УДК 004.43

Доренская Е.А., Семенов Ю.А.

НИЦ «Курчатовский институт» ФГБУ «ГНЦ РФ ИТЭФ», г. Москва, Россия

О ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ, ОРИЕНТИРОВАННОЙ НА МИНИМИЗАЦИЮ ОШИБОК

Аннотация

Рассматривается возможность создания языка описания проблемы, а не алгоритма. С помощью этой техники можно минимизировать программные ошибки и упростить процесс программирования. Сообщается о создании банка алгоритмов. Предложен метод вариации программ на основе входных параметров. Описан метод древовидных графов как способ диалога между компьютером и программистом и дана схема программы на основе этого метода.

Ключевые слова

Язык описания проблемы; минимизация программных ошибок; принцип "many eyes"; компьютерная программа; граф диалога компьютер-программист; банк алгоритмов; метод древовидных графов; граф описания регулярных выражений.

Dorenskaya E.A., Semenov Y.A.

Institute for Theoretical and Experimental Physics, Moscow, Russia

ABOUT THE PROGRAMMING TECHNIQUES, ORIENTED TO MINIMIZE ERRORS

Abstract

The article focuses on the idea of creating a language for the description of the problem, not an algorithm. It talks about how by using it one can minimize software errors and simplify the process of programming. It also talks of the creation of a bank of algorithms. It is proposed a method of program modification, based on input parameters. The method is described by tree graphs as a method of dialogue between the computer and the programmer and given the scheme of the program based on this method.

Keywords

Language for description of problem; minimizing programming errors; principle of "many eyes"; computer program; dialog computer-programmer; bank of algorithms; method of tree graphs; graph of description for regular expressions.

Введение

Сегодня от качества компьютерных программ зависят практически все стороны человеческой жизни, включая здоровье [1]. По этой причине важно минимизировать число программных ошибок. На сегодняшний день хорошим результатом по надежности программного обеспечения считается 0,5 ошибок на 1000 строк кода (это результат использования статических и динамических code-checker'ов [2], например, для ядра ОС Микрософт Windows). Но для многих проблем это слишком плохо.

Впервые данную тему мы подняли в 2013г на семинаре ВМК, где обзор по проблемам программных ошибок делал В.П. Иванников.

В данной статье мы рассматриваем частную задачу оптимизации программирования с целью минимизации ошибок. Более общая задача уже была рассмотрена в [3,4,5,6,12].

Цель исследования

Нашей целью является минимизация числа программных ошибок. Проблема в том, что полностью выявить и устранить ошибки невозможно. Не существует даже хороших методов оценки числа имеющихся программных ошибок [7]. Одним из эффективных проверенных способов сокращения числа ошибок является использование программы большим числом пользователей с последовательным выявлением и корректировкой ошибок (принцип "**many eyes**").

Основная часть

Одним из мотивов программы "open source" является минимизация количества программных ошибок за счет многократной проверки для самых разных приложений. Данная работа является продолжением и развитием идей, изложенных в [8].

Другим подходом, который можно совмещать с методикой many eyes, является минимизация участия человека в написании программы.

Оптимальным решением проблемы представляется описание задачи на естественном языке, например, на русском или английском, с последующим преобразованием этого описания в текст программы на одном из алгоритмических языков высокого уровня.

Судя по всему, для этого потребуется искусственный интеллект. К сожалению, он пока слаб [9], но что хуже, не очевидно, что он позволит получить меньшее число программных ошибок по сравнению с обычным программистом [10].

В процессе формирования исполняемого программного модуля могут использоваться библиотечные программы, макросы, include (программные заготовки и т.д.).

Если раньше задача решалась комбинированием команд процессора, теперь можно для решения проблемы комбинировать более крупные программные модули. Многообразие таких модулей будет более чем на порядок-два больше множества команд процессора. Функция модулей может варьироваться посредством специальных вычисляемых модификаторов.

Такие модули должны размещаться в отдельных банках данных соответствующей специализации. Должна быть построена иерархия этих модулей.

