Тестирование в условиях неполной информации. Подход к разработке спецификаций и генерации тестов

А.С. Камкин (kamkin@ispras.ru)

Аннотация. В статье исследуются вопросы функционального тестирования программных систем в условиях неполной информации. Неполнота информации рассматривается в двух аспектах: статическом, связанном с неполнотой функциональных требований, по которым разрабатываются спецификации и тесты, и динамическом, связанном с неполнотой информации о состоянии целевой системы в процессе тестирования. В работе предлагается подход к разработке функциональных спецификаций и генерации функциональных тестов, основанный на использовании неопределенных значений для моделирования состояния целевой системы, а также трехзначной логики Клини для работы с неполными требованиями и описания свойств системы. В качестве базовой технологии используется технология тестирования UniTESK.

1. Введение

В последнее время в науке и технике интенсивно развиваются и широко распространяются методы, оперирующие с неполной или нечеткой информацией[1] [1]. Примерами теорий, в рамках которых разрабатываются подобные методы, являются математическая статистика, теория нечетких множеств, теория возможностей и многие другие. Методы анализа и обработки неполной информации хорошо зарекомендовали себя в искусственном интеллекте, теории баз данных, системном анализе и исследовании операций. Сегодня их активно используют для моделирования сложных физических, экономических и социальных явлений [2]. Таким образом, работа с неопределенной и нечеткой информацией постепенно становится уделом точных наук.

Программной инженерии, претендующей на точность и методичность инженерной дисциплины, также приходиться иметь дело с неполной информацией. В проектах по разработке программного обеспечения неопределенность возникает на самых ранних фазах. Размытый и туманный характер носят требования к программной системе, которые приходится многократно уточнять, дополнять и согласовывать. После доработок требования используются на следующей фазе, на которой, как правило, имеют дело с более низким уровнем абстракции, поэтому требования вновь кажутся неполными, их снова приходится дорабатывать. Таким образом, на каждой фазе проекта исходная информация оказывается не полностью определенной.

В работе рассматривается лишь один из этапов разработки программного обеспечения — тестирование. Обычно на этапе тестирования располагают более-менее завершенным продуктом и достаточно проработанными

[1] В дальнейшем эпитет *неполный* используется в отношении как, собственно, неполной, так и нечеткой информации, как и его синонимы *неопределенный* и *не полностью определенный*.

требованиями. Казалось бы, что на этом этапе в требованиях не должно быть никаких неопределенностей, но, как показывает практика, это не так. Использование требований для целей тестирования в очередной раз выявляет их недостатки. Требования после многочисленных доработок вновь оказываются неполными. С другой стороны, в процессе тестирования возможна ситуация, когда состояние целевой системы не полностью определено. Например, такая ситуация может возникнуть, когда состояние целевой системы скрыто, а начальное состояние в требованиях не определено. Неполнота информации о состоянии целевой системы может приводить к невозможности адекватной оценки правильности поведения целевой системы. Перед тем, как оценивать поведение, необходимо получить представление о состоянии целевой системы путем подачи тестовых воздействий и анализа реакций на них.

Статья выполнена в контексте тестирования на основе моделей (*MBT*, *model based testing*)[2], более точно, в контексте технологии тестирования UniTESK [3-5], хотя рассматриваемые в ней вопросы актуальны не только для этой технологии. Модель представляет собой некоторое достаточно абстрактное отображение структуры и поведения целевой системы, созданное на базе требований [6]. В процессе тестирования состояние модели можно интерпретировать, как располагаемую информация о состоянии целевой системы, сформулированную в терминах требований.

В работе предлагается подход к разработке функциональных спецификаций и генерации функциональных тестов, позволяющий работать с неполными требованиями к целевой системе и неполной информацией о ее состоянии. Подход основан на использовании неопределенных значений для моделирования состояния целевой системы, а также трехзначной логики Клини (*Kleene*) [7] для работы с требованиями и описания свойств системы. Результаты работы получены в ходе выполнения проекта по разработке открытого тестового набора OLVER для операционной системы Linux [8].

Статья построена следующим образом. Во втором, следующем за введением, разделе делается краткий обзор технологии тестирования UniTESK. В третьем разделе рассматриваются вопросы, связанные с неполнотой функциональных требований и неполнотой информации о состоянии целевой системы в процессе тестирования. В четвертом разделе описывается подход к спецификации систем на основе неопределенных значений, уточняемых типов и трехзначной логики Клини. В пятом разделе рассматриваются вопросы построения тестовых последовательностей по неопределенным обобщенным моделям. В шестом разделе предлагается простое расширение технологии тестирования UniTESK на примере инструмента разработки тестов CTesK [9]. В заключении делается краткое резюме работы.

2. Краткий обзор технологии тестирования UniTESK

Технология тестирования UniTESK [3-5] была разработана в Институте системного программирования РАН [10] на основе опыта, полученного при разработке и применении технологии KVEST (*kernel verification and specification*

---

[2] Часто вместо термина *тестирование на основе моделей* используется близкий термин *тестирование на основе* (*формальных*) *спецификаций* (*specification based testing*, *specification-driven testing*).

technology) [11]. Общими чертами этих технологий являются использование формальных спецификаций в форме *пред-* и *постусловий интерфейсных операций* и *инвариантов типов данных* для автоматической генерации *оракулов* (компонентов тестовой системы, осуществляющих проверку правильности поведения целевой системы), а также применение конечно-автоматных моделей для построения последовательностей тестовых воздействий (*тестовых последовательностей*). В отличие от технологии KVEST, в которой для спецификации требований использовался язык RSL (*RAISE specification language*) [12], технология UniTESK использует расширения широко известных языков программирования.

2.1. Архитектура тестовой системы UniTESK

Архитектура тестовой системы UniTESK была разработана на основе многолетнего опыта тестирования промышленного программного обеспечения из разных предметных областей и разной степени сложности. Учет этого опыта позволил создать гибкую архитектуру, основанную на следующем разделении задачи тестирования на подзадачи:

- Построение тестовой последовательности, нацеленной на достижение нужного покрытия.
- Создание единичного тестового воздействия в рамках тестовой последовательности.
- Установление связи между тестовой системой и реализацией целевой системы.
- Проверка правильности поведения целевой системы в ответ на единичное тестовое воздействие.

Для решения каждой из этих подзадач предусмотрены специальные компоненты тестовой системы (*рисунок 1*): для построения тестовой последовательности и создания единичных тестовых воздействий — *обходчик* и *итератор тестовых воздействий*, для проверки правильности поведения целевой системы — *оракул*, для установления связи между тестовой системой и реализацией целевой системы — *медиатор*. Рассмотрим подробнее каждый из указанных компонентов.

