УДК 681.518.5

Informatics and Mathematical Methods in Simulation
Vol. 3 (2013), No. 3, pp. 201-214

# СТРУКТУРНОЕ ТЕСТИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ С ИСПОЛЬЗОВАНИЕМ ЭЛЕМЕНТОВ КОМПЬЮТЕРНОЙ АЛГЕБРЫ

**Л.С. Ломакина, А.Н. Вигура**

Нижегородский государственный технический университет им. Р.Е. Алексеева,
ул. Минина, 24, Нижний Новгород, 603950, Российская Федерация; e-mail: llomakina@list.ru

Предложен метод структурного тестирования программных систем, основанный на алгебраической модели программы и динамическом символьном выполнении на уровне машинного кода и позволяющий автоматизировать модульное тестирование программ, реализованных на компилируемых языках программирования.

**Ключевые слова**: программные системы, тестирование, генерация тестовых данных, алгебраическая модель, символьное выполнение

## Введение

В настоящее время актуальной является проблема контроля и обеспечения надежности программных систем. Как известно, надежность любой вычислительной системы обусловлена двумя факторами – надежностью аппаратного обеспечения и надежностью программ, управляющих аппаратным обеспечением. При этом, хотя для диагностирования аппаратных средств разработаны эффективные методы, позволяющие выявлять дефекты и формально обосновывать их отсутствие, аналогичная задача для программ в полной мере не решена. Постоянное усложнение используемых на практике в различных отраслях промышленности программных систем привело как в области верификации и тестирования, так и в области синтеза надежных и контролепригодных программ.

Основной проблемой тестирования является принципиальная невозможность нахождения всех дефектов в программном продукте. Как следствие, при тестировании требуется решение следующих взаимосвязанных задач:

1) определение полноты тестирования;

2) выбор множества тестовых воздействий с целью обеспечения заданной полноты тестирования;

3) проверка корректности полученных выходных данных.

При этом несомненно актуальной является задача снижения общих затрат на тестирование и поддержку программного продукта. Одним из путей к решению данной задачи является обнаружение дефектов на ранних стадиях жизненного цикла программной системы, реализуемое с помощью автоматизации тестирования на этапе разработки и мониторинга метрик качества в рамках систем непрерывной интеграции.

Тестирование на этапе разработки представляет собой структурное тестирование, в рамках которого тесты выбираются исходя из граф-модели программной системы, а в качестве метрики качества используется тестовое покрытие кода. В силу этого далее в настоящей работе рассматривается структурное тестирование. Отметим, что на практике полной автоматизации поддаются только модульные тесты, в то время как интеграционное дизайн-тестирование выполняется вручную разработчиками. При этом

тестовые воздействия, выбранные для модульного тестирования, зачастую тоже выбираются разработчиками вручную. Это приводит к существенным затратам времени на тестирование и, соответственно, к повышению затрат на разработку и стабилизацию программного продукта. Таким образом, актуальной является задача разработки методов и реализующих их программных средств, ориентированных на автоматическую оценку полноты тестирования и автоматизированный выбор тестовых воздействий.

Современные подходы (реализованные, в частности, в системах CUTE и CREST [1]) к автоматизации тестирования ориентированы на генерацию тестовых данных и основаны на динамическом символьном выполнении, при котором одновременно производится выполнение программы «в числах» и «в символах», и численные значения данных используются в том случае, когда символьные не могут быть определены. Отметим, что существующие системы не позволяют обрабатывать в символьном виде операции разыменования указателей, что ограничивает их применимость, а также не поддерживают обработку взаимодействия инструментируемого и неинструментируемого кода (например, стандартных подпрограмм). Настоящая работа нацелена на преодоление указанных недостатков. В работе предложен метод модульного тестирования программных систем, основанный на динамическом символьном выполнении на уровне машинного кода и алгебраической модели программы, и позволяющий учитывать в модели операции над указателями, а также обрабатывать вызовы стандартных подпрограмм. На основе предложенного метода разработана программная система, реализующая контролируемое выполнение программы с помощью доработанной виртуальной машины Valgrind [2].

## Основная часть

Структурное тестирование предполагает выбор тестовых воздействий на основе граф-модели программы таким образом, чтобы достичь желаемого значения тестового покрытия (например, покрытия дуг управляющего графа). Общая схема структурного тестирования приведена на рис. 1.