Примерами таких модулей могут быть программы анализа текстов файлов (поиск оговоренных образцов в log-файлах), преобразование массивов данных в графические образы, модули статистической обработки данных и т.д. Такие модули должны стать в данном языке основными структурными элементами, из которых строится текст описания проблемы и сама программа.

Все модули исполняемой программы должны быть написаны на одном и том же алгоритмическом языке.

Следует иметь в виду, что описание проблемы на мета-языке (ML) может также содержать в себе ошибки, могут быть ошибки и в самом макро-трансляторе. Все это будет приводить в ошибкам в конечной программе. Но так как макро-транслятор будет использоваться часто, можно ожидать, что ошибки в трансляторе будут устранены относительно быстро. Ошибки же в описании проблемы должны выявляться в процессе отладки и тестирования программы. Но так как текст описания должен быть относительно коротким, можно предположить, что ошибок там будет мало.

Текст описания проблемы на макроязыке сначала анализируется на предмет выявления области проблемы (граф диалога компьютер-программист). Далее предпринимается попытка разделить проблему на субзадачи. После этого делается попытка найти описание алгоритма, соответствующее каждой из субзадач. Для каждого из найденных алгоритмов определяются списки входных и выходных параметров. Для каждой из функций, если ее описание не удалось найти в банке описаний алгоритмов, создается свой программный модуль или пакет модулей, выполненных согласно правилам Хольцмана [11].

Стоящая перед нами проблема может быть разделена на несколько уровней:

1. Создание банка алгоритмов, где ошибки минимизируются за счет многократного использования программы большим числом пользователей [8];
2. Для проблем, которые могут быть описаны древовидным графом, текст программы формируется из модулей, взятых из банка алгоритмов, а также модулей, написанных самим программистом (возможно помещение этих модулей в банк алгоритмов);
3. Сложные проблемы, характеризуемые графом с циклическими структурами, программа формируется из стандартных исполнительных модулей разного уровня.

Если модель языка описания проблемы отличается от описанной выше, то:

а. Для описания проблемы используется естественный язык;

б. Создается специальный язык, а транслятор переводит описание на один из алгоритмических языков высокого уровня, например, на СИ;

• В варианте “а” потребуется привлечение искусственного интеллекта, с неочевидными последствиями для частоты программных ошибок;

• В варианте “б” либо нужен искусственный интеллект, либо алгоритмический язык нужно модифицировать и приблизить к модели, описанной выше.

При создании банка описания алгоритмов существуют определенные проблемы. Так как источники программных модулей достоверно неизвестны, хакеры могут пытаться поместить в банк модули, зараженные вредоносными кодами. Перед укладкой любых программных модулей в депозитарий их следует проверять антивирусом. Полезно проверять текст модуля на наличие URL, которые могут указывать на вредоносные сайты. Модули с URL в тексте программы в банк помещать не следует.

Метод вариации программ на основе входных параметров

Все существующие программы вариативны. В одних обстоятельствах используется один фрагмент кода, в других – другой. Любой процессор имеет фиксированный набор команд и, комбинируя их, можно решить практически любую задачу. Для управления порядком команд и упрощения задачи программиста были созданы десятки алгоритмических языков разного уровня и назначения. Строка кода на алгоритмическом языке соответствует двум и более командам процессора. Это, кроме удобства программирования, понижает также и вероятность программных ошибок.

Для вариации программы используется вычисление определенных функций и проверка результата этого расчета. По результатам проверки принимается решение о выполнении того или иного фрагмента кода из числа имеющихся.

Алгоритм модифицируется, так чтобы отвечать требованиям списков входных и выходных параметров (см. рис. 1). Нужно помнить, что ошибка в программе модификации будет дублировать свои ошибки в каждое приложение! По это причине требования к таким программам должны быть особо жесткими. Надеяться здесь можно только на то, что такие программы будут достаточно примитивны и ограничены по объему. Возможности модификации конкретного программного модуля не беспредельны. Варианты модификации предусматриваются в процессе создания такого модуля.