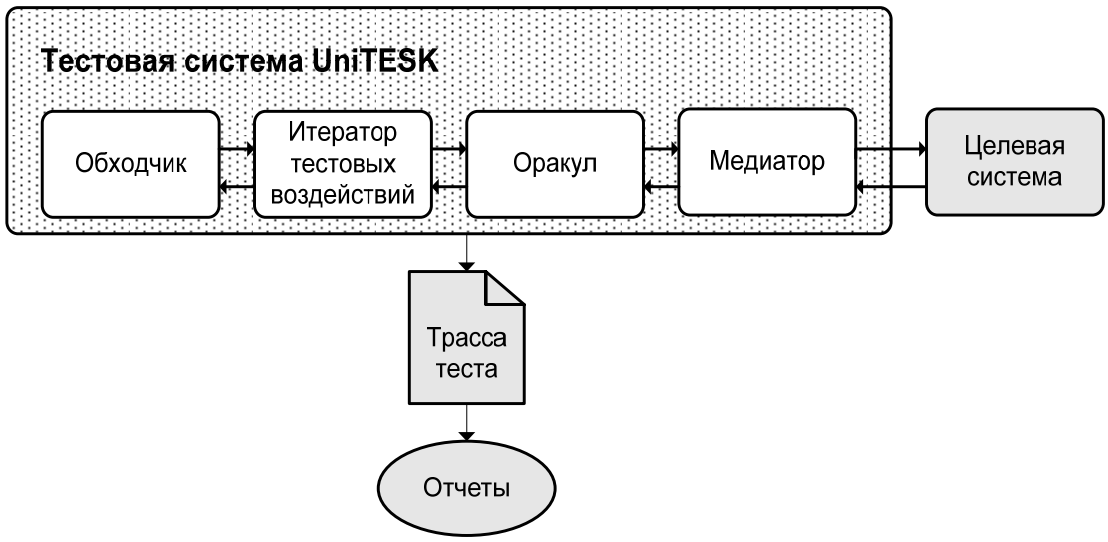

Рисунок 1. Архитектура тестовой системы UniTESK.

Обходчик является библиотечным компонентом тестовой системы UniTESK и предназначен вместе с итератором тестовых воздействий для построения тестовой последовательности. В основе обходчика лежит алгоритм обхода графа состояний *обобщенной конечно-автоматной модели* целевой системы (конечного автомата, моделирующего целевую систему на некотором уровне абстракции). Обходчики, реализованные в библиотеках инструментов UniTESK, требуют, чтобы обобщенная конечно-автоматная модель целевой системы, была детерминированной[3] и имела сильно-связный граф состояний.

Итератор тестовых воздействий работает под управлением обходчика и предназначен для перебора в каждом достижимом состоянии конечного автомата допустимых тестовых воздействий. Итератор тестовых воздействий автоматически генерируется из тестового сценария, представляющего собой неявное описание обобщенной конечно-автоматной модели целевой системы.

Оракул оценивает правильность поведения целевой системы в ответ на единичное тестовое воздействие. Он автоматически генерируется на основе формальных спецификаций, описывающих требования к целевой системе в виде пред- и постусловий интерфейсных операций и инвариантов типов данных.

Медиатор связывает абстрактные формальные спецификации, описывающие требования к целевой системе, с конкретной реализацией целевой системы. Медиатор преобразует единичное тестовое воздействие из спецификационного представления в реализационное, а полученную в ответ реакцию — из реализационного представления в спецификационное. Также медиатор синхронизирует состояние спецификации с состоянием целевой системы.

Трасса теста отражает события, происходящие в процессе тестирования. На основе трассы можно автоматически генерировать различные отчеты, помогающие в анализе результатов тестирования.

3. Полнота функциональных требований

В программной инженерии разработка, тестирование, а также сопровождение и эксплуатация программной системы осуществляются на основе *требований*. На ранних этапах жизненного цикла программной системы требования носят размытый характер, описывая в общих чертах концептуальный замысел разрабатываемой системы. На протяжении всего жизненного цикла системы требования дорабатываются и, пока система используется, их нельзя назвать полностью завершенными. Поскольку в работе исследуются вопросы функционального тестирования, в ней рассматриваются только *функциональные требования* — требования, описывающие функциональность системы, то есть *что* она должна делать, но не описывающие *как* она должна это делать[4].

Основными свойствами требований, характеризующими их качество, являются *понятность*, *непротиворечивость* и *полнота*. Мы будем рассматривать только одно из них — полноту, предполагая при этом, что требования понятны и

---

[3] Исключение составляет обходчик *ndfsm* [13], позволяющий обходить графы состояний для некоторого класса недетерминированных конечных автоматов.

[4] В дальнейшем для краткости будем опускать эпитет *функциональные* и употреблять термин *требования*, понимая под ним именно *функциональные требования*.

непротиворечивы. В дедуктивных науках теория называется *полной*, если всякое высказывание, сформулированное в терминах этой теории, может быть либо доказано, либо опровергнуто [14]. В программной инженерии понятие полноты несколько сложнее. Обычно требования рассматриваются в контексте некоторой деятельности и считаются полными, если на их основе можно решить любую задачу в рамках этой деятельности.

Понятно, что из-за различия в роде деятельности, полнота требований по-разному воспринимается разными группами лиц, связанных с программной системой. Например, для пользователя системы полнота требований, в первую очередь, означает возможность на их основе осуществить любой вариант использования системы (*use case*)[5]; для разработчика системы — правильно ее реализовать; для разработчика тестов — разработать полный набор тестов. Поскольку статья посвящена тестированию, полнота требований в ней рассматривается только с точки зрения разработчика тестов.

3.1. Требования и сценарии взаимодействия с системой

Задачу тестирования можно разбить на две основные подзадачи: построение тестовой последовательности и проверку правильности поведения целевой системы в ответ на поданные тестовые воздействия. В соответствии с этим, разработка тестов по технологии UniTESK разбивается на разработку тестовых сценариев, описывающих способ построения тестовой последовательности, и разработку спецификаций, описывающих проверки правильности поведения целевой системы. Данный раздел посвящен определению полноты требований в контексте разработки тестовых сценариев.

Опр. Будем говорить, что требования к целевой системе являются *сценарно полными*, если они описывают, как на основе интерфейса целевой системы осуществить любой вариант использования системы, а также воссоздать или идентифицировать любую указанную в них ситуацию. В противном случае будем говорить, что требования являются *сценарно неполными*. ■

В сценарно неполных требованиях для некоторых указанных в них ситуаций могут отсутствовать описания способов их достижения или идентификации. В контексте технологии тестирования UniTESK это означает невозможность достичь 100% покрытия тестовых ситуаций, определенных на основе требований в спецификации. Если требования сценарно неполны, значения некоторых атрибутов состояния спецификации оказываются неопределенными на любых тестовых последовательностях.

Опр. Будем говорить, что требования к целевой системе являются *детерминированными*, если они описывают, как на основе интерфейса целевой системы можно контролировать или наблюдать ее поведение при выполнении любого допустимого сценария взаимодействия с ней. В противном случае будем говорить, что требования являются *недетерминированными*. ■

Если требования недетерминированны, то возможны сценарии взаимодействия с целевой системой, при выполнении которых нельзя однозначно определить

[5] Предполается, что варианты использования целевой системы известны, например, описаны в требованиях к системе.

состояние целевой системы. В контексте технологии тестирования UniTESK это означает, что функция вычисления постсостояния спецификации в некоторых ситуациях оказывается недетерминированной.

Проиллюстрируем понятие сценарной полноты и детерминизма требований на следующем простом примере.

Пример. Целевая система является стеком. Интерфейс целевой системы состоит из следующих функций:

- `Stack* create(void);`
- `void push(Stack *stack, Object *obj);`
- `Object* pop(Stack *stack).`

Пусть требования к целевой системе формулируются следующим образом:

> *Функция* `create` *создает пустой стек и возвращает указатель на него. В случае если стек не полностью заполнен, вызов функции* `push` *добавляет в него элемент, противном случае, вызов функции* `push` *не меняет состояние стека. Вызов функции* `pop` *для непустого стека возвращает последний добавленный в него элемент, для пустого стека —* `NULL`.

