Верификация бортового ПО согласно DO-178C

Работа с авиационной индустрией программного обеспечения для решения задач обеспечения рентабельной сертификации

ldra.exponenta.ru

LDRA Software Technology

экспонента
ЦЕНТР ИНЖЕНЕРНЫХ ТЕХНОЛОГИЙ И МОДЕЛИРОВАНИЯ

Содержание

Введение

В ответ на более широкое использование программного обеспечения в бортовых системах Радиотехническая комиссия по организации воздухоплавания[1] (в настоящее время известная как RTCA, Inc.) в сотрудничестве с EUROCAE[2], создала руководящий документ DO-178 «Соображения программного обеспечения в сертификации бортовых систем и оборудования». » Этот документ стал признанным международным сертификационным стандартом для бортового программного обеспечения.

Первоначально опубликованный в 1982 году, переработанный в 1992 году как DO-178B и значительно расширившийся в 2011 году для решения современных технологий и методологий в DO-178C, стандарт отражает опыт, накопленный для удовлетворения сегодняшних потребностей авиационной отрасли.

LDRA активно участвовала в работе комитетов DO-178B[3] и DO-178C[4] в течение почти двух десятилетий. Майк Хеннелл, генеральный директор LDRA, сыграл важную роль в включении в стандарт нескольких целей тестовых метрик, в том числе связанных с анализом структурного покрытия. Набор инструментов LDRA® был первопроходцем автоматической проверки для сертификации как стандарту DO-178B для бортовых программных систем, так и его сопутствующего стандарта DO-278[5] для наземных систем.

Стандарт DO-178C предоставляет подробные рекомендации по разработке и проверке критически важного бортового программного обеспечения. В соответствии с ARP 4754A[6], до разработки системы, анализы функциональной опасности и оценки безопасности системы проводятся для определения вклада системы в потенциальные условия отказа. Затем для определения уровня обеспечения качества (DAL), как показано на рисунке 1, используется степень тяжести условий отказа на самолете и его пассажирах.

[1] http://www.rica.org/

[2] https://www.eurocae.net/

[3] http://www.rica.org/store_product.asp?prodid=581

[4] http://www.rica.org/store_product.asp?prodid=803

[5] http://www.rica.org/store_product.asp?prodid=678%20%20

[6] http://standards.sae.org/arp475a/

DAL	Отказное состояние	Описание
A	Катастрофическое	отказное состояние, для которого принимается, что при его возникновении предотвращение гибели людей оказывается практически невозможным
B	Аварийное	отказное состояние, которое может привести к значительному ухудшению характеристик воздушного судна, и/или физическому утомлению или такой рабочей нагрузке экипажа, что уже нельзя полагаться на то, что он выполнит свои задачи точно и полностью
C	Сложное	отказное состояние, которое может привести к заметному ухудшению характеристик воздушного судна, и/или выходу одного или нескольких параметров за эксплуатационные ограничения, но без достижения предельных ограничений, и/или уменьшению способности экипажа справиться с неблагоприятными условиями, как из-за увеличения рабочей нагрузки, так и из- за условий, понижающих эффективность действий экипажа.
D	Усложнение условий полёта	отказное состояние, которое может привести к незначительному ухудшению характеристик воздушного судна, и/или незначительному увеличению рабочей нагрузки на экипаж.
E	Без последствий	отказное состояние, которое не влияет на характеристики воздушного судна и не увеличивает рабочую нагрузку на экипаж.

Рисунок 1. Уровни обеспечения качества, по таблице 2-1 DO-178C

Затем процесс разработки ARP 4754A распределяет связанные уровни на подсистемы, которые реализуют требования к электронному оборудованию и программному обеспечению системы. DO-178C устанавливает пять «уровней программного обеспечения» и регламентирует цели, которые должны быть удовлетворены. Это означает, что усилия и затраты на создание системы, критически важной для непрерывной безопасной эксплуатации самолета (например, системы управления полетом), обязательно выше, чем требуется для создания системы с незначительным воздействием на самолет в случае отказа (например, детектор дыма в ванной комнате).

DO-178C охватывает весь жизненный цикл программного обеспечения: планирование, разработку и интегральные процессы для обеспечения правильности и надежности программного обеспечения. Интегральные процессы включают проверку программного обеспечения, обеспечение качества программного обеспечения, обеспечение управления конфигурацией и связь с сертифицирующими органами.

Стандарты не обязывают разработчиков использовать инструменты для анализа, тестирования и трассируемости в своей работе. Однако эти инструменты повышают эффективность во всех случаях, кроме самых тривиальных проектов, в той мере, в какой они играют значительную роль в достижении целей летной годности для бортового программного обеспечения на протяжении всего жизненного цикла разработки. Специализированные инструменты, представленные в наборе инструментов LDRA, используются для достижения целей DO-178C, включая двунаправленную трассируемость, управление тестированием, статический анализ исходного кода и динамический анализ исходного и объектного кода.

В этом документе описываются основные процессы разработки и проверки программного обеспечения согласно стандарту и показано, как автоматизация может помочь снизить стоимость разработки и проверки, а так же обеспечить развертывание критически важного программного обеспечения.

Цели процесса DO-178B

DO-178C указывает, что безопасность программного обеспечения должна систематически проверяться на протяжении всего жизненного цикла программного обеспечения. Это включает в себя процессы трассируемости жизненного цикла, разработки программного обеспечения, кодирования, верификации и валидации, используемые для обеспечения правильности, контроля и уверенности в программном обеспечении.

Ключевыми элементами жизненного цикла программного обеспечения DO-178C являются методы установления трассируемости и анализ структурного покрытия. Двунаправленная трассируемость должна устанавливаться на протяжении всего жизненного цикла, от системных требований до требований к высокому уровню программного обеспечения, от требований высокого уровня до требований низкого уровня, а также для тестовых векторов, процедур тестирования и результатов тестирования. Требования низкого уровня должны быть затем связаны с исходным кодом, в котором они реализованы. Анализ структурного покрытия (покрытие кода, связь по данным и связь по управлению) определяет степень исполнения исходного кода системы в процессе тестирования. Используя эти практики, можно обеспечить, что код был реализован для удовлетворения всех требований к системе и что реализованный код был проверен на полноту.

Использование программных средств обеспечивает особенно значительную выгоду при разработке программного обеспечения и проверке программного обеспечения, о чем говорится в разделах 5.0 и 6.0 стандарта соответственно.

DO-178C Раздел 5.0: Процессы разработки программного обеспечения

Пять процессов высокого уровня идентифицированы в разделе ПРОЦЕССЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ DO-178C; Процесс разработки требований к ПО (5.1), Процесс проектирования ПО (5.2), Процесс кодирования программного обеспечения (5.3), Процесс интеграции (5.4) и Трассируемость (5.5).

Идеальные инструменты для управления требованиями (раздел 5.1) во многом зависят от масштабов разработки. Если в офисе мало разработчиков, может оказаться достаточно простой электронной таблицы или документа Microsoft Word. Большие проекты, с командами в географически разных местах, получат преимущества при применении инструментов управления жизненным циклом ПО, таких как IBM Rational DOORS[7], Siemens Polarion PLM[8] или, в более общем случае, аналогичных инструментов, предлагающих поддержку стандартных форматов требований[9].

Продукты этапа проектирования (раздел 5.2) потенциально могут включать в себя проекты на основе моделей, электронные таблицы, текстовые документы и многие другие артефакты, и, очевидно, в их производстве может быть задействовано множество инструментов. Управление статусом каждого из этих элементов и поддержание трассируемости между требованиями, артефактами и последующими этапами разработки обычно приводит к "головной боли" управления проектами. Это рассматривается в разделе 5.5 стандарта, обсуждаемом ниже.

В разделе 5.3 DO-178C указывает, что программное обеспечение должно соответствовать определенным целям процесса кодирования программного обеспечения. Эти цели предусматривают, что разработка исходного кода должна реализовывать требования низкого уровня и соответствовать стандартам кодирования.

