С. К. Черноножкин

Методы тестирования

Учебное пособие

Новосибирск
2004

ББК В185я73-1
УДК 519.6
Ч494

Черноножкин С. К. Методы тестирования: Учеб. пособие / Новосиб. гос. ун-т. Новосибирск, 2004. 166 с.

Учебное пособие ставит своей целью знакомство студентов с одним из основных подходов, применяемых для создания надежных программ, — тестированием. Начальные лекции посвящены обоснованию необходимости и важности данного курса для студентов.

Дано определение тестирования, теста, удачного теста. Рассмотрены психологические и экономические аспекты тестирования. Много внимания уделено критериям тестирования как средству создания **хорошего** набора тестов. Приведены критерии черного и белого ящика, причем для белого ящика рассмотрены наиболее известные критерии как для потока управления, потока данных, так и для наиболее важных языковых конструкций.

Вторая половина курса посвящена задачам автоматизации тестирования. Рассмотрены основные подходы и алгоритмы, обеспечивающие их реализацию. Приведены примеры систем реализующие, излагаемые подходы.

Рецензенты
д-р физ.-мат. наук, проф. В. А. Евстигнеев,
д-р физ.-мат. наук А. Г. Марчук

ОГЛАВЛЕНИЕ

1. ВВЕДЕНИЕ

При разработке больших программных проектов обычно выделяют шесть этапов, составляющих цикл разработки программного обеспечения или жизненный цикл программых [1]:

1) анализ требований, предъявляемых к системе;
2) определение спецификаций;
3) проектирование;
4) кодирование;
5) тестирование (автономное, комплексное и т. д.);
6) эксплуатация и сопровождение.

В данном учебном пособии мы не будем рассматривать все этапы, а остановимся только на **этапе тестирования** программных систем, при этом рассмотрим только некоторые, основные части этого этапа. Для понимания важности данного этапа и обоснования необходимости учебного пособия представим три диаграммы.

Рис. 1. Временные затраты на реализацию этапов цикла разработки программного обеспечения (без этапа сопровождения)

На диаграмме, приведенной на рис. 1, показано приблизительное распределение временных затрат на реализацию отдельных этапов цикла разработки программ. Диаграмма показывает, что этап тестирования может повлечь затраты, которые составят половину общих расходов на создание системы.

На рис. 1 дано распределение временных затрат по этапам разработки для нового проекта и без учета этапа сопровождения. Будучи формально правильной, данная диаграмма не отражает истинного положения дел. Более правильное представление о временных затратах дает рис. 2.

Рис. 2. Временные затраты на реализацию этапов цикла разработки программного обеспечения (с учетом этапа сопровождения)

Ни одна вычислительная система не остается неизменной. Поскольку заказчик зачастую не может четко сформулировать свои требования, он бывает не удовлетворен созданной системой и настаивает на внесении изменений в готовую систему. Обнаруживаются ошибки, пропущенные на этапе тестирования (на самом деле, как мы выясним далее, любая программная система содержит ошибки), и это требует доработки системы. Часто система требует развития как в результате изменения внешнего мира так и аппаратуры. Это видно на диаграмме цикла жизни программного обеспечения, приведенного на рис. 3.

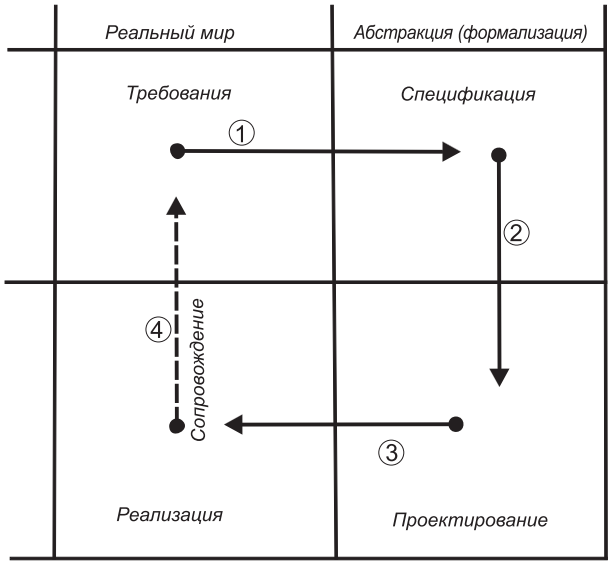


Рис. 3. Диаграмма цикла жизни программного обеспечения

Этап сопровождения включает в себя и работы по тестированию как вновь создаваемых компонент, так и старых при исправлении ошибок, что увеличивает общие затраты на тестирование.

Следовательно, хорошо спланированное тестирование, проводимое специалистами своего дела, и возможная автоматизация нужных подэтапов позволят значительно сократить как сроки разработки, так и материальные ресурсы и обеспечат создание надежного программного продукта. В данном учебном пособии вы познакомитесь с понятиями и методами тестирования, системами автоматизации этапов тестирования и применяемыми в них подходами, что позволит вам в дальнейшем профессионально подходить к тестированию программных продуктов. Основными первоисточниками для данного курса лекций послужили следующие работы [2, 5, 9–13, 30, 35, 38].

2. ПСИХОЛОГИЯ И ЭКОНОМИКА ТЕСТИРОВАНИЯ ПРОГРАММ

Тестирование как объект изучения можно рассматривать с различных чисто технических точек зрения. Однако наиболее важными при изучении тестирования представляются вопросы его экономики и психологии разработчика. Иными словами, достоверность тестирования программы в первую очередь определяется тем, кто будет ее тестировать и каков его образ мышления, и уже затем определенными технологическими аспектами. Поэтому, прежде чем перейти к техническим проблемам, остановимся на этих вопросах.

2.1. Психология

Поначалу может показаться тривиальным жизненно важный вопрос определения термина «тестирование». Необходимость обсуждения этого связана с тем, что часто этот термин определяют неверно, что, в свою очередь, приводит к плохому тестированию. Таковы, например, следующие определения:

- Тестирование представляет собой процесс, демонстрирующий отсутствие ошибок в программе.
- Цель тестирования — показать, что программа корректно исполняет предусмотренные функции.
- Тестирование — это процесс, позволяющий убедиться в том, что программа выполняет свое назначение.

Эти определения описывают нечто противоположное тому, что следует понимать под тестированием, поэтому они неверны. Оставив на время определения, предположим, что если мы тестируем программу, то ее стоимость должна увеличиться (так как тестирование стоит денег и нам желательно возвратить затраченную сумму путем увеличения стоимости программы). Увеличение стоимости возможно только при повышении качества или возрастания надежности программы. Последнее связано с обнаружением и удалением из нее ошибок. Следовательно, программа тестируется не для того, чтобы показать, что она работает, а скорее, наоборот: тестирование начинается с предположения, что в ней есть ошибки (это предположение справедливо практически для любой программы), а затем уже обнаруживается их максимально возможное число. Таким образом, сформулируем наиболее приемлемое определе-

ние: **тестирование — это процесс исполнения программы с целью обнаружения ошибок.**

Наши рассуждения могут показаться тонкой игрой семантик, однако практикой установлено, что именно ими в значительной мере определяется успех тестирования. Дело в том, что верный выбор цели дает важный психологический эффект, поскольку для человеческого сознания характерна целевая направленность. Если поставить целью демонстрацию отсутствия ошибок, то мы подсознательно будем стремиться к этой цели, выбирая тестовые данные, на которых вероятность появления ошибки мала. В то же время если нашей задачей станет обнаружение ошибок, то создаваемый нами тест будет обладать большей вероятностью обнаружения ошибки. Такой подход гораздо заметнее повысит качество программы, чем первый.

Из приведенного определения тестирования вытекает несколько следствий. Например, одно из них состоит в том, что тестирование — процесс деструктивный, т. е. обратный созидательному, конструктивному. Именно этим и объясняется, почему многие считают его трудным. Большинство людей склонны к конструктивному процессу созидания и в меньшей степени — к деструктивному. Из определения следует также, как нужно строить набор тестовых данных и кто должен (а кто нет) тестировать данную программу.

Для усиления определения тестирования проанализируем два понятия: *удачный* и *неудачный тест* — и, в частности, их использование при оценке результатов тестирования. Нередко мы называем тестовый прогон неудачным, если обнаружена ошибка, и, наоборот, удачным, если он прошел без ошибок. Чаще всего это является следствием ошибочного понимания термина «тестирование», так как, по существу, слово *удачный* означает *результативный*, а слово *неудачный* — *нежелательный*, *нерезультативный*. Но если тест не обнаружил ошибки, его выполнение связано с потерей времени и денег и термин *удачный* никак не может быть применен к нему. Поэтому будем называть тестовый прогон удачным, если в процессе его выполнения обнаружена ошибка, и неудачным, если получен корректный результат.

Проведем аналогию с посещением больным врача. Если рекомендованное врачом лабораторное исследование не обнаружило причины болезни, не назовем же мы такое исследование удачным — оно неудачно: ведь пациент потерял время и деньги и он все так же болен. Если же исследование показало, что у больного, например, язва желудка, то

оно является удачным, поскольку врач может прописать необходимый курс лечения. Следовательно, медики используют эти термины в нужном нам смысле. (Аналогия здесь заключается в том, что программа, которую предстоит тестировать, подобна больному пациенту.)

Определения наподобие «тестирование представляет собой процесс демонстрации отсутствия ошибок» порождают еще одну проблему: они ставят цель, которая не может быть достигнута ни для одной программы, даже весьма тривиальной. Результаты психологических исследований показывают, что если перед человеком ставится невыполнимая задача, то он работает хуже. Иными словами, определение тестирования как процесса обнаружения ошибок переводит его в разряд решаемых задач и таким образом преодолевается психологическая трудность.

Другая проблема возникает в том случае, когда для тестирования используется следующее определение: *тестирование — это процесс, позволяющий убедиться в том, что программа выполняет свое назначение*, поскольку программа, удовлетворяющая данному определению, может содержать ошибки. Если программа не делает того, что от нее требуется, то ясно, что она содержит ошибки. Однако ошибки могут быть и тогда, когда она делает то, что от нее не требуется. Например, бухгалтерская программа, правильно начисляющая зарплату, в ведомость дважды включает фамилию одного из сотрудников. Ошибки этого класса можно обнаружить скорее, если рассматривать тестирование как процесс поиска ошибок, а не демонстрацию корректности работы.

Таким образом, тестирование представляется деструктивным процессом попыток обнаружения ошибок в программе, наличие которых предполагается. Набор тестов, способствующий обнаружению ошибки, считается удачным. Естественно, в конечном счете каждый с помощью тестирования хочет добиться определенной степени уверенности в том, что его программа соответствует своему назначению и не делает того, для чего она не предназначена, но лучшим средством для достижения этой цели является непосредственный поиск ошибок. Лучший способ доказательства утверждения **моя программа великолепна** (значит, в том числе не содержит ошибок) — действовать от противного, т. е. попытаться его опровергнуть, обнаружить неточности, нежели просто согласиться с тем, что программа на определенном наборе входных данных работает корректно.

2.2. Экономика тестирования

Определив тестирование таким способом, необходимо на следующем шаге рассмотреть возможность создания множества тестов, обнаруживающих все ошибки программы. Покажем, что даже для самых тривиальных программ в общем случае невозможно построить наборы тестов, способные обнаружить все ошибки. Это, в свою очередь, порождает экономические проблемы, связанные с функциями человека в процессе тестирования. Для доказательства этого факта рассмотрим две стратегии тестирования, а именно тестирование программы как черного ящика и тестирование программы как белого ящика.

2.2.1. Тестирование программы как черного ящика

При использовании этой стратегии программа рассматривается как черный ящик, т. е. целью такого тестирования является выяснение обстоятельств, в которых поведение программы не соответствует ее спецификации. Тестовые данные разрабатываются только в соответствии со спецификацией программы, без учета знаний о ее внутренней структуре.

При таком подходе обнаружение всех ошибок в программе является критерием исчерпывающего входного тестирования. Последнее может быть достигнуто, если в качестве тестовых наборов использовать все возможные наборы входных данных. Необходимость выбора именно этого критерия иллюстрируется следующим примером.

Пример. *Программа вводит три целых числа, которые интерпретируются как длины сторон треугольника. Результатом работы программы является сообщение о типе треугольника: равносторонний, равнобедренный или неравносторонний.*

Если в этой задаче один треугольник корректно признан равносторонним, нет никакой гарантии того, что все остальные равносторонние треугольники так же будут корректно идентифицированы. Так, для треугольника со сторонами 6666, 6666, 6666 может быть предусмотрена специальная проверка и он считается неравносторонним. Поскольку программа представляет собой черный ящик, единственный способ удовлетворения приведенному выше критерию — перебор всех возможных входных значений.

Таким образом, исчерпывающий тест для задачи о треугольниках должен включать равносторонние треугольники с длинами сторон

вплоть до максимального целого числа. Это, безусловно, астрономическое число, но и оно не обеспечивает полноту проверки. Вполне вероятно, что останутся некоторые ошибки. Например, программа может представить треугольник со сторонами 3, 4, 5 неравносторонним, а со сторонами 2, A, 2 — равносторонним. Для того чтобы обнаружить подобные ошибки, нужно перебрать не только все разумные, но и вообще все возможные входные наборы. Следовательно, приходим к выводу, что для исчерпывающего тестирования задачи о треугольниках требуется бесконечное число тестов.

Если такое испытание представляется сложным, то еще сложнее создать исчерпывающий тест для большой программы. Допустим, что делается попытка тестирования методом черного ящика компилятора с языка ПАСКАЛЬ. Для построения исчерпывающего теста нужно использовать все множество правильных программ (фактически их число бесконечно) и все множество неправильных программ (действительно бесконечное число), чтобы убедиться в том, что компилятор обнаруживает все ошибки. Только в этом случае синтаксически неверная программа не будет компилирована. Если же программа имеет собственную память (например, операционная система, база данных или система резервирования билетов), то дело обстоит еще хуже. В таких программах исполнение команды (например, задания, запроса в базу данных, выполнение резервирования) зависит от того, какие события ей предшествовали, т. е. от предыдущих команд. Здесь следует перебрать не только все возможные команды, но и все их возможные последовательности.

Из изложенного следует, что построение исчерпывающего входного теста невозможно. Это подтверждается двумя аргументами: во-первых, нельзя создать тест, гарантирующий отсутствие ошибок; во-вторых, разработка таких тестов противоречит экономическим требованиям. Поскольку исчерпывающее тестирование исключается, нашей целью должна стать максимизация результативности тестирования (иными словами, максимизация числа ошибок, обнаруживаемых одним тестом). Для этого мы можем рассматривать внутреннюю структуру программы и делать некоторые разумные, но, конечно, не обладающие полной гарантией достоверности, предположения (например, разумно предполо-

жить, что если программа сочла треугольник 2, 2, 2 равносторонним, то таким же окажется и треугольник со сторонами 3, 3, 3).

2.2.2. Тестирование программы как белого ящика

Стратегия белого ящика, или стратегия тестирования, управляемого логикой программы, позволяет исследовать внутреннюю структуру программы. В этом случае тестирующий получает тестовые данные путем анализа логики программы (к сожалению, здесь часто не используется спецификация программы).

Сравним способ построения тестов при данной стратегии с исчерпывающим входным тестированием стратегии черного ящика. На первый взгляд может показаться, что достаточно построить такой набор тестов, в котором каждый оператор исполняется хотя бы один раз; нетрудно показать, что это неверно. Не вдаваясь в детали, укажем, что исчерпывающему входному тестированию может быть поставлено в соответствие исчерпывающее тестирование маршрутов. Программа проверена полностью, если с помощью тестов удается осуществить выполнение этой программы по всем возможным маршрутам ее исполнения (всем путям управляющего графа).

Последнее утверждение имеет два слабых пункта. Один из них состоит в том, что число не повторяющих друг друга маршрутов в программе — астрономическое. Чтобы убедиться в этом, рассмотрим управляющий граф довольно простой программы представленный на рис. 4. По-видимому, граф описывает программу из 10—20 операторов, включая цикл, который исполняется не менее 20 раз. Внутри цикла имеется несколько операторов IF. Для того чтобы определить число неповторяющихся маршрутов при исполнении программы, подсчитаем число неповторяющихся маршрутов из точки A в B в предположении, что все развилки взаимно независимы. Это число вычисляется как сумма $5^{20} + 5^{19} + ... + 5^1 = 10^{14}$ и равно 100 трлн, где 5 — число путей внутри цикла. Поскольку трудно оценить это число, приведем такой пример: если допустить, что на составление каждого теста мы тратим 5 мин, то для построения набора тестов нам потребуется примерно 1 млрд лет.

Конечно, в реальных программах условные переходы не могут быть взаимно независимы, т. е. число маршрутов исполнения будет несколько

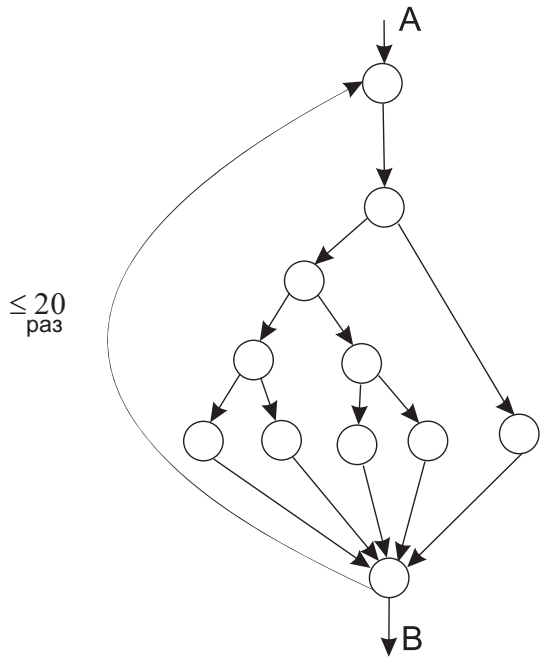


Рис. 4. Управляющий граф небольшой программы

меньше. Однако реальные программы значительно больше, чем простая программа, представленная на рис. 4. Следовательно, исчерпывающее тестирование маршрутов, как и исчерпывающее входное тестирование, не только невыполнимо, но и невозможно.

Второй слабый пункт утверждения заключается в том, что, хотя исчерпывающее тестирование маршрутов является полным тестом и если даже каждый маршрут программы будет проверен, сама программа все равно будет содержать ошибки. Это объясняется следующим образом. Во-первых, исчерпывающее тестирование маршрутов не может дать гарантии того, что программа соответствует спецификации. Например, вместо требуемой программы сортировки по возрастанию случайно была написана программа сортировки по убыванию. В этом случае ценность тестирования маршрутов невелика, поскольку после тестирования в программе окажется одна ошибка — программа неверна.

Во-вторых, программа может быть неверной в силу того, что пропущены некоторые маршруты. Исчерпывающее тестирование маршрутов не обнаружит их отсутствия. В-третьих, исчерпывающее тестирование маршрутов не может обнаружить ошибок, появление которых зависит от обрабатываемых данных. Приведем один пример: допустим, в программе необходимо выполнить сравнение двух чисел на сходимость, т. е. определить, является ли разность между двумя числами меньше предварительно определенного числа. В условном операторе может быть написано выражение ((A - B) $<$ EPSILON). Безусловно, оно содержит ошибку, поскольку необходимо выполнить сравнение абсолютных величин. Однако обнаружение этой ошибки зависит от конкретных значений **A** и **B** и ошибка не обязательно будет обнаружена путем исполнения каждого маршрута программы.

В заключение отметим, что, хотя исчерпывающее входное тестирование предпочтительнее исчерпывающего тестирования маршрутов, ни то, ни другое не могут стать полезными стратегиями, так как обе эти стратегии нереализуемы. Возможно, поэтому реальным путем, который позволит создать хорошую, но, конечно, не абсолютную стратегию, является сочетание тестирования программы как черного и как белого ящиков.

2.3. Принципы тестирования

Принципы тестирования сформулированы исходя из наибольшей важности в тестировании программ вопросов психологии. Принципы интересны тем, что в основном они интуитивно ясны, но в то же время на них часто не обращают должного внимания.

Описание предполагаемых значений выходных данных или результатов должно быть необходимой частью тестового набора.

Если результаты теста не были заранее определены то ошибочные, но правдоподобные результаты могут быть признаны правильными. Здесь мы сталкиваемся с явлением психологии: мы видим то, что хотим увидеть. Другими словами, несмотря на то что тестирование, по определению, — деструктивный процесс, есть подсознательное желание видеть корректный результат. Один из способов борьбы с этим состоит в поощрении детального анализа выходных значений переменных заранее, при разработке теста. Поэтому тест должен включать две компоненты: описание входных данных и описание точного и корректного результата, соответствующего набору входных данных. К сожалению,

часто мы не в состоянии определить конечные результаты и приходится оценивать их правдоподобность.

Необходимо избегать тестирования программы ее автором.

Этот принцип следует из обсуждавшихся ранее положений, которые определяют тестирование как деструктивный процесс. После выполнения конструктивной части при проектировании и написании программы программисту трудно перестроиться на деструктивный образ мышления.

Часто программисты не могут эффективно тестировать свои программы, потому что им трудно демонстрировать собственные ошибки. Это действительно сильный психологический фактор при коллективной разработке. Программист, тщательно тестирующий программу, невольно может работать медленнее, что становится известно другим участникам разработки. Тем самым итоги тестирования оказываются уже не просто делом одного человека, тестирующего свою программу, но и информацией, возбуждающей общественный интерес (и оценку!) участников разработки, в том числе ее руководителей. Перспектива создать о себе мнение как о специалисте, делающем много ошибок и медленно работающем, не воодушевляет программиста, и он подсознательно снижает (**может снизить**) требования к тщательности тестирования. В такой ситуации от всех требуется большое чувство такта и понимание процессов, чтобы поощрять специалистов, проводящих тщательное тестирование.

В дополнение к этой психологической проблеме следует отметить еще одну, более важную: программа может содержать ошибки, связанные с неверным пониманием постановки или описания задачи программистом. Тогда к тестированию программист приступит с таким же недопониманием задачи.

Тестирование можно уподобить работе корректора или рецензента над статьей или книгой. Многие авторы представляют себе трудности, связанные с редактированием собственной рукописи. Очевидно, что обнаружение недостатков в своей деятельности противоречит человеческой психологии.

Отсюда не следует, что программист не может тестировать свою программу. Делается лишь вывод о том, что тестирование будет более

эффективным, если оно выполняется кем-либо другим. Заметим, что все рассуждения не относятся к отладке, т. е. к локализации и исправлению уже известных ошибок. Эта работа эффективнее выполняется самим автором программы.

Программирующая организация не должна сама тестировать разработанные ею программы.

Во многих смыслах проектирующая или программирующая организация подобна живому организму с его психологическими проблемами. Работа программирующей организации или ее руководителя оценивается по их способности производить программы в рамках заданного времени и финансовых средств. Одна из причин такой системы оценок состоит в том, что временные и стоимостные показатели легко измерить, но в то же время чрезвычайно трудно количественно оценить надежность программы. Именно поэтому в процессе тестирования программирующей организации трудно быть объективной, поскольку тестирование в соответствии с данным определением может быть рассмотрено как средство уменьшения вероятности соответствия программы заданным временным и стоимостным параметрам.

Как и ранее, из изложенного не следует, что программирующая организация не может тестировать свои программные продукты; данный процесс в определенной степени может пройти успешно. Утверждается лишь то, что экономически более целесообразно выполнение тестирования каким-либо объективным, независимым подразделением.

Необходимо досконально изучать результаты применения каждого теста.

По всей вероятности, это наиболее очевидный принцип, но и ему часто не уделяется должное внимание. Значительная часть всех выявляемых в итоге ошибок зачастую может быть обнаружена в результате самых первых тестовых прогонов, но они бывают пропущены вследствие недостаточно тщательного анализа результатов каждого тестового прогона. Это особенно актуально для тех программных систем, когда мы не в состоянии точно определить результаты каждого теста.

Тесты для неправильных и непредусмотренных входных данных следует разрабатывать так же тщательно, как для правильных и предусмотренных.

При тестировании программ существует естественная тенденция концентрировать внимание на правильных и предусмотренных входных условиях, а неправильным и непредусмотренным входным данным не

придавать значения. Множество ошибок можно также обнаружить, если использовать программу новым, не предусмотренным ранее способом. Вполне вероятно, что тесты, представляющие неверные и неправильные входные данные, обладают большей обнаруживающей способностью, чем тесты, соответствующие корректным входным данным.

Необходимо проверять: делает ли программа то, для чего она предназначена, и не делает ли она то, что не должна делать.

Это логически просто вытекает из предыдущего принципа. Необходимо проверить программу на нежелательные побочные эффекты. Например, программа расчета зарплаты, которая производит правильные платежные чеки, окажется неверной, если она произведет лишние чеки для работающих или дважды занесет сотрудника в список личного состава.

Не следует выбрасывать тесты, даже если программа уже не нужна.

Эта проблема наиболее часто возникает при использовании интерактивных систем. Обычно тестирующий сидит за терминалом, на лету придумывает тесты и запускает программу на выполнение. При такой практике работы после применения тесты пропадают. После внесения изменений или исправления ошибок необходимо повторять тестирование, тогда приходится заново изобретать тесты. Как правило, этого стараются избегать, поскольку повторное создание тестов требует значительной работы. В результате повторное тестирование бывает менее тщательным, чем первоначальное. Можно переформулировать принцип так: тесты выбрасываются только вместе с программой.

Нельзя планировать тестирование в предположении, что ошибки не будут обнаружены.

Такую ошибку обычно допускают тестировщики, использующие неверное определение тестирования как процесса демонстрации отсутствия ошибок в программе или процесса демонстрации корректного функционирования программы.

Вероятность наличия необнаруженных ошибок в части программы пропорциональна числу ошибок, уже обнаруженных в этой части.

Этот принцип, не первый взгляд не согласующийся с интуитивным представлением, иллюстрируется графиком, приведенным на рис. 5, и подтверждается статистикой для многих программ. Например, допустим, что некоторая программа состоит из модулей или подпрограмм A и B. К определенному сроку в модуле A обнаружено пять ошибок,

а в модуле B — только одна, причем модуль A не подвергался более тщательному тестированию. Тогда из рассматриваемого принципа следует, что вероятность необнаруженных ошибок в модуле A больше, чем в модуле B. Справедливость этого принципа подтверждается тем, что для ошибок свойственно располагаться в программе в виде неких скоплений. В качестве примера можно рассмотреть операционные системы IBM S/370. В одной из версий операционной системы 47 % ошибок приходилось на 4 % модулей системы. Интуитивно понятно, что ошибки могут группироваться в частях программы, разрабатываемых программистами низкой квалификации, или в модулях, слабо проработанных идеологически, или в модулях имеющих большую сложность[1].

Рис. 5. Соотношение числа оставшихся и обнаруженных ошибок

Важность рассматриваемого принципа заключается в том, что он позволяет ввести обратную связь в процесс тестирования. Если в какой-нибудь части программы обнаружено больше ошибок, чем в других, то на ее тестирование должны быть направлены дополнительные усилия.

Тестирование — процесс творческий.

Для тестирования большой программы требуется больший творческий потенциал, чем для ее проектирования и программирования (см. диаграммы). Выше было показано, что нельзя построить полную

[1] См. понятие мер сложности для программных продуктов [3, 4].

систему тестов, обнаруживающих все ошибки. В дальнейшем будем обсуждать методы построения хороших наборов тестов, но применение этих методов должно быть творческим.

Чтобы подчеркнуть основные мысли, высказанные в данном разделе, приведем еще раз три наиболее важных принципа тестирования.

Тестирование — это процесс выполнения программ в целях обнаружения ошибок.

Хорошим считается тест, который имеет высокую вероятность обнаружения еще не выявленной ошибки.

Удачным считается тест, который обнаруживает еще не выявленную ошибку.

3. ПРОЕКТИРОВАНИЕ ТЕСТОВ

Невозможность создания «полного» набора тестов, показанная в предыдущих лекциях, побуждает к разработке стратегии проектирования тестов. Стратегия проектирования заключается в том, чтобы попытаться уменьшить эту «неполноту» настолько, насколько это возможно.

Если ввести ограничения на время, стоимость, машинное время и т. п., то ключевым вопросом тестирования становится следующий: *какое подмножество всех возможных тестов имеет наивысшую вероятность обнаружения большинства ошибок?*

Ответ на этот вопрос дает изучение методологий проектирования тестов.

Наихудшей из всех методологий является тестирование со случайными входными значениями (стохастическое) — процесс тестирования программы путем случайного выбора некоторого подмножества из всех возможных входных величин. Показано, что исчерпывающее тестирование по принципу черного или белого ящика в общем случае невозможно. Однако приемлемая стратегия тестирования может и должна обладать элементами обоих подходов. Можно разработать довольно полный тест, используя определенную методологию проектирования, основанную на принципе черного ящика, а затем дополнить его привлечением методов белого ящика.

Методы черного ящика, обсуждаемые в курсе лекций, следующие:

— эквивалентное разбиение;

— анализ граничных значений;

— применение функциональных диаграмм;

— предположение об ошибке.

Методологии белого ящика можно разделить на три типа: критерии потока управления, критерии потока данных и критерии для определенных языковых конструкций[1]. Мы рассмотрим методы построения тестов всех типов.

Рекомендуемая процедура построения тестов заключается в том, чтобы разрабатывать тесты, используя методы черного ящика, а затем как необходимое условие — дополнительные тесты, используя методы белого ящика.

3.1. Критерии черного ящика

3.1.1. Эквивалентное разбиение

Выше отмечено, что хороший тест имеет приемлемую вероятность обнаружения ошибки и что исчерпывающее входное тестирование программы невозможно. Следовательно, тестирование программы ограничивается использованием небольшого подмножества всех возможных входных данных. Поэтому хотелось бы выбрать для тестирования наиболее подходящее подмножество, т. е. подмножество с наивысшей вероятностью обнаруживающих большинство ошибок.

Правильно выбранный тест этого подмножества должен обладать двумя свойствами:

а) уменьшать, причем более чем на единицу, число других тестов, которые должны быть разработаны для достижения заранее определенной цели «приемлемого» тестирования;

б) покрывать значительную часть других возможных тестов, что в некоторой степени свидетельствует о наличии или отсутствии ошибок до и после применения этого ограниченного множества значений входных данных.

Указанные свойства, несмотря на их кажущееся подобие, описывают два различных положения. Во-первых, каждый тест должен включать столько различных входных условий, сколько это возможно, с тем чтобы минимизировать общее число необходимых тестов. Во-вторых, необходимо пытаться разбить входную область программы на конечное число классов эквивалентности так, чтобы можно было предположить (конечно, не абсолютно уверенно), что каждый тест, являющийся представителем некоторого класса, эквивалентен любому другому тесту этого класса. Иными словами, если один тест класса эквивалентности

[1] Книга Г. Майерса содержит только критерии для потока управления [2].

обнаруживает ошибку, то следует ожидать, что и все другие тесты этого класса эквивалентности будут обнаруживать ту же самую ошибку. И наоборот, если тест не обнаруживает ошибки, то следует ожидать, что ни один тест этого класса эквивалентности не будет обнаруживать ошибки.

Эти два положения составляют основу методологии тестирования по принципу черного ящика, известной как эквивалентное разбиение. Второе положение используется для разработки набора «интересных» условий, которые должны быть протестированы и на их основе создания классов эквивалентности, а первое — для разработки минимального набора тестов, покрывающих эти условия (эти классы).

Примером класса эквивалентности для программы о треугольнике (см. пример на с. 11) является набор *трех равных чисел, имеющих целые значения, большие нуля*. Определяя этот набор как класс эквивалентности, получаем, что если ошибка не обнаружена некоторым тестом данного набора, то маловероятно, что она будет обнаружена другим тестом этого класса. Иными словами, в этом случае время тестирования лучше потратить на что-нибудь другое (на тестирование других классов эквивалентности).

Разработка тестов методом эквивалентного разбиения осуществляется в два этапа: 1) выделение классов эквивалентности и 2) построение тестов.

Выделение классов эквивалентности. Классы эквивалентности выделяются путем выбора каждого входного условия (обычно это предложение или фраза в спецификации) и разбиением его на две или более групп. Для проведения этой операции используют табл. 1. Заметим, что различают два типа классов эквивалентности: правильные классы эквивалентности, представляющие правильные входные данные программы, и неправильные классы эквивалентности, представляющие все другие возможные состояния условий, — ошибочные входные значения. Таким образом, придерживаются одного из принципов тестирования о необходимости уделять столько же внимание неправильным или неожиданным условиях сколько и правильным.

Выделение классов эквивалентности на основе спецификации представляет собой в значительной степени эвристический процесс, но при этом существует ряд правил:

1. Если входное условие описывает область значений (например, «целое данное может принимать значения от 1 до 9»), то определяются

Таблица 1

Форма таблицы для перечисления классов эквивалентности

Входные условия	**Правильные классы эквивалентности**	**Неправильные классы эквивалентности**

один правильный класс эквивалентности ($1 \leq$ значение целого данного ≤ 9) и два неправильных (значение целого данного < 1 и значение целого данного > 9).

2. Если входное условие описывает число значений (например, «в автомобиле могут ехать от одного до пяти человек»), то определяются один правильный класс эквивалентности и два неправильных (ни одного и более пяти человек).

3. Если входное условие описывает множество входных значений и есть основание полагать, что каждое значение программа трактует особо (например, «известны способы передвижения пешком и на велосипеде»), то определяется правильный класс эквивалентности для каждого значения и один неправильный класс эквивалентности (например, «на лошади»).

4. Если входное условие описывает ситуацию «должно быть» (например, «первым символом идентификатора должна быть буква»), то определяется один правильный класс эквивалентности (первый символ — буква) и один неправильный (первый символ — не буква).

5. Если есть любое основание считать, что различные элементы класса эквивалентности трактуются программой неодинаково, то данный класс эквивалентности разбивается на меньшие классы эквивалентности.

Этот процесс далее будет кратко показан на конкретном примере.

Построение тестов. Второй шаг заключается в использовании классов эквивалентности для построения тестов. Этот процесс включает в себя:

1. Назначение каждому классу эквивалентности уникального номера.

2. Проектирование новых тестов, каждый из которых покрывает как можно большее число непокрытых правильных классов эквивалентности, до тех пор пока все правильные классы эквивалентности не будут покрыты тестами.

3. Запись тестов, каждый из которых покрывает один и только один из непокрытых неправильных классов эквивалентности, до тех пор пока все неправильные классы эквивалентности не будут покрыты тестами.

Причина покрытия неправильных классов эквивалентности индивидуальными тестами состоит в том, что определенные проверки с ошибочными входными данными скрывают или не позволяют исполнить другие проверки с ошибочными входными данными.

Пример. Предположим, что при разработке компилятора требуется протестировать синтаксическую правильность оператора МАССИВ.

Спецификация приведена ниже.

В спецификации элементы, написанные латинскими буквами, обозначают синтаксические единицы, которые в реальных операторах должны быть заменены соответствующими значениями, в квадратные скобки заключены необязательные элементы, многоточие показывает, что предшествующий ему элемент может быть повторен подряд несколько раз.

Оператор МАССИВ используется для описания массивов, его форма: МАССИВ op[,op] ... , где op есть описатель массива, который имеет вид: name(d[,d] ...), name — имя массива, а d — индекс массива. Имена массива могут содержать от одного до шести символов — букв или цифр, причем первой должна быть буква. Допускается от одного до семи индексов. Форма индекса [lb :] ub, где lb и ub задают нижнюю и верхнюю границы индекса массива. Граница может быть либо константой, принимающей значения от -65534 до 65535, либо целой переменной (без индексов). Если lb не определена, то предполагается, что она равна единице. Значение ub должно быть больше или равно lb. Если lb определена, то она может принимать любое целое значение.

(Конец спецификации.)

Первый шаг заключается в том, чтобы выделить входные условия и по ним определить классы эквивалентности (табл. 2). Классы эквивалентности в таблице обозначены числами в скобках.

Следующий шаг — построение теста, покрывающего один или более правильных классов эквивалентности. Например, тест

МАССИВ A(2)

Таблица 2

Классы эквивалентности

Входные условия	**Правильные классы эквивалентности**	**Неправильные классы эквивалентности**
Число описателей массивов	Один (1), больше одного (2)	Ни одного (3)
Длина имени массива	От 1 до 6 символов (4)	0 (5), больше 6 (6)
Имя массива	Имеет в составе буквы (7) и цифры (8)	Что-то еще (9)
Имя массива начинается с буквы	Да (10)	Нет (11)
Число индексов	1—7 (12)	0 (13), больше 7 (14)
Верхняя граница	Константа (15), целая переменная (16)	Имя элемента массива (17), что-то иное (18)
Имя целой переменной	Имеет в составе буквы (19) и цифры (20)	Что-то еще (21)
Целая переменная начинается с буквы	Да (22)	Нет (23)
Константа	-65534 — 65535 (24)	Меньше -65534 (25), больше 65535 (26)
Нижняя граница определена	Да (27), нет (28)	
Верхняя граница относительно нижней	Больше (29), равна (30)	Меньше (31)
Значение нижней границы	Отрицательное (32), 0 (33), положительное (34)	
Нижняя граница	Константа (35), целая переменная (36)	Имя элемента массива (37), что-то иное (38)
Оператор расположен внескольких строках	Да (39), нет (40)	

покрывает классы 1, 4, 7, 10, 12, 15, 24, 28, 29 и 40. Далее определяются один или более тестов, покрывающих оставшиеся правильные классы эквивалентности. Так, тест

МАССИВ A12345(I,9,J4XXXX,65535,1,KLM,100), BBB(-65534 : 100, 0 : 1000, 10 : 10, I : 65535)

покрывает оставшиеся классы. Перечислим неправильные классы эквивалентности и соответствующие им тесты:

(3) : МАССИВ
(5) : МАССИВ (10)
(6) : МАССИВ A234567(2)
(9) : МАССИВ A.1(2)
(11) : МАССИВ 1A(10)
(13) : МАССИВ B
(14) : МАССИВ B (4,4,4,4,4,4,4,4)
(17) : МАССИВ B (4, A (2))
(18) : МАССИВ B(4,,7)
(21) : МАССИВ C(I.,10)
(23) : МАССИВ C(10,1J)
(25) : МАССИВ D(- 65535 : 1)
(26) : МАССИВ D(65536)
(31) : МАССИВ D(4 : 3)
(37) : МАССИВ D(A(2) : 4)
(38) : МАССИВ D(. : 4)

3.1.2. Анализ граничных значений

Граничные условия — это ситуации, возникающие непосредственно выше, ниже или на границах входных и выходных классов эквивалентности. Как показывает опыт, тесты, исследующие граничные условия, приносят большую пользу, чем тесты, которые их не исследуют. Анализ граничных значений отличается от эквивалентного разбиения в двух отношениях:

1. Выбор любого элемента в классе эквивалентности в качестве представительного при анализе граничных значений осуществляется таким образом, чтобы проверить тестом каждую границу этого класса. Рассматриваются тесты для каждой границы класса эквивалентности и тесты незначительного выхода за границы.