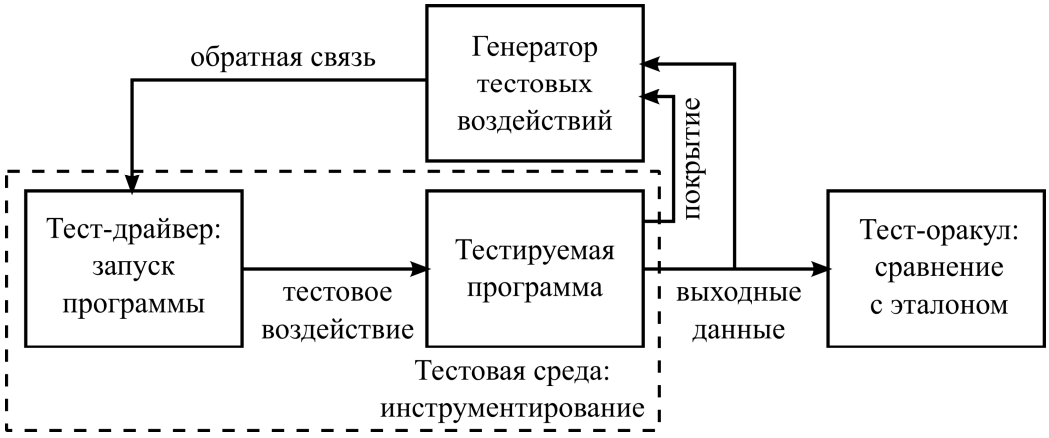


**Рис. 1.** Схема структурного тестирования

Отметим, что как генерация тестовых данных, так и определение тестового покрытия требуют модификации тестируемой программной системы, поскольку в общем случае факты выполнения тех или иных блоков кода системы не являются

наблюдаемыми – данная проблема решается вставкой инструментального кода, выполняющего трассировку программы. Таким образом, чтобы реализовать динамическую генерацию тестовых воздействий, требуется решить следующие задачи:

1) Раскрытие структуры программной системы - построение ее модели структуры с целью дальнейшего выбора тестовых путей.

2) Модификация тестируемой программной системы – вставка инструментального кода, отслеживающего ход выполнения программы и операции с программными переменными.

3) Символьное выполнение программы – определение условий прохождения тестовых путей в зависимости от входных данных в символьном виде.

4) Выбор тестовых воздействий на основе полученных условий прохождения путей.

**Базовая модель программы**

Нашей задачей является определение тестового воздействия для выбранного тестового пути, для чего нужно определить условие прохождения этого пути в зависимости от входных переменных в символьном виде (в виде алгебраического выражения). С целью решения данной задачи будем использовать алгебраическую модель программы в виде множества связанных по управлению блоков машинных инструкций [3], как показано на рис. 2.

Предлагаемая модель программы основана на понятии алгебраического выражения. Алгебраическое выражение [3] представляет собой один или несколько термов, соединенных между собой знаками операций и знаками последовательности операций (скобками). Множество выражений определим следующим образом:

$$Expr = Term \cup \left\{(o,a) : o \in Op, a \in Expr^{arity(o)}\right\},$$

где

*Expr* — множество выражений;

*Term* — множество термов;

*Op* — множество возможных операций;

$arity: Op \to N$ — отображение, определяющее для каждой операции количество аргументов.

На уровне машинных инструкций ЭВМ оперирует рациональными числами – константами и располагающимися по определенным адресам в памяти переменных. Исходя из этого, определим множество термов следующим образом: $Term = Q \cup Addr \times Types$, где $Addr \subset N \cup \{0\}$ – множество допустимых адресов, *Types* — множество примитивных типов данных

Инструкции представляют собой элементы следующего множества:

$$Ins = \left\{(addr, expr, bt, bf)\right\} \subseteq Addr \times Expr \times Addr \times Addr,$$

где

*Addr* — множество допустимых адресов памяти, *addr* — адрес инструкции, *expr* — выполняемое выражение;

*bt* — адрес следующей инструкции в случае истинности значения *e*;

*bf* — адрес следующей инструкции в случае ложности значения *e*.

Для ветвлений $bt \neq bf$, для прочих инструкций $bt = bf$. Множество *Addr* определяет допустимые адреса программных инструкций и переменных. Инструкции

сгруппированы в базисные блоки $bb_i$, составляющие множество $BB \subseteq Ins^* = \bigcup_{i=0}^{\infty} Ins^i$, каждый из которых представляет собой неделимую последовательность инструкций, в которой выделена только одна входная инструкция и существует не менее одной выходной инструкции, передающей управление на начало другого базисного блока. Выделив во всем множестве базисных блоков входной и выходной блоки, сформируем модель программы следующим образом. Обозначим адрес входного базисного блока через *entry*, а адрес выходного базисного блока через *exit*. Программу будем представлять в виде тройки $P = (BB, entry, exit)$.

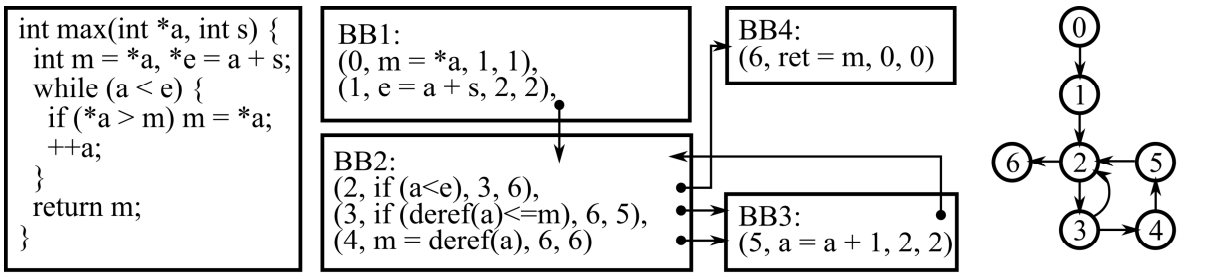


**Рис. 2.** Базовая модель и управляющий граф программы

Построение модели будем осуществлять с помощью анализа машинного кода программы, что по сравнению с методами анализа программ, работающими с исходными текстами, дает следующие преимущества:

- отсутствие необходимости выполнения трудоемкого синтаксического анализа исходных текстов и учета индивидуальных особенностей различных компиляторов;
- отсутствие проблемы обработки взаимодействия инструментируемого и неинструментируемого кода в том случае, когда не все исходные тексты доступны (в частности, при вызовах подпрограмм стандартной библиотеки).

