Секция 5

МАТЕМАТИЧЕСКИЕ ОСНОВЫ ИНФОРМАТИКИ И ПРОГРАММИРОВАНИЯ

УДК 519.682

АНАЛИТИЧЕСКИЙ ПОДХОД В ТЕОРИИ КОНТЕКСТНО-СВОБОДНЫХ ЯЗЫКОВ В НОРМАЛЬНОЙ ФОРМЕ ГРЕЙБАХ

О. И. Егорушкин, К. В. Сафонов

Словарем языка является конечное множество $X = \{x_1, \ldots, x_n\}$ слов языка, называемое терминальным множеством, а $Z = \{z_1, \ldots, z_m\}$ — множество вспомогательных символов z_j, необходимых для задания грамматических правил, называемое нетерминальным множеством; $W^* = (X \cup Z)^*$ — соответствующая свободная полугруппа относительно операции конкатенации.

Кс-грамматике соответствует система полиномиальных уравнений

$$z_j = p_j(x, z), z = 1, \ldots, m, \tag{1}$$

называемая системой уравнений Хомского — Щютценберже, а кс-языком называется первая компонента z_1 ее решения $(z_1(x), \ldots, z_m(x))$, получаемого методом последовательных приближений. В результате итераций компоненты z_j выражаются формальными степенными рядами.

В настоящей работе предлагается изучать вместо системы уравнений Хомского — Щютценберже эквивалентную ей систему в нормальной форме Грейбах, в которой все многочлены имеют вторую степень по переменной z; переход к нормальной форме Грейбах всегда возможен за счет увеличения, быть может, числа переменных. Для уравнений второй степени аналитические методы исследования представляются более перспективными: их эффективность в большей мере зависит от степени уравнений, чем от числа переменных.

Более точно, нормальная форма Грейбах системы (1) подразумевает, что все входящие в нее многочлены имеют вид $p_j(x, z) = f_j(z) + g_j(x, z) + h_j(x)$, где $f_j(z)$ — квадратичная форма от переменных $z_1, \ldots, z_m$, многочлен $g_j(x, z)$ линеен по $z_1, \ldots, z_m$, а многочлен $h_j(x)$ от них не зависит. А значит, можно считать, что система (1) имеет вид

$$z_j = f_j(z) + g_j(x, z) + h_j(x), j = 1, \ldots, m. \tag{1*}$$

Поставим в соответствие формальному степенному ряду (многочлену) ряд (многочлен) с комплексными переменными, задав отображение терминальных x_i и нетерминальных z_i символов из множества $X \cup Z$ в множество комплексных переменных. Получаем фиксированный гомоморфизм, который ставит в соответствие формальному ряду его *коммутативный образ* (2) — сходящийся в окрестности нуля степенной ряд от комплексных переменных

$$ci(r) = \sum_k a_k z^k, \tag{2}$$

где

$$a_k z^k = a_{k_1,\ldots,k_n} z_1^{k_1} \ldots z_1^{k_n},$$

$$a_k = \sum_{\sharp x_1(w_i)=k_1,\ldots,\sharp x_n(w_i)=k_n} < r, w_i >,$$

символ $\sharp c(d)$ означает число вхождений символа c в моном d.

Рассмотрим коммутативный образ кс-языка z_1

$$ci(z_1) = a_k x^k = a_{k_1,\ldots,k_n} x_1^{k_1} \ldots x_n^{k_n}, \tag{3}$$

который является алгебраической функцией, а также коммутативный образ

$$ci(L) = \sum_{k_0 \geqslant 0, k \geqslant 0} L_{k_0,k} x_0^{k_0} x^k \tag{4}$$

некоторого линейного языка над терминальными символами $x_0, x_1, \ldots, x_n$, который является рациональной функцией.

Будем называть ряд (3) *диагональю* ряда (4), если при всех $k_1, \ldots, k_n$ выполнено условие

$$a_{k_1,\ldots,k_n} = L_{k_1,k_1,k_2,\ldots,k_n}.$$

Теорема 1. Если однородные многочлены второй степени $f_2(z), \ldots, f_n(z)$, входящие в систему (1*), не зависят от переменной z_1 и система уравнений

$$f_j(z) = 0, j = 2, \ldots, n,$$

имеет единственный нуль $z = 0$, то коммутативный образ кс-языка, порожденного системой (1*), является диагональю некоторого линейного языка.