2. При разработке тестов рассматривают не только входные условия (пространство входов), но и пространство результатов (т. е. выходные классы эквивалентности).

Анализ граничных значений требует определенной степени творчества и специализации в рассматриваемой проблеме. Анализ граничных значений, как и многие другие аспекты тестирования, в значительной мере основывается на способностях человеческого интеллекта. Тем не менее приведем несколько общих правил метода.

1. Построить тесты для границ области и тесты с неправильными входными данными для ситуаций незначительного выхода за границы области, если входное условие описывает область значений. Например, если правильная область входных значений есть -1,0—+1,0, то написать тесты для ситуаций -1,0, 1,0, -1,001 и 1,001.

2. Построить тесты для минимального и максимального значений условий и тесты, большие и меньшие этих значений, если входное условие удовлетворяет дискретному ряду значений. Например, если входной файл может содержать от 1 до 255 записей, то разработать тесты для 0, 1, 255 и 256 записей.

3. Использовать правило 1 для каждого выходного условия. Например, если программа вычисляет ежемесячный расход топлива и если минимум расхода составляет 0,00 л, а максимум — 65 л, то построить тесты, которые вызывают расходы с 0,00 л и 65 л. Кроме того, построить, если это возможно, тесты, которые вызывают отрицательный расход и расход больше 65 л. Не всегда можно получить результат вне выходной области, но тем не менее следует рассмотреть эту возможность.

4. Использовать правило 2 для каждого выходного условия. Например, если система отображает на экране терминала состояние памяти в зависимости от входного запроса, но никак не более 10 строк, то построить тесты такие, чтобы программа отображала ноль, одну и 10 строк, и тест, который мог бы вызвать выполнение программы с ошибочным отображением 11 строк.

5. Если вход или выход программы есть упорядоченное множество (например, последовательный файл, линейный список, таблица), то сосредоточить внимание на первом и последнем элементах этого множества.

6. Попробовать свои силы в поиске других граничных условий.

Чтобы проиллюстрировать необходимость анализа граничных значений, можно использовать программу анализа треугольника, приведенную ранее. Для задания треугольника входные значения должны быть целыми положительными числами и сумма любых двух из них должна быть больше третьего. Если определены эквивалентные разби-

ения, то целесообразно определить одно разбиение, в котором это условие выполняется, и другое, в котором сумма двух целых не больше третьего. Следовательно, двумя возможными тестами являются 3–4–5 и 1–2–4. Тем не менее здесь есть вероятность пропуска ошибки. Например, если выражение в программе было закодировано как A + B $\geq$ C вместо A + B $>$ C, то программа ошибочно сообщала бы нам, что числа 1–2–3 представляют треугольник.

Таким образом, существенное различие между анализом граничных значений и эквивалентным разбиением заключается в том, что анализ граничных значений исследует ситуации, возникающие на и вблизи границ эквивалентных разбиений.

3.1.3. Метод функциональных диаграмм

Одним из недостатков критериев анализа граничных значений и эквивалентного разбиения является то, что они не исследуют комбинаций входных условий.

Тестирование комбинаций входных условий — непростая задача, поскольку даже при построенном эквивалентном разбиении входных условий число комбинаций может быть астрономически велико. Если нет систематического способа выбора комбинаций подмножества входных условий, то, как правило, выбирается произвольное подмножество комбинаций, возможно приводящее к неэффективному тесту.

Метод функциональных диаграмм или диаграмм причинно-следственных связей помогает систематически выбирать высокорезультативные тесты. Он дает полезный побочный эффект, так как позволяет обнаруживать неполноту и неоднозначность исходных спецификаций. Функциональная диаграмма представляет собой формальный язык, на который транслируется спецификация, написанная на естественном языке. Базовые элементы этого языка представлены на рис. 6. Построение тестов этим методом осуществляется в несколько этапов.

1. Спецификация разбивается на «рабочие» участки. Это связано с тем, что функциональные диаграммы становятся слишком громоздкими при применении данного метода к большим спецификациям.

2. В спецификации определяются причины и следствия. Причина есть отдельное входное условие или класс эквивалентности входных условий. Следствие есть выходное условие или преобразование системы (остаточное действие, которое входное условие оказывает на состояние программы или системы). Причины и следствия определяются путем

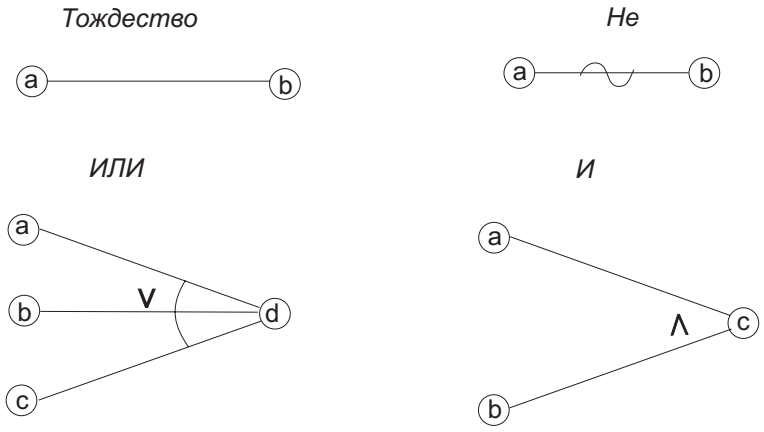


Рис. 6. Базовые элементы языка функциональных диаграмм

последовательного чтения спецификации. При этом выделяются слова или фразы, которые описывают причины и следствия. Каждой причине и следствию приписывается отдельный номер.

3. Анализируется семантическое содержание спецификации, которая преобразуется в булевский граф, связывающий причины и следствия (с помощью базовых элементов). Это и есть функциональная диаграмма.

4. Диаграмма снабжается примечаниями, задающими ограничения и описывающими комбинации причин и (или) следствий, которые являются невозможными из-за синтаксических или внешних ограничений.

5. Путем методического прослеживания состояний условий диаграммы она преобразуется в таблицу решений с ограниченными входами. Каждый столбец таблицы решений соответствует тесту.

6. Столбцы таблицы решений преобразуются в тесты.

Базовые символы для записи функциональных диаграмм показаны на рис. 6. Каждый узел диаграммы может находиться в двух состояниях — 0 или 1; 0 обозначает состояние «отсутствует», а 1 — «присутствует». Функция *тождество* устанавливает, что если значение a есть 1, то и значение b есть 1; в противном случае значение b есть 0. Функция *не* устанавливает, что если a есть 1, то b есть 0; в противном случае b есть 1. Функция *или* устанавливает, что если a, или b, или c есть 1, то d есть 1; в противном случае d есть 0. Функция *и* устанавливает, что если и a, и b есть 1, то и c есть 1; в противном случае c есть 0. Последние две функции разрешают иметь любое число входов.

В большинстве программ определенные комбинации причин невозможны из-за синтаксических или внешних ограничений (например, символ не может принимать значения A и B одновременно). В этом случае используются дополнительные логические ограничения, изображенные на рис. 7. Ограничение E устанавливает, что E должно быть истинным, если хотя бы одна из причин — a или b — принимает значение 1 (a и b не могут принимать значение 1 одновременно). Ограничение I устанавливает, что по крайней мере одна из величин a, b или c всегда должна быть равной 1 (a, b и c не могут принимать значение 0 одновременно). Ограничение O устанавливает, что одна и только одна из величин a или b должна быть равна 1. Ограничение R устанавливает, что если a принимает значение 1, то и b должна принимать значение 1, т. е. невозможно, чтобы a было равно 1, а b — 0.

Часто возникает необходимость в ограничениях для следствий. Ограничение M устанавливает, что если следствие a имеет значение 1, то следствие b должно принять значение 0.

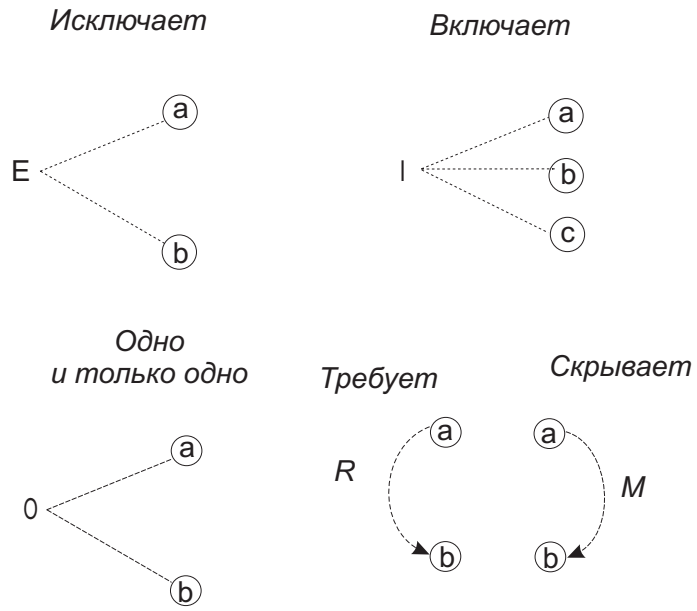


Рис. 7. Символы ограничений

Для иллюстрации изложенного рассмотрим диаграмму, отображающую ниже приведенную спецификацию.

Первый символ строки должен быть буквой A или B, а второй — цифрой, и в этом случае файл обновляется. Если первый символ неправильный, то выдается сообщение X12, а если второй символ неправильный — сообщение X13.

В табл. 3 приведены причины, следствия и соответствующие им номера.

Таблица 3

Таблица с причинами и следствиями

Номер	**Причины**	**Номер**	**Следствия**
1	Первый символ A	70	Файл обновляется
2	Первый символ B	71	Выдается сообщение X12
3	Второй символ — цифра	72	Выдается сообщение X13

Функциональная диаграмма показана на рис. 8 слева. Отметим, что здесь создан промежуточный узел 11. (Следует убедиться в том, что диаграмма действительно отображает данную спецификацию, задавая причинам все возможные значения и проверяя, принимают ли при этом следствия правильные значения.)

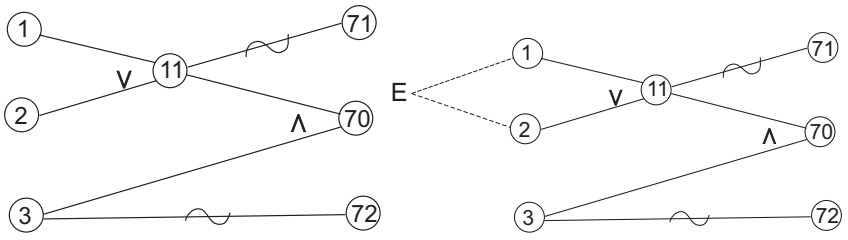


Рис. 8. Пример функциональной диаграммы

Хотя левая диаграмма, приведенная на рис. 8, отображает спецификацию, она содержит невозможную комбинацию причин — причины 1 и 2 не могут быть установлены в 1 одновременно, но возможно, чтобы присутствовала одна из них. Следовательно, они связаны ограничением E (рис. 8, правая диаграмма).

Процедура генерации таблицы решений заключается в следующем:

1. Выбрать некоторое следствие, состояние которого должно быть установлено в 1.

2. Найти все комбинации причин (с учетом ограничений), которые установят это следствие в 1, прокладывая из этого следствия обратную трассу через диаграмму.

3. Построить столбец в таблице решений для каждой комбинации причин.

4. Для каждой комбинации причин определить состояния всех других следствий и поместить их в соответствующий столбец таблицы решений.

При выполнении этого шага необходимо руководствоваться тремя положениями, задающими ограничения:

1. Если обратная трасса прокладывается через узел *или*, выход которого должен принимать значение 1, то одновременно не следует устанавливать в 1 более одного входа в этот узел. Такое ограничение на установку входных значений называется чувствительностью пути. Цель данного правила — избежать пропуска определенных ошибок из-за того, что одна причина маскируется другой.

2. Если обратная трасса прокладывается через узел u, выход которого должен принимать значение 1[1], то все комбинации входов, приводящие выход в 1[2], должны быть в конечном счете перечислены. Однако когда исследуется ситуация, где один вход есть 0, а один или более других входов есть 1, не обязательно перечислять все условия, при которых остальные входы могут быть 1.

3. Если обратная трасса прокладывается через узел u, выход которого должен принимать значение 0, то необходимо указать лишь одно условие, согласно которому все входы являются нулями. (Когда узел u находится в середине графа и его входы исходят из других промежуточных узлов, может существовать чрезвычайно большое число ситуаций, при которых все его входы принимают значения 0.)

Пояснения положений, используемые при прокладке обратной трассы через диаграмму для состояний рис. 9:

1. Если X должен быть равен 1, то не следует рассматривать ситуацию, где $a = b = 1$ (положение 1).

---

[1] Очевидно, в данном ограничении в работе [2] имеется опечатка. См. также пояснение для состояния 3 в п. 3 рис 9.

[2] Та же опечатка.

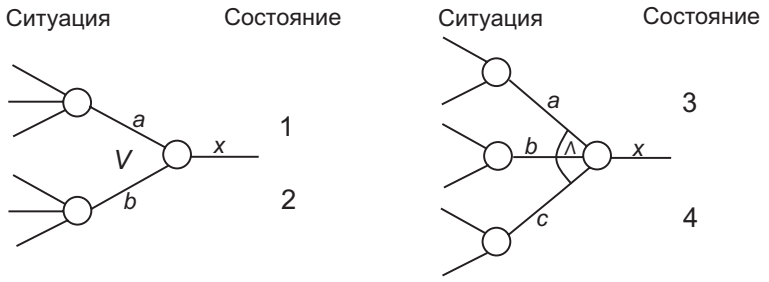


Рис. 9. Положения для прокладки обратной трассы через диаграмму

2. Если X должен быть равен 0, то перечислить все ситуации, где $a = b = 0$.

3. Если X должен быть равен 1, то перечислить все ситуации, где $a = b = c = 1$.

4. Если X должен быть равен 0, то рассмотреть только одну ситуацию, где $a = b = c = 0$ (положение 3). Для состояний a, b и c со значениями 001, 010, 100, 011, 101 и 110 рассмотреть только одну, любую из этих ситуаций (положение 2).

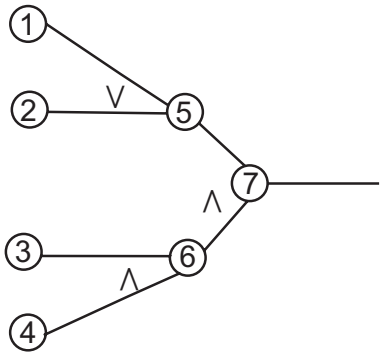


Рис. 10. Пример функциональной диаграммы для иллюстрации обратной трассировки

Рис. 10 приведен в качестве примера функциональной диаграммы. Пусть требуется так задать входные условия, чтобы установить выходное состояние в значение 0. Согласно положению 3, следует рассмат-

ривать только один случай, когда узлы 5 и 6 — нули. По положению 2 для состояния, при котором узел 5 принимает значение 1, а узел 6 — значение 0, следует рассматривать только один случай, когда узел 5 принимает значение 1 (не перечисляя другие возможные случаи, когда узел 5 может принимать значение 1). Аналогично для состояния, при котором узел 5 принимает значение 0, а узел 6 — значение 1, следует рассматривать только один случай, когда узел 6 принимает значение 1 (хотя в данном примере он является единственным). В соответствии с положением 1 если узел 5 должен быть установлен в состояние 1, то не рекомендуется устанавливать узлы 1 и 2 в состояние 1 одновременно. Таким образом, возможны пять состояний узлов 1—4, (например, значения 0 0 0 0 (5=0, 6=0), 1 0 0 0 (5 = 1, 6 = 0), 1 0 0 1 (5 = 1, 6 = 0), 1 0 1 0 (5 = 1, 6 = 0), 0 0 1 1 (5 = 0, 6 =1)), а не 13, которые приводят к выходному состоянию 0.

На первый взгляд эти положения могут показаться необъективными, но они преследуют важную цель — уменьшить комбинаторику диаграммы. Их применение позволяет избежать ситуаций, которые приводят к получению малорезультативных тестов. Если не исключать малорезультативные тесты, то общее число тестов, порождаемых по большой функциональной диаграмме, получается астрономическим. Поэтому самое лучшее — исключить малорезультативные тесты в процессе анализа диаграммы.

Пример. Команда ПОКАЗ используется для того, чтобы отобразить на экране состояние памяти ЭВМ. Синтаксис команды задается выражением

$$\text{ПОКАЗ} \quad \left[\begin{array}{l}\text{адрес 1}\\ \underline{0}\end{array}\right] \quad \left[\begin{array}{l}-\text{адрес 2}\\ -\text{КОНЕЦ}\\ .\text{число байт}\\ .\underline{1}\end{array}\right] \qquad (1)$$

Квадратные скобки обозначают возможное отсутствие операндов. Подчеркнутые операнды соответствуют умолчательным значениям (т. е. если операнд опущен, то принимается умолчательное значение).

Первый операнд («адрес 1») определяет адрес первого байта области памяти, содержимое которой должно быть отображено на экран.

Второй операнд определяет объем памяти, который должен быть отображен. Если «адрес 2» определен, то он, в свою очередь, указывает адрес последнего байта области памяти, которую необходимо отобра-

зить на экран. «Адрес 1» и «адрес 2» — шестнадцатеричные числа и их длина не превышает шести символов. «Адрес 2» должен быть больше или равен начальному адресу («адрес 1»). Оба адреса должны принимать значения из действительной области памяти машины. Если в качестве второго операнда определено «КОНЕЦ», то память отображается до последнего действительного адреса машины. Если же в качестве операнда указано «число байт», то выводится заданное число байтов памяти, которые нужно отобразить (начиная с байта с адресом «адрес 1»). Данный операнд является шестнадцатеричным числом длиной от одной до шести цифр. Сумма значений операндов «число байт» и «адрес 1» не должна превышать действительного размера памяти плюс единица. Состояние памяти отображается на экран терминала в виде одной или нескольких строк следующего формата:

xxxxxx = слово 1 слово 2 слово 3 слово 4,

где xxxxxx есть шестнадцатеричный адрес байта в первом слове.

Всегда отображается полное число слов (четырех байтовых последовательностей, где адрес первого байта в слове кратен четырем) независимо от значения операнда «адрес 1» или отображаемого объема памяти. Все выходные строки всегда содержат четыре слова (16 байт). Первый байт отображаемой области памяти находится в пределах первого слова.

Могут иметь место следующие сообщения об ошибках:

Ml Неправильный синтаксис команды.

M2 Запрашивается адрес больше допустимого.

M3 Запрашивается область памяти с нулевым или отрицательным адресом.

Примеры команды ПОКАЗ:

ПОКАЗ

отображает содержимое первых четырех слов памяти (стандартное значение начального адреса 0, а стандартное значение счетчика байтов 1);

ПОКАЗ 77F

отображает слово, содержащее байт с адресом 77F, и три последующих слова;

ПОКАЗ 77F — 407A

отображает слова, содержащие байты с адресами от 77F до 407A;

ПОКАЗ 77F .6

отображает слова, содержащие шесть байт, начиная с адреса 77F;

ПОКАЗ 50FF — КОНЕЦ

отображает слова, содержащие байты, начиная с адреса 50FF и до конца памяти.

Первый шаг заключается в тщательном анализе спецификации с тем, чтобы идентифицировать причины и следствия.

Причинами являются:

1. Наличие первого операнда.

2. Операнд «адрес 1» содержит только шестнадцатеричные цифры.

3. Операнд «адрес 1» содержит от одного до шести символов.

4. Операнд «адрес 1» находится в пределах действительной области памяти.

5. Второй операнд есть «КОНЕЦ».

6. Второй операнд есть «адрес 2».

7. Второй операнд есть «число байт».

8. Второй операнд отсутствует.

9. Операнд «адрес 2» содержит только шестнадцатеричные цифры.

10. Операнд «адрес 2» содержит от одного до шести символов.

11. Операнд «адрес 2» находится в пределах действительной области памяти.

12. Операнд «адрес 2» больше или равен операнду «адрес 1».

13. Операнд «число байт» содержит только шестнадцатеричные цифры.

14. Операнд «число байт» содержит от одного до шести символов.

15. «Число байт» + «адрес 1» $\leq$ размер памяти + 1.

16. «Число байт» $\geq$ 1.

17. Запрашиваемая область памяти настолько велика, что для отображения требуется много строк на экране.

18. Начало выводимой области памяти не выравнено на границу слова.

Каждой причине соответствует произвольный единственный номер. Заметим, что для описания второго операнда необходимы четыре причины (5—8), так как второй операнд может принимать значения: 1) «КОНЕЦ»; 2) «адрес 2»; 3) «число байт»; 4) может отсутствовать; 5) неопределенное значение, т. е. ни одно из указанных выше.

Следствия:

91. На экран выводится сообщение Ml.

92. На экран выводится сообщение М2.

93. На экран выводитсяся сообщение М3.

94. Память отображается одной строкой.

95. Для отображения содержимого памяти требуется много строк.

96. Первый байт отображаемой области памяти выравнен на границу слова.

97. Первый байт отображаемой области памяти не выравнен на границу слова.

Второй шаг — разработка функциональной диаграммы. Узлы причин перечислены по вертикали у левого края страницы; узлы следствий собраны по вертикали у ее правого края. Тщательно анализируется семантическое содержание спецификации, с тем чтобы связать причины и следствия (т. е. показать, при каких условиях имеет место следствие). На рис. 11 приведена соответствующая функциональная диаграммы. Промежуточный узел 32 представляет синтаксически правильный первый операнд, узел 35 — синтаксически правильный второй операнд, а узел 36 — синтаксически правильную команду. Если значение узла 36 есть 1, то следствие 91 (сообщение об ошибке) отсутствует. Если значение узла 36 есть 0, то следствие 91 имеет место. Тщательно проверьте эту диаграмму и убедитесь в том, что она точно отображает спецификацию.

Если диаграмму на рис. 11 непосредственно использовать для построения тестов, то создание многих из них на самом деле окажется невозможным. Это объясняется тем, что определенные комбинации причин не могут иметь места из-за синтаксических ограничений. Например, причины 2 и 3 не могут присутствовать без причины 1. Причина 4 не может присутствовать, если нет причин 2 и 3. На рис. 12 показана окончательная диаграмма со всеми дополнительными ограничениями. Заметим, что может присутствовать только одна из причин 5, 6, 7 или 8. Другие ограничения причин являются условиями типа требует. Причина 17 (много строк на экране) и причина 8 (второй операнд отсутствует) связаны отношением не; причина 17 может присутствовать только в отсутствие причины 8. Опять-таки необходимо тщательно исследовать все ограничения, налагаемые на условия. Третьим шагом является генерация таблицы решений.

Преобразуем функциональную диаграмму, изображенную на рис. 12, в таблицу решений. Выберем первым следствием 91. Следствие 91 имеет место, если узел 36 принимает значение 0. Значение узла 36 есть 0, если значение узлов 32 и 35 есть 0,0, 0,1 или 1,0 и применимы положения 2 и 3. Хотя подобное преобразование — трудоемкий процесс, но можно проложить обратную трассу от следствий к причинам, учесть при этом

ограничения причин и найти комбинации последних, которые приводят к следствию 91.

Результирующая таблица решений представлена в виде двух таблиц (табл. 4, 5). Столбцы 1—11 представляют ситуации, при которых имеет следствие 91. (Столбцы (тесты) 1—3 представляют условия, где узел 32 есть 0, а узел 35 есть 1. Столбцы 4—10 представляют условия, где узел 32 есть 1, а узел 35 есть 0. С помощью положения 3 определена только одна ситуация (колонка 11) из 21 возможной, когда значения узлов 32 и 35 есть 0. Пробелы представляют «безразличные» ситуации (т. е. состояние причины несущественно) или указывают на то, что состояние причины очевидно вследствие состояний других зависимых причин (например, для столбца 1 известно, что причины 5, 7 и 8 должны принимать значения 0, так как они связаны ограничением «одно и только одно» с причиной 6).) Столбцы 12—15 представляют ситуации, при которых имеет место следствие 92, а столбцы 16 и 17 — ситуации, при которых имеет место следствие 93 и т. д. Последний шаг заключается в том, чтобы преобразовать таблицу решений в 38 тестов. Данный набор тестов представлен ниже. Числа возле каждого теста обозначают следствия, которые, как ожидается, должны иметь место. Предполагаем, что последний адрес памяти машины есть 7FFF.

1. ПОКАЗ 234AF74 — 123 (91)
2. ПОКАЗ 2ZX4 — 3000 (91)
3. ПОКАЗ НННННННН — 2000 (91)
4. ПОКАЗ 200 (91)
5. ПОКАЗ 0 — 22222222 (91)
6. ПОКАЗ 1 — 2X (91)
7. ПОКАЗ 2 — ABCDEFGHI (91)
8. ПОКАЗ 3 .1111111 (91)
9. ПОКАЗ 44 .$42 (91)
10. ПОКАЗ 100 .$$$$ (91)
11. ПОКАЗ 10000000 — M (91)
12. ПОКАЗ FF — 8000 (92)
13. ПОКАЗ FFF .7001 (92)
14. ПОКАЗ 8000 — КОНЕЦ (92)
15. ПОКАЗ 8000 — 8001 (92)
16. ПОКАЗ AA — A9 (93)
17. ПОКАЗ 7000 .0 (93)
18. ПОКАЗ 7FF9 — КОНЕЦ (94,97)

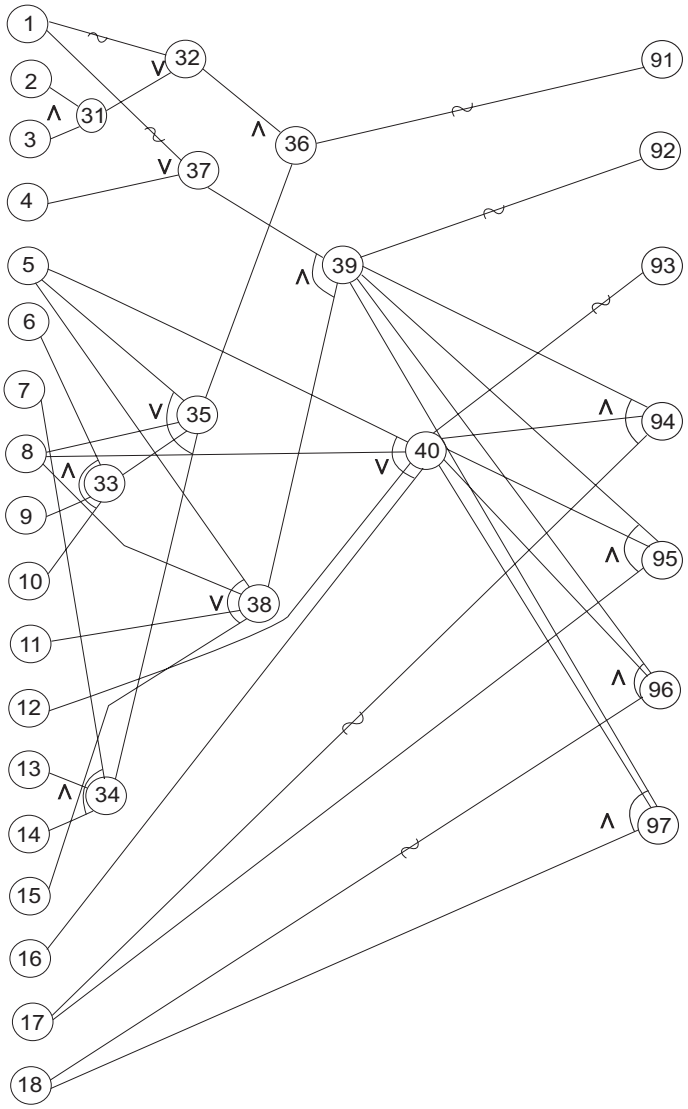


Рис. 11. Функциональная диаграмма без ограничений

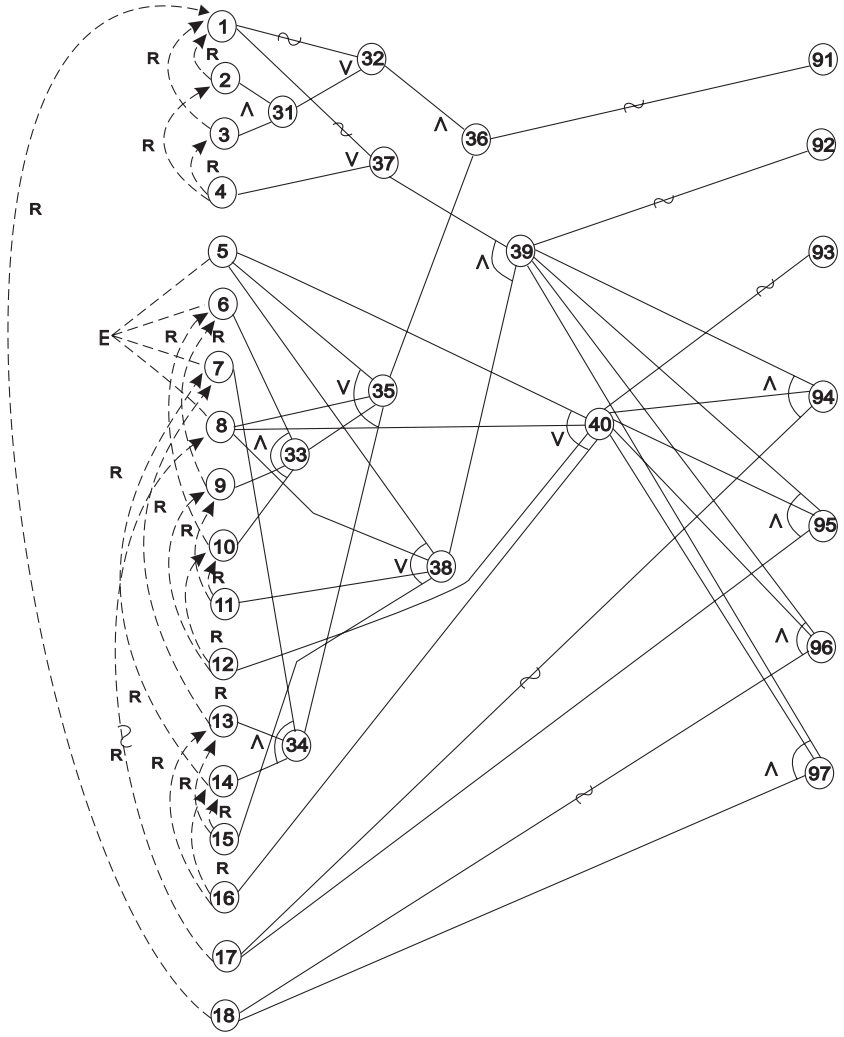


Рис. 12. Окончательная функциональная диаграмма команды ПОКАЗ

Таблица 4

Таблица решений (первая половина)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	0	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
4												1	1	0	0	1	1	1	1	1
5				0										1				1		
6	1	1	1	0	1	1	1				1	1			1	1				1
7				0				1	1	1			1				1			
8				0															1	
9	1	1	1		1	0	0				0	1			1	1				1
10	1	1	1		0	1	0				1	1			1	1				1
11												0			0	1				1
12																0				1
13								1	0	0			1				1			
14								0	1	0			1				1			
15													0							
16																	0			
17																		0	0	0
18																		1	1	1
91	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
92	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0
93	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
94	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
95	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
96	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
97	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

Таблица 5

Таблица решений (вторая половина)

	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
1	1	0	0	0	0	1	1	1	1	1	1	1	0	0	0	1	1	1
2	1					1	1	1	1	1	1	1				1	1	1
3	1					1	1	1	1	1	1	1				1	1	1
4	1					1	1	1	1	1	1	1				1	1	1
5		1				1				1			1			1		
6				1				1			1			1			1	
7	1				1				1			1			1			1
8			1				1											
9				1				1			1			1			1	
10				1				1			1			1			1	
11				1				1			1			1			1	
12				1				1			1			1			1	
13	1				1				1			1			1			1
14	1				1				1			1			1			1
15	1				1				1			1			1			1
16	1				1				1			1			1			1
17	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
18	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0
91	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
92	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
93	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
94	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
95	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
96	0	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	1
97	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0

19. ПОКАЗ 1 (94,97)
20. ПОКАЗ 21 — 29 (94,97)
21. ПОКАЗ 4021 .A (94,97)
22. ПОКАЗ — КОНЕЦ (94,96)
23. ПОКАЗ (94,96)
24. ПОКАЗ — F (94,96)
25. ПОКАЗ . E (94,96)
26. ПОКАЗ 7FF8 — КОНЕЦ (94,96)
27. ПОКАЗ 6000 (94,96)
28. ПОКАЗ A0 — A4 (94,96)
29. ПОКАЗ 20 .8 (94,96)
30. ПОКАЗ 7001 — КОНЕЦ (95,97)
31. ПОКАЗ 5 — 15 (95,97)
32. ПОКАЗ 4FF .100 (95,97)
33. ПОКАЗ — КОНЕЦ (95,96)
34. ПОКАЗ — 20 (95,96)
35. ПОКАЗ .11 (95,96)
36. ПОКАЗ 7000 — КОНЕЦ (95,96)
37. ПОКАЗ 4 — 14 (95,96)
38. ПОКАЗ 500 .11 (95,96)

Заметим, что в том случае, когда двум или более различным тестам соответствует один и тот же набор причин, следует стараться выбирать различные значения причин, с тем чтобы хотя бы незначительно улучшить результативность тестов. Заметим также, что из-за ограниченного размера памяти тест 22 является нереализуемым (при его использовании будет получено следствие 95 вместо 94, как отмечено в тесте 33). Следовательно, реализуемо только 37 тестов.

Замечания. Применение функциональных диаграмм — систематический метод генерации тестов, представляющих комбинации условий. Альтернативой является специальный выбор комбинаций, но при этом существует вероятность пропуска многих «интересных» тестов, определенных с помощью функциональной диаграммы.

При использовании функциональных диаграмм требуется трансляция спецификации в булевский граф. Следовательно, этот метод открывает перспективы его применения для обнаружения неполноты и неоднозначности в исходных спецификациях. Например, спецификация устанавливает, что все выходные строки содержат четыре слова, однако это справедливо не во всех случаях; так, для тестов 18 и 26 это неверно,

поскольку для них начальный адрес отображаемой памяти отличается от конечного адреса памяти машины менее чем на 16 байт.

Метод функциональных диаграмм позволяет построить набор полезных тестов, однако его применение обычно не обеспечивает построение всех полезных тестов, которые могут быть определены. Так, в нашем примере мы ничего не сказали о проверке идентичности отображаемых на экран терминала значений данных данным в памяти и об установлении случая, когда программа может отображать на экран любое возможное значение, хранящееся в ячейке памяти. Кроме того, функциональная диаграмма неадекватно исследует граничные условия. Конечно, в процессе работы с функциональными диаграммами можно попробовать покрыть граничные условия. Например, вместо определения единственной причины «адрес 2» $\geq$ «адрес 1» можно определить две причины «адрес 2» = «адрес 1» и «адрес 2» $>$ «адрес 1».

Однако при этом граф существенно усложняется и число тестов становится чрезвычайно большим. Поэтому лучше отделить анализ граничных значений от метода функциональных диаграмм. Например, для спецификации команды ПОКАЗ могут быть определены следующие граничные условия[1]:

1. «Адрес 1» длиной в одну цифру.
2. «Адрес 1» длиной в шесть цифр.
3. «Адрес 1» длиной в семь цифр.
4. «Адрес 1» = 0.
5. «Адрес 1» = 7FFF.
6. «Адрес 1» = 8000.
7. «Адрес 2» длиной в одну цифру.
8. «Адрес 2» длиной в шесть цифр.
9. «Адрес 2» длиной в семь цифр.
10. «Адрес 2» = 0.
11. «Адрес 2» = 7FFF.
12. «Адрес 2» = 8000.
13. «Адрес 2» = «адрес 1» .
14. «Адрес 2» = «адрес 1» + 1.
15. «Адрес 2» = «адрес 1» - 1.
16. «Число байт» длиной в одну цифру.
17. «Число байт» длиной в шесть цифр.

[1]Приведем их, компенсируя отсутствия примера при рассмотрении соответствующего критерия

18. «Число байт» длиной в семь цифр.
19. «Число байт» = 1.
20. «Адрес 1» + «число байт» = 8000.
21. «Адрес 1» + «число байт» = 8001.
22. Отображение шестнадцати байт (одна строка).
23. Отображение семнадцати байт (две строки).

Вовсе не означает, что следует писать 60 (37 + 23) тестов. Поскольку функциональная диаграмма дает только направление в выборе определенных значений операндов, граничные условия могут входить в полученные из нее тесты. В нашем примере, переписывая некоторые из первоначальных 37 тестов, можно покрыть все 23 граничных условия без дополнительных тестов. Таким образом, получаем небольшой, но убедительный набор тестов, удовлетворяющий поставленным целям.

Заметим, что метод функциональных диаграмм согласуется с некоторыми принципами тестирования, изложенными ранее. Его неотъемлемой частью является определение ожидаемого выхода каждого теста (все столбцы в таблице решений обозначают ожидаемые следствия). Заметим также, что данный метод помогает выявить ошибочные побочные следствия. Например, столбец (тест) 1 устанавливает, что должно присутствовать следствие 91 и что следствия 92—97 должны отсутствовать.

Наиболее трудным при реализации метода является преобразование диаграммы в таблицу решений. Это преобразование представляет собой алгоритмический процесс. Следовательно, его можно автоматизировать посредством написания соответствующей программы.

3.1.4. Предположение об ошибке

Некоторые люди обладают умением *выискивать* ошибки и без привлечения какой либо методологии тестирования. Объясняется это тем, что человек, обладающий практическим опытом, часто подсознательно применяет метод проектирования тестов, называемый предположением об ошибке. При наличии определенной программы он интуитивно предполагает вероятные типы ошибок и затем разрабатывает тесты для их обнаружения.

Процедуру для метода предположения об ошибке описать трудно, так как данный метод в значительной степени является интуитивным. Основная идея его заключается в том, чтобы перечислить в некотором списке возможные ошибки или ситуации, в которых они могут появиться, а затем на основе этого списка написать тесты. Предположения о

возможных ошибках могут появиться по разным причинам, но основные из них следующие:

— чтение документации о программе и выявление неясностей в ней;

— чтение спецификации и желание прояснения непонятных или двусмысленных положений;

— опыт тестировщика при тестировании подобных систем или некоторые общие для всех систем моменты.

