;
- *Users* ⊂ *Subjects* — множество пользователей;
- *Resources* ⊂ *Objects* — множество ресурсов;
- *Relations* — множество отношений;
- *PrimitiveRelations* ⊂ *Relations* — множество примитивных отношений;
- *InducedRelations* ⊂ *Relations* — множество порожденных отношений;
- *TransitiveRelations* ⊂ *PrimitiveRelations* — множество транзитивных отношений;
- *Classes* — множество классов;
- *Types* — множество типов отношений;
- *Privileges* — множество привилегий;
- *Actions* — множество операций над объектами;
- *Chains* — множество допустимых цепочек отношений;
- *EnvironmentConditions* — множество условий среды исполнения;
- *Policies* — множество политик;

и следующие отношения:

- *ObjectRelations* ⊂ *Objects* × *PrimitiveRelations* × × *Objects* — отношение, определяющее то, какие из объектов системы связаны тем или иным примитивным отношением;
- ⩽ ⊂ *Classes* × *Classes* — отношение, определяющее иерархию классов;
- *RelationClassAccessGranted* ⊂ *Relations* × *Classes* × × *Actions* — отношение, определяющее, какие действия с объектом определенного класса может совершать пользователь, связанный с ним определенным отношением;
- *RelationClassAccessDenied* ⊂ *Relations* × *Classes* × × *Actions* — отношение, определяющее, какие действия пользователь не может совершать с объектом определенного класса в зависимости от отношений между этим пользователем и объектом;
- *ClassAccess* ⊂ *Classes* × *Actions* — отношение, определяющее, какие операции допустимы для объектов каждого класса;

а также следующие отображения:

- *ObjectClass: Objects* ⊂ *Classes* — отображение, определяющее класс каждого объекта;
- *RelationLeftClass: Relations* ⊂ *Classes* — отображение, определяющее класс объектов, которые могут быть связаны с другими объектами определенным отношением;
- *RelationRightClass: Relations* ⊂ *Classes* — отображение, определяющее класс объектов, которые могут быть связаны с другими объектами определенным отношением;
- *ChainInduces: Chains* ⊂ *InducedRelations* — отображение, определяющее отношение, порожденное каждой из цепочек.

Тогда будем считать, что в системе задана **реляционная модель логического разграничения доступа к ее ресурсам на основе цепочек отношений — Chain-Relation Based Access Control (ChRelBAC)**.

Главными элементами модели ChRelBAC, вокруг которых выстраивается подход к организации контроля доступа, являются отношения между объектами и цепочки этих отношений. Модель позволяет учитывать не только весь спектр свойств объектов, все их атрибуты, но и отношения между ними, свойства этих отношений, свойства внешней среды выполнения. Таким образом, в полном объеме учитываются сведения о сущностях системы, на основе которых реализована модель ChRelBAC. Данный подход позволяет формулировать политики безопасности на основе правил, составленных из логических утверждений об объектах и отношениях между ними. Подобного рода конструкции гораздо более логичны и понятны конечному пользователю, поскольку наиболее полно отражают реально существующую административную структуру организации, в которой работает информационная система, а также ее внутренние бизнес-процессы. Например, конструкция *"сотрудник работает в подразделении"* гораздо более понятна по смыслу и легче читается, нежели аналогичные *«объект "Сотрудник" -> объект "Работа" -> объект "Подразделение"»*. Использование цепочек, в свою очередь, позволяет кратко описывать сложные последовательности отношений, что делает правила более гибкими и компактными.

5. Сравнительный анализ возможностей ABAC, EBAC и ChRelBAC

В настоящем разделе представлены результаты сравнительного анализа моделей с позиции заранее определенных в разд. 1 критериев, сгруппированных

в категории. Знаки "+" и "–" в табл. 1 и табл. 2 означают соответственно факт выполнения или невыполнения отдельного функционального требования и требования к формальному языку описания правил.

1. Функциональные требования (см. табл. 1).

Сравнительный анализ данных, представленных в табл. 1, позволяет сделать вывод, что и модель ABAC, и модель EBAC, и модель ChRelBAC:

— поддерживают создание правил разграничения доступа с использованием атрибутов объектов (субъектов и ресурсов) и параметров внешней среды;

— позволяют ассоциировать с пользователем не только разрешение, но и запрет на выполнение некоторого действия с объектом;

— позволяют управлять доступом к "своим" ресурсам.

Главное отличие между этими моделями состоит в том, что модель ABAC, в отличие от моделей EBAC и ChRelBAC, не позволяет описывать явные отношения. В свою очередь, разница между моделями ChRelBAC и EBAC с точки зрения выполнения функциональных требований предметной области

Таблица 1

Сравнение моделей ABAC, EBAC и ChRelBAC на предмет выполнения функциональных требований

Функциональные требования		ABAC	EBAC	ChRelBAC
Объектами системы считаются как пользователи, так и ресурсы, субъектами — пользователи		+	+	+
Допускаются разные виды отношений	Отношения между пользователями	–	+	+
	Отношения между пользователями и ресурсами	–	+	+
	Отношения между ресурсами	–	+	+
Отношения не обязаны быть симметричными		–	+	+
При задании правил разграничения доступа должна присутствовать возможность учитывать атрибуты отношений между объектами		–	–	+
При задании правил разграничения доступа должна присутствовать возможность учитывать атрибуты объектов — как пользователей, так и ресурсов		+	+	+
При задании правил разграничения доступа должна присутствовать возможность учитывать параметры внешней среды		+	+	+
Должна присутствовать возможность ассоциировать с пользователем не только разрешение, но и запрет на доступ к некоторым ресурсам		+	+	+
Пользователь должен иметь возможность управлять доступом к "своим" ресурсам		+	+	+

Таблица 2

Сравнение моделей ABAC (на примере XACML [20]), EBAC (на основании описания языка из работы [10]) и ChRelBAC (описание языка представлено в работе [14]) на предмет выполнения требований к формальному языку описания правил

Требования к формальному языку описания правил	ABAC	EBAC	ChRelBAC
Язык должен поддерживать задание политик безопасности для разграничения доступа и делегирования полномочий на временную передачу прав доступа агенту, действующему от имени клиента, а также политик, реализующих элементы администрирования	+	+	+
Язык должен предоставлять способы структурирования требований к политикам, относящимся к большим системам с миллионами объектов	+	+	+
Язык должен предоставлять возможности по описанию правил, реализующих составные политики безопасности	+	+	+
Пользователю должна быть предоставлена возможность анализа политик на предмет наличия конфликтов и противоречий в требованиях. Более того, должна присутствовать возможность определения того, какие политики применимы к некоторому объекту или к каким объектам применима выбранная политика	+	+	+
Язык должен быть расширяемым и способным поддерживать новые типы политик, потребности в которых могут возникнуть в будущем	+	+	+
Язык должен быть понятным и легким для освоения конечными пользователями политик	+	+	+
Язык должен поддерживать задание политик безопасности для разграничения доступа и делегирования для временной передачи прав доступа агенту, действующему от имени клиента, а также политик, выражающих управленческую деятельность	+	+	+

заключается в том, что ChRelBAC поддерживает использование атрибутов отношений при задании правил, а EBAC не поддерживает.

2. Формализм.

С позиции оценки возможности математически строгого, "замкнутого" описания можно сделать следующие выводы.

- Будучи хорошо проработанными с концептуальных позиций с достаточно обширным набором базовых сущностей, модели ABAC, EBAC и ChRelBAC обладают необходимыми возможностями для описания даже сложно организованной предметной области в рамках определений, принятых в каждой из этих моделей. Определенная слабость модели ABAC заключается в том, что она не позволяет описать отношения в явном (регламентированном) виде, в то время как в моделях EBAC и ChRelBAC реализованы концептуальные положения, позволяющие описать как отношения, так и сложные структуры на их основе, а именно — пути в EBAC и цепочки в ChRelBAC.
- И модель ABAC, и модель EBAC, и модель ChRelBAC не требуют больших затрат ресурсов, как аппаратно-программных, так и административных, при добавлении в систему новых пользователей, объектов, отношений и правил.

3. Эргономичность.

- Базовые положения ABAC, EBAC и ChRelBAC хорошо проработаны и относительно легки для понимания конечными пользователями. Сложность восприятия политик и реализующих их правил доступа напрямую зависит от реализации модели конкретным поставщиком программных средств контроля доступа. Однако в случае ABAC некоторые правила получаются избыточно сложными, поскольку отсутствует механизм учета реляционных отношений между субъектами и объектами. В EBAC и ChRelBAC этот недостаток отсутствует. Использование путей в EBAC и цепочек отношений в ChRelBAC позволяет сделать описание правил понятным и компактным. Вместе с тем следует отметить, что концепция путей EBAC не позволяет повторно использовать результаты одного пути в теле другого, в отличие от цепочек ChRelBAC, где порожденное одной цепочкой отношение может быть использовано в теле другой.
- Сравнительный анализ требований к формальному языку описания правил в моделях ABAC, EBAC и ChRelBAC, представленный в табл. 2, свидетельствует о том, что по этой категории критериев они равноценны.

4. Доказательные возможности.

Для политик, описанных в рамках базовых положений модели ABAC, для политик, описанных с использованием подходов модели EBAC, а также для политик, описанных в рамках методологии ChRelBAC, можно построить конечный автомат, изменяющий свое состояние при получении запроса на доступ от пользователя. Анализ состояний этого автомата позволяет проверить отсутствие противоречий в требованиях принятых политик разграничения доступа. Для этих моделей можно также использовать предложенные в работе [19] способы их описания, объединения и анализа на предмет выполнения определенных заранее требований информационной безопасности.

Поскольку модели EBAC и ChRelBAC допускают представление домена в виде графа, то для него можно применять теоремы из теории графов. Например, можно осуществлять поиск неявных связей между объектами, которые могут потенциально привести к несанкционированным доступам. Или, например, можно "взвесить" дуги графа и таким образом оценить быстродействие вычисления запроса на доступ. Кроме того, для модели ChRelBAC можно проводить проверки на предмет отсутствия циклов отношений и отсутствия ссылок цепочек самих на себя, когда порожденное цепочкой отношение используется в этой же цепочке.

5. Наличие отражающих общественное одобрение рекомендаций и/или нормативных документов на разных уровнях.

- Модель ABAC была разработана и до настоящего времени поддерживается организацией NIST, которая выпустила документ "NIST Special Publication 800-162 "Guide to Attribute Based Access Control (ABAC) Definition and Considerations", в котором содержится общее описание основных концепций ABAC и рекомендации по вариантам ее использования. Этой же организацией проводятся конференции для компаний, использующих или разрабатывающих системы на основе методологии ABAC. Однако канонического формального определения этой модели в открытых источниках пока представлено не было.
- Модель EBAC была разработана коллективом авторов из Бельгии — Богаэртсом, Декатом, Лагайссе и Джусеном — и представлена на конференции "31st Annual Computer Security Applications Conference". К настоящему времени представлена лишь одна статья [10] с описанием данной модели. Создана программная реализация, однако она давно не обновлялась.
- Модель ChRelBAC разрабатывалась коллективом авторов данной статьи, для нее проработано формальное описание, создано уже несколько программных реализаций. Опубликованы научные статьи, посвященные этой модели. Она была представлена на различных конференциях, в том числе и на международных.

Заключение

Представлены результаты сравнительного анализа возможностей новой модели логического разграничения доступа ChRelBAC с возможностями моделей ABAC и EBAC в приложении к выбранной предметной области — наукометрии. В качестве критериев и требований для такого анализа использовались:

1) функциональные требования;
2) возможности математически строгого "замкнутого" описания;
3) требования к эргономичности;
4) доказательные возможности;
5) наличие отражающих общественное одобрение рекомендаций и/или нормативных документов на разных уровнях.

Результаты такого анализа могут быть полезны исследователям, которые занимаются разработкой

программного обеспечения моделей логического разграничения доступа к ресурсам в социальных сетях.

Описаны основные предпосылки к созданию новой модели ChRelBAC, которые сформировались на основе опыта, полученного в ходе трехлетней эксплуатации системы ИСТИНА, которая адекватно представляет все особенности многопользовательских систем управления контентом в области наукометрии. В подобного рода системах, как и в социальных сетях, немалое значение имеют отношения между объектами (субъектами и ресурсами), которые приходится принимать во внимание при задании правил разграничения доступа.

Показано, что по основным характеристикам, таким как полнота множества объектов системы, простота и логичность концептуальных положений модели, легкость освоения и богатство языка ее формального описания, по доказательным возможностям, новая модель ChRelBAC не только не уступает другим моделям, но в некоторых вопросах и превосходит их. Показано также, что за счет введения особых механизмов (отношений, атрибутов отношений, цепочек отношений) модель ChRelBAC гораздо лучше подходит для решения задач в сфере наукометрии и в иных предметных областях, схожих с ней по технологии работы с данными.