Далее часть кодов можно ввести внутрь стандартных модулей, сделав их более универсальными. Эти коды (сходные с макросами, применяемыми в некоторых языках) будут генерироваться программно на основе параметров программы, заданных извне. Туда попадут части программ, адаптирующих данные для работы стандартных модулей. Это могут быть описания регулярных выражений. Программы подготовки данных, например, для работы Gnuplot и т.д. Для решения этих проблем создаются специализированные программные интерпретаторы.

Особое внимание нужно уделить выявлению программных функций, которые оказываются особо часто востребованными. Именно такие программы следует в первую очередь включить в библиотеку описаний алгоритмов.

Одним из путей минимизации числа программных ошибок может стать создание большого числа библиотечных программ, а также обеспечение адаптации библиотечных программ для решения более широкого круга задач.

Требования к описаниям программных модулей для людей и программ принципиально различны. Такие описания для людей могут быть составлены на естественном языке в предположении, что клиент будет их просматривать последовательно. Любые описания и комментарии в программах должны выполняться на английском языке (так мы сделаем язык описаний универсальным).

Для компьютера описание программы может представлять собой список ключевых слов ($\mathbf{W_k}$), которые наилучшим образом характеризуют данный алгоритм или регулярное выражение, например, $\mathbf{W_k}=(\mathbf{w_{1,k}}, \mathbf{w_{2,k}}, \mathbf{w_{3,k}}, \ldots, \mathbf{w_{n,k}})$, где $\mathbf{k}$ – номер описания алгоритма, $\mathbf{n}$ – число ключевых слов, описывающих данный конкретный алгоритм). Этот список должен составлять человек, желательно автор данной программы. Число таких ключевых слов для каждого из алгоритмов индивидуально.

Задание делится на части, каждая из которых однородна и решает одну и только одну задачу.

Программа может формироваться на основе входных параметров (***параметрическое управление***), например, как в случае модификации кода для многовариантной программы, например, **m_gnuplot** рис. 1. Программы, созданные в результате диалога, могут быть также помещены в банк алгоритмов.

Описание процедуры может содержать строку:

<input parameter list> <action> <output parameter list>, где оператор <action> должен быть тщательно описан, так чтобы исключить неоднозначность интерпретации. Имени <action> может соответствовать запись в банке алгоритмов. Должно существовать описание на русском и английском языках, а также список ключевых слов, которые могут использоваться при поиске.

В пределах одного типа **<action>** может быть много субтипов. Для типов и субтипов должны быть определены конкретные операции, например:

А. Вычисления, включая параллельные;
Б. Анализ текста и его преобразование;
В. Статистический анализ;
Г. Диагностика сети;
Д. Мониторинг операций реального времени;
Е. Работа с базами данных;
Ж. Работа со списками;
З. Работа с объектами Интернета вещей;
И. Компьютерная аналитика (когнитивный компьютинг);
К. Криптографические вычисления и т.д.;

Если проблема может быть решена программным модулем, хранящемся в банке алгоритмов, ищем и используем такой модуль.

Если нет, используем ***диалоговый язык программирования***, где программа задает программисту последовательность вопросов, требующих ответов. Программа предлагает для каждого вопроса возможные варианты ответов, а программист выбирает один или несколько вариантов. По завершении диалога программа формирует текст исполняемого кода. Примером может служить вариант для регулярных выражений (рис. 3). Если выбор хотя бы в одной точке сделан не правильно, будет получена программы, которая решает не ту проблему, которая имелась в виду.

Среди списка входных и выходных параметров могут быть: *переменные (числовые и строковые), массивы переменных, списки, файлы, каталоги, базы данных и даже программные модули, извлекаемые из банка алгоритмов со своими списками входных параметров.* Транслятор должен тщательно анализировать входные и выходные параметры и диагностировать допущенные программистом ошибки. Рекурсивных вложений параметров нужно избегать, так как это делает программу непрозрачной и увеличивает вероятность ошибок.