Поскольку данная процедура не может быть четко определена, лучшим способом обсуждения смысла предположения об ошибке представляется разбор примеров. Если в качестве примера рассмотреть тестирование подпрограммы сортировки, то нужно исследовать следующие ситуации:

1. Сортируемый список пуст.
2. Сортируемый список содержит только одно значение.
3. Все записи в сортируемом списке имеют одно и то же значение.
4. Список уже отсортирован.

Другими словами, требуется перечислить те специальные случаи, которые могут быть не учтены при проектировании программы.

Для команды ПОКАЗ из предыдущего раздела целесообразно рассмотреть следующие тесты метода предположения об ошибке:

1. ПОКАЗ 100 — (неполный второй операнд).
3. ПОКАЗ 100 . (неполный второй операнд).
4. ПОКАЗ 000 — 0000FF (нули слева).

3.2. Критерии белого ящика

Тестирование по принципу белого ящика характеризуется степенью, в какой тесты выполняют или покрывают логику (исходный текст) программы. Как показано ранее, исчерпывающее тестирование по принципу белого ящика предполагает выполнение каждого пути в программе; но поскольку в программе с циклами выполнение каждого пути обычно нереализуемо, то тестирование всех путей не рассматривается как перспективное.

3.2.1. Критерии потока управления

При классификации критериев белого ящика выделяют понятия поток управления и поток данных в программах. Тестовые элементы определяются в терминах данных понятий и поэтому критерии подразде-

ляются на критерии потока управления и критерии потока данных. В книге Г. Майерса [2] присутствует описание только критериев потока управления, а именно:

- Покрытие операторов.
- Покрытие решений.
- Покрытие условий.
- Покрытие решений/условий.
- Комбинаторное покрытие условий.

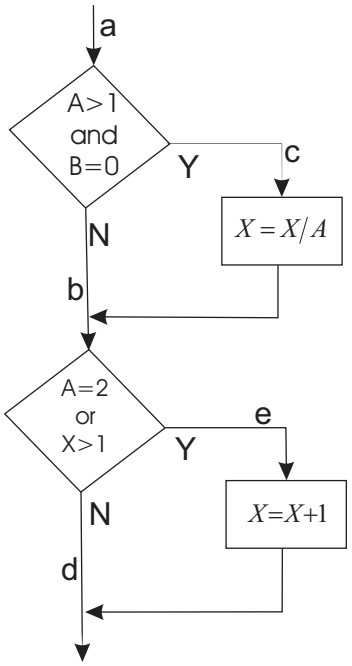


Рис. 13. Пример небольшой программы

Покрытие операторов. Если отказаться полностью от тестирования всех путей, то можно показать, что минимальным требованием к критериям покрытия потока управления является выполнение каждого оператора программы по крайней мере один раз (критерий покрытия операторов (или критерий C0)). К сожалению, это слабый критерий, так как выполнение каждого оператора хотя бы один раз есть необ-

ходимое, но недостаточное условие для приемлемого тестирования по принципу белого ящика. На рис. 13 представлен управляющий граф небольшой программы, которая должна быть протестирована. Исходный текст этой программы имеет следующий вид:

```
PROCEDURE M(A, B, X: REAL);
BEGIN
IF ((A > 1) AND (B = 0)) THEN
    X = X/A;
END;
IF ((A = 2) OR (X > 1)) THEN
    X = X+1;
END;
END M;
```

Можно выполнить каждый оператор, написав и исполнив единственный тест, который реализовал бы путь *ace*. Иными словами, если бы в точке *a* были установлены значения $A = 2$, $B = 0$ и $X = 3$, каждый оператор выполнялся бы один раз (в действительности X может принимать любое значение).

К сожалению, этот критерий хуже, чем кажется на первый взгляд. Например, пусть первое решение записано как *или*, а не как *и*[1]. При тестировании по данному критерию эта ошибка не будет обнаружена. Пусть во втором решении второе условие записано как $X > 0$; эта ошибка также не будет обнаружена. Кроме того, существует путь, в котором X не изменяется (путь *abd*). Если здесь ошибка, то и она не будет обнаружена. Таким образом, критерий покрытия операторов является настолько слабым, что его обычно не используют.

Критерий покрытия решений. Более сильным критерием среди критериев этого класса является критерий покрытия решений, или покрытия переходов (критерий покрытия дуг или критерий C1). Согласно данному критерию должно быть записано достаточное число тестов, такое, что каждое решение на этих тестах примет значение истина и ложь по крайней мере один раз (это для случая операторов с логическим выражением, т. е. для операторов IF, WHILE и REPET ... UNTIL, в случае же оператора CASE примет все возможные значения). Иными словами, каждое направление перехода должно быть реализовано по крайней мере один раз (отсюда покрытие дуг или покрытие переходов).

[1] Под решением понимается логическое выражение, стоящее, например, в условном операторе, а под условием — его простое подвыражение.

Можно показать, что покрытие решений обычно удовлетворяет критерию покрытия операторов. Поскольку каждый оператор лежит на некотором пути, исходящем либо из оператора перехода, либо из точки входа программы, при выполнении каждого направления перехода каждый оператор должен быть выполнен. Однако существуют следующие исключения. Первое — патологическая ситуация, когда данная ветвь программы никогда не выполняется. Второе встречается в программах или подпрограммах с несколькими точками входа; данный оператор может быть выполнен только в том случае, если выполнение программы начинается с соответствующей точки входа. Так как покрытие операторов считается необходимым условием, то покрытие решений, которое представляется более сильным критерием, должно включать покрытие операторов. Следовательно, покрытие решений требует, чтобы каждое решение имело результатом все возможные значения и при этом каждый оператор выполнялся бы по крайней мере один раз. Альтернативный и более легкий способ выражения этого требования состоит в том, чтобы каждое решение имело результатом все возможные значения и при вызове программы каждой точке входа должно быть передано управление по крайней мере один раз.

В программе, представленной на рис. 13, покрытие решений может быть выполнено двумя тестами, покрывающими либо пути *ace* и *abd*, либо пути *acd* и *abe*. Если мы выбираем последнее альтернативное покрытие, то входными значениями двух тестов являются $A = 3,\ B = 0,\ X = 3$ и $A = 2,\ B = 1,\ X = 1$.

Покрытие решений — более сильный критерий, чем покрытие операторов, но и он имеет свои недостатки. Например, путь, где X не изменяется (если выбрано первое альтернативное покрытие), будет проверен с вероятностью 50 %. Если во втором решении существует ошибка (например, $X < 1$ вместо $X > 1$), то ошибка не будет обнаружена двумя тестами предыдущего примера.

Критерий покрытия условий. Данный критерий требует создания такого числа тестов, чтобы все возможные результаты каждого условия в решении выполнялись по крайней мере один раз. Поскольку, как и при покрытии решений, это покрытие не всегда приводит к выполнению каждого оператора, к критерию требуется дополнение, которое заключается в том, что при вызове программы или подпрограммы каждой точке входа должно быть передано управление по крайней мере один раз.

Программа на рис. 13 имеет четыре условия: $A > 1$, $B = 0$, $A = 2$ и $X > 1$. Следовательно, требуется разработать такое число тестов, чтобы реализовать ситуации, $A > 1$, $A \leq 1$, $B = 0$ и $B \neq 0$ в точке a и $A = 2$, $A \neq 2$, $X > 1$ и $X \leq 1$ в точке b. Тесты, удовлетворяющие критерию покрытия условий, и соответствующие им пути:

1. $A = 2$, $B = 0$, $X = 4$ $\quad$ ace,
2. $A = 1$, $B = 1$, $X = 1$ $\quad$ abd.

Заметим, что, хотя аналогичное число тестов для этого примера уже было создано (с помощью предыдущего критерия), критерий покрытия условий обычно лучше покрытия решений, поскольку он может (но не всегда) вызвать выполнение условий в решениях, не реализуемых при покрытии решений.

Хотя применение критерия покрытия условий на первый взгляд удовлетворяет критерию покрытия решений, это не всегда так. Хорошим примером это демонстрирующим является оператор IF (A & B) THEN ... ELSE ... END, для выполнения критерия покрытия условий требуется два теста — A есть *истина*, B есть *ложь* и A есть *ложь*, B есть *истина*. Но в этом случае не выполняется THEN часть оператора IF и, следовательно, не выполняется критерий покрытия решений. Тесты критерия покрытия условий для ранее рассмотренного примера покрывают результаты всех решений, но это только случайное совпадение. Например, два альтернативных теста $A = 1$, $B = 0$, $X = 3$ и $A = 2$, $B = 1$, $X = 1$ покрывают результаты всех условий, но только два из четырех результатов решений (они оба покрывают путь abe и, следовательно, не выполняют результат *истина* первого решения и результат *ложь* второго решения).

Критерий покрытия решений/условий. Разрешением данной проблемы является критерий, названный покрытием решений/условий. Он требует такого набора тестов, чтобы все возможные результаты каждого условия в решении выполнялись по крайней мере один раз, все результаты каждого решения выполнялись по крайней мере один раз и каждой точке входа передавалось управление по крайней мере один раз.

Недостатком критерия покрытия решений/условий является невозможность его применения для выполнения всех результатов всех условий; часто это имеет место, вследствие того что определенные условия скрыты другими условиями.

Причина этого заключается в том, что результаты условий в выражениях *и* и *или* могут скрывать и блокировать действие других условий. Например, если условие *и* есть *ложь*, то никакое из последующих условий в выражении не будет выполнено. Аналогично если условие *или* есть *истина*, то никакое из последующих условий не будет выполнено, т. е. если для вычисления значения всего выражения (в данных терминах решения) достаточно полученного значения его подвыражения (в данном случае условия), то дальнейшее вычисление выражения проводится не будет. Следовательно, критерии покрытия условий и покрытия решений/условий недостаточно чувствительны к ошибкам в логических выражениях.

Комбинаторное покрытие условий. Критерием, который решает эти проблемы, является комбинаторное покрытие условий. Он требует создания такого числа тестов, чтобы все возможные комбинации результатов условия в каждом решении и все точки входа выполнялись по крайней мере один раз. Легко видеть, что набор тестов, удовлетворяющий критерию комбинаторного покрытия условий, удовлетворяет также и критериям покрытия решений, покрытия условий и покрытия решений/условий.

По этому критерию в программе на рис. 13 должны быть покрыты тестами следующие восемь комбинаций:

1) $A > 1,\ B = 0$,
2) $A > 1,\ B \neq 0$,
3) $A \leq 1,\ B = 0$,
4) $A \leq 1,\ B \neq 0$,
5) $A = 2,\ X > 1$,
6) $A = 2,\ X \leq 1$,
7) $A \neq 2,\ X > 1$,
8) $A \neq 2,\ X \leq 1$.

Заметим, что комбинации 5—8 представляют собой значения второго оператора IF. Поскольку X может быть изменено до выполнения этого оператора, то значения, необходимые для его проверки, следует восстановить исходя из логики программы.

Для того чтобы протестировать эти комбинации, необязательно использовать все восемь тестов. Фактически они могут быть покрыты четырьмя тестами. Приведем входные значения тестов и комбинации, которые они покрывают:

$A = 2,\ B = 0,\ X = 4$ покрывает 1,5;
$A = 2,\ B = 1,\ X = 1$ покрывает 2,6;
$A = 1,\ B = 0,\ X = 2$ покрывает 3,7;
$A = 1,\ B = 1,\ X = 1$ покрывает 4,8.

То, что четырем тестам соответствуют четыре различных пути на рис. 13, является случайным совпадением. На самом деле представленные выше тесты не покрывают всех путей, они пропускают путь *acd*. Например, для тестирования следующей программы:

IF ((X = Y) & (LENGTH(Z) = 0) & END) THEN J = l; ELSE I = 1; END;

требуется восемь тестов, хотя она покрывается двумя путями. В случае циклов число тестов для удовлетворения критерию комбинаторного покрытия условий обычно больше, чем число путей.

Таким образом, для программ, содержащих только одно условие на каждое решение, минимальным является критерий, набор тестов которого: 1) вызывает выполнение всех результатов каждого решения по крайней мере один раз и 2) передает управление каждой точке входа по крайней мере один раз (чтобы обеспечить выполнение каждого оператора программы по крайней мере один раз). Для программ, содержащих решения, каждое из которых имеет более одного условия, минимальный критерий состоит из набора тестов, вызывающих выполнение всех возможных комбинаций результатов условий в каждом решении и передающих управление каждой точке входа программы по крайней мере один раз.

3.2.2. Критерии потока данных

Для определения потоковых критериев нам необходимо дать определения def/use графа и дополнительно двух множеств. Для иллюстрации вводимых понятий мы определим простой язык программирования. Основные определения и понятия не зависят от языка программирования, данный простой язык и его синтаксис позволит нам упростить изложение примеров и определений[1]. Язык программирования разрешает только простые переменные и содержит следующие операторы:

Начальный оператор: **start**.

Оператор ввода: **read** $x_1, \cdots, x_n$,
где $x_1, \cdots, x_n$ — переменные.

Оператор присваивания: $y := f(x_1, \cdots, x_n)$,
где f — n-арная функция ($n \geq 0$), а $y, x_1, \cdots, x_n$ — переменные.

Оператор вывода: **print** $e_1, \cdots, e_n$,
где e_i $(i = 1, \cdots, n)$ — переменная или константа.

[1]Материал для лекций по семейству потоковых критериев основан на статье S. Rapps E. Weyuker [13].

Оператор безусловного перехода: **goto m**,
где **m** — метка.

Оператор условного перехода: **if** $\mathrm{p}(x_1, \cdots, x_n)$ **then goto m**,
где p — n-арный предикат ($n > 0$), $x_1, \cdots, x_n$ — переменные, и **m** — метка. Заметим, что 0-арные предикаты, такие как **TRUE** и **FALSE** запрещены.

Конечный оператор (останов): **stop**.

Каждый оператор может быть помечен уникальным идентификатором — меткой.

Будем использовать следующие обозначения и определения:

— ребро управляющего графа соединяющее вершины i и j — (i, j);

— путь это конечная последовательность вершин $(n_1, \cdots, n_k)$, $k \geq 2$ такая, что существует ребро от n_i к n_{i+1} для $i = 1, \cdots, k-1$;

— полный путь — путь первая вершина которого соответствут начальной вершине графа (оператору **start**), а последняя вершина соответствует конечной вершине (оператор **stop**).

Каждое вхождение переменной в программу будем классифицировать как определяющее (def) или использующее. Использующее вхождение, в свою очередь, будем классифицировать как вычислительное (c-use) или предикатное (p-use).

Оператор присваиваивания $y := f(x_1, \cdots, x_n)$ содержит определяющее вхождение переменной y и вычислительное использование переменных $x_1, \cdots, x_n$.

Оператор ввода **read** $x_1, \cdots, x_n$ содержит определяющее вхождение переменных $x_1, \cdots, x_n$.

Оператор вывода **print** $x_1, \cdots, x_n$ содержит вычислительное использование переменных $x_1, \cdots, x_n$.

Оператор условного перехода **if** $\mathrm{p}(x_1, \cdots, x_n)$ **then goto m** содержит предикатное использование переменных $x_1, \cdots, x_n$.

Вершина управляющего графа содержит вычислительное или определяющее использование переменной если операторы вершины содержат def или c-use использование данной переменной.

Для пояснения всех определений приведем пример простой программы на предложенном языке:

```
start
read   x,y
if     y < 0 then goto A
pow := y
```

```
        goto   B
A       pow := -y
B       z   := 1
C       if     pow = 0 then goto D
        z   := z*x
        pow := pow-1
        goto   C
D       if     y >= 0 then goto E
        z   := 1/z
E       answer := z+1
        print answer
        stop
```

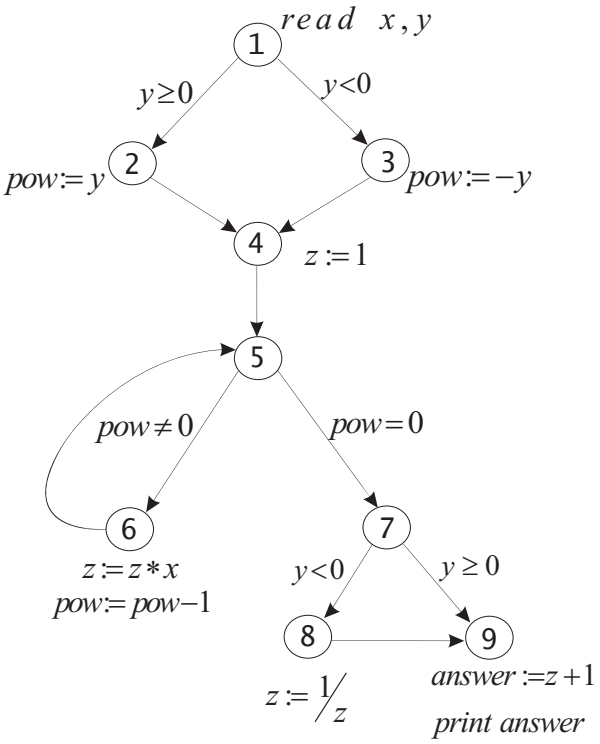


Рис. 14. Управляющий граф программы

На рис. 14 представлен управляющий граф для этой программы и, например, вершина 6 содержит def и c-use использование переменных z и pow и только вычислительное использование переменной x.

Будем различать глобальное и локальное вычислительное использование переменной. Вычислительное использование (c-use) переменной x называется глобальным, если она вычислительно используется не в той вершине в которой определена. Иначе, т. е. если c-use переменной находится в той же вершине, что и def, имеем локальное вычислительное использование переменной. Локальные вычислительные использования для нас не интересны, так как если исполнение теста достигло какой-то вершины графа, то все операторы в этой вершине обязательно исполнятся. Нас интересует глобальный поток данных.

Оператор условного перехода всегда является последним оператором вершины и имеет два преемника — две различные вершины. Значения переменных, входящих в предикат условного оператора, прямо определяют, какая из вершин преемников будет исполняться, поэтому мы предикатное использование будем связывать с ребром, а не с вершиной, в которой условие расположено. Если последним оператором вершины i является оператор «**if** $\mathrm{p}(x_1, \cdots, x_n)$ **then goto m**» и вершины j и k — преемники вершины i, то ребра (i, j) и (i, k) содержат предикатное использование (p-use) переменных $x_1, \cdots, x_n$. Ребра (5,6) и (5,7) на рис. 14 содержат p-use переменной pow.

Пусть x — переменная, входящая в программу. Путь $(i, n_1, \cdots, n_m, j)$, $m \geq 0$, не содержащий определения x в вершинах $n_1, \cdots, n_m$, называется путем без переопределения переменной x от вершины i до вершины j. Путь $(i, n_1, \cdots, n_m, j, k)$, $m \geq 0$, не содержащий определения x в вершинах $n_1, \cdots, n_m, j$, называется путем без переопределения переменной x от вершины i до ребра (j, k). Ребро (i, j) является путем без переопределения переменной x от вершины i до ребра (i, j). Определение переменной x в вершине i является глобальным определением, если оно является последним определением x в данной вершине и существует путь без переопределения переменной x от вершины i до или вершины, содержащей глобальное вычислительное использование переменной x, или ребра, содержащего ее предикатное использование. Определение переменной x в вершине i является локальным определением, если существует локальное вычислительное использование x в данной вершине, которое расположено после ее определения, и нет другого определения x

между ними. Определение переменной «*answer*» в вершине 9 на рис. 14 является локальным.

Определим *def/use граф* как управляющий граф, в котором с каждым ребром связано множество, а с каждой вершиной — два множества. Определим их: *def(i)* — множество переменных, которые имеют глобальное определение в вершине i; *c-use(i)* — множество переменных, которые имеют глобальное вычислительное использование в вершине i; *p-use(i, j)* — множество переменных, которые предикатно используются на ребре (i, j). Ребро (i, j) для которого множество *p-use(i, j)* не пусто называется помеченным ребром, иначе (i, j) — непомеченное ребро. Таким образом, *def/use* граф — это управляющий граф, в котором для каждой вершины i вычислены множества *def(i)* и *c-use(i)*, а для каждого ребра (i, j) — *p-use(i, j)*.

Для управляющего графа, приведенного на рис. 14, эти множества представлены в табл. 6.

Таблица 6

Множества *def, c-use* и *p-use* для УГ, представленного на рис. 14

Вершина	**c-use**	**def**	**Ребро**	**p-use**
1	∅	{ x,y }	(1, 2)	{ y }
2	{ y }	{ pow }	(1, 3)	{ y }
3	{ y }	{ pow }	(5, 6)	{ pow }
4	∅	{ z }	(5, 7)	{ pow }
5	∅	∅	(7, 8)	{ y }
6	{ x,z,pow }	{ z,pow }	(7, 9)	{ y }
7	∅	∅		
8	{ z }	{ z }		
9	{ z }	∅		

Замечание. Переменная «*answer*», которая имеет только локальное определение и локальное вычислительное использование, не присутствует в этих множествах. Ребра (2, 4), (3, 4), (4, 5), (6, 5) и (8, 9) не помечены.

Для формулировки критериев нам потребуется определение еще двух множеств. Пусть i — некоторая вершина и x — переменная, такая, что $x \in def(i)$. Тогда $dcu(x, i)$ — множество всех вершин j, таких, что $x \in$ c-use(j) и для которых существует путь без переопределения переменной x от i до j; $dpu(x, i)$ — множество всех ребер (j, k), таких, что

$x \in$ p-use(j,k) и для которых существует путь без переопределения переменной от i до j. Значения этих множеств для графа на рис. 14 следующие:

$dcu(x,1) = \{6\}$, $dpu(x,1) = \emptyset$,
$dcu(y,1) = \{2,3\}$, $dpu(y,1) = \{(1,2),(1,3),(7,8),(7,9)\}$,
$dcu(pow,2) = \{6\}$, $dpu(pow,2) = \{(5,6),(5,7)\}$,
$dcu(pow,3) = \{6\}$, $dpu(pow,3) = \{(5,6),(5,7)\}$,
$dcu(z,4) = \{6,8,9\}$, $dpu(z,4) = \emptyset$,
$dcu(z,6) = \{6,8,9\}$, $dpu(z,6) = \emptyset$,
$dcu(pow,6) = \{6\}$, $dpu(pow,6) = \{(5,6),(5,7)\}$,
$dcu(z,8) = \{9\}$, $dpu(z,8) = \emptyset$.

Пусть P — множество полных путей def/use графа программы. Мы говорим, что вершина i содержится в P (или P покрывает i), если P содержит путь $(n_1, \cdots, n_m)$, такой, что $i = n_j$ для некоторого j, $1 \le j \le m$. Ребро (i_1, i_2) содержится в P (или покрывается), если P содержит путь $(n_1, \cdots, n_m)$, такой, что $i_1 = n_j$, $i_2 = n_{j+1}$ для некоторого j, $1 \le j \le m-1$. Путь $i_1, \cdots, i_k$ содержится в P(покрывается P), если P содержит путь $(n_1, \cdots, n_m)$ и $i_1 = n_j, i_2 = n_{j+1}, \cdots, i_k = n_{j+k-1}$ для некоторого j, $1 \le j \le m-k+1$. Мы говорим, что P исполнимо, если каждый путь, содержащийся в P, проходится в процессе исполнения программы на множестве входных данных. Путь $(n_1, \cdots, n_j, n_k)$ называется *du-путем* относительно переменной x, если n_1 содержит глобальное определение переменной x и выполняется одно из двух условий: 1) n_k содержит вычислительное использование переменной x и $(n_1, \cdots, n_j, n_k)$ — путь без переопределения переменной x; 2) (n_j, n_k) содержит предикатное использование переменной x и $(n_1, \cdots, n_j)$ — путь без переопределения переменной x и без циклов.

Семейство критериев. Пусть G — def/use граф и P — множество полных путей G. Тогда:

• P удолетворяет критерию покрытия операторов (*all-nodes*), если каждая вершина G содержится в P (покрывается P).

• P удолетворяет критерию покрытия дуг (*all-edges*), если каждое ребро G покрывается P.

• P удолетворяет критерию *all-defs*, если для каждой вершины i графа G и для каждой переменной $x \in def(i)$ P содержит путь без переопределения переменной x от i до некоторого элемента $dcu(x,i)$ или $dpu(x,i)$.

• P удолетворяет критерию *all-p-uses*, если для каждой вершины i

графа G и для каждой переменной $x \in def(i)$ P содержит путь без переопределения переменной x от i до всех элементов $dpu(x,i)$.

• P удолетворяет критерию *all-c-uses/some-p-uses*, если для каждой вершины i графа G и для каждой переменной $x \in def(i)$ P содержит некоторый путь без переопределения переменной x от i до каждой вершины из $dcu(x,i)$, а если $dcu(x,i)$ пусто, то P должно содержать путь без переопределения переменной x от i до некоторого ребра из $dpu(x,i)$. Этот критерий требует, чтобы каждое вычислительное использование переменной x, определенной в вершине i покрывалось некоторым путем из P. Если нет такого вычислительного использования, то некоторое предикатное использование определения x в вершине i должно быть покрыто.

• P удолетворяет критерию *all-p-uses/some-c-uses*, если для каждой вершины i графа G и для каждой переменной $x \in def(i)$ P содержит путь без переопределения переменной x от i до всех элементов $dpu(x,i)$, а если $dpu(x,i)$ пусто, то P должно содержать путь без переопределения переменной x от i до некоторой вершины из $dcu(x,i)$.

• P удолетворяет критерию *all-uses*, если для каждой вершины i графа G и для каждой переменной $x \in def(i)$ P содержит путь без переопределения переменной x от i до всех элементов $dcu(x,i)$ и до всех элементов $dpu(x,i)$.

• P удолетворяет критерию *all-du-path*, если для каждой вершины i графа G и для каждой переменной $x \in def(i)$ P содержит каждый du-путь относительно переменной x.

• P удолетворяет критерию *all-path*, если P содержит все полные пути графа G.

Для сравнения определенных критериев семейства введем понятия: «критерий c_1 включает критерий c_2», «критерий c_1 строго включает критерий c_2» и несравнимость критериев. Критерий c_1 *включает критерий* c_2, если для любого def/use графа G любое множество полных путей этого графа G, которое удовлеворяет c_1 также удовлетворяет и c_2. Критерий c_1 *строго включает критерий* c_2, обозначается $c_1 \Rightarrow c_2$, если c_1 включает c_2 и для некоторого def/use графа g существует множество полных путей, которое удовлетворяет критерию c_2, но не c_1. Заметим, что это транзитивное отношение. Критерии c_1 и c_2 несравнимы, если неверно как $c_1 \Rightarrow c_2$, так и $c_2 \Rightarrow c_1$. Семейство критериев можно частично упорядочить по включению, и приводимая ниже теорема задает этот порядок.

Теорема. Критерий c_i строго включает критерий c_j тогда и только тогда, если это соответствует рис. 15 с учетом транзитивности этого отношения.

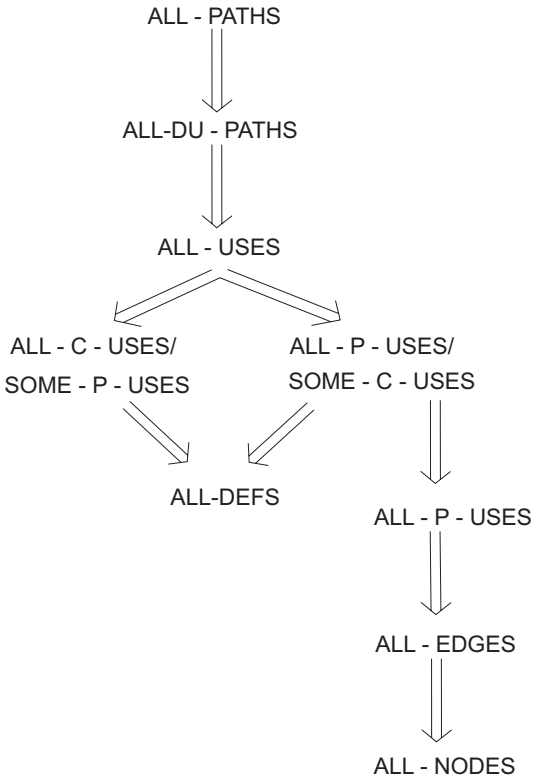


Рис. 15. Схема строгого включения семейства критериев

Доказательство данной теоремы мы опустим, оно приведено в работе [13].

Замечание. Все приведенные критерии можно переформулировать в традиционной манере. Приведем такие формулировки только для двух наиболее известных потоковых критерия.

Критерий **all-defs** требует создания такого колличества тестов, чтобы каждое определение любой переменной достигало своего использования (естественно, при исполнении этого набора и по пути не переопределялось).

Критерий **all-uses** требует создания такого колличества тестов, чтобы каждое определение любой переменной достигало всех своих использований (естественно, при исполнении этого набора и по пути не переопределялось).

Рассмотрим пример, показывающий важность использования потоковых критериев для построения тестов.

Пример. Программа данного примера вычисляет $\sqrt{p}$, $0 \leq p < 1$ с точностью e, $0 < e \leq 1$.

```
        start
        read   P,E
        D   := 1
        X   := 0
        C   := 2*P
        if     C >= 2 then goto MD
MA      if     D =< E then goto MC
        D   := D/2
        T   := C-(2*X+D)
        if     T < 0 then goto MB
        X   := X+D
        C   := 2*(C-(2*X+D))
        goto   MA
MB      C   := 2*C
        goto   MA
MC      print  X
        stop
MD      print  'ERROR'
        stop
```

Программа содержит ошибку, операторы в вершине 5 должны быть переставлены местами (рис. 16).

Множество путей {(1, 6), (1, 2, 3, 4, 2, 3, 5, 2, 7)} удовлетворяет критерию покрытия дуг (all-edges), но ошибку не обнаруживают. Проблема в том, что определение переменной C в вершине 5 нигде не используется, пока множество путей не включают путь без переопределения переменной C от вершины 5 к вершине 3. Итак, мы не сможем обнаружить ошибку, пока путь (5, 2, 3) не будет добавлен. Критерия all-p-uses также не достаточно для обнаружения ошибки, так как {(1, 6), (1, 2, 7), (1, 2, 3, 5, 2, 7), (1, 2, 3, 4, 2, 3, 4, 2, 7)} удовлетворяют данному критерию, но не содержат (5, 2, 3).

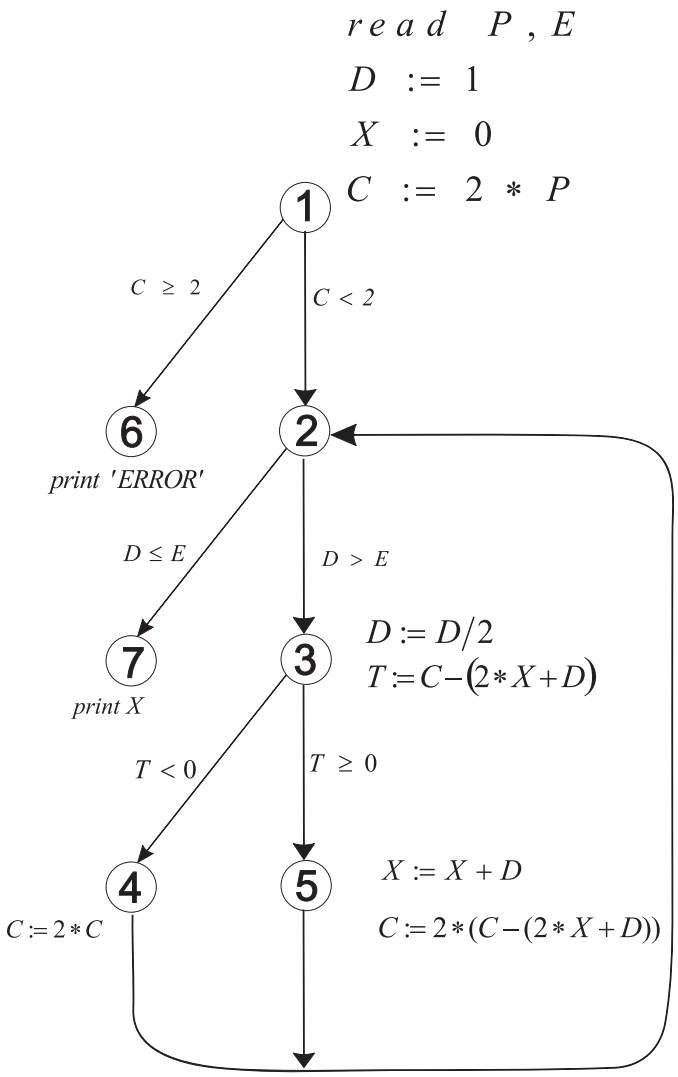


Рис. 16. Управляющий граф для приведенной программы

Критерий all-defs требует, чтобы каждое определение переменной достигало своего использования, и поэтому любое множество путей удовлетворяющее данному критерию должно включать (5, 2, 3). Множество путей {(1, 2, 3, 5, 2, 3, 5, 2, 7), (1, 2, 3, 4, 2, 3, 4, 2, 7)} удовлетворяет критерию all-defs и должно выявлять ошибку, но вершина 6 и ребро (1, 6) не проверяются. И любые проблемы в этой вершине и ребре не будут обнаружены. Более того, это множество не содержит путь (1, 2, 7), который будет исполняться если входные данные некорректные и $E > 1$.

Критерий all-c-uses/some-p-uses требует присутствия в множестве путей от каждого определения переменной к каждому возможному вычислительному использованию этого определения. Для рис. 16 это означает, что любое множество путей, удовлетворяющее критерию all-c-uses/some-p-uses, должно включать пути (4, 2, 3, 4), (4, 2, 3, 5), (5, 2, 3, 4), (5, 2, 3, 5). Однако данный критерий не включает даже критерий покрытия дуг. Так, множество путей {(1, 2, 3, 5, 2, 3, 5, 2, 7), (1, 2, 3, 4, 2, 3, 5, 2, 7), (1, 2, 3, 4, 2, 3, 4, 2, 7), (1, 2, 3, 5, 2, 3, 4, 2, 7)} удовлетворяет критерию all-c-uses/some-p-uses, но не включают ребро (1, 6) и путь (1, 2, 7) и может не обнаружить возможные проблемы, связанные с данными областями программы.

Так как программа не имеет предикатного использования для определения переменной C в вершинах 4 и 5, то для выполнения критерия all-p-uses/some-c-uses требуется, чтобы пути (5, 2, 3) и (4, 2, 3) содержались в искомом множестве путей. Поэтому множество путей этого критерия будет обнаруживать ошибку. Также ребро (1, 6) должно быть включено, и, следовательно, возможные проблемы могут быть обнаружены. Дополнительно путь (1, 2, 7) должен быть включен и возможные потенциальные проблемы тоже должны быть выявлены. Однако некоторые комбинации итогов предикатов могут быть не включены, например такие, как (5, 2, 3, 4) и (4, 2, 3, 5), и проблемы, связанные с этими путями могут, быть не обнаружены.

Критерий all-uses включает критерии all-p-uses/some-c-uses и all-c-uses/some-p-uses и поэтому может обнаружить все описанные ранее ошибки. Вот одно из множеств путей, удовлетворяющих этому критерию {(1, 6), (1, 2, 3, 5, 2, 3, 5, 2, 7), (1, 2, 7), (1, 2, 3, 4, 2, 3, 5, 2, 7), (1, 2, 3, 4, 2, 3, 4, 2, 7), (1, 2, 3, 5, 2, 3, 4, 2, 7)}.

Определенные ранее критерии достаточно хорошо тестируют программы, но не полностью учитывают имеющуюся специфику определенных конструкций языков программирования.

Одним из наиболее важных операторов языков прграммирования является оператор цикла. Поэтому появляется необходимость в следующем критерии, который учитывает наличие циклов в программе.

Критерий покрытия циклов(all-loops). Критерий покрытия циклов (или all-loops) требует построения такого набора тестов, для которого все циклы исполнялись бы 0, 1 и более одного раза.

Имеется в виду, что цикл исполняется m раз, если управление достигает входа в цикл, и затем, не выходя из тела цикла, ровно m раз проходится обратная дуга. Отметим, что цикл исполняется 0 раз не в том случае, когда он не лежит на пути исполнения программы, а лишь тогда, когда упраление достигает входа этого цикла и затем выходит из цикла, не проходя через обратную дугу.

Так как циклы предназначены обычно для обработки каких-нибудь структур данных (входных, выходных или промежуточных), то набор тестов, удовлетворяющий критерию all-loops, будет соответствовать тестам, проверяющим различные классы эквивалентности для этих структур. При этом проверяются также и граничные условия. (Имеются в виду критерии «черного ящика».)

Например, если переменная l — список и в программе встречается цикл типа

```
WHILE  l<>NIL  DO
    . . . (* обработать очередной элемент списка *)
    l := l^.next;
END;
```

то требование, чтобы этот цикл исполнялся 0, 1 и несколько раз, эквивалентно требованию, чтобы были подобраны такие тесты, что l — пустой, одноэлементный и многоэлементный список.

Если в программе есть циклы, исполняющиеся постоянное число раз, то они исключаются из множества циклов, рассматривающихся критерием all-loops.

Если данные на входе цикла такие, что этот цикл может исполниться не меньше min раз и не больше max раз, то условие покрытия этого цикла (в соответствии со стратегией проверки граничных условий)

нужно изменить и потребовать, чтобы он исполнялся min, $min + 1$, $min + 1 < k < max - 1$, $max - 1$ и max раз.

Однако обычно, при автоматическом построении тестов или при проверке полноты по критерию all-loops, последние оговорки не учитываются и предполагается, что любой цикл исполняется 0, 1 и произвольное число раз.

Критерий покрытия рекурсивных процедур. Другой важной языковой конструкцией является процедура. Мы рассмотрим критерий покрытия рекурсивных процедур — аналог критерия покрытия циклов. Данный критерий требует, чтобы множество тестов обеспечивало исполнение рекурсивной процедуры с глубиной рекурсии 0, 1 и k, где k>1 [1].

3.3. Формальное определение критериев тестирования

Пусть $\mathcal{P}$ — множество всех программ, и пусть P — программа из $\mathcal{P}$ с множеством входных данных In и множеством выходных данных Out. Набор тестов T для программы P — это некоторое подмножество входных данных: $T \subseteq In$.

Критерий тестирования C — это такой предикат, что

$$C(P,T) \iff T \text{ достаточно для тестирования } P \text{ согласно данному критерию.}$$

Такое определение не очень полезно для целей построения набора тестов, удовлетворяющего критерию, поэтому определим C с помощью множества требуемых элементов и понятия покрытия.