Список литературы

1. **Васенин В. А., Иткес А. А., Шапченко К. А.** О применении моделей разграничения доступа в социальных сетях к одному классу многопользовательских систем управления контентом // Программная инженерия. 2015. № 4. С. 10—19.
2. **U. S. Department** of Defense, Information Assurance (IA), DoD Directive 8500.1. U. S. Department of Defense, Washington, D. C., 24.10.2002. 25 p.
3. **International** Committee for Information Technology Standards, American National Standard for Information Technology — Role Based Access Control, ANSI/INCITS 359-2012. American National Standards Institute, New York, 29.05.2012. 56 p.
4. **Goldberg I., Wagner D., Thomas R., Brewer E. A.** A Secure Environment for Untrusted Helper Applications // Proceedings of the Sixth USENIX UNIX Security Symposium. 25.10.2011. 13 p.
5. **Барбиш Д., Давис Б.** Руководство FreeBSD. Часть IV. Сетевые коммуникации. Глава 26. Межсетевые экраны. URL: https://www.freebsd.org/doc/ru/books/handbook/firewalls.html
6. **Галатенко А. В., Галатенко В. А.** К постановке задачи разграничения доступа в распределенной объектной среде // Программная инженерия. 2013. № 5. С. 27—30.
7. **Federal** Chief Information Officers Council, Federal Identity, Credential, and Access Management (FICAM) Roadmap and Implementation Guidance (Version 1.0). Office of Management and Budget, Washington, D. C., 10.11.2009. 220 p.
8. **Federal** Chief Information Officers Council, Federal Identity, Credential, and Access Management (FICAM) Roadmap and Implementation Guidance (Version 2.0). Office of Management and Budget, Washington, D. C., 02.12.2011. 478 p.
9. **Hu V. C., Ferraiolo D., Kuhn R.** et. al. NIST Special Publication 800-162, Guide to Attribute Based Access Control (ABAC) Definition and Considerations, 2014. 37 p.
10. **Bogaerts J., Decat M., Lagaisse B., Joosen W.** Entity-Based Access Control: supporting more expressive access control policies // ACSAC 2015 — Proceedings of the 31st Annual Computer Security Applications Conference, 2015. P. 291—300.
11. **Fong P. W.** Relationship-Based Access Control: Protection Model and Policy Language // Proceedings of the first ACM conference on Data and application security and privacy, 2011. P. 191—202.
12. **Carminati B., Ferrari E., Perego A.** Rule-based access control for social networks // OTM Meaningful Internet Systems 2006, Springer, 2006. P. 1734—1744.
13. **Zhang R.** RelBAC-Relation-Based Access Control, PhD dissertation, 2009. 145 p.
14. **Васенин В. А., Иткес А. А., Шапченко К. А., Бухонов В. Ю.** Реляционная модель логического разграничения доступа на основе цепочек отношений // Программная инженерия. 2015. № 9. С. 11—19.
15. **Садовничий В. А., Афонин С. А., Бахтин А. В.** и др. Интеллектуальная система тематического исследования научно-технической информации ("ИСТИНА"). М.: Изд-во Мос. ун-та, 2014. 262 с.
16. **Damianou N., Dulay N., Lupu E., Sloman M.** The Ponder Policy Specification Language // Workshop on Policies for Distributed Systems and Networks. 29—31.01.2001, Bristol, UK. Springer-Verlag LNCS, 2001, Vol. 1995. P. 18—39
17. **Hu V. C., Kuhn D. R., Xie T.** Property Verification for Generic Access Control Models // Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on (Volume 2), 17—20.12.2008. P. 243—250.
18. **Карпов Ю. Г.** MODEL CHECKING. Верификация параллельных и распределенных программных систем. СПб.: БХВ-Петербург, 2010. 560 с.
19. **Шапченко К. А.** Методы и программные средства исследования моделей логического разграничения доступа на предмет выполнения требований по безопасности: дисс. ... канд. физ.-мат. наук. МГУ им. М. В. Ломоносова, Москва, 2010.
20. **OASIS.** eXtensible Access Control Markup Language (XACML) Standard, Version 3.0. URL: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf, 2013.

Access Control Models in Multiuser Scientometric Content Management Systems

V. A. Vasenin[1, 2], e-mail: vasenin@msu.ru, **A. A. Itkes**[2], e-mail: itkes@imec.msu.ru,
V. Yu. Bukhonov[1], e-mail: bukhonovvyu@gmail.com, **A. V. Galatenko**[1], e-mail: agalat@msu.ru,
[1] Faculty of Mechanics and Mathematics, Lomonosov Moscow State University, Moscow, 119991, Russian Federation,
[2] Scientific Research Institute of Mechanics, Lomonosov Moscow State University, Moscow, 119192, Russian Federation

Corresponding author:

Bukhonov Vladimir Yu., Postgraduate Student, Faculty of Mechanics and Mathematics, Lomonosov Moscow State University, Moscow, 119991, Russian Federation
E-mail: bukhonovvyu@gmail.com

Received on August 02, 2016
Accepted on September 05, 2016

This paper presents the results of the gap analysis of three modern access control models — Attribute Based Access Control (ABAC), developed by Hu, Ferraiolo, Kuhn, Schnitzer, Sandlin, Miller, Scarfone from NIST; Entity Based Access Control (EBAC), developed by Bogaerts, Decat, Lagaisse, Joosen; Chain-Relation Based Access Control (ChRelBAC), developed by authors of these paper. The short descriptions, including formal definitions and basic concepts, are given for each model. As such, in this paper the authors analyze them in reference to content management in multiuser systems, based on "user-user", "user-resource" and "resource-resource" relations. Results of the analysis are presented in this work.

The capabilities of the models' practical usage in Scientometrics are compared as applied to information-analytical system "ISTINA". Main functions of this system include gathering and processing information about the results of scientific and educational activities in big organization for management decision-making. The experience of working with "ISTINA" as a typical multiuser scientometric content management system has allowed the authors to formulate requirements and criteria, which were used for comparative analysis of ABAC, EBAC and ChRelBAC.

Keywords: *access control, entity, relations, attributes, chains of relations, Chain-Relation Based Access Control, Attribute Based Access Control, Entity Based Access Control*

For citation:

Vasenin V. A., **Itkes A. A., Bukhonov V. Yu.**, **Galatenko A. V.** Access Control Models in Multiuser Scientometric Content Management Systems, *Programmnaya Ingeneria,* 2016, vol. 7, no. 12. pp. 547—558.

DOI: 10.17587/prin.7.547-558

References

1. **Vasenin V. A., Itkes A. A., Shapchenko K. A.** O primenenenii modeley razgranicheniya dostupa v socialnikh setyakh k odnomu klassu mnogopolzovatelskikh system upravleniya contentom (Social Network Access Control Models in Multiuser Content Management Systems), *Programmnaya Ingeneria*, 2015, no. 4, pp. 10—19 (in Russian).

2. **U. S. Department** of Defense, Information Assurance (IA), DoD Directive 8500.1, U. S. Department of Defense, Washington, D. C., 24.10.2002, 25 p.

3. **International** Committee for Information Technology Standards, American National Standard for Information Technology — Role Based Access Control, ANSI/INCITS 359-2012, American National Standards Institute, New York, 29.05.2012, 56 p.

4. **Goldberg I., Wagner D., Thomas R., Brewer E. A.** A Secure Environment for Untrusted Helper Applications, *Proceedings of the Sixth USENIX UNIX Security Symposium*, 25.10.2011, 13 p.

5. **Barbish J. J., Davis B.** FreeBSD Handbook Chapter 29. Firewalls Part IV. Network Communication, available at: https://www.freebsd.org/doc/en/books/handbook/firewalls.html

6. **Galatenko A. V., Galatenko V. A.** K postanovke zadachi razgranicheniya dostupa v raspredelennoy obyektnoy srede (Discussing Definition of Problems of Access Control in Distributed Object System), *Programmnaya Ingeneria,* 2013, no. 5, pp. 27—30 (in Russian).

7. **Federal** Chief Information Officers Council, Federal Identity, Credential, and Access Management (FICAM) Roadmap and Implementation Guidance (Version 1.0). Office of Management and Budget, Washington, D. C., 10.11.2009, 220 p.

8. **Federal** Chief Information Officers Council, Federal Identity, Credential, and Access Management (FICAM) Roadmap and Implementation Guidance (Version 2.0). Office of Management and Budget, Washington, D. C., 02.12.2011, 478 p.

9. **Hu V. C., Ferraiolo D., Kuhn R., Schnitzer A., Sandlin K., Miller R., Scarfone K.** NIST Special Publication 800-162, Guide to Attribute Based Access Control (ABAC) Definition and Considerations, 2014, 37 p.

10. **Bogaerts J., Decat M., Lagaisse B., Joosen W.** Entity-Based Access Control: supporting more expressive access control policies, *ACSAC 2015 — Proceedings of the 31st Annual Computer Security Applications Conference*, 2015, pp. 291—300.

11. **Fong P. W.** Relationship-Based Access Control: Protection Model and Policy Language, *Proceedings of the first ACM conference on Data and application security and privacy*, 2011, pp. 191—202.

12. **Carminati B., Ferrari E., Perego A.** Rule-based access control for social networks, *OTM Meaningful Internet Systems 2006*, Springer, 2006, pp. 1734—1744.

13. **Zhang R.** RelBAC-Relation-Based Access Control, PhD dissertation, 2009, 145 p.

14. **Vasenin V. A., Itkes A. A., Shapchenko K. A., Bukhonov V. Yu.** Relyacionnaya model logicheskogo razgranicheniya dostupa na osnove cepochek otnosheniy (Chain-Relation Based Access Control), *Programmnaya Ingeneria*, 2015, no. 9, pp. 11—19 (in Russian).

15. **Sadovnichiy V. A., Afonin S. A., Bakhtin A. V.** et. al. *Intellektualnaya sistema tematicheskogo issledovaniya nauchno-tekhnicheskoy informacii. («ISTINA»)* (Intellectual System for Thematical Analysis of Scientific and Technical Information), Moscow, Lomonosov Moscow State University Publishing House, 2014, 262 p. (in Russian).

16. **Damianou N., Dulay N., Lupu E., Sloman M.** The Ponder Policy Specification Language, *Workshop on Policies for Distributed Systems and Networks,* Bristol, UK, 29—31.01.2001, Springer-Verlag LNCS, 2001, vol. 1995, pp. 18—39.

17. **Hu V. C., Kuhn D. R., Xie T.** Property Verification for Generic Access Control Models, *Embedded and Ubiquitous Computing* 2008 — *EUC'08*, *IEEE/IFIP International Conference on* (Volume 2), 17—20.12.2008, pp. 243—250.

18. **Karpov Yu. G.** *MODEL CHECKING. Verifikaciya parallelnikh i raspredelennikh programmnikh sistem* (Parallel and Distributed Software Systems Verification), Saint Petersburg, 2010, 560 p. (in Russian).

19. **Shapchenko K. A.** *Metodi i programmnie sredstva issledovaniya modeley logicheskogo razgranicheniya dostupa na predmet vipolneniya trebovaniy po bezopasnosti* (Methods and software tools for access control models analysis for the purpose of checking the fulfillment of security requirements): PhD dissertation, Moscow, Lomonosov Moscow State University, 2010 (in Russian).

20. **OASIS.** eXtensible Access Control Markup Language (XACML) Standard, Version 3.0. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf, 2013.

УДК 004.4 DOI: 10.17587/prin.7.559-567

И. Б. Бурдонов, д-р физ.-мат. наук, вед. науч. сотр., e-mail: igor@ispras.ru,
А. С. Косачев, канд. физ.-мат. наук, вед. науч. сотр., e-mail: kos@ispras.ru,
Институт системного программирования РАН, Москва

Исследование графов коллективом двигающихся автоматов

Исследование графов автоматами является корневой задачей во многих приложениях. К таким приложениям относятся верификация и тестирование программных и аппаратных систем, а также исследование сетей, в том числе сети Интернет и GRID на основе формальных моделей. Модель системы или сети сводится к графу, свойства которого нужно исследовать.

Настоящая публикация — вторая в серии трех статей, посвященных исследованию графов автоматами. Если в первой статье автомат был один, то в данной статье по дугам ориентированного графа двигаются несколько автоматов, обмениваясь сообщениями по независимой от графа сети связи. Такая постановка задачи оправдана тогда, когда исследование графа одним автоматом (компьютером) либо требует недопустимо большого времени, либо граф не помещается в памяти одного компьютера, либо и то и другое. Поэтому требуется параллельное и распределенное исследование графа коллективом автоматов.

Изучается также возможность обхода недетерминированного графа.

Ключевые слова: *ориентированный граф, упорядоченный граф, нумерованный граф, неизвестный граф, недетерминированный граф, обход графа, Δ-обход, автомат, робот, полуробот*

Введение

Настоящая статья является второй в серии статей по исследованию графов автоматами. В первой статье [1] рассматривался обход графа одним автоматом, двигающимся по ребрам графа. В данной статье задача обхода графа решается коллективом автоматов, которые могут не только двигаться по ребрам графа, но и обмениваться между собой сообщениями по сети связи, независимой от графа. В заключительной части статьи рассмотрен случай недетерминированного графа. В следующей, третьей, статье будет рассмотрена "инвертированная" модель, в которой автоматы неподвижно "сидят" в вершинах графа, но могут обмениваться между собой сообщениями, которые передаются по ребрам графа.

