ПРОГРАММИРОВАНИЕ, 2022, № 4, с. 17–26

ПРОГРАММНАЯ ИНЖЕНЕРИЯ, ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ ПРОГРАММ

УДК 681.3.06

АНАЛИЗ СТРУКТУРНОГО ПОКРЫТИЯ ТОЧЕК ВХОДА И ВЫХОДА, НЕОБХОДИМЫЙ ДЛЯ ДОСТИЖЕНИЯ ЦЕЛЕЙ, ОПРЕДЕЛЕННЫХ В DO-178C

© 2022 г. В. П. Козырев[a,b,*]

[a] *Национальный исследовательский ядерный университет "МИФИ", 115409 Москва, Каширское ш., 31, Россия*

[b] *ООО "ЛаБС" (Advalange), 123001 Москва, ул. Садовая-Кудринская, д. 25, к. 4, Россия*

*****E-mail: vkozyrev@list.ru, Vladimir.Kozyrev@advalange.com*

Поступила в редакцию 18.10.2021 г.
После доработки 11.01.2022 г.
Принята к публикации 23.01.2022 г.

В число целей процесса верификации, определенных в документе RTCA DO-178C, входит анализ структурного покрытия программного обеспечения (ПО), включающий анализ покрытия структурных элементов исходного кода программ в соответствии с критериями SC, DC и MC/DC, а также анализ связности компонентов ПО по данным и по управлению. Критерии покрытия структурных элементов используются уже много лет (документ RTCA DO-178B опубликован в 1992 г.), однако их определение в DO-178B/C не является однозначным. В частности, для критерия DC не определены понятия точек входа, выхода и их покрытия, и разработчики инструментов сбора и анализа структурного покрытия (ИССП) определяют их по своему усмотрению. В статье сделана попытка устранения этой неоднозначности для программ, написанных на языках C/C++, и предложены решения, реализация которых в ИССП, по мнению автора, необходима с точки зрения достижения целей анализа структурного покрытия, определенных в DO-178C.

DOI: 10.31857/S0132347422040033

1. ВВЕДЕНИЕ

Анализ покрытия исходного кода является одним из методов верификации, применяемых при разработке ПО, требующего высокой надежности, в частности – авиационного ПО, разработка которого регламентируется документом RTCA DO-178C [1]. В этом документе указано, что "анализ структурного покрытия определяет, какая структура кода, включая интерфейсы между компонентами, не была выполнена тестовыми процедурами, основанными на требованиях", а также определены (в разделе 6.4.4.2) задачи и цели анализа структурного покрытия, в которые входят:

– анализ информации о структурном покрытии, собранной во время тестирования на основе требований, для подтверждения, что степень структурного покрытия соответствует уровню ПО, и

– анализ, подтверждающий, что при тестировании на основе требований была проверена связность по данным и по управлению между компонентами кода.

В [4] первая задача была названа анализом покрытия структурных элементов программы, а вторая – анализом покрытия связей, которая, в свою очередь, включает задачи анализа связей по управлению и анализа связей по данным. Будем использовать такую терминологию и в этой статье.

Из определения задач анализа структурного покрытия следует, что этот анализ должен выполняться на основе информации о прохождении потоков управления и данных в ПО при выполнении тестов, написанных по требованиям к нему. Сбор такой информации называется сбором структурного покрытия (или просто сбором покрытия), и обычно он выполняется с помощью специальных инструментов сбора структурного покрытия (ИССП), которые, кроме того, выполняют первичный анализ собранного покрытия на предмет его полноты в соответствии с актуальным критерием и формируют отчет о покрытии, анализируемый затем уже человеком. Разными производителями выпущено довольно много таких инструментов, предназначенных для достижения целей DO-178C в части анализа структур-

ного покрытия. Но поскольку DO-178C, как и его предшественник – DO-178B [2], не содержат ни исчерпывающих определений критериев структурного покрытия, ни их интерпретаций применимых для конкретных языков программирования, используемых при разработке ПО, эти критерии и их интерпретации доопределяются разработчиками инструментов, вследствие чего характеристики и функциональность существующих ИИСП в той или иной степени различны.

В DO-178C определено три критерия покрытия структурных элементов, применяющихся при анализе покрытия ПО в соответствии с его уровнем. Эти критерии применяются при анализе покрытия ПО уровней C, B и A соответственно:

– SC (Statement Coverage) – покрытие операторов,

– DC (Decision Coverage) – покрытие решений и

– MC/DC (Modified Condition/Decision Coverage) – модифицированное покрытие условий и решений.

