АНАЛИТИЧЕСКАЯ ВЕРИФИКАЦИЯ КОНФОРМНОСТИ

И.Б. Бурдонов, А.С. Косачев

1. Введение

Верификация конформности понимается как проверка соответствия исследуемой системы заданным требованиям. Когда говорят о верификации на основе формальных моделей, то предполагают, что система отображается в реализационную модель (реализацию), требования – в спецификационную модель (спецификацию), а их соответствие – в бинарное отношение конформности. В настоящей работе мы ограничиваемся, так называемыми, функциональными требованиями, которые выражаются в терминах взаимодействия реализации с окружением. Спецификация описывает требования, налагаемые на такое взаимодействие. Реализация конформна спецификации, если ее взаимодействие удовлетворяет этим требованиям.

Такая конформность может проверяться путем тестовых экспериментов, когда тест подменяет собой окружение исследуемой системы. Однако, если реализация известна, то проверка конформности может быть выполнена аналитически.

Одной из наиболее распространенных моделей является система помеченных переходов (LTS – Labelled Transition System). Известны как методы генерации программы по ее LTS-модели, так и методы извлечения LTS-модели из текста программы, работающие при некоторых ограничениях на используемые конструкции языка программирования. В данной статье мы не рассматриваем эти методы и предполагаем, что как спецификация, так и реализация заданы явно в виде LTS-моделей.

2-ой раздел статьи содержит основные положения теории конформности, изложенной в [1-3]. В 3-ем разделе предлагается алгоритм аналитической верификации конформности и приводится оценка его сложности.

2. Теория конформности

2.1. Семантика взаимодействия и его безопасность

Отношение конформности основано на той или иной модели взаимодействия. Мы рассматриваем семантики взаимодействия, которые используют только внешнее, наблюдаемое поведение системы и не учитывают её внутреннее устройство, отражаемое на уровне LTS-модели в понятии состояния. Мы можем наблюдать только такое поведение реализации, которое, во-первых, «спровоцировано» окружением (управление) и, во-вторых, наблюдаемо во внешнем взаимодействии. Такая семантика взаимодействия может моделироваться с помощью так называемой машины тестирования [1-6]. Она представляет собой «чёрный ящик», внутри которого находится реализация. Управление сводится к тому, что оператор машины (моделируя работу окружения) последовательно нажимает кнопки на клавиатуре машины, «разрешая» реализации выполнять те или иные действия. Наблюдения (на «дисплее» машины) бывают двух типов: наблюдение выполняемого реализацией действия и наблюдение отказа как отсутствия выполняемых действий.

Подчеркнём, что при управлении оператор разрешает реализации выполнять именно множество действий, а не обязательно одно действие. Например, в реактивных системах действия разбиваются на стимулы и реакции. Каждый стимул посылается в реализацию с помощью отдельной кнопки, которая разрешает единственное действие — прием стимула. Однако получение реакций от реализации управляется одной кнопкой, разрешающей реализации послать любую реакцию.

Каждой кнопке соответствует своё множество разрешаемых действий. Действие может наблюдаться, если реализация может его выполнить и оно разрешено нажатой кнопкой. Отказ наблюдается, если реализация не может выполнить ни одно из действий, разрешаемых нажатой кнопкой. После наблюдения (действия или отказа) кнопка отжимается, и все внешние действия запрещаются. Далее оператор может нажать другую (или ту же самую) кнопку.

Возможности взаимодействия определяются тем, какие «кнопочные» множества есть в машине, а также, для каких кнопок возможно наблюдение отказа. В данной работе для простоты изложения мы будем предполагать, что отказы наблюдаемы для всех кнопок. Тем самым, семантика взаимодействия определяется алфавитом внешних (наблюдаемых) действий **L** и набором **R** кнопочных множеств. Предполагается, что $\cup\mathbf{R}=\mathbf{L}$. Такую семантику мы называем **R**-семантикой.