Порождающая кс-язык система уравнений (1*) в нормальной форме Грейбах позволяет установить, является данный формальный степенной ряд контекстно-свободным языком или нет. Сгруппируем коммутативный образ формального языка в ряд по однородным многочленам

$$ci(z_1) = \sum_j (a)_j(x), \tag{5}$$

где $(a)_j(x)$ — однородный многочлен степени j. Достаточно установить, что ряд (5) удовлетворяет уравнению степени 2, поскольку такую степень имеет система (1*). Возводя ряд (5) в квадрат, получим ряд $(ci(z_1))^2 = \sum_j (a)_j^2(x)$. Тот факт, что оба ряда, будучи умноженными на многочлены, дают в сумме тождественный нуль, означает, что достаточно длинные отрезки этих рядов линейно зависимы.

Теорема 2. Для того чтобы ряд по однородным многочленам (5) удовлетворял полиномиальному уравнению степени 2, необходимо и достаточно, чтобы при всех $j \geqslant j_0$, $l \geqslant l_0$ выполнялось равенство

$$\begin{vmatrix} (a)_j(x) & \ldots & (a)_{j+l}(x) & (a)_j^2(x) & \ldots & (a)_{j+l}^2(x) \\ (a)_{j+1}(x) & \ldots & (a)_{j+l+1}(x) & (a)_{j+1}^2(x) & \ldots & (a)_{j+l+1}^2(x) \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ (a)_{j+q}(x) & \ldots & (a)_{j+l+q}(x) & (a)_{j+q}^2(x) & \ldots & (a)_{j+l+q}^2(x) \end{vmatrix} \equiv 0,$$

где $q = 2l + 1$.

УДК 004.94

ПОИСК ПРОГРАММНЫХ ОШИБОК В АЛГОРИТМАХ ОБРАБОТКИ СЛОЖНО-СТРУКТУРИРОВАННЫХ ДАННЫХ

А. Н. Макаров

При разработке программных комплексов часто решается задача по интеграции в них программных модулей, для которых отсутствуют исходные коды и сопроводительная техническая документация. Для обеспечения надежности функционирования комплекса в целом может потребоваться выполнить анализ бинарного кода используемых модулей.

В данном случае под анализом бинарного кода подразумевается проверка корректности работы программного обеспечения (ПО) и отсутствия программных ошибок. Существуют различные классификации ошибок, встречающихся в ПО [1, 2]. Далее считаем, что *программная ошибка* — это ошибка реализации ПО, допущенная разработчиками на этапе кодирования. Проявлением программной ошибки является аварийное завершение процесса, связанного с соответствующим ПО.

Сократим анализ бинарного кода. Исследоваться будут только те программные модули (или их части), которые отвечают за обработку входных данных. Этому есть объяснение. Разрабатываемая и используемая методика тестирования бинарного кода [3] хорошо себя зарекомендовала для тестирования ПО, которое обрабатывает сложно-структурированные входные данные. Именно при создании программного кода, обрабатывающего сложно-структурированные данные, со многими внутренними связями (явными и неявными), на качестве кода сказывается человеческий фактор.

Применяют несколько основных подходов, которым отдается предпочтение при решении задачи тестирования бинарного кода [4, 5]:

— обратная инженерия (*reverse engineering*) — применяется с целью получения ПО на языке ассемблера или на языке высокого уровня;
— анализ двоичного кода — предполагает наличие анализирующего приложения, которое читает собранное ПО и просматривает его с применением некоторых эвристических правил;
— тестирование нагрузкой, или стрессовое тестирование — используется набор файлов сценариев, которые посылают ПО разнообразные входные данные различного размера и структуры.

В последнее время получил распространение один из видов стрессового тестирования — фазинг (*fuzzing*) [1]. Его преимущество — возможность быстрого получения результатов.

Тестирование алгоритмов обработки сложно-структурированных данных, основанное на стрессовом тестировании, можно представить в виде последовательности тестов. Каждый тест выполняется за четыре шага.

Шаг 1. С помощью процедур формирования входных данных подготавливаются входные данные, которые будут переданы исследуемому процессу.

Шаг 2. Запускается исследуемый процесс и ему передаются сформированные на первом шаге входные данные.

Шаг 3. Регистрируется состояние исследуемого процесса. Если исследуемый процесс на переданных ему входных данных завершает работу аварийно, то собранная информация о работе процесса поможет разобраться в причинах ошибки.

Шаг 4. На последнем шаге теста, в случае аварийного завершения исследуемого процесса, восстанавливается корректность его последующих запусков для независимого выполнения очередного теста.

Применение стрессового тестирования позволяет обнаружить программные ошибки за короткое время.