Однако, пожалуй, лишь определение критерия покрытия SC ("*каждый оператор программы был вызван хотя бы один раз*") не вызывает особых вопросов, связанных с его реализацией и использованием в ИССП, чего нельзя сказать о критериях DC ("*каждая точка входа и выхода в программе пройдена хотя бы раз, а также каждое решение в программе приняло все возможные значения хотя бы один раз*") и MC/DC ("*каждая точка входа и выхода в программе пройдена хотя бы раз, каждое условие в решении в программе было выбрано при всех возможных исходах хотя бы раз, каждое решение в программе приняло все возможные значения хотя бы один раз, а также показано, что каждое условие в решении независимо влияет на исход данного решения*"), хотя они и используются в процессах верификации, проводимых в соответствии с DO-178B/C, уже много лет. Ни в DO-178C, ни в сопутствующих документах нет определений понятий точек входа и выхода, которые входят в определения критериев покрытия DC и MC/DC. Что же касается критериев покрытия связности – по управлению и, в особенности, по данным, то в DO-178C нет достаточно четкого определения методов их анализа, обеспечивающих достижение целей, объявленных в этом документе. Разъяснения относительно целей анализа покрытия связности представлены в документе RTCA DO-248C [3], однако вопрос о том, какую именно информацию необходимо иметь для достижения целей анализа покрытия связности, остается открытым. Другими словами, эти документы не определяют, что именно должен делать ИССП.

Попытка определения функциональности ИССП, необходимой для достижения целей, определенных в DO-178C, была предпринята в статье [4], где были рассмотрены проблемы анализа структурного покрытия исходного кода программ, написанных на языках C/C++, и фактически были определены основные требования к ИССП, которые необходимо рассматривать в контексте достижения этих целей. В [4] были определены три группы задач анализа структурного покрытия, обозначенных как: анализ покрытия структурных элементов – в соответствии с критериями SC, DC и MC/DC, анализ покрытия связей по управлению (Control Coupling) и анализ покрытия связей по данным (Data Coupling). Наиболее полно в статье [4] были рассмотрены проблемы анализа покрытия структурных элементов, однако некоторые проблемы, относящиеся к этой теме, в ней не были рассмотрены.

Одной из таких проблем является упомянутая выше проблема интерпретации понятий точек входа и выхода, входящих в определения критериев DC и MC/DC, которая была лишь обозначена в [4], и автор решил привести здесь свои соображения на эту тему (которые можно рассматривать в качестве дополнения к [4]). Автор попытался выделить точки входа и выхода в конструкциях языков C/C++, определить критерии их покрытия и предложить вариант представления результатов их покрытия пользователю ИССП. При этом некоторые из приводимых здесь выводов и рекомендаций, касающихся представления результатов покрытия, не могут быть строго аргументированы и подтверждены, например, ссылками на какие-либо регламентирующие документы, но делаются на основе личного опыта автора и его представлениях о целесообразности того или иного решения для достижения целей анализа покрытия, исходя из того, что интерфейс ИССП должен быть адекватен потребностям пользователя, обеспечивая получение пользователем всей необходимой для анализа покрытия информацией и минимизируя его затраты на проведение этого анализа (этот тезис поясняется ниже).

Так как способы представления данных о покрытии нигде не регламентируются, они определяются разработчиками инструментов, и на их решения могут влиять различные факторы. Среди них могут быть стереотипы, сложившиеся у разработчиков и пользователей инструментов за время применения практик анализа структурного покрытия (документ DO-178B был принят в 1992 г.), представления разработчиков о процессе сбора и анализа покрытия, включающие, в частности, аспекты, связанные с управлением этим процессом, или какие-то иные, в том числе субъективные, соображения. Вследствие этого информация, выдаваемая ИССП, может оказаться не адекватной потребностям пользователя. Например, некоторые из существующих ИССП выдают информацию о процентном соотношении покрытых и не покрытых структурных элементов программ. Возможно, данные о процентах по-

крытия могут иметь какое-то применение, но вот при планировании процесса анализа покрытия они, возможно, будут бесполезными, поскольку для оценки необходимых затрат ресурсов на анализ обнаруженных “дыр” в покрытии более важно знать их абсолютное, а не относительное количество.

При проектировании интерфейса следует учитывать, что в DO-178C для инструментов верификации, к которым относятся ИССП, предусмотрена их квалификация, целью которой является обеспечение гарантии того, что инструмент не пропустит ошибку в верифицируем объекте. Для ИССП это требование следует понимать так, что инструмент должен обнаруживать все не покрытые структуры ПО и сообщать о них пользователю. А поскольку анализ покрытия проводится человеком на основе данных, предоставляемых ему ИССП, следует иметь в виду, что избыточность в данных, выдаваемых инструментом, может как упростить, так и усложнить этот анализ. Например, если инструмент явно не сообщает о результате покрытия некоторого структурного элемента, но этот результат может быть вычислен по данным о покрытии каких-то других элементов, это придется делать пользователю, а если в отчете о покрытии будет слишком много данных о покрытых элементах, это может затруднить поиск данных о не покрытых элементах, и при неудачно выбранном способе представления данных пользователь может в этом случае просто их не заметить.

Однако в целом проблема проектирования интерфейсов ИССП является открытой. Автор считает, что ее исследование и решение позволили бы стандартизировать и за счет этого упростить для пользователя интерпретацию выдаваемых инструментами данных о покрытии и, кроме того, упростить процесс квалификации ИИСП, требуемый DO-178C.