Итак, пусть для некоторого критерия C задано множество требуемых элементов $REQ_C(P)$ (функция, определенная на $\mathcal{P}$) и предикат COV_C, определенный на множестве $\mathcal{P} \times In \times REQ_C$ с неформальной семантикой:

$$COV_C(P,x,r) \iff \text{при исполнении программы } P \text{ на входных данных } x \text{ требуемый элемент } r \text{ покрывается.}$$

Теперь можно определить сам предикат C:

$$C(P,T) \Longleftrightarrow \forall\, r \in REQ_C(P)\ \ \exists\, t \in T : COV_C(P,t,r).$$

[1] В разделе, посвященном описанию системы ОСТ, мы рассмотрим еще несколько критериев для процедур, правда, эти критерии разработаны специально для данной системы и ранее не использовались.

Иногда множество REQ определяется так, что в него попадают нереализуемые требуемые элементы, т. е. такие $r \in REQ_C(P)$, что

$$\neg\exists\, x \in In : COV_C(P, x, r),$$

поэтому более точное определение выглядит следующим образом:

$$\begin{aligned} C(P,T) \iff & \forall\, r \in REQ_C(P)\ \&\ (\exists\, x \in In : COV_C(P,x,r)) \\ & \exists\, t \in T : COV_C(P,t,r). \end{aligned}$$

Определим теперь операции объединения $+$ и перемножения $\times$ критериев. Пусть C_1 и C_2 — критерии с соответствующими REQ_1, COV_1 и REQ_2, COV_2. Тогда $C = C_1 + C_2$, если

$$\begin{aligned} REQ_C(P) &= REQ_1(P) \cup REQ_2(P), \\ COV_C(P,x,r) &\iff (r \in REQ_1(P)\ \&\ COV_1(P,x,r)) \vee \\ &\quad\ (r \in REQ_2(P)\ \&\ COV_2(P,x,r)); \end{aligned}$$

$C = C_1 \times C_2$, если

$$\begin{aligned} REQ_C(P) &= REQ_1(P) \times REQ_2(P), \\ COV_C(P,x,(r_1,r_2)) &\iff COV_1(P,x,r_1)\ \&\ COV_2(P,x,r_2). \end{aligned}$$

В дальнейшем будем определять критерии тестирования в основном с помощью REQ и COV. При этом из того, какая информация о программе и ее исполнении используется при определении критериев, можно выделить два класса критериев: *критерии функционального тестирования* и *критерии структурного тестирования*, которые и рассматриваются далее.

3.3.1. Критерии функционального тестирования

Критерии функционального тестирования строятся из предположения, что чем более разнообразные входные данные выбираются, а выходные данные получаются при исполнении программы, тем больше вероятность обнаружения ошибок. Таким образом, при определении критериев функционального тестирования программа рассматривается как функция из In в Out и ее внутренняя структура игнорируется.

В качестве примера критерия функционального тестирования рассмотрим критерий, получающийся из наиболее часто используемой стратегии функционального тестирования — стратегии разбиения на классы

эквивалентностей, при которой входные данные разбиваются на непересекающиеся подмножества:

$$In = In_1 \cup In_2 \cup \cdots \cup In_n$$

и требуется выбрать тест из каждого подмножества:

$$\begin{aligned} REQ(P) &= \{In_1,\ In_2,\ \ldots,\ In_n\}, \\ COV(P,x,r) &\iff x \in r. \end{aligned}$$

Аналогичные критерии можно определить для Out и для $In \times Out$.

3.3.2. Критерии структурного тестирования

В отличие от критериев функционального тестирования, критерии структурного тестирования учитывают внутреннюю структуру программы, т. е. тот факт, что результат вычисления получается не сразу, а только через некоторое число шагов. Кроме этого, программа имеет некоторые состояния, путем изменения которых и производятся вычисления.

Таким образом, критерии структурного тестирования строятся из предположения, что чем в более разнообразных состояниях побывает программа, тем больше вероятность обнаружения ошибок. Кроме того, для некоторых видов ошибок можно выделить состояния в программе, которые могут привести к обнаружению этих ошибок.

Итак, при определении критериев структурного тестирования ключевым является понятие состояния программы, т. е. состояние вычисления в некоторый момент времени. В качестве состояния вычисления можно рассматривать разные параметры: текущий исполняемый оператор, значения всех переменных, текущий стек процедур и т. д. В зависимости от того, какая часть информации о состоянии программы рассматривается, можно выделить два класса критериев: *графовые* (состояние — текущий исполняемый оператор) и *логические* (состояние — предикат от переменных и/или выражений).

Графовые критерии структурного тестирования

Пусть $G = (V, E)$ — управляющий граф программы P, причем программа разбита на процедуры, т. е. имеется множество процедур $PROC$

и G является объединением непересекающихся управляющих графов $G_{pr} = (V_{pr}, E_{pr})$ процедур из $PROC$:

$$V = \bigcup_{pr \in PROC} V_{pr}, \qquad E = \bigcup_{pr \in PROC} E_{pr}.$$

При этом каждый граф G_{pr} имеет единственный вход $entry(pr)$ и единственный выход $exit(pr)$.

Вершину v графа G с дополнительной информацией о соответствующем вычислении в программе назовем *оператором программы*. В операторе программы могут быть вызовы процедур. Через $CALL(v)$, будем обозначать множество процедур, вызываемых в операторе v, естественно $CALL(v) \subseteq PROC$. Будем считать, что при исполнении оператора программы сначала вызываются процедуры в операторе v, а затем исполняется сам оператор v, т. е. *путь исполнения* процедуры pr имеет вид ('+' означает конкатенацию последовательностей)

$$run(pr) = entry(pr) + \sum_{i=1}^{n} (r_i + q_i) + exit(pr),$$

где $r_i = \sum_{p \in CALL(q_i)} run(p)$, $q_i \in V_{pr}$, $q = [entry(pr), q_1, \ldots, q_n, exit(pr)]$ — это путь в графе G_{pr}.

В дальнейшем будем использовать следующее множество путей:

$$PT(run(pr)) = \{q\} \cup PT(r_1) \cup \cdots \cup PT(r_n),$$

которое определяется для некоторого пути исполнения $run(pr)$ процедуры pr указанного выше вида и является множеством путей в соответствующих управляющих графах.

Из множества процедур выделим главную процедуру $p_0 \in PROC$, с которой начинается исполнение программы, и будем обозначать через $run(P, x)$ путь исполнения процедуры p_0, по которому проходит выполнение программы P на входных данных $x \in In$.

Предположим теперь, что в некотором операторе $v \in V$ программы имеется ошибка. Тогда, для того чтобы набор тестов мог обнаружить эту ошибку, должен выполняться по крайней мере следующий критерий.

Критерий покрытие операторов

$$\begin{aligned} REQ_{C_0}(P) &= V, \\ COV_{C_0}(P, x, v) &\iff v \in run(P, x). \end{aligned}$$

Далее будем считать, что последовательность $q = [q_1, \ldots, q_s]$ является подпоследовательностью $r = [r_1, \ldots, r_n]$, и писать $q \subseteq r$, если

$$\exists\, i \in [1..n-s+1] \;\; \forall\, j \in [1..s] \;\; q_j = r_{i+j-1}.$$

Следующие критерии включают критерий C_0 и предназначены для того, чтобы заставить набор тестов исполнять программу по достаточно разнообразным путям.

Критерий покрытие дуг

$$\begin{aligned} REQ_{C_1}(P) &= E, \\ COV_{C_1}(P, x, (v_1, v_2)) &\iff \exists\, q \in PT(run(P, x)) : [v_1, v_2] \subseteq q \end{aligned}$$

((v_1,v_2) — дуга управляющего графа E, $[v_1,v_2]$ — участок пути).

Критерий покрытие всех путей длины n

$$\begin{aligned} REQ_{C_n}(P) &= \{(v_1, \ldots, v_{n+1}) | (v_i, v_{i+1}) \in E \text{ для } 1 \le i \le n\}, \\ COV_{C_n}(P, x, r) &\iff \exists\, q \in PT(run(P, x)) : r \subseteq q. \end{aligned}$$

Критерий CP (покрытие пар дуг)

$$CP = C_1 \times C_1.$$

При определении критерия покрытия циклов будем считать, что каждый цикл l имеет единственный вход $entry(l)$, который имеет единственного предшественника вне цикла $init(l)$ и единственного преемника внутри цикла $body(l)$ (рис. 17). Циклы в управляющих графах структурированных программ либо обладают этим свойством, либо могут быть преобразованы к такому виду путем добавления вершин $init(l)$ и $body(l)$.

Если у $entry(l)$ цикла l есть преемник, лежащий вне цикла, то будем называть l *циклом типа 0*. Иначе, т. е. если $body(l)$ — единственный преемник $entry(l)$, будем называть l *циклом типа 1*.

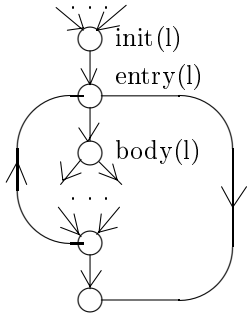


Рис. 17. Структура циклов в программе

Определим функцию, которая по пути $q = [q_1, \ldots, q_n]$ и циклу l возвращает множество чисел, каждое из которых равно числу исполнений цикла на пути q:

$$\begin{aligned} loop\_run(q,l) \;=\; \{\quad & k \geq 0 \mid \exists\, i,j : 1 \leq i < j \leq n, \\ & q_i = init(l), q_j = init(l), \\ & \forall\; s : i < s < j : q_s \neq\; init(l), \\ & |\{q_t : i < t < j \;\&\; q_t = body(l)\}| = k \quad \}, \end{aligned}$$

где $|A|$ означает число элементов множества A.

Критерий all-loops (покрытие циклов)

$$\begin{aligned} REQ_{lp}(P) \quad = \quad & \{(l,\{0\}),\ (l,\{1\}),\ (l,\{x \in N | x > 1\})| \\ & \quad l \text{ — цикл типа } 0\} \cup \\ & \{(l,\{1\}),\ (l,\{x \in N | x > 1\})| \\ & \quad l \text{ — цикл типа } 1\}, \\ COV_{lp}(P,x,(l,s)) \quad \Longleftrightarrow \quad & \exists\; q \in PT(run(P,x)) : (loop\_run(q,l) \cap s) \neq \emptyset. \end{aligned}$$

Пусть VAR — множество всех переменных программы, $Def(x)$ — множество вершин из V, в которых переменная x из VAR определяется, т. е. ей присваивается какое нибудь значение, и $Use(x)$ — множество вершин, в которых x используется. Определим критерии, основанные на потоке данных в программе.

Вначале определим предикат $clear\_reach(q, v, u, x)$, означающий, что участок пути $[v, u] \subset q$ $(v, u \in V)$ не содержит присваиваний переменной $x \in VAR$:

$$clear\_reach(q, v, u, x) \iff \exists\, i, j \ : \ i < j \ : \ q_i = v \ \&\ q_j = u \ \& \\ \forall\, s \ : \ i < s < j \ : \ q_s \notin Def(x).$$

Предположим, что в правой части оператора присваивания `v := expr` есть ошибка, в результате которой значение выражения `expr` вычисляется неправильно. Тогда, для того чтобы эта ошибка могла быть обнаружена, по крайней мере необходимо, чтобы значение переменной `v` использовалось где-нибудь после этого присваивания. Это условие проверяется следующим критерием.

Критерий all-defs (покрытие всех определений)

$$\begin{aligned} REQ_d(P) &= \{(x, v) \mid x \in VAR,\ v \in Def(x)\}, \\ COV_d(P, x, (z, v)) &\iff \exists\, q \in PT(run(P, x))\ \ \exists\, u \in Use(x) \mid \\ &\quad clear\_reach(q, v, u, z). \end{aligned}$$

Следующие критерии потока данных включают критерий all-defs и предназначены для того, чтобы заставить набор тестов исполнять программу так, что значения, используемые в конкретном операторе, были вычислены (для разных тестов) в разных местах программы, что увеличивает вероятность найти ошибку.

Критерий all-uses (покрытие всех def-use цепочек)

$$\begin{aligned} REQ_{du}(P) &= \{(x, v, u) \mid x \in VAR,\ v \in Def(x), \\ &\quad u \in Use(x)\ \ \&\ \exists\, q - \text{путь в } G\ : \\ &\quad clear\_reach(q, v, u, x)\}, \\ COV_{du}(P, x, (z, v, u)) &\iff \exists\, q \in PT(run(P, x)) : clear\_reach(q, v, u, z). \end{aligned}$$

При таком определении критерий all-uses не включает критерий C1, поэтому Раппс и Вейкер предложили модификацию этого критерия [13], в котором использование переменной в предикатной вершине (т. е. в

вершине ветвления) — p-use — заменяется на использование этой переменной во всех выходящих из данной вершины дугах. По-другому это можно сформулировать следующим образом.

Критерий all-uses2 (модифицированный вариант критерия all-uses)

$$
\begin{array}{rcl}
REQ_{du2}(P) & = & \{(x,v,u,u') \mid x \in VAR,\ v \in Def(x),\\
 & & u \in Use(x),\ (u,u') \in E\\
 & & \&\ \exists\ q \text{ — путь в } G\ :\\
 & & clear\_reach(q,v,u,x)\},\\
COV_{du2}(P,x,(z,v,u,u')) & \Longleftrightarrow & \exists\ w \in PT(run(P,x))\\
 & & \exists\ q = [q_1,\ldots,q_n] \subseteq w\ :\\
 & & q_1 = v,\ q_{n-1} = u,\ q_n = u',\\
 & & clear\_reach(q,v,u,z).
\end{array}
$$

Введем теперь функцию, позволяющую для вершины $v \in V$ определить множество используемых в ней переменных:

$$UseVar(v) = \{x \in VAR \mid v \in Use(x)\}.$$

Следующий критерий был предложен Ласки и Корелом [39].

Критерий U-context (покрытие контекстов используемых переменных)

$$
\begin{array}{rcl}
REQ_{uc}(P) & = & \{\ \ (v,\ (v_1,x_1),\ \ldots,\ (v_k,x_k)) \mid\\
 & & UseVar(v) = \{x_1,\ldots,x_k\} \neq \emptyset,\\
 & & \forall\ i : v_i \in Def(x_i),\\
 & & \exists\ q \text{ — путь в } G \mid\\
 & & cov\_context(q,\ (v,(v_1,x_1),\ldots,(v_k,x_k)))\quad \},\\
COV_{uc}(P,x,r) & \Longleftrightarrow & r = (v,(v_1,x_1),\ldots,(v_k,x_k)),\\
 & & \exists\ q \in PT(run(P,x)) : cov\_context(q,r),
\end{array}
$$

где предикат $cov\_context$ определяется следующим образом:

$$\begin{aligned} cov\_context(q,r) \iff\ & r=(v,(v_1,x_1),\ldots,(v_k,x_k)),\\ & \exists\ i,\ i_1,\ldots,i_k\ :\\ & q_i=v,\ q_{i_1}=v_1,\ldots,q_{i_k}=v_k\ \ \&\\ & \forall\ j\ :\ 1\le j\le k\ :\ i_j<i\ \&\\ & (\forall\ s\ :\ i_j<s<i\ :\ q_s\notin Def(x_j)). \end{aligned}$$

Этот критерий можно модифицировать так же, как это было сделано с критерием all-uses [46]. Иногда эти критерии рассматриваются в их модифицированном варианте, как, например, в работе [47], содержащей теоретический анализ способности к выявлению ошибок одного критерия по отношению к другому для большого числа различных критериев структурного тестирования.

Следующий критерий, в отличие от всех предыдущих, учитывает разбиение программы на процедуры, а также тот факт, что в программах вместо циклов зачастую используется рекурсия. Поэтому требуется аналог критерия покрытия циклов для рекурсивных процедур программы.

Пусть $REC\subseteq PROC$ — множество рекурсивных процедур программы P. Определим функцию $rec\_depth(q,pr)$, которая дает множество глубин рекурсивных вызовов процедуры $pr\in REC$ для пути исполнения $q=[q_1,\ldots,q_n]$:

$$\begin{aligned} rec\_depth(q,pr)\ =\{\ & k\ge 0\mid \exists\ i,j\ :\ 1\le i\le j\le n\ \ \&\\ & q_i=entry(pr)\ \ \&\ \ q_j=exit(pr)\ \ \&\\ & (\forall\ s: i<s<j: q_s\ne entry(pr)\ \&\ q_s\ne exit(pr))\ \&\\ & |\{q_t|1\le t\le j\ \&\ q_t=entry(pr)\}|\ -\\ & |\{q_t|1\le t\le j\ \&\ q_t=exit(pr)\}|\ =k\ \}. \end{aligned}$$

Критерий all-rec (покрытие рекурсии)

$$\begin{aligned} REQ_{rec}(P) &= \{(pr,\{0\}),\ (pr,\{1\}),\ (pr,\{x\in N|x>1\})\\ &\quad \mid pr\in REC\ \},\\ COV_{rec}(P,x,(a,s)) &\iff rec\_depth(run(P,x),a)\cap s\ne\emptyset. \end{aligned}$$

Логические критерии структурного тестирования

Логические критерии формулируются путем задания логического условия (предиката) в некотором месте программы, и это условие зависит от ее переменных. Таким образом, требуемый элемент определяется логическим условием и покрывается тогда и только тогда, когда управление достигает соответствующей части программы, и значение переменных таково, что условие истинно.

В качестве состояний для этих критериев рассматриваются *состояния памяти*, т. е. отображения s множества всех переменных VAR в множество их значений. Значение некоторого выражения e в состоянии s, т. е. при соответствующих s значениях переменных, будем записывать в виде $e(s)$.

По аналогии с последовательностью $run(P,x) = [v_1, \ldots, v_n]$ определим последовательность состояний памяти $states(P,x) = [s_1, \ldots, s_{n+1}]$, которая получается при исполнении программы P на входных данных x. При этом s_i соответствует значениям всех переменных программы непосредственно перед исполнением оператора v_i, s_1 определяется входными данными x, а s_{n+1} определяет выходные данные $P(x)$.

Непосредственно последовательность $states(P,x)$ использоваться не будет, вместо этого определим $ST(P,x,v)$ — множество состояний программы P перед оператором $v \in V$ при исполнении ее на входных данных x:

$$\begin{aligned} ST(P,x,v) \quad = \quad \{s \mid \quad & run(P,x) = [v_1, \ldots, v_n], \\ & states(P,x) = [s_1, \ldots, s_{n+1}], \\ & \exists\ i\ :\ v_i = v\ \&\ s = s_i\ \}. \end{aligned}$$

Кроме этого, нам еще потребуется функция $val(v,s,e)$, которая по выражению e (из оператора v) дает его значение во время исполнения оператора v при состоянии памяти s. Если выражение e не вычисляется, то $val(v,s,e) = \perp$. Например, оператор `x := x&y;` при короткой схеме вычисления логических выражений эквивалентен `x:= if x then y else false;` и если $x = FALSE$, то y не будет вычисляться, т. е. $val(v,s,y) = \perp$.

Будем рассматривать два способа передачи параметров: по значению и по имени. Если выражение e — формальный параметр вызова процедуры, то будем обозначать это же выражение, но после вызова процедуры, через $\downarrow e$. При этом, если e — формальный параметр, передаваемый по значению, то $val(v,s,\downarrow e) = val(v,s,e)$. Если же e — переменная и

является формальным параметром для `VAR`-параметра процедуры, то $val(v, s, \downarrow e)$ равно значению e после вызова процедуры.

Также будем считать, что при вычислении выражений, в которые входят взятые из v подвыражения, нужно взять все значения этих подвыражений, получающиеся при исполнении оператора v при состоянии памяти s, подставить в исходное выражение и затем вычислить его. Выражение равно $\perp$, если хотя бы одно из подвыражений равно $\perp$.

Следующие критерии будем определять путем задания $EREQ(v)$ — множества требуемых элементов в $v \in V$, и логического выражения $ECOV(r)$, истинность которого означает покрытие $r \in EREQ(v)$. При этом REQ и COV определяются следующим образом:

$$\begin{aligned} REQ(P) &= \{(v, EREQ(v)) |\ v \in V\ \&\ EREQ(v) \neq \emptyset\}, \\ COV(P, x, (v, r)) &\iff \exists\ s \in ST(P, x, v) \mid val(v, s, ECOV(r)). \end{aligned}$$

Следующий критерий позволяет как получить более разнообразный набор тестов, так и обнаружить некоторые ошибки в логических выражениях: например использование `&` вместо `OR` и наоборот, неправильное использование скобок и т. д.

Критерий покрытия условий

$$\begin{aligned} EREQ(v) &= \{(e_1, TRUE),\ (e_1, FALSE),\ (e_2, TRUE), \\ &\quad (e_2, FALSE) \mid \text{в оператор } v \text{ входит выражение} \\ &\quad e_1 \& e_2 \text{ или } e_1\ OR\ e_2.\ \}, \\ ECOV(e, c) &\iff e = c. \end{aligned}$$

Достаточно частой ошибкой является использование $l \leq r$ вместо $l < r$ и $l \geq r$ вместо $l > r$ и наоборот. Для обнаружения этих ошибок требуется следующий критерий.

Критерий покрытия отношений

$$\begin{aligned} EREQ(v) &= \{(l, r) \mid \text{ в оператор } v \text{ входит одно из выражений} \\ &\quad l < r,\ l \leq r,\ l > r,\ l \geq r \text{ и } l, r \text{ — целого типа}\}, \\ ECOV(l, r) &\iff l = r. \end{aligned}$$

Следующие слабомутационные критерии предназначены для обнаружения неправильного использования арифметических операций и имен переменных и требуют построения такого набора тестов, который позволил бы выявить все такие неточности. Пусть AR — множество арифметических операций, т.е. $AR = \{+, -, *, /\}$.

Критерий слабой мутации арифметических операций

$$EREQ(v) = \{(e_1, e_2, op_1, op_2) \mid op_1 \in AR,\ op_2 \in AR \backslash \{op_1\} \text{ и выражение } e_1\ op_1\ e_2 \text{ входит в оператор } v\},$$
$$ECOV(r) \iff r = (e_1, e_2, op_1, op_2), e_1\ op_1\ e_2\ \neq\ e_1\ op_2\ e_2.$$

Критерий слабой мутации имен переменных

$$EREQ(v) = \{(x_1, x_2) \mid x_1, x_2 \in VAR \text{ и имеют один и тот же тип и область действия, } x_1 \text{ и } x_2 \text{ — различные переменные и } x_1 \in UseVar(v)\ \},$$
$$ECOV(x_1, x_2) \iff x_1\ \neq\ x_2.$$

Идея определения следующего критерия состоит в объединении методов тестирования черного и белого ящиков, а именно метода разбиения на классы эквивалентности, примененного к процедурам программы, которые можно рассматривать как функции, отображающие входные параметры процедуры и используемые глобальные переменные в результат процедуры (если есть) и измененные глобальные переменные.

Для этого критерия должны быть заданы тестовые условия для некоторых типов данных, т. е. условия, которым должен удовлетворять набор тестов для тестирования программы (или процедуры) с входными данными этого типа. Например, для перечислимых типов — это либо все значения данного типа, если их конечное число, либо некоторые значения данного типа, в противном случае (определяемые, например, с помощью критериев черного ящика); для списков — пустой список, одноэлементный список и многоэлементный список; для диапазонов — значения на границах диапазона и т. д. Более подробно этот вопрос рассматривается в работе [19].

Итак, тестовым условием типа T будем называть предикат $c(x)$, где x имеет тип T. Пусть

$$g_1 = [c_1^1(x), \ldots, c_{n_1}^1(x)], \ldots, g_s = [c_1^s(x), \ldots, c_{n_s}^s(x)]$$

— группы тестовых условий, заданных пользователем для типа T. Определим множество условий для тестирования данных типа T:

$$\begin{aligned} COND(T) \quad = \quad & \{c_1^1(x), \neg c_1^1(x) \& c_2^1(x), \ldots, \neg c_1^1(x) \& \ldots \& \neg c_{n_1-1}^1(x) \& c_{n_1}^1(x), \\ & \ldots, \\ & c_1^s(x), \neg c_1^s(x) \& c_2^s(x), \ldots, \neg c_1^s(x) \& \ldots \& \neg c_{n_s-1}^s(x) \& c_{n_s}^s(x)\}. \end{aligned}$$

В этом определении из каждой группы сформировано разбиение данных типа T на классы эквивалентностей и затем все условия объединены в одно множество $COND(T)$. При определении следующих критериев будем использовать запись $e[t_1/x_1, \ldots, t_n/x_n]$ для выражения, получающегося заменой переменных $x_1, \ldots, x_n$ в выражении $e(x_1, \ldots, x_n)$ на $t_1, \ldots, t_n$.

Критерий покрытия входных параметров

$$\begin{aligned} EREQ(v) \quad = \quad & \{c[t/x] \mid \exists\, pr \in PROC \ : \ v = entry(pr), \\ & \quad t \text{ — входной параметр } pr \text{ типа } T, \\ & \quad c(x) \in COND(T) \ \}, \\ ECOV(c(t)) \iff & \ c(t). \end{aligned}$$

При определении следующего критерия будем считать, что результат процедуры-функции запоминается в переменной `RESULT`, значение которой вычисляется до исполнения оператора $exit(pr)$.

Критерий покрытия результатов процедур

$$\begin{aligned} EREQ(v) \quad = \quad & \{c[RESULT/x] \mid \exists\, pr \in PROC \ : \ v = exit(pr), \\ & \quad RESULT \text{ для } pr \text{ имеет тип } T, \\ & \quad c(x) \in COND(T) \ \}, \\ ECOV(c(t)) \iff & \ c(t). \end{aligned}$$

Определим теперь критерий интеграционного тестирования, который предназначен для обнаружения ошибок, возникающих в результате неправильного вызова процедуры или неправильной обработки результатов, возвращаемых процедурой. Для этого критерия должны быть заданы группы тестовых условий для вызовов процедур, причем они разделяются на условия «до вызова» и «после вызова».

Тестовое условие «до вызова» для процедуры $pr(x_1, \ldots, x_n)$ — это логическое выражение вида $c(x_1, \ldots, x_n)$, а «после вызова» — $c(x_1, \ldots, x_n, RESULT)$, где RESULT на самом деле может присутствовать в c только для процедуры-функции и означает возвращаемое значение, т. е.

$$val(v, s, RESULT(pr)) = val(v, s, pr(e_1, \ldots, e_n)).$$

Итак, пусть пользователь определил для процедуры pr группы тестовых условий «до вызова» и «после вызова». Так же, как с тестовыми условиями для типов, сделаем разбиение на классы эквивалентностей и объединим группы «до вызова» в множество $COND\_BEFORE(pr)$, а «после вызова» в множество $COND\_AFTER(pr)$.

Критерий покрытия вызовов процедур

$$\begin{aligned} EREQ(v) \quad = \quad & \{c[e_1/x_1, \ldots, e_n/x_n],\ c'[\downarrow e_1/x_1, \ldots, \downarrow e_n/x_n] \mid \\ & \text{в вершине } v \in V \text{ есть вызов процедуры } pr(e_1, \ldots, e_n), \\ & c \in COND\_BEFORE(pr), \\ & c' \in COND\_AFTER(pr)\ \}, \\ ECOV(c) \iff & c. \end{aligned}$$

3.4. Мутационный подход

Мутационное тестирование — мощная, основанная на ошибках техника тестирования модульного уровня[1]. Так как этот подход основан на ошибках, то он нацелен на тестирование и выявление некоторых специфичных типов ошибок, а именно простых синтаксических изменений

[1] Данный раздел относится к разделу построения тестов, однако его можно отнести и к разделу автоматизации построения тестов и оценки полноты набора тестов.

программ. Мутационное тестирование базируется на двух предположениях: *гипотезе о квалифицированном программисте (competent programmer hypothesis)* и *эффекте взаимосвязи (coupling effect)*. Гипотеза о квалифицированном программисте предполагает, что квалифицированный программист имеет тенденцию к написанию программ, близких к правильным. Другими словами, если программа, написанная квалифицированным программистом, не корректна, то она отличается от правильной версии несколькими относительно простыми ошибками. Эффект взаимосвязи указывает на то, что множество тестов, выявляющих все простые ошибки в программе, также способен обнаруживать более сложные ошибки[1]. Мутационное тестирование можно описать следующим образом. Простые ошибки вносятся в программу путем создания новых версий программы, каждая из которых содержит по одной ошибке. Эти порожденные версии тестируемой программы называются мутантами. Наша задача — создать набор тестов, которые выявляют мутантов, т. е. обнаруживают несовпадение результата исполнения мутанта с правильным результатом, и в этом случае мутант считается «убитым». Когда это случается мутант становится «мертвым» и не участвует более в процессе тестирования, так как ошибка, представленная мутантом, выявлена и, что более важно, мутант выполнил свою функцию — создан тест. Иногда мутант не может быть убит на любом множестве тестов. Мутанты этого типа — программы функционально эквивалентные исходной программе[2].

Качество набора тестов измеряется *мутационной оценкой (mytation scores)*, которая вычисляется по формуле

$$MS(P,T) = \frac{M_k}{M_t - M_q},$$

где P — тестируемая программа; T — набор тестов; M_k — число убитых мутантов; M_t — общее число мутантов сгенерированных для программы; M_q — число мутантов эквивалентных исходной программе.

Множество тестов называется *мутационно достаточным (mutation-adequate)* если его мутационная оценка 100 %.

[1]Следует отметить, что эффект взаимосвязи исследовался довольно подробно и имеются как эксперементальные [33], так и теоретические работы [32, 34], подтверждающие это.

[2]Для снижения временных затрат на определение мутантов эквивалентных исходной программе разрабатывались подходы по их автоматическому выявлению [28].

Ниже приведен текст маленькой функции на ФОРТРАНЕ с тремя мутированными строками (помечены символом «†»). Следует заметить, что каждый оператор с мутацией должен заменять исходный в программе и так полученная программа представлять отдельную программу — мутант исходной.

```
                FUNCTION Min(I,J)
1                Min = I
†                Min = J
2                IF (J .LT. I) Min = J
†                IF (J .GT. I) Min = J
†                IF (J .LT. Min) Min = J
3                RETURN
```

Данный подход можно описать с помощью набора мутационных преобразований, которые и позволяют порождать мутантов из тестируемой программы. Так, в текущем варианте системы Mothra [35] для мутационного тестирования используется 22 мутационных преобразования. Они приведены в табл. 7.

Все эти мутационные преобразования можно разделить на три широких класса:

• *Анализ операторов:* замена каждого оператора на **TRAP**; замена каждого оператора на **CONTINUE**; замена каждого оператора в подпрограмме на **RETURN**; замена метки в каждом **GOTO** операторе; замена метки в каждом **DO** операторе.

• *Анализ предикатов:* взятие абсолютной величины значений выражений и ее отрицательное значение; замена каждой арифметической операции на любую другую; замена каждой операции отношения любой другой; замена каждой логической операции на любую другую; вставку унарных операций перед выражениями; изменения значений констант; изменения операторов **DATA**.

• *Корректность неаккуратности (coincidental correctness):* замена скалярных переменных, ссылок на массивы и констант на другие скалярные переменные, ссылки массивов и константы; замена ссылок на имена массивов на имена других массивов.

Преобразования из этого множества не только удовлетворяют многим критериям белого ящика, например покрытию операторов и покрытию дуг, но и прямо моделируют многие типы ошибок. Так, преобразования корректности неаккуратности представляют блок преобразований, исследующих случаи неправильного использования программистом имени переменной или имени массива. Преобразования анализа

Таблица 7

Мутационные преобразования в системе Mothra

Type	**Descripton**
aar	array reference for array reference replacement
abs	absolute value insertion
acr	array reference for constant replacement
aor	arithmetic operator replacement
asr	array reference for scalar variable replacement
car	constant for array reference replacement
cnr	comparable array name replacement
cnr	constant replacement
csr	constant for scalar variable replacement
der	**DO** statement end replacement
dsa	**DATA** statement alteration
glr	**GOTO** label replacement
lcr	logical connector replacement
ror	relational operator replacement
rsr	**RETURN** statement replacement
san	statement analysis (replacement by TRAP)
sar	scalar variable for array reference replacement
scr	scalar for constant replacement
sdl	statement deletion
src	source constant replacement
svr	scalar variable replacement
uoi	unary operation insertion

предикатов представляют ошибки, которые программист делает внутри выражения, например использует некорректную операцию сравнения или неправильную арифметическую операцию. Преобразования анализа операторов позволяют выявить несколько типов ошибок. Замена на **TRAP** позволяет убедиться, что каждый оператор достижим; замена на **CONTINUE** позволяет проверить необходимость каждого оператора для правильного исполнения программы; замена на **RETURN** позволяет убедиться, что операторы, следующие за замененным, необходимы. Замена метки эмулирует ошибку при использовании неправильной метки.

Однако вычислительные затраты на реализацию данного подхода очень большие. Например, для функции **Min**, приведенной выше, число мутантов 44. И каждый мутант надо оттранслировать и исполнять, пока не будет построен тест, который его убъет. Для оценки числа мутантов предложены следующие аппроксимационные модели [27]:

$$Mutants = \beta_0 + \beta_1 * Vars + \beta_2 * Varrefs + \beta_3 * Vars * Varrefs,$$

где $Vars$ — число переменных, $Varrefs$ — число вхождений имен переменных.

$$Mutants = \beta_0 + \beta_1 * Lines + \beta_2 * Lines * Lines,$$

$$Mutants = \beta_0 + \beta_1 * Vars + \beta_2 * Vars * Vars.$$

Эти две модели предсказывают число мутантов лучше предыдущей, правда, только для простых программ, состоящих из одной подпрограммы или процедуры или основной программы без подпрограмм.

Так как число программых модулей — важная характеристика программы, то была предложена следующая формула:

$$Mutants = \beta_0 + \beta_1 * Vars + \beta_2 * Varrefs + \beta_3 * Units + \beta_4 * Vars * Varrefs.$$

Для уменьшения вычислительных затрат были разработаны следующие варианты данного подхода (далее просто мутационный подход будем называть строгим мутационным подходом):

— слабое (weak) мутационное тестирование;

— устойчивое (firm) мутационное тестирование;

— избирательное (selective) мутационное тестирование.

Слабое мутационное тестирование. Данная модификация мутационного подхода введена Howden [30] и может быть описана следующим образом.

Рассмотрим программу P и ее простой компонент C. Пусть C' — мутационная версия C и P' — мутант P, содержащий C'. Набор тестов называется достаточным для слабого мутационного тестирования, если он приводит к получению различных результатов для C и C', хотя бы на одном исполнении C'. Заметим, что если даже результаты C' и C различны, результаты P'и P могут быть одинаковыми. Слабое мутационное тестирование требует создания менее строгого набора тестов, так как требуется чтобы были разные результаты C и C'.

Howden при определении слабого мутационного тестирования не определил, что такое компонент. Он рассматривал пять типов программных компонент:

1) использование переменной;
2) определение переменной;
3) арифметическое выражение;
4) логическое выражение;
5) выражение отношения.

Позднее Offutt и Lee [31] в своей системе слабого мутационного тестирования Леонардо определили компонент как место в программе, в котором сравниваются состояния исходной программы и мутанта. Они предложили следующие варианты точек сравнения:

1. *EXpression-WEAK/1.* EX-WEAK/1 сравниваются состояния после исполнения самого внутреннего выражения с мутацией.

2. *STatement-WEAK/1.* ST-WEAK/1 сравниваются состояния после исполнения оператора с мутацией.

3. *Basic-Block-WEAK/1.* BB-WEAK/1 сравниваются состояния после первого исполнения основного блока с мутацией.

4. *Basic-Block-WEAK/N.* BB-WEAK/N сравниваются состояния после N-того исполнения основного блока с мутацией.

Рассмотрим пример. Ниже приведена функция **Max**, к которой применили мутационное преобразование **CAR**:

```
1       int max(int* input, int size)
2       {
3       int j, result;
4       result = input[0];
5       for (j = 1; j < size; j++) {
6       if (input[j] > result)
7       result = j;
8       }
9       return result;
10      }
```

Мутационное преобразование **CAR** заменило элемент массива *input[result]* на переменную *result* в строке 6. При EX-WEAK/1 значение выражения *result* должно сравниваться с значением выражения *input[result]*. В случае ST-WEAK/1 значение предиката *input[j] > result* должно сравниваться со значением *input[j] > input[result]*. При BB-WEAK/1 должны сравниваться состояния программ перед стокой 8.

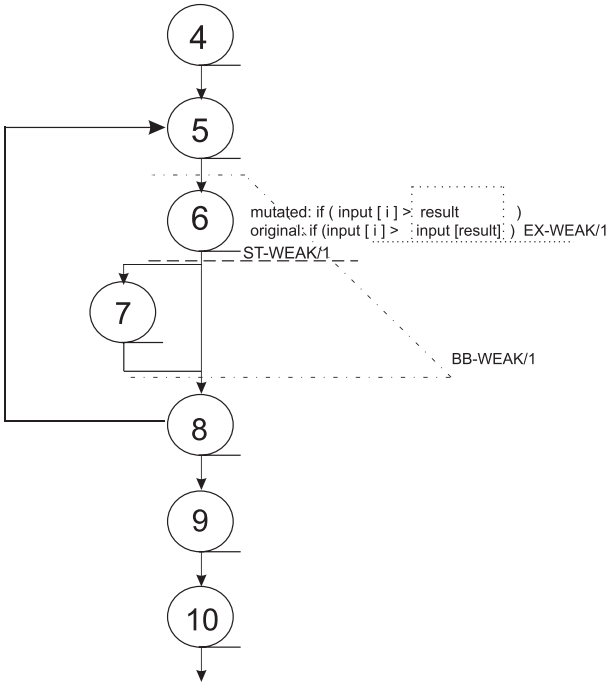


Рис. 18. Точки сравнения для различных вариантов сравнения при слабо мутационном тестировании

В случае BB-WEAK/N состояния программ должны сравниваться каждый раз, когда достигается строка 8. Точки сравнения для всех четырех вариантов показаны на рис. 18.

Измерения проведенные Offutt и Lee показали, что число тестов генерируемых для слабомутационного тестирования значительно (drastically) меньше чем для строгомутационного тестирования. Поэтому естественно и затраты на слабомутационное тестирование меньше. Но 100 % адекватное слабомутационное множество тестов не обеспечивает 100 % строгой мутационной оценки в большинстве случаев. Поэтому, множество тестов для слабомутационного тестирования эффективно только на уровне компонент.