Сначала повторим кратко основные понятия, введенные в работе [1]. Будем рассматривать ориентированные графы, а их ребра, как это принято, будем называть дугами. Введем обозначения: n — число вершин; m — число дуг; d — длина максимального пути (маршрута без самопересечения) в графе. Автомат на графе можно рассматривать как аналог машины Тьюринга: вместо ячеек ленты машины используются вершины графа, а вместо движения по ленте влево или вправо выполняется движение по дугам графа в направлении их ориентации.

Автомат, который конечен на всем рассматриваемом классе графов, будем называть *роботом*. Если автомат не конечный, но граф не помещается в его память, то такой автомат будем называть *полуроботом*. По сути, это то же самое, что локальный алгоритм на графе. Если граф помещается в память автомата, то это *неограниченный автомат*.

Обход графа. Как правило, исследование графа сводится к *обходу графа*, под которым понимается построение и проход автоматом маршрута, содержащего все дуги (и, тем самым, все вершины) графа. Когда автоматов много и они работают параллельно, можно говорить о покрытии дуг графа не одним маршрутом, а набором маршрутов. Иными словами, при обходе требуется, чтобы по каждой дуге прошел хотя бы один автомат. Автомат начинает работать с выделенной начальной вершины графа, которую для краткости будем называть *корнем*.

Структура на графе. При любом обходе графа возникает структура на графе, при описании которой будем использовать три краски: белую, серую и черную. Непройденные дуги, т. е. дуги, по которым автомат не проходил, окрашены в белый цвет; пройденные дуги серые или черные. Непройденная вершина, т. е. вершина, в которой автомат еще не был, тоже белая. Пройденная вершина черная, если автомат прошел по всем выходящим из вершины дугам (все они серые или черные), иначе вершина серая (есть выходящая белая дуга). *Пройденный граф* — подграф, порожденный пройденными дугами (серыми и черными). Он представляет собой цепочку компонентов сильной связности, причем из каждого компонента, кроме последнего, ведет одна дуга в следующий компонент. Эта дуга называется *разделяющей*, а ее конец — корнем следующего компонента. Корень графа понимается как корень первого компонента. *Прямая дуга* — это дуга, по которой автомат первый раз пришел в какую-то вершину; прямые дуги порождают прямое дерево — остов пройденного графа,

ориентированный от корня. Остальные пройденные дуги будем называть *хордами* прямого дерева. Серое дерево — это поддерево прямого дерева, содержащее корень и все серые вершины, причем все его листья тоже серые. Дуги серого дерева серые, остальные прямые дуги и хорды черные.

Три типа алгоритмов. Основные типы алгоритмов обхода — это DFS (обход в глубину), BFS (обход в ширину) и "жадный" алгоритм.

В DFS-алгоритме серое дерево состоит только из одного пути. Пройдя хорду, автомат по пройденному графу всегда возвращается в начало хорды. Такое действие называется *откат по хорде.* Если конец серого пути становится черным, автомат возвращается в начало последней дуги серого пути и перекрашивает эту дугу в черный цвет. Такое действие называется *откат по дереву.*

В BFS-алгоритме серое дерево может ветвиться. После прохода хорды автомат движется по пройденному графу не до начала хорды, а только до серого дерева, а потом по нему вверх до ближайшей серой вершины. Если листовая вершина серого дерева становится черной, то выполняется откат по дереву в начало той дуги дерева, которая заканчивается в этой листовой вершине, и эта дуга перекрашивается в черный цвет.

Оба отмеченных выше алгоритма для откатов используют лес *обратных деревьев.* Обратное дерево — это остов сильно связного компонента пройденного графа, ориентированный к корню компонента.

"Жадный" алгоритм не использует прямое и обратные деревья. Каждый раз, когда текущая вершина становится черной, по пройденному графу определяется путь из текущей вершины в ближайшую серую вершину, который автомат и проходит.

Упорядоченный граф. Упорядоченным графом называется граф, в котором дуги, выходящие из вершины, перенумерованы, начиная с 1. Для каждой вершины при этом задана своя нумерация выходящих дуг. Для движения по графу автомат, находящийся в вершине, указывает номер выходящей дуги, по которой он "хочет" пройти. Поскольку номер дуги является выходным символом автомата, для того чтобы автомат был конечным, нужно чтобы полустепень исхода вершины была ограничена сверху. Это ограничение обычно обходится с помощью преобразования каждой вершины в цепочку вершин, каждая из которых имеет полустепень исхода S_{out} не больше 2 (рис. 1).

Неизвестный граф, неизбыточные и свободные автоматы. Если граф известен и помещается в память автомата, то можно вычислить его обход минимальной длины, а потом пройти по этому маршруту. Граф неизвестен, если он неизвестен заранее, до прохода по всем его дугам. На каждом шаге автомат может "знать" только часть графа: пройденный граф и, быть может, что-то о белых дугах, выходящих из серых вершин. Автомат *неизбыточный*, если он "знает" полустепень исхода каждой пройденной вершины. Автомат *свободный*, если он этого не "знает", но "узнает" о наличии или отсутствии белой выходящей дуги при попытке пройти по ней. Если дуга с указанным номером присутствует, то автомат идет по ней, а если нет, то получает отказ и остается на месте. Для свободного автомата, допуская вольность речи, будем говорить о цвете вершин и дуг "с точки зрения" автомата: если вершина черная, но автомат еще не "знает" об этом, то для него она серая; соответственно понимается серое дерево.

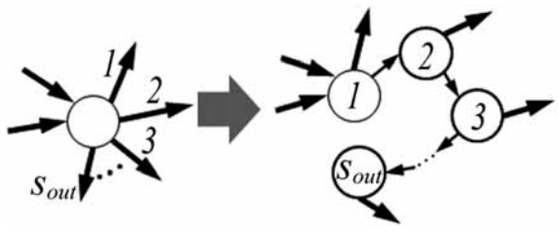

Рис. 1. Преобразование вершины в цепочку вершин

Нумерованный граф. Если все вершины графа пронумерованы и в каждой вершине записан ее номер, то такой граф будем называть нумерованным. Робот, как конечный автомат, не может использовать номер вершины, поскольку этот номер неограничен на любом классе графов с произвольным числом вершин. Однако для полуробота и неограниченного автомата номер вершины бывает полезным. Такой автомат может и сам нумеровать вершины графа в порядке их обнаружения (когда первый раз попадает в вершину). Неограниченный автомат по мере обхода может запоминать в своей памяти весь пройденный граф. Для такого автомата номер вершины — это единственное, что нужно хранить в самой вершине, поскольку все остальное он может хранить в своей памяти, индексируя номером вершины. На нумерованном графе неограниченный автомат может только читать из вершины ее номер и не будет писать в нее.

Обмен сообщениями. При обходе графа коллективом автоматов предполагается, что автоматы могут не только читать и писать в вершины графа, но и обмениваться между собой сообщениями по независимой от графа сети связи (сети передачи данных). Предполагается, что сеть связи гарантирует прохождение сообщения "от точки к точке", т. е. от автомата-отправителя к автомату-получателю по адресу последнего.

Генерация автоматов. Автоматы генерируются в корне графа и оттуда двигаются по дугам графа. Сам обход понимается как покрытие дуг графа маршрутами, которые проходятся всеми автоматами, т. е. по каждой дуге должен пройти хотя бы один автомат.

1. Обход известного графа неограниченными автоматами

Теперь рассмотрим задачу обхода графа коллективом неограниченных автоматов. Автоматы генерируются в корне графа. Автомат, который генерирует другие автоматы, будем называть *генератором*, а остальные — *движками*. Будем предполагать, что число генерируемых автоматов не ограничено. Граф не обязательно сильно связный, предполагается, что покрываются дуги графа достижимости. Поскольку, кроме прохода дуг есть еще обмен сообщениями, будем оценивать время обхода, считая, что пересылка сообщения и проход по дуге занимают не больше одного такта.

Сначала рассмотрим случай известного графа. Автоматы не пишут в вершины и не читают из них.

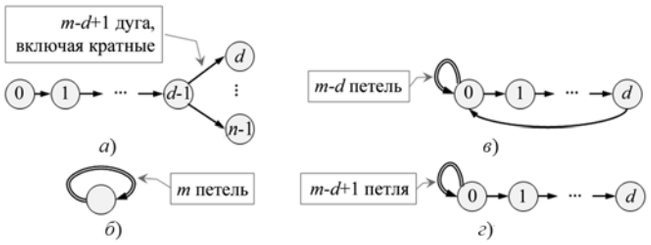

Рис. 2. Примеры графов для коллектива неограниченных автоматов:

n — число вершин; m — число дуг; d — максимальная длина пути в графе

Граф известен генератору и все нужные пометки он может делать в своей памяти. Генератор сообщает каждому движку, куда ему идти, в виде последовательности номеров дуг. Если за один такт генерируется один движок, то время обхода на самом плохом графе равно m (рис. 2, *а*), а на самом хорошем графе $\boldsymbol{O}(\sqrt{m})$ (рис. 2, *б*). Если за один такт генератор может сгенерировать неограниченное, но конечное число движков, то для известного графа генератор может сгенерировать сразу m движков. Время обхода на самом плохом графе равно $d + 1$ (рис. 2, *в*), а на самом хорошем графе равно d (рис. 2, *г*).

2. Обход неизвестного графа неограниченными автоматами

Для неизвестного графа в памяти генератора хранится только пройденный граф. Когда движок попадает в новую вершину, ему надо узнать ее номер и, если автомат неизбыточный, то еще и полустепень исхода вершины. Для простоты будем считать, что граф нумерованный. Генератор на каждом такте сообщает движку, по какой дуге ему сейчас идти. Движок либо идет по указанной дуге и сообщает генератору номер вершины в конце этой дуги, либо (если автомат свободный) получает отказ, если дуги с таким номером нет, и сообщает об этом генератору. После этого движок ждет дальнейших указаний. Генератор направляет движок по непройденной дуге, а если ее нет, то по любой дуге, через которую достижима серая вершина. Если же и такой дуги нет, то генератор останавливает движок.

Если за один такт генерируется один движок (*алгоритм 1*), то порядок времени обхода не меняется по сравнению с известным графом, а именно — $\boldsymbol{O}(m)$ даже для свободного автомата. Действительно, будем говорить, что движок находится на дуге, если это последняя дуга, которую он прошел. Тогда: 1) движок через $\boldsymbol{O}(1)$ тактов уходит с дуги или останавливается, если из конца этой дуги недостижима серая вершина; 2) на каждой дуге одновременно находится не более одного движка. Поэтому, пока есть серые вершины, генератор может генерировать один движок каждые $\boldsymbol{O}(1)$ тактов без нарушения этих двух условий. Тем самым, через $\boldsymbol{O}(m)$ тактов обход закончится. При этом указываем лишь порядок времени обхода, чтобы учесть затраты на передачу сообщений и срабатывание отказов, когда дуги с указанным номером нет.

Если за один такт генератор может сгенерировать неограниченное, но конечное число движков (*алгоритм 2*), то оценка меняется. Если автоматы неизбыточные, то движок, "узнавая" полустепень исхода вершины, сообщает ее генератору. Тем самым, генератор "знает" число непройденных дуг, выходящих из каждой пройденной вершины. На каждом такте генератор генерирует число движков, равное разности между числом непройденных дуг, выходящих из пройденных вершин, и числом уже сгенерированных движков. Рассмотрим путь длины L, выходящий из корня. Если i-я вершина пути достигается на такте t_i, то на этом такте генератор может сгенерировать столько движков, сколько дуг выходит из i-й вершины минус 1 (один движок уже в вершине). Значит, вершина $i + 1$ будет достигнута не более чем через i тактов после этого: $t_{i+1} \leqslant t_i + i$. Поскольку $t_1 = 0$, имеем $t_L \leqslant L(L-1)/2$. Так как $L \leqslant d$, то время обхода $\boldsymbol{O}(d^2)$.

Покажем, что алгоритм не может работать быстрее. Рассмотрим граф, состоящий из пути длины d, вершины которого пронумерованы от корня 1, 2, 3, ..., d, и из каждой вершины i выходят m_i дуг, причем все они, кроме дуги пути $i \to i + 1$, терминальные. Пусть нумерация выходящих дуг такая, что сначала проходятся терминальные дуги, выходящие из вершины, а потом уже дуга пути. Пусть первый движок достигает вершины i за t_i тактов. Тогда число сгенерированных за это время движков $f(t_i)$ не зависит от m_i. Поэтому всегда можно подобрать такой граф, чтобы $f(t_i) < m_i$. Следовательно, на такте t_i "не хватает" движков, чтобы пройти все m_i дуг, выходящих из вершины i. Поэтому движок, который пройдет по дуге $i \to i + 1$, а эта дуга проходится последней среди всех дуг, выходящих из вершины i, будет сгенерирован после такта t_i. Для того чтобы этому движку пройти от корня до вершины $i + 1$, потребуется не менее i тактов. Следовательно, $t_{i+1} \geqslant t_i + i$. Имеем: $t_1 = 0$, $t_2 \geqslant 1$, $t_3 \geqslant 3$, $t_4 \geqslant 5$, ..., $t_d \geqslant d(d-1)/2$. Тем самым, время обхода — $\Omega(d^2)$ при достаточно большом m.