Исследования проблем анализа структурного покрытия были начаты автором несколько лет назад совместно с М.А. Сабуровым, памяти которого автор хотел бы посвятить эту статью.

2. ПОКРЫТИЕ ТОЧЕК ВХОДА И ВЫХОДА

В определения критериев DC и MC/DC (включающего в себя критерий DC) входят понятия точек входа и выхода, однако определений этих понятий документ [1] не содержит. Краткое пояснение смысла этих понятий содержалось в документе DO-248B [5]: “... entry and exit points include low level entry and exits for decision and control constructs (e.g., IF-THEN-ELSE, DO WHILE, CASE) and such things as implicit ELSE conditions, CASE defaults, and implicit jumps”. Однако оно исчезло из следующей редакции этого документа – DO-248C [3], и для того, чтобы говорить о реализации сбора покрытия по критериям DC и MC/DC, необходимо определить, какие элементы в программах на языках C/C++, рассматриваемых в этой статье, следует считать точками входа и выхода.

Достаточно общим можно считать подход к определению понятий точек входа и выхода, связанный с организацией передачи управления в программе. В соответствии с этим подходом под точкой входа следует понимать точку в программе, в которую управление может быть передано в результате нарушения последовательного выполнения элементов исходного кода программы, в качестве которых в языках C/C++ могут рассматриваться операторы и выражения. Соответственно, точкой выхода следует считать точку, из которой выполняется такой переход. При этом следует рассматривать как переходы между функциями (к этой категории будем относить и методы классов в C++), выполняемые при вызовах функций и возвратах из них, так и переходы внутри функций, для организации которых в языках C/C++ могут использоваться, например, условный оператор, операторы цикла, оператор безусловного перехода и некоторые другие языковые конструкции. При таком определении покрытием точек входа/выхода следует считать прохождение потока управления через них, связанное с передачей управления, нарушающей последовательное выполнение элементов программы, при этом точку входа будем считать покрытой, если на нее было передано управление, а точку выхода – если при ее выполнении была выполнена такая передача управления.

В приведенном определении критерия DC можно выделить две части, на которые далее по тексту будем ссылаться как на первую и вторую соответственно. Первая часть определяет, что каждая точка входа и выхода в программе должна быть пройдена хотя бы раз (покрытие точек входа и выхода), а вторая – что каждое решение в программе приняло все возможные значения хотя бы один раз (покрытие логики). Далее по тексту понятие “покрытие решения” будет использоваться в контексте второй части критерия DC, при этом будем считать решение покрытым полностью (или просто покрытым), если в ходе тестирования оно приняло оба своих значения, покрытым частично, если оно приняло только одно значение – истина или ложь, и не покрытым, если оно ни разу не выполнилось.

Целесообразность выделения точек входа и выхода в тех или иных языковых конструкциях определяется целями анализа структурного покрытия, которые для ПО уровней A и B включают, в частности, оценку того, все ли операторы и ветвления в программе были выполнены в ходе ее

тестирования. Рассматривая в этом контексте первую часть определения критерия DC, можно обнаружить, что в языках C/C++ для управляющих конструкций, в которых ветвление выполняется на основе значений некоторого решения (логического выражения), покрытие этого решения, обеспечивает и покрытие соответствующих ветвей в этих конструкциях. Поэтому в таких конструкциях, например, в операторе if, нет смысла отдельно анализировать покрытие точек входа и выхода, соответствующих передачам управления по результату вычисления его условия: для оценки полноты тестов такой анализ не даст ничего нового по сравнению с анализом того, что каждое решение в программе приняло все возможные значения хотя бы один раз.

Аналогичный вывод можно сделать и относительно нецелесообразности отдельного рассмотрения точек выхода и входа, образующихся при использовании в программе операторов вызовов функций, поскольку анализ их покрытия обеспечивается анализом покрытия операторов по критерию SC: покрытие оператора вызова функции и оператора, следующего за ним, фактически означает покрытие соответствующих точек выхода и входа в вызывающей функции. Однако при вхождении вызовов функций в выражения, вычисляемые при выполнении программы, например, в логические выражения, которые вычисляются в языках C/C++ по короткой схеме, такой вывод уже сделать нельзя. Убедиться в выполнении всех присутствующих в программе вызовов функций можно было бы при анализе покрытия связности по управлению, однако при таком анализе должны рассматриваться связи между компонентами, каждый из которых в общем случае может включать множество функций, а передачи управления между функциями, входящими в один и тот же компонент, могут и не рассматриваться. Кроме того, поскольку сбор покрытия связности, как по данным, так и по управлению, должен проводиться при выполнении только интеграционных тестов, его вообще не следует рассматривать как возможный источник информации о покрытии структурных элементов, которое может собираться как по интеграционным, так и по модульным тестам. Следовательно, при наличии вызовов функций в логических выражениях необходимо убедиться, что каждый такой вызов был выполнен хотя бы однажды, а после его выполнения управление хотя бы однажды было возвращено из вызванной функции, и сообщать пользователю ИССП обо всех случаях невыполнения этого условия. Фактически здесь речь идет об уточнении определения критерия DC.

