Тестирование Веб-приложений

Введение

Вычислительные и коммуникационные системы используются все чаще и с каждым днем все глубже входят в нашу повседневную жизнь. Компании и отдельные пользователи все больше зависят в своей работе от web-приложений. Веб-приложения соединяют различные отделы внутри компаний, различные компании и простых пользователей. Веб-приложения очень динамичны, а их функциональные возможности непрерывно растут. Непрерывно возрастает потоковый трафик средств информации и запросов, формируемых переносными и встроенными устройствами. Вследствие этого возрастает сложность систем такого рода. Очевидно, что для понимания, анализа, разработки и управления такими системами нужны количественные методы и модели, которые помогают оценить различные *сценарии* функционирования, исследовать структуру и состояние больших систем. Наблюдаются тенденции к постоянному росту спроса на Веб-службы. Таким образом, проблемы, связанные с недостаточной производительностью будут возникать и в будущем, и, в конце концов, они станут превалирующими при планировании и вводе в эксплуатацию новых Веб-служб и увеличении пользователей Интернета. Веб-приложения становятся все более распространенными и все более сложными, играя, таким образом, основную роль в большинстве онлайновых проектов. Как и во всех системах, основанных на взаимодействии между клиентом и сервером, уязвимости Веб-приложений обычно возникают из-за некорректной обработки запросов клиента и/или недостаточной проверки входной информации со стороны разработчика.

В первой части данной лекции мы рассмотрим вопросы специфичные для тестирования и *отладки* Веб-приложений.

Будут рассмотрены принципы следующих подходов к тестированию Веб-приложений [1, 2]:

- *функциональное тестирование* ;
- *тестирование пользовательского интерфейса* ;
- *тестирование удобства использования* ;
- нагрузочное и *стрессовое тестирование* ;
- *проверка ссылок* и HTML-кода;
- *тестирование безопасности*.

Также будет приведен обзор средств *автоматизации тестирования* Веб-приложений.

С общими вопросами тестирования и верификации информационных систем предлагается ознакомиться в курсе Интернет Университета Информационных Технологий "Верификация программного обеспечения" [3].

Во второй части лекции будут рассмотрены подходы и инструментальные средства *отладки* CSS, а также *отладки* и *профилирования* JavaScript.

19.1.2. Подходы к функциональному тестированию Веб-приложений

Функциональное тестирование (*functional testing*) – процесс верификации соответствия функционирования продукта его начальным спецификациям [4].

Характерным примером может быть проверка того, что программа подсчета выплат по банковской ссуде выдает корректные выкладки на любые введенные сумму ссуды и срок ее возврата. Обычно подобные проверки проводятся вручную, иногда к этому подключаются конечные пользователи в качестве бета-тестеров. Однако программные системы становятся все сложнее, а комбинации различных входных параметров и поддерживаемых операционных систем нередко исчисляются десятками и сотнями.

Перечислим некоторые из методов *функционального тестирования* веб-приложений [5, 6]:

1. *Record & Play* – основан на возможности средств *автоматизации тестирования* автоматически генерировать код.
2. *Functional Decomposition* – в основе лежит разбиение всех компонент фреймворка по функциональному признаку на бизнес-функции (реализуют/проверяют бизнес-функциональность приложения), user-defined функции (вспомогательные функции, которые еще имеют привязку к тестируемому приложению или к конкретному проекту), утилиты (функции общего назначения, не привязанные к конкретному приложению, технологии, проекту).
3. *Data-driven* – основан на том, что к некоторому тесту или группе тестов привязывается источник данных, и этот тест или набор тестов циклически выполняется для каждой записи из этого источника данных. Вполне может применяться в комбинации с другими подходами.
4. *Keyword-driven* – представляет собой фактически движок для обработки посылаемых ему команд, а сами инструкции выносятся во внешний источник данных.
5. *Object-driven* – основан на том, что основные ходовые части фреймворка реализованы в виде объектов, что позволяет собирать тесты по кирпичикам.
6. *Model-based* – основан на том, что тестируемое приложение (или его части) описывается в виде некоторой поведенческой модели.

Самым распространенным является подход, называемый Capture & *Playback* (другие названия – Record & *Playback*, Capture & Replay) [6]. Суть этого подхода заключается в том, что *сценарии* тестирования создаются на основе работы пользователя с тестируемым приложением. Инструмент перехватывает и записывает действия пользователя, результат каждого действия также запоминается и служит эталоном для последующих проверок. При этом в большинстве инструментов, реализующих этот подход, воздействия (например, нажатие кнопки мыши) связываются не с координатами текущего положения мыши, а с объектами HTML-интерфейса (кнопки, поля ввода и т.д.), на которые происходит воздействие, и их атрибутами. При тестировании инструмент автоматически воспроизводит ранее записанные действия и сравнивает их результаты с эталонными, точность сравнения может настраиваться. Можно также добавлять дополнительные проверки – задавать условия на свойства объектов (цвет,

расположение, размер и т.д.) или на функциональность приложения (содержимое сообщения и т.д.).

Основное достоинство этого подхода – простота освоения. Создавать тесты с помощью инструментов, реализующих данный подход, могут даже пользователи, не имеющие навыков программирования.

Вместе с тем, у подхода имеется ряд существенных недостатков. Для разработки тестов не предоставляется никакой автоматизации; фактически, инструмент записывает процесс *ручного тестирования*. Если в процессе записи теста обнаружена ошибка, то в большинстве случаев создать тест для последующего использования невозможно, пока ошибка не будет исправлена (инструмент должен запомнить правильный результат для проверки). При изменении тестируемого приложения набор тестов трудно поддерживать в актуальном состоянии, так как тесты для изменившихся частей приложения приходится записывать заново.

Тестирование пользовательского интерфейса

Часть программной системы, обеспечивающая работу интерфейса с пользователем – один из наиболее нетривиальных объектов для верификации [7]. Нетривиальность заключается в двояком восприятия термина "пользовательский интерфейс".

С одной стороны пользовательский интерфейс – часть программной системы. Соответственно, на пользовательский интерфейс пишутся функциональные и низкоуровневые требования, по которым затем составляются тест-требования и тест-планы. При этом, как правило, требования определяют реакцию системы на каждый ввод пользователя (при помощи клавиатуры, мыши или иного устройства ввода) и вид информационных сообщений системы, выводимых на экран, печатающее устройство или иное устройство вывода. При верификации таких требований речь идет о проверке *функциональной полноты* пользовательского интерфейса – насколько реализованные функции соответствует требованиям, корректно ли выводится информация на экран.

С другой стороны пользовательский интерфейс – "лицо" системы, и от его продуманности зависит эффективность работы пользователя с системой. Факторы, влияющие на эффективность работы, в меньшей степени поддаются формализации в виде конкретных требований к отдельным элементам, однако должны быть учтены в виде общих рекомендаций и принципов построения пользовательского интерфейса программной системы. Проверка интерфейса на эффективность человеко-машинного взаимодействия получила название проверки удобства использования (usability verification, в русскоязычной литературе в качестве перевода термина usability часто используют слово "практичность").

Функциональное тестирование **пользовательского интерфейса** состоит из пяти фаз [7]:

- анализ требований к пользовательскому интерфейсу;
- разработка тест-требований и тест-планов для проверки пользовательского интерфейса;

- выполнение тестовых примеров и сбор информации о выполнении тестов;
- определение полноты покрытия пользовательского интерфейса требованиями.
- составление отчетов о проблемах в случае несовпадения поведения системы и требований, либо в случае отсутствия требований на отдельные интерфейсные элементы.

Все эти фазы точно такие же, как и в случае тестирования любого другого компонента программной системы. Отличия заключаются в трактовке некоторых терминов в применении к пользовательскому интерфейсу и в особенностях автоматизированного сбора информации на каждой фазе.

Так, тест-планы для проверки пользовательского интерфейса, как правило, представляют собой *сценарии*, описывающие действия пользователя при работе с системой. *Сценарии* могут быть записаны либо на естественном языке, либо на формальном языке какой-либо системы автоматизации пользовательского интерфейса. Выполнение тестов при этом производится либо оператором в ручном режиме, либо системой, которая эмулирует поведение оператора.

При сборе информации о выполнении тестовых примеров, как правило, применяются технологии анализа выводимых на экран форм и их элементов (в случае графического интерфейса) или выводимого на экран текста (в случае текстового), а не проверка значений тех или иных переменных, устанавливаемых программной системой.

Под полнотой покрытия пользовательского интерфейса понимается то, что в результате выполнения всех тестовых примеров каждый интерфейсный элемент был использован хотя бы один раз во всех доступных режимах.

Отчеты о проблемах в пользовательском интерфейсе могут включать в себя как описания несоответствий требований и реального поведения системы, так и описания проблем в требованиях к пользовательскому интерфейсу. Основной источник проблем в этих требованиях – их тестонепригодность, вызванная расплывчатостью формулировок и неконкретностью.

Тестирование пользовательского интерфейса может проводиться различными методами – как вручную при непосредственном участии оператора, так и при помощи различного инструментария, автоматизирующего выполнение тестовых примеров. Рассмотрим эти методы более подробно.

Ручное тестирование

Ручное тестирование **пользовательского интерфейса** проводится тестировщиком-оператором, который руководствуется в своей работе описанием тестовых примеров в виде набора *сценариев*. Каждый *сценарий* включает в себя перечисление последовательности действий, которые должен выполнить оператор и описание важных для анализа результатов тестирования ответных реакций системы, отражаемых в пользовательском интерфейсе. Типичная форма записи *сценария* для проведения *ручного*

тестирования – таблица, в которой в одной колонке описаны действия (шаги *сценария*), в другой – ожидаемая реакция системы, а третья предназначена для записи того, совпала ли ожидаемая реакция системы с реальной и перечисления несовпадений.

В табл. 19.1 приведен пример *сценария* для *ручного тестирования пользовательского интерфейса* [7].

Таблица 19.1. Пример *сценария* для *ручного тестирования пользовательского интерфейса*

№ п/п	**Действие**	**Реакция системы**	**Результат**
1	Щелкните на значке "Система" и выберите пункт меню "Администрирование системы".	Появится окно ввода логина и пароля	**Верно**
2	Введите в появившееся окно ввода имя пользователя "admin" и пароль "admin". Затем нажмите кнопку "OK".	Появится окно "Администрирование системы". В верхнем правом углу должно быть выведено имя вошедшего пользователя admin.	**Неверно** Окно имеет название "Управление системой"

Ручное тестирование пользовательского интерфейса удобно тем, что контроль корректности интерфейса проводится человеком, т.е. основным "потребителем" данной части программной системы. К тому же при чисто косметических изменениях в интерфейсах системы, не отраженных в требованиях (например, при перемещении кнопок управления на 10 пикселей влево) анализ успешности прохождения теста будет выполняться не по формальным признакам, а согласно человеческому восприятию.