Первоначально входные данные для тестирования формировались в статичном режиме, на основе различных эвристических правил и без восстановления алгоритмов работы исследуемого ПО.

Для повышения эффективности тестирования было решено сочетать стрессовое тестирование и динамический анализ на основе трассировки исследуемого процесса. Трассировка помогает сгенерировать входные данные.

При этом возникают две взаимосвязанные задачи. Первая — сопоставление входных данных и результатов трассировки, для чего строится граф потока данных.

Вторая задача — анализ собранной трассы, при этом можно выделить различные подзадачи: обнаружение функций, «схлопывание» циклов, минимизация объемов хранимых данных (поскольку объемы трассы измеряются гигабайтами) и другие. Для решения данной задачи предполагается использовать возможности среды *Ida Pro*. Интеграция со средой *Ida Pro* позволит минимизировать объем трассы и решить ряд подзадач с помощью штатных средств дизассемблера.

Таким образом, выявление программных ошибок в ПО без исходных текстов путем дополнительного тестирования механизмов обработки входных данных позволяет повысить надежность разрабатываемых программных комплексов.

ЛИТЕРАТУРА

1. *Козиол Д., Личфилд Д., Эйтэл Д., и др.* Искусство взлома и защиты системы. СПб.: Питер, 2006. 416 с.
2. *Ховард М., Лебланк Д.* Защищенный код. 2-е изд. М.: Издательско-торговый дом «Русская редакция», 2005. 704 с.
3. *Макаров А. Н.* Метод автоматизированного поиска программных ошибок // Безопасность информационных технологий. Вып. 2. М.: МИФИ, 2008. С. 101–104.
4. *Хогланд Г., Мак-Гроу Г.* Взлом программного обеспечения: анализ и использование кода. М.: Издательский дом «Вильямс», 2005. 400 с.
5. *Eilam E.* Reversing: Secrets of Reverse Engineering. Wiley Publishing, 2005. 589 p.

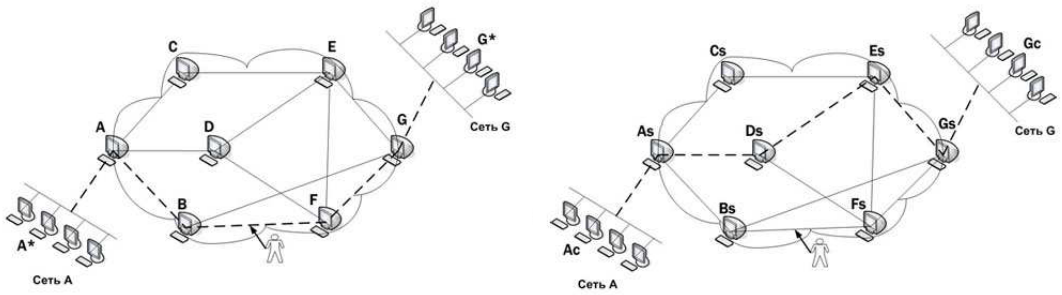
УДК 004.738

МАРШРУТИЗИРУЕМЫЙ СЕРВИС ПЕРЕДАЧИ ДАННЫХ

В. И. Никонов

Настоящая работа продолжает исследование [1], посвященное разработке алгоритмов разделения данных в распределенных сетях. Этот метод выступает в качестве альтернативы снижению вычислительных затрат при использовании шифрования.

Одним из видов активных сетевых атак является класс атак, основанных на снифинге [2]. Приведем пример, в котором злоумышленник, обладая знаниями, что некоторая организация регулярно передает данные из A в G, может довольно точно определить маршрут от A до G в момент времени Δt и осуществить перехват на каком-нибудь из участков следования трафика (см. рис. 1,a).



а *б*

Рис. 1. Работа протоколов маршрутизации между A и G в момент Δt. Вариант возможной атаки на участке $B - F$ (*а*). Изменение маршрута трафика за счет использования доверенных серверов D_S,E_S (*б*)

Разработан маршрутизируемый сервис S_M передачи данных через распределенные сети. S_M — клиент-серверное приложение, позволяющее пользователю передавать данные специфичным маршрутом. Характер маршрута определяется базой критериев S_M. В данной статье приведено описание работы, посвященной исследованию критерия безопасности передачи.

В роли маршрутизаторов для S_M выступает некоторое множество доверенных серверов распределенной сети. Под доверенным сервером будем понимать некоторый многофункциональный сервер распределенной сети, к которому злоумышленник не имеет доступа.