Таким образом, если прохождение через точку входа или выхода может быть вызвано ветвлением, которое не обнаруживается анализом покрытия операторов и логики – в соответствии со второй частью определения критерия DC, то анализ покрытия таких точек необходим. Например, такой анализ необходим для ветвей оператора switch, при использовании безусловных переходов по меткам, или когда выход из функции производится на оператор, не следующий сразу за оператором ее вызова, что возможно, например, при использовании механизмов setjump/longjump и try/catch. Однако явное выделение точек входа или выхода может оказаться целесообразным и в случаях, когда их покрытие может быть определено из покрытия операторов и логики, но дополнительная (избыточная) информация позволяет упростить пользователю анализ результатов покрытия, выдаваемых инструментом. Понятие простоты анализа будем рассматривать здесь неформально, оценивая полноту информации, необходимой для анализа покрытия, выводимой ИССП обычно в некотором отчете.

На основе приведенных рассуждений рассмотрим далее более подробно конструкции языков C/C++ на предмет необходимости или целесообразности выделения в них точек входа и выхода и отображения их в отчете о покрытии, выдаваемом инструментом. Не рассматривая конкретных вариантов возможных форм представления отчета о покрытии, будем считать, что в отчете должна содержаться информация о покрытии структурных элементов исходного кода, актуальная для выбранного критерия покрытия. И поскольку речь идет об анализе покрытия исходного кода, выводимая в отчете информация должна быть так или иначе ассоциирована с конкретными элементами или фрагментами этого кода. В связи с этим, при рассмотрении языковых конструкций будут предлагаться и возможные варианты “привязки” выделяемых в них точек входа и выхода к исходному коду, необходимой для выполнения анализа его покрытия. Разумеется, способы представления информации о покрытии могут быть различными, и при рассмотрении предлагаемых вариантов следует учитывать, что рассмотрение проблем организации интерфейсов ИССП не входит в цели настоящей статьи.

2.1. Функции

В контексте анализа покрытия точек входа и выхода специфика функций как структурных элементов программы состоит в том, что передачи управления возможны как в пределах одной функции, так и между разными функциями. Заметим, что при анализе покрытия по критериям DC и MC/DC имеет значение только сам факт передачи управления и не имеет значения, откуда или куда при этом управление передается. Поэтому здесь способы передачи управления между функциями нас будут интересовать в контексте определения элементов программы (операторов,

выражений), которые следует рассматривать как точки входа и выхода.

В языках C/C++ для передачи управления между функциями могут использоваться разные способы, основным из которых можно считать выполнение вызова функции и возврата из нее при выполнении оператора return или по достижению конца тела void-функции. Будем назвать такой способ, а также соответствующие ему точки входа и выхода функции регулярными. Неявные вызовы функций конструкторов, деструкторов и переопределений операторов в языке C++ также отнесем к регулярным. Кроме регулярных возможны и другие (нерегулярные) способы передачи управления между функциями, когда управление передается не в начало тела функции или когда возврат управления из функции выполняется в точку, расположенную не сразу после точки ее вызова. В этом разделе рассматриваются регулярные способы передачи управления, а нерегулярные – в следующем.

Чтобы обеспечить возможность анализа покрытия точек входа и выхода необходимо, прежде всего, выявить их при синтаксическом анализе программы, а затем по результатам сбора покрытия предоставить пользователю инструмента отчет об обнаруженных в программе точках входа/выхода и их покрытии. Выше отмечено, что для сокращения объема информации, которую придется обрабатывать пользователю при анализе покрытия, некоторые точки могут явно не выделяться в отчете. Например, рассмотрим следующий фрагмент кода:

int F(int x) { int y; y = f1(x); return y; }.

В этом примере анализ покрытия точек выхода и входа, соответствующих вызову функции f1(x), может быть выполнен на основе данных о покрытии операторов, причем результат покрытия точки выхода (т.е. покрыта она или нет) будет соответствовать результату покрытия оператора y = f1(x), а результат покрытия точки входа – результату покрытия следующего за ним оператора return, и явное указание в отчете результатов покрытия точек входа и выхода в этом случае можно считать излишним, так как их покрытие определяется покрытием операторов.

Усложним немного этот пример, добавив в оператор присваивания вызов другой функции:

int F(int x) { int y; y = f1(x) + f2(x); return y; }.

В этом примере покрытие обоих операторов в функции F будет означать и покрытие точек входа и выхода, соответствующих вызовам обеих функций – f1(x) и f2(x). Если же оператор присваивания окажется покрытым, а оператор return – нет, из этого факта будет следовать только то, что вызов f1(x) был выполнен и соответствующая ему точка выхода была покрыта, а точка входа для вызова f2(x) покрыта не была, так как оператор return может оказаться не покрытым, если не будет выполнен регулярный выход из любой из двух функций – f1 или f2.

Внесем еще одно изменение в рассматриваемый пример, сделав выражение в операторе присваивания логическим:

int F(int x, int y) {return x && (y || f(x)); }.