## Диагностическая модель

Для определения полноты тестирования и выбора тестовых воздействий будем использовать взвешенный управляющий граф программы (УГП) с детализацией до инструкций. Управляющий граф программы является одной из основных диагностических моделей в структурном тестировании программных систем. В УГП вершины соответствуют элементам программы (операторам, линейным участкам, базисным блокам и т.п.), а дуги — передачам управления между этими операторами. УГП имеет строго одну входную и одну выходную вершину.

Таким образом, будем задавать управляющий граф кортежем $G = (V, E, s, e, w)$, $s \in V$, $e \in V$, где $V \subseteq Insn$ — множество вершин, $E \in V \times V$ — множество дуг, $s$ — входная вершина, $e$ — выходная вершина, $w$ — отображение, ставящее в соответствие вершинам и дугам графа их вес. Путь $P$ в графе может быть представлен как упорядоченная последовательность вершин: $P = (v_1, v_2, .., v_i, .., v_m)$, $P \in V^*$. Для каждой вершины и дуги управляющего графа определим вес $w(v)$, представляющий собой число прохождений по данной вершине или дуге в ходе тестирования программы: $w : V \cup E \to N \cup \{0\}$. При запуске программы с подачей на вход некоторого тестового

воздействия в управляющем графе проходится некоторый путь (трасса), при этом веса проходимых дуг увеличиваются на 1 при каждом прохождении. Таким образом, на основе данного графа может быть определено значение покрытия ветвей:

$$C_b = \frac{\left|\{e : e \in E \wedge w(e) > 0\}\right|}{|E|}.$$

Данное значение будем использовать в качестве критерия полноты тестирования (целью является достижение значения $C_b = 1$).

Управляющий граф программы может быть легко построен по базовой модели программы, поскольку последняя определяет как множество инструкций, так и связи по управлению между ними: в качестве множества вершин графа выбирается множество инструкций всех базисных блоков программы, а множество дуг определяется исходя из связей по управлению между инструкциями.

**Общая схема подхода**

В рамках предлагаемого подхода выбор последующих тестовых воздействий будем осуществлять на основе фактических путей и символьных зависимостей переменных, полученных при предыдущих тестовых запусках. С целью получения этих сведений каждый тестовый запуск производится с помощью динамического символьного выполнения, при этом первый тестовый запуск производится с подачей на вход программы произвольного тестового воздействия. При тестировании выполняется следующая последовательность действий:

1) Первый запуск программы осуществляется с подачей на вход тестового воздействия, сгенерированного случайным образом. Входные данные помечаются в исходном тексте тест-драйвера программистом.

2) Перед каждым выполнением программы осуществляется построение алгебраической модели программы и вставка инструментального кода с использованием виртуальной машины Valgrind:

- Valgrind осуществляет декомпиляцию тестируемой программы и ее разбиение на базисные блоки. Каждый блок разбирается с целью определения характера выполняемых в нем инструкций. На основании этих данных осуществляется построение модели программы;
- в каждый блок добавляется инструментальный код отслеживания операций с памятью и пути передачи управления в программе;
- осуществляется компиляция модифицированных базисных блоков в машинный код целевой ЭВМ;
- в переменные, помеченные программистом как входные, записываются значения из текущего тестового воздействия;
- модифицированный код тестируемой программы выполняется центральным процессором ЭВМ непосредственно. При этом добавленный инструментальный код выполняет сбор статистики по тестовому покрытию и осуществляет динамическое символьное выполнение, сохраняя конкретные и символьные значения переменных и условий прохождения ветвлений.

3) После тестового запуска становится известным путь, пройденный в программе, и условие его прохождения в символьном виде. На основании этих данных выбирается следующий тестовый путь и соответствующее ему тестовое воздействие, после чего осуществляется переход к шагу 2. Если достигнуто требуемое значение тестового покрытия, тестирование может быть завершено.

**Символьное выполнение**

При каждом тестовом запуске производится динамическое символьное выполнение (рис. 2) от точки входа *entry* до точки выхода *exit*, при котором для каждой переменной *x* в памяти сохраняются следующие значения:

1) конкретные (численные) значения, определяемые отображением

$$val: Addr \times Types \to Q,$$

ставящее в соответствие каждой переменной, располагающейся по данному адресу и имеющей заданный тип, ее численное значение (которое будем в дальнейшем называть конкретным значением);

2) символьные значения, определяемые отображением

$$sym: Addr \times Types \to Expr \cup \{\lambda\},$$

ставящее в соответствие каждой переменной, располагающейся по данному адресу и имеющей заданный тип, ее символьное значение;

3) входные переменные, определяемые отображением

$$input: Addr \times Types \to Terms \cup \{\lambda\}.$$

Входные переменные непосредственно доступны для изменения в процессе тестирования.

Сохранение для каждой программной переменной одновременно конкретного и символьного значения позволяет в том числе определять символьные значения целей указательных переменных при их разыменовании.