На доверенных серверах A_S, B_S, C_S, D_S, E_S, F_S, $G_S \in F$ устанавливается серверная часть сервиса — S_{MS}, выполняющая автоматическую «интеллектуальную» маршрутизацию трафика. Обозначим через F множество всех доверенных серверов с S_{MS}, через F_i — конкретный доверенный сервер $i \in [1, n]$.

На рис. 1,*б* показано, что использование S_M позволило избежать прохождения трафиком подконтрольного злоумышленнику участка. Данное решение S_M (итоговый маршрут) является вероятностным с вероятностью принятия p_j, $j \in [1, k]$. Здесь k — количество различных маршрутов от A_S до G_S на графе с вершинами A_S, B_S, C_S, D_S, E_S, F_S, G_S и ребрами, определяемыми текущей топологией сети. Расчет значений p_j будет рассмотрен далее.

Напомним, что в процессе передачи с помощью S_M данные проходят через некоторое число доверенных серверов, равное *f*. Выбор каждого следующего сервера происходит динамически и описывается гипергеометрическим распределением $HG(c; a_i, n, c)$. Параметры распределения: n — число всех используемых доверенных серверов; $c = 1$ (в случае использования инструмента мультиплексирования трафика $c > 1$), a_i — число недоступных для F_i серверов из числа всех серверов (определяется динамически). Таким образом, итоговый маршрут трафика от отправителя до получателя при использовании S_M и f доверенных серверов (из n доступных) будет выбран с вероятностью

$$p_j = \binom{n - a_0}{c} \cdot \binom{n - 1 - a_1}{c} \cdot \ldots \cdot \binom{n - f - a_f}{c}, j \in [1, k],$$

где a_i — число недоступных серверов для F_i при выборке F_{i+1} доверенного сервера на $i + 1$ шаге. Оценим вероятность успешной атаки p_A, когда злоумышленник контролирует участок между доверенными серверами F_t и F_{t+1}. При неизвестном про-

странственном расположении F_i считаем атаку успешной, если при работе сервиса S_M передатчики F_t и F_{t+1} были выбраны на i и $i+1$ этапе передачи, $t \in [1, n]$, $i \in [1, f]$:

$$p_A = \frac{2}{n - a_0} \cdot \frac{1}{n - 1 - a_1} + \frac{2}{n - 1 - a_1} \cdot \frac{1}{n - 2 - a_2} + \cdots + \frac{2}{n - (f-1) - a_{f-1}} \cdot \frac{1}{n - f - a_f}.$$

Эта формула легко распространяется на случай подконтрольных злоумышленнику участков между s доверенными серверами F_t, F_{t+1},...,F_{t+s}. Так, при достаточно большом n и достаточно малых a_i и f, причем $n >> f$ и $n >> a_i$, оценка p_A представляется в виде

$$p_A = O\left(\frac{1}{n^2}\right).$$

При использовании мультиплексирования ($c > 1$) задача злоумышленника еще более усложняется. Варианты атак злоумышленника на разнесенный трафик рассматриваются в [3].

ЛИТЕРАТУРА

1. *Ефимов В. И., Файзуллин Р. Т.* Система мультиплексирования разнесенного TCP/IP трафика // Вестник Томского госуниверситета. Приложение. 2005. № 14. С. 115–118.
2. *Avi Kak.* Port Scanning, Vulnerability Scanning, and Packet Sniffing // Computer and Network Security. 2008. № 23. С. 29–38.
3. *Ефимов В. И.* Атака на систему разнесенного TCP/IP трафика на основе анализа корреляции потоков // Информационные технологии моделирования и управления. 2005. № 6(24). С. 859–863.