Вызов функции f(x) в выражении x && (y || f(x)) будет выполнен, если значение параметра x функции F будет не равно нулю, а значение параметра y окажется нулевым, однако для покрытия этого выражения по критерию DC достаточно двух наборов параметров функции F: {x = 0} и {x = 1; y = 1}, и в этом случае вызов функции f(x) не будет выполнен, хотя решение x && (y || f(x)) будет покрыто.

В языке C++ кроме явных вызовов возможно выполнение и неявных вызовов функций – конструкторов, деструкторов и переопределений операторов. Ниже приведен пример программы, в которой выполняются такие неявные вызовы. Строки кода, в которых происходят неявные вызовы функций, помечены соответствующими комментариями.

```
1 class A
2 {
3   int x;
4 public:
5   A(int i = 0) { x = i; }                  // Конструктор по умолчанию
6   A(const A& z) { x = z.x + 10; } // Конструктор копирования
7   ~A()       { x = 0; }                    // Деструктор
8   const A operator+(A& a)
9     { return A(const x + a.x);}     // вызов конструктора по умолчанию
10  A operator-()
11     { return A(-x); }                    // вызов конструктора по умолчанию
12 };
13
```

```
14 A a1;                    // вызов конструктора A(0)
15
16 void F(const A& x)
17 {
18   A a2(1);               // вызов конструктора A(2)
19   A a3 = -a2;            // вызов функции перегрузки унарного минуса
20   A* a4 = new A;         // вызов конструктора A(0)
21   A a5 = x;              // вызов конструктора копирования
22   A a6 = *a4;            // вызов конструктора копирования
23   delete a4;             // вызов деструктора для переменной a3
24   A a7 = a1 + 1;         // вызов конструктора A(1), перегрузки бинарного
25                          // плюса и деструктора для константы A(1)
26 }                        // вызов деструктора для переменных a7, a6, a5, a3, a2
27
28 int main()
29 {
30   F(5);                  // вызов конструктора константы A(5), функции F и
31                          // деструктора для константы A(5)
32 }                        // вызов деструктора для переменной a1
```

Даже в таком, весьма простом, примере выполняется довольно много неявных передач управления, и можно предположить, что в более сложных программах неявных передач управления будет намного больше, причем некоторые из них могут оказаться неочевидными с точки зрения человека, выполняющего анализ покрытия. Например, в строках 19 и 24 выполняется вызов функций переопределяющих операторы “–” и “+”, из которых, в свою очередь, вызываются конструкторы по умолчанию, а непосредственно в строках 19 и 24 вызовов конструкторов не происходит (такие результаты были получены при выполнении этого примера в среде MS Visual Studio).

Из рассмотрения приведенных примеров можно сделать вывод о том, что в случаях, когда точки входа или выхода, соответствующие явным или неявным вызовам функций, оказываются не покрытыми, отображение этих фактов в отчете о покрытии необходимо. Если же точки входа или выхода оказываются покрытыми, явное представление их в отчете о покрытии представляется избыточным, поскольку основной целью анализа структурного покрытия следует считать доказательство отсутствия не покрытых структурных элементов, а не их перечисление. Но в любом случае, покрыта точка или нет, возникнет вопрос о том, каким образом представить ее в отчете. Например, в приведенном выше фрагменте кода точки входа и выхода, соответствующие явным вызовам функций и перегруженных операторов, можно ассоциировать непосредственно с их представлениями в коде, в частности, с символами “F”, “–” и “+” в строках 30, 19 и 24, а неявные вызовы конструкторов и деструкторов – со строками кода, отмеченными соответствующими комментариями. Однако решение этого вопроса не входит в цели настоящей статьи.

Идентификация точек входа в тело функции и выхода из нее представляется более простой задачей, чем идентификация точек входа и выхода ее вызовов. В качестве регулярной точки входа в функцию естественно рассматривать какую-то точку в ней, выполняемую перед первым исполняемым оператором в теле функции. А так как объектный код, выполняемый при входе в функцию, при анализе покрытия исходного кода нас не интересует, точку входа, в принципе, можно ассоциировать с любым структурным элементом определения функции, расположенным текстуально выше первого исполняемого оператора ее тела, и наиболее подходящим, в плане универсальности, представляется отнесение этой точки к фигурной скобке, задающей начало тела функции. Соответственно, при выводе в отчете информации о ее покрытии будут использоваться координаты этой скобки.

Так как регулярные выходы из функции производятся при выполнении операторов return, а для void-функции – и при завершении выполнения ее тела, соответствующие точки выхода естественно ассоциировать с операторами return и фигурной скобкой, завершающей тело функции. При этом следует учитывать, что если в конце тела void-функции, непосредственно перед завершающей ее тело фигурной скобкой, расположен оператор return, то нет смысла рассматривать эту скобку в качестве отдельной точки выхода. В этом случае, вероятно, будет целесообразно объединить ее с предшествующим ей оператором return.

Аналогичное решение можно предложить и для функций, возвращающих значение, поскольку некоторые компиляторы допускают отсутствие операторов return или наличие исполняемых операторов, отличных от return, перед завершающей фигурной скобкой в таких функциях.