[7] http://www.-03ibm.com/software/products/en/ratidoor

[8] http://polarion.pim.automation.siemens.com/

[9] http://www.omg.org/spec/ReqIF/

Дальнейшее определение стандартов кодирования программного обеспечения приведено в Разделе 11.8 DO-178C:

- Язык (языки) программирования, который будет использоваться, и / или определенные подмножества. Для языка программирования устанавливается подход к однозначному определению синтаксиса, поведения управления, поведения данных и побочных эффектов языка. Это может потребовать ограничения использования некоторых особенностей языка.
- Стандарты представления исходного кода, включая ограничение длины строки, отступы и использование пустой строки.
- Стандарты документации исходного кода, например, имя автора, история изменений, входы и выходы и затронутые глобальные данные.
- Соглашения об именах для компонентов, подпрограмм, переменных и констант.
- Условия и ограничения, налагаемые на разрешенные соглашения о кодировании, такие как степень связи между программными компонентами и сложность логических или числовых выражений и обоснование их использования.
- Ограничения на использование инструментов кодирования.

Инструменты статического анализа автоматизируют «проверку» исходного кода, упрощают проверку соответствия, уменьшают вероятность ошибок и более экономичны, сравнивая просматриваемый код с правилами, продиктованными выбранным стандартом кодирования программного обеспечения (рисунок 2). Несоответствия выделяются в соответствии с разделом 6.4.3d стандарта. Набор инструментов также может оценить сложность рассматриваемого кода, чтобы он оставался ниже безопасного порога для системы, и его средство анализа потока данных может использоваться для идентификации любых неинициализированных или неиспользуемых переменных и / или констант, как указано в разделе 6.4.3.f.

Рисунок 2. Проверка соответствия стандарту кодирования MISRA C ++: 2008 с помощью набора инструментов LDRA

Стандарты кодирования

Существует множество стандартов кодирования с разными атрибутами, но, тем не менее, с сильным сходством, особенно при ссылках на один и тот же язык. Самые популярные стандарты включают:

1. C/C++
1.1. MISRA C:1998MISRA C++:2008 MISRA C:2004JSF++ AV
1.2. MISRA C:2012 /HIC++ AMD1 / ADD2
1.3. CERT CWE

2. Ada
2.1. Ravenscar Spark

3. Java:
3.1. CWE, CERT J

Существует множество заранее определенных подмножеств языков (иногда называемых «стандартами кодирования»), доступных для языков C, C ++ и Ada (см. сбоку), и стандарт не означает, что стандарт предприятия (СТП) не может быть предпочтительным. Стандарт можно установить полностью с нуля или, что более прагматично, основываясь на установленном стандарте с изменениями, соответствующими конкретному проекту. Важно, чтобы развернутый инструмент статического анализа был также гибким.

В разделе 5.5 DO-178C указывает, что правильность процесса разработки и проверки на основе требований определяется покрытием требований или трассируемостью. Этот анализ гарантирует, что требования двунаправленно связаны между системными и высокоуровневыми требованиями, требованиями высокого уровня и низкого уровня и, наконец, требованиями низкого уровня и исходным кодом.

Статический анализ не только обеспечивает полезную проверку стандартов кодирования. Он также показывает основную структуру программного обеспечения, которая требуется для подтверждения такой трассируемости.

Если все будет следовать жизненному циклу разработки как в стандарте, то поддержание трассируемости, возможно, одноразовая, тривиальная задача. Требования никогда не изменятся, и тесты никогда не вызовут проблем. Но, к несчастью, это редкое явление.

Итак, представим, что произойдет, если проблема обнаруживается во время статического анализа.

- Возможно, в требованиях есть противоречие. Если это так, требования должны измениться. Но какие другие части программного обеспечения затронуты этим?

- Возможно, существует требование низкого уровня, которое не трассируется до требования высокого уровня. Что нужно изменить, чтобы решить это?

- Возможно, у конечного пользователя есть новое требование. Какое влияние это окажет на установленные требования, проект и исходный код?

Проблемы, связанные с функциональностью, скорее всего, будут устранены во время динамического анализа позже в жизненном цикле. Это означает, что головоломка причинно-следственной связи становится еще более сложной, когда что-то нужно изменить.

Использование решений по управлению трассируемостью / покрытию требований и управлению тестированием, которое интегрировано с анализом кода, анализом связи по данным и управлению, а также низкоуровневым тестированием и инструментами покрытия кода, устраняет проблемы управления проектами, связанные с такой сложностью (рисунок 3). Эти решения гарантируют, что матрицей трассируемости требований, даже при разрозненных источниках и вплоть до исходного кода и тестовых векторов, намного проще управлять.
Это ведет к сокращению затрат, так как матрица трассируемости всегда находится в актуальном состоянии.

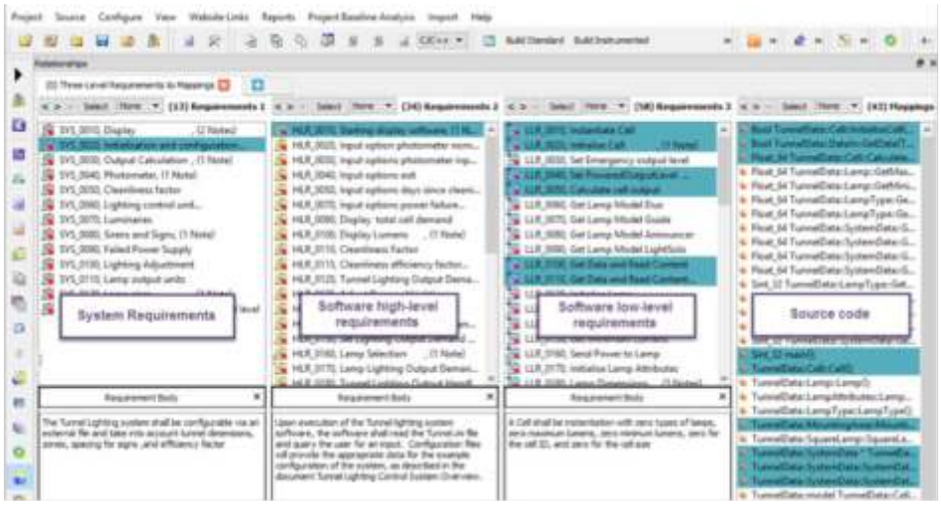


Рисунок 3. Автоматизация управления трассируемостью с TBmanager из набора инструментов LDRA

DO-178C Раздел 6.0: Процесс верификации программного обеспечения

В отличие от статического анализа, который можно рассматривать как автоматизированную «инспекцию» исходного кода, динамический анализ включает в себя выполнение исполняемого объектного кода (далее, EOC – Executable Object Code) по частям или целиком, используя целевую среду, представляющую ту, в которой будет развернуто готовое приложение. Это исполнение используется для предоставления доказательств как функциональной корректности, так и выполнения частей кода («структурное покрытие»).

DO-178C обсуждает обе эти концепции, идентифицируя цели для обеспечения покрытия тестами требований высокого и низкого уровня, а также для обеспечения соответствующего покрытия тестированием как структуры программного обеспечения, так и связей по данным и управлению.

«Тестовые вектора и процедуры», на которые делается ссылка в стандарте, могут включать тесты низкого уровня (иногда называемые модульными тестами), интеграционные тесты или системные тесты, а также комбинацию из всех трех.

Низкоуровневые тесты предназначены для проверки выполнения требований низкого уровня. Процедуры тестирования должны быть созданы, рассмотрены и выполнены, чтобы гарантировать, что программное обеспечение не содержит никаких нежелательных функций. Затем могут быть выполнены низкоуровневые тесты на целевом оборудовании или имитируемой среде, указанной в Плане верификации программного обеспечения (далее, SVP – Software Verification Process). Как только процедуры тестирования выполняются, фактические выходы фиксируются и сравниваются с ожидаемыми результатами, а результаты пройденных и не пройденных тестов документируются (рисунок 4).

Тестирование интеграции программного обеспечения предназначено для проверки взаимосвязи между программными компонентами в отношении требований к архитектуре программного обеспечения. На практике механизмы, используемые для низкоуровневого тестирования, часто расширяются, чтобы проверить поведение в дереве вызовов.