При этом *ручное тестирование* имеет и существенный недостаток – для его проведения требуются значительные человеческие и временные ресурсы. Особенно сильно этот недостаток проявляется при проведении *регрессионного тестирования* и вообще любого повторного тестирования – на каждой итерации повторного *тестирования пользовательского интерфейса* требуется участие тестировщика-оператора. В связи с этим в последнее десятилетие получили распространение средства автоматизации *тестирования пользовательского интерфейса*, снижающие нагрузку на тестировщика-оператора.

Сценарии на формальных языках

Естественный способ автоматизации *тестирования пользовательского интерфейса* – использование программных инструментов, эмулирующих поведение тестировщика-оператора при *ручном тестировании пользовательского интерфейса*.

Такие инструменты используют в качестве входной информации *сценарии* **тестовых примеров**, записанные на некотором формальном языке,

операторы которого соответствуют действиям пользователя – вводу *команд, перемещению* курсора, активизации пунктов меню и других интерфейсных элементов.

При выполнении автоматизированного теста инструмент тестирования имитирует действия пользователя, описанные в *сценарии*, и анализирует интерфейсную реакцию системы. При этом для определения ожидаемого состояния пользовательского интерфейса могут использоваться различные методы – либо анализ снимков экрана и сравнение их с эталонными, либо доступ к данным интерфейсных элементов средствами операционной системы (например, доступ ко всем кнопкам окна по их дескрипторам и получение значений текста).

И при передаче информации в тестируемый интерфейс и при получении информации для анализа могут использоваться два способа доступа к элементам интерфейса [7]:

- позиционный, при котором доступ к элементу осуществляется при помощи задания его абсолютных (относительно экрана) или относительных (относительно окна) координат и размеров;
- по идентификатору, при котором доступ к элементу осуществляется при помощи получения интерфейсного элемента при помощи его уникального идентификатора в пределах окна.

При внесении изменений в пользовательский интерфейс при использовании первого метода в результате проведения *регрессионного тестирования* будет выявлено большое количество не прошедших тестов – достаточно изменения местоположения одного ключевого интерфейсного элемента, как все *сценарии* начнут работать неверно. Соответственно при таком методе *автоматизации тестирования* необходимо менять значительную часть *сценариев* в системе тестов при каждом изменении интерфейса системы. Такой метод *автоматизации тестирования* подходит для систем с устоявшимся и редко изменяемым интерфейсом.

Второй метод *автоматизации тестирования* более устойчив к изменению расположения интерфейсных элементов, но изменения тестовых примеров могут потребоваться и здесь в случае изменения логики работы интерфейсных элементов. Например, пусть в первой версии системы при нажатии на кнопку "Передать данные" передача данных начиналась сразу, и выводилось окно с индикатором прогресса. *Сценарий* тестового примера в этом случае включает в себя имитацию нажатия на кнопку и обращение к индикатору прогресса для получения значения прогресса в процентах.

Тестирование удобства пользования

Тестирование удобства пользования – тест, который можно назвать практически главным для всех интерактивных сервисов, взаимодействующих с пользователем – это тест на "usability" или на удобство использования. Такое тестирование одно из самых дорогих, потому что наиболее ценную информацию можно получить только от реальных пользователей, наблюдая за их работой с вашим сайтом – а такие исследования требуют дорогостоящей инфраструктуры и времени, их сложно автоматизировать. Это метод

тестирования, направленный на установление степени удобства использования, обучаемости, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий.

Выделяют следующие этапы *тестирования удобства использования* пользовательского интерфейса [7]:

- Исследовательское – проводится после формулирования требований к системе и разработки прототипа интерфейса. Основная цель на этом этапе – провести высокоуровневое обследование интерфейса и выяснить, позволяет ли он с достаточной степенью эффективности решать задачи пользователя.
- Оценочное – проводится после разработки низкоуровневых требований и детализированного прототипа пользовательского интерфейса. Оценочное тестирование углубляет исследовательское и имеет ту же цель. На данном этапе уже проводятся количественные измерения характеристик пользовательского интерфейса: измеряются количество обращений к системе помощи по отношению к количеству совершенных операций, количество ошибочных операций, время устранения последствий ошибочных операций и т.п.
- Валидационное – проводится ближе к этапу завершения разработки. На этом этапе проводится анализ соответствия интерфейса программной системы стандартам, регламентирующим вопросы удобства интерфейса, проводится общее тестирование всех компонент пользовательского интерфейса с точки зрения конечного пользователя. Под компонентами интерфейса здесь понимается как его программная реализация, так и система помощи и руководство пользователя. Также на данном этапе проверяется отсутствие дефектов удобства использования интерфейса, выявленных на предыдущих этапах.
- Сравнительное – данный вид тестирования может проводиться на любом этапе разработки интерфейса. В ходе сравнительного тестирования сравниваются два или более вариантов реализации пользовательского интерфейса.

Тестирование удобства пользования дает оценку уровня удобства использования приложения по следующим пунктам [7]:

- производительность, эффективность (efficiency) – сколько времени и шагов понадобится пользователю для завершения основных задач приложения, например размещение новости, регистрации, покупка и т.д.
- правильность (*accuracy*) – сколько ошибок сделал пользователь во время работы с приложением
- активизация в памяти (*recall*) – как много пользователь помнит о работе приложения после приостановки работы с ним на длительный период времени? (повторное выполнение операций после перерыва должно проходить быстрее, чем у нового пользователя)
- эмоциональная реакция (emotional response) – как пользователь себя чувствует после завершения задачи – растерян, испытал стресс. Порекомендует ли пользователь систему своим друзьям.

Исследование и оценка сайтов может проводиться разными методами, разработанными экспертами по юзабилити. Общим для всех методов является постановка реальных задач перед пользователями, а также фиксирование результатов тестирования для дальнейшего анализа. Для участия в юзабилити-тестировании отбираются пользователи, соответствующие целевой аудитории, они также не должны быть слишком знакомы с разработкой. Для того чтобы выявить и оценить наибольшее количество присутствующих на сайте проблем, необходимо привлекать реальных пользователей [8].

На удобство использования пользовательского интерфейса влияют следующие факторы [7]:

- легкость обучения – быстро ли человек учится использовать систему;
- эффективность обучения – быстро ли человек работает после обучения;
- запоминаемость обучения – легко ли запоминается все, чему человек научился;
- ошибки – часто ли человек допускает ошибки в работе;
- общая удовлетворенность – является ли общее впечатление от работы с системой положительным.

Все эти факторы, несмотря на свою неформальность, могут быть измерены. Для таких измерений выбирается группа типичных пользователей системы, в процессе их работы с системой измеряются показатели их работы с системой (например, количество допущенных ошибок), а также им предлагается высказать собственные впечатления от работы с системой при помощи заполнения опросных листов.

Как правило, при *тестировании удобства использования* пользовательского интерфейса используются некоторые эвристические критерии и характеристики, которые заменяют точные оценки в классическом тестировании программных систем.

Например, Якоб Нильсен в своей работе [9] выделил 10 эвристических характеристик удобного пользовательского интерфейса, которые с его точки зрения должны проверяться при *тестировании удобства использования* интерфейса:

1. Наблюдаемость состояния системы – система всегда должна оповещать пользователя о том, что она в данный момент делает, причем через разумные промежутки времени;
2. Соотнесение с реальным миром – терминология, использованная в интерфейсе системы должна соотноситься с пользовательским миром, т.е. это должна быть терминология проблемной области пользователя, а не техническая терминология.
3. Пользовательское управление и свобода действий – пользователи часто выбирают отдельные интерфейсные элементы и используют функции системы по ошибке. В этом случае необходимо предоставлять четко определенный "аварийный выход", при помощи которого можно вернуться к предыдущему нормальному состоянию. К таким "аварийным выходам" относятся, например, функции отката и обратного отката.

4. Целостность и стандарты – для обозначения одних и тех же объектов, ситуаций и действий должны использоваться одинаковые слова во всех частях интерфейса. Более того, терминология сообщений в пользовательском интерфейсе должна учитывать соглашения конкретной платформы.

5. Помощь пользователям в распознавании, диагностике и устранении ошибок – Сообщения об ошибках должны быть написаны на естественном языке, а не заменяться кодами ошибок. Сообщения об ошибках должны четко определять суть возникшей проблемы и предлагать ее конструктивное решение.

6. Предотвращение ошибок – продуманный дизайн пользовательского интерфейса, предотвращающий появление ошибок пользователя всегда лучше хорошо продуманных сообщений об ошибках. При проектировании интерфейса необходимо либо полностью устранить элементы, в которых могут возникать ошибки пользователя, либо проверять ввод пользователя в этих элементах и сообщать ему о потенциально возможном возникновении проблемы.

7. Распознавание, а не вспоминание – при создании интерфейса необходимо минимизировать нагрузку на память пользователя, делая объекты, действия и опции ясными, доступными и явно видимыми. Пользователь не должен запоминать информацию при переходе от одного диалогового окна к другому. Во всех необходимых местах должны быть доступны контекстные инструкции по использованию интерфейса.

8. Гибкость и эффективность использования – в интерфейсе должны быть предусмотрены горячие клавиши (не обязательные к использованию начинающим пользователем) – они часто значительно ускоряют работу опытного пользователя. Иными словами, система должна предоставлять два способа работы – для новичков и для опытных пользователей. Желательно при этом давать возможность пользователю автоматизировать часто повторяющиеся действия.

9. Эстетичный и минимально необходимый дизайн – Окна не должны содержать не относящуюся к делу или редко используемую информацию. Каждый интерфейсный элемент, содержащий бесполезную информацию, играет роль *информационного шума* и отвлекает пользователя от действительно полезных интерфейсных элементов.

10. Помощь и документация – Несмотря на то, что в идеальном случае лучше, когда системой можно пользоваться без документации, она все равно необходима – как в виде системы помощи, так и, возможно, в виде печатного руководства. Информация в документации должна быть структурирована таким образом, чтобы пользователь мог легко найти нужный раздел, посвященный решаемой им задаче. Каждый такой ориентированный на конкретную задачу раздел должен помимо общей информации содержать пошаговые руководства по выполнению задачи и не должен быть слишком длинным.

Все эти эвристики могут использоваться при *тестировании удобства использования* пользовательского интерфейса. Достаточно очевидно, что при тестировании удобства слабо применимы способы *автоматизации тестирования* при помощи *сценариев* и подобные методы. Один из наиболее эффективных методов проверки интерфейса на удобство – использование формальной инспекции. Вопросы в бланке инспекции могут быть как общего характера (так, например, можно использовать в качестве вопросов перечисленные выше 10 эвристик), так и вполне конкретными. Например, в работе [10] приводится список контрольных вопросов, которые желательно проверять при *тестировании удобства использования* Веб-сайтов. С некоторыми изменениями эти вопросы применимы и для обычных оконных интерфейсов.

Тестирование безопасности

Отдельно следует отметить **тест на безопасность** [12]. Это очень важный тип тестов, так как от безопасности сервера зависит практически все – и сам бизнес, и доверие пользователей, и сохранность информации. Правда, в отличие от других тестов, *тестирование безопасности* следует проводить регулярно. Кроме того, тестированию подвергается не только сам конкретный сайт или веб-приложение, а весь сервер полностью – и веб-сервер, и операционная система, и все сетевые сервисы. Как и в случае других тестов, программа "прикидывается" реальным пользователем-взломщиком и пытается применить к серверу все известные ей методы атаки и проверяет все уязвимости. Результатом работы будет отчет о найденных уязвимостях и рекомендации по их устранению.

