УДК 519.685/519.686

А.А. ЛЕТИЧЕВСКИЙ[*] (мл.)

ЭКСПЕРИМЕНТЫ С ДЕДУКТИВНЫМ ТЕСТИРОВАНИЕМ РЕАКТИВНЫХ СИСТЕМ

[*]Институт кибернетики им. В.М. Глушкова НАН Украины, Киев, Украина

Анотація. *Застосування дедуктивних методів у тестуванні можливе при визначенні відповідності програмного забезпечення вихідним вимогам. Передбачається, що програмний код розробляється за вимогами, які можуть бути задані в деякій формальній мові. Техніка дедуктивного тестування з використанням символьного виконання програми дозволяє отримати найбільш повне покриття спостережуваних даних. У даній статті розглядається приклад дедуктивного тестування реактивної системи.*
Ключові слова: *автоматизація тестування, формальна верифікація, інженерія вимог, формальні методи, автоматизація доказів.*

Аннотация. *Применение дедуктивных методов в тестировании возможно при определении соответствия программного обеспечения исходным требованиям. Предполагается, что программный код разрабатывается по требованиям, которые могут быть заданы в некотором формальном языке. Техника дедуктивного тестирования с использованием символьного выполнения программы позволяет получить наиболее полное покрытие наблюдаемых данных. В данной статье рассматривается пример дедуктивного тестирования реактивной системы.*
Ключевые слова: *автоматизация тестирования, формальная верификация, инженерия требований, формальные методы, автоматизация доказательств.*

Abstract. *The application of deductive methods in testing technology is possible at definition of equivalence of software product to initial requirements. It is anticipated that programme code is developed due to the requirements that could be presented as formal specifications. Deductive testing technique by using symbol programme execution allows obtaining the most complete coverage of all visible data. The example of deductive testing of reactive system is considered in this paper.*
Keywords: *automation of testing, formal verification, requirements engineering, formal methods, proving automation.*

1. Введение

Современные технологии тестирования программного и аппаратного обеспечения в основном базируются на автоматизации тестирования методом «черного ящика». Сам «черный ящик» представляет собой модель или готовое изделие программно-аппаратного комплекса, а технология тестирования заключается в изучении его поведения или реакций, полученных как ответы на внешние воздействия. Существует также проблема надежности тестирования, в которой изучаются вопросы адекватности «черного ящика» требуемым свойствам создаваемой программы.

Мы рассматриваем тестирование программного кода реактивных систем, то есть систем, которые многократно получают от внешней среды входные воздействия, а в среду возвращают реакции. Взаимодействие со средой происходит в виде обмена сообщениями или данными. Если в требованиях к системе встречаются временные ограничения, то такая система относится к классу систем реального времени.

Как математический объект реактивная система представляет собой транзиционную систему, действия которой подразделяются на входные воздействия и реакции. В одном действии могут быть также совмещены входные воздействия и реакции, даже конечные последовательности таких действий.

Стандартная технология тестирования представляет собой, в первую очередь, описание функциональных спецификаций, которые содержат описание реакций системы на

входящие воздействия, а затем создание тестового набора, который соответствует описанным реакциям. Такой тестовый набор может быть создан автоматически, если функциональные спецификации или требования к системе представлены в виде формальных спецификаций. Существует ряд инструментов, которые позволяют автоматически сгенерировать тестовый набор, удовлетворяющий некоторому покрытию [1]. Под покрытием понимают некоторый критерий для оценки качества тестирования, такой, как покрытие требований, покрытие данных, а также покрытие строк кода.

Тестирование методом «черного ящика» происходит с помощью контроля конкретных значений атрибутов системы, которые наблюдаются в качестве реакции системы на некоторый набор конкретных воздействий. Полученные значения автоматически сравниваются с прогнозируемыми, и делается вывод о прохождении теста.

Таким образом, для обнаружения всех ошибок в программном коде системы необходимо проверить все комбинации всех конкретных значений входных данных, что для большинства систем сделать невозможно. Поэтому применяют выборочные данные, соответствующие некоторому разумному выбору, например, использование граничных значений, эквивалентных разбиений.