Если совместить алгоритмы 1 и 2, генерируя движки первого и второго рода, то получится алгоритм с оценкой $\boldsymbol{O}(\min\{d^2, m\})$. Необходимость в таком совмещении показывается примером графа, состоящего из пути длины d, в каждой вершине которого, кроме последней, добавлена терминальная дуга. Алгоритмы 1 и 2 обходят этот граф за время порядка $\boldsymbol{O}(m) = \boldsymbol{O}(d)$ и $\boldsymbol{O}(d^2)$ соответственно.

Если автоматы свободные, то время обхода неограниченно растет вместе с ростом числа дуг m даже при фиксированной длине максимального пути d. Действительно, в противном случае для каждого алгоритма A максимальное время обхода на классе графов с фиксированным d конечно: $t(A, d) < \infty$. Обход завершается тогда, когда какой-то движок получает последний отказ, т. е. получен отказ для каждой пройденной вершины. После этого достаточно одного такта, чтобы сообщение об отказе дошло до генератора, который определяет, что это был последний отказ, и сообщает "вовне" о завершении обхода.

Пусть время $t(A, d)$ достигается на графе G с числом дуг m. Тогда для любого $m` > t(A, d)$ можно рассмотреть граф $G`$ с той же длиной максимального

пути d и числом дуг $m`$, который отличается от графа $G$ тем, что в той вершине графа $G$, где получаем последний отказ, добавляются петли, число которых равно $m` - m$. Время обхода графа $G`$, очевидно, больше $t(A, d)$, что противоречит его максимальности. Утверждение доказано. Итак, время обхода свободными автоматами по порядку не меньше d^2 при достаточно большом числе дуг m и неограниченно растет вместе с ростом m.

3. Практический пример и проблема больших графов

Практическим примером обхода ориентированного графа коллективом двигающихся автоматов могут служить результаты, представленные в работе [2], которая посвящена тестированию модели цифровой аппаратуры с помощью сети компьютеров. В графе переходов модели 84 561 состояние и 338244 дуги. Число компьютеров менялось от 1 до 100 (Intel Core2 Quad Q9400, 2,66 GHz; 4 Gb; Linux; Ethernet). В применявшемся DFS-алгоритме не было генератора, движки статически распределялись по компьютерам (один в один) и каждый из них общался только с соседями в соответствии с топологией связей. Такой распределенный обход графа преследовал цель уменьшить время обхода. В табл. 1 показано время тестирования программной реализации модели в зависимости от числа компьютеров и топологии связей.

В этом практическом примере граф, хотя и большой, но все-таки помещался в память одного компьютера, т. е. в память автомата. А что делать, если граф не помещается в память автомата? Когда ранее рассматривался обход графа коллективом свободных неограниченных автоматов, выяснилось, что память вершин автоматам, в общем-то, не нужна, а достаточно того, чтобы граф был нумерованным.

Сохраняя эти предположения, будем считать, что, если граф не помещается в память одного автомата, т. е. этот автомат является полуроботом, то граф все-таки должен помещаться в суммарную память нескольких полуроботов, число которых неограниченно. Это важно при тестировании, где никакой памяти вершин нет, поскольку вершина — это состояние тестируемой системы. Его можно наблюдать, но в него нельзя писать, можно только перейти в другое состояние в результате тестового воздействия.

4. Обход неизвестного графа коллективом полуроботов

Алгоритм обхода опубликован в работе [3], ограничимся изложением его основной идеи. Кроме генератора и движков добавляется новый тип автомата — регулятор вершины. Именно в суммарной памяти регуляторов и хранится описание пройденного графа. Все автоматы свободные. Алгоритм строит прямое дерево пройденного графа, помечая прямые дуги и хорды в их начальных вершинах. Регулятор вершины не взаимодействует с графом и хранит информацию только о ближайшем окружении вершины: номер вершины; адрес *родителя*, т. е. регулятора начала входящей прямой дуги; для каждой выходящей прямой дуги — адрес *потомка*, т. е. регулятора конца дуги. Регулятор управляет перемещением движков через свою вершину. Обозначим ответы на ряд вопросов, которые возникают при рассмотрении этого алгоритма обхода.

Откуда берутся регуляторы? Функции регулятора корня выполняет генератор. Для другой вершины регулятор создает движок, первым попавший в вершину. **Как движок узнает, что он первый?** Для этого выполняется поиск регулятора по номеру вершины. Регуляторы связаны в список по их адресам, и по списку посылается сообщение с номером вершины и адресом движка. Если регулятор не найден, то движок в эту вершину попал первым.

Как регулятор управляет движением движков? Движок "спрашивает" (с помощью обмена сообщениями) у регулятора текущей вершины: **"*Куда идти*?"** Регулятор отвечает: "***Иди по дуге"***, и указывает номер этой дуги. Дуги перебираются в порядке возрастания номеров: 1, 2, 3 и т. д. Если такой дуги не оказалось, движок сообщает об этом регулятору, который уже знает число выходящих дуг. Далее регулятор перебирает номера по циклу. Этого достаточно, чтобы обойти граф.

Как узнать о конце обхода? Как и в BFS-алгоритме для одного автомата здесь имеется прямое дерево и его серое поддерево. Отличие состоит лишь в том, что один автомат сам должен после хорды пытаться строить новую ветвь дерева, а когда автоматов много, то эти ветви создаются ими параллельно. Вместо отката используется сообщение ***конец***, посылаемое по дуге в обратном направлении, т. е. из автомата

Таблица 1

Результаты тестирования программной реализации модели цифровой аппаратуры

Число компьютеров k	Топология связей	Время прогона теста $T(k)$, мин	Ускорение $T(1) / T(k)$	Коэффициент эффективности $T(1) / kT(k)$
1	—	803,3	1	1
81	Кольцо	12,2	66	0,81
81	Тор 9 × 9	11,4	70	0,87
100	Кольцо	10,2	79	0,79
100	Тор 10 × 10	9,5	85	0,85

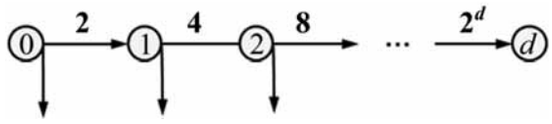

Рис. 3. Пример экспоненциального времени обхода

в конце дуги регулятору начала дуги. При откате по хорде отправитель — это движок, прошедший по хорде, а при откате по прямой дуге отправитель — это регулятор конца дуги. Регулятор-получатель отмечает дугу как *законченную.* Законченная дуга, по сути, то же, что черная дуга. **Как движок узнает, что он прошел по хорде?** Это происходит, когда движок не первым пришел в вершину, т. е. при опросе регуляторов регулятор найден. Сам регулятор посылает ***конец*** родителю, когда все выходящие дуги стали законченными. Если это регулятор корня, то конец обхода.

При такой работе, если не принять никаких дополнительных мер, то время обхода экспоненциально. Это видно из примера на рис. 3. Дуга вниз имеет номер 1, а дуга направо — номер 2. Над дугой написан номер движка в порядке генерации, который первым проходит эту дугу. Из этого примера следует, что "вредно" ходить по законченным дугам.

Как уменьшить время обхода? Для этого регулятору не надо посылать движки по законченным дугам, такие дуги блокируются до конца обхода.

Как уменьшить число одновременно существующих движков? Для этого нужно уничтожать движок, если он: прошел по хорде, или прошел по прямой дуге, а дальше ему некуда идти, потому что все выходящие дуги оказались законченными. Тогда каждый движок проходит путь длиной $\boldsymbol{O}(n)$, после чего выполняет опрос за время $\boldsymbol{O}(n)$ и становится регулятором или уничтожается. Значит, движок живет $\boldsymbol{O}(n)$ тактов. Если за такт генерируется не более одного движка, число одновременно существующих движков $\boldsymbol{O}(n)$.

Как уменьшить число генерируемых движков? Если не тормозить генерацию движков, то их будет сгенерировано столько, сколько длится обход. Для торможения реализуется старт-стопный механизм перемещения движков по прямой дуге. Регулятор направляет движок по прямой дуге только после получения от регулятора конца дуги сообщения ***запроса*** движка. Дуга временно блокируется до получения следующего ***запроса***, который будет отправлен, когда движок пойдет дальше. Если все выходящие дуги заблокированы, то приходящий движок либо уничтожается, если все дуги закончены, либо ожидает разблокировки. Запрос не посылается, поэтому в вершине оказывается не более одного ждущего движка. Если все выходящие из корня дуги блокированные, генератор приостанавливает генерацию движков. Общее число движков $\boldsymbol{O}(m)$.

Какова оценка времени обхода? Для оценки времени обхода будем считать, что проход по дуге и пересылка сообщения выполняются за один такт. Также за один такт генерируется не более одного движка. Адрес автомата — это его номер. Время обхода равно $\boldsymbol{O}(m + nd)$. Второе слагаемое возникает вследствие опроса регуляторов: время одного опроса не больше числа регуляторов, которое не больше числа вершин n, а на прямом пути от корня до листа длиной не более d опросы регуляторов происходят строго последовательно в порядке от корня к листу.

В табл. 2 приведены размеры сообщения и памяти полуробота, а также, для сравнения, те же размеры для неограниченных автоматов. В случае преобразования вершины в цепочку вершин в правых столбцах вместо n следует читать m. Из табл. 2 видно, что размер сообщения остается тем же, в нем два основных параметра: адрес автомата и номер дуги. Если полустепень исхода вершины не ограничена, то выигрыша по памяти автомата не получается. Однако этого и следовало ожидать, поскольку в этом случае локализация информации о графе не может дать выигрыша: полустепень исхода может достигать числа дуг в графе. Если адрес автомата не используется повторно, то получается даже проигрыш, поскольку число автоматов тоже может достигать числа дуг в графе. Для ограниченной полустепени исхода число дуг столько же по порядку, сколько вершин. Получается выигрыш в n раз. Если граф получен с помощью преобразования вершины в цепочку вершин (см. рис. 1), когда получается полустепень исхода 2, число вершин преобразованного графа по порядку равно числу дуг исходного графа. Как следствие, получается выигрыш в m раз. Здесь имеем пример классического правила — "если хотите сэкономить память, придется пожертвовать временем", а именно, $\boldsymbol{O}(m + nd)$ вместо $\boldsymbol{O}(m)$.

Таблица 2

Размеры сообщений и памяти автомата

Полустепень исхода ограничена?		Нет		Да	
Адрес автомата используется повторно?		Нет	Да	Нет	Да
Размер сообщения	Неограниченный автомат	$\boldsymbol{O}(\log m)$		$\boldsymbol{O}(\log n)$	
	Полуробот	$\boldsymbol{O}(\log m)$		$\boldsymbol{O}(\log n)$	
Размер (памяти) автомата	Неограниченный автомат	$\boldsymbol{O}(m\log n)$		$\boldsymbol{O}(n\log n)$	
	Полуробот	$\boldsymbol{O}(m\log m)$	$\boldsymbol{O}(m\log n)$	$\boldsymbol{O}(\log n)$	

5. Роботы

Минимальная длина обхода (маршрута, проходящего по всем дугам графа) ориентированного графа с n вершинами и m дугами равна $\boldsymbol{O}(nm)$, или, в терминах длины максимального пути графа d как максимальной длины пути (маршрута без самопересечения) в графе, $\boldsymbol{O}(dm)$. В работе [1] рассматривались алгоритмы обхода ориентированного графа одним роботом. Наилучшая известная оценка длины обхода $\boldsymbol{O}(nm + n^2\log\log n)$, где n — число вершин, m — число дуг, или, в терминах длины максимального пути d, — $\boldsymbol{O}(dm + dn\log\log n)$.

Второе слагаемое появляется вследствие проблемы, которая возникает при реализации отката, а именно — возврата из конца дуги в ее начало (откат по хорде и откат по дереву), для чего автомат должен пройти некоторый маршрут. Этот маршрут существует в пройденном графе и состоит из пути по обратному дереву до серого дерева и пути по серому дереву до начала дуги. Вместе с этой дугой маршрут образует цикл, по которому автомат и может двигаться. Проблема в том, что робот не "знает", в какую вершину ему надо вернуться, поскольку начало дуги специально не помечено. Поэтому он вынужден двигаться по циклу несколько раз (подробнее см. в работе [1]).

Проблема отката легко решается, если использовать два робота. Длина обхода уменьшается до минимальной по порядку $\boldsymbol{O}(nm)$ или $\boldsymbol{O}(dm)$. Идея алгоритма состоит в том, что роботы идут по графу синхронно, но с расстоянием в одну дугу. Поэтому, когда первый робот "узнает", что нужно делать откат, т. е. возвращаться на одну дугу назад, в начале этой дуги как раз стоит второй робот. Достаточно сообщить ему, чтобы он пометил вершину, в которой находится, и вместе с первым шел по циклу до помеченной вершины. При откате по хорде первый робот находится в конце хорды, а второй — в начале хорды. При откате по дереву первый робот находится в листовой вершине серого дерева, а второй — в предыдущей вершине дерева.