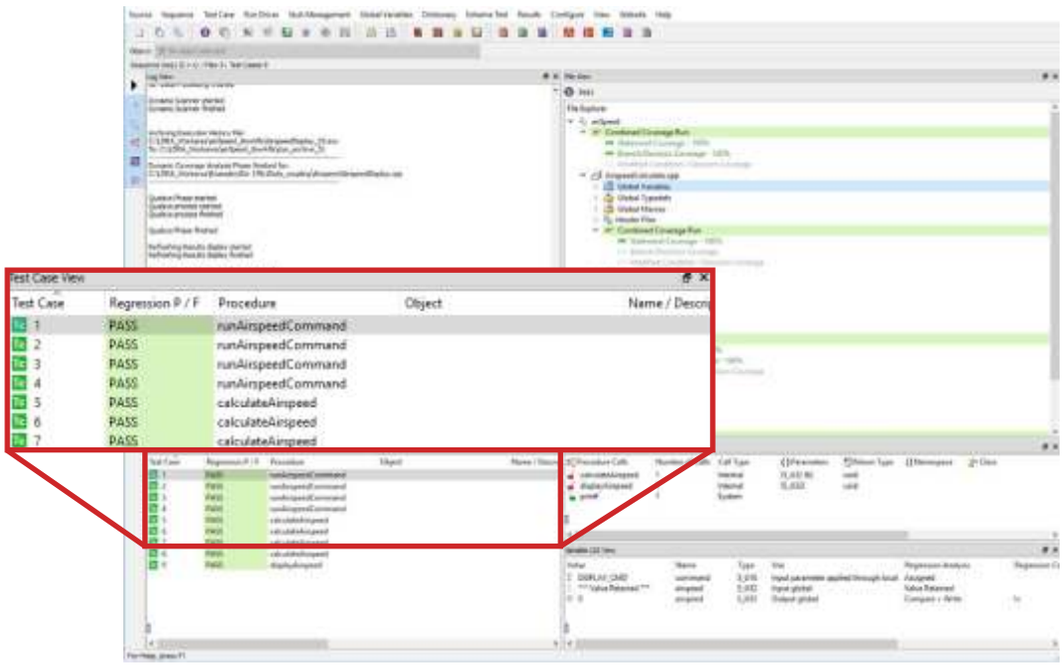


Рисунок 4. Автоматизация низкоуровневого и интеграционного тестирования с TBrun из набора инструментов LDRA

Если изменения становятся необходимыми - возможно, в результате неудачного динамического теста или в результате изменений требований - тогда все низкоуровневые и интеграционные тесты должны быть повторно запущены (выполнено регрессионное тестирование). Эти регрессионные тесты могут быть автоматизированы и систематически повторно применены по мере разработки, чтобы гарантировать, что новая функциональность не будет нарушать установленную и проверенную.

Отслеживание статуса проекта в таком потоке является сложной задачей. Автоматизация поддержания двунаправленной взаимосвязи между продуктами разных этапов разработки экономит много времени и делает ошибки гораздо менее вероятными - не только в плане развития требований и исходного кода, но и тестирования, основанного на требованиях для требований высокого и, при необходимости, низкого уровня.

Рисунок 5. Трассируемость требований в графическом пользовательском интерфейсе (GUI) TBmanager, из набора инструментов LDRA

Область слева от рисунка 5 показывает графическое представление политики трассировки между требованиями и тестами тестов разных областей. Требования и тестовые вектора затем создаются или импортируются и связаны друг с другом, поэтому можно оценить полноту требований и покрытие тестами. Тестовые вектора могут быть быстро и легко рассмотрены и разработаны на основе требований, определяя и заполняя пробелы в покрытии требований.

Двунаправленный анализ также может идентифицировать тестовые вектора, которые не связаны с требованиями, выявляя необходимость внесения соответствующих изменений в требования, тестовые вектора или соотношения трассируемости. Это может обеспечить анализ в отношении изменений требований, тестов или кода и их потенциального воздействия на временные рамки проекта. Также, это может обеспечивать понимание целей для регрессионного тестирования, что сведет к минимуму нагрузку на дополнительные проверки. И он может предоставить необходимые доказательные артефакты, такие как матрицы трассируемости, чтобы показать, что тестовое покрытие требований высокого уровня и низкого уровня было достигнуто[10].

На рисунке 6 показана матрица трассируемости между требованиями высокого уровня и функциональными тестовыми примерами. Тридцать три из тридцати четырех требований имеют связанный тестовый вектор, поэтому полное покрытие тестами требований высокого уровня все еще не достигнуто. Этот уровень прозрачности необходим для обеспечения привязки всех требований к тестам.

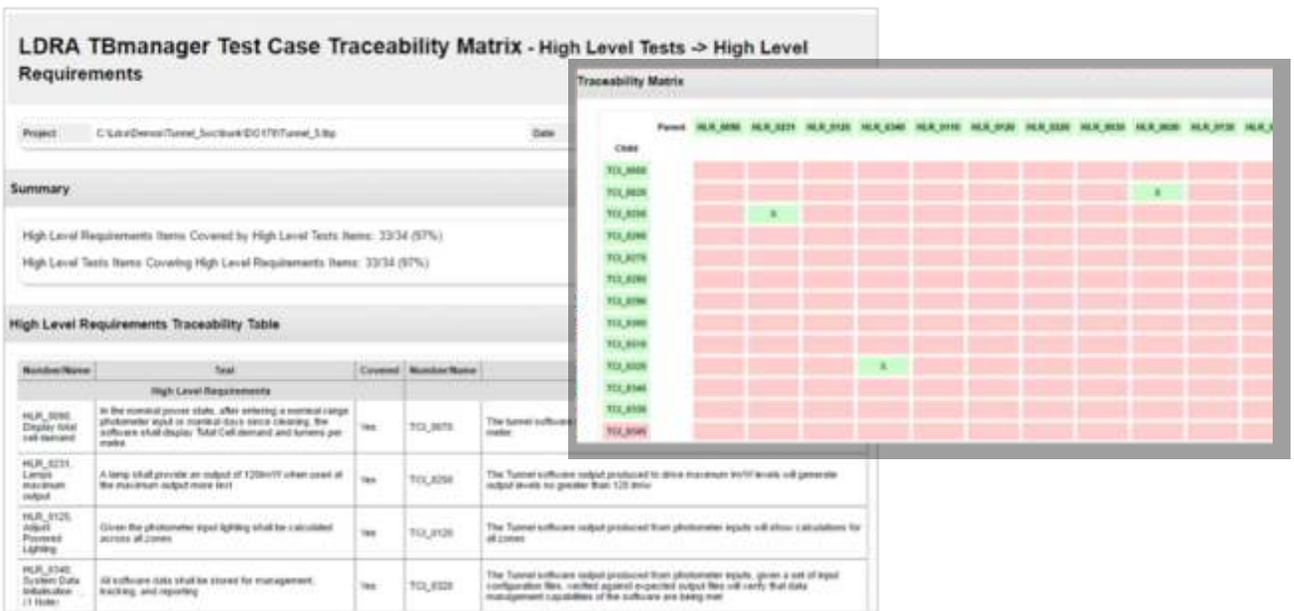


Рисунок 6. Пример матрицы трассируемости (трассируемость требований высокого уровня к тестовым векторам), из TBmanager

Цели анализа структурного покрытия DO-178C

Структурное покрытие (далее, SC – Structure Coverage) используется для определения того, какие структуры кода и интерфейсы компонентов выполнялись во время выполнения процедур тестирования на основе требований, что облегчает эмпирическое измерение эффективности теста на основе требований. Как следует из названия, анализ структурного покрытия (далее, SCA – Structure Coverage Analysis) включает в себя анализ SC, чтобы определить, есть ли какие-либо части кода, которые недостаточно изучены, и, если таковые есть, то почему.

[10] Таблица A-7, цели 3 и 4, DO-178C

Достижение целей A-7.5, 6 и 7 (рисунок 8) включает в себя сопоставление показателей структурного покрытия, как правило, путем «наложения» копии исходного кода, то есть наложения его на вызовы функций для сопоставления данных покрытия и выполнения этого инструментированного кода, используя тестовые вектора, основанные на требованиях. Эти тестовые вектора в основном опираются на требования высокого уровня, и дополняются низкоуровневыми требованиями.

Затем SCA применяется для оценки эффективности этого тестирования путем измерения того, какая часть кода была выполнена. Покрытие частей кода, не выполненных до сих пор, может потребовать дополнительных тестовых векторов или модификаций существующих тестовых векторов, изменений в требованиях, удаления «мертвого кода»[11] или, возможно, идентификации деактивированного кода, и непредвиденного функционала. Итеративный цикл «рассмотрение, анализ, проверка», как правило, необходим для обеспечения того, чтобы покрытие программного обеспечения было достигнуто, а требования низкого уровня проверены, и графическое представление является большой помощью.