Устойчивое мутационное тестирование. Стратегии используемые при строгом мутационном тестировании и слабом мутационном те-

Таблица 8

Возможный выбор кода для Max

1	j
2	if (j > input[result] result = j;

3	for (j = 1; j < size; j++)
	{
	if (input[j] > input[result])
	result = j;
	}

4	result = 0;
	for (j = 1; j < size; j++)
	{
	if (input[j] > input[result])
	result = j;
	}

5	int max(int* input, int size) {
	int j, result;
	result = 0;
	for (j = 1; j < size; j++)
	{
	if (input[j] > input[result])
	result = j;
	}
	return result;
	}

стировании располагаются на концах шкалы тестирования. Строгое мутационное тестирование дает эффективный результат, но требует огромных вычислительных затрат. Слабое мутационное тестирование порождает значительно небольшое число тестов, но они эффективны только для программных компонент. Устойчивое мутационное тестирование призвано обеспечить баланс между вычислительными затратами и приемлемой эффективностью.

Устойчивое мутационное тестирование является расширением слабого мутационного тестирования. При слабом мутационном тестировании состояние программы сравнивается после каждого исполнения

компоненты. При строгомутационном тестировании вывод программы сравнивается после завершения исполнения программы. При устойчивом мутационном тестировании состояние программы сравнивается в некоторые временные срезы исполнения программы, минимальным является срез выполнения мутированного оператора. Точные компоненты для сравнения определяются выбором пользователя. Как пишут Woodward и Halewood [36], устойчивое мутационное тестирование предлагает пользователю большую свободу в следующих случаях:

1. Выбор кода. Пользователь может выбрать часть программы, которая может быть как некоторой программной структурой, так и последовательностью структур. Возможная селекция программного кода для программы **Max** приведена в табл. 8.

2. Выбор объекта сравнения. Любой объект, который определен или на который ссылаются и определяют в выделенной области, может использоваться для результирующего сравнения. Табл. 9 содержит возможные переменные для всех возможных участков кода из табл. 8, которые могут использоваться для сравнения.

3. Выбор набора мутационных преобразований. Некоторые мутационные преобразования могут быть применены к выделенному коду программы.

Таблица 9

Возможные сравнения результатов для Max

Выбор кода	**Возможный результат сравнения**
1	j
2	result, input[result], j
3	result, input[result], input[j], j, size
4	result, input[result], input[j], j, size
5	result, input[result], input[j], j, input, size

Огромным преимуществом устойчивого мутационного тестирования является то, что пользователь может определять части кода, которые должны тестироваться, переменные для сравнения и используемые мутацинные преобразования.

Некоторые варианты выбора для устойчивого мутационного тестирования приведены в табл. 10.

Устойчивое мутационное тестирование менее затратное, чем строгомутационное тестирование, но сохраняются значительные вычислительные затраты.

Таблица 10

Возможные комбинации выбора для Max

	Выбор кода	**Сравнение результата**	**Мутационное преобразование**	**Результат**

1	for (j = 1; j < size; j++) {
	if (input[j] > input[result])
	result = j;
	}
		result	ACR для j:
			result = j на
			result = input[j];
				Сравнение значения 'result' в конце исполнения цикла

2	if (j > input[result]) result = j;	result	ROR на >: > на <	Сравнение значения 'result' после каждого исполнения оператора result = j;

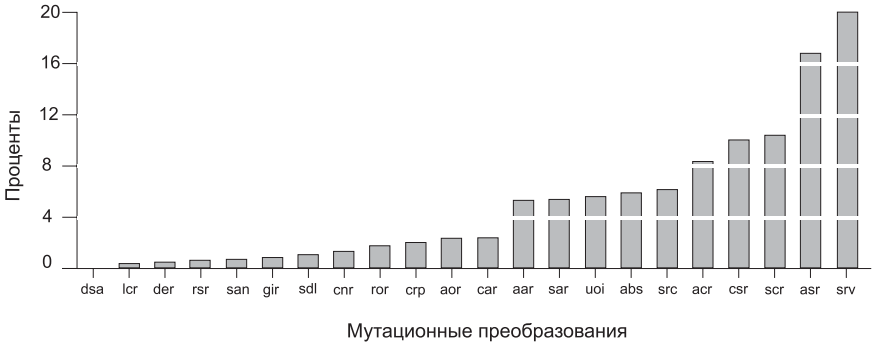


Рис. 19. Вклад мутационных преобразований в число мутантов

Избирательное мутационное тестирование. Как слабое, так и устойчивое мутационное тестирование, остаются вычислительно-затратными. Число мутантов, создаваемых в этих технологиях, такое же, как и при строгом мутационном тестировании. Offutt с соавт. [27] предположили, что затраты на мутационное тестирование уменьшатся с уменьшением числа генерируемых мутантов. Идея избирательного мутационного тестирования состоит в исключении мутационных преобразований, которые порождают наибольшее число мутантов. Исходя из приведенных ранее моделей, предсказывающих число мутантов, Mathur [29] и Offutt с соавт. [27] предложили не использовать два мутационных преобразования — SVR (Scalar variable replacement) и ASR (array for scalar replacement). Эти два преобразования порождают большиство мутантов, так как SVR заменяет каждую скалярную переменную в программе на любую совместимую скалярную переменную, обнаруженную

в программе. ASR заменяет каждую скалярную переменную в программе на каждую ссылку на массив совместимого типа. Этот вывод подтверждается и эксперементами, проведенными Offutt с соавт. На рис. 19 показан вклад каждого мутационного преобразования в число порождаемых мутантов. Проведенные эксперементы позволяют также утверждать, что избирательное (2-selective — когда отказываемся от двух указанных преобразований) мутационное тестирование так же хорошо, как и строгое мутационное тестирование. Исследования проводились и для 4-избирательного и 6-избирательного мутационного тестирования и результаты оказались достаточно приемлемыми. Цена применения этих подходов естественно снижалась.

Итоги. 1. Специалисты оценивают данное направление в тестировании с оптимизмом и надеются на скорое практическое использование рассмотренных подходов при тестировании программных продуктов. Для достижения этого используются разные подходы по снижению затратности.

2. Данный подход можно использовать как для построения хорошего набора тестов, так и для оценки полноты (или мутационной полноты) имеющегося набора тестов.

4. СТРАТЕГИЯ

Обсуждавшиеся методологии проектирования тестов могут быть объединены в общую стратегию. Причина их объединения становится очевидной: каждый метод обеспечивает создание определенного набора «хороших» тестов, но ни один из них сам по себе не может дать полный набор тестов. Приемлемая стратегия состоит в следующем:

1. Если спецификация содержит комбинации входных условий, то начать рекомендуется с применения метода функциональных диаграмм.

2. Определить правильные и неправильные классы эквивалентности для входных и выходных данных и дополнить, если это необходимо, тесты, построенные на предыдущих шагах.

3. В любом случае необходимо использовать анализ граничных значений. Напомним, что этот метод включает анализ граничных значений входных и выходных переменных. Анализ граничных значений дает набор дополнительных тестовых условий, но, как замечено в разделе, посвященном функциональным диаграммам, многие из них (если не все) могут быть включены в тесты метода фунциональных диаграмм.

4. Для получения дополнительных тестов рекомендуется использовать метод предположения об ошибке.

5. Проверить логику программы на полученном наборе тестов. Здесь тестировщик в первую очередь должен определить, насколько высоки требования к надежности тестируемого программного продукта[1]. В зависимости от этого определяется тот комплексный критерий белого ящика, который будет определять набор покрываемыех тестовых ситуации логики нашей программы. Считается, что минимальным требованием надежности должно быть выполнение критерия покрытия решений (однако имеется и множество примеров, когда ограничиваются критерием покрытия операторов). Если необходимость выполнения комплексного критерия покрытия приводит к построению тестов, не встречающихся среди построенных на предыдущих четырех шагах, и если эти тестовые ситуации не являются нереализуемыми (т. е. определенные тестовые ситуации невозможно покрыть вследствие природы программы), то следует дополнить уже построенный набор тестов тестами, число которых достаточно для удовлетворения критерия покрытия. Эта стратегия опять-таки не гарантирует, что все ошибки будут найдены, но вместе с тем ее применение обеспечивает приемлемый компромисс. Реализация подобной стратегии весьма трудоемка, но ведь никто и никогда не утверждал, что тестирование программы — легкое дело.

5. ТЕСТИРОВАНИЕ МОДУЛЕЙ

Рассматривая проблемы тестирования, мы до сих пор не касались организации процедуры тестирования и размера тестируемых программ. Большие программы состоят из модулей, которые, в свою очередь, состоят из подпрограмм или процедур, и все это требует специальных способов структурирования процесса тестирования. Поэтому мы рассмотрим начальный шаг структурирования — тестирование модулей.

Тестирование модулей представляет собой процесс тестирования отдельных подпрограмм или процедур программы. Подразумевается, что, прежде чем начинать тестирование программы в целом, следует протестировать отдельные небольшие части, образующие эту программу. Такой подход мотивируется тремя причинами. Во-первых, появляется возможность управлять комбинаторикой тестирования, поскольку

[1] В большинстве случаев требования к надежности ему предъявляются вместе с программой для тестирования.

первоначально внимание концентрируется на небольших модулях программы. Во-вторых, облегчается задача отладки программы, т. е. обнаружение места ошибки и исправление текста программы. Наконец, в-третьих, допускается параллелизм, что позволяет одновременно тестировать несколько модулей.

Процесс тестирования модулей рассматривается в двух аспектах: способы построения наборов тестов и порядок, в котором модули тестируются и собираются в программу.

5.1. Проектирование тестов

При проектировании тестов для тестирования модулей должна быть доступна следующая информация: спецификация на разрабатываемую систему, спецификация модуля и его текст. Спецификация модуля обычно содержит описание входных и выходных параметров модуля и его функций.

Тестирование модулей в основном ориентировано на принцип белого ящика. Это объясняется прежде всего тем, что принцип белого ящика труднее реализовать при переходе в последующем к тестированию более крупных единиц, например программ в целом. Кроме того, последующие этапы тестирования ориентированы на обнаружение ошибок различного типа, т. е. ошибок, не обязательно связанных с логикой программы, а возникающих, например, из-за несоответствия программы требованиям пользователя.

Процедура создания набора тестов для тестирования модулей такова: на основании спецификации системы и описания модуля разработать набор тестов, воспользовавшись критериями черного ящика, а далее, исполнив этот набор тестов, оценить, насколько полно он покрывает логику нашего модуля в соответствии с критериями белого ящика, например C1. При необходимости набор тестов дорабатывается.

5.2. Организация процесса тестирования

Реализация процесса тестирования модулей опирается на два ключевых положения: построение эффективного набора тестов (этот вопрос рассмотрен выше) и выбор способа, посредством которого модули комбинируются при построении из них рабочей программы. Второе положение является важным, так как оно задает форму написания тестов модуля, типы средств, используемых при тестировании, порядок коди-

рования и тестирования модулей, стоимость генерации тестов и стоимость отладки (т. е. локализации и исправления, ошибок). Рассмотрим два подхода к комбинированию модулей: пошаговое и монолитное тестирование, а также два варианта пошагового подхода: тестирование снизу вверх (восходящее) и тестирование сверху вниз (нисходящее).

5.2.1. Пошаговое и монолитное тестирование

Вопрос: что лучше — выполнить по отдельности тестирование каждого модуля, а затем сформировать из них рабочую программу или же каждый модуль для тестирования подключать к набору ранее оттестированных модулей? Первый подход обычно называют монолитным методом тестирования, или методом «большого удара» при тестировании программы; второй подход известен как пошаговый метод тестирования.

В качестве примера рассмотрим программу, представленную на рис. 20. Прямоугольниками обозначены шесть модулей программы (подпрограммы или процедуры). Линиями показана иерархия управления: модуль A вызывает модули B, C и D, модуль B вызывает модуль E и т. д. При монолитном подходе тестирование выполняется следующим образом. Сначала тестируются шесть модулей, входящих в программу, причем тестирование каждого осуществляется независимо от других. При различных условиях и числе исполнителей модули могут тестироваться последовательно или параллельно. Затем из модулей собирается исполняемая программа.

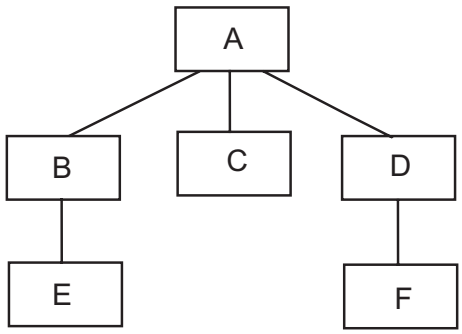


Рис. 20. Пример программы состоящей из шести модулей

Для тестирования любого модуля требуются специальный модуль-драйвер и один или несколько модулей-заглушек. Например, если тестируется модуль B, то первоначально нужно разработать тесты, а затем написать небольшую программу, которая передаст модулю B входные тестовые данные, необходимые для его исполнения. Для этой цели можно использовать и отладочные средства. Драйвер должен также отобразить программисту некоторые сведения о результатах работы модуля B. Кроме того, поскольку в модуле B есть вызов модуля E, необходимо написать модуль-заглушку, которому будет передано управление при исполнении вызова модуля E. Модулю-заглушке, используемому вместо модуля E во время тестирования, присваивается тоже имя E, и он должен имитировать функции этого модуля. После завершения тестирования всех шести модулей они собираются в единую программу.

Метод пошагового тестирования предполагает, что модули тестируются не изолированно друг от друга, а для выполнения тестов подключаются к набору уже ранее оттестированных модулей.

Число возможных реализаций пошагового метода велико. Ключевым является вопрос о том, как следует начинать тестирование программы — сверху или снизу. Выяснение недостатков и преимуществ этих подходов пока оставим и допустим, что тестирование начинается снизу. Первоначально можно последовательно или параллельно выполнить тестирование модулей E, C и F. При этом для каждого модуля придется сделать драйвер; заглушки же здесь не нужны. Следующий шаг — тестирование модулей B и D, но не независимо, а совместно с модулями E и F соответственно. Другими словами, для тестирования модуля B разрабатывается новый драйвер, и выполняется тестирование пары B—E. Пошаговый процесс продолжается до тех пор, пока к набору оттестированных модулей не будет подключен последний модуль, в данном случае модуль A. Заметим, что эта процедура может быть развита и сверху вниз.

Некоторые обобщения:

1. Монолитное тестирование требует больших затрат труда. Для программы на рис. 20 необходимо создать пять драйверов и пять заглушек с учетом того, что для верхнего модуля драйвер не нужен. При пошаговом тестировании снизу вверх потребуется только пять драйверов, а сверху вниз — только пять заглушек. Сокращение трудозатрат

объясняется тем, что при тестировании сверху вниз тестируемые модули выполняют функции драйверов, а при тестировании снизу вверх ранее оттестированные модули играют роль заглушек.

2. При пошаговом тестировании раньше обнаруживаются ошибки в интерфейсах между модулями, так как раньше начинается сборка программы. При монолитном тестировании модули объединяются на последней фазе процесса тестирования, и если интерфейсы спроектированы неправильно, то цена данной ошибки может быть значительной.

3. Отладка программ при пошаговом тестировании легче. Локализовать ошибку при монолитном тестировании довольно трудно, поскольку она может находиться в любом модуле программы. При пошаговом тестировании ошибки в основном связаны с тем модулем который подключается последним.

4. Результаты пошагового тестирования более совершенны. Например, при пошаговом тестировании модуля B одновременно с ним выполняется или модуль A, или модуль E (в зависимости от нисходящего или восходящего тестирования). Хотя эти модули ранее уже тщательно оттестированы, совместное их использование с модулем B возможно создаст новые тестовые ситуации (модуль всегда более сложен, чем заглушка), которые приведут к выявлению ошибок, не обнаруженных при автономном тестировании.

При монолитном тестировании результаты ограничены только этим модулем. Другими словами, из-за того что при пошаговом тестировании оттестированные ранее модули используются в качестве драйверов или заглушек, они подвергаются дополнительной проверке при тестировании текущего модуля.

5. Расход машинного времени при монолитном тестировании меньше. Это объясняется тем, что, тестируя текущий модуль, мы одновременно с ним исполняем и все модули, которые он вызывает при пошаговом восходящем подходе, а при монолитном только сам модуль и заглушки (похожая картина наблюдается и при пошаговом нисходяшем подходе). Следовательно, число исполняемых машинных команд во время прогона теста при пошаговом тестировании явно больше, чем при монолитном. Однако указанное превышение может компенсироваться тем, что монолитное тестирование требует больше заглушек и драйверов, чем пошаговое, и в результате расходуется машинное время на разработку этих драйверов и заглушек[1].

[1] Все определяется тем, насколько велики или малы временные характеристики

6. Использование монолитного метода предоставляет большие возможности для параллельной организации работы на начальной фазе тестирования (тестирования всех модулей одновременно). Это положение может иметь важное значение при выполнении больших проектов, в которых много модулей и много исполнителей, поскольку численность персонала, участвующего в проекте, максимальна на начальной фазе.

В заключение отметим, что пп. 1—4 демонстрируют преимущества пошагового тестирования, а пп. 5—6 — его недостатки. Поскольку для современного этапа развития вычислительной техники характерны тенденции к уменьшению стоимости аппаратуры и увеличению стоимости труда, последствия ошибок в математическом обеспечении весьма серьезны, а стоимость устранения ошибки тем меньше, чем раньше она обнаружена; преимущества, указанные в пп. 1—4, выступают на первый план. В то же время ущерб, наносимый недостатками (пп. 5—6), невелик. Все это позволяет сделать вывод, что пошаговое тестирование является предпочтительным.

Нисходящее и восходящее тестирование. Убедившись в преимуществах пошагового тестирования перед монолитным, исследуем две крайние стратегии пошагового тестирования: нисходящее и восходящее.

Нисходящее тестирование. Нисходящее тестирование начинается с верхнего, головного модуля программы. Строгой, корректной процедуры подключения очередного последовательно тестируемого модуля не существует. Единственное правило, которым следует руководствоваться при выборе очередного модуля, состоит в том, что им должен быть один из модулей, вызываемых модулем, предварительно прошедшим тестирование.

Для иллюстрации этой стратегии рассмотрим рис. 21. Изображенная на нем программа состоит из 12 модулей A—L. Допустим, что модуль J содержит операции чтения из внешней памяти, а модуль I — операции записи.

Первый шаг — тестирование модуля A. Для его выполнения необходимо написать модули-заглушки, замещающие модули B, C и D. Зачастую неверно понимают функции, выполняемые модулями-заглушками. Так, утверждения: «заглушка должна только выполнять вывод сообщения, о поключении модуля» или «достаточно, чтобы заглушка существовала, не выполняя никаких действий вообще» в большинстве слу-

---

исполняемых одновременно модулей и насколько сложны заглушки.

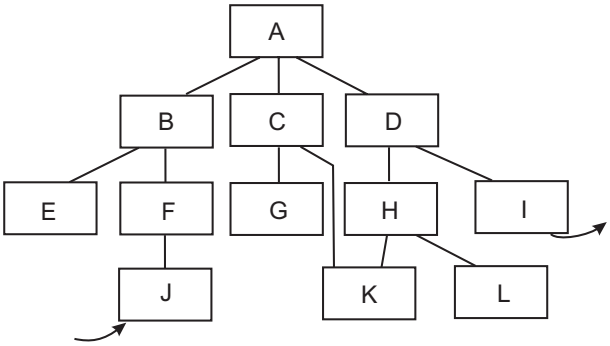


Рис. 21. Пример программы, состоящей из двенадцати модулей

чаев ошибочны. Когда модуль A вызывает модуль B, A предполагает, что B выполняет некую работу, и модуль A получает результаты работы модуля B (например, в форме значений выходных переменных). Когда модуль B просто возвращает управление или выдает сообщение без передачи в модуль A определенных осмысленных результатов, последний работает неверно не вследствие ошибок в самом модуле, а из-за несоответствия ему модуля-заглушки. Более того, результат может оказаться неудовлетворительным, если ответ модуля-заглушки не меняется в зависимости от разных тестовых данных. Следовательно, создание модулей-заглушек — задача нетривиальная.

При обсуждении метода нисходящего тестирования часто упускают еще одно положение, а именно форму представления тестов в программе. В нашем примере вопрос состоит в том, как тесты должны быть переданы модулю A? Ответ на этот вопрос не является совершенно очевидным, поскольку верхний модуль в типичной программе сам не получает входных данных и не выполняет операций ввода-вывода. Верхнему модулю (в нашем случае модулю A) данные передаются через одну или несколько заглушек.

Возникает еще одна проблема: поскольку модуль A вызывает модуль B (который обеспечивает ввод теста) один раз, следует решить, каким образом передать в A несколько тестов. Одно из решений состоит в том, чтобы вместо модуля B сделать несколько версий заглушки, каждая из которых имеет один фиксированный набор тестовых данных. Тогда для пропуска любого тестового набора нужно несколько раз исполнить программу, причем всякий раз с новой версией модуля-заглушки, замеща-

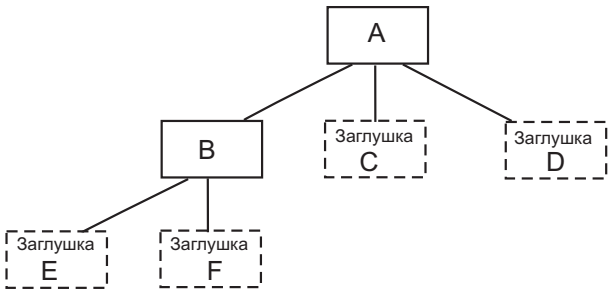


Рис. 22. Второй шаг при несходящем тестировании

ющего модуль *B*. Другой вариант решения — записать наборы тестов во внешнюю память, заглушкой читать их и передавать в модуль *A*. В общем случае создание заглушки может быть более сложной задачей, чем в рассмотренном выше примере. Кроме того, нередко из-за характеристик программы оказывается необходимым сообщать тестируемому модулю данные от нескольких заглушек, замещающих модули нижнего уровня.

После завершения тестирования модуля A одна из заглушек заменяется реальным модулем и добавляются заглушки, необходимые уже этому модулю. Например, на рис. 22 представлена следующая версия программы.

После тестирования верхнего (головного) модуля тестирование может выполняться в различных последовательностях. Так, если последовательно тестируются все модули, то возможны следующие варианты:

ABCDEFGHIJKL ABEFJCGKDHLI ADHIKLCGBFJE ABFJDIECG

При параллельном выполнении тестирования могут встречаться иные последовательности. Например, после тестирования модуля A одним программистом может тестироваться последовательность $A—B$, другим — $A—C$, третьим — $A—D$. В принципе, нет такой последовательности, которой бы отдавалось предпочтение, но рекомендуется придерживаться двух основных правил:

1. Если в программе есть критические в каком-либо смысле части (возможно, модуль *G*), то целесообразно выбирать последовательность, которая включала бы эти части как можно раньше. Критическими могут быть сложный модуль, модуль с новым алгоритмом или модуль

со значительным числом предполагаемых ошибок (модуль, склонный к ошибкам).

2. Модули, включающие операции ввода-вывода, также необходимо подключать в последовательность тестирования как можно раньше.

Целесообразность первого правила очевидна, второе же следует обсудить дополнительно. Напомним, что при проектировании заглушек возникает проблема, заключающаяся в том, что одни из них должны содержать тесты, а другие — организовывать выдачу результатов. Если к программе подключается реальный модуль, содержащий операции ввода, то представление тестов значительно упрощается. Форма их представления становится идентичной той, которая используется в реальной программе для ввода данных. Точно так же, если подключаемый модуль содержит функции, отображающие результаты работы программы, то отпадает необходимость в заглушках, показывающих результаты тестирования. Пусть, например, модули J и I выполняют функции ввода-выхода, а G — некоторая критическая функция; тогда пошаговая последовательность может быть следующей: $ABFJDICGEKHL$ и после шестого шага становится такой, как показано на рис. 23.

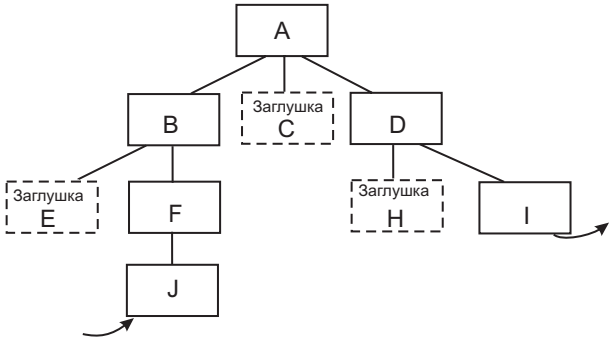


Рис. 23. Промежуточное состояние при нисходящем тестировании

При достижении этой стадии представление тестов и анализ результатов тестирования существенно упрощаются. Появляются дополнительные преимущества — имеется рабочая версия программы, выполняющая реальные операции ввода-вывода, но часть внутренних функций имитируется заглушками. Эта рабочая версия позволяет выявить ошибки и проблемы, связанные с организацией взаимодействия с человеком; она дает возможность продемонстрировать программу пользова-

телю (и заказчику) и является положительным моральным стимулом[1]. Все это, безусловно, достоинства стратегии нисходящего тестирования.

Однако нисходящее тестирование имеет ряд серьезных недостатков. Пусть состояние проверяемой программы соответствует показанному на рис. 23. На следующем шаге нужно заменить заглушку модулем *H*. Для тестирования этого модуля требуется спроектировать (или они спроектированы ранее) тесты по правилам, приведенным ранее. Заметим, что все тесты должны быть представлены в виде реальных данных, вводимых модулем *J*. При этом возникают две проблемы. Во-первых, между модулями *H* и *J* имеются промежуточные модули (*F, B, A* и *D*), поэтому может оказаться невозможным передать модулю такой текст, который соответствовал бы каждой предварительно описанной ситуации на входе модуля *H*. Во-вторых, даже если есть возможность передать все тесты, то из-за «дистанции» между модулем *H* и точкой ввода в программу возникает довольно трудная интеллектуальная задача — оценить, какими должны быть данные на входе модуля *J*, чтобы они соответствовали требуемым тестам модуля *H*.

Третья проблема состоит в том, что результаты выполнения теста демонстрируются модулем, расположенным довольно далеко от модуля, тестируемого в данный момент. Следовательно, установление соответствия между тем, что демонстрируется, и тем, что происходит в модуле на самом деле, достаточно сложно, а иногда просто невозможно. Допустим, мы добавляем к схеме, приведенной на рис. 23, модуль *E*. Результаты каждого теста определяются путем анализа выходных результатов модуля *I*, но из-за стоящих между модулями *E* и *I* промежуточных модулей трудно восстановить действительные выходные результаты модуля *E* (т. е. те результаты, которые он передает в модуль *B*).

Последняя проблема заключается в том, что на практике часто переходят к тестированию следующего модуля до завершения тестирования предыдущего. Это объясняется двумя причинами: во-первых, трудно передавать тестовые данные через модули-заглушки; во-вторых, модули верхнего уровня используют ресурсы модулей нижнего уровня. Из рис. 21 видно, что тестирование модуля *A* может потребовать несколько версий заглушки модуля *B*. Программист, тестирующий программу, ре-

[1] Нередко сроки реализации программных продуктов срываются. Причиной этого является наша неспособность достаточно точно оценить сроки реализации ПО, особенно большого объема. Поэтому наличие рабоющей версии программы очень важно. Мы сами в состоянии увидеть результаты нашей работы, показать ее заказчику и при необходимости договориться о продлении ее срока.

шает так: «Я сразу не буду полностью тестировать модуль A — сейчас это трудная задача. После подключения модуля J станет легче представлять тесты, и тогда я вернусь к тестированию модуля A». Конечно, здесь важно не забыть проверить оставшуюся часть модуля тогда, когда это предполагалось сделать. Аналогичная проблема возникает в связи с тем, что модули верхнего уровня также запрашивают ресурсы для использования их модулями нижнего уровня (например, открывают файлы). Иногда трудно определить, корректно ли эти ресурсы были запрошены (например, верны ли атрибуты открытия файлов) до того момента, пока не начнется тестирование использующих их модулей нижнего уровня.

Г. Майерсом приводится еще одна проблема, связанная с различным пониманием процесса нисходящего проектирования различными специалистами (см. работы [2, 45]) и приводящая, по его мнению, к совмещению процесса проектирования и тестирования. (Достоинство это или недостаток каждая группа специалистов решает по своему, я поддерживаю точку зрения Э. Йодана [45].)

Восходящее тестирование. Рассмотрим восходящую стратегию пошагового тестирования. Во многих отношениях восходящее тестирование противоположно нисходящему; преимущества нисходящего тестирования становятся недостатками восходящего тестирования и, наоборот, недостатки нисходящего тестирования становятся преимуществами восходящего. Имея это в виду, обсудим кратко стратегию восходящего тестирования.

Данная стратегия предполагает начало тестирования с модулей, не вызывающих другие модули. Как и ранее, здесь нет четкой процедуры для выбора модуля, тестируемого на следующем шаге, которой бы отдавалось предпочтение. Единственное правило состоит в том, чтобы очередной модуль вызывал уже оттестированные модули.

Если вернуться к рис. 21, то первым шагом должно быть тестирование модулей *E, J, G, K, L* и *I* последовательно или параллельно. Для каждого из них требуется свой модуль-драйвер, т. е. модуль, который вызывает тестируемый модуль, передавая ему тестовые данные, и отображает полученные результаты. В отличие от заглушек, драйвер может последовательно вызывать тестируемый модуль несколько раз. В большинстве случаев драйверы проще разработать, чем заглушки.

Как и в предыдущем случае, на последовательность тестирования влияет критичность природы модуля. Если мы решаем, что наиболее

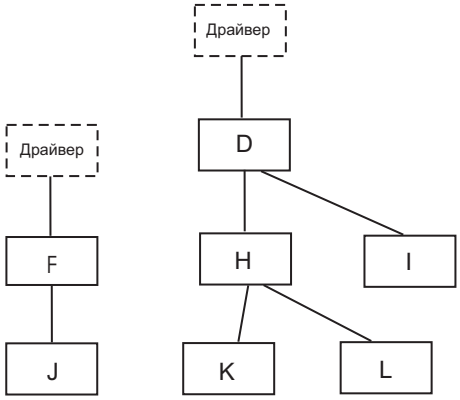


Рис. 24. Промежуточное состояние при восходящем тестировании

критичны модули D и F, то промежуточное состояние будет соответствовать изображенному на рис. 24. Следующими шагами могут быть тестирование модуля E, затем модуля B и объединение модуля B с предварительно оттестированными модулями *E, F, J.*

Недостаток рассматриваемой стратегии заключается в том, что концепция построения структуры рабочей программы на ранней стадии тестирования отсутствует. Действительно, рабочая программа не существует до тех пор, пока не добавлен последний модуль (в примере модуль A), и это уже готовая программа.

Здесь отсутствуют проблемы, связанные с невозможностью или трудностью создания всех тестовых ситуаций, характерные для нисходящего тестирования. Драйвер как средство тестирования применяется непосредственно к тому модулю, который тестируется, нет промежуточных модулей, которые следует принимать во внимание. Не существует также и трудностей с незавершенностью тестирования одного модуля при переходе к тестированию другого, потому что при восходящем тестировании нет сложностей с представлением тестовых данных.

Сравнение этих пошаговых стратегий. При сравнении стратегий нисходящего и восходящего тестирования нельзя противопоставить их друг другу, как это делалось при сравнении пошагового и монолитного подходов. В табл. 11. показаны их относительные недостатки и преимущества. Первое преимущество каждого из методов могло бы

явиться решающим фактором, однако трудно сказать, где больше ошибок: в модулях верхнего уровня или модулях нижних уровней типичной программы. Поэтому при выборе стратегии целесообразно взвесить все пункты таблицы с учетом характеристик конкретной программы.

В заключение отметим, что рассмотренные стратегии нисходящего и восходящего тестирования не являются единственно возможными при пошаговом подходе.

Замечание. Несмотря на большое число недостатков нисходящего тестирования, часто предпочтение отдается именно этому подходу по причине «главного» преимущества — раннего формирования работающей версии программы.

5.3. Исполнение теста

Исполнение теста — завершающий этап тестирования модулей. Далее описан определенный набор рекомендаций и положений, которых следует придерживаться на этом этапе.

Если исполнение теста приносит результаты, не соответствующие предполагаемым, то это означает, что либо модуль имеет ошибку, либо неверны предполагаемые результаты (ошибка в тесте). Для устранения такого рода недоразумений нужно тщательно проверять набор тестов («тестировать» тесты).

Применение автоматизированных средств позволяет снизить трудоемкость процесса тестирования. Например, существуют средства, которые позволяют избавиться от потребности в драйверах (например, отладчики). Системы контроля полноты набора тестов для определенного набора критериев позволяют определять непокрытые тестовые ситуации и далее дорабатывать наборы тестов (о таких системах сказано ниже). Потоковые анализаторы и инструменты обнаружения ошибок на их основе дают возможность статически выявлять некоторые классы ошибок [24].

При подготовке к тестированию модулей целесообразно еще раз пересмотреть психологические и экономические принципы, обсуждавшиеся ранее. Вспомните, что, как показано ранее, необходимой частью тестового набора является описание ожидаемых результатов.

Во время тестирования модулей возникают и психологические проблемы, связанные с личностью тестирующего. Программистам полезно поменяться модулями, чтобы не тестировать свои собственные. Так, программист, сделавший вызывающий модуль, является хорошим кан-

Таблица 11

Сравнение нисходящего и восходящего тестирования

Преимущества	Недостатки
Нисходящее тестирование	
1. Имеет преимущества, если ошибки допущены главным образом в верхней части программы. 2. Представление теста облегчается после подключения функций ввода-вывода. 3. Раннее формирование структуры программы позволяет провести ее демонстрацию пользователю и служит моральным стимулом	1. Необходимо разрабатывать модули-заглушки, которые часто оказываются сложнее, чем кажется вначале. 2. До применения функций ввода-вывода может быть сложно представлять тестовые данные в заглушки. 3. Может оказаться трудным или невозможным создать тестовые условия. 4. Сложнее оценка результатов тестирования. 5. Стимулируется незавершение тестирования некоторых модулей
Восходящее тестирование	
1. Имеет преимущества, если ошибки допущены главным образом в модулях нижнего уровня. 2. Легче создавать тестовые условия. 3. Проще оценка результатов	1. Необходимо разрабатывать модули-драйверы. 2. Программа как единое целое не существует до тех пор, пока не добавлен последний модуль

дидатом для тестирования вызываемого модуля. Заметим, что это относится только к тестированию, а не к отладке, которую всегда должен выполнять автор модуля. Не следует выбрасывать результаты тестов; представляйте их в такой форме, чтобы можно было повторно воспользоваться ими в будущем. Если в некотором подмножестве модулей обнаружено большое число ошибок, то эти модули должны стать объектом дальнейшего тестирования. Наконец, следует помнить, что задача тестирования заключается не в демонстрации корректной работы модулей, а в выявлении ошибок.

5.4. Методы ручного тестирования

Рассмотрим несколько нетрадиционных методов тестирования, основанных не на применении ЭВМ, а на методах так называемого «ручного» тестирования. Эксперименты показали, что методы ручного тестирования достаточно эффективны с точки зрения нахождения ошибок. Описанные здесь методы предназначены для периода разработки, когда программа закодирована, но тестирование на ЭВМ еще не началось.

Следует заметить, что из-за неформальной природы методов ручного тестирования первой реакцией часто является скептицизм, ощущение того, что простые и неформальные методы не могут быть полезными. Однако их использование показало, что они не «уводят в сторону». Скорее эти методы способствуют существенному увеличению производительности и повышению надежности программы. Во-первых, они обычно позволяют раньше обнаружить ошибки, уменьшить стоимость исправления последних и увеличить вероятность того, что корректировка произведена правильно. Во-вторых, психология программистов, повидимому, изменяется, когда начинается тестирование на ЭВМ. Возрастает внутреннее напряжение и появляется тенденция исправлять ошибки как можно быстрее. В результате программисты допускают больше промахов при корректировке ошибок, уже найденных во время тестирования на ЭВМ, чем при корректировке ошибок, найденных на более ранних этапах.

5.4.1. Инспекции и сквозные просмотры

Инспекции исходного текста и сквозные просмотры являются основными методами ручного тестирования. Так как эти два метода имеют много общего, они рассматриваются здесь сначала совместно. Различия между ними обсуждаются в представленных далее пунктах.

Инспекции и сквозные просмотры включают в себя чтение или визуальную проверку программы группой лиц. Оба метода предполагают некоторую подготовительную работу. Завершающим этапом является «обмен мнениями» — собрание, проводимое участниками проверки. Цель такого собрания — выявление ошибок, но не их устранение (т. е. тестирование, а не отладка)[1].

Фактически **инспекция** и **сквозной просмотр** — просто новые названия старого метода «проверки за столом» (состоящего в том, что программист просматривает свою программу перед ее тестированием), однако они гораздо эффективнее по причине того, что в процессе участвует не только автор программы, но и другие лица. Результатом использования этих методов, по-видимому, также является более низкая стоимость отладки (исправления ошибок), так как во время поиска ошибок обычно точно определяется их природа. Кроме того, с помощью данных методов обнаруживают группы ошибок, что позволяет в дальнейшем корректировать сразу несколько ошибок. При тестировании на ЭВМ обычно выявляют только симптомы ошибок (например, программа не завершилась или напечатала некорректный результат), а сами они определяются в дальнейшем.

Эксперименты по применению этих методов показали, что с их помощью для типичных программ можно находить от 30 до 70 % ошибок логического проектирования и кодирования. В работе Г. Майерса [2] делается также вывод о том, что при нахождении определенных типов ошибок ручное тестирование более эффективно, чем тестирование на ЭВМ, в то время как для других типов ошибок верно противоположное утверждение. Поэтому предлагается, чтобы использование инспекций, сквозных просмотров и тестирование с использованием ЭВМ дополняли друг друга. Эффективность обнаружения ошибок может уменьшиться, если тот или иной из этих подходов не будет использован.

Наконец, хотя эти методы весьма важны при тестировании новых программ, они представляют не меньшую ценность при тестировании модифицированных программ. Опыт показал, что в случае модификации существующих программ вносится большее число ошибок (измеряемое числом ошибок на вновь написанные операторы), чем при написании новой программы. Следовательно, модифицированные программы также должны быть подвергнуты тестированию с применением данных методов.

[1] Данное утверждение поддерживается не всеми специалистами.

Инспекции исходного текста. Инспекции исходного текста представляют собой набор приемов обнаружения ошибок при изучении (чтении) текста группой специалистов.