Темой дальнейших исследований могли бы стать алгоритмы обхода графа большим числом роботов. Для коллектива неограниченного числа полуроботов или неограниченных автоматов, как показано выше, оценки времени обхода существенно лучше, чем в случае одного автомата [1]. Можно было бы ожидать, что аналогичный результат может быть получен и для роботов, число которых больше двух. Правда, здесь возникает трудность обмена сообщениями между автоматами, поскольку требуется указывать адрес автомата-получателя, который, тем самым, становится частью выходного символа автомата-отправителя, а размер выходного символа должен быть ограничен для конечного автомата, т. е. робота. Как следствие, либо число роботов должно быть ограничено сверху, что может не дать ожидаемого эффекта, либо нужно найти какой-то другой способ адресации автоматов, например, опираясь на топологию сети связи автоматов (тор, кольцо и т. д. как в разд. 3), использовать локальный адрес автоматов.

6. Недетерминированный граф

В настоящем разделе рассмотрим стратегию обхода недетерминированного графа коллективом автоматов.

6.1. Справедливый недетерминизм и предположение о Δ-дугах

Упорядоченный граф будем называть *недетерминированным*, если одним номером дуги может быть помечена не одна, а несколько выходящих дуг. Будем называть Δ-*дугой* множество дуг с одним номером и одной начальной вершиной. Когда автомат в вершине указывает номер выходящей дуги, недетерминированным образом выбирается одна из дуг соответствующей Δ-дуги. Будем считать, что этот недетерминированный выбор задается недетерминированной функцией выбора, которая для каждой Δ-дуги возвращает дугу из этой Δ-дуги.

Понятно, что если на функцию выбора не налагается никаких ограничений, то, вообще говоря, обход графа невозможен, так как может оказаться, что при любом вызове функции выбора одна из дуг Δ-дуги никогда не выбирается. Будем именовать недетерминизм *справедливым* [5], если для любой Δ-дуги в любой бесконечной последовательности значений функции выбора каждая дуга из этой Δ-дуги встречается бесконечное число раз. Заметим, что дуга должна встречаться бесконечное число раз, а не хотя бы один раз, поскольку в противном случае возможность обхода графа не гарантируется. Примером может служить граф, представленный на рис. 3 при $d > 1$, если считать, что две дуги, выходящие из одной вершины, образуют одну Δ-дугу. При этом функция выбора для каждой Δ-дуги только один раз возвращает дугу, ведущую направо, а автоматы генерируются в корне — вершине 0.

Если граф — это граф переходов модели тестируемой системы, то гипотеза о справедливом недетерминизме эквивалентна гипотезе о глобальном тестировании: при бесконечном числе прогонов теста (включая рестарт системы) бесконечное число раз будут получены все возможные варианты поведения тестируемой системы.

Гипотеза о справедливом недетерминизме гарантирует возможность обхода любого нумерованного графа неограниченным числом свободных полуроботов. Для этого достаточно, чтобы генератор в корне постоянно генерировал движки, а в каждой вершине (идентифицируемой по ее номеру) движки перебирали номер Δ-дуги по циклу (число Δ-дуг в вершине свободный автомат определяет, получая отказ).

Возникает естественный вопрос: "Как узнать о конце обхода?" Для ответа на него достаточно следующего *предположения о* Δ-*дугах*: в Δ-дуге нет кратных дуг; для каждой Δ-дуги можно узнать число дуг в ней. Первое из этих предположений необходимо для того, чтобы можно было различить дуги одной Δ-дуги по их конечным вершинам. Второе нужно для того, чтобы узнать, все ли дуги Δ-дуги пройдены. Будем считать, что когда движок пытается пройти по

дуге с указанным номером, он либо получает отказ, либо проходит по дуге и получает число дуг в соответствующей Δ-дуге.

6.2. Обход недетерминированного графа коллективом полуроботов

Алгоритм обхода справедливо недетерминированного ориентированного графа коллективом двигающихся полуроботов опубликован в работе [5], ограничимся изложением его основной идеи. Попробуем приспособить к недетерминированному графу алгоритм обхода детерминированного графа с помощью движков и регуляторов (см. разд. 4).

Первое решение достаточно простое: пусть движок, проходя по дуге, сообщит регулятору начала дуги в какую вершину он попал. Регулятор проверяет, туда ли он направлял движок: если не туда, движок уничтожается (или останавливается), а если туда, ему разрешают продолжить работу. Однако это решение не представляется удачным. Например, Δ-дуга состоит из двух незаблокированных прямых дуг: *a* и *b*. Регулятор посылает первый движок по дуге *a*, а второй — по дуге *b*. Однако вследствие недетерминизма все получается наоборот: первый движок проходит дугу *b*, а второй — дугу *a*. Регулятор уничтожает оба движка. А на следующем шаге все повторяется. Недетерминизм справедливый: "честно" перебираются дуги в Δ-дуге: *a*, *b*, *a*, *b* и т. д. Но регулятор при этом уничтожает все движки.

Второе решение подсказывает этот же пример. На самом деле все равно, какую дугу пройдет движок *a* или *b*, так как обе они незаблокированные. В этом случае регулятор должен вести себя просто "умнее": если движок прошел одну из незаблокированных дуг, то все в порядке. Только при проходе заблокированной дуги движок уничтожается. Это решение правильное, однако неэффективное. Дело в том, что движку приходится опрашивать регуляторы после прохода каждой дуги.

Третье решение основано на приеме, аналогичном тому, что использовался в детерминированном случае. Там движок не опрашивал регуляторы, если проходил по прямой дуге, так как регулятор ее начала уже знал регулятор конца дуги. Однако в недетерминированном случае движок может пройти по другой дуге. Вместе с тем движок всегда знает номер вершины, в которую попал. Если регулятор для каждой прямой выходящей дуги хранит не только регулятор ее конца, но и номер вершины, он может "подсказать" движку по номеру вершины адрес ее регулятора. Это осуществляется обменом сообщениями между движком и регулятором. В этом случае поиск регулятора выполняется, как и в детерминированном случае, только после прохода еще непройденной дуги.

Как оценить такой алгоритм? Размер сообщений и памяти автомата увеличиваются по сравнению с детерминированным случаем, но оба размера имеют тот же порядок. Время обхода при рассматриваемой стратегии, конечно, зависит от недетерминированной функции выбора. Если она на самом деле детерминирована, то получаем ту же оценку $\boldsymbol{O}(m + nd)$.

Пример другой функции справедливого выбора — это *t*-недетерминизм [4, 5], который для фиксированного числа *t* гарантирует, что за *t* попыток пройти по Δ-дуге будут пройдены все ее дуги. Заметим, что для *t*-недетерминированного графа нет необходимости в предположении о Δ-дугах, представленном в подразд. 6.1. Действительно, если автоматы *t* раз выполнят проход по Δ-дуге, каждый раз запоминая (в регуляторе начала дуги) дугу, по которой они проходят, то, во-первых, гарантированно будут пройдены все дуги Δ-дуги, включая кратные дуги, а во-вторых, регулятор "узнает" число дуг в Δ-дуге с точностью до кратности, т. е. "узнает" число различных концов дуг этой Δ-дуги. Алгоритм обхода неизвестного ориентированного *t*-недетерминированного графа одним неограниченным автоматом предложен в работе [4]. Оценка времени обхода экспоненциальная. Для коллектива полуроботов алгоритм предложен в работе [5], но тоже, по крайней мере, с экспоненциальной оценкой. Примером может служить граф, представленный на рис. 3, если все дуги, выходящие из одной вершины, принадлежат одной Δ-дуге, а "зловредная" функция выбора только каждый второй раз направляет движок по дуге направо.

Для практического применения желательна полиномиальная оценка, для чего нужно использовать какие-то дополнительные ограничения. Например, в *t*-недетерминированном графе есть *k* детерминированных дуг (число дуг в Δ-дуге равно 1), и по ним можно попасть из корня в любую вершину. Будем считать, что после прохода недетерминированной дуги движок уничтожается, как после прохода хорды в детерминированном случае. Алгоритм состоит из двух этапов. На первом этапе будем проходить недетерминированные Δ-дуги по одному разу, что эквивалентно обходу детерминированного графа, который получается из исходного заменой каждой недетерминированной Δ-дуги одной детерминированной терминальной дугой. Строится прямое дерево из детерминированных дуг. Время обхода $\boldsymbol{O}(p + nd)$, где p — число Δ-дуг. На втором этапе не будем проходить детерминированные хорды, а каждую недетерминированную Δ-дугу будем проходить "оставшиеся" t-1 раз, что эквивалентно обходу детерминированного графа, который получается из исходного удалением детерминированных хорд (остается n-1 детерминированная дуга) и заменой каждой недетерминированной Δ-дуги t-1 детерминированными терминальными дугами. Время обхода $\boldsymbol{O}((p\text{-}k)(t\text{-}1) + n\text{-}1 + nd)$. Суммарное время $\boldsymbol{O}(pt + nd)$.

6.3. Стратегия Δ-обхода

Другой подход к обходу недетерминированного графа — это модификация самой цели обхода: вместо того чтобы проходить по всем дугам, требуется пройти по всем Δ-дугам, т. е. хотя бы по одной дуге в каждой Δ-дуге [6]. Это имеет практический смысл при тестировании программной реализации модели системы: в каждом состоянии системы пробуем подать на нее каждое тестовое воздействие хотя бы по одному разу.

Будем называть Δ-*маршрутом* множество маршрутов с одной начальной вершиной, которое "ветвится"

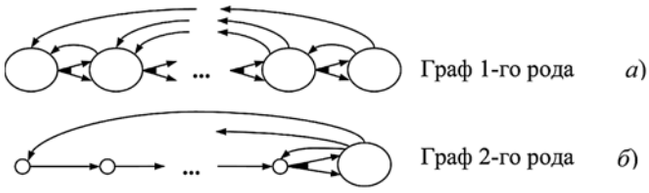

Рис. 4. Графы 1-го и 2-го рода для Δ-обхода

по всем дугам каждой проходимой Δ-дуги. Будем называть Δ-*обходом* такой Δ-маршрут, который проходит по всем Δ-дугам. Если при движении по графу автомат называет номер следующей Δ-дуги в соответствии с Δ-обходом, то он гарантированно пройдет какой-то маршрут из Δ-обхода, т. е. пройдет по всем Δ-дугам. Граф называется *сильно* Δ-*связным*, если для каждой пары вершин a и b существует Δ-маршрут, все маршруты которого начинаются в a и заканчиваются в b.

Для существования Δ-обхода, начиная с любой начальной вершины, необходима и достаточна сильная Δ-связность. При фиксированной начальной вершине сильная Δ-связность только достаточна, а необходимое и достаточное условие использует соответствующим образом определенные недетерминированные графы 1-го и 2-го рода [6]. Граф 1-го рода (рис. 4, *а*) — это цепочка компонентов сильной Δ-связности, где для каждого i-го компонента, кроме последнего, все дуги, выходящие из него и ведущие *вперед* (в компоненты с большим номером), образуют одну связующую Δ-дугу, ведущую в следующий $(i + 1)$-й компонент, а корень графа лежит в 1-м компоненте. Заметим, что некоторые дуги (но не Δ-дуги!) могут вести *назад*, в компоненты с меньшими номерами. Δ-обход существует тогда и только тогда, когда граф 1-го рода. Однако если граф неизвестен, то автомат может гарантированно выполнить Δ-обход только тогда, когда каждый компонент, кроме последнего, состоит из одной вершины, из которой выходит только одна дуга, ведущая в следующий компонент. Такой граф называется графом 2-го рода (рис. 4, *б*). Для детерминированного графа эти определения совпадают с соответствующими определениями в подразд. 3.1 в работе [1].

В работе [6] представлен алгоритм Δ-обхода неизвестного сильно Δ-связного нумерованного графа. Длина обхода $\boldsymbol{O}(np)$, где n — число вершин; p — число Δ-дуг. Память вершины и автомата $\boldsymbol{O}(s\log n)$, где s — ограничение сверху на число Δ-дуг, выходящих из одной вершины. Однако этот алгоритм имеет одну особенность: автомат, пройдя по дуге и прочитав информацию из конечной вершины дуги, может выполнить запись в начальную вершину дуги. Эту работу может моделировать неограниченный автомат, если для каждой вершины (по ее номеру) он хранит информацию, связанную с этой вершиной, в своей памяти.

Другой вариант: два полуробота, обменивающиеся между собой сообщениями. Аналогично двум роботам, описанным в разд. 5, автоматы двигаются синхронно с отставанием на одну дугу. Тогда первый автомат, пройдя дугу, посылает сообщение, содержащее информацию из конца дуги, второму автомату, который в это время находится в начале дуги и может сделать запись в нее.