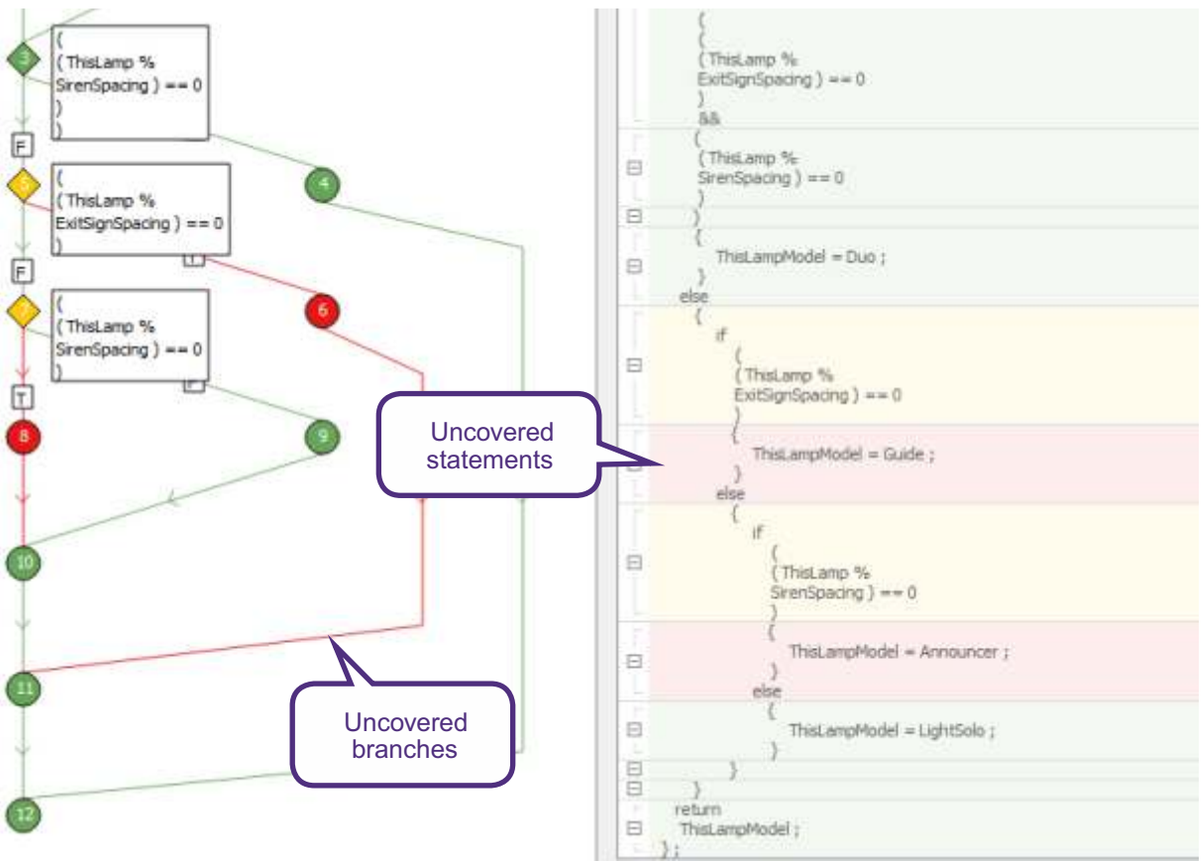


Рисунок 7. Графическая визуализация покрытия кода в графе вызовов в наборе инструментов LDRA

Можно показать, что системные требования были правильно декомпозированы, реализованы и проверены путем объединения полной трассируемости от требований к коду и тестовым векторам с достижением полного покрытия функциональными тестами и целей структурного покрытия.

[11] Т В DO-178C - недосягаемый код (более широкий термин, включающий как неисполняемый, так и не трассируемый на требования код)

№	Описание цели	Пункт DO-178C	DO-178C Уровень A	DO-178C Уровень B	DO-178C Уровень C	DO-178C Уровень B
5	Покрытие структуры программы (модифицированное покрытие условий/решений)[12] по результатам испытаний достигается	6.4.4.2	✓	Не требуется	Не требуется	Не требуется
6	Покрытие структуры программы (покрытие решений) по результатам испытаний достигается	6.4.4.2a 6.4.4.2b	✓	✓	Не требуется	Не требуется
7	Покрытие структуры программы (покрытие операторов) по результатам испытаний достигается	6.4.4.2a 6.4.4.2b	✓	✓	✓	Не требуется
8	Покрытие структуры программы (связи по данным и по управлению) по результатам испытаний достигается	6.4.4.2c	✓	✓	✓	Не требуется
9	Верификация дополнительного кода, не трассируемого к исходному достигается	6.4.4.2d	✓	Не требуется	Не требуется	Не требуется
✓	Удовлетворяется набором инструментов LDRA, который отвечает требованию о независимости.					

Рисунок 8. Цели SCA для каждого уровня программного обеспечения (Таблица A-7)

На рисунке 8 показано, что цели A7-5, 6 и 7 DO-178C связаны с достижением 100% покрытия MC/DC, решений и операторов соответственно. Необходимая комбинация этих целей зависит от уровня безопасности.

Для систем уровня A структурного покрытия на уровне исходного кода недостаточно. Компиляторы часто добавляют дополнительный код или изменяют поток управления, и часто их поведение не является детерминированным. Чтобы гарантировать, что функциональность не скомпрометирована, DO-178C 6.4.4.2.b гласит:

«...уровень ПО есть A и компилятор генерирует объектный код, который не трассируется прямо на операторы «Исходного кода». В этом случае следует провести дополнительную верификацию объектного кода, чтобы убедиться в правильности таких сгенерированных последовательностей».

Автоматический механизм, обеспечивающий доказательства этой проверки, может сделать этот процесс намного более эффективным. Поскольку существует прямая однозначная взаимосвязь между объектным кодом и кодом ассемблера, один из способов для этого инструмента - отобразить графическое представление исходного кода

[12] Далее – MC/DC

наряду с эквивалентным представлением кода ассемблера. Верификация объектного кода (далее, OCV – Object Code Verification) измеряет покрытие как на уровне исходного кода, так и на уровне ассемблера, путем поочередного инструментирования (рис. 9).

Этот подход обеспечивает средства для демонстративной и детерминированной проверки исполняемого объектного кода (EOC) в целевой системе. Для того чтобы OCV была эффективной, она должен поддерживать микропроцессор, соответствующий набор команд и компилятор, развернутый в этой системе.

Для каждого тестового вектора используются три дискретных режима, чтобы быстро идентифицировать «дополнительный код», упомянутый в стандарте, и значительно сократить кропотливый ручной анализ.

1. Тестируемый код выполняется без инструментирования для подтверждения правильной работы.
2. Тестовый вектор выполняется с инструментированием на уровне исходного кода.
3. Наконец, тестовый вектор выполняется с помощью инструментария на уровне кода ассемблера, чтобы идентифицировать любые непокрытые инструкции или ветви, которые могут быть вставлены или изменены во время процесса компиляции и компоновки.

Как правило, несколько дополнительных тестов на основе требований могут быть добавлены для проверки того дополнительного кода для соответствия цели A7-9 (рисунок 7).