УДК 519.8

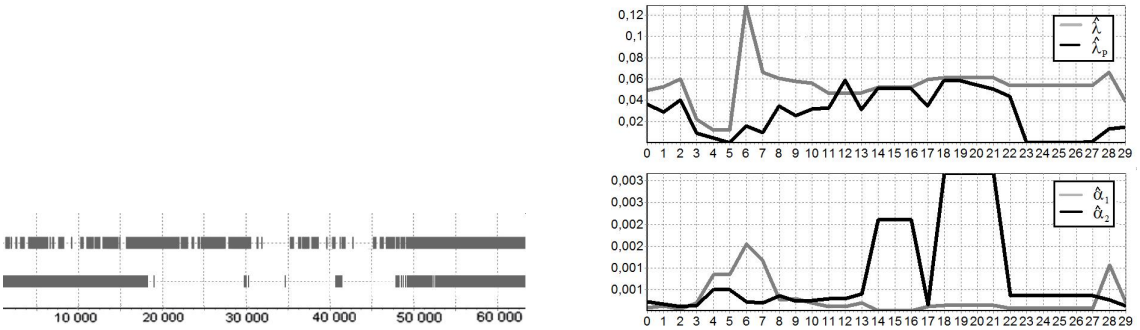
АППРОКСИМАЦИЯ СЕТЕВОГО ТРАФИКА МОДЕЛЬЮ АЛЬТЕРНИРУЮЩЕГО ПОТОКА СОБЫТИЙ

О. В. Ниссенбаум, И. Б. Пахомов

Трафик пользователя в сети, как правило, имеет переменную интенсивность и с той или иной степенью достоверности может быть представлен потоком событий с кусочно-постоянной стохастически изменяемой интенсивностью [1]. Рассмотрим трафик пользователя компьютерной сети с точки зрения соответствия модели асинхронного альтернирующего потока. Такой поток имеет два состояния, в первом из которых наблюдается пуассоновский поток с параметром λ, а во втором события потока отсутствуют. Интервалы, на которых поток находится в первом или втором состоянии, распределены по экспоненциальному закону с параметром α_1 и α_2 соответственно. Сравним полученные результаты с результатами для модели пуассоновского потока с интенсивностью λ_P.

Статистика трафика в виде временных моментов получения пакетов данных была собрана с компьютеров одной локальной сети. Данные сгруппированы по 1 мин, в каждой группе производилась оценка параметров. Использованы оценки параметров асинхронного альтернирующего потока, полученные в [2], и оценка моментов для интенсивности пуассоновского потока [3]. На рис. 1 черными штрихами обозначены моменты поступления пакетов данных за два отрезка времени по 2 мин. На верхней части рисунка (эксп. 1) трафик достаточно равномерен, на нижней (эксп. 2) выделяются периоды высокой интенсивности трафика и периоды «молчания». Оценки параметров

для эксп. 1: $\hat{\lambda} = 0{,}0458$, $\hat{\alpha}_1 = 0{,}0004$, $\hat{\alpha}_2 = 0{,}0011$, $\hat{\lambda}_P = 0{,}0332$. Для эксп. 2: $\hat{\lambda} = 0{,}0522$, $\hat{\alpha}_1 = 0{,}0002$, $\hat{\alpha}_2 = 0{,}0003$, $\hat{\lambda}_P = 0{,}0338$. Оценка $\hat{\lambda}_P$ для обоих случаев близка, с точки зрения простейшего потока эти случаи неразличимы. Оценки параметров альтернирующего потока отличаются существенно, эта модель — более гибкая.



а *б*

Рис. 1. Моменты поступления пакетов данных (2 эксперимента) (*а*). Динамика оценок (*б*)

На рис. 1,*а* представлена динамика оценок параметров трафика пользователя, работавшего с сетевыми сервисами и посещавшего сайты, в течение часа. Средние оценок, представленных на рис. 1,*б*, составили: $\overline{\hat{\lambda}}$=0,0534, $\overline{\hat{\alpha}}_1$=0,0003, $\overline{\hat{\alpha}}_2$=0,0009, $\overline{\hat{\lambda}}_P$=0,0281. Выборочные дисперсии оценок: $D_{\hat{\lambda}}$=0,0003, $D_{\hat{\alpha}_1}$=0,00001, $D_{\hat{\alpha}_1}$=0,00001, $D_{\hat{\lambda}_P}$=0,0004. Анализ результатов позволяет сделать следующие выводы: 1) стабильность оценок говорит о том, что модель альтернирующего потока более адекватно описывает трафик пользователя, чем модель пуассоновского; 2) резкое изменение оценок для альтернирующего потока позволяет использовать их для анализа сетевой активности.

ЛИТЕРАТУРА

1. *Головко Н. И., Каретник В. О., Танин В. Е., Сафонюк И. И.* Исследование моделей систем массового обслуживания в информационных сетях // Сиб. жур. индустр. матем. 2008. Т. XI. № 2(34). С. 50–58.
2. *Васильева Л. А., Горцев А. М.* Оценивание параметров дважды стохастического потока событий в условиях его неполной наблюдаемости // Автоматика и телемеханика. 2002. № 3. С. 179–184.
3. *Вентцель Е. С., Овчаров Л. А.* Теория случайных процессов и ее инженерные приложения. М.: Высш. шк., 2000. 383 с.