Ошибки, связанные с проверкой корректности ввода, часто довольно сложно обнаружить в большом объеме кода, взаимодействующего с пользователем. Это является основной причиной того, что разработчики используют методологию тестирования приложений на проникновение для их обнаружения. Веб-приложения, однако, не имеют иммунитета и к более традиционным способам атаки. Весьма распространены плохие механизмы аутентификации, *логические ошибки*, непреднамеренное *раскрытие информации*, а также такие традиционные ошибки для обычных приложений как переполнение буфера. Приступая к тестированию Веб -приложений, необходимо учитывать все перечисленное, и должен применяться методический процесс тестирования ввода/вывода по схеме "черного ящика" в сочетании (если возможно) с аудитом исходного кода [13].

Существуют различные инструменты позволяющие производить автоматическое *тестирование безопасности*, выполняя такие задачи как cross site scripting, *SQL injection*, включая переполнение буфера, подделка параметра, несанкционированный доступ, манипуляции с HTTP запросами и т.д. [14]

Нагрузочное тестирование

Следующим видом тестирования является **тест на устойчивость к большим нагрузкам** – Load-testing, *stress*-test или performance test [15]. Такой тест имитирует одновременную работу нескольких сотен или тысяч

посетителей (каждый из которых может "ходить" по сайту в соответствии со своим *сценарием*), проверяя, будет ли устойчивой работа сайта под большой нагрузкой. Кроме этого, можно имитировать кратковременные пики нагрузки, когда количество посетителей скачкообразно увеличивается – это очень актуально для новостных ресурсов и других сайтов с неравномерной аудиторией. В таком тесте проверяется не только и не столько сам сайт, сколько совместная слаженная работа всего комплекса – аппаратной части сервера, веб-сервера, программного ядра (engine) и других компонентов сайта.

Основными целями *нагрузочного тестирования* являются [15]:

- оценка производительности и работоспособности приложения на этапе разработки и передачи в эксплуатацию;
- оценка производительности и работоспособности приложения на этапе выпуска новых релизов, патч-сетов;
- оптимизация производительности приложения, включая настройки серверов и оптимизацию кода;
- подбор соответствующей для данного приложения аппаратной (программной платформы) и конфигурации сервера.

В *нагрузочное тестирование* входят следующие тесты:

Тестирование производительности (Performance testing)

Задачей *тестирования производительности* является определение масштабируемости приложения под нагрузкой, при этом происходит [15]:

- измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций;
- определение количества пользователей, одновременно работающих с приложением;
- определение границ приемлемой производительности при увеличении нагрузки (при увеличении интенсивности выполнения этих операций);
- исследование производительности на высоких, предельных, стрессовых нагрузках.

Стрессовое тестирование (Stress Testing)

Стрессовое тестирование позволяет проверить насколько приложение и система в целом работоспособны в условиях стресса и также оценить способность системы к регенерации, т.е. к возвращению к нормальному состоянию после прекращения воздействия стресса [16]. Стрессом в данном контексте может быть повышение интенсивности выполнения операций до очень высоких значений или аварийное изменение конфигурации сервера. Также одной из задач при *стрессовом тестировании* может быть оценка деградации производительности, таким образом, цели *стрессового тестирования* могут пересекаться с целями *тестирования производительности*.

Объемное тестирование (Volume Testing)

Задачей *объемного тестирования* является получение оценки производительности при увеличении объемов данных в базе данных приложения, при этом происходит [15]:

- измерение времени выполнения выбранных операций при определенных интенсивности выполнения этих операций;
- может производиться определение количества пользователей, одновременно работающих с приложением.

Тестирование стабильности или надежности (Stability / Reliability Testing)

Задачей *тестирования стабильности* (надежности) является проверка работоспособности приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки. Времена выполнения операций могут играть в данном виде тестирования второстепенную роль. При этом на первое место выходит отсутствие утечек памяти, перезапусков серверов под нагрузкой и другие аспекты, влияющие именно на стабильность работы.

Моделирование Транзакций (Transaction Simulation, TS)

Этот метод используется в большом числе различных продуктов, в частности, в пакетах AppManager и Vivinet Manager компании NetlQ.

Метод TS основан на использовании программных GUI-роботов (GUI - Graphical User Interface) [17]. GUI-робот – это специальная программа, которая "заставляет" реальное пользовательское приложение работать в автоматическом режиме, без участия самого пользователя (человека). Примерами средств, предназначенных для создания GUI-роботов, являются пакеты Rational *Visual Test* и Rational Robot компании Rational Software, а также пакет WinRunner компании Mercury Interactive.

Программный GUI-робот, как и реальный пользователь приложения, анализирует содержимое экрана и вводит данные с клавиатуры. Другими словами, он взаимодействует с пользовательским приложением через тот же интерфейс, что и человек. При этом GUI-робот работает по программе (скрипту), к коду которой всегда есть доступ (в отличие от кода самого пользовательского приложения). Поэтому, если в скрипт GUI-робота встроить специальные вызовы, например, перед началом и после окончания транзакции, то можно измерить время выполнения этой транзакции.

Основное достоинство метода TS заключается в том, что он позволяет измерять производительность работы приложения "с точки зрения пользователя" и, при этом, не требует доступа к коду пользовательского приложения [18]. Недостаток же метода в том, что он позволяет измерять время выполнения только рабочих транзакций и не может использоваться для измерения времени выполнения системных транзакций.

Метод "Анализ данных на стороне клиента" (Client Capture, CC)

Данный метод реализован, например, в программном пакете Vital Suite компании Lucent [17]. Метод основан на извлечении данных о работе приложения из операционной системы компьютера, где установлено пользовательское приложение. Для этого на компьютере пользователя устанавливается специальный Агент, который отслеживает взаимодействие приложения и ОС и, таким образом, получает информацию о доступности и времени реакции пользовательского приложения. Достоинство данного метода в том, что при его использовании нет необходимости модернизировать код приложения. Недостаток – дополнительные накладные расходы на

компьютере пользователя (клиента) и ограниченный набор поддерживаемых приложений.

Метод "Анализ Сетевого Трафика" (Network Sniffing, NS)

Примером такого устройства является NetScout WAN *Probes,* компании NetScout [17].Он основан на извлечении информации о производительности приложений из сетевого трафика. Для этого в сети устанавливаются специальные Зонды (как правило, аппаратные), которые в режиме реального времени захватывают сетевой трафик, анализируют его и "извлекают" данные о времени реакции приложений, доступности приложения и т.п. APM-средства на базе метода "Network *Sniffing*" выпускаются, в основном, производителями аппаратных анализаторов сетевых протоколов.

Он имеет следующие недостатки. Во-первых, для того, чтобы в режиме реального времени из сетевого трафика извлекать информацию о времени реакции приложений, компьютеры, где установлены Зонды, должны иметь очень высокую производительность. Второй недостаток метода NS заключается в том, что Зонд может "понимать" только заранее заданный набор пользовательских приложений, что также ограничивает использование данного метода.

Поскольку применение для целей тестирования большого числа клиентских машин с практической точки зрения нецелесообразно, встает задача обеспечения независимости пользовательских запросов, генерируемых одной или несколькими клиентскими машинами.

В настоящее время применяются два способа обеспечения независимости запросов: использование многопоточности и создание распределенных систем [19]. Многопоточность в той или иной форме применяется во всех рассмотренных средствах тестирования. Данный метод не позволяет воспроизводить и в явной форме задавать характер потока запросов (равномерный, пульсирующий и т.д.) и его статистические характеристики (средняя величина интервала времени между последовательными запросами, дисперсия и др.). Величина выделяемого нити кванта времени (а, следовательно, и величина интервала между запросами) зависит от производительности вычислительной системы и ее загруженности, а также от алгоритмов разделения времени, применяемых операционной системой. Непредсказуемые колебания интенсивности генерируемого потока запросов не позволяют достоверно оценить стабильность работы сервера. Увеличение числа нитей повышает уровень независимости запросов, однако конкуренция между нитями снижает ур овень нагрузки, что делает данный подход неприменимым для тестирования высокопроизводительных серверов.

Применение распределенных вычислений позволяет добиться большей реалистичности тестирования, так как статистические характеристики генерируемого потока запросов приближаются к характеристикам, наблюдаемым в условиях эксплуатации сервера. Кроме того, становится возможным увеличение интенсивности нагрузки. Однако применение данного подхода предъявляет значительно более высокие требования к архитектуре системы и планированию тестов.

Проверка HTML-кода

Еще одним типом тестирования является **проверка верности HTML-кода страниц** сайта [23]. Для такого рода тестирования написано множество утилит – от простых скриптов на Perl до мощных валидаторов, проверяющих весь сайт на соответствие стандартам (а некоторые валидаторы могут в автоматическом режиме исправлять найденные недочеты, например, пропущенные закрывающие теги и т.д.). Часто такие средства встраивают в Веб-редакторы, существуют браузеры с встроенными валидаторами.

Например, *FireBug* интегрируется с браузером Firefox, чтобы обогатить инструментарий разработчика [24]. Вы сможете редактировать, отлаживать и исследовать CSS, HTML и JavaScript вживую, на любой веб-странице. *Firebug* предоставляет возможность делать экспериментальные изменения в HTML и смотреть, как они тут же отображаются на странице, производить мониторинг сетевых запросов.

В комплекте с IE8 поставляется инструмент "средства разработчика", который, по сути, является аналогом *FireBug* [25].

Обзор автоматизации тестирования

В процессе создания информационных систем нередки ошибки и дефекты – это вполне ожидаемое и нормальное явление, а в условиях ограниченных временных ресурсов и высоких требований к качеству программных продуктов неизбежно возникает необходимость в организации эффективного контроля и управления всем процессом тестирования. Контроль качества ПО невозможен сегодня без автоматизации всех задач тестирования [20].

Ручное тестирование является затратным по времени, трудоемким и часто монотонным процессом. Оно приводит к возникновению проблем, особенно при ограниченных ресурсах и жестких сроках. Если вам нужно улучшить тестирование приложений для проверки корректности их работы, важно двигаться в сторону автоматизации всех ручных задач тестирования.

В современных условиях, когда циклы разработки сокращаются, автоматизированное тестирование позволяет как профессионалам, так и новичкам быстро достичь высококачественных результатов тестирования приложений. Инструментальные средства автоматизации записывают взаимодействие пользователей с приложением, а сформированные на этой основе *сценарии* используются для последующих тестов. В двух словах, *автоматизация тестирования* позволяет оптимизировать качество сложных приложений эффективным по стоимости способом за приемлемое время. Это помогает быстрее выпустить программное обеспечение более высокого качества.