Инспектирующая группа обычно состоит из четырех человек, один из них выполняет функции председателя. Председатель должен быть компетентным программистом, но не автором программы; он не должен быть знаком с ее деталями. В обязанности председателя входят подготовка материалов для заседаний инспектирующей группы и составление графика их проведения, ведение заседаний, регистрация всех найденных ошибок и принятие мер по их последующему исправлению. Председателя можно сравнить с инженером отдела технического контроля. Членами группы являются автор программы, проектировщик (если он не программист) и специалист по тестированию.

Общая процедура заключается в следующем. Председатель заранее (например, за несколько дней) раздает листинг программы и проектную спецификацию остальным членам группы. Они знакомятся с материалами до заседания. Инспекционное заседание разбивается на две части:

1. Программиста просят рассказать о логике работы программы. Во время беседы возникают вопросы, преследующие цель выявления ошибок. Практика показала, что рассказ о своей программе слушателям представляется уже эффективным методом обнаружения ошибок и многие ошибки находит сам программист, а не другие члены группы.

2. Программа анализируется по списку вопросов для выявления исторически сложившихся общих ошибок программирования данного коллектива.

Председатель является ответственным за обеспечение результативности дискуссии. Ее участники должны сосредоточить свое внимание на нахождении ошибок, а не на их корректировке. (Исправление ошибок выполняется программистом после инспекционного заседания.)

По окончании заседания программисту передается список найденных ошибок. Если список включает много ошибок или если эти ошибки требуют внесения значительных изменений, председателем может быть принято решение о проведении после корректировки повторной инспекции программы. Список анализируется и ошибки распределяются по категориям, что позволяет совершенствовать его в целях повышения эффективности будущих инспекций.

Время и место проведения инспекции должны быть спланированы так, чтобы избежать любых прерываний инспекционного заседания.

Его оптимальная продолжительность, по-видимому, лежит в пределах 1,5—2 часов. Так как это заседание является мозговым штурмом, увеличение его продолжительности ведет к снижению продуктивности. При этом подразумевается, что большие программы должны рассматриваться за несколько инспекций, каждая из которых может быть связана с одним или несколькими модулями или подпрограммами.

Для того чтобы инспекция была эффективной, должны быть установлены соответствующие отношения. Если программист воспринимает инспекцию как акт, направленный лично против него, и, следовательно, занимает оборонительную позицию, процесс инспектирования не будет эффективным. Программист должен рассматривать инспекцию в позитивном и конструктивном свете: объективно инспекция является процессом нахождения ошибок в программе и таким образом улучшает качество его работы. По этой причине рекомендуется результаты инспекции считать конфиденциальными материалами, доступными только участникам заседания. В частности, использование результатов инспекции руководством может нанести ущерб целям этого процесса.

Процесс инспектирования в дополнение к своему основному назначению, заключающемуся в нахождении ошибок, выполняет еще ряд полезных функций. Результаты инспекции позволяют программисту увидеть сделанные им ошибки и способствуют его обучению на собственных ошибках, он обычно получает возможность оценить свой стиль программирования, выбор алгоритмов и методов тестирования. Остальные участники также приобретают опыт, рассматривая ошибки и стиль программирования других программистов.

Наконец, инспекция является способом раннего выявления наиболее склонных к ошибкам частей программы, позволяющим сконцентрировать внимание на этих частях в процессе выполнения дальнейшего тестирования на ЭВМ (один из принципов тестирования).

Сквозные просмотры. Сквозной просмотр, как и инспекция, представляет собой набор процедур и методов обнаружения ошибок, осуществляемых группой лиц, просматривающих текст программы. Такой просмотр имеет много общего с процессом инспектирования, но отличаются их процедуры и используются другие методы обнаружения ошибок.

Подобно инспекции, сквозной просмотр проводится как непрерывное заседание, продолжающееся 1—2 часа. Группа по выполнению сквозного просмотра состоит из 3—5 человек. В нее входят председатель,

функции которого подобны функциям председателя в группе инспектирования, секретарь, который записывает все найденные ошибки, и специалист по тестированию. Мнения о том, кто должен быть четвертым и пятым членами группы, расходятся. Конечно, одним из них должен быть программист. Относительно пятого участника имеются следующие предположения: 1) высококвалифицированный программист; 2) эксперт по языку программирования; 3) начинающий (на точку зрения которого не влияет предыдущий опыт); 4) человек, который будет в конечном счете эксплуатировать программу; 5) участник какого-нибудь другого проекта; 6) кто-либо из той же группы программистов, что и автор программы.

Начальная процедура при сквозном просмотре такая же, как и при инспекции: участникам за несколько дней до заседания раздаются материалы, позволяющие им ознакомиться с программой. Однако процедура заседания отличается от процедуры инспекционного заседания. Вместо того чтобы просто читать текст программы и использовать список ошибок, участники заседания «выполняют роль вычислительной машины». Лицо, назначенное тестирующим, предлагает собравшимся небольшое число подготовленных тестов, представляющих собой наборы входных данных (и ожидаемых результатов) для программы или модуля. Во время заседания каждый тест выполняется. Это означает, что тестовые данные подвергаются обработке в соответствии с логикой программы. Состояние программы (т. е. значения переменных) отслеживается на бумаге или доске.

Конечно, число тестов должно быть небольшим и они должны быть простыми по своей природе, потому что скорость выполнения программы человеком на много порядков меньше, чем у машины. Следовательно, тесты сами по себе не играют критической роли, скорее они служат средством для первоначального понимания программы и основой для вопросов программисту о логике проектирования и принятых допущениях. В большинстве сквозных просмотров при выполнении самих тестов находят меньше ошибок, чем при опросе программиста.

Как и при инспекции, мнение участников является решающим фактором. Замечания должны быть адресованы программе, а не программисту. Другими словами, ошибки не должны рассматриваться как слабость человека, который их совершил, не должны быть мотивом для его критики и унижения. Они свидетельствуют о сложности процесса создания программ и неидеальности человеческого разума.

Сквозные просмотры должны протекать так же, как и описанный ранее процесс инспектирования. Побочные эффекты, получаемые во время выполнения этого процесса (установление склонных к ошибкам частей программы и обучение присутствующих на основе анализа ошибок, стиля программирования и применяемых методов), характерны и для процесса сквозных просмотров.

6. КРИТЕРИЙ ЗАВЕРШЕНИЯ ТЕСТИРОВАНИЯ

Одним из наиболее трудных является вопрос о том, когда следует закончить тестирование программы, поскольку не представляется возможным определить, является ли выявленная ошибка последней. Действительно, даже для небольших программ было бы неразумно полагать, что в итоге будут найдены все ошибки. Тем не менее тестирование придется завершать хотя бы по экономическим соображениям. Этот вопрос решают обычно либо чисто волевым способом, либо с помощью какого-нибудь критерия. Наибольшее распространение получили следующие критерии:

1. Время, отведенное по графику работ на тестирование, истекло.
2. Все тесты выполнены без выявления ошибок (т. е. они неудачны).

Оба эти критерия не годятся; первый можно выполнить ничего не делая (он не содержит оценки качества тестирования); второй плох по причине того, что он не зависит от качества тестов. Кроме того, второй критерий непродуктивен, так как подсознательно побуждает к построению тестов с низкой вероятностью обнаружения ошибок. Как отмечалось ранее, мышление человека ориентируется на поставленную цель. Если программист ставит перед собой цель — «не перетрудиться», т. е. решить задачу при неудачных тестах, то он будет подсознательно разрабатывать тесты, которые ведут его к этой цели, избегая построения полезных деструктивных тестов с высокой обнаруживающей способностью.

Существует три категории более или менее приемлемых критериев. К первой из них относятся критерии, основанные на использовании определенных методологий проектирования тестов. Можно определить условия завершения тестирования каждой фазы тестирования с помощью тестов, получаемых на основании определенных комплексных критериев тестирования. Например, условие завершения тестирования модулей может требовать построения тестов на основе критериев: эквивалентное разбиение, анализ граничных значений и C1. Все получающиеся в результате тесты должны стать неудачными.

Эти критерии лучше двух определенных ранее, но с ними связаны три проблемы. Во-первых, они бесполезны на той фазе тестирования, когда определенные методологии становятся непригодными. Во-вторых, такое измерение субъективно, так как нет гарантии, что данный специалист использовал нужную методологию (например, анализ граничных значений) правильно и точно[1]. В-третьих, вместо того чтобы поставить цель и дать возможность выбора наиболее подходящего пути ее достижения, рассмотренные критерии предписывают использование конкретных методологий проектирования тестов, но не ставят никакой цели[2]. Таким образом, критерии первой категории полезны и необходимы на начальных фазах тестирования и их следует применять.

Критерий второй категории, видимо, представляет наибольшую ценность, так как четко формулирует условие завершения тестирования. Поскольку цель тестирования — поиск ошибок, почему бы в качестве критерия не выбрать некоторое заранее установленное число ошибок? Например, можно установить, что проверка модуля завершается после обнаружения трех ошибок. Возможно, условием завершения проверки системы следует считать обнаружение и устранение 90 ошибок или общее время тестирования — 3 месяца, смотря по тому, какое из событий наступит раньше.

Заметим, что такой критерий подтверждает определение тестирования. С ним связаны две, в принципе, решаемые проблемы. Одна из них заключается в установлении способа получения числа выявляемых ошибок. Для этого требуется:

1. Оценить общее число ошибок в программе.

2. Выяснить, какой процент этих ошибок можно обнаружить с помощью тестирования.

3. Определить, какая именно часть ошибок возникла в процессе проектирования и во время каких фаз тестирования целесообразно их выявлять.

Грубую оценку общего числа ошибок можно получить несколькими методами. Один из них основан на анализе данных о разработке других программ. Существует целый ряд моделей предсказания, часть которых требует тестировать программу в течение какого-то периода вре-

[1]Следует отметить, что полноту тестов для критериев белого ящика мы можем контролировать достаточно эффективно.

[2]Можно и не согласиться с данным пунктом: цель поставлена (разработать полный набор тестов по комплексному критерию) и строго определен путь, но не тестировщиком, а выбор отсутствует.

мени, фиксировать интервалы времени между обнаружением последовательных ошибок и затем подставлять эти интервалы как параметры в формулу. Некоторые модели предполагают, что в программу искусственно вносятся ошибки, но об их наличии не объявляется. Программа тестируется определенное время, после чего проверяется соотношение обнаруженных и необнаруженных ошибок из числа тех, которые были в нее внесены. Существуют модели, предполагающие, что в течение определенного времени программа тестируется двумя независимыми тестовыми бригадами, а затем анализируются ошибки, найденные каждой бригадой, и ошибки, обнаруженные обеими бригадами вместе. Эти параметры используются для оценки общего числа ошибок.

Выяснение процента ошибок, которые можно найти методом тестирования, должно учитывать природу программы и последствия невыявленных ошибок и может быть в некоторой степени произвольным.

Другая очевидная проблема, связанная с критерием второй категории, — это проблема переоценки. Что если к моменту начала тестирования определенной фазы остается ошибок меньше, чем требуется определить? Основываясь на данном критерии, завершить данную фазу не удастся никогда. Возникает довольно странная ситуация: не хватает ошибок, программа оказывается слишком «хорошей». Казалось бы, проблемы не должно быть, поскольку это именно то, к чему мы стремимся. Однако если она возникает, то для ее решения требуется лишь немного здравого смысла. Если в заданное время не удается найти требуемое число ошибок, руководитель проекта может пригласить со стороны эксперта, который, проанализировав тесты, выскажет свое мнение о причинах возникновения проблемы: тесты неадекватны (1) или удачны (2), но в программе действительно мало ошибок.

Критерий третьей категории основан в значительной степени на здравом смысле и интуиции. Для его применения требуется строить график числа ошибок, найденных в единицу времени. Анализируя форму кривой, часто можно определить, следует ли продолжать тестирование на данной фазе или закончить ее и перейти к следующей.

Пусть некоторая программа проходит некоторую фазу тестирования. Строим график числа ошибок, найденных в ней за неделю. Даже если удовлетворяется критерий необходимого числа найденных ошибок, кривая за семь недель тестирования такова (рис. 25, *слева*), что неразумно прекратить тестирование программы на этой фазе. Поскольку скорость обнаружения ошибок велика (находят много ошибок), поэто-

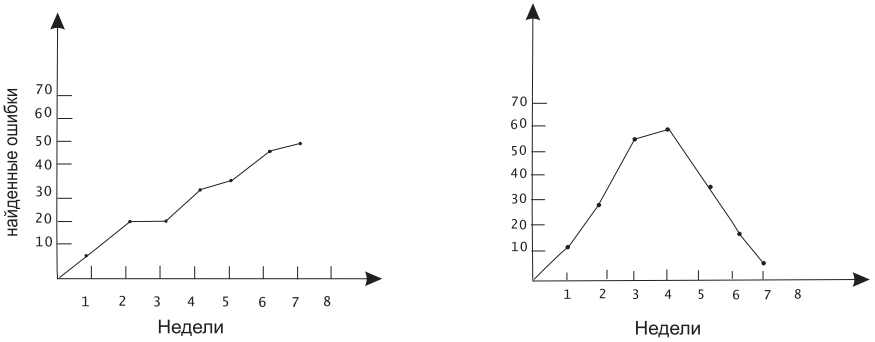


Рис. 25. Оценка завершения тестирования по графику ошибок, выявленных в единицу времени

му наиболее правильное решение — продолжить тестирование на данной фазе, проектируя дополнительные тесты.

Если график выявленных ошибок имеет вид, представленный на правой части рис. 25, что означает: эффективность обнаружения ошибок значительно уменьшилась, можно закончить тестирование на данной фазе и перейти к следующей. Конечно, необходимо рассмотреть и другие факторы, которые могли привести к уменьшению эффективности обнаружения ошибок, например невысокую обнаруживающую способность имеющихся тестов и т. п.

На рис. 26 показано, что происходит, когда забывают строить график числа обнаруженных ошибок. Диаграмма описывает три фазы тестирования очень большой системы программного обеспечения; она построена как результат исследования этого проекта после его завершения. Вывод очевиден: после шестого периода не следовало переходить к другим фазам тестирования. На протяжении всего шестого периода скорость обнаружения ошибок была высокой (а для тестирования чем выше эта скорость, тем лучше), переход же на следующую фазу привел к ее значительному снижению.

Наилучшим критерием завершения тестирования является, видимо, комбинация трех рассмотренных выше критериев. Для тестирования модулей оптимальным будет, очевидно, первый критерий завершения. На других фазах тестирования критерием завершения может служить останов по достижении заранее заданного числа обнаруженных ошибок или по достижении момента, предопределенного графиком работ, смот-

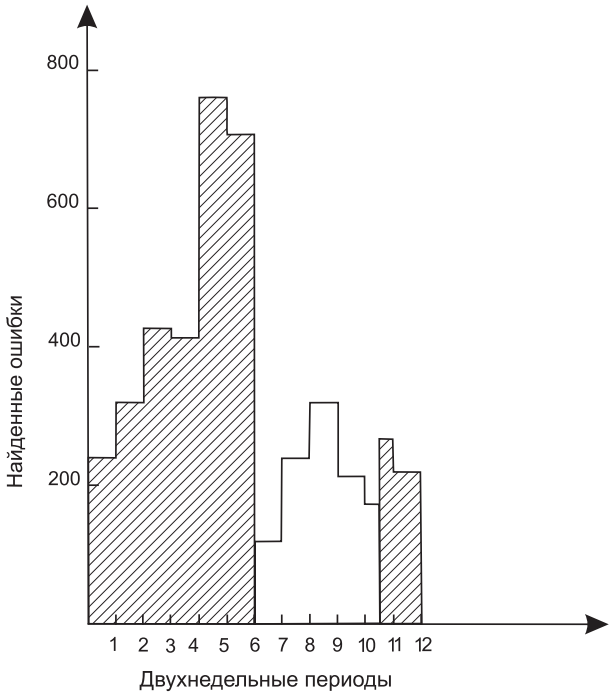


Рис. 26. Исследование процесса тестирования по завершении большого проекта

ря по тому, такое событие наступит раньше (при условии, что анализ диаграммы зависимости числа ошибок от времени тестирования покажет снижение продуктивности этой проверки).

7. АВТОМАТИЗАЦИЯ ТЕСТИРОВАНИЯ

При тестировании программного обеспечения наиболее трудоемкими являются следующие этапы тестирования:

— создание хорошего набора тестов;

— оценка имеющего набора тестов на полноту по совокупности критериев;

— исполнение и оценка результатов этого исполнения для тестового набора.

Остановимся на имеющихся подходах автоматизации этих этапов, при этом основное внимание уделим двум первым.

7.1. Автоматизация построения тестов

Одним из наиболее известных подходов автоматизации построения тестов является *символьное* исполнение программ. В середине 70-х гг. несколькими коллективами независимо и практически одновременно были разработаны экспериментальные системы построения тестов с использованием символьного выполнения программ [5]. Часть систем предлагали полную автоматизацию генерации тестов.

7.1.1. Символьное исполнение программ и построение тестов

Символьное исполнение первого вида. Цель символьного выполнения программы первого вида — определить по заданному пути программы, каким условиям должны удовлетворять входные данные, чтобы выполнение программы на этих данных осуществлялось именно по этому пути.

При символьном выполнении программы арифметические вычисления заменяются подстановкой символьных значений данных и фактически проводятся алгебраические вычисления, но без приведения подобных.

Рассмотрим небольшой пример, который прояснит существо подхода.

Пример 1.Дана программа

```
PROCEDURE POWER(X,Y);          1
     Z := 1;                   2
     J := 1;                   3
Lab: IF Y >= J THEN            4
     Z := Z*X;                 5
     J := J+1;                 6
     GOTO Lab; END;            7
     RETURN(Z);                8
END POWER;                     9
```

Рассмотрим, каким условиям должны удовлетворять входные данные, чтобы был выполнен путь {1, 2, 3, 4, 5, 6, 7, 4, 8}? Можно получить соответствующие ограничения следующим образом. После выполнения каждого оператора будем записывать символьные значения всех переменных программы и отношения между ними (будем называть это состоянием). Тогда начальное состояние выглядит так:

а) значения переменных: $X = \alpha_1$, $Y = \alpha_2$, Z и J не заданы;

б) отношения: нет.

Здесь α_1 и α_2 — символьные значения входных данных. Все остальные состояния после выполнения операторов нашего заданного пути приведены в табл. 12.

Таблица 12

Состояния для заданного пути

Вершина	X	Y	Z	J	**Отношения**
2	α_1	α_2	1	Неопределено	Нет
3	α_1	α_2	1	1	Нет
4	α_1	α_2	1	1	$\alpha_2 \geq 1$
5	α_1	α_2	$1 * \alpha_1$	1	$\alpha_2 \geq 1$
6	α_1	α_2	$1 * \alpha_1$	1+1	$\alpha_2 \geq 1$
7	α_1	α_2	$1 * \alpha_1$	1+1	$\alpha_2 \geq 1$
4	α_1	α_2	$1 * \alpha_1$	1+1	$\alpha_2 \geq 1$, $\alpha_2 < 1 + 1$
8	α_1	α_2	$1 * \alpha_1$	1+1	$\alpha_2 \geq 1$, $\alpha_2 < 1 + 1$

Состояние после выполнения оператора 8 содержит *формулу реализуемости пути* — необходимые и достаточные условия для выполнеия пути. Эти условия описываются системой неравенств из колонки «Отношения» табл. 12. Видно, что если $1 \leq \alpha_2 < 2$, то будет выполняться нами рассматриваемый путь. Если бы формула реализуемости пути не имела решения, то это означало бы, что данный путь невыполним ни при каких входных данных.

Помимо главной задачи — определения формулы реализуемости пути, мы получили также некоторую дополнительную полезную информацию: 1)символьное значение результата процедуры при выполнении данного пути (значение переменной Z); 2)параметр процедуры X не влияет на выполнение данного пути.

На основании формулы реализуемости пути мы можем сгенерировать тест. В нашем случае это пара чисел, где первое произвольное, а второе есть любое решение неравенства $1 \leq \alpha_2 < 2$.

В данном примере мы реализовали символьное исполнение программы с начала пути, однако возможно символьное исполнение и с конца пути. Символьное исполнение с конца пути позволяет вообще не рассматривать те переменные, которые не влияют на передачу управления и не хранить символьные значения переменных, а хранить только набор создаваемых ограничений.

Метод символьного исполнения программы в обратном направлении заключается в следующем. Пусть дан путь, для которого нужно построить условия реализуемости. Будем идти по нему от конца к началу. Если встречается какое-то условие, которое должно выполняться для того, чтобы программа исполнялась именно по этому пути, то оно заносится в список ограничений CL. Если же встречается присваивание $x := expr$, то все вхождения переменной x в ограничениях из CL заменяются на $expr$. В результате получаем CL — условия реализуемости данного пути.

Пример 2.Для процедуры `gcd` нужно построить условия реализуемости пути {a, b, c, e, a, b, d, e, a, f, g}[1].

```
 PROCEDURE gcd(x, y: INTEGER): INTEGER;
  BEGIN
      (*a*) WHILE x <> y DO
      (*b*)     IF x > y THEN
      (*c*)            x :=x-y;
                ELSE
      (*d*)            y:=y-x;
                END(*IF*);
      (*e*) END(*WHILE*);
      (*f*)      RETURN x;
(*g*)END gcd;
```

Работа приведенного выше алгоритма по шагам, соответствующим вершинам пути, представлена в табл. 13.

В результате получаем следующие условия реализуемости:

$$NOT(x - y <> y - (x - y)),$$
$$NOT(x - y > y),$$
$$x - y <> y,$$
$$x > y,$$
$$x <> y.$$

Первое ограничение эквивалентно $2 * x = 3 * y$, и если взять $x = 3$, $y = 2$, то видно, что остальные ограничения тоже удовлетворяются и,

[1]Данный пример демонстрирует работу одного из компонентов системы TGS проекта СОКРАТ [9, 10].

Таблица 13

Работа алгоритма символьного исполнения по шагам

Шаг(вершина)	**Ограничения**
g	Нет
f	Нет
a	$NOT(x <> y)$
e	$NOT(x <> y)$
d	$NOT(x <> y - x)$
b	$NOT(x <> y - x)$, $NOT(x > y)$
a	$NOT(x <> y - x)$, $NOT(x > y)$, $x <> y$
e	$NOT(x <> y - x)$, $NOT(x > y)$, $x <> y$
c	$NOT(x - y <> y - (x - y))$, $NOT(x - y > y)$, $x - y <> y$
b	$NOT(x - y <> y - (x - y))$, $NOT(x - y > y)$, $x - y <> y$, $x > y$
a	$NOT(x - y <> y - (x - y))$, $NOT(x - y > y)$, $x - y <> y$, $x > y$, $x <> y$

таким образом, условия реализуемости выполняются. Следовательно, получен тест, соответствующий данному пути.

Символьное исполнение второго вида. Целью символьного выполнения программы второго вида является определение по зафиксированным ограничениям на входные данные пути программы, который будет выполнен на этих данных, и символьных значений переменных в конце пути (формула-функция, реализуемая данной программой).

В качестве примера рассмотрим процедуру POWER первого примера. Обозначим, как и ранее, значение первого параметра через α_1, второго — через α_2. Допустим, что задано следующее ограничение на входные данные: $\alpha_2 < 1$.

Нетрудно установить, что состояние программы после выполнения операторов 1, 2, 3 будет соответствовать указанному в приведенной ниже таблице.

Выполн. путь	X	Y	Z	J	**Отношения**
{1, 2, 3}	α_1	α_2	1	1	$\alpha_2 < 1$

Это состояние позволяет нам утверждать, что условие оператора 4 ложно и поэтому следующим оператором будет выполнен оператор 8. Результатом работы процедуры будет значение $Z = 1$.

Следует, однако, заметить,что не во всех случаях можно определить, по которой из потенциально возможных ветвей программы надлежит продолжить символьное выполнение. Например, если в рассмотренном выше примере ограничение заменить на $\alpha_2 < 2$, то определить однозначно путь дальнейшего выполнения начиная с оператора 4 не удастся, так как возможны обе альтернативы (при $1 \leq \alpha_2 < 2$ управление передается на оператор 5, а при $\alpha_2 < 1$ — на оператор 8). В таких случаях исследуются обе альтернативы, процесс символьного выполнения разветвляется. Как установить, по какому пути продолжить символьное выполнение в случае разветвления? Можно использовать один из двух способов. Первый заключается в следующем: берем систему отношений над символьными входными данными перед выполнением IF-оператора (обозначим эту систему α). Добавляем к α булевское выражение из IF-оператора, в котором переменные заменены их символьными значениями (это выражение обозначим через β). Полученная система $\alpha + \beta$ является системой уравнений и неравенств над символьными входны-

ми значениями. Одновременно составляем и систему $\alpha + \neg\beta$. Решаем обе системы.

Если система α имеет решение, то возможны три случая:

1) $\alpha + \beta$ имеет решение, а $\alpha + \neg\beta$ решения не имеет;

2) $\alpha + \beta$ не имеет решения, а $\alpha + \neg\beta$ решение имеет;

3) как $\alpha + \beta$, так и $\alpha + \neg\beta$ имеет решение.

В случае 1 следует продолжить символьное выполнение по THEN-ветви, в случае 2 — по ELSE-ветви, а в случае 3 исполнение должно продолжаться по обеим ветвям.

Второй способ определяется формальным доказательством теорем. Обозначим через pc (path condition) конъюнкцию отношений над символьными входными данными из состояния перед выполнением IF-оператора.

Параллельно пытаемся доказать две теоремы: 1) $pc \supset \beta$ и 2) $pc \supset \neg\beta$. Лишь одна из этих теорем может оказаться истинной (исключая тривиальный случай, когда pc тождетвенно ложно). Если истинна теорема 1, то символьное выполнение следует продолжить по THEN-ветви; если истинна теорема 2, то — по ELSE-ветви. Если ни 1-ая, ни 2-ая теоремы не являются истинными выражениями, то должно произойти разветвление процесса символьного исполнения.

Преимущества символьного выполнения. Применение данного подхода имеет приведенные ниже основные преимущества.

1. Символьное выполнение первого вида применяется для генерации тестов и позволяет создавать тесты целенаправленно, учитывая логическую структуру программы. Таким образом, удается всесторонне протестировать программу, например автоматически создать минимальный набор тестов для критерия C1. Понятно, что при случайной генерации такое невозможно.

2. Символьное выполнение второго вида дает возможность одним выполнением программы на символьных входных данных заместить огромное (иногда бесконечное) число выполнений на конкретных значениях. Если при этом удается разбить область входных значений программы на классы, эквивалентные с точки зрения логики программы, то выполнение программы на символьных значениях, представляющих эти классы, нередко оказывается полным (и конечным) перебором всех возможных входных значений.

Еще одна особенность символьного выполнения — то, что мы получаем более или менее исчерпывающее описание функции, которую реализует программа.

Проблемы реализации. Применение подхода символьного исполнения для реальных языков программирования сталкивается с рядом проблем.

1. Число путей программы, как правило, очень велико. При этом их длина почти всегда ничем не ограничена. Как выбрать пути для анализа и построения тестов, чтобы эти пути достаточно хорошо покрывали программу? Разные системы решают эти вопросы по-разному (см. обзор в работе [5]). Мы позже остановимся на некоторых системах и посмотрим, как в них решаются эти проблемы.

2. Как практически определять выполнимость пути программы? Выше сказано, что возможны два варианта: доказательство теорем и решение систем уравнений и неравенств.

3. Как обрабатывать переменные с индексами? Для демонстрации проблемы рассмотрим предикат $a[i+1] = a[j]$. Этот предикат можно удовлетворить, если положить $i+1 = j$. Если же i и j входные переменные, то такой подход неправомерен. Рассмотрение же всех потенциально возможных случаев приводит к огромному числу вариантов.

4. Как обрабатывать обращения к фунциям и процедурам (подпрограммам)? Большинство систем такой возможности не имеют.

Несмотря на перечисленные проблемы, работа Ю. В. Борзова [5] заканчивается следующей отпимистической фразой: «Можно надеяться, что на рубеже 80-х годов появятся промышленные системы символического выполнения программ и автоматической генерации тестов. По-видимому, они могут составлять одну из самых интеллектуальных частей систем автоматизации программирования и отладки. Работы в этом направлении важны для повышения эффективности программирования и надежности программ».

К сожалению, эти надежды не оправдались. Да, мы многого достигли и многое научись делать, но одновременно развивались языки программирования и в них появлялись новые возможности (например, указатели), которые практически свели на нет все наши усилия.

Например, система TGS проекта СОКРАТ [9, 10], которая обеспечивала поддержку построения тестов с помощью символьного исполнения для языка Модула-2 содержит следующие ограничения реализации: «В случае, когда в ограничениях используются глобальные переменные и имеются вызовы процедур, побочным эффектом которых является изменение этих переменных, условия реализуемости могут вычисляться некорректно. К этому же может привести наличие вызовов процедур,

одним из параметров (не VAR!) которых является указатель, а значение, на которое он указывает, меняется внутри вызываемой процедуры и используется в ограничениях.

Неверные условия реализуемости могут также появиться и при использовании функции ADR. Например, для пути {a, b, c, d} в фрагменте программы

```
(*a*)  p := ADR(q);
(*b*)  p^ :=  1;
(*c*)  IF q<5 THEN (*d*)  ...  END;
```

будет приведено следующее условие реализуемости: $q < 5$ вместо правильного $1 < 5$».

7.2. Пример системы автоматизации тестирования — система TGS проект СОКРАТ

В данном разделе рассмотрим методы проектирования тестов (наборов входных данных) для отдельных процедур модуля и соответствующие инструменты, созданные в рамках системы СОКРАТ [16] . Полученный с применением этих методов и инструментов набор тестов может быть использован при тестировании программы, например с помощью имеющейся в системе СОКРАТ системы тестирования и отладки [6].

Сначала небольшой обзор систем построения тестов.

В работе [7] описывается система APOS и приводится алгоритм построения минимального дугового покрытия ациклического графа, который оказался более эффективным по сравнению с ранее использованными методами. Однако способ преобразования управляющего графа к ациклическому виду, который применен при этом, приводил к тому, что получаемое в итоге покрытие могло оказаться не минимальным.

Указанного недостатка лишена система «Автограф» инструментального комплекса программ «Ритм» [40] . Эта система строит минимальное покрытие управляющего графа PL-1 программ, а для построения тестов по этому покрытию авторы предлагают вручную находить условия реализуемости для каждого пути, фактически рекомендуя метод символического исполнения программы в прямом направлении.

В работе [8] описана система ТЕСТОР-ФОРТРАН и критикуются все системы, основанные на применении некоторого критерия тестирования. Критика в основном сводится к двум моментам. Во-первых, создаваемые таким образом тесты системы никак не связаны с функцио-

нальными и структурными свойствами модуля, в частности с распределением ошибок и значимости правильности определенных критических путей графа программы. Отсюда снижение эффективности тестирования и его полноты, излишние затраты ресурсов или невозможность достигнуть требуемой степени надежности программ в рамках имеющихся ресурсов. Во-вторых, большинство существующих систем является технологически неполными. Авторы предлагают метод целенаправленного тестирования — тестируемые пути выбираются пользователем на основе неформального понимания задачи с учетом ее функциональной и структурной специфики.

Недостаточный уровень автоматизации тестирования во многом объясняется малым числом удобных и простых систем, поддерживающих его и позволяющих автоматически оценивать полноту тестирования.

Определения. Под тестом обычно понимается набор значений входных данных модуля (подпрограммы), определяющих его исполнение, и соответствующий ему набор результатов, которые должны быть получены при исполнении модуля на этих входных данных или та нештатная ситуация, что должна произойти. В текущем разделе под тестом будет пониматься только набор значений входных данных модуля.

Каждому тесту можно сопоставить некоторый путь от начала к концу программы — последовательность вершин, которая соответствует последовательности исполнения операторов программы при выполнении этого теста.

Условия реализуемости пути (формула реализуемости) представляют собой систему ограничений, налагаемых на входные данные, при выполнении которых исполнение программы пойдет по заданному пути (см. пример в разд. 7.1.1). Имея путь, можно найти соответствующий ему тест, если подобрать такие входные данные, что выполняются условия реализуемости этого пути.

Условия реализуемости находятся с помощью символьного исполнения программы — ее формальное выполнение на символьных данных. Никаких реальных вычислений при этом не делается, а выполняется лишь текстовая замена переменных в выражениях на их текущее символьное значение, которое определяется последним присваиванием этой переменной некоторого выражения.

7.2.1. Комплексный критерии системы TGS

Комплексный критерий системы состоит из трех критериев: all-defs (покрытия всех определений переменных), C1 и критерия покрытия циклов.

Рассмотрим, почему именно такой критерий выбран. Во-первых, критерий C1 наиболее популярен и, по нашему мнению, является минимально необходимым для приемлемого тестирования программ.

Критерий структурного тестирования покрытия всех определений переменных (или критерий all-defs) основан на потоке данных в программе. Под определением переменной понимается присваивание переменной какого-нибудь значения; и именно это значение должно быть использовано хотя бы на одном из путей в покрытии. (Если определение переменной, т. е. присвоенное ей значение, нигде не используется, то это явная ошибка или лишний оператор.)

Критерий all-defs требует построения такого набора тестов, что для каждого определения переменной существует тест, при исполнении которого это определение достигает своего использования и на пути от определения до использования нет другого определения этой переменной.

Набор тестов, который не удовлетворяет критерию all-defs, не может считаться полным, так как любая ошибка в правой части присваивания, соответствующего определению переменной, которая нигде не использовалась, не может быть обнаружена таким набором тестов.

Для иллюстрации критериев рассмотрим следующую процедуру:

```
PROCEDURE seq(x, n: CARDINAL): CARDINAL;
       VAR y:  CARDINAL;
      BEGIN
(*a*)   IF   x=0  THEN
(*b*)          y:=2;
        ELSE
(*c*)          y:=1;
        END(*IF*);
(*d*)   WHILE  n<>0  DO
(*e*)          x:=x+y;
(*f*)          y:=y*x;
(*g*)          n:=n-1;
```

```
(*h*)   END(*WHILE*);
(*i*)          RETURN x;
(*k*) END seq;
```

При исполнении тестов ($x = 0, n = 0$) и ($x = 1, n = 1$) проходятся пути {a, b, d, i, k} и {a, c, d, e, g, h, d, i, k}, которые покрывают все дуги, и, следовательно, этот набор тестов удовлетворяет критерию C1. Однако определения переменной y в (*b*) и в (*f*) не достигают использований y, которые находятся в (*e*) и в (*f*). Для того чтобы удовлетворить критерию all-defs, можно добавить, например, путь {a, b, d, e, f, g, h, d, e, f, g, h, d, i, k} (ему соответствует тест ($x = 0, n = 2$)).

Для тестирования такой важной конструкции, как цикл, был добавлен критерий покрытия циклов.

7.2.2. Функционирование системы и ее компоненты

Задача поддержки структурного тестирования программ, по существу, состоит из двух частей. Во-первых, необходимо помочь пользователю в построении набора тестов, покрывающих логику программы, а во-вторых, оценить набор тестов на полноту для выбранного критерия тестирования и в каждый текущий момент показать степень тестированности по данному критерию [19,23]. Система обеспечивает поддержку тестирования программ, написанных на двух языках: Модула-2 и языке АССЕМБЛЕР ЕС ЭВМ.

Предполагается следующая технология использования системы. Строится внутреннее представление для каждого модуля (алгоритмы, приведенные в данном разделе, применимы к программам, написанным на Модула-2), т. е. создается управляющий граф для каждой процедуры этого модуля, в том числе для его тела. Строится множество путей, покрывающих все требуемые элементы комплексного критерия каждого из управляющих графов. Для каждого пути вычисляется формула его реализуемости. Путь, формула реализуемости и размеченный текст процедуры (указаны вершины, через которые проходит путь) являются исходным материалом для пользователя при создании теста для данного пути. Для этого пользователю надо или решить формулу реализуемости пути или подобрать исходные данные, используя размеченный текст процедуры и указанный путь.

Если для каждого пути из покрытия графа программы удается построить соответствующий ему тест, то полученный набор тестов будет

удовлетворять комплексному критерию, а для критерия C1 строится минимальное множество путей и, следовательно, минимальный набор тестов. Таким образом, решается первая часть задачи поддержки структурного тестирования.

Вторая часть системы, можно сказать, не зависит от первой, а исходит из предположения: есть набор тестов, неважно как полученный, и он оценивается на полноту по критерию C1; в каждый текущий момент тестирования система обеспечивает информацию о степени тестированности программы и при запросе показывает фрагменты, которые еще не исполнялись.

Для обеспечения перечисленных выше функций система содержит следующие компоненты:

— компонент построения управляющего графа (УГ) программы и сохранения его в архиве (конкретное представление управляющего графа для модула-программ будет приведено ниже, при описании реализации системы);

— компонент построения множества путей, покрывающих все требуемые элементы комплексного критерия, причем для критерия C1 строится минимальное множество путей;

— компонент построения формулы реализуемости для заданного пути, данный компонент реализует символьное исполнение программы по данному пути;

— компонент построения формулы-времени исполнения программы для заданного пути (формула оценивает время исполнения программы по этому пути, причем параметрами являются параметры циклов на данном пути);

— компонент пропуска пакета тестов с контролем комплесного критерия тестирования и оценкой степени тестированности.

7.2.3. Реализация системы

Компонент построения внутреннего представления в виде УГ. Данный компонент подключается к front-end транслятора с языка Модула-2 [14], который в системе СОКРАТ [16] зафиксирован в операционной форме в виде набора переменных процедурного типа — так называемых интерфейсных генерационных процедур. Данный интерфейс кроме кодогенератора используется и другими языковыми процессорами, в том числе и данным компонентом. Front-end транслятора производит синтаксический и семантический анализ текста программы,

сообщает о найденных ошибках и вызывает процедуры генерации, которые каждый языковой процессор определяет по-своему и таким образом вызывает нужные ему действия по обработке программы.

Компонент построения внутреннего представления определяет процедуры генерации так, что они строят внутреннее представление программы — управляющие графы для каждой процедуры обрабатываемого модуля и его тела, — а после обработки входного текста производятся все остальные действия уже с построенным внутренним представлением.

Для каждой процедуры модуля строится внутреннее представление в виде управляющего графа, вершины и дуги которого хранят некоторую информацию. При этом в каждой вершине имеются координаты соответствующего места в тексте программы, используемые при формировании размеченного текста программы (выделяются начала всех операторов).