Кроме внешних действий реализация может совершать внутренние (ненаблюдаемые) действия, обозначаемые символом τ. Эти действия всегда разрешены. Предполагается, что любая конечная последовательность любых действий совершается за конечное время, а бесконечная — за бесконечное время. Бесконечная последовательность τ-действий («зацикливание») называется *дивергенцией* и обозначается символом Δ. Дивергенция сама по себе не опасна, но при попытке выхода из неё, когда оператор нажимает какую-нибудь кнопку, он не знает, нужно ли ждать наблюдения или бесконечно долго будут выполняться только внутренние действия. Поэтому оператор не может ни продолжать взаимодействие, ни закончить его.

Кроме этого мы вводим специальное, также не регулируемое кнопками действие, которое называем *разрушением* и обозначаем символом γ. Оно моделирует любое нежелательное поведение системы, в том числе и ее реальное разрушение, которого нельзя допускать при взаимодействии (например, кнопка немедленного саморазрушения для систем оборонного назначения). Разрушение является одним из способов интерпретации неспецифицированного поведения. Исключение из рассмотрения взаимодействий, разрушающих реализацию, часто мотивируют тем, что реализация должна проверять корректность параметров обращения к ней [7,8]. Если параметры некорректны, реализация либо игнорирует обращение, либо сообщает об ошибке. Такое требование к реализации естественно, если это «система общего пользования»: в ней должна быть предусмотрена «защита от дурака». Однако при взаимодействии с внутренними компонентами или подсистемами, доступ к которым строго ограничен, взаимные проверки корректности обращений излишни. Такое часто встречается при обращении с параметрами сложной внутренней структуры и нетривиальными условиями корректности, когда накладные расходы на проверку неоправданно увеличивают трудозатраты на создание системы, её объём и время выполнения. Альтернативой в этом случае является строгая спецификация предусловий вызова операций [9]. Например, вызов операции освобождения участка памяти, который не был ранее получен через операцию запроса памяти, является нарушением предусловия. Проверке подлежит не поведение компонентов в ответ на некорректные обращения от других компонентов, а правильность обращения компонентов друг к другу. Иными словами, поскольку мы хотим убедиться в правильности реализации, а не окружения, нас не интересует поведение реализации при неправильном взаимодействии с ней. Разрушение в спецификации отмечает те ситуации, когда в реализации допускается любое поведение, в том числе и реальное разрушение.

Семантика разрушения предполагает, что оно не должно возникать при правильном поведении окружения. Взаимодействие, при котором не возникает разрушения и попыток выхода из дивергенции, называется безопасным.

2.2. LTS-модель и её трассы

LTS-модель – это ориентированный граф с выделенной начальной вершиной, дуги которого помечены некоторыми символами. Формально, LTS — это совокупность $\mathbf{S}=LTS(V_S, \mathbf{L}, E_S, s_0)$, где V_S – непустое множество состояний (вершин графа), $\mathbf{L}$ – алфавит внешних действий, $E_S \subseteq V_S \times (\mathbf{L} \cup \{\tau, \gamma\}) \times V_S$ – множество переходов (помеченных дуг графа), $s_0 \in V_S$ – начальное состояние. Переход из состояния s в состояние s' по действию z обозначается $s \text{—} z \rightarrow s'$. Обозначим $s \text{—} z \rightarrow =_{def} \exists s'\ s \text{—} z \rightarrow s'$. Маршрутом LTS называется последовательность смежных переходов: начало каждого перехода, кроме первого, совпадает с концом предыдущего перехода.

Шаг функционирования LTS внутри машины тестирования сводится к выполнению того или иного перехода, определённого в текущем состоянии и разрешаемого нажатой кнопкой (τ - и γ -переходы всегда разрешены). Если таких переходов несколько, выбирается один из них недетерминированным образом.

Состояние s *дивергентно* ($s\uparrow$), если в нём начинается бесконечная цепочка τ-переходов (в частности, τ-цикл); в противном случае состояние *конвергентно* ($s\downarrow$). Состояние *стабильно*, если из него не выходят τ - и γ -переходы. Отказ $P \in \mathbf{R}$ порождается стабильным состоянием, из которого нет переходов по действиям из P.