Такие методы также не дают гарантии обнаружения всех ошибок и адекватности свойств программного кода системы исходным требованиям, так как даже при разумном выборе может быть пропущено некоторое критическое состояние. В данной работе предлагается использовать дедуктивный метод тестирования, где в тестах с целью достижения полного покрытия данных рассматриваются не конкретные значения, а их множества. Для этого используется символьное моделирование формальных спецификаций, в частности, программного кода реактивной системы.

2. Тестирование программного кода реактивных систем с применением формальных методов

Методы автоматизации тестирования разрабатываются в соответствии с моделью процесса разработки программного обеспечения. В качестве исходных требований рассматривается некоторое описание будущей системы на уровне абстракции, соответствующей пониманию заказчика.

Изначально выполняется процедура формализации исходных требований, а также их формальная верификация с помощью специальных инструментов. Далее процесс разработки происходит согласно следующей схеме (рис. 1).

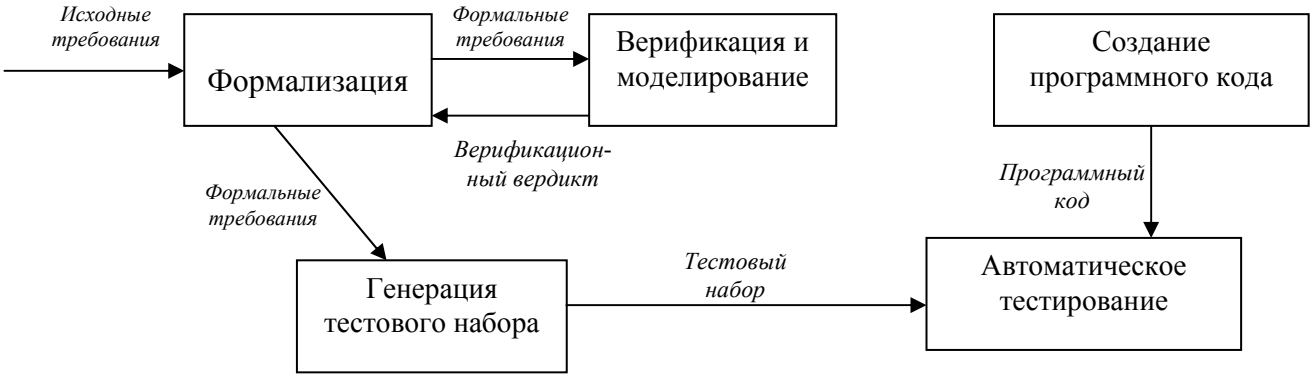

Рис. 1. Процесс разработки

Формальные требования в языке спецификаций, подходящем для верификационных инструментов, представляют модель будущей системы на некотором уровне абстракции.

Здесь нужно зафиксировать уровень абстракции информации, в частности, описание входных и выходных данных, который соответствовал бы и требованиям, и программному коду системы. В этом случае можно автоматически сгенерировать тестовый набор, который был бы применим в тестировании готовой системы.

Программист создает код системы согласно верифицированным требованиям. Он может использовать исходное описание требований, а также и формальные требования. Существует технология, когда из формальных требований возможно сгенерировать программный код автоматически. Полученный код будет соответствовать уровню абстракции требований. Если этот уровень недостаточен, то проводится уточнение программного кода или же понижение уровня абстракции. В этом случае схема выглядит следующим образом (рис. 2).