Эти требования не являются сценарно полными, так как не описывают как достичь или идентифицировать указанную в них ситуацию «стек полностью заполнен». Также эти требования не являются детерминированны, так как не позволяют однозначно определить в какое состояние перейдет целевая система после вызова функции `push`. Если добавить в требования описание ситуации «стек полностью заполнен», указав, например, максимальное возможное число элементов в стеке, требования становятся сценарно полными и детерминированными. ■

3.2. Требования и оценка правильности поведения системы

Теперь рассмотрим определение полноты требований в контексте разработки спецификаций. Одна из основных задач разработчика тестов — определить, какое поведение целевой системы следует считать правильным, а какое ошибочным. Ясно, что требования, претендующие на полноту, должны позволять проводить такую классификацию.

Опр. Будем говорить, что требования к целевой системе являются *контрактно полными*, если на их основе можно однозначно классифицировать поведение целевой системы на правильное или ошибочное на любом допустимом сценарии взаимодействия с ней. В противном случае будем говорить, что требования являются *контрактно неполными*. ■

Контрактная неполнота требований означает, что в оракуле, оценивающем поведение целевой системы, есть «дыры», то есть возможны ситуации, в которых он не может вынести однозначный вердикт о правильности или ошибочности поведения целевой системы.

Проиллюстрируем понятие контрактной полноты требований на следующем примере.

Пример. Рассмотрим целевую систему из предыдущего примера. Пусть требования к функции `pop` формулируются следующим образом:

Вызов функции `pop` *для непустого стека возвращает последний добавленный в него элемент.*

Эти требования не являются контрактно полными, так как в них не описано, как должна вести себя функция `pop` для пустого стека. Если добавить в требования описания того, что вызов функции `pop` для пустого стека должен возвратить `NULL`, требования становятся контрактно полными. ■

3.3. Неполнота информации в тестировании

При разработке тестов часто возникает ситуация, когда нет прямого доступа к внутреннему состоянию целевой системы. Такое тестирование называется *тестированием со скрытым состоянием* (*hidden state testing*). В отличие от *тестирования с открытым состоянием* (*open state testing*), когда состояние целевой системы можно получить непосредственно через ее интерфейс, при тестировании со скрытым состоянием разработчику тестов необходимо четко представлять как изменяется состояние целевой системы при том или ином воздействии на нее. Для возможности определить состояние целевой системы после обработки тестового воздействия важную роль играет детерминизм требований.

Рисунок 2. Виды неполноты информации, возникающие при тестировании.

Другим видом неполноты информации в тестировании является неопределенность начального состояния целевой системы. Такое тестирование называется *тестированием с неизвестным начальным состоянием*. В отличие

от *тестирования с известным начальным состоянием*, при тестировании с неизвестным начальным состоянием на целевую систему сначала нужно подать последовательность тестовых воздействий, приводящую ее в некоторое известное состояние. Здесь важную роль играют сценарная полнота и детерминизм требований.

Подведем итог относительно различных видов неполноты информации, которые могут возникнуть при тестировании программных систем (*рисунок 2*). Во-первых, это может быть контрактная неполнота требований. Во-вторых, при тестировании может быть скрыто состояние целевой системы, что не позволяет непосредственно получать состояние целевой системы через ее интерфейс. В-третьих, может быть не определено начальное состояние целевой системы. В-четвертых, требования могут не определять как изменяется состояние целевой системы при выполнении того или иного воздействия или определять это недетерминированным образом. Наконец, реализация целевой системы может быть недетерминированной.

4. Спецификация в условиях неполной информации

В данном разделе рассматривается подход к разработке функциональных спецификаций, позволяющий работать с неполными требованиями к целевой системе и неполной информацией о ее состоянии.

4.1. Неопределенные значения и уточняемые типы

Поскольку в процессе тестирования возможна ситуация, когда состояние целевой системы не полностью определено, для моделирования состояния системы удобно использовать типы, поддерживающие неопределенные значения. Будем считать, что на множестве значений типов заданы *отношения уточнения*, являющиеся отношениями частичного порядка. Отношения уточнения позволяют сравнивать информативность значений: чем «больше» значение в отношении уточнения, тем больше информации оно несет (*рисунок 3*). Введем несколько понятий.

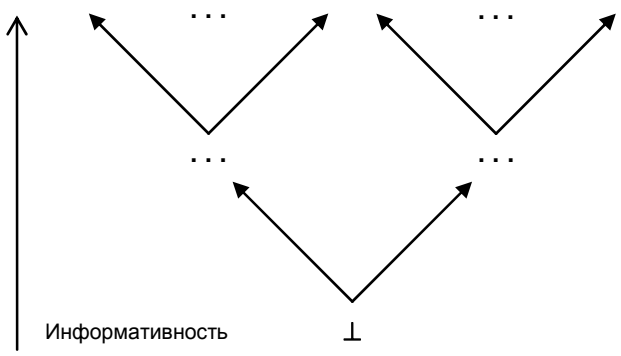

Рисунок 3. Отношение уточнения значений типа.

Опр. Тип $\mathbb{T}$ с заданным на на нем отношением уточнения $\Rightarrow$ называется *уточняемым типом*. Отношение уточнения, заданное на типе $\mathbb{T}$ будем обозначать $\Rightarrow_{\mathbb{T}}$ или просто $\Rightarrow$, если из контекста ясно о каком типе идет речь. ■

Не ограничивая общности рассуждений, будем считать, что у каждого уточняемого типа $\mathbb{T}$ существует единственное минимальное по отношению $\Rightarrow_{\mathbb{T}}$

значение, которое будем обозначать $\perp_{\mathbb{T}}$ или просто $\perp$ и называть *неопределенным значением* типа $\mathbb{T}$. Также будем считать, что любое значение типа $\mathbb{T}$ можно получить за *конечное* число уточнений неопределенного значения $\perp_{\mathbb{T}}$.

Опр. Максимальные значения уточняемого типа $\mathbb{T}$ по отношению $\Rightarrow_{\mathbb{T}}$ называются (*полностью*) *определенными* значениями типа $\mathbb{T}$. Значения уточняемого типа $\mathbb{T}$, не являющиеся полностью определенными, называются *неопределенными* или *не полностью определенными* значениями типа $\mathbb{T}$. ■

Опр. Пусть $\mathbb{T}$ — уточняемый тип, тогда через $\mathbb{T}_c$ будем обозначать (*полностью*) *определенный подтип* типа $\mathbb{T}$, то есть подтип, состоящий из всех полностью определенных значений типа $\mathbb{T}$. ■

Если некоторое свойство $\mathbb{P}$ имеет не полностью определенное значение x, это означает, что на самом деле значением свойства $\mathbb{P}$ является одно из полностью определенных значений, уточняющих x, но какое именно неизвестно. Нужно быть аккуратным при сравнении не полностью определенных значений на равенство. С одной стороны, разным неопределенным значениям может соответствовать одно и то же полностью определенное значение, с другой, одному неопределенному значению могут соответствовать разные полностью определенные значения.