На рис.1 показан пример программы, управляемой входными параметрами. Программа служит для формирования графических распределений с привлечением библиотечных функций gnuplot (библиотека Chart-Graph-3.2, которая содержит в себе несколько модулей). Разные модули Chart-Graph предназначены для получения различных выходных графических форм и требуют разных форматов входных данных.

Если какая-то функция в банке описаний отсутствует, то она может быть создана в тексте описания проблемы на стандартном алгоритмическом языке, например, на PERL. Эта техника используется и в других языках, например, в HTML/XML, где делаются вставки текста на JavaScript или PHP.

В начале и в конце такого модуля ставятся метки-разделители, указывающие на алгоритмический язык вставки. Например:

<perl> (текст программы на языке Perl с учетом рабочей ОС) **</perl>**

На первом уровне M1 (рис. 1) на основе входных данных производится выбор варианта 1-6. На следующих уровнях может производиться уточнение варианта.

Программа, написанная на ML, будет строиться из готовых модулей кода, хранящихся в банках алгоритмов разного уровня (рис. 2,3), а также программных модулей, написанных специально для решения текущей проблемы.

Чтобы выработать язык описания проблемы, надо зафиксировать базовые имена основных задач и их модификаций (тезаурус имен задач). Для каждого из имен должно быть представлено описание/определение, задающие область применимости.

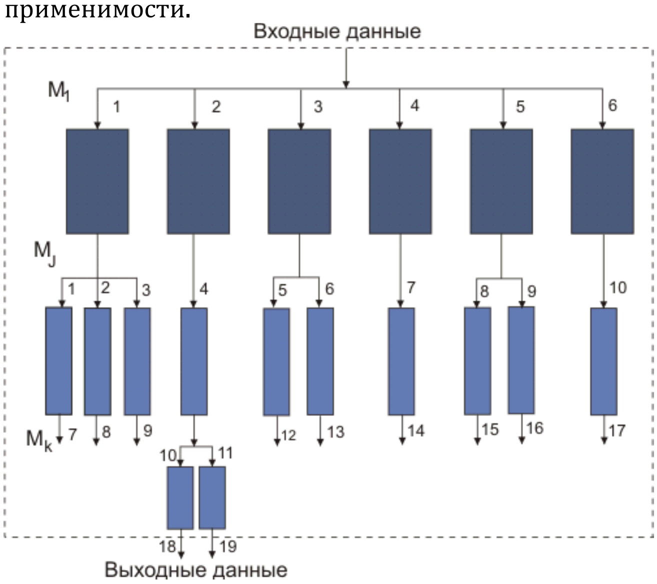

Рис. 1. Схема реализации программы типа ***m_gnuplot***

Эти имена образуют древовидный граф, и конкретная проблема может быть идентифицирована в результате диалога пользователь-программа.

Метод древовидных графов

Одним из путей полуавтоматического формирования программы является диалог между компьютером и программистом. Этот метод применим, когда такой диалог может быть описан древовидным графом и имеется набор готовых программных модулей в банке алгоритмов.

Если сама программа может быть создана в диалоговом режиме, то конечный узел, соответствующий имени проблемы, будет корневым узлом диалога создания программы (рис. 2).

Прямоугольники в нижней части рис. 2 соответствуют разным типам техники создания программы. Вариант метода deep learning используется для проблем распознавания (SPAM, атаки нулевого дня и пр.).