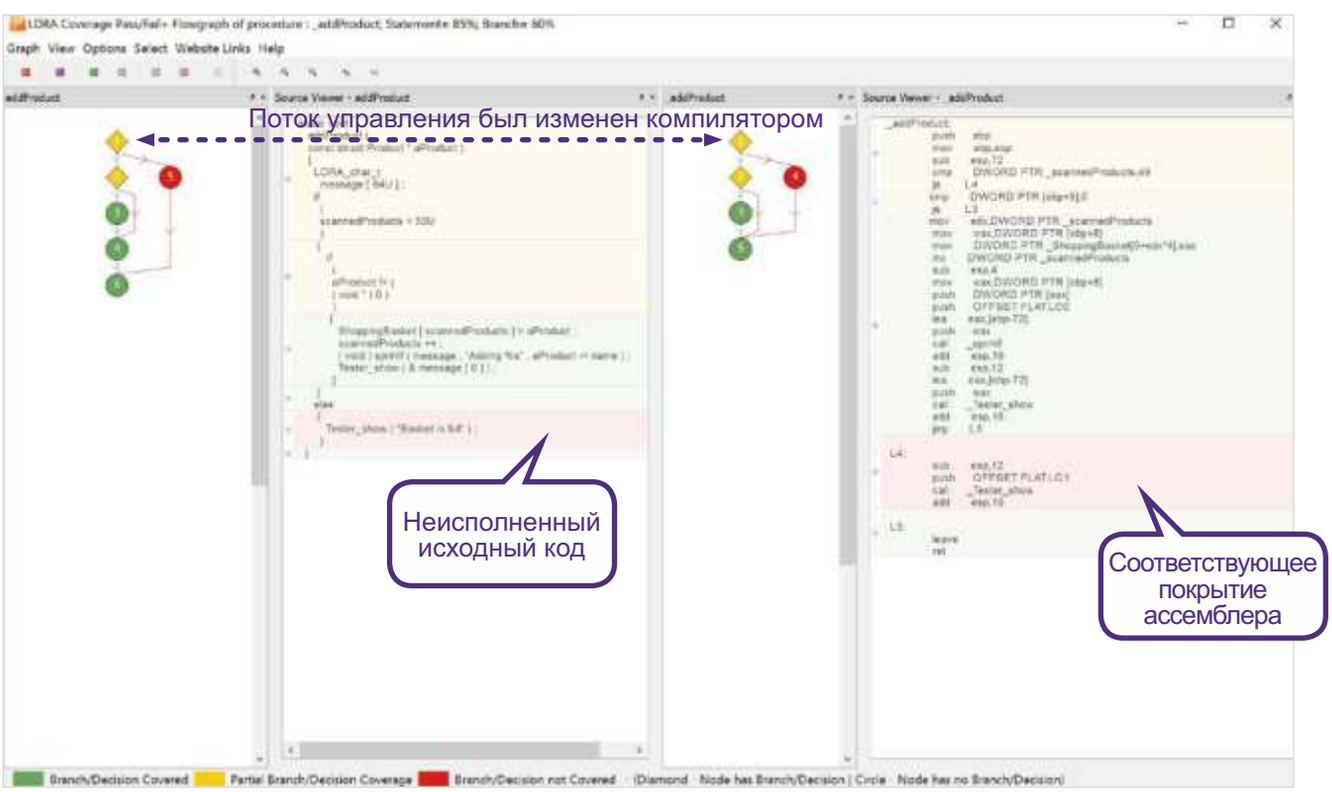


Рисунок 9. Визуализация потока управления и покрытия кода в C и связанного с ним кода ассемблера в наборе инструментов LDRA

Автоматизированное инструментирование исходного кода и анализ покрытия сокращают трудозатраты, что в свою очередь делает технологию масштабируемой и адаптируемой к широкому спектру кросс-компиляторов, целевых платформ, эмуляторов и других встроенных сред. Интеграция с целевыми системами очень расширяема и поддерживает процессоры от простых 8-разрядных устройств до высокопроизводительных многоядерных архитектур, IDE и интеграции ввода-вывода.

Демонстрация связи по данным и управлению

В эволюции стандарта от DO-178B до DO-178C произошла смена акцента на том, как должны демонстрироваться данные и управление.

В разделе 6.4.4.2.c DO-178B требуется: «В результате данного анализа следует подтвердить связь между компонентами кода по данным и по управлению».

В разделе 6.4.4.2.c DO-178C требуется «анализ, подтверждающий, что тестирование на основе требований использовало связь по данным и управлению между компонентами кода».

Таким образом, DO-178C изменяет цель подтверждения связей по данным и управлению: вместо «аналитическое выполнение по тестовым векторам» используется «Измерение, по результатам выполнения тестов»

Это означает, что анализ связей по данным и управлению должен быть выполнен после выполнения кода, и созданные артефакты проверяются на соответствие требованиям и архитектуре системы. Это, в свою очередь, накладывает новое бремя на цикл разработки.

Связь по управлению

Связь по управлению определяется DO-178C как «Вид или степень влияния, оказываемого одной компонентой программы на выполнение другой компоненты». Отчет о покрытии процедур и вызовов функций может быть представлен, как показано на рисунке 7, или в формате отчета для целей архивирования и контроля. Как визуальные, так и текстовые подходы помогают выявлять любые пробелы и приводят к сфокусированной деятельности по верификации.

SCA и связанные с ними артефакты обеспечивают видимость и данные для выполнения этих действий и соответствуют связанной цели A-7.8 (рисунок 7).

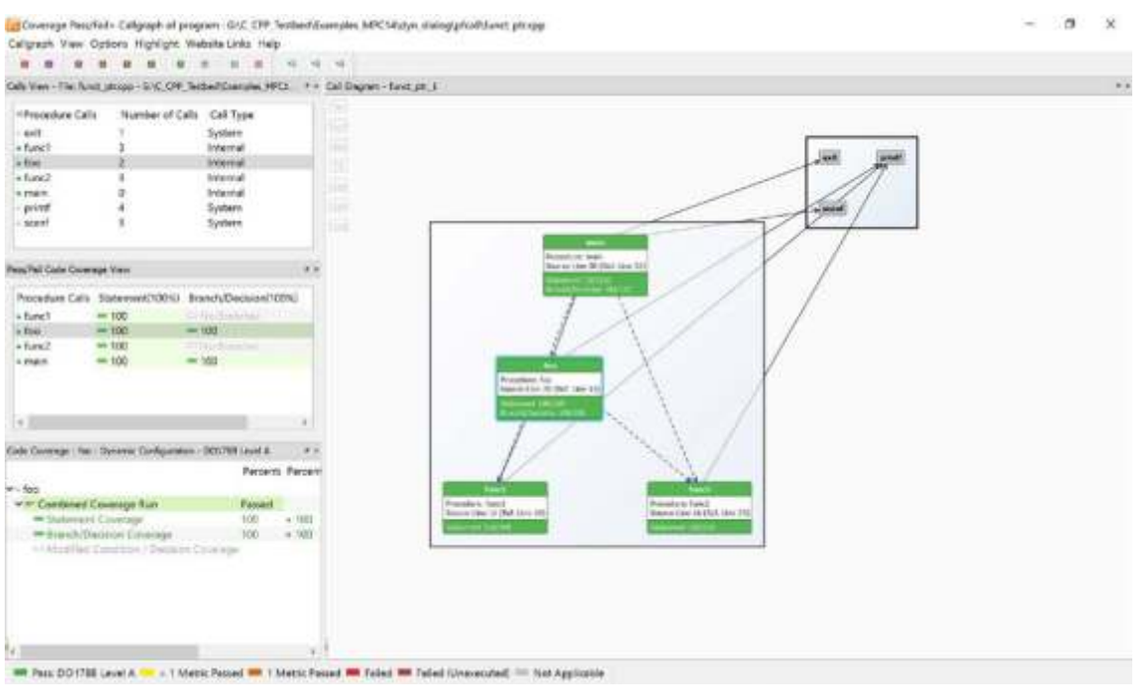


Рисунок 10. Покрытие процедур и вызовов функций, показанное в графах вызовов, сгенерированных набором инструментов LDRA

Связь по данным

Связь по данным определяется DO-178C как «Зависимость компоненты программы от данных, управление которыми осуществляется не только этой компонентой». Цель A-7.8 требует, чтобы «Тестовое покрытие структуры программного обеспечения, как связи по данным, так и связи по управлению достигается». Как и при анализе связи по управлению, любые измерения потока данных должны быть получены из выполнения тестов на основе требований.

Пример на рисунке 11 дублируется из DO-248C, а его реализация проиллюстрирована в функции runAirspeedCommand справа. Ожидаемое поведение этого исходного кода заключается в том, чтобы сначала вычислить воздушную скорость, а затем отобразить ее в этом порядке.

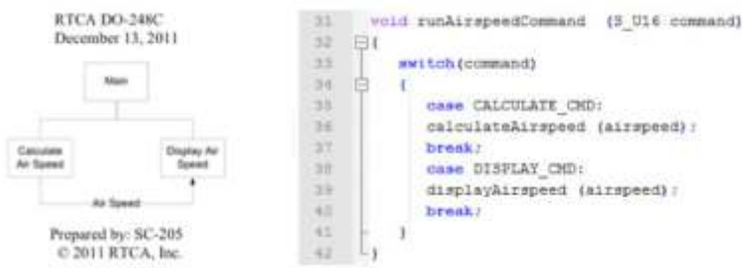


Рисунок 11. Пример из DO-248C1

Рисунок 12. Тестовый вектор, выполняющий runAirspeedCommand с результирующим потоком управления и структурным покрытием, представленным зеленым цветом. Пример из TBrun из набора инструментов LDRA

В тестовом векторе на приведенном выше рисунке показан второй случай в операторе switch, отраженный структурным покрытием ниже. Он также показывает, что display вызывается без обновления скорости полета последним значением. Дополнительные тестовые вектора могут вызывать команду calculateAirspeed, но необязательно после вызова displayAirspeed.