Процесс *автоматизации тестирования* делится на три этапа [20]:

- **Запись**. *Сценарий* тестирования записывается "на лету" по мере работы пользователя с приложением. Можно также вставить точки верификации (verification points) для проверки ответа системы и сделать *сценарии* тестирования зависящими от данных, чтобы выполнять один и тот же *сценарий* с различными наборами входных данных.

• **Улучшение**. Добавление кода, выполняющего разнообразные функции. Типичные изменения *сценариев* тестирования – условное ветвление, рефакторинг и обработка исключительных ситуаций.

• **Воспроизведение**. Выполнение *сценариев*, эмулирующих действия, которые выполнял пользователь приложения при записи теста. Расхождения регистрируются, и тестировщик может сделать вывод о том, хорошо ли функционирует приложение или *регрессионное тестирование* выявило проблемы.

Для *автоматизации тестирования* существует большое количество приложений. Наиболее популярные из них:

• HP LoadRunner, HP QuickTest Professional, HP Quality Center;

• Segue SilkPerformer;

• IBM Rational FunctionalTester, IBM Rational PerformanceTester, IBM Rational TestStudio;

• AutomatedQA TestComplete.

HP LoadRunner – программный продукт для автоматизации *нагрузочного тестирования* широкого набора программных сред и протоколов [21]. Поддерживает SOA, работу с Web-сервисами, Ajax, *RDP*, SQL, продуктами Citrix, платформы Java, .Net, а также все основные ERP- и CRM-приложения от PeopleSoft, Oracle, SAP и Siebel. Пакет HP LoadRunner включает в себя более 60 мониторов сбора данных о тестируемой инфраструктуре и предоставляет детальную диагностику по работе приложений.

HP LoadRunner состоит из следующих приложений [22]:

• Virtual User Generator (VuGen) – служит для разработки нагрузочных скриптов;

• *Load Generator* – служит для генерации нагрузки (генерации виртуальных пользователей);

• Controller – служит для разработки и запуска *сценариев* нагрузки;

• Analysis – служит для анализа результатов *нагрузочного тестирования*.

Средства от IBM Rational [21]:

• IBM Rational Robot – универсальное средство *автоматизации тестирования* общего назначения для команд разработчиков, выполняющих *функциональное тестирование* клиент-серверных приложений. Дает возможность обнаруживать неполадки в ПО благодаря расширению *сценариев* тестирования средствами условной логики, позволяющей целиком охватить тестируемое приложение. Robot позволяет создавать *сценарии* тестирования с вызовом внешних библиотек DLL или исполняемых модулей.

• IBM Rational Performance *Tester* – инструмент нагрузочного и *стрессового тестирования*, с помощью которого можно выявлять проблемы системной производительности и их причины. Позволяет создавать тесты без написания кода и, не требуя навыков программирования. Обеспечивает гибкие возможности моделирования и эмуляции различных пользовательских нагрузок. Выполняет сбор и интеграцию данных о серверных ресурсах с данными о производительности приложений, получаемыми в режиме реального времени.

- IBM Rational Functional *Tester* – набор средств автоматизированного тестирования, позволяющих выполнять функциональное и *регрессионное тестирование*, *тестирование пользовательского интерфейса* и тестирование, управляемое данными. Инструмент применяет технологию ScriptAssure (бесшовная проверка достоверности динамических данных) и функции поиска соответствия по шаблону, позволяющие повысить устойчивость *сценариев* тестирования в условиях частых изменений пользовательских интерфейсов приложений. Тестировщики могут выбрать язык *сценариев* для разработки и настройки тестов: Java в среде Eclipse или Microsoft Visual Basic .Net в среде Visual Studio .Net.
- IBM Rational *Quality Manager* – решение для реализации процессов управления тестированием и качеством, поддерживает сотрудничество участников групп по разработке программных продуктов, предоставляя им возможность обмениваться информацией, применять средства автоматизации для сокращения графиков выполнения проектов, а также составлять отчеты по проектным показателям для принятия обоснованных решений. Rational *Quality Manager* может быть дополнен средством управления ресурсами тестирования Rational Test Lab, обеспечивающим учет ресурсов *тестирования (серверов*), их бронирование, автоматизацию развертывания тестовой среды на сервере и запуск скриптов тестирования, а также отчетность по использованию ресурсов тестирования.
- Rational *Quality Manager* и Rational Test Lab созданы на базе открытой платформы Jazz, которая предоставляет стандартные интерфейсы и удобные возможности для интеграции с решениями партнеров и других производителей.

Ключевые термины

Функциональное тестирование, Тестирование пользовательского интерфейса, Ручное тестирование, Сценарии, Тестирования удобства использования, Проверка ссылок, Тестирование безопасности, Нагрузочное тестирование, Тестирование производительности, Стрессовое тестирование, Объемное тестирование, Тестирование стабильности, Моделирование Транзакций, Метод "Анализ данных на стороне клиента", Метод "Анализ Сетевого Трафика", Автоматизации тестирования, Проверка HTML-кода.

Тестирование параллельных программ

Тестирование приложений с параллельной обработкой - задача непростая. Ошибки распараллеливания сложно выявить из-за недетерминированности поведения параллельных приложений. Даже если ошибка обнаружена, ее часто сложно воспроизвести повторно. Кроме того, после модификации кода, не так просто убедиться, что ошибка действительно устранена, а не замаскирована. Все это можно назвать и по-другому, а именно, что ошибки в параллельной программе являются классическими "гейзенбагами".

Гейзенбаг (англ. Heisenbug) - термин, используемый в программировании для описания программной ошибки, которая исчезает или меняет свои свойства при попытке её обнаружения [2]. Данное название является игрой слов и происходит от физического термина «Принцип неопределённости Гейзенберга», который на бытовом уровне понимается как изменение наблюдаемого объекта в результате самого факта наблюдения, происходящее в квантовой механике. В русской терминологии более часто используется термин «плавающая ошибка». Примером могут являться ошибки, которые проявляются в окончательном варианте программы (“релизе”), однако не видны в режиме отладки, или ошибки синхронизации в многопоточном приложении.

Таким образом, задача параллельного тестирования во многом сводится к проблеме создания инструментов диагностики, минимально влияющих на поведение программы или создающих необходимые условия для ее проявления. Поэтому посмотрим на классические методологии тестирования под новым углом.

Введение

Говоря о разработке программного обеспечения, выделяют этап тестирования и отладки приложения. Понятия тестирование и отладка слабо разделяют между собой. Это связано с тем, что после нахождения ошибки при тестировании ее устранение часто связано с отладкой программы. А отладку приложения в свою очередь можно называть тестированием методом белого

ящика. Иногда под этапом отладки понимают одновременно и поиск и устранение ошибок.

Мы все-таки разделим эти два понятия, и сосредоточимся в этой статье на тестировании параллельного программного обеспечения.

Тестирование программного обеспечения - процесс выявления ошибок в программном обеспечении с использованием различных инструментов и анализ работоспособности программы конечными пользователями [1].

Отладка - процесс обнаружения программистом в коде ошибок, вывяленных в ходе тестирования программного обеспечения с целью их дальнейшего устранения. Отладка обычно подразумевает использование специализированных инструментов для отслеживания состояния программы в ходе ее исполнения.

Отладка параллельных программ - дело неблагодарное, требующее аккуратности и специализированных инструментов. Отладке параллельных программ посвящено много статей и следует посветить еще больше, поскольку данная тематика весьма актуальна в связи с активным развитием многоядерных систем и новых технологий создания параллельных программ. В области инструментария также существуют пробелы. Но прежде, чем заняться отладкой - ошибку нужно найти. При этом некоторые методы отладки не только обнаруживают ошибку, но и сразу локализуют место ее нахождения. Поэтому займемся тестированием.

1. Сложности тестирования параллельных программ

Тестирование приложений с параллельной обработкой - задача непростая. Ошибки распараллеливания сложно выявить из-за недетерминированности поведения параллельных приложений. Даже если ошибка обнаружена, ее часто сложно воспроизвести повторно. Кроме того, после модификации кода, не так просто убедиться, что ошибка действительно устранена, а не замаскирована. Все это можно назвать и по-другому, а именно, что ошибки в параллельной программе являются классическими "гейзенбагами".

Гейзенбаг (англ. Heisenbug) - термин, используемый в программировании для описания программной ошибки, которая исчезает или меняет свои свойства при попытке её обнаружения [2]. Данное название является игрой слов и происходит от физического термина «Принцип неопределённости Гейзенберга», который на бытовом уровне понимается как изменение наблюдаемого объекта в результате самого факта наблюдения, происходящее в квантовой механике. В русской терминологии более часто используется термин «плавающая ошибка». Примером могут являться ошибки, которые проявляются в окончательном варианте программы (“релизе”), однако не видны в режиме отладки, или ошибки синхронизации в многопоточном приложении.

Таким образом, задача параллельного тестирования во многом сводится к проблеме создания инструментов диагностики, минимально влияющих на поведение программы или создающих необходимые условия для ее

проявления. Поэтому посмотрим на классические методологии тестирования под новым углом.

2. Методики поиска ошибок в параллельных программах

Методики поиски ошибок в параллельных приложениях, как и в последовательных можно разделить на динамический анализ, статический анализ, проверку на основе моделей и доказательство корректности программы [3].

Формальное доказательство корректности программы является крайне сложной процедурой, особенно когда речь заходит о параллельном программировании и практически не применяется промышленной разработки программного обеспечения.

Динамический анализ подразумевает под собой необходимость запуска приложения и выполнения различный последовательностей действий, целью которых ставится выявление некоренного поведения программы. Последовательность действий может задаваться как человеком при ручном тестировании, так и с использованием различных инструментов, реализующих нагрузочное тестирование или, например, проверку целостности данных. В случае параллельных программ наибольший интерес представляют инструменты подобные Intel Thread Checker, которые оснащают исходный код приложения средствами мониторинга и протоколирования, которые позволяют выявлять взаимоблокировки (как явные, так и потенциальные), зависания, гонки и так далее [4].

Статический анализ работает только с программным кодом приложения, не требуя его запуска. В качестве преимущества можно отметить детальность и полноту охвата анализируемого кода. Следует отметить, что в сфере параллельного программирования методология статического анализа используется достаточно редко и представлены малым количеством инструментов, которые являются скорее исследовательскими, чем коммерческими продуктами.

Проверка на основе модулей представляет собой автоматическую генерацию тестов по заданным правилам. Проверка на основе моделей позволяет формально обосновать отсутствие дефектов в тестируемой части кода, на основе заданной разработчиком правил преобразования данных. В качестве примера можно назвать инструмент KISS, разработанный в Microsoft Research для параллельных программ на C.

Все перечисленные методики имеют свои недостатки, что не позволяет положиться при разработке параллельных программ только на одну из них.

Динамический анализ требует запуска программ, чувствителен к среде исполнения, существенно замедляет скорость выполнения приложения. Достаточно трудно осуществить покрытие тестами всего параллельного кода. Часто зафиксировать состояние гонки (race conditions) удается, только если оно было в данном сеансе работы программы. То есть, если средство динамического анализа сообщает, что ошибок нет, вы все равно не можете быть уверены в этом.