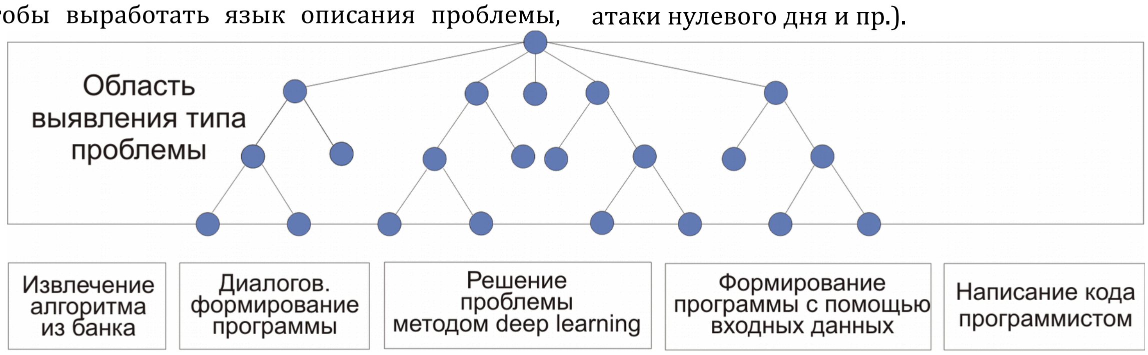

Рис. 2. Метод графа для описания проблемы в диалоговом режиме

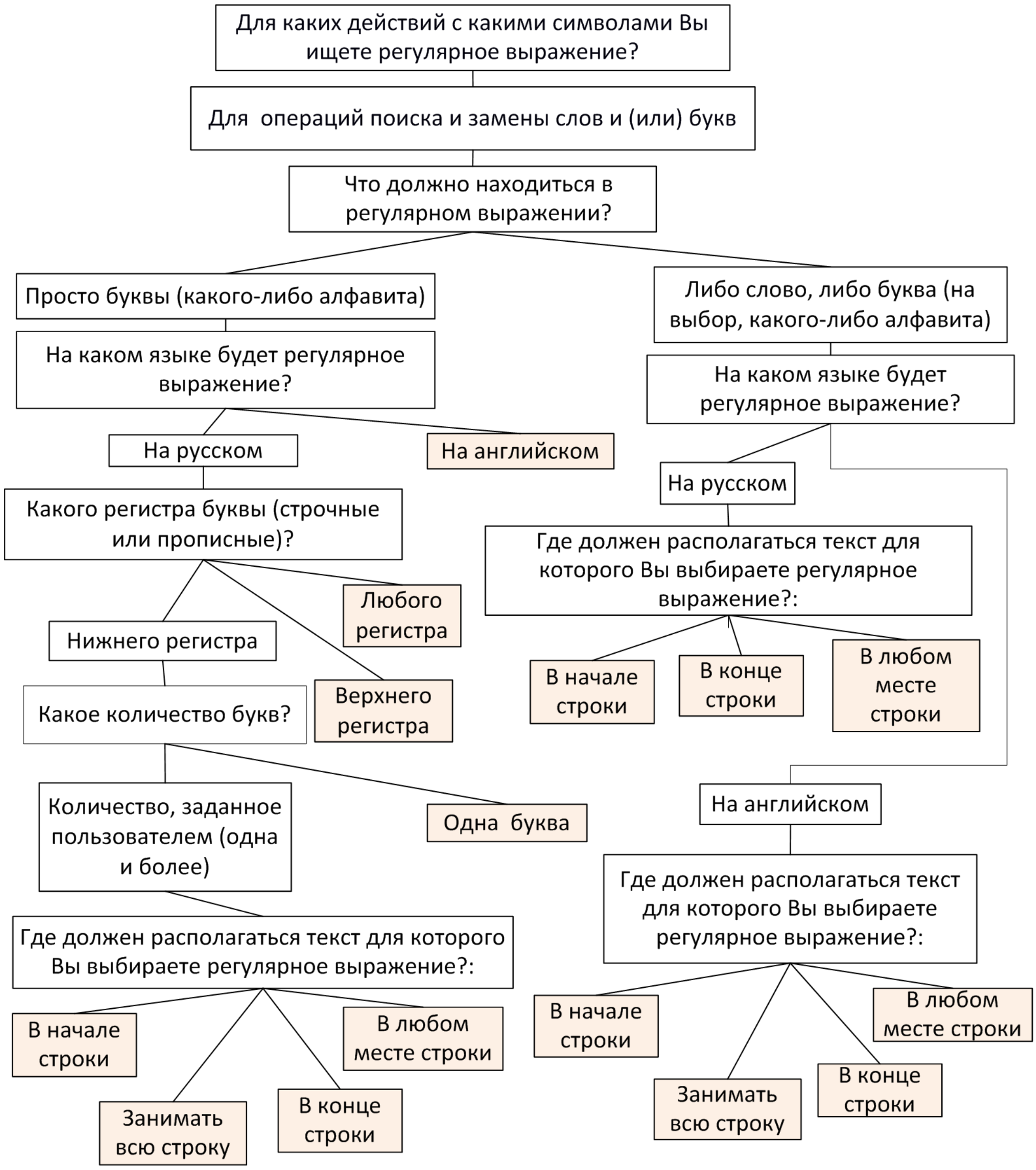

Рис. 3. Граф описания регулярных выражений

Оптимально формирование списка ключевых слов для поиска программного модуля посредством диалога (вопрос-ответ). Диалог при составлении списка ключевых слов для банка алгоритмов и при формировании запроса поиска в банке алгоритмов должен проводиться идентично, что будет гарантировать идентичность списков ключевых слов. В случае регулярного выражения метод древовидного графа успешно работает (рис. 3). Дерево графа определяет число вариантов регулярных выражений, которые могут быть сформированы.

Каждая диалоговая программа ориентирована на создание определенного класса кодов. Число таких программ будет достаточно велико и со временем будет расти.

Программа или конкретный человек может сформировать свой список ключевых слов, характеризующих алгоритм, который нужен $\mathbf{W_i=(w_{1,i}, w_{2,i}, w_{3,i}, \ldots, w_{m,i})}$, где $\mathbf{i}$ – номер запроса алгоритма в банк данных, а $\mathbf{m}$ – число ключевых слов в списке. Причем, этот список может отличаться как по содержанию, так и по длине от списка, хранящегося в банке описаний алгоритмов. Задача может осложняться существованием синонимов для некоторых ключевых слов, а также наличием опечаток. Для исключения опечаток надо обязательно проводить орфографический контроль. Порядок ключевых слов в запросе и в поле описания банка данных при такой схеме будет практически всегда отличаться.