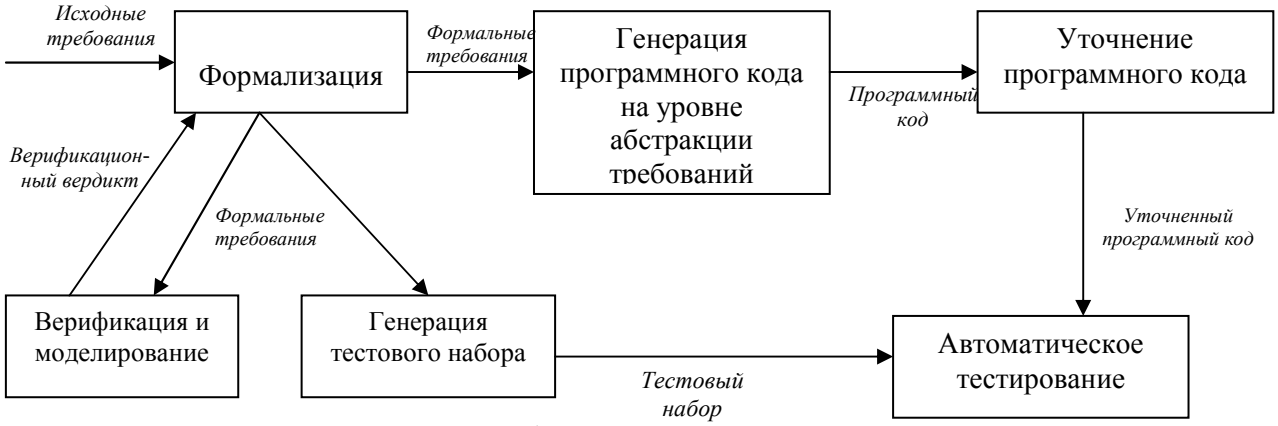

Рис. 2. Процесс разработки, включающий генерацию кода

Автоматическое тестирование представляет собой процедуру, в которой входом являются исходный программный код системы и некоторый тестовый набор, представляющий последовательности реакций системы в соответствии с входными воздействиями.

Для поддержки традиционного метода тестирования «черного ящика» тестовый набор представляет последовательность входных воздействий на программный код и множество выходных реакций, которые сравниваются с выходными реакциями программного кода. При этом программный код непосредственно выполняется или моделируется.

Использование же символьных методов позволит работать с множествами наборов входных данных. Так, например, на вход программного кода поступает не конкретное значение некоторой переменной, а формула, которая описывает множество значений этой переменной. В этом случае программный код моделируется и на выходе получаем также формулу, которая описывает множество значений выходных данных или некоторое символьное состояние программы. Сравнивая полученную формулу с ожидаемой, можно сделать вывод об адекватности поведения программы изначальным требованиям. Далее мы рассмотрим подробнее эту технологию, определив более точно каждый этап.

3. Формализация требований и генерация тестового набора

Требования к реактивным системам представляют собой описания реакций системы на внешние воздействия. Эти реакции содержат изменение среды системы и выходные сообщения. Для формализации таких требований мы будем использовать язык базовых протоколов, синтаксис и семантика которого описаны в [2], а примеры и методика формализации в [3].

Базовый протокол представляет собой описание реакции системы на внешнее воздействие, а также изменение среды системы, которая определяется ее атрибутами.

Базовый протокол содержит предусловие, в котором представлено триггерное событие, а также условия, при которых оно происходит. Предусловие является формулой в некотором базовом логическом типизированном языке, содержащем арифметические и логические операции.

Базовый протокол также содержит постусловие, в котором могут быть императивные операторы присваивания и формула в базовом языке.

Таким образом, базовый протокол представляет реакцию системы как некоторый переход транзиционной системы, который изменяет ее состояние.

Мы различаем конкретные и символьные модели требований к системе. Состояние конкретной модели требований определяется конкретными наборами значений атрибутов системы. Состояние символьной модели представляет собой формулу в базовом языке модели.

Рассмотрим пример требований к системе, которая моделирует работу некоторого станка по изготовлению неких изделий.

R1. Станок-автомат должен принимать на вход заготовки двух типов. Станок должен состоять из двух встроенных внутренних механизмов, один из которых воспринимает на вход заготовки типа *A*, а другой типа *B*.

R2. Заготовки типа *A* поставляются на вход в кассетах по 20 единиц, а типа *B* по 16 единиц. Новая кассета поставляется на вход в начале дня, если предыдущие заготовки исчерпаны.

R3. Максимальная производительность станка 10 изделий в день.

R4. Выходом станка являются изделия, собранные из деталей типа *A* и деталей типа *B*, которые изготавливаются встроенными механизмами соответственно из заготовок типа *A* и типа *B*. Для каждого изделия используется по одной детали типа *A* и типа *B*.

R5. Возможны общие потери в производстве станка в количестве 2 изделий как брак.

R6. Если количество заготовок меньше, чем производительность станка или механизма, то изготавливается то минимальное количество изделий, которое можно получить из имеющихся заготовок, то есть равное количеству заготовок того типа, которых меньше.

R7. Если количество заготовок больше или равно производительности станка, то изготавливается количество изделий, равное производительности станка. Остальные заготовки остаются в кассете для следующего раза.

Необходимо построить программу, моделирующую данный, производящий изделия, станок, которая соответствует вышеуказанным требованиям.