Пример. Рассмотрим пример уточняемого типа $\mathbb{S}_{\mathbb{T}}$, представляющего *нечеткое множество* значений типа $\mathbb{T}$. Нечеткое множество s определяется трехзначной функцией принадлежности $\mathbb{F}_s$: $\mathbb{T} \to \{\text{true}, \text{false}, \perp\}$. $\mathbb{F}_s(x)$ интерпретируется следующим образом: если $\mathbb{F}_s(x) = \text{true}$, то x принадлежит множеству s, если $\mathbb{F}_s(x) = \text{false}$, то x не принадлежит множеству s, если $\mathbb{F}_s(x) = \perp$, то неизвестно принадлежит x множеству s или нет. Отношение уточнения естественно определить таким образом: $s_1 \Rightarrow_{S_T} s_2$ тогда и только тогда, когда из того что $\mathbb{F}_{s_1}(x) = \text{true}$, вытекает, что $\mathbb{F}_{s_2}(x) = \text{true}$, а и из того, что $\mathbb{F}_{s_1}(x) = \text{false}$, вытекает, что $\mathbb{F}_{s_2}(x) = \text{false}$. ■

Опр. Пусть $\mathbb{T}_1, \ldots, \mathbb{T}_n$ — уточняемые типы. Определим на декартовом произведении $\mathbb{T}_1 \times \ldots \times \mathbb{T}_n$ отношение уточнения следующим образом: $(x_1, \ldots, x_n) \Rightarrow_{T_1 \times \ldots \times T_n} (y_1, \ldots, y_n)$ тогда и только тогда, когда $x_1 \Rightarrow_{T_1} y_1, \ldots, x_n \Rightarrow_{T_n} y_n$. ■

От функций, определенных на уточняемых типах будет требовать *регулярности* и *полноты*: чем определеннее значение аргумента, тем определеннее значение функции, причем полностью определенному значению аргумента соответствует полностью определенное значение функции.

Опр. Функция $\mathbb{F}$: $\mathbb{T}_1 \to \mathbb{T}_2$ называется *регулярной*, если из того, что $x \Rightarrow_{T_1} y$ следует, что $\mathbb{F}(x) \Rightarrow_{T_2} \mathbb{F}(y)$. ■

Опр. Функция $\mathbb{F}$: $\mathbb{T}_1 \to \mathbb{T}_2$ называется *полной*, если $\mathbb{F}(\mathbb{T}_{1c}) \subseteq \mathbb{T}_{2c}$, то есть из того, что $x \in \mathbb{T}_{1c}$ следует, что $f(x) \in \mathbb{T}_{2c}$. ■

Обычно в языках программирования и спецификаций есть *базовые* типы и есть *составные* типы, значения которых строятся на основе значений других типов.

Опр. Составной тип называется *регулярным*, если все его конструкторы являются регулярными функциями. ■

Опр. Составной тип называется *полным*, если все его конструкторы являются полными функциями. ■

4.2. Неопределенность и трехзначная логика Клини

Для не полностью определенных состояний целевой системы, не исключены ситуации, когда из-за недостатка информации некоторые логические условия нельзя отнести ни к истинным, ни к ложным. В таких случаях использование двузначной логики для описания свойств целевой системы представляется не совсем адекватным. Для того чтобы описать логическое условие $\mathbb{P}$, которое может быть неопределенным, с помощью двух значений истинности нужно использовать специальные *модальные функции возможности* и *необходимости*: $\Diamond\mathbb{P}$ и $\Box\mathbb{P}$[6] и описать условие парой $(\Diamond\mathbb{P}, \Box\mathbb{P})$[7]. Тогда истинное условие описывается парой (true, true) (*необходимо*), ложное — парой (false, false) (*невозможно*), неопределенное — парой (true, false) (*возможно*, *но не необходимо*).

Описывать логические условия парой модальностей не очень удобно. Естественней положить, что функция истинности может принимать три значения: true (*истина*), false (*ложь*) и $\perp$ (*неопределенность*), а модальные функции возможности и необходимости рассматривать как функции $\{\text{true}, \text{false}, \perp\} \to \{\text{true}, \text{false}\}$, определяемые следующими таблицами истинности:

	◊
false	false
true	true
⊥	true

	□
false	false
true	true
⊥	false

Таблица. Определение модальных функций ◊ и □.

Впервые модальные функции возможности и необходимости подобным образом определил Лукасевич (*Łukasiewicz*), в трехзначной логике $Ł_3$ [15]. В логике $Ł_3$ неопределенное значение $\perp$ интерпретируется как промежуточное между ложью и истиной. В нашем случае неопределенное значение следует интерпретировать иначе — как неизвестное значение, которое может быть как истиной, так ложью, но какое оно именно неизвестно.

Подобным образом неопределенное значение интерпретируется в трехзначной логике Клини (*Kleene*), называемой $\mathbb{K}_3$. При получении дополнительной информации значение логического условия может измениться с

---

[6] На модальные функции возможности и необходимости обычно налагают некоторые ограничения, например, $\Diamond\mathbb{P} \equiv \neg\Box\neg\mathbb{P}$ и $\Box\mathbb{P} \equiv \neg\Diamond\neg\mathbb{P}$.

[7] Обычно модальная логика интерпретируется с помощью системы *возможных миров* (*possible worlds*): в каждом возможном мире интерпретация задается двухзначной функцией истинности, а модальные функции определяются совокупностью значений истинности в возможных мирах.

неопределенного значения ⊥ на false или true, но невозможно, чтобы значение изменилось с true на false или наоборот. Последнее требование называется требованием *регулярности* [7]. Если определить на множестве {true, false, ⊥} отношение уточнения, как показано на следующем рисунке, то регулярность условия означает его монотонность по отношению уточнения.

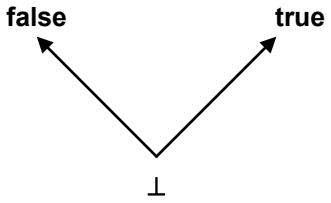

Рисунок 4. Отношение уточнения значений истинности в логике K_3.

Рассмотрим логические связки ¬, ∧ и ∨. Семантику связок можно определять по-разному, основным ограничением для нас является требование регулярности. Существуют два основных способа определения семантики связок: сильный (*сильная логика K_3*, *сильные связки K_3*) и слабый (*слабая логика K_3*, *слабые связки K_3*)[8].

	¬
false	true
true	false
⊥	⊥

∧	**false**	**true**	⊥
false	false	false	⊥
true	false	true	⊥
⊥	⊥	⊥	⊥

∨	**false**	**true**	⊥
false	false	true	⊥
true	true	true	⊥
⊥	⊥	⊥	⊥

Таблица. Слабые связки K_3.

	¬
false	true
true	false
⊥	⊥

∧	**false**	**true**	⊥
false	false	false	false
true	false	true	⊥
⊥	false	⊥	⊥

∨	**false**	**true**	⊥
false	false	true	⊥
true	true	true	true
⊥	⊥	true	⊥

Таблица. Сильные связки K_3.

В своей работе мы используем сильный вариант логики K_3.

5. Тестирование в условиях неполной информации

Как уже неоднократно отмечалось, при тестировании возможна ситуация, когда информация о состоянии целевой системы не доступна полностью тестовой системе. Это означает, что соответствующее состояние модели не полностью определено. В данном разделе описывается подход к генерации тестовых последовательностей, позволяющий работать в таких условиях.

[8] Возможны и другие варианты определения семантики связок, например, так называемая *короткая логика*.