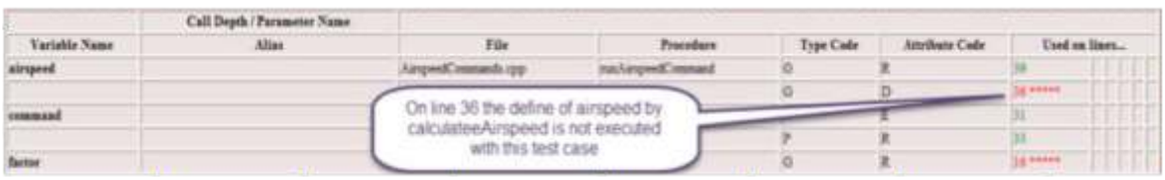


Рисунок 13. Отчет набора инструментов LDRA, показывающий переменную без ссылок во время выполнения. Эти артефакты используются для достижения цели A-7.8

Вышеприведенный отчет был получен на основе выполнения вышеприведенного тестового вектора. Он показывает информацию о динамическом потоке данных, показывающую, что воздушная скорость фактически не была

записана в строку 36 и не обновлялась до ее отображения, что потенциально отображало неточную информацию. В общем, наблюдаемый поток данных обеспечивает информацию, необходимую для согласования взаимодействия данных, требований и архитектуры, и поведения приложения.

Анализ связи по данным сосредоточен на наблюдении и анализе элементов данных, таких как airspeed, поскольку они установлены и используются («пары установка/использования») вне границ программного компонента. Ручное выполнение этого анализа с помощью отладчиков является трудоемким, трудно повторяемым и подверженным ошибкам. Автоматизация резко снижает эти риски.

Объектно-ориентированная технология

В начале 2000-х годов объектно-ориентированная технология рассматривалась в авионике как новая и непроверенная. Примерно в это же время команда разработчиков сертификационных центров (CAST) опубликовала статьи (CAST 4 и 8 в 2000 и 2002 годах, соответственно), чтобы перечислить ее проблемы и ограничения.

Когда DO-178B был обновлен до DO-178C, было решено, что эти проблемы, уязвимости и последующие дополнительные цели, связанные с объектно-ориентированными технологиями, будут рассматриваться не по оригинальному стандарту, а по дополнению DO-332[13]. Это новое дополнение описывает концепции и ключевые особенности объектно-ориентированных технологий и связанных с ними подходов, обсуждает их влияние на процессы планирования, разработки и проверки и перечисляет их уязвимости.

Цели ОО

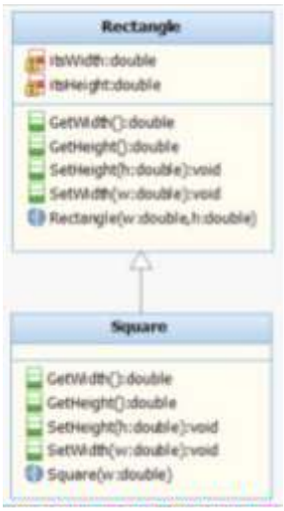
Две цели были включены в дополнение DO-332:

- A-7 OO.10 Проверка согласованности локальных типов (раздел OO.6.7.1)
- A-7 OO.11 Проверка надежности управления динамической памятью

Полезно понять принцип замещения Лискова по отношению к первому из них.
«Пусть q (x) - свойство, доказуемое об объектах x типа T. Тогда q (y) должно быть истинным для объектов y типа S, где S - подтип T»

В объектно-ориентированных языках наследование позволяет переопределить поведение суперклассов подклассами. Как описано в DO-332 FAQ # 14, обеспечение безопасного использования наследования, переопределения метода и динамической отправки является сложным, поскольку характер этих методов может сделать неясным из простого анализа, какой метод выполняется в любой точке вызова в программе. Переопределенное поведение в инстанцированных подклассах может изменить поведение, выходящее за рамки предполагаемой области суперкласса и согласования типа соответствия (см. Рисунок 14). Как описано в разделе DO-332 OO.6.7.1, это означает, что предварительные условия родительского класса не должны усиливаться, а постусловия и инварианты, определенные в состоянии класса, не должны быть ослаблены.

[13] В России – Р-332




```
virtual void Rectangle::SetWidth (double w)  {
      itsWidth=w;
}
    Post condition: itsHeight does not change

void Square::SetWidth (double w) {
     Rectangle::SetWidth(w);
     Rectangle::SetHeight(w);
}
    Post condition: itsHeight set to w
```

Рисунок 14. Родительский и дочерний классы, показывающие код, который нарушает согласованность типов

С точки зрения верификации DO-332 00.6.7.2 предлагает выполнить одно из следующих действий:

- Проверка взаимозаменяемости с использованием формальных методов.
- Необходимо убедится, что каждый класс проходит все тесты всех своих родительских типов, которые класс может заменить
- Для каждой точки вызова требуется проверить каждый метод, который можно вызвать в этой точке вызова (пессимистичное тестирование[14])

Первый из них относится к небольшому числу разработчиков, которые используют формальные методы, в то время как третий (как правило, называемый «плоским тестированием класса») требует, чтобы каждая возможная отправка проверялась в каждой точке вызова в программе. Это может легко вызвать «комбинаторный взрыв»[15] тестовых векторов, что резко увеличивает нагрузку на верификацию.

Это оставляет второй вариант наиболее практичным для большинства задач, так как позволяет обеспечить согласованность типов без затрат на пессимистичное тестирование. Для этого требуется, чтобы каждый класс и его методы проходили все тесты каждого суперкласса, для которого он может быть заменен (DO-332 FAQ № 34).

На рисунке 14 показано, что суперкласс Rectangle и его методы явно задают высоту и ширину с помощью соответствующих методов, но класс Square «закорачивает» их, устанавливая их в функции SetWidth.

Поскольку высота и ширина квадрата одинаковы, это кажется приемлемым. Однако, поскольку последовательность типов нарушена, тестовые вектора для метода SetWidth для класса Rectangle могут не передавать их для своего подтипа Square.

Повторное использование тестовых векторов из родительского класса Rectangle на подклассе Square укажет такое соответветствие типов (рисунок 15).

[14] https://arxiv.org/abs/0910.0996

[15] https://ru.wikipedia.org/wiki/Комбинаторный_взрыв

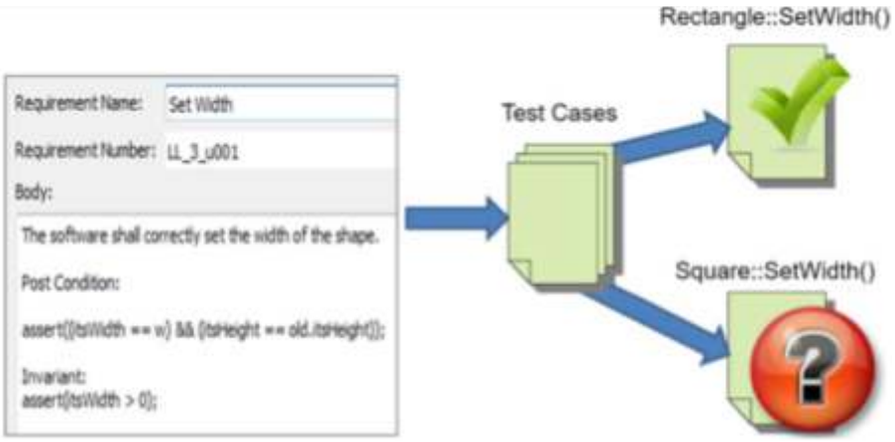


Рисунок 15. Тестовые вектора для класса Rectangle применяются к его подклассу Square, чтобы обеспечить согласованность локальных типов

Непройденый тестовый вектор для класса Rectangle показывает, что, как и ожидалось, установка ширины не влияет на высоту. Когда тот же тестовый вектор применяется к методу SetWidth класса Square, можно ясно видеть, что его значение изменяется, нарушая согласованность типов (рисунок 16).