В случае параллельных приложений статический анализ крайне сложен и часто невозможно предсказать поведение программы, так как неизвестен допустимый набор входных значений для различных функций и способ их вызова. Эти значения можно прогнозировать на основе остального кода, но крайне ограниченно, так как возникает огромное пространство возможных состояний и объем проверяемой информации (вариантов) увеличивается до недопустимых значений. Также средства статического анализа часто дают большое количество ложных сообщений о потенциальных ошибках и требуют немалых усилий для их минимизации.

На практике проверка на основе моделей действенна лишь для небольших базовых блоков приложения. В большинстве случаев очень сложно автоматически построить модель на основе кода, а создание моделей вручную - крайне трудоемкое и ресурсоемкое занятие. Фактически необходимо написать те же алгоритмы преобразования данных, но в ином представлении. Как и в случае статического анализа возникает проблема быстрого расширении пространства состояний. Расширение пространства состояний можно в частично контролировать, применяя методы редукции со сложной эвристической логикой, но это приводит к тому, что некоторые дефекты будут пропущены. В качестве примера инструмента проверки для тестирования проектов параллельных приложений на основе моделей можно назвать Zing.

Как видно, использование какого-то одного подхода к тестированию параллельных приложений неэффективно и хорошим решением является использовать несколько методик для тестирования одного и погоже программного продукта.

3. Новые технологии - новые инструменты

Поскольку при решении задачи увеличения производительности выбор был сделан в пользу многоядерных процессоров, то это повлекло и новый виток в развитии инструментальных средств разработки. Для многоядерных систем с общей памятью более удобным оказывается использование таких технологий программирования, как OpenMP, вместо более привычных MPI или стандартных средств распараллеливания, предоставляемых операционными системами (fork, beginthread).

Для новых технологий нужны и новые инструменты их поддержки. Сейчас следует активно следить за новыми системами поддержки параллельного программирования, появляющимися на рынке программного обеспечения. Они могут существенно облегчить ваш труд и быстрее адаптировать ваши приложения для эффективного использования параллельной среды. Одним из таких инструментов является разработанный в компании «Системы программной верификации» статический анализатор кода VivaMP.

Как уже было сказано, статический анализ параллельных программ сложен и малоэффективен, так как необходимо хранить крайне много информации о возможных состояниях программы. Это совершенно справедливо при использовании таких технологий параллельного

программирования как MPI или распараллеливания средствами операционной системы.

С технологией OpenMP ситуация обстоит лучше и часто можно реализовать эффективный статический анализ, обладающий хорошими показателями. Это связано с тем, что технология OpenMP ориентирована на распараллеливание изолированных участков кода. OpenMP как бы позволяет делать программу параллельной "по кусочкам", с помощью расстановки специальных директив в наиболее критичным по быстродействию частям кода. В результате параллельный код оказывается сгруппирован и не зависит от других частей приложения, что позволяет провести его качественный анализ.

До недавнего времени направление статического анализа OpenMP программ практически было не освоено. В качестве примера можно привести, пожалуй, только достаточно качественную диагностику, выполняемую компилятором Sun Studio. Статический анализатор VivaMP заполнил это нишу. Это специализированный инструмент для поиска ошибок в параллельных программах, разработанных с использованием технологии OpenMP на языке Си и Си++ [5].

Данный анализатор как обнаруживает явные ошибки, так и предупреждает о потенциально опасном коде. В качестве диагностики ошибок можно привести пример обнаружения использование одной переменной для записи из параллельных потоков без необходимой синхронизации:

```
int sum1 = 0;
int sum2 = 0;
#pragma omp parallel for
  for (size_t i = 0; i != n; ++i)
  {
    sum1 += array[i]; // V1205
    #pragma atomic
    sum2 += array[i]; //Fine
  }
```

А в качестве примера диагностики потенциально опасно опасного кода можно привести пример использования flush для указателя. Несложно ошибиться, забыв, что операция flush будет применена именно к указателю, а не к данным, на которые он ссылается. В результате приведенный код может быть как корректным, так и некорректным:

```
int *ptr;
...
#pragma omp flush(ptr) // V1202
int value = *ptr;
```

В следующей версии анализатора VivaMP также будут реализованы некоторые проверки связанные с выявлением неэффективного параллельного кода. Например, будут критические секции там, где было бы достаточно использовать более быструю директиву atomic:

```
#pragma omp critical
{
  a++;  //Slow
}
#pragma omp atomic
a++; //Good
```

Заключение

Различные формы распараллеливания существуют в мире программного обеспечения уже давно. Однако для создания массовых коммерческих приложений, использующих возможности многоядерных процессоров в полной мере, требуются иные методы разработки, отличные от применяемых при создании последовательных приложений. Хочется надеяться, что данная статья прольет свет на те сложности, с которыми связана разработка параллельных приложений, и программист со всей серьезностью отнесется к выбору наиболее подходящих средств разработки и тестирования таких программ.

Различные средства автоматизации тестирования.

Тестирование программного обеспечения

Раздел 1. Основные понятия тестирования Раздел 2. Критерии выбора тестов Раздел 3. Разновидности тестирования

Раздел **4.** Особенности индустриального тестирования

Раздел 5. Регрессионное тестирование

Раздел **4.** Особенности индустриального тестирования

Автоматизация тестирования

Автоматизация тестирования – использование специального ПО (помимо тестового ПО), для выполнения и контроля выполнения тестов, а также сравнения ожидаемого и фактического результатов работы ПО. Автоматизация тестирования ПО позволяет осуществлять выполнение часто повторяющихся рутинных и необходимых для максимизации тестового покрытия задач.

Основные виды автоматизированного тестирования**:**

-автоматизированное тестирование кода, т.е. тестирование на уровне программных модулей, классов и библиотек;

-автоматизированное тестирование графического пользовательского интерфейса, позволяющее генерировать пользовательские события, т.е. нажатия клавиш, события от манипулятора графической информации ("мыши"), отслеживающие реакцию ПО и соответствие этой реакции спецификации;

-автоматизированное тестирование программного интерфейса ПО, предназначенного для взаимодействия с другим ПО.

Автоматизированные тесты ПО – это ПО, предназначенное для тестирования другого ПО.

Автоматизация тестирования

Структура инструментальной системы автоматизации тестирования

Автоматизация тестирования

В ходе выполнения инструментальной системы автоматизации тестирования на каждом из соответствующих этапов создается и сохраняется следующая информация:

1)набор тестов, достаточный для покрытия тестируемого приложения в соответствии с выбранным критерием тестирования, как результат ручной или автоматической разработки (генерации) тестовых наборов и драйвер/монитор пропуска тестового набора;

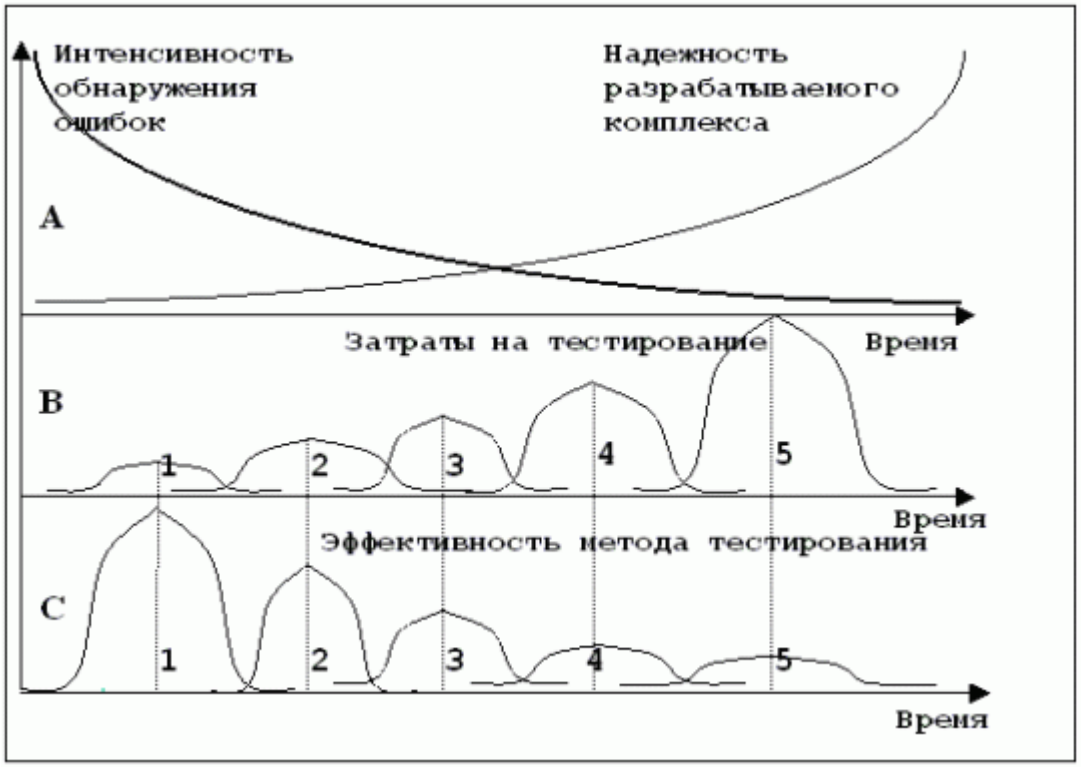
2)результаты прогона тестового набора, зафиксированные в Log-файле.

3)статистика тестового цикла, содержащая:

- результаты пропуска каждого теста из тестового набора и их сравнения с эталонными величинами;

-факты, послужившие основанием для принятия решения о продолжении или окончании тестирования;

-критерий покрытия и степень его удовлетворения, достигнутая в цикле тестирования.



Издержки тестирования

Чем больше трудозатрат вкладывается в процесс тестирования, тем меньше ошибок в продукте остается незамеченными (A).

Со временем, по мере обнаружения более сложных ошибок и дефектов (B), эффективность низкозатратных методов падает вместе с количеством обнаруживаемых ошибок (C).

Таком образом, соответствующие методы тестирования имеют свою нишу, где они хорошо обнаруживают ошибки, тогда как вне ниши их эффективность падает. Поэтому необходимо совмещать различные методы и стратегии отладки и тестирования с целью обеспечения запланированного качества программного продукта при ограниченных затратах, что может быть

достигнуто использованием процесса управления качеством программного продукта.

На практике используются следующие методы тестирования и отладки, упорядоченные по связанным с их применением затратам:

-статические методы тестирования

-модульное тестирование

-интеграционное тестирование

-системное тестирование

-тестирование реального окружения и реального времени

Фазы процесса тестирования

Впроцессе тестирования целесообразно выделить следующие фазы:

-определение целей (требований к тестированию): какие части системы будут тестироваться, какие аспекты их работы будут выбраны для проверки, каково желаемое качество и т.п.

-планирование: создание графика (расписания) разработки тестов для каждой тестируемой подсистемы; оценка необходимых человеческих, программных И аппаратных ресурсов; разработка расписания тестовых циклов.