Можно также использовать идею о регуляторах и движках из разд. 4. Существует один движок, который создает регулятор вершины, когда впервые приходит в эту вершину. Если движок, пройдя дугу, попадает в вершину, где уже есть регулятор, он "узнает" у этого регулятора хранящуюся в нем информацию (она относится к концу дуги), и посылает сообщение, содержащее эту информацию, регулятору начала дуги, который делает запись в свою память. Вопрос лишь в том, что движок должен делать опросы регуляторов. Как описано в разд. 4, опрос можно не делать в том случае, когда движок проходит уже пройденную ранее дугу, если при первом проходе дуги в регуляторе начала дуги запоминать пару (номер конца дуги, адрес регулятора конца дуги). Однако даже в этом случае время Δ-обхода увеличится до $\boldsymbol{O}(nm)$, поскольку все равно нужно опрашивать регуляторы после первого прохода дуги (число дуг m), а каждый опрос требует времени $\boldsymbol{O}(n)$. Темой дальнейших исследований могла бы стать адаптация алгоритма из работы [6] для неограниченного числа движков, генерируемых в корне графа, как описано в разд. 4.

Заключение

Настоящая статья вместе с работой [1] завершают рассмотрение модели исследования графа автоматами, двигающимися по графу. Оценки, представленные в данной статье для коллектива автоматов, позволяют сделать вывод о том, что параллельный обхода графа несколькими автоматами существенно уменьшает время работы. Нерешенными остаются задачи обхода графа коллективом большого числа роботов и Δ-обхода недетерминированного графа коллективом большого числа полуроботов. Представляет также интерес поиск таких функций справедливого выбора и/или ограничений на недетерминированные графы, которые позволяют получать полиномиальные оценки времени обхода. В данной работе приведен только один такой пример: недетерминированный граф с несколькими детерминированными дугами, по которым достижимы все вершины графа.

В последней, третьей работе серии будет рассмотрена "инвертированная" модель, в которой автоматы неподвижно "сидят" в вершинах графа, однако обмениваются между собой сообщениями не по независимой от графа сети связи, а по дугам графа.

Список литературы

1. **Бурдонов И. Б., Косачев А. С.** Исследование графа автоматом // Программная инженерия. 2016. Т. 7, № 11. С. 498—508.

2. **Бурдонов И. Б., Грошев С. Г., Демаков А. В., Камкин А. С., Косачев А. С., Сортов А. А.** Параллельное тестирование больших автоматных моделей // Вестник ННГУ. 2011. № 3. С. 187—193.

3. **Бурдонов И. Б., Косачев А. С.** Обход неизвестного графа коллективом автоматов // Труды Института системного программирования РАН. 2014. Т. 26, № 2. С. 43—86.

4. **Бурдонов И. Б., Косачев А. С.** Полное тестирование с открытым состоянием ограниченно недетерминированных систем // Программирование. 2009. № 6. С. 3—18.

5. **Бурдонов И. Б., Косачев А. С.** Обход неизвестного графа коллективом автоматов. Недетерминированный случай // Труды Института системного программирования РАН. 2015. Т. 27, № 1. С. 51—68.

6. **Бурдонов И. Б., Косачев А. С., Кулямин В. В.** Неизбыточные алгоритмы обхода ориентированных графов. Недетерминированный случай // Программирование. 2004. № 1. С. 2—17.

Graph Learning by Group of Moving Automata

I. B. Bourdonov, igor@ispras.ru, **A. S. Kossatchev,** kos@ispras.ru,
Institute for System Programming RAS, Moscow, 109004, Russian Federation

Corresponding author:

Bourdonov Igor B., Leading Researcher, Institute for System Programming RAS, 109004, Moscow, Russian Federation
E-mail: igor@ispras.ru

Received on August 01, 2016
Accepted on September 14, 2016

Graph learning or exploration tasks can be found in many applications. Among the most important ones we can name software or hardware verification and testing, network exploration, including exploration of big parts of Internet and GRIDs. In many cases system or network model is represented as a graph, which properties should be investigated.

This paper is the second one in a series of works devoted to graph exploration by automata. In the first paper we consider graph exploration by a single automaton, in this one we consider directed graph exploration by several automata, which can move along its arcs and communicate through graph-independent network. All arcs outgoing from a vertex of the graph are numbered, such a graph is called an ordered one. To move through some arc from some vertex, an automaton should use this arc's number.

In the modern technical environment the size of actively used systems and networks is growing, so does the size of graphs modeling those systems and networks. When graph exploration by a single automaton (a single machine) requires too large time or a graph cannot be stored in the memory of a single machine, such an exploration and further graph investigation become problematic. So, the task of parallel distributed graph exploration becomes actual. We formalize this task as a task of graph exploration by a set of automata (several machines, which total memory is sufficient to store the graph in it).

Possible solutions of this task can differ depending on the memory size of automata used. As in the first paper, we consider robots, semi-robots and unbounded automata. A robot is a finite automaton with bounded memory. A semi-robot is an automaton, which memory size is bounded by some function of the graph size. An unbounded automaton can store the entire structure of the graph in its internal memory.

In this paper we also present the results on non-deterministic graph exploration by a set of semi-robots. Non-deterministic graph can have several arcs with the same number outgoing from one vertex. Under some restrictions on such a graph, it can be explored by a set of automata. For unbounded nondeterministic case we consider so called Δ-traversal, which may not use every arc, but use every valid arc number in each vertex at least once. To have a Δ-traversal a directed graph should be bound in some specific sense.

In the next paper we consider immobile automata fixed in graph vertices and communicating with messages sent along graph arcs.

Keywords: *directed graph, ordered graph, numerated graph, unknown graph, nondeterministic graph, graph learning, graph exploration, graph traversal, automaton, robot, semirobot, automata group*

For citation:

Bourdonov I. B., Kossatchev A. S. Graph Learning by Group of Moving Automata, *Programmnaya Ingeneria,* 2016, vol. 7, no. 12, pp. 559—567.

DOI: 10.17587/prin.7.559-567

References

1. **Bourdonov I. B.**, **Kossatchev A. S.** Issledovanie grafa avtomatom (Graph learning by automaton), *Programmnaya Ingeneria*, 2016, vol. 7, no. 11, pp. 498—508 (in Russian).

2. **Burdonov I. B., Groshev S. G., Demakov A. V., Kamkin A. S., Kosachev A. S., Sortov A. A.** Parallel'noe testirovanie bol'shih avtomatnyh modelej (Parallel testing of large automata models), *Vestnik NNGU,* 2011, no. 3, pp. 187—193 (in Russian).

3. **Bourdonov I. B.**, **Kossatchev A. S.** Obhod neizvestnogo grafa kollektivom avtomatov (Graph learning by a set of automata), *Trudy ISP RAN,* 2014, vol. 26, no. 2, pp. 43—86 (in Russian).

4. **Bourdonov I. B.**, **Kossatchev A. S.** Complete Open-State Testing of Limitedly Nondeterministic Systems, *Programming and Computer Software*, 2009, vol. 35, no. 6, pp. 301—313.

5. **Bourdonov I. B.**, **Kossatchev A. S.** Obhod neizvestnogo grafa kollektivom avtomatov. Nedeterminirovannyj sluchaj (Graph learning by a set of automata. The nondeterministic case), *Trudy ISP RAN,* 2015, vol. 27, no. 1, pp. 51—68 (in Russian).

6. **Bourdonov I. B.**, **Kossatchev A. S., Kuliamin V. V.** Irredundant Algorithms for Traversing Directed Graphs: The Nondeterministic Case, *Programming and Computer Software*, 2004, vol. 30, no. 1, pp. 2—17.

УДК 004.5 DOI: 10.17587/prin.7.568-574

Д. И. Читалов[1], мл. научный сотр., e-mail: cdi9@yandex.ru,
Е. С. Меркулов[1, 2] мл. науч. сотр., вед. инженер,
С. Т. Калашников[1, 2], канд. техн. наук, нач. отдела,
[1] Отдел фундаментальных проблем аэрокосмических технологий, Челябинский научный центр Уральского отделения РАН, г. Миасс
[2] АО "ГРЦ Макеева", г. Миасс

Разработка графического интерфейса пользователя для программного комплекса OpenFOAM

Описана архитектура приложения с графическим интерфейсом для решателя rhoCentralFoam, входящего в состав программного комплекса OpenFOAM. Представлена диаграмма компонентов приложения. Определены инструментальные средства, используемые при разработке. Приведено краткое описание разработанной программы.

Ключевые слова: *графический интерфейс пользователя, OpenFOAM, язык программирования Python, открытое программное обеспечение, rhoCentralFoam*

Введение

В настоящее время в области математического моделирования процессов механики сплошных сред и, в частности, таких ее разделов, как механика жидкости и газа, механика деформируемого твердого тела, активно используют пакеты прикладных программ численного моделирования. Как правило, это коммерческие продукты с закрытым исходным кодом (ANSYS, ESI GROUP, Siemens, ТЕСИС и др.). Альтернативой является программное обеспечение с открытым исходным кодом, в частности, пакет OpenFOAM. Благодаря применению универсального стандарта разработки решателей и модулей комплекс OpenFOAM является одним из наиболее эффективных средств численного моделирования механики жидкости и газа. По качеству получаемых с его помощью результатов он не уступает коммерческим аналогам [1—3].

К недостаткам программного комплекса OpenFOAM можно отнести отсутствие графического интерфейса пользователя. Для постановки задач и их решения используют текстовые файлы и терминал, что значительно снижает производительность труда при работе с данным средством. Учитывая растущий интерес мирового сообщества исследователей и инженеров к OpenFOAM, ряд зарубежных фирм начали разрабатывать графический интерфейс пользователя к нему [4]. В качестве примера можно привести программную оболочку HELIX OS, имеющую версию с открытым исходным кодом [5]. Для отечественного пользователя использование зарубежных пакетов программ (в том числе и с открытым исходным кодом) сопряжено с определенными трудностями. Прежде всего, это необходимость оплаты технической поддержки и отсутствие сопроводительной документации на русском языке.

1. Постановка задачи

Программный комплекс OpenFOAM реализован с помощью высокоуровневого языка программирования C++ и состоит из библиотеки классов для проведения операций численного моделирования и библиотеки программ-решателей, которые используют эти классы при решении конкретных задач математического моделирования.

Решение задач в среде OpenFOAM проходит в три перечисленных далее этапа.

- Предобработка (*preprocessing*) — генерация или импорт из сторонних средств расчетной сетки, задание математической модели процесса, указание начальных и граничных условий;
- Обработка (*processing*) — решение задачи с помощью выбранного решателя;
- Постобработка (*postprocessing*) — анализ полученного решения с помощью различных средств визуализации — построение картин распределения переменных по областям, построение графиков и т. д. По умолчанию такой анализ проводится с помощью среды ParaView.

Постановка и решение задачи в комплексе OpenFOAM относительно сложны и требуют знания широкого набора консольных утилит и команд, а также учета специфики файловой структуры рабочей директории. Для решения задачи пользователь

должен создать проект задачи, который состоит из ряда текстовых файлов, расположенных в строго определенных директориях в папке проекта. При этом число текстовых файлов зависит от выбранного решателя. Каждый из данных файлов должен быть заполнен в соответствии с синтаксисом OpenFOAM.

Учитывая вышеперечисленное, авторами данной статьи было решено создать собственное приложение — rCF_GUI, которое с помощью графического интерфейса пользователя будет обеспечивать автоматизацию процессов на этапах предобработки и постановки задач на решение в среде OpenFOAM для конкретного решателя — rhoCentralFoam.

Решатель rhoCentralFoam — один из наиболее востребованных решателей, входящих в состав OpenFOAM [6]. Он хорошо зарекомендовал себя при решении задач гиперзвуковой аэродинамики и по точности получаемых результатов превосходит методы, на которых построены такие пакеты, как Fluent [7].

Программное приложение, обеспечивающее графический пользовательский интерфейс, должно иметь следующие возможности:

- выбор и загрузка для редактирования уже имеющегося проекта задачи или создание нового проекта задачи;
- графическое отображение структуры проекта задачи — начальные и граничные условия, физическая модель, настройки решателя и др.;
- запуск импорта файла расчетной сетки;
- отображение каждого раздела в проекте задачи (граничные условия, начальные условия и др.) в форме, в которой пользователь имеет возможность откорректировать доступные переменные;
- запуск и останов процесса решения задачи;
- графическое отображение процесса решения;
- запуск некоторых необходимых для постобработки утилит;
- запуск утилиты ParaView для проведения постобработки решенной задачи.

Для создания структуры директорий и файлов проекта предполагается использовать шаблоны файлов с предопределенными стандартными условиями.

2. Средства разработки

Интерфейсы многих современных программных приложений создают с помощью высокоуровневого языка программирования Python, к основным достоинствам которого можно отнести следующие.

- Наличие большого числа поддерживаемых библиотек.
- Низкие временные затраты на разработку. Благодаря удобочитаемому синтаксису и простоте освоения повышается производительность труда разработчика и упрощается дальнейшее обслуживание кода [8, 9].
- Кроссплатформенность.

Графические библиотеки, реализованные на языке Python, в целом имеют схожие функциональные возможности. Наибольшей популярностью пользуется библиотека PyQT, которая позволяет создавать полноценные интерфейсы с привлекательным дизайном [10, 11].

Графическая оболочка rCF_GUI реализована на языке Python 3.4 с применением средств библиотеки графических компонентов PyQt4 и создана в среде разработки IDLE. Для реализации функции графического отображения хода решения задачи используются библиотеки numpy и pylab. Библиотека numpy отвечает за обработку массивов данных, поступающих от решателя rhoCentralFoam, а библиотека pylab — за отображение графиков в реальном времени в главном окне программы.