Мы рассматриваем эту систему как реактивную, которая воспринимает в качестве входных воздействий количество заготовок типа *A* и типа *B*, поступающих в начале дня, а выходной реакцией является количество изготовленных за день изделий.

Определим переменные в этой программе:

A – количество заготовок типа A;

B – количество заготовок типа B;

D – планируемое количество произведенных изделий;

D_OUT – количество произведенных изделий с учетом возможного брака.

Множество базовых протоколов, соответствующее требованиям, состоит из пяти протоколов. Первые два представляют ввод заготовок в станок. Остальные три моделируют процесс изготовления изделий. Таким образом, получаем следующую модель требований.

P1: A = 0 <Ввод кассет с заготовками типа A> A := 20

P2: B = 0 <Ввод кассет с заготовками типа B> B := 16

P3: $(A \le B) \wedge (A \le 10)$ <Произвести и выдать изделия в количестве A> D := A; A: = 0; B := B - A; $(D\_OUT \ge D - 2) \wedge (D\_OUT \le D)$

P4: (A > B) ∧ (B ≤ 10)<Произвести и выдать изделия в количестве B> D :=B; B := 0; A := A - B; (D_OUT ≥ D - 2) ∧ (D_OUT ≤ D)
P5: (A ≤ B) ∧ (A ≥ 10) ∨ (A > B) ∧ (B ≥ 10) <Произвести и выдать изделия в количестве 10> A:= A - 10; B := B – 10; D:=10; (D_OUT ≥ 8) ∧ (D_OUT ≤ 10)

Мы также определяем порядок на множестве базовых протоколов, который задает последовательность их применения согласно вышеуказанным требованиям. Данный порядок мы можем изобразить в виде диаграммы (рис. 3).

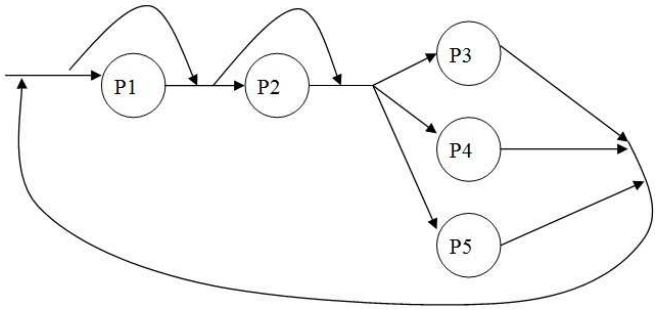

Рис. 3. Порядок на множестве базовых протоколов

Согласно диаграмме процесс работы станка начинается с ввода заготовок типа *A* (протокол *P1*) и типа *B* (протокол *P2*). Если количество заготовок больше нуля, то их ввод не производится, что воплощено в стрелочках, обходящих протоколы *P1* и *P2*. После этого производится изготовление изделий согласно количеству заготовок (протоколы *P3 – P5*).

Зафиксировав некоторое начальное состояние модели требований $(A=0)\wedge(B=0)$, мы можем моделировать описанные требования посредством применения базовых протоколов. При этом мы получаем трассы, содержащие символьное состояние среды после каждого применения базового протокола. Эти трассы могут быть основой теста для поверки программы, которая должна соответствовать вышеизложенным требованиям.

Ниже представлена одна из возможных последовательностей базовых протоколов и соответствующих символьных состояний среды.