Если узел графа (рис. 3) однозначно определяет регулярное выражение, то ключевые слова будут не нужны, да и сама база данных или хэш-массив также будут не нужны, так как граф будет выходить непосредственно на регулярное выражение. На рис. 3 приведен фрагмент дерева для получения регулярного выражения.

Программа, реализующая алгоритм, отображенный на рис. 3, может быть также помещена в банк алгоритмов.

Программы диалоговой техники программирования могут создаваться с привлечением шаблонных схем, куда программист заносит тексты вопросов и возможных ответов в поля определенной формы, подготавливая программу диалога.

Рассмотрим в качестве примера диалогового метода формирования программ систему для мониторинга и обеспечения безопасности WEB-сервера.

Данная программа должна анализировать журнальные файлы (access_log, error_log, secure и др.), выявлять случаи атак и блокировать доступ к WEB-узлу с соответствующего IP-адреса, если число атак превысило пороговый уровень.

Задание порогов атак отдельно для каждого типа, имя базы данных, таблицы (если имеются), имена журнальных файлов для каждого типа атак, требования к формированию графики производится в темном блоке на рис. 4 “Зона обработки результата”). Предполагается, что в банке описаний алгоритмов имеются модули, соответствующие субзадачам, названным в вертикальных прямоугольниках рис. 4. Может выбираться один вариант или любая их комбинация.