Добавим в каждом стабильном состоянии s виртуальные петли $s \text{—} P \rightarrow s$, помеченные порождаемыми отказами P, и Δ-переходы из дивергентных состояний $s \text{—} \Delta \rightarrow$. В полученной LTS рассмотрим маршруты, не продолжающиеся после Δ - и γ -переходов. Трассой назовём последовательность пометок на переходах такого маршрута с пропуском символов τ. Если маршрут с трассой σ начинается в состоянии s и заканчивается в состоянии s', то будем обозначать это $s \Rightarrow \sigma \Rightarrow s'$. Обозначим $s \Rightarrow \sigma \Rightarrow =_{def} \exists s'\ s \Rightarrow \sigma \Rightarrow s'$. Множество трасс, начинающихся в состоянии s, обозначим $\boldsymbol{T}(s) = \{\sigma \mid s \Rightarrow \sigma \Rightarrow\}$.

2.3. Гипотеза о безопасности и безопасная конформность

Определим отношение безопасности «*кнопка безопасна после трассы*»: нажатие кнопки P после трассы σ не вызывает попытку выхода из дивергенции (после трассы нет дивергенции) и не вызывает разрушения (после действия, разрешаемого кнопкой). При безопасном взаимодействии будут нажиматься только безопасные кнопки. Формально, кнопка P безопасна:

<u>в состоянии s</u>: P ***safe*** $s =_{def} s\downarrow\ \&\ \neg s \Rightarrow \langle \gamma \rangle \Rightarrow\ \&\ \forall z \in P\ \neg s \Rightarrow \langle z, \gamma \rangle \Rightarrow$;

<u>во множестве состояний S</u>: P ***safe*** $S =_{def} \forall s \in S\ P$ ***safe*** s;

<u>после трассы σ, начинающейся в состоянии s</u>: P ***safe*** s ***after*** $\sigma =_{def} P$ ***safe*** $(s$ ***after*** $\sigma)$,

где s ***after*** $\sigma =_{def} \{s' \mid s \Rightarrow \sigma \Rightarrow s'\}$.

Безопасность кнопок влечёт безопасность действий и отказов. Отказ P безопасен, если безопасна кнопка P. Действие z безопасно, если оно разрешается некоторой безопасной кнопкой P, то есть, $z \in P$. Трасса, начинающаяся в состоянии s, безопасна, если 1) $\neg s \Rightarrow \langle \gamma \rangle \Rightarrow$, 2) каждый символ трассы безопасен после непосредственно предшествующего ему префикса трассы. Множества безопасных трасс, начинающихся в

состоянии s, обозначим $\boldsymbol{Safe}(s)$. По умолчанию трассы и маршруты начинаются в начальном состоянии LTS.

Требование безопасности взаимодействия выделяет класс *безопасных* реализаций. Этот класс определяется следующей *гипотезой о безопасности*: реализация $\mathbf{I}$ *безопасна* для спецификации $\mathbf{S}$, если 1) в реализации нет разрушения с самого начала, если этого нет в спецификации, 2) после общей безопасной трассы реализации и спецификации любая кнопка, безопасная в спецификации, безопасна после этой трассы в реализации:

$\mathbf{I}$ ***safe for*** $\mathbf{S}$ $=_{\mathbf{def}}$ $(\langle\gamma\rangle \notin \boldsymbol{T}(s_0) \Rightarrow \langle\gamma\rangle \notin \boldsymbol{T}(i_0))$ & $\forall \sigma \in \boldsymbol{Safe}(s_0) \cap \boldsymbol{T}(i_0)$ $\forall P \in \mathbf{R}$

$(P$ ***safe*** s_0 ***after*** $\sigma \Rightarrow P$ ***safe*** i_0 ***after*** $\sigma)$.