Программные требования для работы с графической оболочкой rCF_GUI представлены далее.

Вид ОС	Linux
Интерпретатор	Python 3.4
Библиотеки	PyQt4, numpy, pylab
Дополнительное ПО	OpenFOAM

3. Архитектура программы и алгоритм ее работы

На рис. 1 представлена диаграмма файловой структуры графической оболочки. В директории программы находится главный исполняемый файл rCF_GUI.py и четыре поддиректории, в которых хранятся служебные файлы и компоненты программы.

Основные компоненты, отвечающие за формирование пользовательских форм и диалогов, расположены в поддиректории /forms/. Модуль file_system.py отвечает за формирование главного графического окна и организацию потоков выполнения в программе. Остальные модули с постфиксом _form отвечают за формирование форм и диалогов. В поддиректории /add_classes/ расположен модуль file_form_class.py, отвечающий за связь модуля file_system.py с другими модулями в директории /forms/.

В поддиректории /matches/ расположены директории с шаблонами файлов проекта задачи OpenFOAM. В поддиректории /styles/ расположены файлы визуальной стилизации приложения.

На рис. 2 представлена блок-схема алгоритма работы с программой. Алгоритм подразумевает взаимодействие пользователя через графический интерфейс с двумя внешними процессами. Первым из них является процесс, в котором происходит выполнение решения задачи в OpenFOAM. Второй процесс обеспечивает визуализацию с использованием ParaView. Эти процессы запускаются с помощью графического интерфейса пользователя.

При запуске программы открывается стартовое окно, которое предусматривает возможность выполнения двух процедур, а именно — создание нового проекта задачи либо открытие существующего. В зависимости от этого определяются параметры запуска главного окна. Внешние виды стартового и главного окон подробно описаны в разд. 4.

Если подлежащего выполнению проекта в пакете нет, то из базовых шаблонов, хранящихся в служеб-

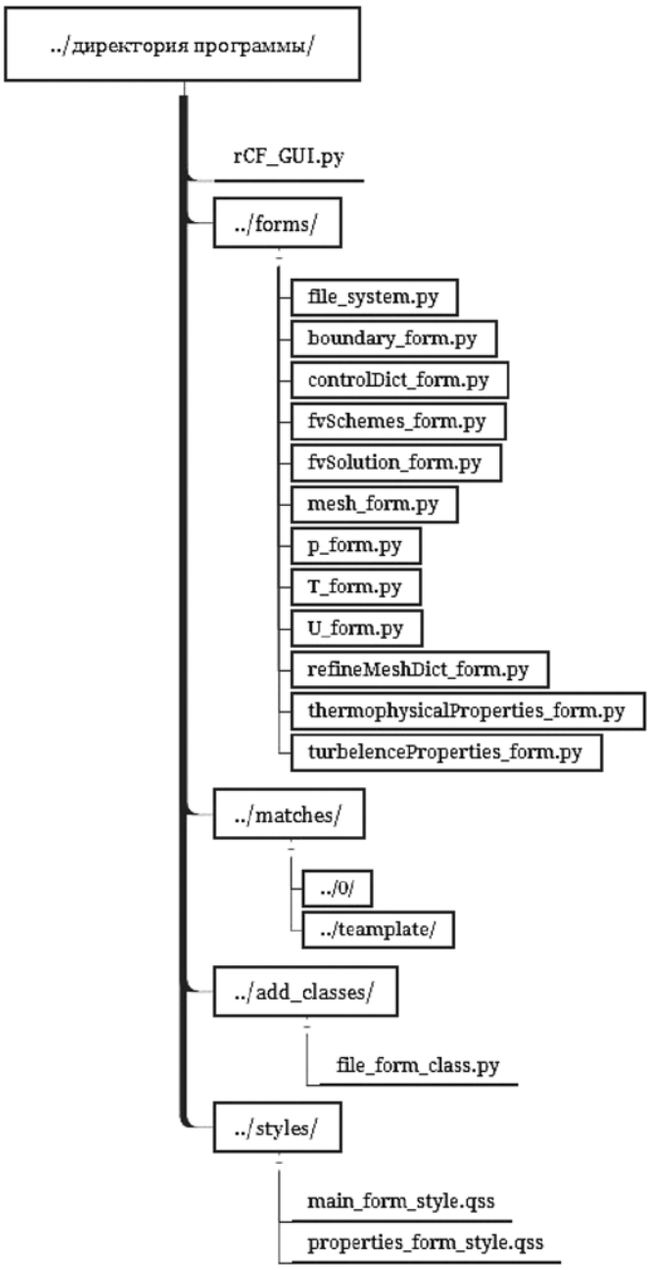

Рис. 1. Файловая структура программы

ных директориях программы, формируется стандартная структура проекта задачи ("Процедура создания"). Информация о содержании этих файлов и папок поступает в главное окно. Если подлежащий выполнению проект в пакете присутствует, то окно запуска считывает информацию о структуре проекта и отправляет ее в главное окно, минуя процедуру создания нового проекта ("Процедура открытия"). Далее пользователь определяет или изменяет уже существующие исходные данные, настраивает решатель и численную схему и запускает процесс решения. В ходе решения в главном окне программы отображается графическая информация о ходе решения — графики невязок численного решения.

Рис. 2. Алгоритм работы с программой

В ходе решения могут возникнуть три различных события, которые обрабатываются программой. Если происходит останов решения по ошибке, то программа выводит сообщение об этом. Пользователь в этом случае имеет возможность отредактировать исходные данные в главном окне программы и запустить процесс решения заново. При необходимости пользователь может принудительно остановить процесс решения, исправить исходные данные и запустить процесс решения заново, либо запустить процесс визуализации. Программный останов процесса решения происходит при достижении условий, заданных в настройках решателя.

После успешного получения решения пользователь может запустить процесс визуализации для анализа результатов, после чего он может либо изменить исходные данные и запустить задачу заново, либо завершить работу приложения.

Исходные данные автоматически сохраняются в рабочей директории проекта задачи после каждого действия пользователя.

Для запуска и управления внешними процессами программа организована как многопоточное приложение. Это сделано для того, чтобы избежать блокировки GUI-потока и увеличить эффективность выполнения программы. Управление потоками реализовано с помощью класса QtCore.QThread, а управление процессами — с помощью модуля subprocess. На рис. 3 представлена схема организации процессов и потоков.

Под контролем механизмов основного потока находятся стартовое окно программы и главное окно программы с которыми пользователь последовательно взаимодействует. Стартовое окно передает начальную информацию (имя проекта, его расположение, тип решателя) в главное окно, после чего закрывается. Код стартового окна реализован в файле rCF_GUI.py.

Основной код разметки главного окна (класс main_window) и его функций находится в модуле file_system.py. Класс main_window, предназначенный для визуализации основного рабочего пространства программы, создает элементы управления и средства визуализации процесса постановки задачи и ее решения. В частности, в главном окне отображается дерево проекта задачи OpenFOAM — оно графически отображает файловую структуру рабочей директории. Такое представление позволяет получить быстрый доступ к любому конфигурационному файлу проекта и задать необходимые значения переменных или выбрать необходимые управляющие параметры. К их числу относятся: физическая модель (способ расчета вязкости, теплоемкости и др.) и параметры решателя (выбор численной схемы и ее настройки). Выбор осуществляется посредством выпадающих списков, полей ввода данных и переключателей, которые реализованы с помощью одиннадцати интерфейсных блоков. Интерфейсные блоки реализованы в модулях с постфиксом _form, расположенных в поддиректории../forms/ (см. рис. 1). Встраивание этих модулей в главное окно происходит с помощью модуля file_form_class.py и одноименного класса, реализованного в нем. Модуль file_form_class.py отслеживает действия пользователя и выводит на экран требуемый интерфейсный блок.

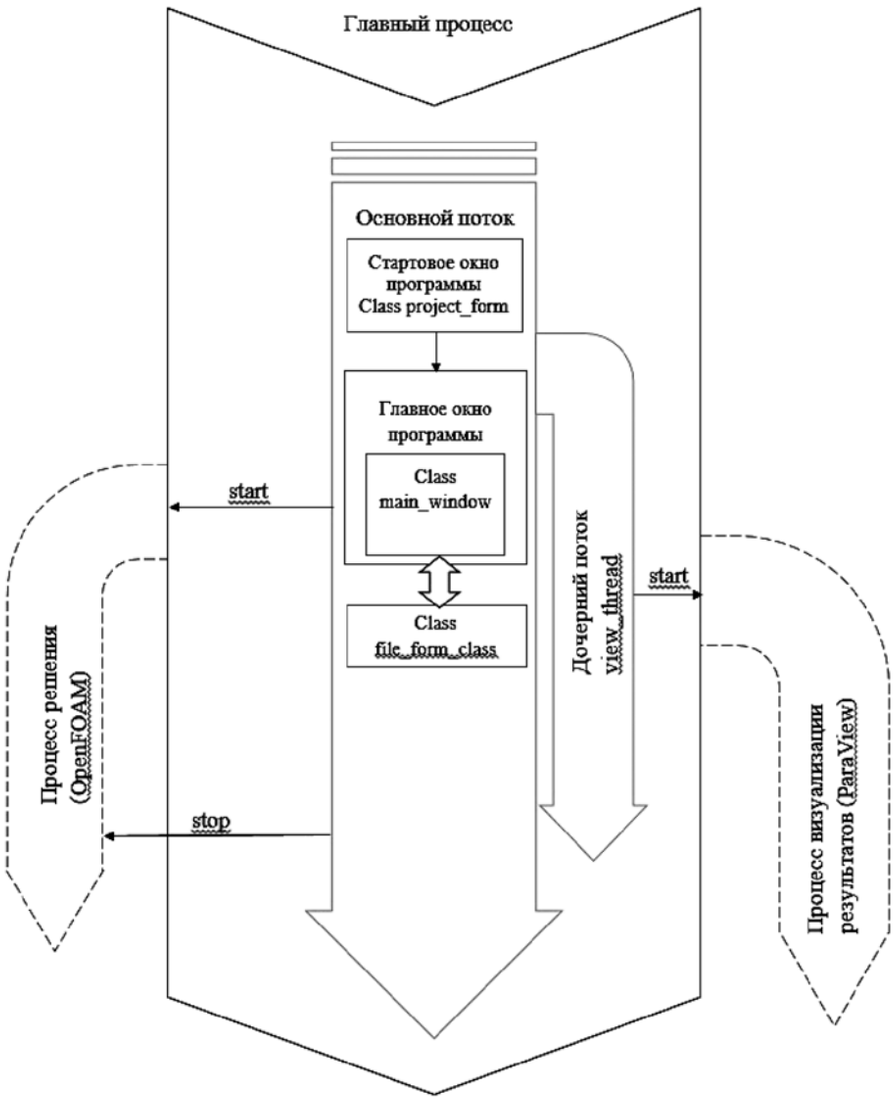

Рис. 3. Схема организации процессов и потоков

К числу интерфейсных блоков относятся модули, отвечающие за задание граничных условий (p.py, U.py, T.py), термофизических свойств (thermophysicalProperties_form.py), свойств турбулентности (turbelenceProperties_form.py), параметров решателя (controlDict_form.py) и численной схемы (fvSchemes_form.py и fvSolution_form.py), блок импорта сеточной модели (mesh_form.py), блок определения граничных условий (boundary_form.py) и блок управления измельчением сеточной модели (refineMeshDict_form.py). Интерфейсный блок для утилиты импорта расчетной сетки mesh_form импортирует сетку из выбранной рабочей директории с заданными пользователем параметрами. Формат файлов расчетной сетки может быть различным. Данный интерфейсный блок позволяет импортировать любые типы расчетных сеток, предусмотренные в среде OpenFOAM.

Основной задачей, которую решают с помощью рассматриваемого графического интерфейса,

является изменение конфигурационных файлов проекта задачи. За выполнение этой задачи и отвечают интерфейсные блоки. С каждым параметром конфигурационного файла связана определенная переменная конкретного интерфейсного блока, отвечающего за данный конфигурационный файл (пример для параметра Cp — подчеркнуто на рис. 4, см. третью сторону обложки). Доступ к изменяемым параметрам конфигурационных файлов осуществляется за счет регулярных выражений (пример для параметра Cp — на рис. 4, *б* выделено прямоугольником). Для работы с регулярными выражениями используется стандартный модуль re.

При запуске пользователем решения главный поток программы с помощью метода Popen модуля subprocess порождает процесс, выполняющий решение задачи в OpenFOAM:

```
proc = subprocess.Popen(["bash " + full_dir +
"/SOLVER_BASH"], cwd = full_dir, shell = True,
stdout = file, stderr = file)
```

Так как решение запускается в своем собственном "процессе", это не приводит к блокировке главного окна программы. После запуска процесса решения задачи механизмы основного потока контролируют ход решения путем анализа данных, которые записываются при выполнении процесса в служебный файл. Анализируя данный файл, основной поток либо представляет информацию о ходе решения в графическом виде, либо выводит сообщение об ошибках.