5.1. Неопределенные обобщенные модели

Для генерации тестовых последовательностей обычно используют не сами модели тестируемых систем, а их обощения, называемые *обобщенными моделями*. Например, в технологии тестирования UniTESK используются обобщенные конечно-автоматные модели, описанные в тестовых сценариях. Поскольку состояния модели могут быть неопределенными, не исключено что и состояния обобщенной модели (*обобщенные состояния*) окажутся неопределенными.

Пусть $\mathbb{M}$ — уточняемый тип, представляющий состояния модели, $\mathbb{S}$ — уточняемый тип, представляющий обобщенные состояния, а $\mathbb{F}: \mathbb{M} \to \mathbb{S}$ — *функция обобщения модели* — функция, ставящая в соответствие каждому состоянию модели обобщенное состояние. Будем считать, что функция $\mathbb{F}$ является регулярной и полной. В этом случае она задает факторизацию на множестве полностью определенных значений типа $\mathbb{M}$[9]. Для удобства будем считать, что $\mathbb{F}(\perp_{\mathbb{M}}) = \perp_{\mathbb{S}}$.

5.2. Генерация тестов на основе неопределенных моделей

Теперь рассмотрим естественное обобщение технологии тестирования UniTESK для генерации тестовых последовательностей на основе неопределенных обобщенных моделей.

В случае когда состояние модели неопределено, тестовые воздействия можно разделить на три группы:

- тестовые воздействия, которые *допустимы* для этого состояния;
- тестовые воздействия, которые *недопустимы* для этого состояния;
- тестовые воздействия, для которых *не известно*, допустимы они или нет для этого состояния.

Таким образом, итератор тестовых воздействий для каждого обобщенного состояния определяет нечеткое множество тестовых воздействий. Из общих соображений понятно, что чем определеннее обобщенное состояние, тем определенней должен быть набор тестовых воздействий, которые нужно подать в этом обобщенном состоянии, а для полностью определенных обобщенных состояний, набор тестовых воздействий должен быть полностью определен. Обходчик может подавать только те тестовые воздействия, для которых точно известно, что они являются допустимыми в текущем состоянии модели.

В дальнейшем будем предполагать монотонность уточнений состояния целевой системы в процессе тестирования. Это означает, что ни на каком шаге тестирования не происходит потеря информации о состоянии целевой системы, информация может только накапливаться. Монотонность уточнений состояния тесно связана с детерминизмом требований. Пробелы в требованиях, также как и допущения о возможности недетерминированного поведения целевой системы ведут к потере информации в процессе тестирования.

---

[9] Факторизация задается отношением эквивалентности $\theta = \{ (x, y) \in \mathbb{M}_c \times \mathbb{M}_c \mid \mathbb{F}(x) = \mathbb{F}(y) \}$.

На каждом шаге тестирования тестовая система подает на целевую систему некоторое тестовое воздействие. В ответ на которое, целевая система изменяет свое состояние и выдает реакцию. Используя имеющуюся информацию и полученную реакцию, тестовая система корректирует свое представление о состоянии целевой системы (изменяет и уточняет состояние модели), после чего выносит вердикт о правильности или ошибочности поведения целевой системы. Таким образом, каждый шаг тестирования, с одной стороны, связан с модификацией состояния целевой системы, с другой — с получением дополнительной информации о нем.

В соответствии с традиционным требованием детерминизма обобщенной конечно-автоматной модели в технологии тестирования UniTESK, в нашем случае естественно потребовать детерминизма только в полностью определенных обобщенных состояниях. Очевидно, что требование детерминизма в не полностью определенных обобщенных состояниях является слишком жестким: неопределенные состояния открыты для уточнения, а возможных уточнений, вообще говоря, может быть несколько. Введем некоторые вспомогательные понятия.

Опр. Уточнение состояния целевой системы называется *существенным*, если оно вызывает уточнение обобщенного состояния. В противном случае, уточнение называется *несущественным*. ■

После существенного уточнения состояния целевой системы тестовая система переходит на другой информационный уровень. В предположении монотонности уточнений состояния целевой системы, тестовая система больше не вернется в обобщенное состояние, предшествующее существенному уточнению.

Опр. Модификация состояния целевой системы называется *существенной*, если она вызывает изменение обобщенного состояния. В противном случае, модификация называется *несущественной*. ■

Недетерминизм в существенных модификациях часто означает, что поведение целевой системы зависит от еще не определенной части состояния и, поскольку проверить правильность поведения в этом случае все равно нельзя, тестовая система должна исключать такие тестовые воздействия пока не накопит информацию о состоянии целевой системы, достаточную чтобы вынести вердикт о правильности или ошибочности поведения целевой системы.

Последнее требование можно ослабить. Если в процессе тестирования тестовая система перешла в обобщенное состояние, которое является уточнением одного из ранее пройденных обобщенных состояний, это означает, что тестовая система получила дополнительную информацию о состоянии целевой системы, следовательно, тестовую последовательность до настоящего момента можно рассматривать как носящую вспомогательный, уточняющий характер.

В вместо традиционного для технологии тестирования UniTESK требования сильной связности графа состояний обобщенной конечно-автоматной модели, естественно потребовать сильной связности для графа обобщенных состояний, расширенного дугами, ведущих из уточняющих обобщенных состояний в уточняемые. Такие дуги вполне естественны, поскольку находясь в некотором обобщенном состоянии, тестовая система в то же время находится и во всех уточняемых им обобщенных состояниях.

5.2.1. Графы с уточняемыми вершинами

В данном разделе вводится понятие графа с уточняющими вершинами, используемое для формального представления неопределенных обобщенных моделей. Вершины графа интерпретируются как обобщенные состояния целевой системы, дуги — как переходы между обобщенными состояниями, а раскраска дуг — как стимулы, инициирующие соответствующие переходы.

Опр. *Ориентированным графом* или просто *графом* $\mathbb{G}$ называется совокупность трех объектов $(\mathbb{GV}, \mathbb{GX}, \mathbb{GE})$: $\mathbb{GV}$ — множество *вершин*, $\mathbb{GX}$ — множество *раскрасок*, $\mathbb{GE} \subseteq \mathbb{GV} \times \mathbb{GX} \times \mathbb{GV}$ — множество *дуг*. ■

Заметим, что мы не рассматриваем раскраску дуг графа реакциями (для наших целей реакции не существенны), поэтому в дальнейшем будем называть элементы множества $\mathbb{GX}$ *стимулами*.

Опр. Граф $\mathbb{G}$ называется *конечным*, если множества $\mathbb{GV}$ и $\mathbb{GE}$ конечны. ■

Опр. Граф $\mathbb{G}$ называется *детерминированным*, если для любых двух дуг $(u_1, x_1, v_1) \in \mathbb{GE}$ и $(u_2, x_2, v_2) \in \mathbb{GE}$ из того, что $u_1 = u_2$ и $x_1 = x_2$ следует что $v_1 = v_2$. ■

Опр. Для графа $\mathbb{G}$ говорят, что стимул $x \in \mathbb{GX}$ *допустим* в вершине $u \in \mathbb{GV}$, если в графе $\mathbb{G}$ существует дуга вида (u, x, v). ■

Опр. *Маршрутом* в графе $\mathbb{G}$ называется любая (возможно пустая) последовательность смежных дуг: $\{(v_i, x_i, v_{i+1})\}_{i=0,n-1}$, где $n \geq 0$. При этом вершина v_0 называется *началом* маршрута, а вершина v_n — его *концом*. ■

Опр. *Обходом* графа $\mathbb{G}$ называется маршрут в графе $\mathbb{G}$, содержащий все его дуги. ■