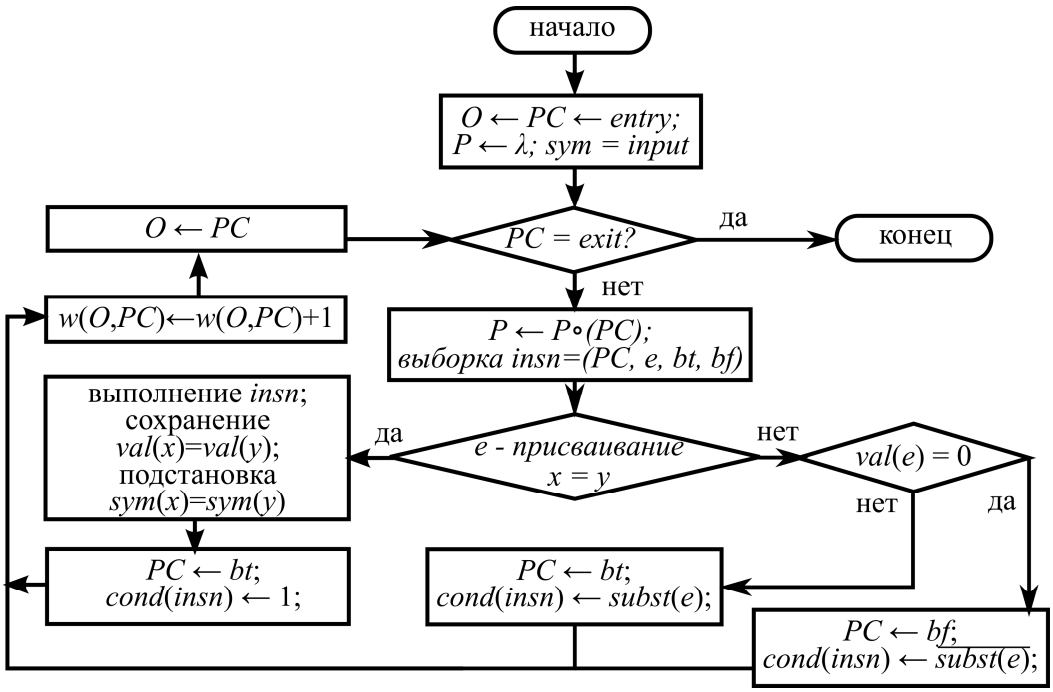


**Рис. 3.** Алгоритм динамического символьного выполнения

Для удобства введено обозначение пустого кортежа $\lambda$, которое будем использовать для записи отсутствия объекта по определенному адресу памяти.

Суть символьного выполнения заключается в подстановке в выражение каждой выполняемой инструкции символьных значений ее операндов. Подставляя эти значения в условия прохождения ветвлений, получаем необходимые выражения для выбора тестовых воздействий. Результатом является пройденный тестовый путь $P=(v_0,...,v_k)$, веса дуг управляющего графа $w(v_i,v_{i+1})$ и условия прохождения вершин управляющего графа $cond(v_i)$, принадлежащих тестовому пути. Роль термов в выражениях условий играют константы и входные переменные *input* – таким образом, решая систему условий, получаем соответствующее ей тестовое воздействие.

На рис. 4 наглядно изображено символьное выполнение двух последовательно расположенных машинных инструкций — инструкции присваивания и инструкции ветвления.

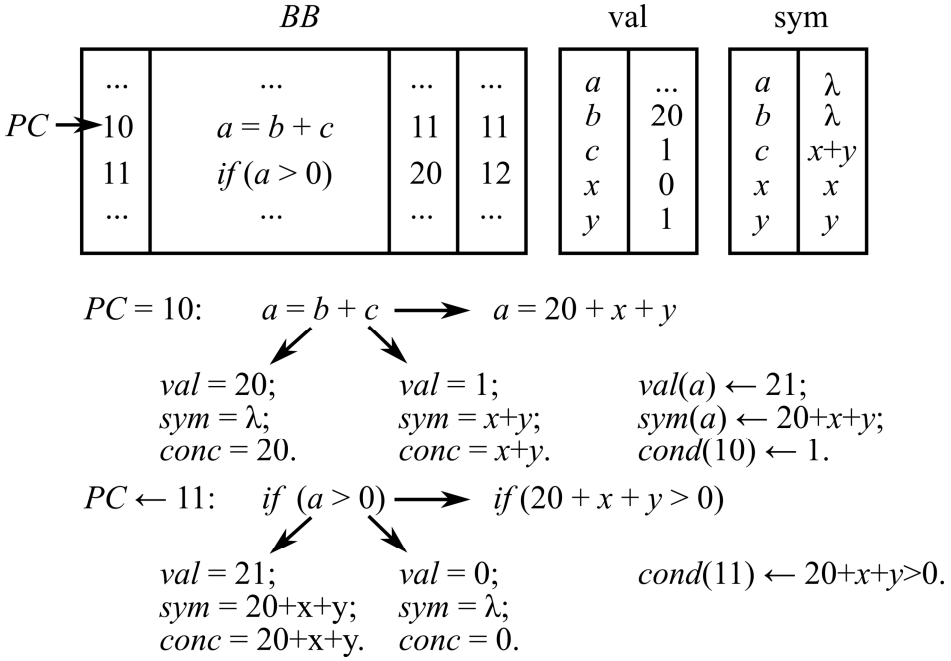


**Рис. 4.** Пример динамического символьного выполнения

**Выбор тестовых путей**

По завершении символьного выполнения на основе полученных данных о покрытии и условии прохождения текущего пути осуществляется выбор следующего тестового пути. Предположим, что при некотором тестовом запуске был пройдет тестовый путь $P=(v_1,v_2,..,v_n)$, и были определены условия прохождения его вершин $cond(v_i)$.