-разработка тестов, то есть тестового кода для тестируемой системы, если необходимо - кода системы автоматизации тестирования и тестовых процедур (выполняемых вручную)

-выполнение тестов - реализация тестовых циклов.

-анализ результатов.

После анализа результатов возможно повторение процесса тестирования,

начиная с планирования, разработки тестов или даже с уточнения и/или переопределения целей.

естовый цикл

Тестовый цикл – это цикл исполнения тестов, включающий выполнение тестов и анализ результатов тестового процесса.

Тестовый цикл включает следующую последовательность действий:

1)Проверка готовности системы и тестов к проведению тестового цикла

2)Подготовка тестовой машины в соответствии с требованиями, определенными на этапе планирования (например, полная очистка

иперeустановка системного программного обеспечения). Конфигурация тестовой машины, так же, как и срез системы, должны быть однозначно воспроизводимыми.

3)Воспроизведение среза системы.

4)Прогон тестов в соответствии с задокументированными процедурами.

5)Сохранение тестовых протоколов (test log)

6)Анализ протоколов тестирования и принятие решения о том прошел или не прошел каждый из тестов (Pass/Fail)

7)Анализ и документирование результатов цикла.

Планирование тестирования

Тестовый план - это документ или набор документов, содержащий:

1)Тестовые ресурсы.

2)Перечень функций и подсистем, подлежащих тестированию.

3)Тестовую стратегию, включающую:

-анализ функций и подсистем с целью определения наиболее слабых мест, то есть областей функциональности тестируемой системы, где появление дефектов наиболее вероятно

-определение стратегии выбора входных данных для тестирования

-определение потребности в автоматизированной системе тестирования идизайн такой системы

4)Расписание тестовых циклов

5)Фиксацию тестовой конфигурации: состава и конкретных параметров аппаратуры и программного окружения

6)Определение списка тестовых метрик, которые на тестовом цикле необходимо собрать и проанализировать, например, метрик, оценивающих степень покрытия тестами набора требований, степень покрытия кода тестируемой системы, количество и уровень серьезности дефектов, объем тестового кода и другие характеристики.

Типы тестирования

В тестовом плане определяются и документируются различные типы тестов. Типы тестов могут быть классифицированы по двум категориям:

-по тому, что подвергается тестированию (по виду подсистемы или продукта)

-по способу выбора входных данных

Типы тестирования по виду подсистемы или продукта**:**

1)Тестирование основной функциональности, когда тестированию подвергается собственно система, являющаяся основным выпускаемым продуктом

2)Тестирование инсталляции включает тестирование сценариев первичной инсталляции системы, сценариев повторной инсталляции (поверх уже существующей копии), тестирование деинсталляции, тестирование инсталляции в условиях наличия ошибок в инсталлируемом пакете, в окружении или в сценарии и т.п.

3)Тестирование пользовательской документации включает проверку полноты и понятности описания правил и особенностей использования продукта, наличие описания всех сценариев и функциональности, синтаксис и грамматику языка, работоспособность примеров и т.п.

Типы тестирования по способу выбора входных данных**:**

1)Функциональное тестирование, при котором проверяется:

-покрытие функциональных требований.

-покрытие сценариев использования.

2)стрессовое тестирование, при котором проверяются экстремальные режимы использования продукта.

3)Тестирование граничных значений.

4)Тестирование производительности.

5)Тестирование на соответствие стандартам.

6)Тестирование совместимости с другими программно-аппаратными комплексами.

7)Тестирование работы с окружением.

8)Тестирование работы на конкретной платформе

На практике используются и комбинируются различные типы тестов для обеспечения заданного качества продукта.

Подходы к разработке тестов

-подходы, основанные на выборе тестовых данных (тестирование спецификаций, тестирование сценариев)

-и подходы, основанные на реализации тестового кода (ручная разработка тестов,

генерация тестов)

Тестирование спецификации

При разработке тестов, основанных на функциональной спецификации продукт требования к продукту являются основным источником, определяющим, какие тест будут разработаны. Для каждого требования пишется один или более тестов, которые в совокупности должны проверить выполнение данного требования в продукте.

Тестирование сценариев

осуществляется следующим образом:

-определяется модель использования, включающая операционное окружение продукта и "актеров". Актером может быть пользователь, другой продукт, аппаратная часть и др.

-разрабатываются сценарии использования продукта, которые могут быть строго определенным, параметризованным или разрешать некоторую степень неопределенности.

-разрабатывается набор тестов, покрывающих заданные сценарии, с учетом степени неопределенности, заложенной в сценарии, и, при этом, каждый тест покрывает один сценарий, несколько сценариев, или, наоборот, часть сценария.

Ручная разработка тестов

Наиболее распространенным способом разработки тестов является создание тестового кода вручную. Это наиболее гибкий способ разработки тестов, однако характерная для него производительность труда инженеров-тестировщиков в создании тестового кода не намного выше скорости создания кода продукта, а объемы тестового кода на практике зачастую превышают объем кода продукта в 5 и более раз.

Генерация тестов

Учитывая существенные трудозатраты на ручную разработку тестов целесообразно использование автоматизированных способов получения тестового кода, таких как использование специальных тестовых языков (скриптов)

игенерации тестов. В настоящее время некоторые языки спецификаций, используемые для описания алгоритмов тестирования, могут быть использованы

идля генерации тестового кода, одним из примеров является язык MSC.

Использование подхода генерации тестового кода позволяет значительно поднять производительность тестирования, а также преобразовать формализацию (кодировку) сценариев в достаточно интеллектуальную деятельность.

Выполнение тестов**:**

-подход ручного тестирования

-подход автоматического исполнения (прогон) тестов.

Ручное тестирование

Ручное тестирование заключается в выполнении задокументированной процедуры, где описана методика выполнения тестов, задающая порядок тестов и для каждого теста - список значений параметров, который подается на вход, и список результатов, ожидаемых на выходе. Поскольку процедура предназначена для выполнения человеком, в ее описании для краткости могут использоваться некоторые значения по умолчанию, ориентированные на здравый смысл, или ссылки на информацию, хранящуюся в другом документе.

Пример фрагмента процедуры ручного тестирования:

1)Подать на вход три разных целых числа.

2)Запустить тестовое исполнение.

3)Проверить, соответствует ли полученный результат таблице [ссылка на документ1] с учетом поправок [ссылка на документ2].

4)Убедиться в понятности и корректности выдаваемой сопроводительной информации.

Автоматизированное тестирование

Автоматизация рассмотренного ранее примера приводит к созданию скрипта, задающего тестируемому продукту три конкретных числа и перенаправляющего вывод продукта в файл с целью его анализа, а также содержащего конкретное значение желаемого результата, с которым сверяется получаемое при прогоне теста значение.

Пример фрагмента скрипта автоматизированного тестирования:

1)Выдать на консоль имя или номер теста и время его начала.

2)Вызвать продукт с фиксированными параметрами.

3)Перенаправить вывод продукта в файл.

4)Проверить равенство возвращенного продуктом значения ожидаемому (эталонному) результату, зафиксированному в тесте.

5)Проверить вывод продукта, сохраненный в файле (п.3), на равенство заранее приготовленному эталону.

6)Выдать на консоль результаты теста в виде вердикта PASS/FAIL и в случае FAIL - краткого пояснения, какая именно проверка не прошла.

7)Выдать на консоль время окончания теста.

Сравнение ручного и автоматизированного подхода

	Ручное	Автоматизированное
Задание	Гибкость в задании данных. Позволяет	Входные значения строго

входных	использовать разные значения на разных циклах	заданы
значений	прогона тестов, расширяя покрытие	
Проверка результата	Гибкая, позволяет тестировщику оценивать	Строгая. Нечетко
	нечетко сформулированные критерии	сформулированные критерии
		могут быть проверены только
		путем сравнения с эталоном
Повторяемость	Низкая. Человеческий фактор и нечеткое	Высокая
	определение данных приводят к	
	неповторяемости тестирования	
Надежность	Низкая. Длительные тестовые циклы приводят к	Высокая, не зависит от длины
	снижению внимания тестировщика	тестового цикла
Чувствительность	Зависит от детальности описания процедуры.	Высокая. Незначительные
к незначительным	Обычно тестировщик в состоянии выполнить	изменения в интерфейсе часто
изменениям в	тест, если внешний вид продукта и текст	ведут к коррекции эталонов
продукте	сообщений несколько изменились	
Скорость выполнения	Низкая	Высокая
тестового набора		
Возможность	Отсутствует. Низкая скорость выполнения	Поддерживается
генерации тестов	обычно не позволяет исполнить	
	сгенерированный набор тестов	

Документирование тестовых процедур

Тестовые процедуры - это формальный документ, содержащий описание необходимых шагов для выполнения тестового набора.

Вслучае описания ручных тестов тестовые процедуры должны содержать полное описание всех шагов и проверок, позволяющих протестировать продукт

ивынести вердикт PASS/FAIL.

Вслучае описания автоматизированных тестов тестовые процедуры должны содержать достаточную информацию для запуска тестов и анализа результатов.

Описание тестов разрабатывается для облегчения анализа и поддержки тестового набора, может быть реализовано в произвольной форме, но при этом должно выполнять следующие задачи:

-анализировать степень покрытия продукта тестами на основании описания тестового набора.

-для любой функции тестируемого продукта найти тесты, в которых функция используется.

-для любого теста определить все функции и их сочетания, которые данный тест использует (затрагивает)

-вывить структуру и взаимосвязи тестовых файлов

-выявить принцип построения системы автоматизации тестирования

Документирование дефекта

Каждый дефект, обнаруженный в процессе тестирования, должен быть задокументирован и отслежен. При обнаружении нового дефекта его заносят в базу дефектов. Для этого лучше всего использовать специализированные базы, поддерживающие хранение и отслеживание дефектов, например, вида DDTS.

При занесении нового дефекта рекомендуется указывать:

-наименование подсистемы, в которой обнаружен дефект.

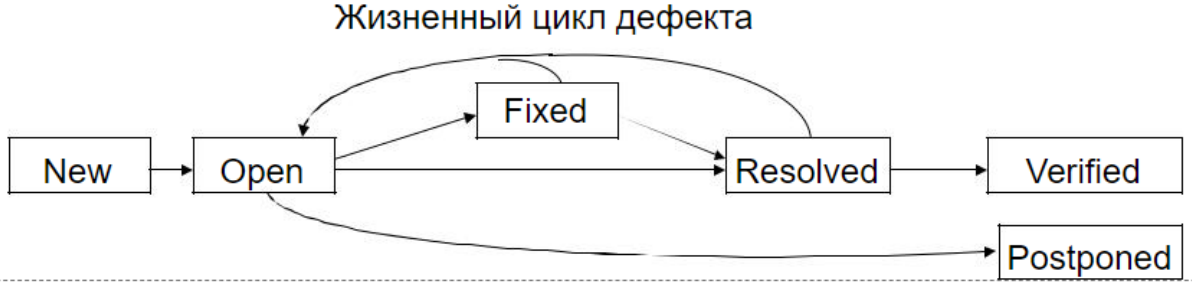
-версия продукта (номер build), на котором дефект был найден.