Опр. Граф $\mathbb{G}$ называется *сильно-связным*, если для любых двух вершин $u, v \in \mathbb{GV}$ существует маршрут в графе $\mathbb{G}$ с началом в вершине u и концом в вершине v. ■

Опр. Если на множестве $\mathbb{GV}$ вершин графа $\mathbb{G}$ задано отношение уточнения $\Rightarrow$, будем говорить, что граф $\mathbb{G}$ является *графом с уточняемыми вершинами*. ■

Для удобства будем считать, что графы с уточняемыми вершинами всегда содержат полностью неопределенную вершину $\perp$ и являются *полными*, то есть в них для любой не полностью определенной вершины существует хотя бы одна уточняющая ее полностью определенная вершина. ■

Опр. Подграф $\mathbb{G}'=(\mathbb{GV}', \mathbb{GX}', \mathbb{GE}')$ графа с уточняемыми вершинами $\mathbb{G}=(\mathbb{GV}, \mathbb{GX}, \mathbb{GE})$ называется *полностью определенным*, если все его вершины полностью определены, то есть $\mathbb{GV}' \subseteq \mathbb{GV}_c$. ■

Опр. Подграф $\mathbb{G}'=(\mathbb{GV}', \mathbb{GX}', \mathbb{GE}')$ графа с уточняемыми вершинами $\mathbb{G}=(\mathbb{GV}, \mathbb{GX}, \mathbb{GE})$ называется его *информационной проекцией*, если:

- различные вершины из $\mathbb{GV}'$ несравнимы в отношении уточнения;
- добавление в $\mathbb{GV}'$ любой вершины $v \in \mathbb{GV} \backslash \mathbb{GV}'$ нарушает первое свойство;

- множество дуг $\mathbb{GE}'$ содержит все дуги графа $\mathbb{G}$, соединяющие вершины из $\mathbb{GV}'$. ■

Поскольку различные вершины информационной проекции несравнимы в отношении уточнения, будем рассматривать информационные проекции как традиционные графы, то есть графы без заданного на множестве их вершин отношения уточнения.

Опр. Информационная проекция графа с уточняемыми вершинами $\mathbb{G}$, содержащая все полностью определенные вершины $\mathbb{GV}$, называется *главной*. ■

Опр. Граф с уточняемыми вершинами называется *детерминированным*, если любая его информационная проекция детерминированна. ■

Опр. Граф с уточняемыми вершинами $\mathbb{G}$ называется *открытым*, если для любой его информационной проекции $\mathbb{G}'$, за исключением главной, существует дуга $(u, x, v) \in \mathbb{GE} \backslash \mathbb{GE}'$, раскрашенная стимулом, который отличается от стимулов дуг $\mathbb{GE}'$, выходящих из u. ■

Опр. Граф с уточняемыми вершинами $\mathbb{G}$ называется *сильно-связным*, если для любых двух вершин $u, v \in \mathbb{GV}$ либо существует маршрут, начинающийся в u и заканчивающийся в v, либо $v \Rightarrow u$. ■

Очевидно, что если граф с уточняемыми вершинами является сильно-связным, то любая его информационная проекция является сильно-связной.

Опр. Дуга (u, x, v) в графе с уточняемыми вершинами называется *уточняющей*, если $u \Rightarrow v$. Уточняющая дуга называется *строго уточняющей*, если $u \neq v$. ■

Опр. Маршрут $\{(v_i, x_i, v_{i+1})\}_{i=0,n-1}$ в графе с уточняемыми вершинами $\mathbb{G}$ называется *уточняющим маршрутом*, если для любых i и j таких что $0 \le i < j \le n$, либо v_i и v_j несравнимы в отношении $\Rightarrow$, либо $v_i \Rightarrow v_j$. Уточняющий маршрут называется *строго уточняющим*, если существуют i и j такие что $0 \le i < j \le n$, $v_i \Rightarrow v_j$ и $v_i \neq v_j$. ■

Опр. Граф с уточняемыми вершинами называется *монотонным*, если любой маршрут в нем является уточняющим. ■

Опр. *Уточняющим обходом* графа $\mathbb{G}$ называется маршрут в графе $\mathbb{G}$, начинающийся с полностью неопределенной вершины $\perp$ и содержащий все дуги главной информационной проекции $\mathbb{G}$. ■

Опр. Строго уточняющий маршрут $\{(v_i, x_i, v_{i+1})\}_{i=0,n-1}$ в графе с уточняемыми вершинами $\mathbb{G}$ называется *простым*, если никакой его префикс $\{(v_i, x_i, v_{i+1})\}_{i=0,m-1}$, где $m < n$, не является строго уточняющим маршрутом. ■

Рассмотрим уточняющий маршрут $\mathbb{P}=\{(v_i, x_i, v_{i+1})\}_{i=0,n-1}$ в графе с уточняемыми вершинами $\mathbb{G}$, начинающийся с полностью неопределенной вершины $v_0 = \perp$. Его можно представить в виде конкатенации: $\mathbb{P}=\mathbb{P}_0 \dots \mathbb{P}_{N-1}\ \mathbb{P}_N$ ($\mathbb{P}_j=\{(v_i, x_i, v_{i+1})\}_{i=k_j,k_{j+1}-1}$, $0 = k_0 < \dots < k_{N+1} = n$), в которой маршруты $\mathbb{P}_j$ для $0 \le j < N$ являются строго уточняющими, а маршрут $\mathbb{P}_N$ является уточняющим, но не является строго уточняющим маршрутом, то есть различные вершины маршрута $\mathbb{P}_N$ несравнимы в отношении уточнения.

Опр. Пусть $\mathbb{GV}_j=\{v_{k_j},\ldots,v_{k_{j+1}-1}\}$, где $0 \le j < N$, — все вершины маршрута $\mathbb{P}_j$ кроме последней, $\mathbb{GV}_N=\{s_{k_N},\ldots,s_{k_{N+1}}\}$ — все вершины маршрута $\mathbb{P}_N$. Множество вершин $\mathbb{GV}_j$, где $0 \le j \le N$, будем называть *уровнем неопределенности* j уточняющего маршрута $\mathbb{P}$. При этом для $j < N$ будем говорить, что маршрут $\mathbb{P}_j$ *уточняет вершины* с уровня неопределенности j до уровеня неопределенности j+1. ■

Заметим, что различные вершины одного уровня неопределенности несравнимы в отношении уточнения, поэтому они являются подмножеством множества вершин некоторой информационной проекции.

Рассмотрим следующий рисунок. Сплошные стрелки изображают обычные дуги графа; жирные стрелки изображают дуги внутри одного уровня неопределенности; стрелки, изображенные пунктиром, связывают уточняющие вершины с уточняемыми. Горизонтальные линии разделяют уровни неопределенности, которых на рисунке три.