Чтобы определить тестовое воздействие, приводящее к прохождению некоторого альтернативного тестового пути, нам нужно:

1) Найти непокрытую дугу $e=(v_k,v_k^*)$ в управляющем графе, начальная вершина которой $v_k$ принадлежит пути $P=(v_0,v_1,...,v_i,v_{i+1},..,v_k)$. Поскольку все дуги рассматриваемого пути были покрыты при предыдущем тестовом запуске, $v_k^*$ не может принадлежать $P$ — т.е. дуга $e$ может начинаться только в инструкции ветвления.

2) Найти нарушающее условие $c$ в виде конъюнкции условий прохождения вершин пути $P$ от начальной до $v_k$ включительно и инверсии условия вершины $v_{k+1}$: $c = \overline{cond(v_{i+1})} \bigwedge_{j=0}^{i} cond(v_i)$. Инверсия последнего условия требуется, чтобы управление в инструкции ветвления $v_k$ передавалось по альтернативной дуге $e$.

3) Найти тестовое воздействие, решив нарушающее условие.

После каждого тестового запуска просматриваются все пройденные ранее пути на предмет наличия непокрытых дуг, исходящих из принадлежащих им вершин. При отсутствии таких дуг тестирование может быть завершено (поскольку достигнуто полное покрытие ветвей). Блок-схема алгоритма выбора тестовых путей и тестовых воздействий приведена на рис. 5.

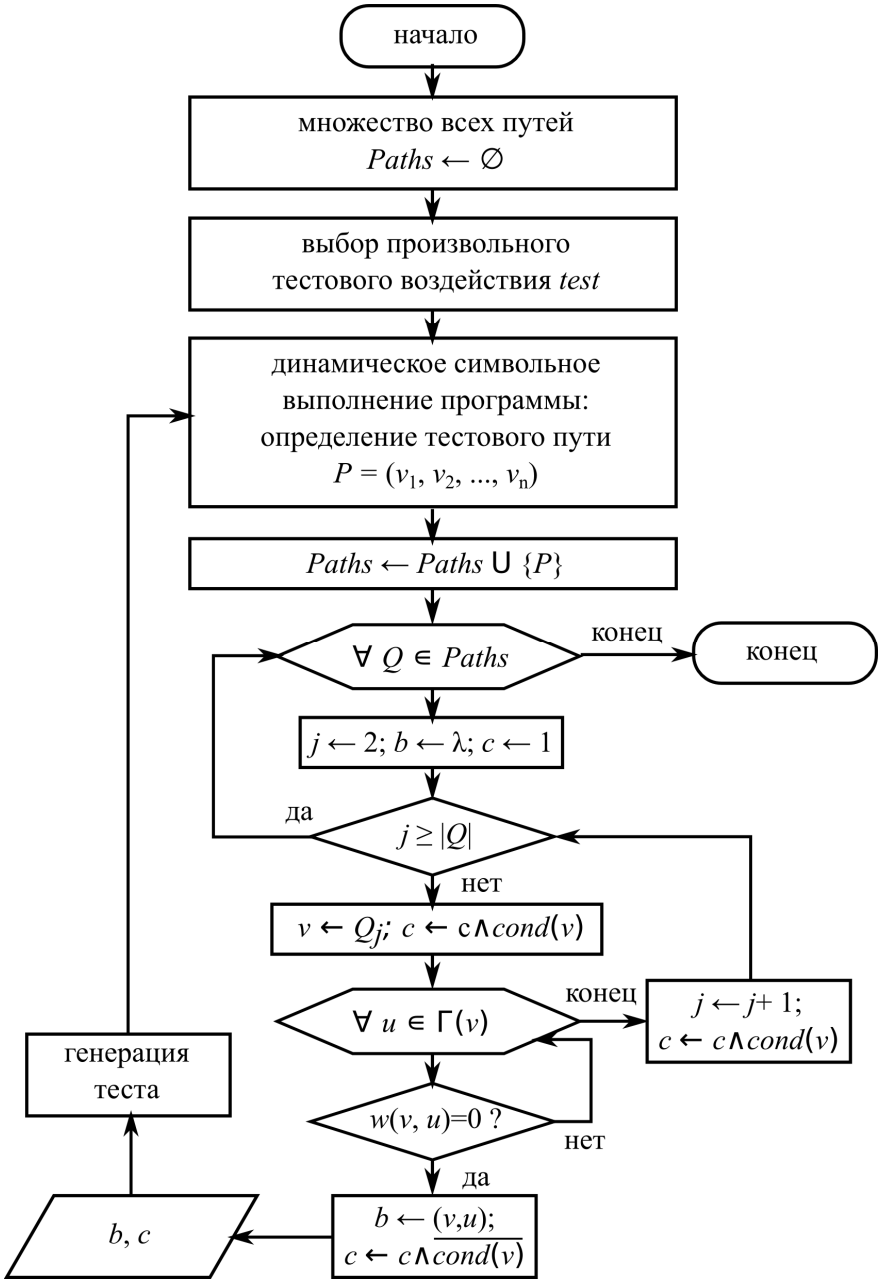


**Рис. 5.** Алгоритм выбора тестовых путей и тестовых воздействий

Преимуществом представленного подхода по сравнению с существующими является как учет операций над указателями в базовой модели и символьном выполнении, так и анализ программы на уровне машинного кода, позволяющий избежать синтаксического анализа и решить проблему обработки вызовов сторонних подпрограмм.