Таблица 1. Состояния среды в модели

	$(A=0)\wedge(B=0)$
$P1$	$(A=20)\wedge(B=0)$
$P2$	$(A=20)\wedge(B=16)$
$P5$	$(A=10)\wedge(B=6)\wedge(D=10)\wedge(D\_OUT\leq 10)\wedge(D\_OUT\geq 8)$
$P4$	$(A=4)\wedge(B=0)\wedge(D=6)\wedge(D\_OUT\leq 6)\wedge(D\_OUT\geq 4)$
$P2$	$(A=4)\wedge(B=16)\wedge(D=6)\wedge(D\_OUT\leq 6)\wedge(D\_OUT\geq 4)$
$P3$	$(A=0)\wedge(B=12)\wedge(D=4)\wedge(D\_OUT\leq 4)\wedge(D\_OUT\geq 2)$
$P1$	$(A=20)\wedge(B=12)\wedge(D=4)\wedge(D\_OUT\leq 4)\wedge(D\_OUT\geq 2)$
$P5$	$(A=10)\wedge(B=2)\wedge(D=10)\wedge(D\_OUT\leq 10)\wedge(D\_OUT\geq 8)$
$P4$	$(A=8)\wedge(B=0)\wedge(D=2)\wedge(D\_OUT\leq 2)\wedge(D\_OUT\geq 0)$
$P2$	$(A=8)\wedge(B=16)\wedge(D=2)\wedge(D\_OUT\leq 2)\wedge(D\_OUT\geq 0)$
$P3$	$(A=0)\wedge(B=8)\wedge(D=8)\wedge(D\_OUT\leq 8)\wedge(D\_OUT\geq 6)$
$P1$	$(A=20)\wedge(B=8)\wedge(D=8)\wedge(D\_OUT\leq 8)\wedge(D\_OUT\geq 6)$
$P4$	$(A=12)\wedge(B=0)\wedge(D=8)\wedge(D\_OUT\leq 8)\wedge(D\_OUT\geq 6)$
$P2$	$(A=12)\wedge(B=16)\wedge(D=8)\wedge(D\_OUT\leq 8)\wedge(D\_OUT\geq 6)$
$P5$	$(A=2)\wedge(B=6)\wedge(D=10)\wedge(D\_OUT\leq 10)\wedge(D\_OUT\geq 8)$
$P3$	$(A=0)\wedge(B=4)\wedge(D=2)\wedge(D\_OUT\leq 2)\wedge(D\_OUT\geq 0)$
$P1$	$(A=20)\wedge(B=4)\wedge(D=2)\wedge(D\_OUT\leq 2)\wedge(D\_OUT\geq 0)$
$P4$	$(A=16)\wedge(B=0)\wedge(D=4)\wedge(D\_OUT\leq 4)\wedge(D\_OUT\geq 2)$
$P2$	$(A=16)\wedge(B=16)\wedge(D=4)\wedge(D\_OUT\leq 4)\wedge(D\_OUT\geq 2)$
$P5$	$(A=6)\wedge(B=6)\wedge(D=10)\wedge(D\_OUT\leq 10)\wedge(D\_OUT\geq 8)$
$P3$	$(A=0)\wedge(B=0)\wedge(D=6)\wedge(D\_OUT\leq 6)\wedge(D\_OUT\geq 4)$

Каждое новое состояние среды получается с помощью предикатного трансформера функцией pt, имеющей в качестве аргументов текущее состояние среды – E_{old}, предусловие Precond и постусловие Postcond базового протокола. Таким образом, новое состояние E_{new} среды получается следующим образом:

$$E_{new} = pt\left(E_{old} \wedge Precond, Postcond\right).$$

В дальнейшем для моделирования также используется функция обратного предикатного трансформера, позволяющая получать предыдущие состояния среды от текущего.

$$E_{old} = pt^{-1}\left(E_{new}, Postcond\right) \wedge Precond.$$

Описание свойств функции pt содержится в работе [4].

4. Символьное моделирование программного кода

Система может быть спроектирована любыми способами программистом с использованием исходных требований. Программист может либо написать код, соответствующий этим требованиям, либо использовать генерацию программного кода и уточнять его в соответствии с требуемым уровнем абстракции.