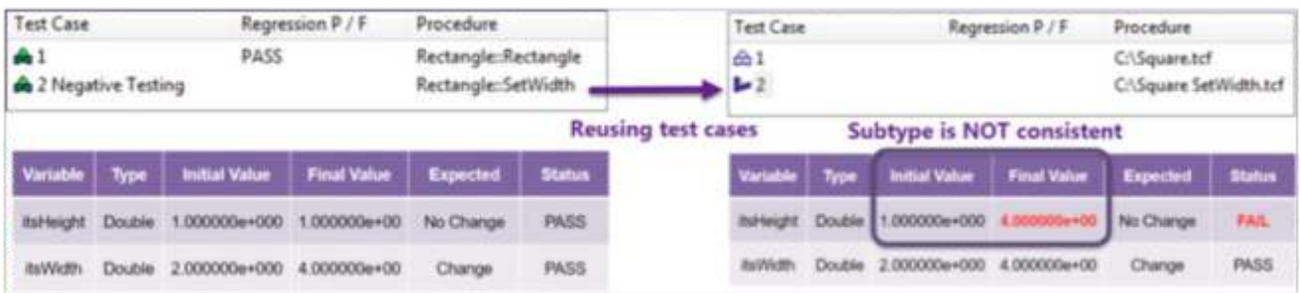


Рисунок 16. Переиспользование тестовых векторов родительского класса для проверки подкласса для обнаружения несогласованных подтипов

Для выполнения DO-332 A-7 00.6.8.1 может быть развернут ряд инструментов статического и динамического анализа, и связанные уязвимости разъясняются в Приложении 00.D.1.6. 1.

Отслеживание распределения памяти и освобождения памяти помогает обеспечить правильное освобождение памяти, а также связанные проверки перед разыменованием. Низкоуровневое тестирование предоставляет механизм для изучения различных сценариев распределения / освобождения, чтобы обеспечить устранение уязвимостей, описанных в 00.D.1.6.1. Временные хуки в рамках низкоуровневых тестов помогают характеризовать синхронизацию распределения / освобождения, а динамический анализ потока данных помогает отслеживать ссылки и обновления элементов данных во время выполнения, чтобы обнаруживать потерянные обновления и устаревшие ссылки.

Другие соображения

При использовании объектно-ориентированных технологий или связанных с ними методов необходимо учитывать и другие факторы:

Трассируемость между исходным и объектным кодом
Как упоминается в 00.D.1.2.1, установление трассируемости между исходным и объектным кодом может быть сложнее в объектно-ориентированных языках. Решения OCV обеспечивают графическое сравнение покрытия кода

ассемблера и покрытия языка высокого уровня (например, C ++), чтобы гарантировать, что данные о покрытии исходного кода учитывают изменения в структуре исполняемого объектного кода (EOC) по сравнению с исходным кодом.

Трассируемость для дочерних классов
В 00.5.2.2.i говорится: «Разрабатывайте иерархию классов, совместимую с локальными типами, с соответствующими требованиями низкого уровня, когда полагаетесь на замещение». Другими словами, требование, которое трассируется к методу, реализованному в классе, также трассируется на метод в своих подклассах, когда метод переопределяется в этом подклассе (FAQ # 9). Статический анализ и визуализация кода раскрывают отношения наследования в анализируемом коде, что облегчает обнаружение и устранение различий в трассируемости в иерархиях классов.

Стандарт кодирования для объектно-ориентированных языков
Такие языки, как C ++, обеспечивают огромную синтаксическую и семантическую гибкость. Стандарты, такие как MISRA C ++ 2008 и JSF AV ++, помогают быстро определить подмножество языков и лучшие практики, чтобы обеспечить основу для стандартов кодирования программного обеспечения, используемых в конкретных проектах.

Проблемы с структурным покрытием и низкоуровневым тестированием
Структурное покрытие деструкторов, создание сложных типов данных для тестирования, тестирование шаблонных классов и перегруженных операторов и доступ к частным членам - вот лишь некоторые из проблем, возникающих при работе с объектно-ориентированными технологиями или связанными с ними технологиями. Инструменты должны быть обеспечивать решение этих проблем и снижать затраты на верификацию, сохраняя при этом целостность / достоверность деятельности по верификации.

Разработка на основе моделей
Как и для объектно-ориентированных технологий, разработка на основе моделей (MBD) рассматривается в дополнение к DO-178C, называемому DO-331.[16]

DO-331 использует подход, в котором модели спецификации и модели проекта занимают место соответственно требований высокого и низкого уровня. Текстовые требования могут быть связаны с моделями (рис. 17).

Процессы, порождающие артефакты жизненного цикла	**Пример MB 1**	**Пример MB 2**	**Пример MB 3**	**Пример MB 4**	**Пример MB 5**
Процессы разработки требований и проектирования ПО	Требования к ПО	Требования по которым разработана модель	Требования по которым разработана модель	Требования по которым разработана модель	Требования по которым разработана модель
Процессы разработки требований и проектирования ПО	Требования по которым разработана модель	Модель спецификации	Модель спецификации	Модель проекта	Модель проекта
	Модель проекта	Модель проекта	Текстовое описание		
Процесс кодирования	Исходный код	Исходный код	Исходный код	Исходный код	Исходный код

Рисунок 17. Примеры использования модели[17]

[16] В России – Р-331

[17] Основаны на Таблице MB.1.1, DO-331

Популярные инструменты, такие как MathWorks® Simulink®[18], IBM® Rational® Rhapsody®[19] и ANSYS® SCADE[20], могут автоматически генерировать код. DO-331 MB.5.0 (Процессы разработки программного обеспечения) разъясняет вопросы трассируемости, типовых стандартов и многого другое как для требований к программному обеспечению, так и для процессов проектирования, в которых используются такие инструменты. MB.5.3 (Процесс кодирования программного обеспечения) является просто перекрестной ссылкой на эквивалентный раздел в DO-178C, подчеркивая тот факт, что наиболее эффективные мероприятия процесса кодирования, по-прежнему применяются, если код создается вручную на основе набора текстовых требований или на основе моделей, или автоматически генерируется с помощью инструмента.

Проекты, использующие автоматически сгенерированный код, почти всегда содержат некоторый ручной код и часто включают унаследованные компоненты, созданные вручную. Можно применять различные стандарты кодирования для этих различных подмножеств кода, такие как MISRA-C 2012 для ручного кода, MISRA-C 2012 Приложение E для автоматически сгенерированного кода и пользовательский стандарт кодирования для унаследованного кода.

DO-331 MB 6.0 (процесс верификации программного обеспечения) разъясняет лучшие практики применимые к MBD, а DO-331 MB.6.8.2 (Симуляции модели для проверки исполняемого объектного кода) разъясняет, какие цели верификации могут быть частично удовлетворены на уровне модели, и какие проверки должны быть выполнены на уровне целевого вычислителя.

Частичный зачет в модели

«Проверка исполняемого объектного кода в первую очередь выполняется путем тестирования. Этому может частично помочь сочетание симуляции модели и конкретного анализа ... Эта комбинация может быть использована для частичного удовлетворения следующих целей тестирования программного обеспечения».

Эти цели включают в себя соответствие EOC требованиям высокого уровня и низкого уровня, покрытие тестами структуры программного обеспечения, а также связи по данным и управлению. Для полного удовлетворения этих целей необходимо выполнить дополнительные действия по верификации на целевом оборудовании. В документе далее говорится, что, когда симуляции модели используются, чтобы частично удовлетворять целям верификации программного обеспечения и тестирования по требованиям высокого уровня, тогда необходимо обеспечить, чтобы одна и та же модель использовалась для генерации кода и для создания EOC.

В нем также указывается, что планы должны определять, какие требования и связанные с ними мероприятия по тестированию и сбору покрытия тестами должны выполняться на уровне модели и какие должны выполняться на целевом устройстве.

Требуется верификация на целевом вычислителе

DO-331 MB.6.8.2 гласит, что
«... конкретные тесты все равно должны выполняться в целевой среде ... Следующие цели верификации программного обеспечения и покрытия не могут быть удовлетворены симуляциями моделей, поскольку симуляции должны основываться на требованиях, из которых разработана модель».

Эти перечисленные цели включают в себя робастность EOC, соответствие требованиям низкого уровня и покрытие низкоуровневых требований тестами.

[18] https://uk.mathworks.com/products/simulink.html
[19] http://www.-03.ibm.com/software/products/en/ratirhapfami
[20] http://www.ansys.com/products/embedded-software/ansys-scade-suite

Далее добавляется информация о различных формах целей верификации, которые могут быть достигнуты только на целевом вычислителе, включая подтверждение совместимости с целевым оборудованием и тестирование интеграции оборудования и программного обеспечения. В нем также перечислены различные типы ошибок, которые могут или не могут быть обнаружены на уровне симуляций и могут быть обнаружены только на целевом оборудовании.