## Программная реализация

На основе предложенных в настоящей работе моделей и алгоритмов была разработана программная система автоматизации тестирования, предназначенная для тестирования программ, реализованных на компилируемых языках программирования.

Система предоставляет следующие функциональные возможности:

1) Генерация тестовых воздействий на основе динамического символьного выполнения (с целью автоматизации модульного тестирования ПО).

2) Подсчет покрытия кода (покрытие строк и ветвей) без необходимости сборки специальных исполняемых образов.

3) Визуализация покрытия кода в динамике без необходимости перезапуска тестируемой программы.

4) Сохранение дампов трасс выполнения программы с целью отложенного анализа и локализации дефектов.

5) Независимость от используемого языка программирования, поскольку отслеживание выполнения программы производится на уровне отдельных инструкций машинного кода.

6) Поддержка различных аппаратных архитектур ЭВМ. Система поддерживает все архитектуры, для которых реализована трансляция кода в Valgrind (x86, AMD64, POWER, ARM).

Система состоит из трех модулей – инструментального модуля, модуля анализа и тест-драйвера, связанных между собой разделяемой памятью (рис. 6). Инструментальный модуль основан на свободно распространяемой утилите Valgrind, представляющей собой виртуальную машину на основе динамической рекомпиляции. Инструментальный модуль отслеживает адреса и характер выполняемых инструкций и операций с памятью, и отправляет трассу в модуль анализа, вставляя для этого в тестируемую программу необходимый код на этапе выполнения. Задачей модуля анализа является как оценка тестового покрытия кода, так и выбор тестовых путей и определение тестовых воздействий на основе результатов символьного выполнения с последующей передачей найденных значений тест-драйверу.

Данная система использовалась в ходе вычислительных экспериментов, результаты которых приведены ниже.