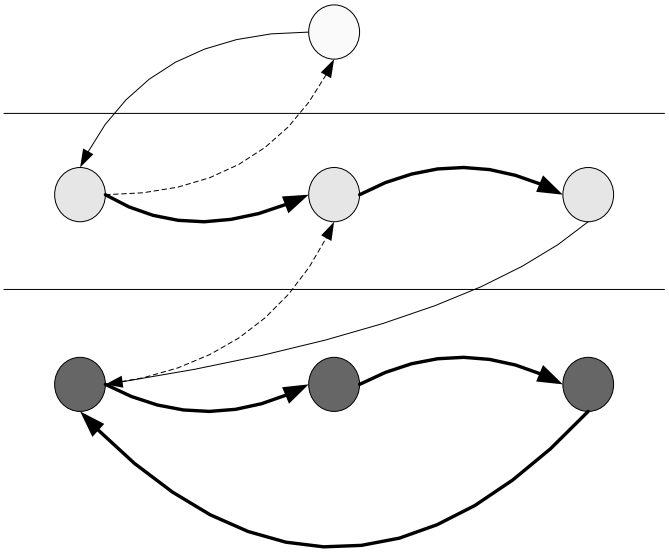

Рисунок 5. Уточняющий обход графа с уточняемыми вершинами.

Заметим, что уровни неопределенности могут пересекаться.

Пример. Рассмотрим граф с множеством вершин $\{true, false, \bot\}\times\{true, false, \bot\}$, каждая пара из которых соединена дугой. Маршрут $\mathbb{P}=\{(\bot, \bot), (false, \bot), (true, \bot), (false, false), (true, \bot)\}$[10], очевидно, является уточняющим. Ему соответствуют три уровня неопределенности:

- $\mathbb{GV}_0 = \{(\bot, \bot)\}$;
- $\mathbb{GV}_1 = \{(false, \bot), (true, \bot)\}$;
- $\mathbb{GV}_2 = \{(false, false), (true, \bot)\}$.

Видно, что уровни неопределенности $\mathbb{GV}_1$ и $\mathbb{GV}_2$ уточняющего маршрута $\mathbb{P}$ имеют общую вершину $(true, \bot)$. ■

[10] Здесь для удобства маршрут представлен в виде последовательности вершин.

5.2.2. Алгоритмы уточняющего обхода графов

В технологии тестирования UniTESK тестовая последовательность строится в результате обхода графа состояний обобщенной конечно-автоматной модели целевой системы. Для тестирования в условиях неполной информации мы предлагаем использовать определяемые ниже *алгоритмы уточняющего обхода*.

Опр. *Алгоритмом движения по графу* называется алгоритм, который в процессе своей работы строит маршрут в графе. ■

Опр. *Алгоритмом уточняющего движения по графу с уточняемыми вершинами* называется алгоритм, который в процессе своей работы строит уточняющий маршрут в графе. ■

Как и в работе [16], будем считать, что алгоритму предоставляются две специальные внешние операции `status()`, возвращающая идентификатор текущей вершины, и `call(x)`, которая осуществляет переход из текущей вершины по дуге, помеченной стимулом x. Предусловием операции `call(x)` является допустимость стимула x в текущей вершине. Маршрут строится алгоритмом как последовательность дуг, проходимых последовательными вызовами операции `call`.

Опр. Для маршрута в графе вершину будем называть *пройденной*, если этот маршрут содержит хотя бы одну инцидентную ей дугу. ■

Опр. Для маршрута в графе вершину будем называть *завершенной*, если этот маршрут содержит все выходящие из вершины дуги. ■

Опр. *Неизбыточным алгоритмом* называется алгоритм движения по графу, который зависит только от пройденной части графа и допустимости стимулов в текущей вершине. ■

Также как в работе [16], будем считать, что допустимость стимулов алгоритм может определить с помощью специальной внешней операции `next()`, которая возвращает стимул, неспецифицированным образом выбираемый среди не выбранных ранее стимулов, допустимых в текущей вершине (осуществляя тем самым итерацию стимулов в вершине). Если все стимулы, допустимые в текущей вершине, уже выбирались, будем считать, что `next()` возвращает специальный стимул ε.

Опр. *Алгоритмом уточняющего обхода графа* называется алгоритм, который в процессе своей работы строит уточняющий обход графа. ■

Нас будут интересовать только такие алгоритмы, которые останавливаются через конечное число шагов.

Рассмотрим общую схему неизбыточных алгоритмов обхода графов.

```
// пока есть незавершенные вершины
while(!completed())
{
  // если текущая вершина незавершена
  if(next(status()) != ε)
  {
    // подать очередной стимул
    call(next(status()));
  }
  else
  {
    // попасть в пройденную незавершенную вершину
    rollback();
  }
}
```

Операция `completed()` возвращает `true` тогда и только тогда, когда в графе все вершины завершены, то есть не существует вершин графа, из которых ведут не пройденные дуги.

Операция `rollback()` строит маршрут из текущей вершины в вершину, в которой есть еще не пройденные дуги. Подходы к реализации этой операции могут быть различными. Алгоритмы, основанные на обходе остова графа, выбирают самую дальнюю от корня (*поиск в глубину*) или самую ближнюю к корню (*поиск в ширину*) незавершенную вершину, достижимую из текущей [16]. Другим подходом (*жадный алгоритм*) является выбор ближайшей достижимой незавершенной вершины, но это требует больших затрат памяти[11].

Утв. Для графов с уточняемыми вершинами, являющихся:

- конечными,
- монотонными,
- детерминированными,
- открытыми,
- сильно-связными

существует неизбыточный алгоритм уточняющего обхода.

Док. Рассмотрим следующий алгоритм, точнее его идею, в рамках определенной ранее схемы. Поскольку граф является монотонным, любой маршрут в нем является уточняющим, следовательно, для любого маршрута определены уровни неопределенности. Алгоритм хранит в памяти некоторую структуру данных, связанных с текущим уровнем неопределенности, например, подобную описанной в работе [16].

Операция `complete()` возвращает `true` тогда и только тогда, когда все пройденные вершины на текущем уровне неопределенности завершены, то есть операция `next()` для них возвратила ε.

Операция `rollback()` строит маршрут из текущей вершины в незавершенную вершину текущего уровня неопределенности, в которой есть еще не пройденные

[11] В этом случае объем требуемой памяти равен как минимум $O(m(\log n + X) + nI)$ вместо $O(n(\log n + I + X))$, где n — число вершин, m — число дуг, I — размер идентификатора вершины, X — размер идентификатора стимула.

дуги. Например, это можно сделать на основе остова [16]. Поскольку различные вершины одного уровня неопределенности несравнимы в отношении уточнения, пройденный на текущем уровне неопределенности граф является подграфом некоторой информационной проекции исходного графа, а следовательно, является детерминированным и сильно-связным, что и требуется в [16].

В случае, если в операции `call` происходит выбор дуги в вершину, уточняющую одну из пройденных вершин текущего уровня неопределенности, текущий уровень неопределенности изменяется, структуры данных освобождаются, алгоритм работает так, как если бы новое состояние было начальным.

Поскольку граф конечен, детерминирован и сильно-связен, любая его информационная проекция является конечной, детерминированной и сильно-связной, поэтому внутри любого уровня неопределенности, за исключением последнего, можно гарантированно достичь вершину, в которой допустим стимул, такой что операция `call` обязательно выберет дугу, выводящую из этого уровня неопределенности. Такие вершина и стимул существуют, поскольку граф является открытым. Таким образом, за конечное число шагов алгоритм достигнет последнего уровня неопределенности, состоящего из полностью определенных вершин исходного графа. Обход которого завершит уточняющий обход исходного графа. ■

6. Простое расширение технологии UniTESK

Рассмотрим как можно расширить технологию тестирования UniTESK средствами тестирования в условиях неполной информации на примере инструмента разработки тестов CTesK [9]. В своих предложениях по расширению мы руководствовались тем соображением, что трудоемкость расширения должна быть минимальной.