По запросу пользователя из основного потока графической оболочки процессу решения посылается сигнал прерывания. После завершения решения задачи пользователь может воспользоваться режимом постобработки его результатов. При запуске процедуры постобработки порождается дочерний поток. Дочерний поток, в свою очередь, запускает процесс, в котором происходит работа среды ParaView.

4. Внешний вид программы

При запуске графической оболочки rCF_GUI открывается стартовое окно программы (рис. 5), в котором осуществляется создание нового проекта задачи или выбор уже имеющегося.

После нажатия кнопки "Сохранить" перед пользователем появляется главное окно программы. На рис. 6, см. четвертую сторону обложки, приведен пример главного рабочего окна с открытым проектом.

Главное окно графической оболочки состоит из нескольких блоков.

Блок *1* представляет собой панель с кнопками управления процессом численного моделирования.

В блоке *2* реализовано дерево проекта, отображающее структуру директорий и файлов, входящих в проект.

В блоке *3* отображаются формы, с помощью которых происходит задание исходных данных в задаче.

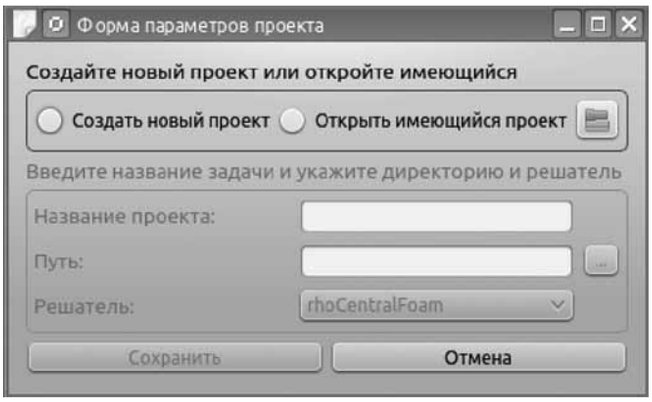

Рис. 5. Стартовое окно rCF_GUI

Блок *4* выполняет функцию оповещения пользователя о событиях, связанных с работой приложения. Дополнительное цветовое оформление сообщения позволяет улучшить визуальное восприятие оповещения.

Кроме редактирования конфигурационных файлов проекта задачи, разработанное приложение обеспечивает запуск решателя на выполнение и его останов, визуализацию процесса решения в реальном времени (см. графики сходимости переменных на рис. 7, четвертая сторона обложки) и запуск различных утилит OpenFOAM (утилиты обработки результатов расчетов, утилиты декомпозиции задачи и т. д.). Выполнение перечисленных функций осуществляется с помощью запуска соответствующих команд OpenFOAM в дочерних потоках. Это позволяет оптимизировать структуру программы, увеличивая ее быстродействие, позволяя оставить главное окно программы доступным для пользователя.

Заключение

Представлена реализация графического интерфейса для программной среды OpenFOAM для одного из решателей, имеющихся в базовом релизе OpenFOAM v3.0. В качестве инструментальных средств разработки выбран язык программирования Python 3.4 и библиотека графических компонентов PyQt4.

Разработана графическая оболочка, отвечающая представленным в данной работе требованиям. В настоящее время она находится в стадии опытной эксплуатации, но уже сейчас позволяет значительно упростить работу с решателем rhoCentralFoam.

На следующем этапе работ возможности графического пользовательского интерфейса будут расширены за счет реализации доступа к большему числу встроенных в пакет OpenFOAM утилит и решателей.

На разработанное программное обеспечение получено свидетельство отдела регистрации программ ЭВМ, баз данных и ТИМС Федерального государственного бюджетного учреждения "Феде-

ральный институт промышленной собственности" № 2016613637 от 01.04.2016, подтверждающее права собственности.

Список литературы

1. **Васильев В. А., Ницкий А. Ю., Крапошин М. В., Юскин А. В.** Исследование возможности параллельных вычислений задач гидроаэродинамики с использованием открытого пакета программ OpenFOAM на кластере "СКИФ Урал" ЮУрГУ // Тр. Междунар. конф. "Параллельные вычислительные технологии" (ПаВТ-2010). Уфа, 29 марта — 2 апреля 2010 г. Челябинск: Издательский центр ЮУрГУ, 2010. С. 422—430.

2. **Крапошин М. В., Самоваров О. И., Стрижак С. В.** Опыт использования СПО для проведения расчетов пространственной гидродинамики промышленного масштаба // Тр. конф. "Свободное программное обеспечение". СПб: Изд-во СПБГПУ, 2010. С. 44—46.

3. **Ambrosino F., Raia S., Funel A., Podda S., Migliori S.** Cross checking OpenFOAM and Fluent results of CFD simulations in ENEA-GRID environment. ENEA, Dipartimento Tecnologie Fisiche e nuovi Materiali, 2008. URL: http://www.eneagrid.enea.it/papers_presentations/papers/NapoliEScience2008_09_CRESCO.pdf.

4. **GUI,** URL: http://openfoamwiki.net/index.php/GUI

5. **ENGYS**, URL: http://www.engys.com

6. **Greenshields C. J., Weller H. G., Gasparini L., Reese J. M.** Implementation of semi-discrete, non-staggered central schemes in a colocated, polyhedral, finite volume framework, for high-speed viscous flows // International Journal for Numerical Methods in Fluids. 2010. Vol. 63, Issue 1. P. 1—21.

7. **Merkulov E. S., Lepikhov A. V., Pridannikov A. V.** Numerical simulation of the high-altitude hypersonic flow of a body under shock-wave interaction // XXX International Conference of Interaction of Intense Energy Fluxes and Matter, 2015 Elbrus. URL: http://www.ihed.ras.ru/elbrus15/program/restore.php?id = 1469

8. **Лутц М.** Программирование на Python, том I. Пер. с англ. СПб.: Символ-Плюс, 2011. 992 с.

9. **Лутц М.** Программирование на Python, том II. Пер. с англ. СПб.: Символ-Плюс, 2011. 992 с.

10. **Прохоренок Н. А.** Python 3 и PyQt. Разработка приложений. СПб.: БХВ-Петербург, 2012. 704 с.

11. **Хахаев И. А.** Практикум по алгоритмизации и программированию на Python. М.: Альт Линукс, 2010. 126 с.

Development of a Graphical User Interface for the OpenFOAM Toolbox

D. I. Chitalov, cdi9@yandex.ru, **Ye. S. Merkulov**, mes1@yandex.ru, **S. T. Kalashnikov**, src@makeyev.ru, Department of Fundamental Problems of Aerospace Technologies of Chelyabinsk Scientific Centre of Ural Branch of RAS, Miass, 456317, Russian Federation

Corresponding author:

Chitalov Dmitry I., Junior Scientific Officer, Department of Fundamental Problems of Aerospace Technologies of Chelyabinsk Scientific Centre of Ural Branch of RAS, Miass, 456317, Russian Federation, e-mail: cdi9@yandex.ru

Received on May 05, 2016
Accepted on September 09, 2016

The paper is devoted to the description of architecture of an application creating a graphical user interface for the rhoCentralFoam solver that is a part of the OpenFOAM ToolBox. The diagram of the application components is presented. The tools used in the development are also defined. The paper also presents a brief overview of the developed application.

Currently in the field of mathematical modeling of the processes of the mechanics of continuous media and, in particular, of its sections as gas and fluid mechanics, mechanics of deformable solids, there are widely used packages of applied numerical simulation programs, in particular, of OpenFOAM package of open source. Thanks to the universal standard of solvers and modules development, it is one of the most effective means of numerical simulation of gas and fluid mechanics.

Statement and solution of problems in OpenFOAM are relatively complex and require knowledge of a wide range of console commands and utilities, as well as taking into account the specifics of the working directory file structure. To solve the problem the user must create a project task, which consists of a series of text files, located in the well-defined directories in the project folder. The number of text files depends on the solver. Each of these files must be completed in accordance with the syntax of OpenFOAM.

Considering the above, the authors of this article, have decided to create their own application — rCF_GUI, which, via a graphical user interface, will provide the process automation in the pre-treatment stages and setting goals to address in OpenFOAM solver for a particular environment — rhoCentralFoam.

Keywords: *graphical user interface, OpenFOAM, Python programming language, open source software, rhoCentralFoam*

For citation:

Chitalov D. I., **Merkulov Ye. S.**, **Kalashnikov S. T.** Development of a Graphical User Interface for the OpenFOAM Toolbox, *Programmnaya Ingeneria*, 2016, vol. 7, no. 12, pp. 568—574.

DOI: 10.17587/prin.7.568-574

References

1. **Vasilyev V. A., Nitsky A. Yu., Kraposhin M. V., Yuskin A. V.** Issledovanie vozmognosti parallelnih vichisleniy zadach gidrodinamiki s ispolzovaniem otkritogo paketa program OpenFOAM na klastere UUrGU (Investigation of the possibility of parallel computation of fluid dynamics problems with using open OpenFOAM software package on a cluster "SKIF Ural" of the SUSU), *III megdunarodnaia konferencia "Parallelnie vichislitelnie technologii"* (PaVT-2010), Ufa, Marcp 29 — April 2, 2010, Chelyabinsk, Izdatelskii centr UUrGU, 2010, pp. 422—430 (in Russian).

2. **Kraposhin M. V., Samovarov O. I., Strizhak S. V.** Opit ispolzovania SPO dla provedenia raschetov prostranstvennoi gidrodinamiki promishlennogo masshtaba (Experience of the using of open source software for the calculation of the spatial scale industrial hydrodynamics), *III Konferencia "Svobodnoe programmnoe obespechenie"*, Saint Petersburg, Izd-vo SPBGPU, 2010, pp. 44—46 (in Russian).

3. **Ambrosino F., Raia S., Funel A., Podda S., Migliori S.** Cross checking OpenFOAM and Fluent results of CFD simulations in ENEA-GRID environment, ENEA, Dipartimento Tecnologie Fisiche e nuovi Materiali, 2008, available at: http://www.eneagrid.enea.it/papers_presentations/papers/NapoliEScience2008_09_CRESCO.pdf

4. **GUI,** available at**:** http://openfoamwiki.net/index.php/GUI

5. **ENGYS,** available at**:** http://www.engys.com

6. **Greenshields C. J., Weller H. G., Gasparini L., Reese J. M.** Reese Implementation of semi-discrete, non-staggered central schemes in a colocated, polyhedral, finite volume framework, for high-speed viscous flows, *International Journal for Numerical Methods in Fluids,* 2010, vol. 63, Issue 1, pp. 1—21.

7. **Merkulov E. S., Lepikhov A. V., Pridannikov A. V.** Numerical simulation of the high-altitude hypersonic flow of a body under shock-wave interaction, *XXX International Conference of Interaction of Intense Energy Fluxes and Matter,* 2015, Elbrus, available at: http://www.ihed.ras.ru/elbrus15/program/restore.php?id=1469

8. **Lutz M.** *Programmirovanie na Python* (Programming Python), volume I. Translation from English, Saint Petersburg, Simvol-Plyus, 2011, 992 p. (in Russian).

9. **Lutz M.** *Programmirovanie na Python* (Programming Python*),* volume II. Translation from English, Saint Petersburg, Simvol-Plyus, 2011, 992 p. (in Russian).

10. **Prokhorenok N. A.** *Python 3 i PyQt. Razrabotka prilogenii* (Python 3 and PyQt. Application Development), Saint Petersburg, BKhV-Peterburg, 2012, 704 p. (in Russian).

11. **Khakhayev I. A.** *Practicum po algorithmizacii i programmirovaniu na Python* (Practical work on algorithms and programming in Python), Moscow, ALT Linux, 2010, 126 p. (in Russian).

ИНФОРМАЦИЯ

3—4 марта 2017 г.
в отеле Holiday Inn Moscow Vinogradovo состоится

4-я Международная научно-практическая конференция

«Инструменты и методы анализа программ, TMPA-2017»

На конференции будут представлены доклады признанных специалистов в области программной инженерии и конкурсные доклады, прошедшие рецензирование несколькими независимыми экспертами.

Темы, рассматриваемые на конференции, включают (но не ограничиваются):

- автоматизацию тестирования программного обеспечения
- статический анализ программ
- верификацию
- динамические методы анализа программ
- тестирование и анализ параллельных и распределенных систем
- тестирование и анализ высоконагруженных систем и систем высокой доступности
- анализ и верификацию программно-аппаратных систем
- методы создания качественного программного обеспечения
- инструментальные средства анализа, тестирования и верификации

Подробности: http://tmpaconf.org

Указатель статей, опубликованных в журнале "Программная инженерия" в 2016 г.

ООО "Издательство "Новые технологии". 107076, Москва, Стромынский пер., 4
Технический редактор *Е. М. Патрушева*. Корректор *Е. В. Комиссарова*

Сдано в набор 04.10.2016 г. Подписано в печать 17.11.2016 г. Формат 60×88 1/8. Заказ PI1216
Цена свободная.

Оригинал-макет ООО "Адвансед солюшнз". Отпечатано в ООО "Адвансед солюшнз".
119071, г. Москва, Ленинский пр-т, д. 19, стр. 1. Сайт: **www.aov.ru**