Основная информация в вершине графа может быть трех типов:

а) присваивание — соответствует обычному присваиванию или получается при инициализации и приращении управляющей переменной цикла FOR;

б) ветвление — хранится выражение типа BOOLEAN, если ветвление соответствует условию операторов IF, WHILE, REPEAT, FOR (FOR моделируется оператором WHILE); либо выражение произвольного типа, если ветвление соответствует оператору CASE с этим выражением;

в) отсутствие информации — вершины такого вида соответствуют концам циклов WHILE, FOR, LOOP, началам циклов LOOP, REPEAT, вызовам процедур, выходным вершинам управляющих графов процедур.

Дополнительно к этой информации в вершине может храниться кратное присваивание (если в соответствующем месте программы есть вызов процедуры с VAR-параметрами).

Кратное присваивание [15] — это одновременное присваивание нескольким переменным сразу, т. е. конструкция типа

$$v_1, v_2, ..., v_n := expr_1, expr_2, ..., expr_n,$$

означающая, что сначала нужно вычислить адреса $v_1, v_2, ..., v_n$ (если есть индексирование или взятие значения указателей), затем найти значения $expr_1$, $expr_2$, ..., $expr_n$ и, наконец, произвести по порядку присваивания:

$$v_1 := value\ expr_1;$$

$$v_2 := value\ expr_2;$$

$$\ldots;$$

$$v_n := value\ expr_n.$$

Информация на дугах может быть следующих видов:

а) отсутствие информации;

б) пометка TRUE — для дуги, выходящей из условного оператора и соответствующей передачи управления в случае истинности условия;

в) пометка FALSE — аналогично пункту *б*;

г) пометка, соответствующая одной из меток оператора CASE.

Во внутреннем представлении отсутствуют вызовы процедур. Если есть вызов процедуры с VAR-параметрами, то он представляется как кратное присваивание соответствующим переменным. Если в программе был вызов процедуры без VAR-параметров, то ему соответствует пустая вершина, т. е. информация об этом вызове во внутреннем представлении отсутствует.

Для работы с массивами и записями используется взгляд на переменные такого типа, как на функции от селектора [15].

Селектор — произвольная последовательность операций индексации и выбора значения поля (допустимая для типа переменной v), при этом используется следующее обозначение:

(v | v<селектор> = <выр>).

Например, пусть a — такой массив, что a[1..4] = (1, 2, 3, 4), т. е. a[1] = 1, a[2] = 2, a[3] = 3, a[4] = 4. Тогда $(a|a[2] = 10)$ — это массив (1, 10, 3, 4) и $(a|a[2] = 10)[2] = 10$.

Описанная техника работы с переменными позволяет обрабатывать одинаковым образом переменные всех типов и упростить реализацию символьного исполнения программы.

Компонент построения множества путей, покрывающих все требуемые элементы комплексного критерия. Терминология этого пункта соответствует терминологии, используемой В. А. Евстигнеевым [44].

Итак, каждому критерию соответствует множество требуемых элементов, которые необходимо покрыть. Будем говорить, что путь покрывает требуемый элемент, если при исполнении программы по этому пути: для критерия C1 — проходится соответствующая дуга, для all-defs —

определение переменной достигает своего использования, для all-loops — цикл исполняется требуемое число раз.

Сначала строится минимальное дуговое покрытие управляющего графа. Для его построения используются следующие основные шаги:

— сведение УГ к графу с одним входом и одним выходом (если в этом есть необходимость);

— нахождение и отделение зон в графе;

— построение ациклического графа Герца;

— редукция полученного графа по методу DD-путей;

— покрытие зон и покрытие графа Герца;

— построение полного покрытия (преобразование путей графа Герца в пути исходного графа).

Вначале отделяются зоны графа (или компоненты сильной связности). Далее исходный граф преобразуется в ациклический граф Герца — граф, полученный сведением каждой его бикомпоненты в одну вершину (бикомпонента — это максимальная по включению компонента сильной связности). Затем для каждой зоны находится путь, начинающийся на входе в зону и покрывающий все ее дуги. Ациклический граф Герца покрывается по методу, предложенному К. А. Иыуду и М. М. Ариповым [7] и уточненному В. А. Шимаровым [17]. Добавляя в нужные места к полученным путям графа Герца покрытия зон, получаем полное покрытие исходного графа.

Основные алгоритмы построения минимального дугового покрытия. В приводимых ниже алгоритмах используется факт структурированности программ на Модула-2. Поэтому получаемый управляющий граф программы обладает свойством одновходовости зон, которое означает, что все зоны графа имеют единственный вход. Для языка АССЕМБЛЕР и любого другого языка, позволяющего создать неструктурированную программу, необходимо построенный УГ, имеющий зоны с более чем одним входом, преобразовать к графу с одновходовыми зонами и уже далее — к ациклическому графу Герца. В целях решения этой проблемы использовались алгоритмы и методы, аналогичные применяемым при оптимизации программ [18].

Пусть $G = (V, E)$ — ориентированный граф, каждая дуга $e = (u, v)$ которого может иметь пометку $M(e)$, где $M(e)$ — последовательность вершин из V, которая соответствует некоторому пути из u в v в графе G; $IN(v)$ — множество предшественников вершины v в графе G; $OUT(v)$ — множество потомков v; $d_{in}(v)$ — полустепень захода верши-

ны v, т. е. число вершин в $IN(v)$; $d_{out}(v)$ — полустепень исхода вершины v, т. е. число вершин в $OUT(v)$. В приводимых ниже алгоритмах «+» означает конкатенацию последовательностей.

Будем называть VE-путем последовательность вершин и дуг вида $p = [v_1, e_1 = (v_1, v_2), v_2, e_2 = (v_2, v_3), v_3, ..., v_n]$. Определим функцию, которая преобразует VE-путь в обычный путь:

$$convert\_path(p) = [v_1, M(e_1), v_2, M(e_2), v_3, ..., v_n].$$

Для отделения зоны от графа используется приведенный ниже алгоритм расщепления зоны.

$SPLITE\_ZONE(v, t)$ — алгоритм расщепления зоны.

Вход: v — вход в зону Z с обратной дугой (t, v).

Выход: а) модифицированный граф;

б) v' — вход отщепленной зоны.

Начало

1. Создать v' — копию v.
2. $IN_{new}(v) := \{u$ из $IN(v) | u$ не лежит внутри зоны $Z\}$.
   $IN(v') := \{u$ из $IN(v) | u$ лежит внутри зоны $Z\}$.
3. $OUT(v') := OUT(v)$.
   $OUT_{new}(v) := \{\}$.
4. Для всех u — вершин внутри зоны произвести перечисленные ниже действия.

Если есть дуга (u, w) такая, что w не лежит в Z, то:

а) создать дугу (v, w) с пометкой $M(v, w) = convert\_path(P[v, u]$, за исключением первой вершины $v) + M(u, w)$, где $P[v, u]$ — некоторый VE-путь от v к u;

б) удалить дугу (u, w).

Конец

Следующий алгоритм используется для уменьшения числа вершин в графе, что приводит к более эффективной работе алгоритма построения минимального покрытия.

$REDUCE(entry)$ — алгоритм разбиения вершин графа на DD-пути (DD-путь — это простой путь между двумя D-вершинами, а D-вершина — вершина, из которой выходит больше одной дуги).

Вход: граф с входной вершиной entry, дуги помечены.

Выход: модифицированный граф.

Начало

Для всех v-вершин графа:

Если $d_{out}(v) = 1$, то для единственной u из $OUT(v)$ и для всех w из $IN(v)$:

а) дугу (w, v) заменить на (w, u);

б) $M(w, u) := M(w, v) + [v] + M(v, u)$.

Конец

Для построения минимального по числу путей дугового покрытия ациклического графа применяется следующий алгоритм, впервые предложенный в работе [7]. Здесь этот алгоритм приводится в более точном и понятном виде.

$COVER\_ACYCLE(entry)$ — алгоритм покрытия ациклического графа.

Вход: ациклический граф со входом $entry$ (допускаются параллельные дуги и петли).

Выход: минимальное по числу путей дуговое покрытие графа — C — множество VE-путей.

Начало

$REDUCE(entry)$;

REPEAT

Начать формирование нового пути p;

$i := entry$; добавить i к p;

WHILE i имеет потомка j DO

добавить $(i, j), j$ к p;

IF $d_{out}(i) > 1$ и $d_{in}(j) > 1$ THEN

(i, j) — g-дуга; (* см. на рисунке граф 1 *)

(i, j) исключается из графа;

ELSIF $d_{out}(i) > 1$ и $d_{in}(j) = 1$ THEN

(i, j) — h-дуга; (* см. на рисунке граф 2 *)

(i, j) отмечается;

ELSIF $d_{out}(i) = 1$ и $d_{in}(j) > 1$ и на пути между последней отмеченной h-дугой и i нет g-дуги THEN

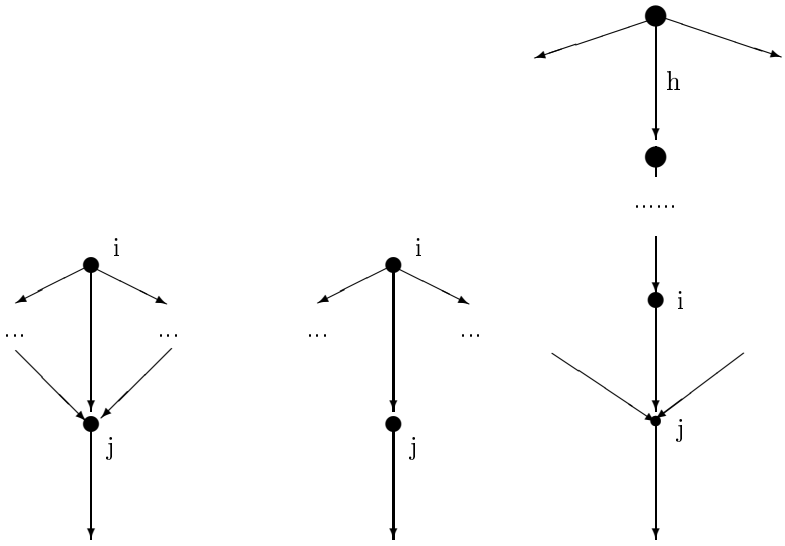
(* см. на рисунке граф 3 *) исключается последняя отмеченная h-дуга

END;

$i := j$;

END(*WHILE*);

добавить p к C;



UNTIL в пути не было g- и h-дуг (* т. е. для всех вершин v в последнем пройденном пути выполнялось $d_{out}(v) \leq 1$ и $d_{in}(v) \leq 1$ *);
Конец

Далее приводится алгоритм построения составного покрытия. Составное покрытие — это последовательность $[c_1, c_2, ..., c_n]$ покрытий, в которой c_1 — покрытие графа Герца для исходного графа, а $c_2, ..., c_n$ — покрытия зон графа.

В алгоритме используется функция *convert*. Пусть $c = \{p_1, p_2, ..., p_n\}$ — покрытие. Тогда

$$convert(c) = \{convert\_path(p_1), ..., convert\_path(p_n)\}.$$

$COMPLEX(entry, BE)$ — алгоритм построения составного покрытия.

Вход: ориентированный граф. Для каждой вершины v заданы $IN(v)$ и $OUT(v)$. BE — список обратных дуг. Все дуги имеют пустую пометку.

Выход: составное покрытие S.

Начало

1. $S :=$ пустая последовательность;
2. Для всех $e = (u, v)$ из BE:

а) $BE := BE \setminus e$;

б) BE' := такие дуги из BE, у которых начальная и конечная вершины принадлежат зоне, соответствующей обратной дуге (u, v);

в) $BE := BE \setminus BE'$;

г) $v' := SPLITE\_ZONE(v, u)$ (* используется одновходовость графа *);

д) исключить дугу e из графа;

е) $S := COMPLEX(v', BE') + S$.

3. $S := convert(COVER\_ACYCLE(entry)) + S$

Конец

Для построения покрытия исходного графа из составного покрытия каждое $c = \{p_1, ..., p_n\}$ — покрытие зоны Z с входом v — заменяется на путь $p_1 + p_2 + ... + p_n$, а затем этот путь вставляется вместо вершины v в какой-нибудь путь графа Герца исходного графа (или в один из путей той зоны, в которой лежит Z).

Рассмотрим, как дополнить этот набор путей так, чтобы обеспечить выполнение критериев all-defs и all-loops.

При создании алгоритма, который, имея множество требуемых элементов, возвращал бы множество путей, покрывающих эти требуемые элементы, необходимы две операции (которые определяются для каждого критерия по-разному):

1) есть путь и требуемый элемент; нужно определить, покрывается ли требуемый элемент данным путем;

2) есть требуемый элемент; найти покрывающий его путь.

Реализация этих операций для критериев all-defs и all-loops не представляет особого труда и точно соответствует определениям критериев. Теперь приведем собственно алгоритм, который по множеству требуемых элементов и уже существующему покрытию находит пути, покрывающие еще непокрытые требуемые элементы.

Алгоритм COV_REQ(*Req*, *Exist*)

Вход:

Req — множество требуемых элементов;

Exist — множество путей — имеющееся покрытие (в качестве начального значения используется минимальное покрытие для критерия C1).

Выход: C — множество путей, которое дополняет *Exist* так, что все *Req* покрыты.

Начало
    Для каждого r из Req
    если не существует в $Exist$ пути, покрывающего r, то
        Начало
            $p :=$ путь, покрывающий r;
            $C := C + \{p\}$;
            $Exist := Exist + \{p\}$;
        Конец;
Конец.

Этот алгоритм применяется два раза: сначала к минимальному дуговому покрытию добавляются пути для выполнения критерия all-defs, а затем к получившемуся множеству путей добавляются пути, удовлетворяющие критерию all-loops.

В действительности конечно же написаны две версии этого алгоритма: для критерия all-defs и для критерия all-loops.

Таким образом, как можно видеть из описания алгоритма, мы отказались от попытки добавить минимально возможное число путей. Это объясняется тем, что задача построения минимального покрытия по критерию all-defs является, как можно доказать, NP-трудной, а для критерия all-loops хотя и можно придумать эффективный алгоритм, но довольно громоздкий. При этом число требуемых элементов в Req для «реальных» программ обычно невелико и количество путей, полученных с помощью описанного алгоритма, практически всегда совпадает с минимально возможным.

Компонент построения условий реализуемости (символьное исполнение программы) описан ранее (см. подразд. 7.1.1).

Пример построения тестов с помощью системы TGS (применение системы). Пусть для всех процедур модуля demo.mod нужно построить набор тестов, удовлетворяющий критерию C1. После исполнения компонентов построения минимального покрытия и символьного исполнения для полученных путей получаем размеченный текст модуля и множество путей с условиями их реализуемости:

Размеченный текст модуля

```
MODULE demo; (* 11-May-94 *)

PROCEDURE power(x, n: INTEGER): INTEGER;
VAR res: INTEGER;
BEGIN
```

```
(*a*)  IF   (x=0) AND (n<=0) THEN
        (* Ошибка: неправильные входные данные*)
(*b*)  HALT;
(*c*)  ELSIF  n<0  THEN
         (*d*)IF  x=1 THEN (*e*) RETURN 1
              ELSE (*f*) RETURN 0
              END;
     END;
     (*g*) res:=1;
(*h*)WHILE  n # 0 DO
    (*i*)IF  ODD(n) THEN
            (*j*) n:=n-1;
            (*k*) res:=res*x;
         ELSE
            (*l*) n:=n DIV 2;
            (*m*) x:=x*x;
         END;
(*n*)END(*WHILE*);
    (*o*) RETURN res;
(*p*)END power;

PROCEDURE strange(VAR a: INTEGER);
BEGIN
(*a*)   IF a>10 THEN
(*b*)        INC(a);
END;
(*c*)   IF a<=11 THEN
(*d*)         a:=-a;
END;
(*e*) END strange;
(*a*) END demo.
```

Множество путей

Модуль: demo
Путей в покрытии: 7
Вершин в путях: 38
Условий в путях: 18

power

1: acg{h}op

2: acdfp
3: acdep
4: abp
———————— Цикл {h}
1: hilmnh
2: hijknh
———————— Полное покрытие
1: acghilmnhijknhop
2: acdfp
3: acdep
4: abp
———————— Условия реализуемости
1: NOT ((x=0) AND (n<=0)), NOT (n<0), n#0, NOT ODD(n),
n DIV 2 # 0, ODD(n DIV 2), NOT(n DIV 2 - 1 # 0)
2: NOT((x=0) AND (n<=0)), n<0, NOT(x=1)
3: NOT((x=0) AND (n<=0)), n<0, x=1
4: (x=0) AND (n<=0)
———————————————— power
———————————————— strange
1: abcde
2: ace
———————— Полное покрытие
1: abcde
2: ace
———————— Условия реализуемости
1: a>10, a+1<=11
2: NOT(a>10), NOT(a<=11)
———————————————— strange
———————————————— demo
1: a
———————— Полное покрытие
1: a
———————— Условия реализуемости
1: TRUE
———————————————— demo

Построим вначале тесты для процедуры *power*. При анализе ограничений обычно лучше начинать с более сильных (т. е. более строгих),

например с равенств. Условия реализуемости для первого пути содержат одно равенство — NOT(n DIV 2 − 1 # 0), которое равносильно n DIV 2 = 1. Замечаем, что из этого равенства следует выполнение некоторых других ограничений из условий реализуемости первого пути, поэтому их можно исключить из рассмотрения. Просматривая затем следующее по силе ограничение NOT ODD(n), получаем, что n — четное, и, следовательно, равно 2; *x* можно взять любым, например -5. Еще раз проверяем, что такие входные данные удовлетворяют условиям реализуемости, и получаем тест, соответствующий первому пути.

Остальные тесты найти совсем просто. В результате получаем следующий набор тестов:

1) n = 2, x = -5; 2) n = -1, x = 2; 3) n = -5, x = 1; 4) x = 0, n = 0.

Приступая к построению тестов для процедуры *strange*, обнаруживаем, что как первый, так и второй пути являются нереализуемыми (их условия реализуемости тождественно ложны). Поэтому, просмотрев текст процедуры, изменяем покрытие, переставляя подпути *ce* и *cde* в исходных путях. Получаем следующее покрытие: 1) abce; 2) acde. Затем находим соответствующие тесты: 1) a = 11; 2) a = 10.

Компонент оценки набора тестов на полноту системы TGS. Задача данного инструмента состоит в обеспечении контроля полноты исполняемого набора тестов для критерия C1. Для заданного набора тестов (неважно как полученного) компонент показывает степень тестированности (отношение числа пройденных ветвей при исполнении программы к общему числу ветвей в управляющем графе программы) и непокрытые тестовые элементы или утверждение о полноте набора тестов.

Решение данной задачи обычно обеспечивается инструментированием программы (см. подразд. 7.3.1) для контроля путей исполнения, и в конце исполнения подводятся итоги: вычисляется метрика тестированности и определяются непокрытые элементы. При этом инструментироваться может как исходный, так и исполняемый текст программы. В нашем конкретном случае, так как СОКРАТ — кросс-система программирования, исполнение программ на персональном компьютере поддерживается специальной системой тестирования и отладки, возможен иной путь. Данная система создает среду для исполнения и обеспечивает возможность с помощью «точек останова» указать в программе те места, в которых хотим получить требуемую информацию, и задать процедуры-методы обработки данных «точек останова».

Таким образом, с точки зрения пользователя инструментов для обеспечения контроля на полноту критериев необходимо в пакет команд тестирования включить команды:

— *начать контроль* — активирует средства контроля пакета тестов на полноту;

— *завершить контроль* — деактивирует средства контроля и подводит итоги даного сеанса контроля и всех предшествующих, если они были, и записывает собранную информацию в архив системы.

По первой команде загружается управляющий граф тестируемой (контролируемой) программы и на основе содержащейся в нем информации об определяющих конструкциях программы для данных критериев расставляются «контрольные точки» в соответствующих местах исполняемой программы. Так, для критерия C1 управляющий граф содержит информацию о началах всех линейных участков в программе и при исполнении программы на предъявленных тестах будет подсчитываться число исполнений каждого линейного участка, а также число переходов в каждой развилке по TRUE и FALSE (для CASE — по каждой из альтернатив).

Эта собранная информация обрабатывается по команде завершения контроля, и вычисляется метрика тестированности по критерию, которая и говорит о полноте исполненного набора тестов. При необходимости возможен показ тех элементов программы, которые на настоящий момент еще не покрыты.

Итак, сам алгоритм контроля на полноту по критериям довольно прост и состоит в расстановке «контрольных точек» по специально подготовленному для каждого из критериев управляющему графу с информацией, которая как раз и должна обеспечить возможность расстановки «контрольных точек», а также в написании процедур-методов обработки останова в контрольной точке. Данные процедуры обычно увеличивают значение некоторого счетчика, показывающего число исполнений конструкции, соответствующей данной контрольной точке. По завершении контроля необходимо обработать собранную таким образом информацию и дать заключение о «полноте» набора тестов.

Заключение. В качестве достоинств созданных компонентов можно отметить следующие. Во-первых, они не заставляют пользователя придерживаться некоторой, жестко фиксированной методологии тестирования, а позволяют ее изменять в зависимости от класса создаваемых программ, квалификации пользователя (в области тестирования),

ресурсов, отводимых на тестирование, и т. д. Например, пользователь, не знающий методов тестирования (или не имеющий достаточно времени на их применение), может воспользоваться инструментом поддержки структурного тестирования и получить набор тестов, удовлетворяющий комплексному критерию структурного тестирования. Другой пользователь может строить тесты вручную, по методам «черного ящика» и затем использовать инструмент оценки построенного набора тестов на полноту для проверки выполнения критериев «белого ящика» (этот более трудоемкий метод позволяет качественнее протестировать программу). Также возможно использование обоих этих подходов: первый применяется программистом, создающим программу, второй — группой контроля качества.

Во-вторых, архитектура данных компонентов позволяет подключение других языков программирования. Например, в настоящий момент поддерживается применение некоторых инструментов для Модула-2 и ассемблера ЕС ЭВМ.

7.3. Автоматизация оценки полноты набора тестов

Обычно программист сначала отлаживает программу на некоторых «случайных» тестах и затем, после исправления всех найденных ошибок, переходит к целенаправленному тестированию программы. Поэтому проблему получения хорошего набора тестов в общем случае можно сформулировать следующим образом:

Дана программа P и набор тестов T. Достаточно ли этого набора тестов T для «полного» тестирования программы P, если нет, то какие тесты нужно добавить?

Полное тестирование программы означает, что набор тестов может обнаружить любую ошибку в программе, и если результаты исполнения программы на этом наборе тестов совпадают с ожидаемыми, то правильность программы доказана «от противного». Однако, как мы убедились, задача построения полного набор тестов является неразрешимой, поэтому возможно лишь частичное решение этой проблемы, т. е. для данного набора тестов иногда можно сказать, какие тесты необходимо добавить еще.

Для этого были разработаны методы (критерии) тестирования, которые требуют построения набора тестов с определенными свойствами. Эти критерии основываются как на анализе типичных ошибок в

программах, так и на предположении, что более разнообразный набор тестов имеет бóльшую вероятность обнаружить ошибку.

Возможны два варианта инструментальной поддержки для пользователя, желающего получить набор тестов, удовлетворяющий критериям тестирования. Во-первых, можно помогать сразу строить необходимый набор тестов (т. е. решать задачу для пустого набора тестов) — инструменты такого типа называются *инструментами проектирования тестов* (*test design tools* в работе [20]) — и, во-вторых, проверять, выполнены ли критерии для имеющегося уже набора тестов, и если нет, то помогать пользователю построить дополнительные тесты (т. е. решать задачу в общем виде) — инструменты такого вида называются *инструментами контроля тестированности (test evaluation tools)*, или *инструментами покрытия*[1] *(coverage tools)*.

Инструменты контроля тестированности имеют два основных назначения. Во-первых, они должны оценить набор тестов, его качество, выявить плохо протестированные части программы (например, наименее протестированные процедуры) и сообщить эту информацию пользователю; во-вторых, показать, в чем же конкретно заключается слабость набора тестов, и помочь дополнить его.

Эти две цели в действительности несколько противоречивы, так как во втором случае выбор критериев тестирования ограничен критериями, для которых тесты можно построить относительно просто. В первом случае это ограничение отсутствует.

Таким образом, инструменты контроля тестированности можно разбить на два класса в зависимости от цели, которой они больше достигают: *инструменты оценки набора тестов* и *инструменты улучшения набора тестов*. Заметим, что эта классификация весьма условна, так как многие инструменты можно отнести к обеим группам и граница между этими группами весьма расплывчата.

Инструменты оценки набора тестов могут использоваться следующим образом: пользователь создает каким-то образом набор тестов, затем оценивает его и, если уровень тестированности программы низок, дополняет этот набор тестов новыми тестами и т. д. При этом увеличе-

[1] Термин «покрытие» рассматривается в следующем смысле: определяется множество требуемых элементов (для каждого критерия оно свое). При исполнении набора тестов элемент множества требуемых элементов помечается (говорим: покрывается), если он либо достигается, либо выполняется, и т. д. и т. п. [21]. Иначе говоря, порождаемое при исполнении программы на наборе тестов множество требуемых элементов покрывает задаваемое критерием.

ние уровня тестированности — показатель его изобретательности, стимулирующий построение хороших тестов. Этот вариант тестирования хорош тогда, когда пользователь не применяет какую-то фиксированную технологию тестирования, а выбирает тесты произвольным образом, надеясь на свою изобретательность.

Инструменты оценки набора тестов можно использовать для выделения небольшого эффективного подмножества из большого набора тестов, полученного случайным образом или оставшегося от предыдущих версий программы и сильно разросшегося. Так как проверку результатов исполнения тестов часто приходится делать вручную, то при использовании меньшего набора тестов затраты на тестирование могут заметно уменьшиться. Существуют методы для выделения подмножества тестов, которое покрывает те же требуемые элементы для различных критериев, что и исходный набор тестов (см., например, [22]).

Таким образом, выбор критериев для инструментов оценки набора тестов обусловливается их способностью к обнаружению ошибок, а также эффективностью реализации. При этом чаще всего уровень тестированности определяется как отношение числа непокрытых требуемых элементов критерия к числу всех требуемых элементов критерия, хотя в некоторых работах предлагаются более совершенные подходы для вычисления уровня тестированности [21, 23].

К инструментам такого типа следует отнести системы, использующие мутационные критерии[1] [25] (из наиболее эффективных реализаций следует отметить работу [26]). Также сюда относятся инструменты, которые оценивают вероятность возможной ошибки, оставшейся необнаруженной данным набором тестов. Это, например, динамическое мутационное тестирование [37], где значения некоторых переменных изменяются случайным образом (моделируется ошибка в операторе присваивания) и оценивается способность набора тестов обнаружить это изменение (т. е. его чувствительность к ошибкам).

Инструменты улучшения набора тестов можно также использовать для оценки набора тестов, но выбор критериев в этих инструментах ограничен только такими критериями, для которых тесты можно по-

[1] В основе всех мутационных критериев лежит идея выделения некоторых программных компонентов и задания для каждого компонента набора преобразований — мутаций. Заданные преобразования позволяют получить из рассматриваемой программы множество программ, называемых мутантами. Набор тестов для тестирования рассматриваемой программы оценивается его способностью выявлять мутантов среди порожденного множества программ (см. раздел «Мутационный подход»).

строить сравнительно легко. При этом пользователь может как применять формальную технологию тестирования, например технологию тестирования подсистем, описанную в работе [19], так и просто постепенно строить набор тестов, используя подобный инструмент. В первом случае этот инструмент будет демонстрировать ошибки и недостатки, допущенные при проектировании и реализации тестов [Там же], во втором — набор тестов полностью строится с его помощью.

Из всех критериев тестирования наиболее популярным является C1, и в настоящее время существует большое число коммерческих инструментов контроля тестированности [20], проверяющих его выполнение (в основном для программ, написанных на языке C/C++). Из отечественных разработок отметим инструментальный комплекс РИТМ [40], в котором также реализована проверка выполнения критерия C1. Кроме того, есть небольшое число систем, например C-COVER и PiSCES [20], которые дополнительно проверяют выполнение критериев покрытия условий/решений и комбинаторного покрытия условий [2].

Среди инструментов из списка [20] больше всего критериев проверяется системой GCT [41], имеющей следующие критерии: C1, покрытие условий, покрытие циклов, слабомутационные критерии [30] (в том числе покрытие отношений), а также критерии для мультипроцессных программ. Анализируя использование этой системы, Б. Марик утверждает [42], что из слабомутационных критериев в инструментах такого типа (т. е. когда пользователю приходится вручную строить тесты для выполнения критериев или проверять, что такой тест невозможно построить) эффективным является лишь критерий покрытия отношений; для остальных слабомутационных критериев, наподобие мутации арифметических операторов и имени переменной, время, затрачиваемое на построение тестов, не окупается числом найденных ошибок.

Б. Марик предлагает следующий метод тестирования: тесты строятся исходя из спецификаций, неформального описания программы, предположений о возможных ошибках с помощью описанных в работе [48] методов. Затем они исполняются и с помощью системы GCT находятся непокрытые требуемые элементы. Эти непокрытые требуемые элементы показывают слабость набора тестов, а также то, в чем заключается слабость. Далее рекомендуется перестать концентрироваться на покрытии, а попытаться улучшить набор тестов. Таким образом, достижение 100 %-го покрытия является не самоцелью, а лишь средством получения хорошего набора тестов.

Для демонстрации идеи Б. Марика приведем отрывок из работы [48].

«...Допустим, мы тестируем следующий фрагмент программы:

```
1: operation_failed = operation();
2: if (operation_failed)
3:    switch (type_of_failure)
4:    {
5:      case FAILURE_TYPE_1: ...
6:      case FAILURE_TYPE_2: ...
7:      case FAILURE_TYPE_3: ...
8:    }
```

Инструмент контроля полноты покрытия может сообщить о том, что случаи для ошибок типа 1 и 3 не рассматривались с помощью сообщений типа:

```
line 5: case was taken 0 times.
line 7: case was taken 0 times.
```

Сильным искушением было бы написать тесты, которые вызывают ошибки `operation()` типа 1 и 3 и остановиться на этом. Однако в действительности покрытие сообщает только лишь о том, что слабо проверялись различные типы ошибок `operation()`. Более внимательный анализ мог бы выявить, что в действительности `operation()` могла бы иметь и 4-й тип ошибки, который не обрабатывается тестируемой программой — возможно, из-за ошибки пропуска случая в ней[1].

Внимательный анализ тяжелее, чем поиск простых способов выполнения критериев. Однако после тщательного предварительного проектирования тестов это занимает лишь небольшую часть общих усилий».

На наш взгляд, это прекрасная идея и она заслуживает внимания, однако понятно, что при таком подходе от пользователя требуется хорошее знание методов «черного ящика».

Основным способом реализации подобных систем является инструментация исходного кода программы. Под инструментацией понимается вставка определенного программного кода в текст исследуемой программы таким образом, чтобы он обеспечивал контроль покрытия всех тестовых элементов комплексного критерия. Более подробно данную процедуру рассмотрим на примере системы OCT.

[1] Ошибка пропуска случая — наиболее серьезная ошибка, которая не может быть найдена с помощью критериев структурного тестирования. Она заключается в том, что в программе (или даже в спецификации) пропущен разбор некоторого случая. Ошибки такого типа часто обнаруживаются только при эксплуатации программы и поэтому обычно стоят очень дорого. — Примеч. авт.

7.3.1. Пример системы контроля тестированности программ — система ОСТ

Система контроля тестированности программ, написанных на языке Модула-2/Оберон-2 — ОСТ (**O**beron-2/Modula-2 **C**overage **T**ool), реализована в рамках системы XDS [43]. Эта система предназначена главным образом для улучшения набора тестов, и ее достоинством является использование **комплексного критерия тестирования**, состоящего из большого числа полезных критериев тестирования, которые позволяют получить хороший набор тестов.

Комплексный критерий в описываемой системе включает следующие критерии: C1, покрытия условий, покрытия циклов, покрытия отношений, покрытия рекурсивных процедур (аналог критерия покрытия циклов), критерий интеграционного тестирования — покрытие вызовов процедур (так, как он описан в работе [19]), а также критерий покрытия входных параметров и результатов процедур.

В последних двух критериях тестовые условия для вызовов процедур и типов входных параметров и результатов описываются с помощью специально разработанного языка. На этом языке были определены тестовые условия для библиотечных процедур Модула-2 и Оберон-2, имеющиеся в системе XDS, что позволяет проверять выполнение интеграционных критериев тестирования в программах пользователя, в которых присутствуют вызовы этих библиотечных процедур.

В комплексном критерии не представлены критерии потока данных, что объясняется двумя причинами: во-первых, для них практически невозможно показать непокрытые элементы в компактном виде так, как это принято в данной системе, а во-вторых, разработка систем, основанных на критериях потока данных, достаточно сложна, что, возможно, объясняет отсутствие коммерческих систем подобного типа. Как показал наш опыт реализации критерия all-defs в системе TGS (комплексный критерий состоял из трех критериев: C1, покрытия циклов и all-defs), число вновь создаваемых тестов для его выполнения (если сначала строились тесты для выполнения критерия C1, потом для критерия покрытия циклов и в конце для all-defs) довольно часто равнялось нулю.

7.3.2. Реализация системы

Система реализована методом *инструментации* исходной программы, т. е. в исходный текст программы вставляется некоторый код, со-

бирающий информацию о выполнении программы и сохраняющий ее после завершения программы. Затем эта информация (полученные после исполнения набора тестов) обрабатываются генератором отчетов, который сообщает пользователю степень тестированности инструментированных модулей и при необходимости показывает непокрытые тестовые условия и, таким образом, помогает построить дополнительные тесты. Далее будет дано подробное описание механизма работы системы, инструментации исходной программы и языка описания тестовых условий.

Механизм работы системы

В состав системы входят три программы: *инструментатор*, *генератор отчетов* и *утилита проверки* правильности файлов с описанием тестовых условий.

Система функционирует следующим образом.

1. **Инструментация.** Модуль или несколько модулей программы инструментируются с помощью программы *инструментатор*, в результате чего:

• исходный модуль, например `x.mod`, копируется в `x.sav` (с тем чтобы потом можно было просто восстановить модуль без инструментации);

• в `x.mod` вставляется необходимый код для проверки тестовых условий;

• создается файл `x.log`, состоящий из нулей и предназначенный для хранения информации о покрытии;

• создается файл `x.inf`, содержащий информацию о связи тестовых условий с видом соответствующего критерия и их расположением в программе.

При инструментации может использоваться специальный файл, содержащий тестовые условия для типов и вызовов процедур программы, возможно также содержащий ссылку на файл, с описанием тестовых условий для библиотечных процедур системы XDS.

2. **Создание исполняемого кода инструментируемой программы.** Используются стандартные средства компиляции и сборки системы XDS [43], в результате чего получается исполняемый файл `x.exe` инструментированной программы. При этом для разрешения внешних связей, появившихся вследствие вставленных в текст при инструментации внешних процедур, к имеющимся модулям программы добавляется

модуль OctLog, поставляемый с системой и содержащий необходимые описания типов и процедур.

3. **Пропуск набора тестов.** Полученный инструментированный файл x.exe исполняется на наборе тестов, полнота которого оценивается. Инструментированный файл x.exe работает точно так же, как и неинструментированный, но в начале работы он читает *.log-файлы инструментированных модулей, затем при исполнении программы на наборе тестов вставленные инструментатором процедуры проверки увеличивают число выполнения покрываемых тестовых условий. После окончания работы программы информация о покрытии сохраняется в соответствующих *.log-файлах.

4. **Генерация отчета.** При генерации отчета используются файлы *.inf, *.log и создается либо краткий отчет о степени тестированности всех инструментированных модулей, либо подробный отчет для одного модуля — файл *.rep, содержащий исходный текст модуля со вставленными сообщениями о непокрытых требуемых элементах и количестве выполнения покрытых.

Реализация иструментации исходной программы

Для проверки выполнения критериев в тестируемый модуль вставляются вызовы процедур из модуля OctLog. Последний обеспечивает сохранение данных о покрытых тестовых условиях для каждого модуля и предоставляет процедуры, необходимые для регистрации выполнения тестовых условий. Фрагменты модулей OctLog.def и OctLog.mod можно найти в приложении.

После завершения программы вызывается финализатор модуля[1] OctLog, который сохраняет результаты проверки выполнения тестовых условий в соответствующих файлах.

Итак, при инструментации в любой модуль вставляется объявление переменной OCT_log, которая будет указывать на данные о покрытии тестовых условий для модуля, и код для ее инициализации с помощью процедуры new_log, первый параметр которой — общее число тестовых условий в модуле, а второй — имя модуля:

```
MODULE test;           MODULE test;
  . . .                IMPORT OctLog;
BEGIN          ===>    VAR OCT_log: OctLog.LOG;
  . . .                  . . .
```

[1] Часть модуля, вызываемая перед завершением исполнения программы.

```
END test.              BEGIN
                         OCT_log:= OctLog.new_log(112, 'test');
                         . . .
                       END test.
```

Затем делаются вставки для проверки выполнения каждого из критериев. Далее приведены примеры вставок для всех критериев.

Критерий C1

- Инструментируются начала следующих операторов:
  - входы процедур;
  - альтернативы `CASE`;
  - альтернативы оператора `WITH` Оберона-2;
  - ветвь `ELSE` операторов `CASE` и `WITH`, если она есть;
  - начало тела цикла `FOR`;
  - первый оператор после цикла `FOR`;
  - обработка исключений `EXCEPT`.

Перед первым оператором последовательности операторов вставляется вызов процедуры `reg`, которая регистрирует выполнение тестового условия с указанным номером:

```
PROCEDURE x;           PROCEDURE x;
BEGIN                  BEGIN  OctLog.reg(OCT_log, 1);
  . . .        ===>      . . .
END x;                 END x;
```

- Логические решения:
  - условный оператор `IF`;
  - условие из `WHILE`;
  - условие из `UNTIL`.

Соответствующее условие «окружается» процедурой `reg_bool`, которая проверяет выполнение двух тестовых условий (TRUE и FALSE) и возвращает анализируемое условие неизменным (имеет три параметра: первый — переменная `OCT_log`; второй — номер (например k) тестового условия, соответствующего истинному значению условия, следующий номер (в нашем случае k + 1) будет использоваться для ложного значения условия; третий — само условие):

```
IF cond THEN     IF OctLog.reg_bool(OCT_log, 2, cond) THEN
```

```
    . . .     ===>          . . .
  END;              END;
```

Здесь тестовое условие с номером 2 будет соответствовать `cond = TRUE`, а с номером 3 — `cond = FALSE`.