В данных требованиях мы абстрагированы от двух механизмов, обрабатывающих заготовки типа *A* и типа *B*, которые являются составляющими системы и о структуре которых на уровне требований мы ничего не знаем. Эти механизмы производят детали типа *A* и типа *B*. Каждый из них может иметь определенную производительность и определенный процент брака. При проектировании систем допускают переиспользование ранее спроектированных таких механизмов со своими собственными свойствами в новой системе.

```
main(){
  int A=0,B=0,D,A_OUT=0,B_OUT=0;
  for(i=0;;i++){
    /* Input of cartridges in machine*/
    if(A==0) A = 20; /*s1*/
    if(B==0) B = 16; /*s2*/
    /* producing of parts of type A */
    if (A>=15){A_OUT=A_OUT+15;A=A-15;} /*s3*/
    else {A_OUT=A_OUT+A;A=0;}
    //precondition: ZZ=A_OUT /*s31*/
    PERCENT_A(&A_OUT);
    //postcondition: (A_OUT>=ZZ-3)&&(A_OUT<=ZZ)
    /* producing of parts of type B*/
    if(B>=9){B_OUT=B_OUT+9;B=B-9;} /*s4*/
    else {B_OUT=B_OUT+B;B=0;}
    //precondition: ZZ=B_OUT /*s41*/
    PERCENT_B(&B_OUT);
    //postcondition: (B_OUT>=ZZ-1)&&(B_OUT<=ZZ)
    /*producing of output product*/
    if((A_OUT>=B_OUT)&&(A_OUT<10)){ /*s5*/
      D=A_OUT; B_OUT=B_OUT-A_OUT; A_OUT=0;}
    if((A_OUT>B_OUT)&&(B_OUT<10)){ /*s6*/
      D=B_OUT; A_OUT=A_OUT-B_OUT; B_OUT=0; }
    if((A_OUT<=B_OUT)&&(A_OUT>=10)||   /*s7*/
       (A_OUT>=B_OUT)&&(B_OUT>=10)){
         A_OUT:=A_OUT-10; B_OUT:=B_OUT-10: D:=10;}
    produce(D); /*s8*/
    //postcondition: (D_OUT>=D-2)&&(D_OUT<=D)
  }
}
```

Рис. 4. С-программа

Таким образом, задача программиста состоит в том, чтобы создать код, представляющий более низкий уровень абстракции с учетом информации о составляющих механизмах, а с другой стороны, этот код должен соответствовать исходным требованиям. Для связи модели с ее реализацией естественно требовать, чтобы программа использовала все атрибуты модели требований, в частности, те, которые используются для взаимодействия системы со средой.

Пусть мы имеем некоторые уточнения, относящиеся к работе механизмов *A* и *B*, являющихся составляющими системы, заключающиеся в том, что механизм, производящий детали типа *A*, имеет производительность 15 деталей, а типа *B* – 9 деталей в сутки. При этом количество возможного брака составляет 3 детали и 1 деталь соответственно.

Учитывая эти уточнения, можно написать С-программу (рис. 4).

В данной программе мы уточнили производство деталей типа *A* и типа *B*. Но будет ли тогда данная программа соответствовать исходным требованиям?

Для дедуктивного тестирования будет использоваться символьное моделирование С-программы. Мы рассматриваем некоторое подмножество С-программ, состоящих из операторов присваивания, условных операторов, операторов цикла и процедур. В этом случае символьное моделирование будет производиться так же, как и моделирование требований. Отличием будет то, что подпрограммы или библиотечные функции должны быть аннотированы пред- и постусловиями. Таким образом, мы имеем следующие состояния среды после выполнения каждого оператора (рис. 5).

C-statements	Symbolic states of environments
int A=0,B=0,D, A_OUT = 0,B_OUT = 0;	(A=0)∧(B=0) ∧ (A_OUT=0) ∧ (B_OUT=0)
for (i=0;;i++) {	(A=0) ∧ (B=0) ∧ (A_OUT=0) ∧ (B_OUT=0) ∧ (i=0)
if (A==0) { A = 20;} /*s1*/	(A=20) ∧ (B=0) ∧ (A_OUT=0) ∧ (B_OUT=0) ∧ (i=0)
if (B==0){ B = 16;} /*s2*/	(A=20) ∧ (B=16) ∧ (A_OUT=0) ∧ (B_OUT=0) ∧ (i=0)
if (A>=15) { A_OUT = A_OUT + 15;A = A – 15;}/*s3*/	(A=5) ∧ (B=16) ∧ (A_OUT=15) ∧ (B_OUT=0) ∧ (i=0)
PERCENT_A(&A_OUT);	(A=5) ∧ (B=16) ∧ (12<=A_OUT<=15) ∧ (B_OUT=0) ∧ (i=0)
if (B>=9) { B_OUT = B_OUT + 9;B = B – 9;} /*s4*/	(A=5) ∧ (B=7) ∧ (12<=A_OUT<=15) ∧ (B_OUT=9) ∧ (i=0)
PERCENT_B(&B_OUT);	(A=5) ∧ (B=7) ∧ (12<=A_OUT<=15) ∧ (8<=B_OUT<=9) ∧ (i=0)
if ((A_OUT > B_OUT) && (B_OUT < 10)){ D = B_OUT; A_OUT = A_OUT - B_OUT; B_OUT = 0; } /*s6*/	(A=5) ∧ (B=7) ∧ (3<=A_OUT<=7) ∧ (8<=D <=9) ∧ (i=0) ∧ (B_OUT = 0)
produce(D);	(A=5) ∧ (B=7) ∧ (3<=A_OUT<=7) ∧ (i=0) ∧ (B_OUT = 0) ∧ (D_OUT>=6) ∧ (D_OUT<=9)