-описание дефекта.

-описание процедуры (шагов, необходимых для воспроизведения дефекта).

-номер теста, на котором дефект был обнаружен.

-уровень дефекта, то есть степень его серьезности с точки зрения критериев качества продукта или заказчика.



Тестовый отчет

Тестовый отчет обновляется после каждого цикла тестирования и должен содержать следующую информацию для каждого цикла:

-перечень функциональности в соответствии с пунктами требований, запланированный для тестирования на данном цикле, и реальные данные по нему

-количество выполненных тестов – запланированное и реально исполненное

-время, затраченное на тестирование каждой функции, и общее время тестирования.

-количество найденных дефектов.

-количество повторно открытых дефектов.

-отклонения от запланированной последовательности действий, если таковые имели место.

-выводы о необходимых корректировках в системе тестов, которые должны быть сделаны до следующего тестового цикла

Оценка качества тестов

Тестовые метрики

Набор тестовых метрик, который помогает определить эффективность тестирования и текущее состояние продукта:

-покрытие функциональных требований.

-покрытие кода продукта (для модульного уровня тестирования).

-покрытие множества сценариев.

-количество или плотность найденных дефектов. Текущее количество дефектов сравнивается со средним для данного типа продуктов с целью установить, находится ли оно в пределах допустимого статистического отклонения.

-соотношение количества найденных дефектов с количеством тестов на данную функцию продукта. Сильное расхождение этих двух величин говорит либо о неэффективности тестов (когда большое количество тестов находит мало дефектов), либо о плохом качестве данного участка кода (когда найдено большое количество дефектов на не очень большом количестве тестов)

-количество найденных дефектов, соотнесенное по времени, или скорость поиска дефектов. Если производная такой функции близка к нулю, то продукт обладает качеством, достаточным для окончания тестирования и поставки заказчику.

Оценка качества тестов

Обзоры тестов и стратегии**:**

Тестовый код и стратегия тестирования, зафиксированные в виде документов, улучшаются, если подвергаются коллективному обсуждению. Такие обсуждения называются обзорами и для них должна существовать

процедура проведения и оценки их результатов. Обзоры наряду с тестированием образуют набор методов борьбы с ошибками с целью повышения качества ПО.

Цели обзора тестовой стратегии**:**

-установить достаточность проверок, обеспечиваемых тестированием.

-проанализировать оптимальность покрытия или адекватность распределения количества планируемых тестов по функциональности продукта

-проанализировать оптимальность подхода к разработке кода, генерации кода, автоматизации тестирования.

Цели обзора тестового кода**:**

-установить соответствие тестового набора тестовой стратегии.

-проверить правильность кодирования тестов.

-оценить достигнутую степень качества кода, исходя из требований по стандартам, простоте поддержки, наличию комментариев и т.п.

-если необходимо, проанализировать оптимальность тестового кода с целью удовлетворения требований к быстродействию и объему.

LINT(1)
НАЗВАНИЕ

lint - верификатор C-программ
СИНТАКСИС
lint [-a] [-b] [-h] [-u] [-v] [-x] [-l библ] [-n] [-p] [-c] [-o библ] файл ...
ОПИСАНИЕ

Команда lint пытается обнаружить в заданных файлах, содержащих C-программы, конструкции, которые, возможно, являются ошибочными, немобильными или излишними. Более строго, чем при компиляции, выполняется проверка соответствия типов. Среди обнаруживаемых дефектов - недостижимые операторы; циклы, в которые входят не с начала; описанные, но не используемые автоматические переменные; логические выражения с константными значениями. Кроме того, проверяется использование функций и обнаруживаются функции, возвращающие значения в одних местах, но не возвращающие в других; функции, вызываемые с различным числом аргументов или с аргументами разных типов; функции, значения которых не используются, и функции, значения которых не возвращаются, но используются.

Файлы-аргументы, имена которых оканчиваются на .c, считаются исходными C-файлами. Аргументы, имена которых оканчиваются на .ln, считаются результатом предыдущих вызовов lint с использованием опций -c или -o. Файлы .ln аналогичны об ектным (.o) файлам, которые создаются командой , если в качестве входных файлов заданы .c файлы. Файлы с другими расширениями игнорируются с выдачей предупреждения.

Программа lint обрабатывает все .c, .ln и llib-lбибл.ln (заданные указанием -l библ) файлы в том порядке, в котором они перечислены в командной строке. По умолчанию lint подсоединяет к концу списка файлов свою стандартную библиотеку C-программ llib-lc.ln. Однако, если используется опция -p, вместо стандартной подсоединяется мобильная C-библиотека программы lint llib-port.ln. Если опция -c не указана, второй проход lint проверяет этот список файлов на взаимную совместимость. В случае задания опции -c файлы .ln и llib-lбибл.ln игнорируются.

Можно указывать произвольное число опций и задавать их в командной строке в любом порядке вперемежку с именами файлов. Следующие опции используются для того, чтобы подавить выдачу некоторых сообщений.

-a -b -h -u -v -x

Не выдавать сообщения о присваиваниях long-значений переменным, не специфицированным как long.

Не выдавать сообщения о недостижимых операторах break. [Программы, сгенерированные при помощи или обычно содержат большое число таких операторов.]
Не применять набор эвристических тестов, предназначенных для того, чтобы попытаться "поймать" ошибки, улучшить стиль и сделать программу компактнее.
Не выдавать сообщения о функциях и внешних переменных, используемых, но не определенных или определенных, но не используемых. (Эта опция полезна, когда при обращении к lint задается подмножество файлов, составляющих одну большую программу.)
Не выдавать сообщения о неиспользуемых параметрах функций.
Не сообщать о внешних переменных, которые нигде не используются.

Потомство

Многие из проверок, которые выполнял *lint* , теперь, учитывая достижения в генерации собственного кода , встроены в компиляторы (иногда с включенной опцией, например -*Wall* для GCC). Эти компиляторы действительно должны для оптимизации исполняемого файла выполнять статический анализ гораздо более продвинутый, чем их предок UNIX.

Несколько *lint* -проверок теперь не нужны, так как стандартизация разных языков программирования значительно уменьшила проблемы с переносимостью. Использование современных платформ разработки и контекстных текстовых редакторов с синтаксическим анализатором и автоматическим отступом также позволяет создавать более безопасный и приятный для чтения исходный код с самого начала.

С появлением и распространением C++ были предприняты попытки адаптировать lint к особенностям этого нового языка; но его изолированное положение обрекло его: теперь на рынке можно найти целый ряд чрезвычайно сложных инструментов для статического анализа исходного кода. Тем не менее, *lint* остается популярным для общих проектов благодаря своему небольшому размеру, стабильности (отсутствие несвоевременных изменений версии), возможностям настройки и чрезвычайной переносимости. Благодаря lint исходные файлы от разных разработчиков могут быть согласованы для соблюдения определенных формальных правил единства, необходимых для автоматического обновления программного обеспечения и его документации.

Верификация модели

Это проверка на соответствие поведения модели замыслу исследователя и моделирования. Т.е. процедуры верификации проводят, чтобы убедиться, что модель ведет себя так, как было задумано. Для этого реализуют формальные и неформальные исследования имитационной модели.

Верификация имитационной модели предполагает доказательство возможности использования создаваемой программной модели в качестве машинного аналога концептуальной модели на основе обеспечения максимального сходства с последней. Цель процедуры верификации — определить уровень, на котором это сходство может быть успешно достигнуто.

Валидация и верификация имитационной модели связаны с обоснованием *внутренней структуры модели,* в ходе этих процедур проводятся испытания внутренней структуры и принятых гипотез, исследуется внутренняя состоятельность модели.

Валидация данных

Валидация данных (data validity) направлена на доказательство того, что все используемые в модели данные, в том числе входные, обладают удовлетворительной точностью и не противоречат исследуемой системе, а значения параметров точно определены и корректно используются.

Эти проверки связаны с проблемным анализом, т.е. анализом и интерпретацией полученных в результате эксперимента данных. *Проблемный анализ* — это формулировка статистически значимых выводов на основе данных, полученных в результате эксперимента на имитационной модели. Проверяется правильность интерпретации полученных с помощью модели данных, оценивается насколько могут быть справедливы статистические выводы, полученные в результате имитационного эксперимента. С этой целью проводят **исследование свойств имитационной модели:** оценивается *точность, устойчивость, чувствительность результатов моделирования.* Эти проверки связаны с выходами модели, сама имитационная модель рассматривается как черный ящик.

Таким образом, на этапе испытания и исследования разработанной имитационной модели организуется комплексное **тестирование модели (testing) - планируемый итеративный процесс, направленный главным образом на поддержку процедур верификации и валидации имитационных моделей и данных.**

Некоторые полезные процедуры тестирования рассмотрим ниже. Более широкое изложение методов тестирования имитационных моделей можно найти в специальной литературе [20, 33, 56].

6.2 Проверка адекватности модели

При моделировании исследователя прежде всего интересует, насколько хорошо модель представляет моделируемую систему (объект моделирования). Модель, поведение которой слишком отличается от поведения моделируемой системы, практически бесполезна.

Различают *модели существующих и проектируемых систем.*

Если реальная система (или ее прототип) существует, дело обстоит достаточно просто. Поэтому для моделей существующих систем исследователь должен выполнить проверку адекватности имитационной модели объекту моделирования, т.е. проверить соответствие между поведением реальной системы и поведением модели.

На *реальную систему* воздействуют *переменные* G*, которые можно измерять, но нельзя управлять, *параметры* X*, которые исследователь может изменять в ходе натурных экспериментов. На выходе системы возможно измерение *выходных характеристикY*.*

При этом существует некоторая неизвестная исследователю зависимость между ними $Y^*=f^*(X^*, G^*)$.

Имитационную модель можно рассматривать как преобразователь входных переменных в выходные. В любой имитационной модели различают составляющие: компоненты, переменные, параметры, функциональные зависимости, ограничения, целевые функции. *Модель* системы определяется как совокупность компонент, объединенных для выполнения заданной функции $Y = f(X, G)$. Здесь Y, X, G - векторы соответственно результата действия модели системы *выходных переменных, параметров моделирования, входных переменных модели.* Параметры модели X исследователь выбирает произвольно, G -принимают только те значения, которые характерны для данных объекта моделирования.

Очевидный подход в оценке адекватности состоит *в сравнении выходов модели и реальной системы при одинаковых* (если возможно) *значениях входов.* И те, и другие данные (данные, полученные на выходе имитационной модели и данные, полученные в результате эксперимента с реальной системой) — статистические. Поэтому *применяют методы статистической теории оценивания и проверки гипотез.*

Используя соответствующий статистический критерий для двух выборок, мы можем проверить статистические гипотезы (H_0) о том, что выборки выходов системы и модели являются выборками из различных совокупностей или (H_1), что они "практически" принадлежат одной совокупности.