Конечно, предлагаемое сопоставление точек входа и выхода с фигурными скобками, ограничивающими тело функции, основывается на субъективном представлении автора о наглядности разметки текста, но в его пользу можно привести такой аргумент, что образующаяся при этом симметрия может упростить восприятие пользователем информации об их покрытии.

2.2. Нерегулярные способы передачи управления

Нерегулярная передача управления может выполняться как внутри одной функции, так и между разными функциями. В частности, это возможно при использовании библиотечных функций setjmp, longjmp и exit, определенных в стандартах языка C. В программах на языке C++ нерегулярные способы передачи управления могут реализовываться и при организации обработчиков исключений с использованием операторов try, catch и throw, причем в этом случае управление на обработчик catch может передаваться как из блока try, так и из любой функции, вызванной из него или находящейся в цепочке вызовов, начинающейся в нем. Кроме того, с использованием ассемблера могут быть реализованы и другие способы передачи управления, однако ограничимся здесь рассмотрением только тех методов, которые определены в стандартах языков C/C++.

Рассматривая функции setjmp, longjmp и exit в контексте достижения целей, определенных в [1], можно заметить, что хотя эти функции и определены в стандарте языка C, они обычно реализуются в библиотеках времени выполнения, которые в случае их использования в бортовом ПО должны рассматриваться как его части, и для их верификации должны применяться общие методы, определенные в [1]. Тем не менее, вызовы этих функций целесообразно рассматривать как точки входа (setjmp) и выхода (longjmp и exit), поскольку на уровне исходного кода программ на C/C++, в которых используются эти вызовы, они представляют собой расширение языка, реализующее соответствующие передачи управления, что делает необходимым регистрацию их покрытия с предоставлением пользователю (в отчете о покрытии) соответствующей информации.

Использование операторов try/catch вносит свою специфику в определение связанных с ними точек выхода, поскольку переход на блок catch может происходить как при выполнении операторов throw, так и при возникновении исключений при выполнении программы внутри блока try, например, при делении на ноль. При использовании операторов throw их естественно рассматривать как точки выхода из блока try, но точки, в которых может возникнуть исключение, вызывающее переход к блоку catch, обнаружить при синтаксическом анализе программы, как правило, невозможно. Да и рассматривать все потенциальные точки выхода из блока try в качестве предмета для анализа их покрытия вряд ли целесообразно, поскольку при тестировании программы вряд ли будет целесообразным (а в общем случае и вряд ли возможным) проверять обработку исключений для каждой такой точки. Скорее всего, при тестировании будет достаточным обеспечить проверку переходов к обработчикам исключений catch из некоторых точек в блоке try или просто проверить, что переходы на блоки catch выполняются.

Руководствуясь приведенными рассуждениями, можно предложить ассоциировать все потенциальные точки возникновения исключений с одной точкой выхода из соответствующего блока try, связав ее, например, с концом блока try – закрывающей фигурной скобкой, но отличая их от регулярной точки выхода, выполняемой по завершению блока try. Но если “срабатывание” каких-либо конкретных точек выхода в программе при возникновении исключений предусмотрено тестами, то анализ их покрытия следует считать необходимым, и для этого в ИССП необходимо предусмотреть возможность их идентификации. Например, это можно сделать путем аннотирования анализируемого исходного кода. Инструмент же в этом случае должен будет отслеживать возникновение исключений в каждой из таких точек по отдельности. Хотя такое решение можно считать субъективным, доводом в его пользу является то, что альтернативой ему будет рассмотрение всех операторов и выражений в блоке try в качестве потенциальных точек выхода, но большинство из них могут оказаться не покрытыми при выполнении тестов, что усложнит анализ полученного покрытия, поскольку пользователю придется анализировать и описывать причины отсутствия покрытия для каждой такой точки.

Что же касается точек входа, связываемых с передачами управления из блока try, то естественно считать такими точками входы в блоки catch (соответствующие открывающие фигурные скобки), а также первый оператор после последнего блока catch группы try/catch, а если его нет, то фигурную скобку, завершающую блок, в который входит try. Соответствующим образом операторы, обеспечивающие нерегулярные передачи управления, могут отображаться и в отчетах о покрытии, выдаваемых пользователю.

2.3. Безусловные переходы по меткам

Безусловные переходы по меткам организуются с использованием операторов goto и меток, при этом оператор goto соответствует точке выхода, а метка – точке входа. Поскольку выполнение операторов goto и, соответственно, меток, по которым они передают управление, может быть обнаружено при анализе покрытия операторов (SC), их выделение в качестве точек входа/выхода в отчетах о покрытии не является обязательным. Однако если не выделять в качестве точек входа метки, анализ их покрытия усложнится, так как потребуется просмотр всего кода функции для поиска операторов goto, обращающихся к ним. Поэтому целесообразно отображать покрытие меток как точек входа в отчетах о покрытии по критериям DC и MC/DC, а операторы goto отображать и как операторы (statements), и как точки выхода.