Определим отношение *конформности*: реализация $\mathbf{I}$ *конформна* спецификации $\mathbf{S}$, если она безопасна и выполнено *тестируемое условие*: любое наблюдение, возможное в реализации в ответ на нажатие безопасной (в спецификации) кнопки, разрешается спецификацией:

$\mathbf{I}$ ***saco*** $\mathbf{S}$ $=_{\mathbf{def}}$ $\mathbf{I}$ ***safe for*** $\mathbf{S}$ & $\forall \sigma \in \boldsymbol{Safe}(s_0) \cap \boldsymbol{T}(i_0)$ $\forall P$ ***safe*** s_0 ***after*** σ

$\boldsymbol{obs}(i_0$ ***after*** $\sigma, P) \subseteq \boldsymbol{obs}(s_0$ ***after*** $\sigma, P)$,

где $\boldsymbol{obs}(M, P) =_{\mathbf{def}} \{u \in P \cup \{P\} \mid \exists m \in M$ & $m \Rightarrow\langle u\rangle \Rightarrow\}$ – множество наблюдений, которые возможны в состояниях из множества M при нажатии кнопки P.

При тестировании гипотеза о безопасности не проверяема и является его предусловием, целью тестирования является проверка тестируемого условия. При аналитической верификации оба условия можно проверить.

3. Алгоритм верификации

Прежде всего, сформулируем ограничения на семантику взаимодействия, реализацию и спецификацию, делающие возможной верификацию конформности за конечное время и рассматриваемые в данной статье. Мы будем предполагать выполнение следующего *условия конечности*: конечны алфавит действий $\mathbf{L}$ (отсюда следует конечность множества кнопок $\mathbf{R}$, каждого кнопочного множества и множества наблюдений $\mathbf{L} \cup \mathbf{R}$), а также реализация и спецификация; то есть, конечны множества их состояний и переходов.

Сначала опишем основную идею алгоритма верификации. Рассмотрим все состояния реализации, достижимые по трассам, безопасным в спецификации: $\cup\{i_0$ ***after*** $\sigma \mid \sigma \in \boldsymbol{Safe}(s_0)\}$. С каждым таким состоянием i свяжем семейство $\mathbf{S}(i)$ множеств состояний спецификации: $\mathbf{S}(i) = \{s_0$ ***after*** $\sigma \mid \sigma \in \boldsymbol{Safe}(s_0)$ & $i \in (i_0$ ***after*** $\sigma)\}$. Это семейство состоит из множеств состояний спецификации в конце трасс, безопасных в спецификации, имеющихся в реализации и заканчивающихся там в состоянии i. В процессе верификации мы будем строить эти семейства $\mathbf{S}(i)$, постепенно добавляя в них множества s_0 ***after*** σ. В самом начале проверяем условие $\langle\gamma\rangle \notin \boldsymbol{T}(s_0) \Rightarrow \langle\gamma\rangle \notin \boldsymbol{T}(i_0)$ как часть гипотезы о безопасности. Если это условие выполнено, то далее в каждый момент времени в каждом состоянии i должны быть выполнены оставшееся условие гипотезы о безопасности и тестируемое условие конформности: $\forall P \in \mathbf{R}$ $\forall S \in \mathbf{S}(i)$ $(P$ ***safe*** $S \Rightarrow P$ ***safe*** i & $\boldsymbol{obs}(\{i\}, P) \subseteq \boldsymbol{obs}(S, P))$. Если эти условия не выполнены, фиксируется ошибка. Если полностью построены семейства $\mathbf{S}(i)$ и ошибки не были обнаружены, то верификация заканчивается с положительным вердиктом.

Теперь опишем структуры данных и работу алгоритма.

Предварительно строятся структуры для спецификации и реализации. Спецификационные структуры не зависят от реализации и используются без модификации для верификации любой реализации в той же **R**-семантике. Соответственно, реализационные структуры не зависят от спецификации и пригодны для проверки ее конформности любой спецификации в той же **R**-семантике. Поэтому создание этих структур мы не будем учитывать при оценке сложности алгоритма верификации.