Критерий покрытия условий. Каждое элементарное подусловие (не содержащее `AND` или `OR`) окружается процедурой `reg_bool`:

```
a AND b   ===>   OctLog.reg_bool(OCT_log, 4, a) AND
                 OctLog.reg_bool(OCT_log, 6, b)
```

Критерий покрытия отношений. При проверке выполнения этого критерия знак операции убирается, а само сравнение (а также проверка выполнения тестового условия) производится процедурами `reg_int_rel` или `reg_card_rel` в зависимости от типа операндов:

```
a+b < b/2   ===>    OctLog.reg_int_rel(OCT_log, 8,
                    OctLog.lt, a+b, b/2)
```

Критерий покрытия циклов. Вставляется объявление уникальной переменной для каждого цикла. Она является локальной для процедуры, в которой находится цикл, и предназначена для отслеживания текущего номера итерации цикла. Непосредственно перед циклом (в точку $init(l)$) вставляется вызов процедуры `reg_init_loop`, а перед первым оператором (в точку $body(l)$) — вызов процедуры `reg_entry_loop`:

```
WHILE c DO ===> VAR OCT_var1: INTEGER; (* локальная *)
  . . .              . . . (* \ своя для каждого цикла *)
                 OctLog.reg_init_loop(OCT_log, 9, OCT_var1);
                 WHILE c DO
                  OctLog.reg_entry_loop(OCT_log, 9, OCT_var1);
                    . . .
```

Критерий покрытия рекурсивных процедур. Вставляется объявление двух уникальных глобальных переменных для каждой рекурсивной процедуры, а также их инициализация. Затем на входе в процедуру вставляется вызов `reg_proc_entry`, а на выходе из нее `reg_proc_exit`. Если есть рекурсивные процедуры-функции, то возникает дополнительная трудность из-за того, что сначала нужно вычис-

лить возвращаемое выражение, а затем вызвать процедуру `reg_proc_exit`. Что делается в этом случае, нетрудно понять из примера:

```
                          VAR OCT_global1, OCT_global2: INTEGER;
                           . . .  (* \ свои для каждой процедуры *)
PROCEDURE f3(): REAL;     PROCEDURE f3(): REAL;
BEGIN                       PROCEDURE OCT_result(ret: REAL): REAL;
  RETURN f3();       ===>   BEGIN
END f3;                       OctLog.reg_proc_exit(OCT_log, 12,
                                       OCT_global1, OCT_global2);
                              RETURN ret;
                            END OCT_result;
                          BEGIN  OctLog.reg_proc_entry(OCT_log, 12,
                                      OCT_global1, OCT_global2);
                             RETURN OCT_result( f3());
                          END f3;
                              . . .
                          OCT_global1:=0; (* в инициализации модуля *)
```

Критерий покрытия входных параметров и результатов. Если p — входной параметр типа T некоторой процедуры и для T заданы тестовые условия, то для проверки этих тестовых условий вставляется код вида

```
IF C1(p) THEN reg(OCT_log, 13);
 ELSIF C2(p) THEN reg(OCT_log, 14);
 ELSIF
 ...
END;
```

Если же процедура возвращает значение типа T, то вставляется локальная процедура `OCT_result2(RESULT: T): T`, тело которой содержит проверку выполнения тестовых условий для `RESULT`, а в теле исходной процедуры все операторы вида `RETURN expr` заменяются на `RETURN OCT_result2(expr)`.

Критерий покрытия интеграционных вызовов. Если в программе есть вызов процедуры `pr(x1,...,xN)`, для которой заданы тестовые условия, то этот вызов заменяется на `OCT_CALL_pr(x1, ... ,xN, <номер тестового условия>)` и в текущий модуль вставляется следующая процедура:

```
PROCEDURE OCT_CALL_pr(x1:T1;...;xN:TN; tc_num: INTEGER): T;
  VAR RESULT: T;
BEGIN
  <тестовые условия 'до вызова'>
  RESULT:= pr(x1,...,xN);
  <тестовые условия 'после вызова'>
  RETURN RESULT;
END OCT_CALL_pr;
```

Если *pr* не является процедурой-функцией, то в приведенной выше процедуре нет описания переменной `RESULT` и нет для нее оператора присваивания.

Правильность приведенной инструментации легко доказывается [11].

7.3.3. Язык описания тестовых условий

Общий вид файла с описанием тестовых условия

Для критериев покрытия входных параметров и результатов процедур и покрытия вызовов процедур инструментатор использует описание тестовых условия. Он берет это описание из файла testcond.oct в текущей директории. Файл имеет следующий вид:

```
<*.oct-файл>::= { <Условия модуля> | <Команда включения> }
```

Можно использовать команду включения для того, чтобы включить в файл testcond.oct другие файлы с описанием тестовых условий. Команда включения имеет вид

```
<Команда включения>  ::= 'INCLUDE' '‘' <Имя файла> '’'  ';'
<Имя файла>          ::= <все символы от '‘' до '’'>
```

Например, чтобы включить описание тестовых условий, содержащихся в файле xdslib.oct, достаточно написать: `INCLUDE ‘xdslib.oct‘`.

Файл xdslib.oct содержит описание тестовых условия для многих библиотечных модулей системы XDS.

Условия модуля предназначены для описания тестовых условий типов и вызовов процедур указанного модуля. Условия модуля имеют вид

```
<Условия модуля> ::= 'MODULE' <Имя модуля> ';'
                       { <Условия типа> | <Условия вызова> }
                     'END' <Имя модуля> '.'
```

В любом месте при описании тестовых условий могут встречаться комментарии двух видов, определяемые так же, как в языке Модула-2.

Описание тестовых условий типа

Условия типа предназначены для описания тестовых условий типа. Их использует критерий покрытия входных параметров и результатов процедур. Условия типа имеют вид

```
<Условия типа>   ::= 'TYPE'  <Имя типа> ';'
                       { <Описание условия> }  { <Группа условий> }

<Группа условий> ::= "="  <Сообщение группы> [ ':' <Условие> ';' ]
                          { <Описание условия> }

<Сообщение группы> ::= <все символы от '=' до '\n' - конца строки
                                        или до ':'>

<Описание условия> ::= "-" <Сообщение> ':' <Условие> ';'

<Сообщение>        ::= <все символы от '-' до ':'>

<Условие>          ::= <все символы от ':' до ';'> | 'OTHER'
```

В описании условия <Сообщение> — это строка символов. Генератор отчетов использует ее при выдаче сообщений (если соответствующее тестовое условие не покрыто).

<Условие> — это выражение типа BOOLEAN, допустимое в языке Оберон-2 или Модула-2. Инструментатор вставляет это условие в инструментируемую программу в необходимых местах. Синтаксическая правильность этого условия не проверяется. Поэтому, если оно задано пользователем синтаксически неправильно, при компиляции инструментированной программы будет выдана ошибка. Чтобы избежать данной ситуации, можно воспользоваться утилитой проверки octcheck.exe.

В <Условии> обычно должен быть символ @. Вместо него инструментатор подставляет нужное имя параметра процедуры. Последним в списке условий возможно использование слова OTHER. Вместо него инструментатор вставляет TRUE.

Например, следующим образом можно определить тестовые условия для типа LIST из модуля List:

```
MODULE List;
  TYPE LIST;
    - пустой список          : List.empty(@);
    - одноэлементный список  : List.empty(List.tail(@));
    - многоэлементный список : OTHER;
END List.
```

Можно задать также несколько групп тестовых условий. При желании также можно указать сообщение группы и условие группы. В приведенном выше примере была только одна группа, для которой сообщение и условие отсутствовали. Группа может быть только одна, и она должна стоять вначале. Пример приведенный ниже пояснит использование групп.

Пусть в модуле Shapes имеются определения:

```
(* Shapes.def: *)
      . . .
      CONST min_x=0;
            min_y=0;
            max_x=100;
            max_y=100;

      TYPE POINT=RECORD
                   x: min_x..max_x;
                   y: min_y..max_y;
                   visible: BOOLEAN;
                 END;
      . . .
```

Для типа POINT можно определить тестовые условия следующим образом:

```
MODULE Shapes;
  TYPE POINT;
      - точка в верхнем левом углу :
              (@.x=Shapes.min_x) AND (@.y=Shapes.min_y);
      - точка в нижнем правом углу :
              (@.x=Shapes.max_x) AND (@.y=Shapes.max_y);
    = точка
      - видна    : @.visible;
      - не видна : OTHER;
    = точку можно увидеть : @.visible;
      - вверху : @.x <= (Shapes.max_x-Shapes.min_x)/2;
      - внизу  : OTHER;
    = точка на границе : (@.x=Shapes.min_x) OR (@.x=Shapes.max_x) OR
                         (@.y=Shapes.min_y) OR (@.y=Shapes.max_y);
      - видна    : visible;
      - не видна : OTHER;
END Shapes.
```

Описание тестовых условий для вызовов процедур

Описание тестовых условий этого вида использует критерий покрытия вызовов процедур и имеет следующий вид:

```
<Условия вызова> ::= 'CALL' <Заголовок процедуры> ';'
                       { <Описание условия> } { <Группа условий> }

<Заголовок процедуры> ::=
   'PROCEDURE' <Имя процедуры> ['('<Параметры>')'[':'<Результат>]]

<Параметры> ::= <все символы от '(' до ')'>

<Результат> ::= <все символы от ':' до ';'>

<Группа условий> ::=
   '=' [ 'AFTER:' ] <Сообщение группы> [ ':' <Условие> ';' ]
                     { <Описание условия> }
```

<Заголовок процедуры> можно получить следующим образом:

— скопировать заголовок исходной процедуры;

— добавить слово CALL перед ним;

— добавить при необходимости имя модуля к типам в заголовке, находящимся внутри этого модуля. Также следует вместо `INTEGER` и `CARDINAL` использовать `OctLog.INT` и `OctLog.CARD` соответственно.

После этого нужно описать тестовые условия для вызова процедуры, что делается так же, как и для типов, только здесь нельзя использовать `@`, а надо использовать имена формальных параметров процедуры. Кроме того, можно использовать глобальные переменные, которые экспортируются.

Есть еще одно отличие. Перед сообщением группы может стоять слово `AFTER`, означающее, что условия этой группы будут проверяться после вызова процедуры. Это важно, когда процедура имеет побочный эффект или формальные VAR-параметры.

Например, при описании

```
MODULE LongStr;
  CALL PROCEDURE StrToReal (str: ARRAY OF CHAR;
                            VAR real: LongStr.float;
                            VAR res:  LongStr.ConvResults);
    = AFTER: результат преобразования
      - все нормально               : res=ConvTypes.strAllRight;
      - значение не  может
        быть представленно          : res=ConvTypes.strOutOfRange;
```

```
      - неправильный формат строки : res=ConvTypes.strWrongFormat;
      - данная строка пуста        : res=ConvTypes.strEmpty;
END LongStr.
```

будет проверяться выполнение того, что в каждом месте вызова процедуры `StrToReal` значение `res` принимало все перечисленные значения после исполнения процедуры.

В группах условий вызова с пометкой `AFTER` можно использовать `RESULT` — результат, возвращенный процедурой-функцией. Например:

```
MODULE Dialogs;
  CALL PROCEDURE user_reply(): CHAR;
    = AFTER: ответ пользователя
      - да                 :  (RESULT='y') OR (RESULT='Y');
      - нет                :  (RESULT='n') OR (RESULT='N');
      - недопустимый ответ :  OTHER;
END Dialogs.
```

Условия являются выражениями типа `BOOLEAN` языков Модула-2 или Оберон-2, и их синтаксическая правильность не проверяется, если при написании этих выражений допущена ошибка, это обнаружится лишь при компиляции инструментированного модуля (или группы модулей), что может привести к большому числу однотипных сообщений об ошибках во многих модулях. Аналогичные ошибки возможны также для заголовков процедур, имен модулей и типов. Для того чтобы избежать этого, была создана утилита проверки `octcheck.exe`, которая для данного файла с описанием тестовых условий создает файл `checkme.ob2` на языке Оберон-2 (или `checkme.mod` на Модула-2). Этот файл пользователь должен проверить с помощью стандартного компилятора `XDS` и исправить найденные при его компиляции ошибки в исходном файле с описаниями тестовых условий. В результате инструментированные модули не будут содержать синтаксически некорректных выражений и ошибок при трансляции.

Пример использования системы OCT

Допустим, нужно протестировать итерактивный калькулятор, в котором можно использовать целые и вещественные числа в формате, допустимом в языке ПАСКАЛЬ, и арифметические операции `'+'`,`'-'`,`'*'` и `'/'`. Исполнив инструментированную программу на некотором наборе тестов, получим отчет, фрагмент которого для отдельной процедуры программы приводится ниже.

```
     PROCEDURE readtoken;
       . . .
     BEGIN <16>
!!->   LOOP <*,16,0>
-- Цикл ни разу не исполнялся много (>1) раз
         oldc := c;   get(c);
         CASE oldc OF
!!->     | ' ' : <0>
-- Альтернатива ни разу не исполнялась
         | '+' : <1>  token := add; EXIT;
         | '-' : <2>  token := sub; EXIT;
         | '*' : <1>  token := mul; EXIT;
         | '/' : <1>  token := div; EXIT;
         | '(' : <1>  token := LeftParen; EXIT;
         | ')' : <1>  token := RightParen; EXIT;
         | eol : <2>  token := end; EXIT;
         | '0'..'9' : <7>  (* read a real number *)
           i := 1; s[0] := oldc;
!!->       WHILE <28,7>  (c >= '0') AND <28,0> (c <= '9') DO <1,3,3>
-- Подусловие ни разу не было FALSE
             s[i] := c;   INC(i);   get(c);
           END;
           IF <5,2>  c<>'.' THEN
                 (* add decimal point if none in input *)
             s[i] := '.';
             INC(i);
           ELSE
!!->         REPEAT <*,0,2>  (* read fraction part *)
-- Цикл ни разу не исполнялся 1 раз
               s[i] := c;
               INC(i);
               get(c);
!!->         UNTIL <2,6>  (c < '0') OR <0,6>  (c > '9');
-- Подусловие ни разу не было TRUE
           END;
           s[i] := 0C;
!!->       LongStr.StrToReal <7,0%1%,0%2%,0%3%>
             (s, TokenNumberValue, conv_res);
-- %1% Ни разу не было: результат преобразования -
                          значение не может быть представлено
-- %2% Ни разу не было: результат преобразования -
                          неправильный формат строки
```

```
-- %3% Ни разу не было: результат преобразования -
                          данная строка пуста
!!->       IF <0,7>  conv_res <> ConvTypes.strAllRight THEN
-- Условие оператора IF ни разу не было TRUE
             error('Bad number?');
           END;
           token := number;
           EXIT;
!!->     ELSE <0>
-- Альтернатива ни разу не исполнялась
           error('Bad character');
         END;
       END;
     END readtoken;
```

Замечание. При печати отчета текст модуля (в данном случае — процедуры) изменяется. В местах расположения всех тестовых элементов появляются значения счетчиков, показывающих сколько раз данный элемент был покрыт (в виде <5,7>). Если есть непокрытые тестовые элементы, то строки, их содержащие, помечаются **!!->** и добавляется строка следом, которая содержит сообщение, раскрывающие смысл непокрытого тестового элемента (см., например, 4-ю строку процедры).

```
!!->   LOOP <*,16,0>
-- Цикл ни разу не исполнялся много (>1) раз
```

Дополним набор тестов, анализируя этот отчет. При этом не только будем строить тесты, которые покрыли бы непокрытые тестовые условия, но попытаемся также выявить слабые места данного набора тестов и найти ошибки, которые не могут быть найдены только покрытием тестовых условий.

Например, таким образом обнаруживается первая ошибка в этой процедуре. При изучении сообщения

```
!!->      | ' ' : <0>
-- Альтернатива ни разу не исполнялась
```

можно прийти к выводу, что плохо проверялось использование разделителей во входной строке и поэтому стоит протестировать программу на такой входной строке, в которой есть не только пробелы, но и символы табуляции, конца строки, конца файла и т. д. В результате после исполнения соответствующего теста обнаруживаем, что программа неправильно работает, если встречается символ табуляции, а для пробела — все в норме, т. е. построение теста только для того, чтобы удовлетворить критерию, не привело бы к обнаружению ошибки.

Рассмотрим теперь следующее сообщение:

```
!!->       WHILE <28,7>  (c >= '0') AND <28,0>  (c <= '9') DO <1,3,3>
-- Подусловие ни разу не было FALSE
```

Это тестовое условие можно покрыть, если выбрать тест, где после нескольких цифр некоторого числа встречается какая-нибудь буква, например `123F`. Из этого сразу делаем вывод, что программа плохо проверялась на неправильных входных данных, но, немного подумав, вспомним, что число может быть правильным, даже если после цифр встречается буква, а именно это числа типа `12E-2`. В программе такой случай вообще не рассмотрен и, таким образом, обнаружена довольно серьезная ошибка.

Однако некоторые ошибки находятся и после простого построения теста для выполнения тестового условия, например для следующего тестового условия:

```
!!->         REPEAT <*,0,2>  (* read fraction part *)
-- Цикл ни разу не исполнялся 1 раз
```

Этот цикл исполняется N раз, если число имеет $N-1$ цифр после запятой. Числа вида `23.`, вообще говоря, программа должна считать неправильными, но, исполнив соответствующий тест, убеждаемся, что программа этого не делает.

Для следующих тестовых условий сразу видно, что второе и третье условия являются нереализуемыми.

```
!!->       LongStr.StrToReal <7,0%1%,0%2%,0%3%>
              (s, TokenNumberValue, conv_res);
-- %1% Ни разу не было: результат преобразования -
                        значение не может быть представленно
-- %2% Ни разу не было: результат преобразования -
                        неправильный формат строки
-- %3% Ни разу не было: результат преобразования -
                        данная строка пуста
```

Для того чтобы покрыть первое тестовое условие, попробуем ввести какое-нибудь очень длинное число. В результате, хотя это и не привело к покрытию этого тестового условия, но привело к тому, что переполнился буфер `s`, выявлена еще одна ошибка.

Изучая два последних непокрытых тестовых условия в этой процедуре, приходим к выводу, что набор тестов имеет серьезный недостаток: совсем не проверяет работу программы на неправильных данных. Поэтому следует забыть об этих конкретных тестовых условиях и дополнить набор тестов такими тестами, в которых было бы как можно больше разнообразных неправильных данных.

Итак, в этой процедуре найдены четыре ошибки разной степени серьезности, причем только две из них явились непосредственным следствием построения тестов для покрытия тестовых условий. Другие же две, в том числе одна очень серьезная, были найдены только благодаря анализу слабости набора тестов, с использованием при этом сообщения о непокрытых элементах как косвенную информацию. Кроме этого, был выявлен серьезный недостаток набора тестов, который мог привести к тому, что некоторые ошибки остались бы незамеченными.

Замечание. Мы рассмотрели два способа реализации систем контроля полноты набора тестов на примерах систем TGS и OCT. В первой автоматически расставляются точки останова в местах контроля и пишутся методы их обработки. Во второй инструментируется **исходный текст** модуля. Второй метод получил наибольшее распространение из-за его универсальности, возможности применить к любым критериям. Однако возможны и другие подходы. Так система OCT дополнительно была реализована инструментацией **внутреннего представления** программы при ее компиляции в системе XDS. Фактически во внутреннее представление программы, автоматически при задании определенной прагмы, вносятся изменения так, чтобы обеспечивался сбор и сохранение информации, необходимой для оценки полноты набора тестов, например, так же как и при инструментации исходного кода. Такой подход позволяет уменьшить затраты машинного времени на трансляции и скрыть «кухню» инструментации от пользователя.

7.4. Исполнение и оценка результатов этого исполнения для тестового набора

В процессе тестирования программы тестировщик разрабатывает тесты и далее исполняет тестируемую программу на этих тестах. Таким образом необходимо обеспечить

— хранение разработанных тестов (в даном случае под тестом мы понимаем набор входных данных необходимых нашей программе);

— исполнение тестируемой программы на данных тестах;

— формирование и сохранение результата исполнения теста;

— оценка правильности полученного результата.

Из всех перечисленных задач наиболее трудоемкой является последняя — оценка правильности полученного результата исполнения программы.

Рассмотрим основные подходы при автоматизации этих процессов. Для хранения тестов можно использовать как различные базы данных, так и просто набор директорий файловой системы, располагая различные типы тестов по разным директориям. Это необходимо для того чтобы можно было просто организовать исполнение нашей тестируемой программы на разных типах тестов. Для возможности оценки правильности исполнения тестируемой программы необходимо формировать вывод результата ее исполнения. Например, в работе [49] приводятся следующие имеющиеся у нас возможности.

• **Файл выходных данных.** Идеальным случаем является программа, которая сама сохраняет свои данные в файлах, причем в форматах, подходящих для тестирования.

• **Перенаправление вывода в файл.** В файл на диске можно, например, перенаправить вывод, предназначенный для принтера. Причем перенаправление вывода извне предпочтительнее использования функции самой программы, поскольку гарантирует, что в файл попадет в точности та же информация, что и на принтер.

Возможно перенаправление вывода в файл и при выводе на экран, например, если программа работает в текстовом режиме.

• **Вывод через последовательный порт.** Если имеется возможность удаленного управления компьютером с терминала либо с другого компьютера посредством специального программного обеспечения, этот второй компьютер может записывать принимаемые выходные данные в файл для последующего анализа.

• **Перехват экрана.** Существует множество программ, позволяющих сохранять изображение текущего окна и всего экрана.

• **Перехват вывода программы с помощью специальных тестировочных утилит.** Совершенно очевидно, что программа, разработанная специально для тестирования, имеет гораздо больше возможностей чем утилиты общего пользования.

К этому необходимо добавить следующее: если у нас есть возможность формирования содержимого файла результата, то необходимо структурирование содержащейся в нем информации таким образом, чтобы в дальнейшем можно было просто обеспечить сравнение этих файлов.

Оценка выходной информации. Записав выходную информацию, необходимо оценить ее правильность. Как это сделать? Существует несколько традиционных подходов.

• **Эталонный результат.** Есть подготовленный, например, вручную, правильный, ожидаемый результат исполнения программы.

• **Если существует эталонная программа,** выполняющая то же самое, что и тестируемая, можно сравнить их выводы.

• **Сформировать базу данных правильных выходных данных.** При этом для каждого нового теста придется создавать и новые эталоны выходных данных. Это довольно долгий и трудоемкий процесс, особенно если данные эталонных файлов придется вводить вручную. Здесь неизбежны издержки по выявлению множества ошибок и опечаток. Однако основной объем придется на самое начало.

• **Перехват вывода программы.** Сохраняйте результаты каждого теста, какими бы они ни были. Результат теста записывается в отдельный файл, анализируется сразу или позднее и помечается как хороший или плохой.

После того как тесты будут выполнены повторно, их результаты снова записываются в файлы и сравниваются с предыдущими. Эта работа может быть автоматизирована. Несовпадение нового файла со старым, помеченным как хороший, означает либо появление новой ошибки, либо изменение спецификации. Если очередной результат теста оказывается хорошим, то все предыдущие плохие результаты можно удалить.

Таким образом, база данных результатов тестов постепенно растет, пополняясь все новыми данными и в конечном счете в ней оказывается по файлу на каждый тест, причем в этих файлах только правильные (эталонные) результаты[1].

В организации корректного сравнения выходных данных, особенно сравнений копий экрана, существует целый ряд трудностей. Как, например, сравнивающая программа должна игнорировать различающиеся даты? Что если данные выводятся с различной степенью точности, в разном порядке или в разных местах экрана?

Замечение. Необходимость данной автоматизации особенна актуальна всвязи с *регрессионным* тестированием (основная работа тестировщика). У этого термина два значения, объединеных идеей повторного использования разработанных тестов:

• Вы разработали тест, обнаружили ошибку, и программист ее исправил. Вы снова исполняете этот тест, чтобы убедиться, что ошибки больше нет. Это и есть регрессионное тестироваие. Можно исполнить несколько вариаций исходного теста, чтобы как следует проверить ис-

[1] Конечно, если мы не ошиблись при оценке правильности результата.

правленный фрагмент программы. В данном случае задача регрессионного тестирования состоит в том, чтобы убедиться, что выявленная ошибка полностью исправлена и больше не проявляется.

Возможен подход, когда в набор регрессионных тестов включаются все тесты, обнаружившие ошибку, даже если она исправлена давным-давно. Каждый раз, когда в программу вносятся изменения, все эти тесты исполняются снова. Особенно важно провести такое обстоятельное тестирование, если программа изменяется спустя достаточно длительное время или новым программистом.

• Второй пример применения регрессионного тестирования. После выявления и исправления ошибки проводится стандартная серия тестов, но уже с другой целью: убедиться, что, исправляя одну часть программы, программист не испортил другую. В этом случае тестируется целостность всей программы, а не правильное испраление одной ошибки.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Зелковец М. и др.* Принципы разработки программного обеспечения / М. Зелковец, А. Шоу, Дж. Гэннон. М.: Мир, 1982.

2. *Майерс Г.* Искусство тестирования программ. М.: Финансы и статистика, 1982.

3. *Черноножкин С. К.* Меры сложности программ: (Обзор) // Системная информатика. Новосибирск: Наука, 1997. Вып. 5: Архитектурные, формальные и программные модели. С. 188—227.

4. *Черноножкин С. К.* Меры сложности программ: (Обзор). Новосибирск, 1994. 36 с. (Препринт / СО РАН. ИСИ; N 21).

5. *Борзов Ю. В.* Тестирование программ с использованием символического выполнения // Программирование. 1980. N 1. С. 51—60.

6. *Захаров Л. А.* Организация средств тестирования и отладки в кросс-системе программирования // Среда программированния: методы и инструменты. Новосибирск, 1992. С. 68—79.

7. *Иыуду К. А., Арипов М. М.* Автоматизация генерации путей для тестирования программ, написанных на Фортране // Программирование. 1986. N 2. С. 24—31.

8. *Архангелъский Б. В., Абдуллаев Х. Х.* ТЕСТОР-ФОРТРАН — система тестирования заранее заданных путей управляющего графа программы // Управляющие системы и машины. 1990. N 5. С. 62—72.

9. *Кауфман А. В., Черноножкин С. К.* Инструменты поддержки структурного тестирования в системе СОКРАТ // Средства и инструменты окружений программирования. Новосибирск, 1995. С. 30—45.

10. *Кауфман А. В., Черноножкин С. К.* Структурное тестирование в системе СОКРАТ // Программные системы. Новосибирск, 1996. С. 135—148.

11. *Кауфман А. В., Черноножкин С. К.* ОСТ: система контроля тестированности Модула-2-программ. Новосибирск, 1997. 46 с. (Препринт / СО РАН. ИСИ; N 38).

12. *Кауфман А. В., Черноножкин С. К.* Критерии тестирования и система оценки полноты набора тестов // Программирование 1998. N 6. С. 44—59.

13. *Rapps S., Weyuker E.* Selecting software test data using data flow information // IEEE Trans. on Software Eng. 1985. Vol. SE-14, N 4. P. 367—375.

14. *Кузьминов Т. В.* Программные интерфейсы в трансляторе Модула-Y // Среда программирования: методы и инструменты. Новосибирск, 1992. С. 61—67.

15. *Грис Д.* Наука программирования: Пер. с англ. М: Мир, 1984.

16. *Поттосин И. В.* Система СОКРАТ: окружение программирования для встроенных систем. Новосибирск, 1992. 20 с. (Препринт / РАН. Сиб. отд-ние. ИСИ; N 11).

17. *Шимаров В. А.* Об одном методе тестирования программы на основе минимального покрытия ее графа // Управляющие системы и машины. 1988. N 5. С. 51—52.

18. *Касьянов В. Н.* Оптимизирующие преобразования программ. М.: Наука. Сиб. отд-ние, 1988.

19. *Marick B.* The craft of software testing. — London Englewood Cliffs: Prentice Hall, 1995.

20. *Marick B.* FAQ: testing tool suppliers (публикуется ежемесячно в телеконференции comp.software.testing).

21. *Marre M., Bertolino A.* Reducing and Estimating the Cost of Test Coverage Criteria. IEEE Computer Press, Los Alamitos, California // Proc. ICSE-18, 1996. P. 486—494.

22. *Harrold M., Gupta R., Soffa M.* A methodology for controlling the size of a test suite // ACM Trans. on Software Eng. and Meth. 1993. Vol. 2, N 3. P. 270—285.

23. *Шимаров В. А.* Иерархический подход к оценке сложности тестирования программ. Минск, 1989. 46 с. (Препринт / АН БССР. Ин-т математики; N 46 (396)).

24. *Шелехов В. А., Куксенко С. В.* Статический анализатор семантических ошибок периода исполнения // Программирование. 1998, N 6, С. 23—43.

25. *Budd T.* Mutation analysis: ideas, examples, problems and prospects // Computer Program Testing / Ed. by B. Chandrasekaran and S. Radicchi. Amsterdam: North-Holland Publ., SOGESTA, 1981. P. 129—148.

26. *Untch R., Offut A., Harrold M.* Mutation analysis using mutant schemata. // Software Eng. Notes. 1993. Vol. 18, N 4. P. 139—148.

27. *Offut A., Rotherman G., Zapf C.* An experimental evaluation of selective mutation. // Fifteenth International conference on Software Engineering. 1993. May. Baltimore, Maryland. P. 100—107.

28. *Offut A., Pan J.* Automatically detecting equivalent mutants and infeasible paths. // The J. of Software Testing, Verification and Reliability. 1997. September. Vol 7, N 3. P. 165—192.

29. *Mathur A. P.* Performance, effectiveness, and reliabilite issues in software testing. // IEEE Proceedings of 15-th annual international computer software & application conference. 1991. P. 604—605.

30. *Howden W. E.* Weak mutation testing and completeness of test sets. // IEEE Trans. on Software Eng. 1982. July. Vol. SE-8, N 4. P. 371—379.

31. *Offutt A., Lee Ss.* An empirical evaluation of weak mutation. // IEEE Trans. on Software Eng. 1994. May. Vol. 20, N 5. P. 337—344.

32. *Morell L. J.* A theory of error-based testing. // PhD thesis, university of Maryland. Technical report TR-1395.

33. *Offutt A.* Investigations of the software coupling effect. // ACM Trans. on Software Eng. Methodology 1992. January. 1(1). P. 3—18.

34. *How Tai Wah K. S.* Fault coupling in finite bijective functions. // The Journal of Software Testing, Verification and Reliability. 1995. March. 5(1). P. 3—47.

35. *DeMillo R., Guingi D., Offutt A., King N.* An extended overview of the Mothra software testing environment. // IEEE Proceeding of Second Workshop on Software Testing, Verification and Analysis. 1988. July. P. 142—151.

36. *Woodward M. R., Halewood K.* From weak to strong dead or alive?? An analysis of some mutation testing issues. // IEEE proceedings of second workshop on software testing, verification, and analysis. 1988. July. P. 152—158.

37. *Laski J., Szermer W., Luczycki P.* Dynamic mutation testing in integrated regression analysis. — IEEE Computer Press, Los Alamitos, California // Proc. 15th Intern. Conf. on Software Eng. 1993. P. 108—117.

38. *Frankl P., Weyuker E.* An applicable family of data flow testing criteria // IEEE Trans. on Software Eng. 1988. Vol. SE-14, N 10. P. 1483—1498.

39. *Laski J., Korel B.* A data flow oriented program testing strategy // IEEE Trans. on Software Eng. 1983. Vol. 9, N 3. P. 347—354.

40. *Катков В. П.* РИТМ-технология автоматизации программирования. Минск, 1993.

41. *Marick B.* The generic coverage tool (GCT). 1993. 28 Jan. (Доступно по ftp://cs.uiuc.edu/pub/testing).

42. *Marick B.* Experience with the cost of different coverage goals for testing. Motorola, 1990.

43. *xTech* Development System. Native XDS v 2.12 for IBM Operating System/2. User's Guide. xTech Ltd., 1995.

44. *Евстигнеев А. В.* Применение теории графов в программировании. М.: Наука, 1985.

45. *Йодан Э.* Структурное проектирование и конструирование программ. М.: Мир, 1979.

46. *Clarke L., a. o.* A formal evaluation of data flow path selection criteria // IEEE Trans. on Software Eng. 1989. Vol. 15, N 11. P. 244—251.

47. *Frankl P., Weyuker E.* Provable improvements on branch testing // IEEE Trans. on Software Eng. 1993. Vol. 19, N 10. P. 962—975.

48. *Marick B.* The Craft of Software Testing. vol 1: Subsystem Testing. Illinois, Testing Foundation; 1993.

49. *Сэм Канер и др.* Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений / С. Канер, Д. Фолк, Е. К. Нгуен. Киев: ДиаСофт, 2001. 544 с.

ПРИЛОЖЕНИЕ

Библиотека проверки выполнения тестовых условий

В приложении приводятся фрагменты модулей `OctLog.def` и `OctLog.mod`, необходимые для понимания того, каким образом производится регистрация выполнения тестовых условий в инструментированной программе.

```
DEFINITION MODULE OctLog; (* 17-May-95 *)
  (* Поддержка регистрации покрытия тестовых условий. *)

TYPE LOG;
(* регистрация для модуля - хранилище тестовых условий *)

  .  .  .

TYPE INT=INTEGER;

PROCEDURE new_log(regs_num: CARDINAL;
                  module_name: ARRAY OF CHAR): LOG;
  (* 1) Создает регистрацию с regs_num тестовых условий;
     2) Считывает файл <module_name>.log (если он есть) и
        заполняет им созданную регистрацию.
   *)

PROCEDURE reg(into: LOG; num: CARDINAL);
```

```
  (* Зарегистрировать в регистрации into тестовое
     условие с номером num *)
  (* 1<=num<=into.regs_num *)

PROCEDURE reg_bool(log: LOG; num: CARDINAL;
                   cond: BOOLEAN): BOOLEAN;
   (* Предназначен для инструментации условия
      (выражения типа BOOLEAN).
      В зависимости от cond зарегистрировать:
                     [num для TRUE, num+1 для FALSE].
      Возвращается cond.  *)

(*********************************************************)
PROCEDURE reg_init_loop(log: LOG; num: CARDINAL;
                        VAR count: INT);
   (* Регистрация инициализации цикла.
              count-вспомогательная переменная.
      [num, num+1, num+2] - тестовые условия для
                            [0,1,m - исполнений цикла].
    *)
PROCEDURE reg_entry_loop(log: LOG; num: CARDINAL; VAR count: INT);

PROCEDURE reg_proc_entry(log: LOG; num: CARDINAL;
                         VAR count, direction: INT);
PROCEDURE reg_proc_exit (log: LOG; num: CARDINAL;
                         VAR count, direction: INT);
(* Регистрация покрытия тестовых условий для рекурсивных процедур:
     [num,num+1,num+2] - для глубины рекурсивного вызова [0,1,m].
     ..._entry - должна вызываться в начале исполнения процедуры,
     ..._exit  - в конце   *)
(* При самом первом вызове reg_proc_entry должно быть count=0. *)

(*************** Для критерия покрытия отношений ***************)
CONST
  lt=0;   le=1;   gt=2;   ge=3;

PROCEDURE reg_card_rel(log: LOG; num: CARDINAL; rel: CARDINAL;
                       c1,c2: LONGCARD): BOOLEAN;
          (* reg({c1=c2};  RETURN c1 <rel> c2 *)

PROCEDURE reg_int_rel(log: LOG; num: CARDINAL; rel: CARDINAL;
                      c1,c2: LONGINT): BOOLEAN;
          (* reg({c1=c2};  RETURN c1 <rel> c2 *)
(*************************************************************)

PROCEDURE write_logs;
(* Сохраняет все регистрации, созданные с помощью new_log
   в соответствующих *.log-файлах. *)
```

```
END OctLog.

(** OCT **)
(*  предотвращает инструментацию этого модуля *)
IMPLEMENTATION MODULE OctLog; (* 17-May-95 *)

  .  .  .

PROCEDURE reg(into: LOG; num: CARDINAL);
  VAR index: CARDINAL;
BEGIN
  index:= into^.first+num-1;
  IF regs[index]<MAX(CARDINAL) THEN
     INC(regs[index]);
  END(*IF*);
END reg;

PROCEDURE unreg(into: LOG; num: CARDINAL);
  (* отменить предыдущую reg(into, num) *)
  VAR index: CARDINAL;
BEGIN
  index:= into^.first+num-1;
  IF regs[index]>0 THEN
     DEC(regs[index]);
  END(*IF*);
END unreg;

PROCEDURE reg_bool(log: LOG; num: CARDINAL;
                        cond: BOOLEAN): BOOLEAN;
BEGIN
  IF cond THEN
     reg(log, num);
  ELSE
     reg(log, num+1);
  END(*IF*);
  RETURN cond;
END reg_bool;

PROCEDURE reg_init_loop(log: LOG; num: CARDINAL;
                                  VAR count: INT);
BEGIN
  count:= 0;
  reg(log, num);
END reg_init_loop;

PROCEDURE reg_entry_loop(log: LOG; num: CARDINAL;
                             VAR count: INT);
```

```
BEGIN
  INC(count);
  IF count<=2 THEN
     unreg(log, num+VAL(CARDINAL,count)-1);
     reg(log, num+VAL(CARDINAL,count));
  END;
END reg_entry_loop;

PROCEDURE reg_proc_entry(log: LOG; num: CARDINAL;
                         VAR count, direction: INT);
BEGIN
  INC(count);
  direction:= 1;
END reg_proc_entry;

PROCEDURE reg_proc_exit(log: LOG; num: CARDINAL;
                        VAR count, direction: INT);
BEGIN
  DEC(count);
  IF direction=1 THEN (* зарегистрировать *)
     CASE count OF
     | 0:  reg(log, num);
     | 1:  reg(log, num+1);
     ELSE  reg(log, num+2);
     END;
  END;
  direction:= -1;
END reg_proc_exit;

PROCEDURE reg_card_rel(log: LOG; num: CARDINAL;
                       rel: CARDINAL;
                       c1,c2: LONGCARD): BOOLEAN;
         (* reg({c1=c2};  RETURN c1 <rel> c2 *)
BEGIN
  IF c1=c2 THEN
     reg(log, num);
  END;
  CASE rel OF
  | le:  RETURN c1<=c2;
  | lt:  RETURN c1<c2;
  | ge:  RETURN c1>=c2;
  | gt:  RETURN c1>c2;
  END;
END reg_card_rel;

PROCEDURE reg_int_rel(log: LOG; num: CARDINAL;
                      rel: CARDINAL;
                      c1,c2: LONGINT): BOOLEAN;
```

```
         (* reg({c1=c2};  RETURN c1 <rel> c2 *)
BEGIN
  . . . (* точно так же, как у reg_card_rel *)
END reg_int_rel;

BEGIN
  . . .
FINALLY
  write_logs;
END OctLog.
```