УДК 004.412

К ОПРЕДЕЛЕНИЮ СТЕПЕНИ ИНТЕГРИРОВАННОСТИ ПРОГРАММНЫХ ПОДСИСТЕМ

Д. А. Стефанцов

При разработке защищённых систем обработки информации (СОИ) строится модель нарушителя в виде формального описания набора угроз и/или атак и формулируется политика безопасности (ПБ) в виде набора формальных требований [1]. С изменением модели нарушителя появляется необходимость внесения изменений в ПБ.

Примером подобных изменений может служить реализация политики мандатного разграничения доступа SELinux для операционной системы GNU/Linux, ранее обладавшей только политикой дискреционного разграничения доступа [2]. Тесная интеграция программных реализаций СОИ и ПБ является препятствием к внесению изменений в ПБ [3], в связи с чем возникает проблема выбора такого метода интегрирования программной подсистемы ПБ в программную подсистему СОИ, который обеспечивал бы наименьшую степень интегрированности (далее: СИ) первой во вторую. Решение этой проблемы предполагает наличие количественной характеристики СИ одной программной подсистемы в другую.

В данной работе предлагается вариант формального определения СИ произвольной программной подсистемы, называемой подчинённой, в произвольную программную подсистему, называемую главной, и показывается его применение к оценке СИ подсистемы ПБ в файловую подсистему ядра операционной системы Linux.

Это определение удовлетворяет всем критериям метрики программного обеспечения, сформулированным в [4]. В его основе лежит подсчёт количества идентификаторов (имён переменных, функций, констант и т. п.) подчинённой подсистемы, встречаемых в исходном тексте главной подсистемы. Предполагается, что при модификации подчинённой подсистемы на обработку идентификаторов тратятся некоторые (вычислительные, ёмкостные, временные и др.) ресурсы, совокупность которых характеризует сложность этой обработки, и СИ определяется в зависимости от её величины.

Грубой оценкой СИ программных подсистем может служить выражение $M = {=}\alpha \cdot n$, где n — это количество идентификаторов подчинённой подсистемы, встречаемых в исходном тексте главной подсистемы, и α — средняя сложность обработки одного идентификатора.

Для более точного определения СИ рассмотрим сначала простейший случай главной подсистемы — последовательность f некоторых команд s_i без циклов и ветвлений, т. е. $f = (s_1,\ s_2, \ldots,\ s_k)$. Обозначим $n(s_i)$ количество идентификаторов подчинённой подсистемы в команде s_i. Пусть задана функция $\alpha(n)$ — сложность обработки команды, содержащей n идентификаторов подчинённой подсистемы. Тогда более точную оценку СИ подсистем даёт выражение $M(f) = \sum\limits_{i=1}^{k} \alpha(n(s_i))$.

Предполагается, что функция $\alpha(n)$ определена либо на всём множестве целых неотрицательных чисел, либо на множестве $\{0,\ 1,\ \ldots,\ N\}$ для достаточно большого N. Она должна быть также монотонно неубывающей и обладать свойством $\alpha(0) = 0$.

Рассмотрение случаев циклов и составных условных операторов может быть проведено аналогично случаю неполного условного оператора (НУО), поэтому далее предполагаем, что среди команд в программах могут встречаться только НУО и выражения (команды, не содержащие циклов и ветвлений). В свою очередь, НУО может быть рассмотрен как алгоритм $f = (a; s)$, где a — условие, $s = (s_1, \ldots, s_k)$ — последовательность команд, являющаяся телом условного оператора. Полагая идентификаторы подчинённой подсистемы, встречаемые в параметрах алгоритма и в одной из его команд, связанными, определим степень интегрированности $M(f)$ подчинённой подсистемы в реализацию алгоритма f рекурсивно как

$$M(f) \;=\; M(a;\; (s_1,\; \ldots s_k)) = \sum_{i=1}^{k} M(a; s_i),$$

$$M(a;\; s_i) \;=\; \begin{cases} \alpha(n(a) \;+\; n(s_i)),\; \text{если } s_i \text{ — выражение,} \\ M(a,\; a_i;\; (s_{i1},\; \ldots,\; s_{il})) \;= \sum\limits_{j=1}^{l} M(a,\; a_i;\; s_{ij}), \\ \text{если } s_i = (a_i;\; (s_{i1},\; \ldots,\; s_{il})) \text{ — НУО.} \end{cases}$$

Введённая мера СИ была применена к существующим программным системам — к версиям 0.01 и 2.6.24 ядра операционной системы Linux, являющимся соответственно самой первой и одной из самых последних версий ядра Linux. В качестве главной подсистемы была выбрана подсистема работы с файлами с помощью системных вызовов sys_open(), sys_mkdir(), sys_rmdir(), sys_link(), sys_unlink(), а в качестве подчинённой — реализация дискреционной политики безопасности.