Рис. 5. Состояния среды в С-программе

5. Дедуктивное тестирование

Входом дедуктивного тестирования (рис. 6) является множество трасс, полученных посредством символьного моделирования модели требований. Другим входом является программный код (в данном случае написанный на С).

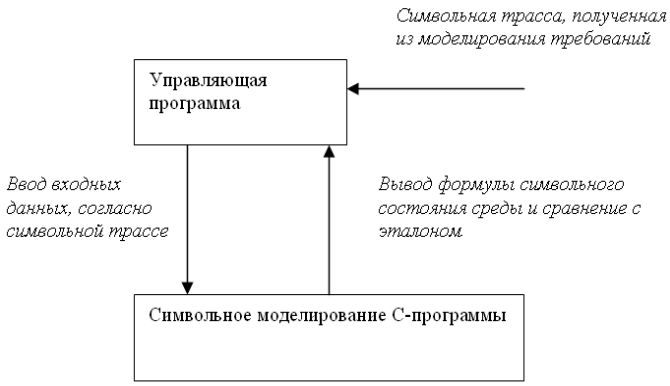

Рис. 6. Процедура дедуктивного тестирования

Мы рассматриваем тестируемую программу и внешнюю среду, посредством которой управляется моделирующая программа.

Управляющая программа имитирует эту внешнюю среду, подавая на вход моделирующей программы данные вместе с формулой, ограничивающей их значения, в соответствии с символьной трассой, полученной в процессе моделирования требований.

В точках взаимодействия управляющей и тестируемой программы производится контроль значений. В качестве такой точки контроля выбрана точка ответа системы, а именно выдача количества деталей после их производства.

В точке контроля мы получаем текущее символьное состояние среды в моделирующей программе и сравниваем его с контрольным значением в трассе. Рассматриваются следующие случаи.

1. Символьное состояние среды на контрольной трассе эквивалентно символьному состоянию среды, полученной в результате символьного моделирования С-программы. Это означает, что ответ программы соответствует требуемой модели требований.

2. Символьное состояние среды, полученной в результате символьного моделирования С-программы, следует из символьного состояния среды на контрольной трассе. Это означает, что на уровне кода запрограммировано поведение, которое выходит за рамки требований.

3. Символьное состояние среды на контрольной трассе следует из символьного состояния среды, полученной в результате символьного моделирования С-программы. Это означает, что на уровне кода не запрограммировано поведение, которое определено в требованиях. Трассу, не соответствующую требованиям, можно найти обратным моделированием на уровне требований.

4. Ни одно из символьных состояний не следует из другого. В этом случае программа не удовлетворяет требованиям.

Рассмотрим дедуктивное тестирование на примере, предложенном выше. В данном примере изначально мы имели эквивалентные состояния среды $(A=0)\wedge(B=0)$ на уровне требований (в символьной трассе) и в самом коде.

Рассмотрим первую точку контроля программы, которая соответствует точке *s8*, представляющей функцию produce .

Контрольное состояние среды в трассе:

(A=10) ∧ (B = 6) ∧ (D_OUT≤ 10) ∧ (D_OUT ≥ 8).

Состояние среды в С-программе:

(A=5) ∧ (B=7) ∧ (3 ≤ A_OUT ≤ 7) ∧ (8 ≤ D_OUT ≤ 9) ∧ (i = 0) ∧ (B_OUT = 0).

Эти две формулы не являются эквивалентными, что говорит о том, что программа работает не в соответствии со сценариями, представляющими требования.

Для анализа формул возможно рассматривать информационно независимые литералы.