Для спецификации рассматривается семейство множеств в конце безопасных трасс: $\boldsymbol{safeder}(s_0) = \{s_0$ ***after*** $\sigma \mid \sigma \in \boldsymbol{Safe}(s_0)\}$. Для каждого множества $S \in \boldsymbol{safeder}(s_0)$ определим:

$\mathbf{A}(S) = (\cup\{P \cup \{P\} \mid P$ ***safe*** $S\}) \cup \{\tau\}$ – множество, состоящее из безопасных наблюдений (действий и отказов) и символа τ;

$\mathbf{B}(S, u) = \cup\{s$ ***after*** $\langle u\rangle \mid s \in S\}$ – множество постсостояний переходов по наблюдению u из состояний из S; также доопределим $\mathbf{B}(S, \tau) = S$. $\mathbf{B}(S, u) = \varnothing$, если таких переходов нет.

Фактически, при этом будет построена power-LTS спецификации, состояниями которой являются множества состояний $S = s_0$ ***after*** σ, где $\sigma \in \boldsymbol{Safe}(s_0)$, а переход $S —u→ S`$ означает, что $S` = \mathbf{B}(S, u) \neq \varnothing$.

Для реализации рассматривается множество состояний, достижимых по безопасным (в реализации) трассам: $\cup\boldsymbol{safeder}(i_0)$. Для каждого состояния $i \in \cup\boldsymbol{safeder}(i_0)$ определим:

$\mathbf{C}(i) = \langle i —u→ i` \mid u = \tau \vee \exists P$ ***safe*** i $u \in P\rangle$ — множество безопасных переходов из i.

$\mathbf{D}(i) = \langle i —u→ i` \mid \forall P$ ***safe*** i $u \notin P\rangle$ – множество опасных переходов из i.

Гипотеза о безопасности эквивалентна условию $\forall i —u\to i` \in \mathbf{D}(i)\ \ \forall S \in \mathbf{S}(i)\ \ u \notin \mathbf{A}(S)$. Тестируемое условие эквивалентно условию: $\forall i —u\to i` \in \mathbf{C}(i)\ \ \forall S \in \mathbf{S}(i)\ \ u \in \mathbf{A}(S) \Rightarrow \mathbf{B}(S,u) \neq \varnothing$.

Работа алгоритма сводится к пошаговому построению множеств $\mathbf{S}(i)$ и проверке безопасности и конформности на каждом шаге. Для работы алгоритма создается вспомогательный список $\mathbf{W}$ пар $(S, i —u\to i`)$, где $S \in \mathbf{S}(i)$ и $i —u\to i` \in \mathbf{C}(i) \cup \mathbf{D}(i)$. В самом начале множество $\mathbf{S}(i_0)=\{S_0\}$, где $S_0=\{s_0$ ***after*** $\langle\,\rangle\}$ — множество состояний спецификации после пустой трассы, а $\mathbf{W}$ содержит все пары $(S_0, i_0 —u\to i`)$, где $i_0 —u\to i` \in \mathbf{C}(i_0) \cup \mathbf{D}(i_0)$.

Шаг алгоритма заключается в следующем. Если список $\mathbf{W}$ пуст, алгоритм заканчивается с вердиктом «конформно». Иначе, выбираем первый элемент $(S, i —u\to i`)$ из списка $\mathbf{W}$, удаляя его из списка. Сначала проверяем гипотезу о безопасности: если $i —u\to i` \in \mathbf{D}(i)$ & $u \in \mathbf{A}(S)$, фиксируется ошибка и алгоритм заканчивается. Если ошибки нет, проверяем тестируемое условие. Если $i —u\to i` \in \mathbf{D}(i)$ или $i —u\to i` \in \mathbf{C}(i)$ & $u \notin \mathbf{A}(S)$, то шаг заканчивается. Если $i —u\to i` \in \mathbf{C}(i)$ & $u \in \mathbf{A}(S)$ & $\mathbf{B}(S,u)=\varnothing$, фиксируется ошибка и алгоритм заканчивается. Если ошибки нет, и $\mathbf{B}(S,u) \in \mathbf{S}(i`)$, шаг заканчивается. В противном случае добавляем $\mathbf{B}(S,u)$ в $\mathbf{S}(i`)$ и помещаем в список $\mathbf{W}$ все пары $(\mathbf{B}(S,u), i` —u` \to i``)$, где $i` —u` \to i`` \in \mathbf{C}(i`) \cup \mathbf{D}(i`)$; шаг заканчивается.