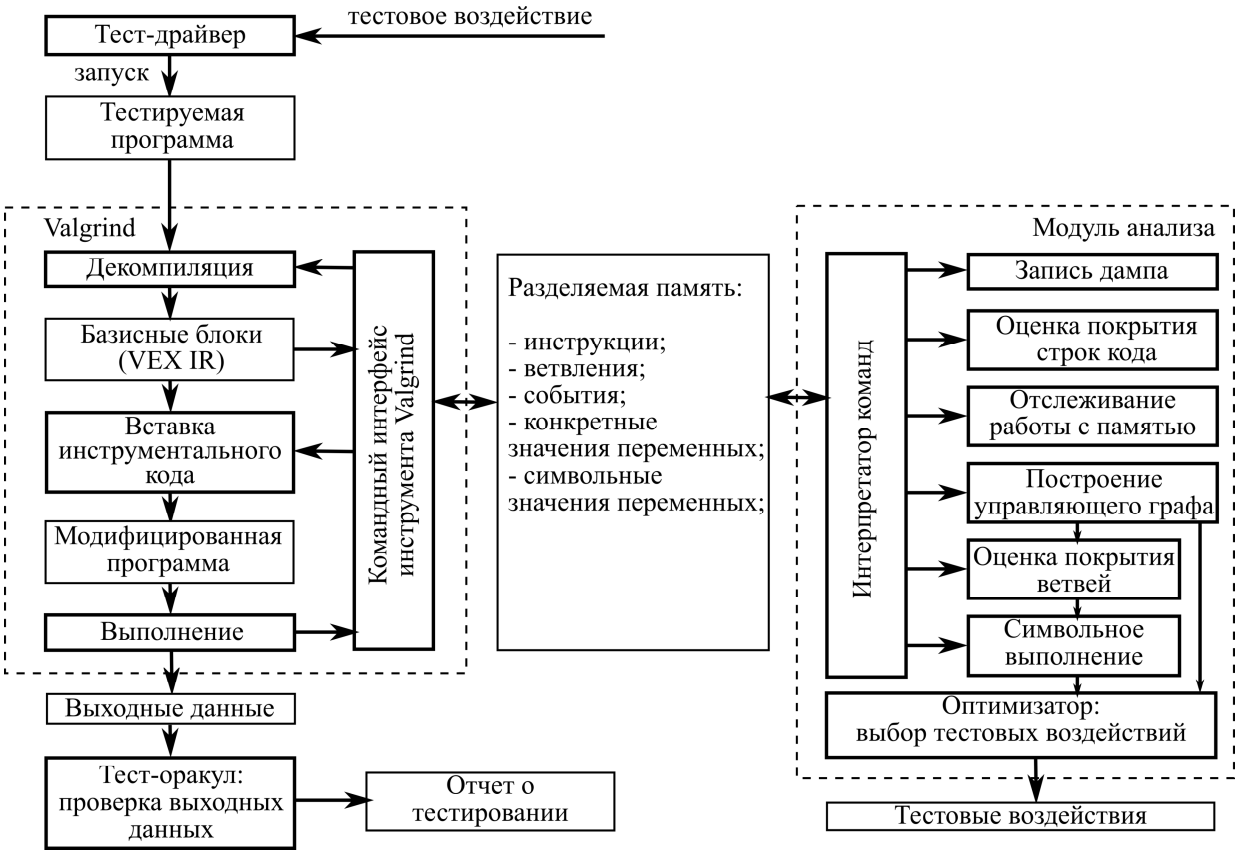


**Рис. 6.** Архитектура системы

### Эксперимент 1. Модульное тестирование

В ходе экспериментов проводилось сравнение разработанной системы с открытой системой CREST [1], ориентированной на тестирование программ на языке C, и случайным тестированием с использованием следующих тестовых программ:

- сортировка методом выбора главного элемента (`sort`);
- сортировка указателей методом выбора главного элемента (`psort`);
- модифицированные варианты с добавленным ветвлением вида `if(a[i]==13)`;
- вычисление частичной суммы числового ряда (`series`);
- быстрая сортировка, реализованная вручную (`qsort`);
- быстрая сортировка из стандартной библиотеки C (`qsort_std`) – выполнялось тестирование обратного вызова функции сравнения;
- вставка и последующее удаление элементов в черно-красное дерево (`rbtree`) из реализации `std::set` в GNU libstdc++.

В процессе тестирования определялось достигнутое покрытие ветвей. Результаты (достигнутое покрытие ветвей $C_b$ и число тестовых воздействий $N$, при котором достигнуто указанное покрытие) приведены в таблице 1.

Интерпретация результатов следующая. Примеры `sort` и `psort` оказались достаточно простыми для достижения полного покрытия кода для всех трех методов. В программах `sort*` и `psort*` система CREST не смогла найти добавленное ветвление из-за отсутствия поддержки операций с указателями. В примере `series` достичь полного покрытия удалось только с помощью символьного выполнения из-за строгого условия входа в подпрограмму, которое при случайном тестировании никогда не выполнялось, при этом разработанная система достигла полного покрытия ветвей меньшим числом тестов. Аналогичная ситуация наблюдается в примере `qsort` (сложные условия

ветвлений не могут быть проверены полностью случайным тестированием). Примеры `qsort` и `rbtree` система CREST выполнить не смогла, разработанная же в настоящей работе система показывает в целом лучшие результаты, чем случайное тестирование.

**Таблица 1.**

Результаты сравнительных экспериментов

| Тест | случайное тестирование | | CREST | | разработанная система | |
|---|---|---|---|---|---|---|
| | $N$ | $C_b$ | $N$ | $C_b$ | $N$ | $C_b$ |
| `sort` | 4 | 1 | 4 | 1 | 4 | 1 |
| `psort` | 4 | 1 | 4 | 1 | 4 | 1 |
| `sort*` | 4 | 0,93 | 9 | 0,93 | 4 | 1 |
| `psort*` | 4 | 0,94 | 9 | 0,94 | 4 | 1 |
| `series` | - | 0 | 8 | 1 | 6 | 1 |
| `qsort` | 13 | 0,94 | 8 | 1 | 5 | 1 |
| `qsort_std` | 3 | 1 | - | 0 | 3 | 1 |
| `rbtree` | 13 | 1 | - | 0 | 6 | 1 |

**Эксперимент 2. Интеграционное дизайн-тестирование**

В качестве эксперимента производилось тестирование одного из компонентов самой системы автоматизации тестирования – граничного решателя, основанного на генетическом алгоритме, а также его графического интерфейса пользователя. Компонент написан на C++/Qt4, состоит из примерно 20 тысяч строк исходного текста (с учетом транслятора символьных условий, базовых библиотек и графического интерфейса), графический интерфейс предоставляет следующую функциональность:

- ввод и компиляцию в машинный код целевой функции в виде алгебраического выражения;
- минимизацию значения целевой функции с помощью генетического алгоритма с использованием распараллеливания;
- визуализацию результатов оптимизации и графиков изменения целевой функции в зависимости от итерации.

Тестирование производилось в интерактивном режиме, в качестве объекта тестирования было выбрано множество классов элементов пользовательского интерфейса, отвечающих за ввод и вывод данных. В процессе тестирования система в динамике обновляет тестовое покрытие по подпрограммам и методам классов и выдает подсказки по выбору тестов.

При тестировании было использовано 24 тестовых воздействия, 10 дуг управляющего графа были помечены как нереализуемые (ветви обработки ошибок в операторах ASSERT, не воспроизводящиеся на практике). С учетом них было достигнуто полное покрытие ветвей тестируемых участков кода.

**Эксперимент 3. Тестирование производительности**

Немаловажным фактором, ограничивающим применимость той или иной системы динамического анализа программ, является влияние инструментирования на временные соотношения работы программы. Под производительностью далее будем понимать производительность среды исполнения программы (аппаратного обеспечения и виртуальной машины) в виде количества машинных инструкций, исполняемых в

среднем в единицу времени. Очевидно, что определенная таким образом производительность зависит от характера выполняемых инструкций, поэтому можно использовать ее только с целью сравнения различных сред исполнения при выполнении одной и той же программы. В табл. 2 приведены результаты синтетического теста Dhrystone [4], выполненного с использованием различных вариантов инструментирования.