2.4. Условные операторы

При выполнении операторов if и условной операции (“?”) управление может передаваться на начала их ветвей (true и false), а также на следующий оператор в коде программы – по завершении ветви true при наличии ветви false, или при отсутствии ветви false (некоторые компиляторы позволяют использовать т.н. бинарные условные операции, в которых нет ветви false). Как отмечено выше (в 2), явное выделение точек входа и выхода, формируемых условными операторами, не является обязательным, поскольку их покрытие полностью определяется покрытием логики. Исходя из этого, можно считать не обязательным и отображение их в отчетах о покрытии.

Однако поскольку выбор интерфейсов ИССП остается, все же, за разработчиками инструментов, возможен и альтернативный подход. Например, в инструменте может быть предусмотрена выдача статистики по всем точкам входа и выхода, и потребуется явное их указание для того, чтобы пользователь смог проверить выданные в отчете данные об их покрытии. Возможна также и реализация в инструменте обоих подходов с выбором пользователем нужного ему представления данных о покрытии.

2.5. Оператор switch

В операторе switch имеет смысл рассматривать в качестве точек входа метки case и default, в которые управление попадает после вычисления значения выражения в заголовке оператора switch. Условно можно считать, что соответствующие точки выхода реализуются в заголовке. В случаях, когда метка default отсутствует, управление из заголовка может передаваться на оператор, следующий за switch, а поскольку такого оператора может и не оказаться, в таких случаях точку входа можно отнести к завершению тела оператора switch, которым может являться скобка “}”, если тело оператора является составным оператором (блоком), или разделитель “;” – в противном случае (стандарт языка С допускает использование в качестве тела switch оператора, не являющегося составным). Несмотря на некоторую искусственность такого решения, оно позволяет избежать проблем сопоставления точек входа с конкретными элементами исходного кода при выводе отчета о покрытии, что может облегчить его восприятие пользователем ИССП.

В качестве точки выхода в операторе switch может выступать оператор break, передающий управление оператору, следующему за switch. Поскольку его покрытие определяется покрытием операторов (SC), его представление в качестве точки выхода не является обязательным, однако, как и для условных операторов (2.4), возможна реализация в ИССП и альтернативного подхода.

Рассматривать же точки выхода, реализуемые в заголовке оператора switch нет смысла, поскольку, во-первых, их невозможно выделить в коде, а во-вторых, каждая такая точка выхода и соответствующая ей точка входа в операторе switch покрываются одновременно. Таким образом, в операторах switch целесообразно отображать в качестве точек выхода операторы break, а в качестве точек входа – операторы case и default, а также скобку “}” или разделитель “;” – в зависимости от конкретного формата оператора switch. В частности, символы “}” или “;” имеет смысл представлять как точки входа только в тех случаях, когда в операторе switch отсутствует метка default. В противном случае эти символы, (так же, как и символ “{“ в начале составного оператора) нет смысла отображать в отчете о покрытии как точки входа/выхода. Исключение составляет случай, когда телом switch является пустой составной оператор (“{}”). В этом случае его целесообразно представлять как непокрытый оператор.

2.6. Операторы цикла

В операторах цикла целесообразно рассматривать неявные точки входа и выхода, связанные с ветвлением программы, которое выполняется для организации цикла (переход на начало тела цикла при выполнении его очередного шага, переход на изменение параметров цикла или проверку условия после очередного выполнения тела цикла, выход из цикла при невыполнении условия цикла). Кроме них в теле оператора цикла могут присутствовать явные точки выхода, порождаемые операторами безусловного перехода – как контекстно связанными с ним (break и continue), так и нет (return, throw, goto и др.). При рассмотрении циклов будем учитывать из них только опе-

раторы break и continue (остальные же операторы рассматриваются в других разделах). Поскольку покрытие этих явных точек выхода определяется покрытием операторов (SC), может быть, например, реализовано их опциональное представление – как точек выхода или просто как операторов. Рассмотрим специфику операторов цикла, определенных в языках C/C++.

2.6.1. Оператор while

В операторе while точки входа реализуются в условии цикла и в начале тела цикла. Точки выхода реализуются в условии и в конце тела цикла, а также могут явно определяться в теле оператора цикла операторами перехода. На основе рассуждений, приведенных в 2, можно сделать вывод о том, что:

– отображать точки входа и выхода в заголовке цикла не нужно;

– для тела цикла, являющегося составным оператором, целесообразно отображать в качестве точек входа и выхода обрамляющие его скобки “{“ и ”}” соответственно;

– если телом цикла является одиночный оператор, целесообразно отображать в качестве точки выхода завершающий его разделитель “;”.

2.6.2. Оператор do

В операторе do (do-while), также, как и в операторе while, точки входа реализуются в условии цикла и в начале тела цикла. Точки выхода реализуются в условии и в конце тела цикла, а также могут явно определяться в теле цикла операторами break и continue. На основе рассуждений, приведенных в 2, так же можно сделать вывод о том, что:

– отображать точки входа и выхода в заголовке цикла не нужно;

– для тела цикла, являющегося составным оператором, целесообразно отображать в качестве точек входа и выхода обрамляющие его скобки “{“ и ”}” соответственно;

– если телом цикла является одиночный оператор, целесообразно отображать в качестве точек входа и выхода ключевые слова do и while соответственно.