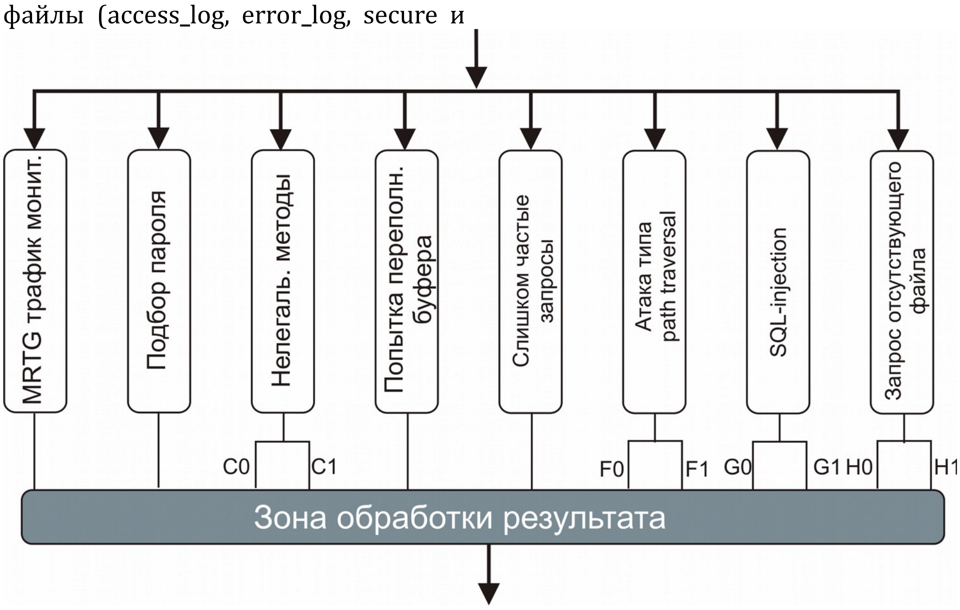

Рис. 4. Граф выбора варианта задачи для программы обеспечения безопасности WEB-сервера

Обозначения на рис. 4: **C0** – использование методов put или delete. **C1** – использование методов post и т.д.. **F1** – детектирование в журнальном файле access_log случаев "../" , "/../../../../", %2e%2e%2f, ..%c0%af или %2e%2e/ отдельно. **G1** – случаи, когда SQL-база на сервере отсутствует. **H1** – выявление в запросе потенциально опасных файлов

После завершения диалога программа формирует из имеющихся модулей программу, решающую данную проблему самостоятельно.

Полученные результаты

Опробованные в работе методы программирования с управлением входными параметрами и диалоговый подход предполагают фиксированный набор вариантов программы и не являются универсальными, но они позволяют подготовить достаточно обширный перечень программ и уменьшить число программных ошибок.

Заключение

Нами предложены некоторые подходы для минимизации программных ошибок. Эти методы наряду минимизацией числа ошибок посредством использования программы большим числом пользователей с последовательным выявлением и корректировкой неправильного кода (принцип **"many eyes"**) позволят поднять надежность программ.

Опробован метод вариации программ на основе входных параметров, который позволяет автоматически выбирать нужный алгоритм действий для заданных входных данных.

На примере приложения Gnuplot продемонстрирована возможность диалогового метода программирования, с помощью которого программа задаёт вопросы программисту и сама формирует текст программы.

В дальнейшем планируется разработать метаязык описания проблемы [8], что предоставит универсальное решение для минимизации числа ошибок.

Литература

1. “Экономика в 2016 году и через 10 лет”, Семенов Ю.А. Экономические стратегии N1, 2017 стр.126.
2. Handbook of Theoretical Computer Science Methods and logics for Proving Programs, Chapter 15, Patrick Cousot, Elsevier Science Publishers B.V., 1990, p.844.
3. Ильин А. В. Конструирование разрешающих структур на задачных графах системы знаний о программируемых задачах // Информационные технологии и вычислительные системы, 2007. №3, с. 30-36.
4. Ильин В. Д. Система порождения программ. М.: Наука, 1989, 264 с.
5. Тыугу Э. Х. Концептуальное программирование. М.: Наука, 1984, 256 с.
6. Ильин А. В., Ильин В. Д. Систематизация знаний о программируемых задачах // Системы и средства информатики, 2014. Том 24, № 3, с. 192-203.
7. https://habrahabr.ru/post/122912/ - Оценка количества ошибок в программе. Модель Миллса
8. "Разработка банка алгоритмов и основ языка описания проблем с целью минимизации числа программных ошибок", Ю.А.Семенов, А.П.Овсянников, Т.В.Овсянникова, "Труды НИИСИ РАН том 6, №2", Москва, 2016, с. 96-100.
9. “Когда компьютеры станут умнее человека”, Chip 09/14, стр 24-25.
10. https://geektimes.ru/post/286304/ Нейросеть DeepCoder учится программировать, заимствуя код у других программ.
11. Правила Хольцмана, http://book.itep.ru/10/holz_rules.htm.
12. Иванников В. П., Белеванцев А. А., Бородин А., Игнатьев В., Журихин Д., Аветисян А. И., Леонов М. Статический анализатор Svace для поиска дефектов в исходном коде программ // Труды Института системного программирования РАН. 2014. Т. 26. № 1. С. 231-250.