Могут быть рекомендованы два основных подхода к оценке адекватности:

1 *способ*: *по средним значениям откликов модели и системы.*

2 *способ*: *по дисперсиям отклонений откликов модели от среднего значения откликов систем.*

А если не существует реальной системы (что характерно для задач проектирования, прогнозирования)? Проверку адекватности выполнить в этом случае не удается, поскольку нет реального объекта. Для целей исследования модели иногда проводят специальные испытания (например, так поступают при военных исследованиях). Это позволяет убедиться в точности модели, полезности ее на практике, несмотря на сложность и дороговизну проводимых испытаний.

Могут использоваться и другие подходы к проведению валидации имитационной модели [56], кроме статистических сравнений между откликами реальной системы и модели. В отдельных случаях полезна валидация внешнего представления, когда проверяется насколько модель выглядит адекватной с точки зрения специалистов, которые с ней будут работать, так называемый *тест Тьюринга* (установление экспертами различий между поведением модели и реальной системы). В процессе валидации требуется постоянный контакт с заказчиком модели, дискуссии с экспертами по системе. Рекомендуется также проводить эмпирическое тестирование допущений модели, в ходе которого может осуществляться графическое представление данных, проверка гипотез о распределениях, анализ чувствительности и др. Важным инструментом валидации имитационной модели является графическое представление промежуточных результатов и выходных данных, а также анимация процесса моделирования. Наиболее эффективными являются такие представления данных, как гистограммы, временные графики отдельных переменных за весь период моделирования, графики взаимозависимости, круговые и линейчатые диаграммы. Методика применения статистических технологий зависит от доступности данных по реальной системе.

Верификация имитационной модели

Верификация модели — есть доказательство утверждений соответствия алгоритма ее функционирования замыслу моделирования и своему назначению. На этапе верификации устанавливается *верность логической структуры модели,* реализуется комплексная отладка с использованием средств трассировки, ручной имитации, в ходе которой проверяется правильность реализации моделирующего алгоритма.

Комплексные процедуры верификации включают неформальные и формальные исследования программы-имитатора. Неформальные процедуры могут состоять из серии проверок следующего типа: проверка преобразования информации от входа к выходу; трассировка модели на реальном потоке данных (при заданных *G* и *X*):

X изменяется по всему диапазону значений контролируется Y;

- можно посмотреть, не будет ли модель давать абсурдные ответы, если ее параметры будут принимать предельные значения;
- "проверка на ожидаемость", когда в модели заменяют стохастические элементы на детерминированные и др.

Полезным при решении указанных задач могут быть также следующие приёмы [56]:

обязательное масштабирование временных параметров в зависимости от выбранного шага моделирования (валидация данных);

валидация по наступлению "событий" в модели и сравнение (если возможно) с реальной системой;

тестирование модели для критических значении и при наступлении редких событий;

фиксирование значений для некоторых входных параметров с последующим сравнением выходных результатов с заранее известными данными;

вариация значениями входных и внутренних параметров модели с последующим сравнительным анализом поведения исследуемой системы;

реализация повторных прогонов модели с неизменными значениями всех входных параметров;

оценка фактически полученных в результате моделирования распределений случайных величин и оценок их параметров (математическое ожидание и дисперсия) с априорно заданными

значениями;

сравнение исследователями поведения и результатов валидируемой модели с результатами уже существующих моделей, для которых доказана достоверность;

для существующей реальной исследуемой системы предсказание её будущего поведения и сравнение прогноза с реальными наблюдениями.

Формальные процедуры связаны с проверкой исходных предположений (выдвинутых на основе опыта, теоретических знаний, интуитивных представлений, на основе имеющейся информации). Общая процедура включает:

построение ряда гипотез о поведении системы и взаимодействии ее элементов;

проверка гипотез с помощью статистических тестов: используют методы статистической теории оценивания и проверки гипотез (методы проверки с помощью критериев согласия (χ^2, Колмогорова-Смирнова, Кокрена и др.), непараметрические проверки и т.д., а также дисперсионный, регрессионный, факторный, спектральный анализы).

6.4 Валидация данных имитационной модели

Валидация данных имитационной модели предполагает *исследование свойств* имитационной модели, в ходе которого оценивается *точность, устойчивость, чувствительность результатов моделирования* и другие свойства имитационной модели.

Наиболее существенные процедуры исследования свойств модели: *оценка точности результатов моделирования; оценка устойчивости результатов моделирования;*

оценка чувствительности имитационной модели.

Получить эти оценки в ряде случаев бывает весьма сложно. Однако без успешных результатов этой работы, доверия к модели не будет, невозможно будет провести корректный проблемный анализ и сформулировать статистически значимые выводы на основе данных, полученных в результате имитации.

Методы доказательства правильности программ

Как известно, универсальные вычислительные машины могут быть запрограммированы для решения самых разнородных задач - в этом заключается одна из основных их особенностей, имеющая огромную практическую ценность. Один и тот же компьютер, в зависимости от того, какая программа находится у него в памяти, способен осуществлять арифметические вычисления, доказывать теоремы и редактировать тексты, управлять ходом эксперимента и создавать проект автомобиля будущего, играть в шахматы и обучать иностранному языку. Однако успешное решение всех этих и многих других задач возможно лишь при том условии, что компьютерные программы не содержат ошибок, которые способны привести к неверным результатам.

Можно сказать, что требование отсутствия ошибок в программном обеспечении совершенно естественно и не нуждается в обосновании. Но как убедиться в том, что ошибки, в самом деле, отсутствуют? Вопрос не так прост, как может показаться на первый взгляд.

К неформальным методам доказательства правильности программ относят отладку и тестирование, которые являются необходимой составляющей на всех этапах процесса программирования, хотя и не решают полностью проблемы правильности. Существенные ошибки легко найти, если использовать соответствующие приемы отладки (контрольные распечатки, трассировки).

Тестирование – процесс выполнения программы с намерением найти ошибку, а не подтвердить правильность программы. Суть его сводится к следующему. Подлежащую проверке программу неоднократно запускают с теми входными данными, относительно которых результат известен заранее. Затем сравнивают полученный машиной результат с ожидаемым. Если во всех случаях тестирования налицо совпадение этих результатов, появляется некоторая уверенность в том, что и последующие вычисления не приведут к ошибочному итогу, т.е. что исходная программа работает правильно.

Мы уже обсуждали понятие правильности программы с точки зрения отсутствия в ней ошибок. С интуитивной точки зрения программа будет правильной, если в результате ее выполнения будет достигнут результат, с целью получения которого и была написана программа. Сам по себе факт безаварийного завершения программы еще ни о чем не говорит: вполне возможно, что программа в действительности делает совсем не то, что было задумано. Ошибки такого рода могут возникать по различным причинам.

В дальнейшем мы будем предполагать, что обсуждаемые программы не содержат синтаксических ошибок, поэтому при обосновании их правильности внимание будет обращаться только на содержательную сторону дела, связанную с вопросом о том, достигается ли при помощи данной программы данная конкретная цель. Целью можно считать поиск решения поставленной задачи, а программу рассматривать как способ ее решения. Программа будет правильной, если она решит сформулированную задачу.

Метод установления правильности программ при помощи строгих средств известен как верификация программ.

В отличие от тестирования программ, где анализируются свойства отдельных процессов выполнения программы, верификация имеет дело со свойствами программ.

В основе метода верификации лежит предположение о том, что существует программная документация, соответствие которой требуется доказать. Документация должна содержать:

♣спецификацию ввода-вывода (описание данных, не зависящих от процесса обработки);

♣свойства отношений между элементами векторов состояний в выбранных точках программы;

♣спецификации и свойства структурных подкомпонентов программы;

♣спецификацию структур данных, зависящих от процесса обработки.

К такому методу доказательства правильности программ относится метод индуктивных высказываний, независимо сформулированный К. Флойдом и П. Науром.

Суть этого метода состоит в следующем:

1) формулируются входное и выходное высказывания: входное высказывание описывает все необходимые входные условия для программы (или программного фрагмента), выходное высказывание описывает ожидаемый результат;

2) предполагая истинным входное высказывание, строится промежуточное высказывание, которое выводится на основании семантики операторов, расположенных между входом и выходом (входным и выходным высказываниями); такое высказывание называется выведенным высказыванием;

3) формулируется теорема (условия верификации):

из выведенного высказывания следует выходное высказывание;

4) доказывается теорема; доказательство свидетельствует о правильности программы (программного фрагмента).

Доказательство проводится при помощи хорошо разработанных математических методов, использующих исчисление предикатов первого порядка.

Условия верификации можно построить и в обратном направлении, т.е., считая истинным выходное высказывание, получить входное высказывание и доказывать теорему: из входного высказывания следует выведенное высказывание.

Такой метод построения условий верификации моделирует выполнение программы в обратном направлении. Другими словами, условия верификации должны отвечать на такой вопрос: если некоторое высказывание истинно после выполнения оператора программы, то, какое высказывание должно быть истинным перед оператором?

Построение индуктивных высказываний помогает формализовать интуитивные представления о логике программы. Оно и является самым сложным в процессе доказательства правильности программы. Это объясняется, во-первых, тем, что необходимо описать все содержательные условия, и, во-вторых, тем, что необходимо аксиоматическое описание семантики языка программирования.

Важным шагом в процессе доказательства является доказательство завершения выполнения программы, для чего бывает достаточно неформальных рассуждений.

Таким образом, алгоритм доказательства правильности программы методом индуктивных высказываний представляется в следующем виде:

1) Построить структуру программы.
2) Выписать входное и выходное высказывания.
3) Сформулировать для всех циклов индуктивные высказывания.
4) Составить список выделенных путей.
5) Построить условия верификации.
6) Доказать условие верификации.
7) Доказать, что выполнение программы закончится.

Этот метод сравним с обычным процессом чтения текста программы (метод сквозного контроля). Различие заключается в степени формализации.

Преимущество верификации состоит в том, что процесс доказательства настолько формализуем, что он может выполняться на вычислительной машине. В этом направлении в восьмидесятые годы проводились исследования, даже создавались автоматизированные диалоговые системы, но они не нашли практического применения.

Для автоматизированной диалоговой системы программист должен задать индуктивные высказывания на языке исчисления предикатов. Синтаксис и семантика языка программирования должны храниться в системе в виде аксиом на языке исчисления предикатов. Система должна определять пути в программе и строить условия верификации.

Основной компонент доказывающей системы - это построитель условий верификации, содержащий операции манипулирования предикатами, алгоритмы интерпретации операторов программы. Вторым компонентом системы является подсистема доказательства теорем.

Отметим трудности, связанные с методом индуктивных высказываний. Повторим, что трудно построить «множество основных аксиом, достаточно ограниченное для того, чтобы избежать противоречий, но достаточно богатое для того, чтобы служить отправной точкой для доказательства высказываний о программах». Вторая трудность - семантическая, заключающаяся в формировании самих высказываний, подлежащих доказательству. Если задача, для которой пишется программа, не имеет строгого математического описания, то для нее сложнее сформулировать условия верификации.

Перечисленные методы имеют одно общее свойство: они рассматривают программу как уже существующий объект и затем доказывают ее правильность.

Метод, который сформулировали К. Хоар и Э. Дейкстра основан на формальном выводе программ из математической постановки задачи.