2.6.3. Оператор for

Оператор for может быть представлен в “классическом” формате, определенном в стандарте языка C, или же в формате “с диапазоном” (range-based), который может использоваться в языке C++. В классическом варианте оператор имеет вид **for** (⟨инициализация⟩; ⟨условие⟩; ⟨модификация⟩) ⟨тело цикла⟩. В формате с диапазоном оператор выглядит следующим образом: **for** (⟨параметр цикла⟩:⟨диапазон⟩) ⟨тело цикла⟩.

Рассмотрим точки входа и выхода, которые ассоциируются с возможными ветвлениями потока управления, происходящими при выполнении оператора. Точки входа реализуются в заголовке, в выражениях ⟨условие⟩, ⟨модификация⟩ и ⟨диапазон⟩, а также в начале тела цикла. Точки выхода реализуются в заголовке – в выражениях ⟨условие⟩, ⟨модификация⟩ и ⟨диапазон⟩, в конце тела цикла, а также могут явно определяться в теле цикла операторами break или continue.

Отображение в отчетах о покрытии точек входа, реализуемых в заголовках циклов, не представляется целесообразным, поскольку их покрытие определяется покрытием логики и операторов соответственно. Точкой выхода для выражения ⟨модификация⟩ всегда является переход на ⟨условие⟩, поэтому ее покрытие также определяется покрытием операторов. Точками выхода для выражений ⟨условие⟩ и ⟨диапазон⟩ являются переходы к телу цикла или выход из цикла. Покрытие точек выхода из условия в цикле for “классического” формата также определяется покрытием решения ⟨условие⟩: если решение не покрыто полностью, то соответствующие точки выхода (true и/или false) следует считать не покрытыми.

Циклы с диапазоном (range-based) с точки зрения логики передач управления для организации цикла можно считать аналогичными циклам while, где в роли условия выступает ⟨диапазон⟩. Хотя ⟨диапазон⟩ и не является логическим выражением, при представлении результатов его покрытия по критериям DC и MC/DC целесообразно рассматривать его как некий аналог решения и считать его:

– не покрытым, если множество, определяемое выражением ⟨диапазон⟩, пустое и тело цикла не выполняется ни разу;

– покрытым полностью, если цикл завершается по исчерпанию множества, определяемого выражением ⟨диапазон⟩;

– покрытым частично, если тело цикла выполняется, но при его выполнении на некоторой итерации происходит выход из цикла.

При этом покрытие точек входа, соответствующих условию цикла, началу тела цикла и завершению цикла, а также точек выхода, соответствующих переходам на эти точки входа, будет определяться покрытием выражения ⟨диапазон⟩.

Для отображения точки входа в тело цикла и точки выхода, из которой управление попадает на проверку условия, в случаях, когда тело цикла является составным оператором, наиболее очевидным способом представляется представление в качестве точки входа в тело цикла его начальную скобку “{“, а в качестве точки выхода – завершающую скобку “}”. Если же тело цикла является одиночным (не составным) оператором, то в качестве представления точки выхода можно использовать завершающий его символ “;”. В отображении же точки входа, которой является сам

этот оператор, нет необходимости, поскольку ее покрытие определяется покрытием условия цикла.

3. ЗАКЛЮЧЕНИЕ

Приведенные в статье соображения позволяют, по мнению автора, более точно понять условия, необходимые для достижения целей, определенных в DO-178C в части анализа покрытия структурных элементов (в терминологии [4]), и сформулировать требования, которые должны учитываться при разработке ИССП, обеспечивающих возможность достижения этих целей. На основе соображений, приведенных в [4] и в настоящей статье, была сделана попытка определить и реализовать такие требования в ИССП, получившем название COVERest [6]. В ходе работ по созданию этого инструмента были продолжены исследования проблем анализа покрытия связности по управлению и данным, однако эти исследования пока не закончены, и автор надеется, что у него будет возможность опубликовать результаты этих работ по их завершению.

СПИСОК ЛИТЕРАТУРЫ

1. RTCA/DO-178C. Software Considerations in Airborne Systems and Equipment Certification. RTCA Inc., 2011.
2. RTCA/DO-178B. Software Considerations in Airborne Systems and Equipment Certification. RTCA Inc., 1992.
3. RTCA/DO-248C. Supporting Information for DO-178C and DO-278A. RTCA Inc., 2011.
4. *Kozyrev V.P., Saburov M.A.* Satisfying DO-178C Structural Coverage Objectives // Programming and Computer Software. 2018. V. 44. № 1. P. 43–50.
5. RTCA/DO-248B. Final report for clarification of DO-178B. RTCA Inc., 2001.
6. COVERest: инструмент автоматизации анализа структурного покрытия кода. URL: https://gosniias.ru/coverest.html (дата обращения: 18.10.2021).