Для Linux 0.01 получено значение СИ, равное $\alpha(1) \;+\; 5\alpha(2) + 5\alpha(3)$, а в случае Linux 2.6.24 СИ равна $3\alpha(1) \;+\; 2\alpha(2) \;+\; 3\alpha(3)$. Как было отмечено, функция α — монотонно неубывающая. Если принять $\alpha(n)$ за константу, то оценка для Linux 0.01 станет равной 26α, а для Linux 2.6.24 — равной 16α. Это уменьшение значения СИ является следствием уменьшения как общего количества идентификаторов подчинённой подсистемы в главной, так и тех из них, которые связаны (входят в один оператор).

ЛИТЕРАТУРА

1. *Landwehr C. E.* Formal models for computer security // ACM Comput. Surv. 1981. September. V. 13. No. 3. P. 247–278.
2. Security-Enhanced Linux / USA National Security Agency., Электрон. дан., 2009. Режим доступа: `http://www.nsa.gov/research/selinux/index.shtml`, свободный.
3. Grand Research Challenges in Information Systems / Computing Research Association., Электрон. дан., 2002. Режим доступа: `http://www.cra.org/reports/gc.systems.pdf`, свободный.
4. *Mills E. E.* Metrics in the software engineering curriculum // Ann. Softw. Eng. 1999. V. 6. No. 1–4. P. 181–200.

УДК 681.3.06:62-507

ТРАНСЛЯЦИЯ ОПИСАНИЙ АВТОМАТОВ, ПРЕДСТАВЛЕННЫХ В ФОРМАТЕ MICROSOFT VISIO, В ИСХОДНЫЙ КОД НА ЯЗЫКЕ C

Л. В. Столяров, И. Р. Дединский, А. А. Шалыто

Существует подход к созданию программ для систем логического управления с использованием конечных автоматов, названный в [1] автоматным программированием (АП). В последнее время этот подход быстро развивается [1, 2]. С его помощью удобно реализуются различные системы логического и событийного управления [3]. АП основывается на использовании конечных автоматов (КА) для описания логики работы программ. КА — это конечное множество состояний, в которых он может находиться, и переходов между этими состояниями. Подход основан на создании графов переходов КА (автоматных схем) [4] с последующей их трансляцией в исходный код программы. Автоматная схема является графическим отражением алгоритма в терминах состояний и переходов. Поэтому при разработке программ на основе АП автоматная схема

документирует алгоритм, что чрезвычайно важно для дальнейшего сопровождения программы. Главная особенность АП состоит в том, что программы, построенные на его основе, в отличие от программ, написанных традиционным путем, могут быть достаточно просто формально верифицированы на основе метода Model Checking [5]. Визуализация логики программы позволяет ускорить ее разработку и отладку, а также отделить алгоритмическую (логическую, управляющую) часть от остального кода. В АП, как и в теории автоматов, используются три понятия: входное воздействие, состояние и выходное воздействие. При этом состояния выделяются на этапе проектирования в явном виде. В АП, как и в других подходах к программированию, важна организация взаимодействия средства трансляции с разработчиком автоматной программы и интеграция в системы автоматической сборки (настройка режимов трансляции, способ запуска, сообщения об ошибках трансляции).

Граф переходов КА состоит из нескольких основных элементов: состояния, групповые состояния, описание входных и выходных воздействий, дуги переходов. *Состояние* имеет имя, список других автоматов для запуска, а также список выходных воздействий для этого состояния. *Событие* — один из видов входных воздействий, которые обеспечивают вызов автомата. Оно чаще всего используется в условиях переходов. *Дуга перехода* соединяет два состояния или группы состояний. Дуга помечается *условием*, представленным в виде произвольного логического выражения, в котором, в частности, используются *входные переменные*, а также списком *выходных воздействий*, которые выполняются при переходе. *Входная переменная* — это входное воздействие, представленное переменной, значение которой может быть использовано в условии перехода.

В настоящее время известны инструментальные средства для поддержки АП, такие, как Visio2Switch [6], MetaAuto [7], UniMod [8]. В этих средствах есть как свои достоинства, так и недостатки — диагностика ошибок в графе переходов недостаточна, надежность и качество сгенерированного кода низки (MetaAuto), интеграция в автоматизированные процессы сборки приложений невозможна или чрезвычайно сложна (Visio2Switch).