Наконец, DO-331 MB.B.11 (FAQ № 11) рассматривает вопросы деятельности покрытия модели:

«Анализ покрытия модели отличается от анализа структурного покрытия, и поэтому анализ покрытия модели не устраняет необходимость достижения целей анализа структурного покрытия в разделе 6.4.4.2 DO-178C».

Далее говорится, что анализ покрытия модели можно рассматривать в очень специфических сценариях «... в каждом конкретном случае и согласованных сертификационными органами ...»

В результате большинство организаций проводят некоторые проверки в рамках модели, но затем подтверждают результаты этих действий на целевом оборудовании, чтобы гарантировать, что они отвечают необходимым критериям для достижения целей.

Интеграция инструментов тестирования и моделирования упрощает этот процесс, используя статический анализ сгенерированного кода, сбор покрытия кода во время выполнения модели и перенос тестов модели в соответствующую форму для выполнения на целевом оборудовании (рисунок 17).

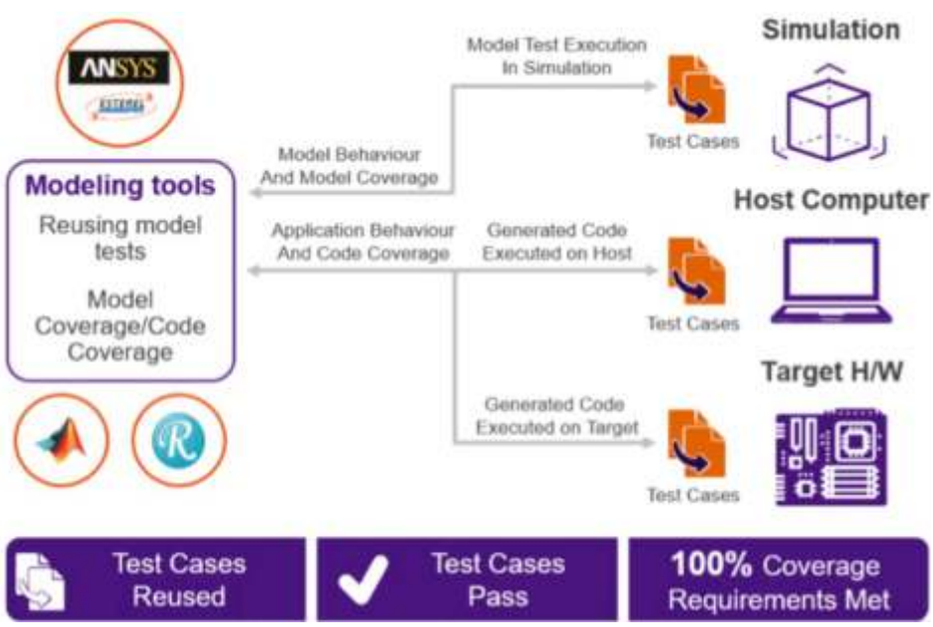


Рисунок 18. Перенос тестовых векторов из инструментов моделирования в набор инструментов LDRA для регрессионного тестирования на целевом вычислителе

Квалификация инструмента

Если программные средства предназначены для автоматизации значительного числа операций DO-178C при создании доказательных артефактов, показывающих, что цели были выполнены, важно обеспечить, чтобы на эти инструменты можно было положиться. DO-178C заявляет, что:
«Цель процесса квалификации инструмента заключается в том, чтобы гарантировать, что инструмент обеспечивает уверенность, по меньшей мере эквивалентную той, что относится к процессам этого документа, устранена, сокращена или автоматизирована».
Квалификация инструмента является важной частью процесса сертификации, и она задокументирована в расширении Software Tool Qualification Considerations (DO-330)[21].

DO-330 вводит концепцию уровня квалификации инструмента (TQL) на основе трех критериев:

1. Инструмент, выход которого является частью бортового программного обеспечения и, таким образом, может породить ошибку
2. Инструмент, который автоматизирует процессы верификации и, таким образом, может не обнаружить ошибку, и выход которого используется для обоснования устранения или сокращения:

- **a.** Процессов верификации, отличные от автоматизированных с помощью инструмента, или
- **b**. Процессов разработки, которые могут повлиять на бортовое программное обеспечение.

3. Инструмент, который в рамках своего предполагаемого использования может не обнаружить ошибку.

Там, где инструмент предназначен для использования в целях верификации, его выход не используется как часть бортового программного обеспечения, поэтому он не может вносить ошибки в программное обеспечение, что делает его инструментом, соответствующим критерию 3. Независимо от применения DAL, такому инструменту всегда назначается квалификационный уровень 5 (рисунок 19).

DAL	Критерий		
	1	2	3
A	TQL-1	TQL-4	TQL-5
B	TQL-2	TQL-4	TQL-5
C	TQL-3	TQL-5	TQL-5
D	TQL-4	TQL-5	TQL-5

Рисунок 19. Матрица уровней квалификации инструмента

Сертифицирующие органы, такие как FAA, CAA, JAA и ENAC, проводят квалификацию инструмента для каждого проекта отдельно, поэтому ответственность за демонстрацию пригодности любых инструментов зависит от организации, разрабатывающей ПО. Тем не менее, возможно использовать пакеты поддержки квалификации инструмента (TQSP), предоставляемые поставщиком инструмента. Такие пакеты обычно содержат ряд документов, начиная с требований к работе инструмента, которые идентифицируют процесс разработки, которому удовлетворяет инструмент, и включают в себя тестовые вектора, чтобы продемонстрировать, что инструмент работает со спецификацией в среде верификации.

[21] В России – Р-330

Квалификационная документация к инструменту должна быть указана в других документах планирования, и она играет ключевую роль в процессе соблюдения стандарта (рисунок 20).

Рисунок 20. Пакеты поддержки квалификации LDRA

Выбор инструмента

Использование инструментов для, управления тестированием, трассируемостью и инструментов статического и динамического анализа для проекта программного обеспечения авионики, отвечающего требованиям сертификации DO-178C, обеспечивает значительную производительность и экономическую выгоду. Инструменты упрощают проверку соответствия стандарту, уменьшают вероятность ошибок и повышают экономичность. Кроме того, они делают создание, управление, обслуживание и документирование трассируемости требований простыми и экономически эффективными. При выборе инструмента для содействия достижению целей DO-178C следует учитывать следующие критерии:

- Предоставляет ли инструмент полную сквозную трассируемость на протяжении всего жизненного цикла через требования, код, тесты, артефакты и цели?
- Предоставляет ли инструмент статический анализ для обеспечения соответствия ведущим в отрасли стандартам кодирования, таким как MISRA, CERT и другие?
- Включает ли инструмент анализ структурного покрытия целевого оборудования, как это изложено в разделе 6.4.4.2 стандарта, включая покрытие на уровне исходного и объектного кодов для проектов уровня A?
- Доступен ли инструмент для всех языков, платформ, компиляторов и целей, необходимых в проекте?
- Этот инструмент был успешно использован таким образом?
- Будет ли поставщик инструмента помогать в квалификации инструмента?
- Является ли поддержка инструмента гибкой и достаточно обширной для удовлетворения меняющихся требований?
- Является ли инструмент простым в использовании?

Использованные материалы

“Liskov's Substitution Principle”,
OODesign.com, http://www.oodesign.com/liskov-s-substitution-principle.html

“MISRA C:2012 - Guidelines for use of the C language in critical systems”
ISBN 978-906400-11-8 (PDF), Март 2013

“MISRA C++:2008 - Guidelines for the use of the C++ language in critical systems”
ISBN 978-906400-04-0 (PDF), Июнь 2008.

“EUROCAE DO-178B Software Considerations in Airborne Systems and Equipment Certification”,
EUROCAE Working Group 12 и RTCA Special Committee 167, December 10, 1992

“RTCA DO-178C Software Considerations in Airborne Systems and Equipment Certification”,
SC-205, Декабрь 13, 2011

“RTCA DO-330 Software Tool Qualification Considerations Techniques Supplement to DO-178C and DO-278A”,
SC-205, Декабрь 13, 2011

“RTCA DO-331 Model-Based Development and Verification Supplement to DO-178C and DO-278A”,
Prepared by SC-205, Декабрь 13, 2011

“RTCA DO-332 Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A”,
SC-205, Декабрь 13, 2011

+7 (495) 009-65-85
info@exponenta.ru
ldra.exponenta.ru