Результаты синтетического теста наглядно показывают превосходство в плане производительности инструментирования на уровне исходных текстов (GCC/gcov). Динамическая рекомпиляция в Valgrind приводит к падению производительности в 4-5 раз даже без вставки инструментального кода — данные накладные расходы являются платой за гибкость динамических методов инструментирования. Для сравнения приведена производительность, достигнутая встроенным инструментом Valgrind memcheck.

Наибольшее влияние на производительность оказывает отслеживание операций с памятью. По сравнению с накладными расходами от этого инструментального кода влияние символьного выполнения практически не заметно. Оптимальная расстановка точек инструментирования здесь представляет собой несомненно важную и интересную задачу и является одним из направлений дальнейшего исследования.

**Таблица 1.**

Результаты синтетического теста

| Тип запуска | Производительность, DMIPS | Время выполнения одной итерации, мкс |
|---|---|---|
| Без инструментирования | 6587 | 0.09 |
| GCC/gcov | 5591 | 0.10 |
| Valgrind (только рекомпиляция) | 1504 | 0.39 |
| Valgrind (memcheck) | 166 | 3.43 |
| Покрытие ветвей | 544 | 1.05 |
| Запись дампа | 547,2 | 1.04 |
| Отслеживание памяти (помеченные участки) | 45,5 | 12.6 |
| Отслеживание памяти (все операции) | 11,8 | 48.0 |
| Динамическое символьное выполнение | 11,7 | 48.6 |

Полученные результаты естественным образом ограничивают применимость представленного подхода модульным тестированием, при котором производительность обычно не является критичной. С другой стороны, оценка покрытий в динамике без отслеживания операций с памятью полезна при выполнении ручного дизайн-тестирования сложной программной системы.

## Выводы

Результаты вычислительных экспериментов показывают, что предложенный метод модульного тестирования в ряде случаев позволяет достичь большего тестового покрытия по сравнению с известными методами. Достигается данный результат в основном за счет динамического анализа программ на уровне машинных кодов и отслеживания операций с указателями. Разработанная авторами программная система с

успехом может применяться как для автоматизации модульного тестирования, так и для облегчения ручного тестирования (в частности, интеграционного тестирования) программных систем на этапе разработки. С другой стороны, динамический анализ программ оказывает сильное влияние на производительность, что ограничивает сферу его применения, в частности, делая символьное выполнение неприменимым на практике для тестирования программной системы в целом. Тем не менее, приведенные в данном разделе проблемы являются техническими и поддаются решению. Таким образом, настоящая работа имеет большой потенциал для дальнейшего исследования – в частности, могут быть сформулированы следующие задачи:

- задача оптимальной расстановки точек инструментирования;
- задача обнаружения и удаления ненужного инструментального кода;
- задача неинтрузивного вычисления тестового покрытия кода.

Решение данных задач в перспективе позволит применять предложенные методы с целью проведения тестирования безопасности и обнаружения уязвимостей в программных системах на уровне интеграционного и системного тестирования.

## Список литературы


1. Burnim, J. Heuristics for Scalable Dynamic Test Generation / J. Burnim, K. Sen // Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering. — IEEE Computer Society Press, 2008. — PP. 443–446.
2. Nethercote, N. Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation / N. Nethercote, J. Seward // Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation. — ACM, New York, 2007. — PP. 89–100.
3. Ломакина, Л.С. Модели и методы тестирования программных систем на основе алгебраического подхода / Л.С. Ломакина, А.Н. Вигура // Системы управления и информационные технологии. — Воронеж, 2013. — Том 52, № 2.1. — С. 157–161.
4. Weicker, R.P. Dhrystone: a synthetic systems programming benchmark / R.P. Weicker // Communications of the ACM. — 1984. — Vol. 27, Iss. 10. — PP. 1013–1030.

## СТРУКТУРНЕ ТЕСТУВАННЯ ПРОГРАМНИХ СИСТЕМ З ВИКОРИСТАННЯМ ЕЛЕМЕНТІВ КОМП'ЮТЕРНОЇ АЛГЕБРИ

Л.С. Ломакіна, А.М. Вігура

Нижегородський державний технічний університет ім. Р.Є. Алексєєв,
вул. Мініна, 24, Нижній Новгород, 603950, Російська Федерація; e-mail: llomakina@list.ru

Запропоновано метод структурного тестування програмних систем, заснований на алгебраїчної моделі програми і динамічному символьному виконанні на рівні машинного коду і дозволяє автоматизувати модульне тестування програм, реалізованих на компільованих мовах програмування.
**Ключові слова:** програмні системи, тестування, генерація тестових даних, алгебраїчна модель, символьне виконання

## STRUCTURAL TESTING OF SOFTWARE SYSTEMS USING ELEMENTS OF COMPUTER ALGEBRA

Lyubov S. Lomakina, Anton N. Vigura

Nizhny Novgorod State Technical University n.a. R.E. Alekseev,
24 Minina str., Nizhny Novgorod, 603950, Russian Federation; llomakina@list.ru

A method of software system structural testing based on the program algebraic model and machine code-level dynamic symbolic execution is proposed. The method allows for unit testing automation for programs written in compiled languages.
**Keywords:** software systems, testing, test data generation, algebraic model, symbolic execution