6.1. Инструмент разработки тестов CTesK

Инструмент CTesK является реализацией концепции UniTESK для языка программирования C. Для разработки компонентов тестовой системы в нем используется язык SeC (*specification extension of C*), являющийся расширением ANSI C. Инструмент CTesK включает в себя транслятор из языка SeC в C, библиотеку поддержки тестовой системы, библиотеку спецификационных типов и генераторы отчетов. Для пользователей Windows имеется модуль интеграции в среду разработки Microsoft Visual Studio 6.0.

Компоненты тестовой системы UniTESK реализуются в инструменте CTesK с помощью специальных функций языка SeC, к которым относятся:

- *спецификационные функции* — содержат спецификацию поведения целевой системы в ответ на единичное тестовое воздействие, а также определение структуры тестового покрытия;
- *медиаторные функции* — связывают спецификационные функции с тестовыми воздействиями на целевую систему;
- *функция вычисления обобщенного состояния* — вычисляет состояние обобщенной конечно-автоматной модели целевой системы;
- *сценарные функции* — описывают набор тестовых воздействий для каждого достижимого обобщенного состояния.

6.2. Простое расширение инструмента CTesK

Для описания предикатов, которые из-за неопределенности состояния целевой системы могут принимать неопределенное значение, предлагается добавить в библиотеку CTesK перечислимый тип `Bool3`, представляющий значения истинности в трехзначной логике, вместе с основными функциями, реализующими отрицание, дизъюнкцию и конъюнкцию:

```
typedef enum
{
  True_Bool3    =  1, // истина
  False_Bool3   =  0, // ложь
  Unknown_Bool3 = -1  // неопределенное значение
} Bool3;

// отрицание
Bool3 not_Bool3(Bool3 arg);
// дизъюнкция
Bool3  or_Bool3(Bool3 lhs, Bool3 rhs);
// конъюнкция
Bool3 and_Bool3(Bool3 lhs, Bool3 rhs);
```

Для удобства работы со значениями типа `Bool3` также в библиотеку CTesK можно добавить модальные функции возможности и необходимости:

```
// модальная функция возможности
bool   may_Bool3(Bool3 arg);
// модальная функция необходимости
bool shall_Bool3(Bool3 arg);
```

Для задания отношения уточнения на множестве значений спецификационных типов предлагается добавить в спецификационные типы поле `.refines`, которое можно инициализировать функцией вида:

```
bool refines(Object *lhs, Object *rhs);
```

Функция возвращает значение `true` тогда и только тогда, когда спецификационный объект `rhs` уточняет спецификационный объект `lhs`.

Используя функцию `refines()`, обходчики в процессе построения тестовой последовательности могут учитывать изменения уровня неопределенности, тем самым подавая вспомогательные инициализирующие тестовые воздействия, которые сейчас приходится писать вручную.

Поскольку значение, возвращаемое постусловием спецификационной функции, имеет двузначный логический тип, а определить правильность или ошибочность поведения целевой системы в условиях неполной информации не всегда возможно, предлагается при определении структуры тестового покрытия выделять специальные ветви функциональности `Unknown`:

```
{ "Описание ветви функциональности", Unknown };
```

Выделенные ветви функциональности описывают ситуации, в которых оракул тестовой системы не обладает достаточной информацией, чтобы вынести однозначный вердикт о правильности или ошибочности поведения целевой системы. При попадании в одну из таких ветвей можно не проверять постусловие. Так как ветви `Unknown` носят вспомогательный характер, информация о их покрытии не должна попадать в отчеты о достигнутом покрытии.

Если после выполнения теста некоторые ветви функциональности, отличные от `Unknown`, оказались непокрытыми, это может быть связано со сценарной неполной требований. Возможно требования не определяют как достичь или идентифицировать соответствующие тестовые ситуации. Предположим, что оракул в качестве вердикта может возвращать неопределенное значение. Если после выполнения теста для некоторых ветвей функциональности вердикт оказывается неопределенным, это может быть связано с контрактной неполнотой требований. Цель тестирования — покрыть все ветви функциональности, вынеся при этом определенные вердикты, и достижение этой цели как правило связано с уточнением требований к целевой системе.

7. Заключение

В работе были рассмотрены вопросы функционального тестирования программных систем в условиях неполной информации. На базе технологии тестирования UniTESK предложен подход к разработке функциональных спецификаций и генерации функциональных тестов, основанный на использовании неопределенных значений для моделирования состояния целевой системы, а также трехзначной логики Клини для работы с неполными требованиями и описания свойств системы. Идеи предложенного подхода нашли свое применение в проекте по разработке открытого тестого набора OLVER для операционной системы Linux.

Литература

1. D. Dibois, H. Prade. *Théorie des possibilités. Applications à la representation des connaissances en informatique*. MASSON, Paris, Milan, Barselone, Mexico, 1988 (Дюбуа Д., Прад А. *Теория возможностей. Приложения к представлению знаний в информатике*: Пер. с фр. — М.: Радио и связь, 1990.)
2. Ю.П. Пытьев. *Возможность. Элементы теории и применения*. М.: Эдиториал УРСС, 2000.
3. http://www.unitesk.com — сайт, посвященный технологии тестирования UniTESK и поддерживающим ее инструментам.
4. А.В. Баранцев, И.Б. Бурдонов, А.В. Демаков, С.В. Зеленов, А.С. Косачев, В.В. Кулямин, В.А. Омельченко, Н.В. Пакулин, А.К. Петренко, А.В. Хорошилов. *Подход UniTesK к разработке тестов: достижения и перспективы*. (Опубликовано на http://www.citforum.ru/SE/testing/unitesk/.)
5. В.В. Кулямин, А.К. Петренко, А.С. Косачев, И.Б. Бурдонов. *Подход UniTESK к разработке тестов*. Программирование, №29(6), 2003.
6. А. Петренко, Е. Бритвина, С. Грошев, А. Монахов, О. Петренко. *Тестирование на основе моделей.* Открытые системы, №09/2003. (Опубликовано на http://citforum.ru/SE/testing/model/.)
7. M. Fitting. *Kleene's three-valued logics and their children*. Fundamenta Informaticae, №20, 1994.
8. http://www.linuxtesting.org — сайт Центра верификации операционной системы Linux.
9. http://www.unitesk.com/products/ctesk — страница инструмента разработки тестов CTesK.

10. http://www.ispras.ru — сайт Института системного программирования РАН.

11. I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. *KVEST: Automated Generation of Test Suites from Formal Specifications*. FM'99: Formal Methods. LNCS 1708, Springer-Verlag, 1999.

12. The RAISE Language Group. *The RAISE Specification Language*. Prentice-Hall BCS Practitioner Series. Prentice-Hall, Inc., 1993.

13. А.В. Хорошилов. *Спецификация и тестирование систем с асинхронным интерфейсом*. Институт системного программирования РАН, Препринт 12, 2006.

14. A. Tarski. *Introduction to Logic and to the Methodology of Deductive Sciences*. New York, 1941. (А. Тарский. *Введение в логику и методологию дедуктивных наук*. М.: ГИИЛ, 1948.)

15. Я.А. Слинин. *Современная модальная логика. Развитие теории алетических модальностей* (*1920 – 1960 гг.*). Издательство Ленинградского университета, 1976.

16. И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин. *Неизбыточные алгоритмы обхода ориентированных графов. Детерминированный случай.* Программирование, №5, 2003.