Целью работы являлось создание инструментального средства для трансляции описаний графов переходов автоматов, представленных в формате MS Visio, в исходный код на языке C и другие формы представления данных (язык XML). Основная задача работы — устранение недостатков, перечисленных выше. В качестве языка реализации выбран язык Microsoft C# для платформы .NET, так как он обеспечивает хорошую поддержку технологии COM, используемой при взаимодействии с MS Visio. Интерфейсы COM-объекта MS Visio позволяют открывать любые файлы формата MS Visio и получать список всех элементов автоматной схемы, описанной в файле. Эта схема состоит из описаний свойств автомата, его элементов и графа переходов. Специализированных шаблонов для описания схемы в стандартной библиотеке MS Visio нет, поэтому одной из задач работы являлась разработка файла с такими шаблонами.

В процессе работы транслятора схема, содержащаяся в исходном документе MS Visio, преобразуется в исходный код для дальнейшего использования в автоматном проекте, а также сохраняется в формате XML. Первым этапом процесса трансляции является чтение данных из исходного документа MS Visio. Для реализации этого этапа был применен подход, основанный на непосредственном взаимодействии с редактором MS Visio при помощи COM-технологии. Следующая стадия — генерация выходных данных (программного кода на языке C). Она происходит с использованием файлов-шаблонов кода из выбираемой пользователем директории. В этих файлах содержатся

специальные метки, которые в процессе трансляции заменяются на соответствующие участки сгенерированного кода. Такой подход позволяет любому пользователю настроить стиль генерируемых файлов на свой вкус, не прикладывая особых усилий.

В работе также было уделено внимание пользовательскому интерфейсу транслятора, так как он является инструментом, который обязан быть удобным. Программа может быть запущена в двух режимах. В первом из них трансляция производится неинтерактивно, и вся информация считывается из командной строки. Этот режим удобен при использовании в средствах автоматической сборки. Во втором режиме интерфейсом программы является диалоговое окно, в котором пользователь может задавать параметры, используя элементы графического интерфейса .NET. Если программа запущена без аргументов командной строки, то она открывается в режиме графического интерфейса.

При разработке проекта также было уделено внимание системе обработки ошибок проектирования автоматной схемы, обнаруженных транслятором в графе переходов. Ошибкой считается некий найденный дефект, без исправления которого разработчиком нельзя продолжить формальную трансляцию. По остальным дефектам выводятся предупреждения, так как они скорее всего являются следствиями высокоуровневых логических ошибок в исходной схеме. Во время трансляции все ошибки и предупреждения выводятся в консоль транслятора.

В результате работы создано инструментальное средство для трансляции графов переходов автоматов, представленных в формате Microsoft Visio, в исходный код на языке C. Средство обеспечивает сохранение исходных данных в XML-файл, диагностику ошибок в исходных данных, интеграцию в автоматические процессы сборки приложений (например, make-файлы, MS Visual Studio build system).

ЛИТЕРАТУРА

1. *Шалыто А. А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. 628 с.
2. *Шалыто А. А.* Логическое управление. Методы аппаратной и программной реализации. СПб.: Наука, 2000. 780 с.
3. *Шалыто А. А., Туккель Н. И.* Реализация автоматов при программировании событийных систем // Программист. 2004. № 2. С. 74–80.
4. *Поликарпова Н. И., Шалыто А. А.* Автоматное программирование. СПб.: Питер, 2009. 176 с.
5. *Гуров В. С., Шалыто А. А., Яминов Б. Р.* Технология верификации автоматных программ без их трансформации во входной язык верификатора // Материалы Междунар. науч.-технич. конф. «Многопроцессорные вычислительные и управляющие системы (МВУС-2007)». Таганрог: НИИ МВС, 2007. Т. 1. С. 198–203.
6. `http://is.ifmo.ru/progeny/visio2switch` — *Головешин А.* Конвертор Visio2Switch. 2002.
7. `http://is.ifmo.ru/projects/metaauto` — *Канжелев С. Ю., Шалыто А. А.* Преобразование графов переходов, представленных в формате MS Visio, в исходные коды программ для различных языков программирования (инструментальное средство MetaAuto). 2005.
8. *Гуров В. С.* Технология проектирования и разработки объектно-ориентированных программ с явным выделением состояний (метод, инструментальное средство, верификация): Дис. ... канд. техн. наук. СПбГУ ИТМО, 2008.