Легко показывается, что при завершении алгоритма, если фиксируется ошибка, то реализация не конформна спецификации, иначе — конформна.

Этот алгоритм, очевидно, легко распараллеливается: шаги алгоритма, соответствующие элементам списка $\mathbf{W}$, могут выполняться параллельно; если ошибка не обнаруживается, алгоритм заканчивается, когда заканчиваются все шаги, а список $\mathbf{W}$ пуст.

Оценим сложность алгоритма как время выполнения в наихудшем случае, в частности, когда распараллеливание ничего на дает. Поскольку каждая пара $(S, i —u\to i`)$ появляется в списке $\mathbf{W}$ не более одного раза, число шагов алгоритма равно $O(m \cdot 2^n)$, где $m=|E_I|$ число переходов реализации, а $n=|V_s|$ число состояний спецификации. На каждом шаге все действия по проверке принадлежности элемента множеству, выборке элемента (из $\mathbf{B}$) и добавлению элемента (в $\mathbf{S}$) могут выполняться за константное время, если множества $\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{D},\mathbf{S}$ заданы в виде массивов с прямой индексацией. Исключение составляет помещение в список $\mathbf{W}$ всех пар $(\mathbf{B}(S,u), i` —u\to i``)$, где $i` —u\to i`` \in \mathbf{C}(i`) \cup \mathbf{D}(i`)$. Суммарно по всем шагам помещается $O(m \cdot 2^n)$ пар. Если множества переходов $\mathbf{C}$ и $\mathbf{D}$ продублированы в виде списков, эти операции суммарно выполняются за время $O(m \cdot 2^n)$. Итоговая оценка $O(m \cdot 2^n)$. Сомножитель 2^n — это максимально возможное число состояний power-LTS спецификации. Если power-преобразование не увеличивает числа состояний спецификации, в частности, если спецификация детерминирована (нет τ -переходов и в каждом состоянии определено не более одного перехода по каждому действию), этот сомножитель заменяется на n. Интуитивно кажется, что оценка сложности алгоритма «в среднем» гораздо меньше, особенно при распараллеливании. Получение такой оценки может составить предмет дальнейших исследований.

ЛИТЕРАТУРА:

1. И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин "Формализация тестового эксперимента" // Программирование, 2007, No. 5
2. И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин "Теория соответствия для систем с блокировками и разрушением". «Наука», 2008
3. И.Б. Бурдонов "Теория конформности для функционального тестирования программных систем на основе формальных моделей". Диссертация на соискание учёной степени д.ф.-м.н., Москва, 2008
4. R.J. van Glabbeek "The linear time – branching time spectrum". CONCUR'90 (J.C.M. Baeten and J.W. Klop, ed.), LNCS 458, Springer-Verlag, 1990, pp 278–297
5. R.J. van Glabbeek "The linear time - branching time spectrum II; the semantics of sequential processes with silent moves". CONCUR'93 (E. Best, ed.), LNCS 715, Springer-Verlag, 1993, pp. 66–81
6. R. Milner "Modal characterization of observable machine behaviour". CAAP 81 (G. Astesiano and C. Bohm, ed.), LNCS 112, Springer-Verlag, 1981, pp. 25–34
7. J. Tretmans "Test Generation with Inputs, Outputs and Repetitive Quiescence" // Software-Concepts and Tools, Vol. 17, Issue 3, 1996
8. L. Heerink "Ins and Outs in Refusal Testing". PhD thesis, University of Twente, Enschede, The Netherlands, 1998
9. C.A.R. Hoare. "An axiomatic basis for computer programming" // Communications of the ACM, 12(10), October 1969