References

1. “Jekonomika v 2016 godu i cherez 10 let”, Semenov Ju.A. Jekonomicheskie strategii N1, 2017 str.126.
2. Handbook of Theoretical Computer Science Methods and logics for Proving Programs, Chapter 15, Patrick Cousot, Elsevier Science Publishers B.V., 1990, p.844
3. Il'in A. V. Konstruirovanie razreshajushhih struktur na zadachnyh grafah sistemy znanij o programmiruemyh zadachah // Informacionnye tehnologii i vychislitel'nye sistemy, 2007. №3, str. 30-36.
4. Il'in V. D. Sistema porozhdenija programm. M.: Nauka, 1989, 264 str.
5. Tyugu Je. H. Konceptual'noe programmirovanie. M.: Nauka, 1984, 256 str.
6. Il'in A. V., Il'in V. D. Sistematizacija znanij o programmiruemyh zadachah // Sistemy i sredstva informatiki, 2014. Tom 24, № 3, str. 192-203.
7. Ocenka kolichestva oshibok v programme. Model' Millsa
8. "Razrabotka banka algoritmov i osnov jazyka opisanija problem s cel'ju minimizacii chisla programmnyh oshibok", Ju.A.Semenov, A.P.Ovsjannikov, T.V.Ovsjannikova, "Trudy NIISI RAN tom 6, №2", Moskva, 2016, str. 96-100.
9. Kogda komp'jutery stanut umnee cheloveka”, Chip 09/14, str. 24-25
10. https://geektimes.ru/post/286304/ Nejroset' DeepCoder uchitsja programmirovat', zaimstvuja kod u drugih program
11. Pravila Holcmana, http://book.itep.ru/10/holz_rules.htm
12. Ivannikov V. P., Belevancev A. A., Borodin A., Ignat'ev V., Zhurihin D., Avetisjan A. I., Leonov M. Staticheskij analizator Svace dlja poiska defektov v ishodnom kode programm // Trudy Instituta sistemnogo programmirovanija RAN. 2014. T. 26. № 1. str. 231-250

Поступила: 15.05.2017

Об авторах:

Доренская Елизавета Александровна, инженер-программист, НИЦ «Курчатовский институт» ФГБУ «ГНЦ РФ ИТЭФ», dorenskaya@itep.ru;

Семёнов Юрий Алексеевич, кандидат физико-математических наук, ведущий научный сотрудник, НИЦ «Курчатовский институт» ФГБУ «ГНЦ РФ ИТЭФ»; заместитель заведующего кафедрой информатики и вычислительных сетей Института нано-, био-, информационных, когнитивных и социогуманитарных наук и технологий, Московский физико-технический институт (государственный университет), semenov@itep.ru.

Note on the authors:

Dorenskaya Elizaveta A., software engineer, Institute for Theoretical and Experimental Physics, dorenskaya@itep.ru;

Semenov Yuri A., Candidate of Physical and Mathematical Sciences, Leading Researcher, Institute for Theoretical and Experimental Physics; Deputy Head of the Department of Informatics and Computer Networks, Department of Nano-, Bio-, Information Technology and Cognitive Science, Moscow Institute of Physics and Technology (State University), semenov@itep.ru.