В отношении переменных A и B можно сказать, что значения в С-среде не соответствуют значениям на контрольной трассе, значит, поведение неверно. *~((B = 6) ⇔ (B = 7))* и *~((A = 10) ⇔ (A = 5)).*

В отношении переменно *D_OUT* можно сказать, что С-программа не моделирует те сценарии, которые представлены требованиями.

$$\sim((8 \le D\_OUT \le 9) \Leftrightarrow (D\_OUT <= 10) \wedge (D\_OUT \ge 8)).$$

Имея отрезок трассы от начальных значений до обнаруженного несоответствия, требованиям можно найти место, где появляется это несоответствие, визуальным анализом или методом обратного моделирования, который также описан в работе [4].

Нарушение для переменных A и B возникает на операторах, помеченных *s3* и *s4*, или же производительность блоков, изготавливающих детали A и B, влияет на общую производительность станка. Для переменной *D_OUT* нарушение обнаруживается в момент выполнения функций *PERCENT_A* и *PERCENT_B*, то есть процент брака встроенных механизмов не совместим с требуемым процентом брака.

6. Выводы

Дедуктивное тестирование является методом, который может решить задачу покрытия тестами всех конкретных значений переменных в программе. В данной методике не

рассматривается вопрос покрытия всего кода или же всех требований тестами. Это рассматривается в работах, посвященных символьному моделированию и генерации трасс с заданным покрытием [5].

Вместе с тем в данном методе существуют проблемы, требующие уточнения в реальных индустриальных примерах.

Первая проблема заключается в определении соответствия между переменными в требованиях и переменными в программе. Одним из условий возможности дедуктивного тестирования является соответствие уровня абстракций в интерфейсе между внешним воздействием на программу и выходом программы для требований и для создаваемого кода. Все переменные должны быть четко описаны в требованиях и использованы в коде.

На этапе кодирования при уточнении требований возможны уточнения некоторых абстракций и, соответственно, появление переменных, которые не относятся к переменным, фигурирующим в требованиях. В этом случае мы выбираем переменные, которые информационно зависимы от переменных, участвующих в требованиях, и проверяем формулу среды в С-программе на эквивалентность с формулой состояния среды в требованиях, относя новые информационные переменные под квантор всеобщности.

Другой проблемой является появление недетерминизма в символьном моделировании С-программы вследствие уточнения кода. Таким образом, в коде появляется информация, которая не отражена в требованиях. Соответственно среда С-программы может быть не эквивалентной среде, фигурирующей в требованиях. В этом случае следует проверять не на эквивалентность, а на импликацию формулы среды в требованиях из формулы среды в С-программе. Тем не менее, покрытие данных может быть определено с помощью техники обратного моделирования.

Рассмотренный пример содержит программу, построенную с помощью линейных арифметических преобразований. Используя методы символьного моделирования, разработанные авторами [2], возможно рассматривать также рациональные числа, списки, функциональные и перечислимые типы данных. В качестве будущих работ планируется использовать ряд индустриальных примеров, а также технику инвариантов.

СПИСОК ЛИТЕРАТУРЫ

1. Кулямин В.В. Формальные подходы к тестированию математических функций / Кулямин В.В. // Труды ИСП РАН. – Москва, 2006. – Т. 10. – С. 69 – 114

2. Спецификация систем с помощью базовых протоколов / А.А. Летичевский, Ю.В. Капитонова, А.А. Летичевский (мл.) [и др.] // Кибернетика и системный анализ. – 2010. – № 4. – С. 3 – 21.

3. Летичевский А.А. (мл.) / Об одном классе базовых протоколов / А.А. Летичевский (мл.) // Проблемы программирования. – 2005. – № 4. – С. 82 – 97

4. Свойства предикатного трансформера системы VRS / А.А. Летичевский, А.Б. Годлевский, А.А. Летичевский (мл.) [и др.] // Кибернетика и системный анализ. – 2005. – № 4. – С. 4 – 14.

5. The technology of Automation Verification and Testing in Industrial Projects / S. Baranov, V. Kotlyarov, A. Letichevsky [et al.] // Proc. of St. Petersburg IEEE Chapter, International Conference, (St. Petersburg, Russia, May 18–21, 2005). – St. Petersburg, Russia, 2005. – P. 81 – 86

6. Clarke L. A system to generate test data and symbolically execute programs / L. Clarke // IEEE Trans. Software Eng. – 1976. – Vol. 2. – P. 215 – 222.

7. Wegbreit B. Symbolic execution and program testing / B. Wegbreit // Communications of ACM. – 1976. – Vol. 19. – P. 385 – 394.

Стаття надійшла до редакції 13.09.2013