УДК 519.686.2

А.В. Колчин

МЕТОД НАПРАВЛЕНИЯ ПОИСКА И ГЕНЕРАЦИИ ТЕСТОВЫХ СЦЕНАРИЕВ ПРИ ВЕРИФИКАЦИИ ФОРМАЛЬНЫХ МОДЕЛЕЙ АСИНХРОННЫХ СИСТЕМ

Предложен метод направленного поиска для автоматического построения тестовых сценариев в процессе верификации. Метод использует определяемые пользователем в виде регулярных выражений цели тестирования и ограничения обхода поведения модели. Описаны стратегии управления поиском и техника ослабления эквивалентности трасс и состояний.

Введение

Техника формальной верификации, получившая название проверки модели (model checking [1]), является одним из наиболее перспективных и широко используемых подходов к решению проблемы автоматизации проверки правильности программ. В настоящее время наиболее популярны такие системы: SPIN [2], SMV [3], NuSMV [4], VeriSoft [5]. К недостаткам верификации можно отнести проблему комбинаторного взрыва числа состояний, а так же тот факт, что свойства проверяются на модели, а не на реальной системе. Для проверки соответствия требованиям реальной системы выполняется тестирование. Так как при тестировании проверить поведение программы во всех ситуациях невозможно, прибегают к определению критериев покрытия и ограничиваются требованием проверки классов тестовых сценариев, удовлетворяющих таким критериям. Трудоемкость создания тестов по функциональным спецификациям вручную (или с применением симуляторов) не приемлема для систем со сложной моделью поведения. Обостряется необходимость не только верификации моделей систем, но и автоматизации создания тестовых сценариев.

Системы, позволяющие автоматизировать построение тестовых наборов на этапе верификации обычно оценивают полноту покрытия тестовым набором по следующим метрикам и критериям: число выполненных операторов программы, ветвей, путей, число проверенных состояний данных и переходов между состояниями, покрытие граничных значений функций и предикатов модели, и др. Генерация тестовых сценариев поддерживается многими инструментами, использующими автоматные модели целевой системы. Такие инструменты хорошо подходят для верификации систем, при разработке которых используются формальные языки спецификаций, например, SDL [6], LOTOS [7], Estelle [8]. Вопрос о том, в каком соотношении находятся показатели тестового покрытия модели и требований к целевой системе в общем случае, как правило, не рассматривается. Таким образом, соответствия между полученными тестовыми сценариями, которые удовлетворяют вышеописанным критериям, и основными требованиями на функциональность целевой системы нет.

Системы верификации [2–4] предполагают, что свойство, которое необходимо проверить на модели, задано в виде некоторой формулы темпоральной логики. Этим можно было бы воспользоваться, сформулировав такую формулу, которая станет ложной на некотором пути (такой путь будет выдан верификационной системой в качестве контр-примера), и при этом путь будет рассматриваться как тестовый сценарий, покрывающий некоторое требование. Однако, на практике однозначного соответствия между состоянием (значением атрибутов), описываемым такой формулой и желаемой последовательностью событий нет; более того, высокий уровень необходимых знаний в области математической логики, предъявляемый пользователю, часто является существенным ограничивающим фактором, усложняющим использование формальных методов в процессе про-

мышленной разработки программного обеспечения. В работах [9, 10, 12] помимо спецификаций поведения системы, на вход подается сценарий тестирования, называемый обычно целью теста (test purpose), такой сценарий задается пользователем в виде последовательности сообщений, которыми обмениваются компоненты модели, например, в виде MSC [10, 11].

Направленный поиск и создание тестовых сценариев

Описываемый в данной работе метод позволяет пользователю задавать критерии покрытия, которые будут одновременно служить направлением поиска трасс для тестовых сценариев при обходе пространства поведения модели. В отличие от [2–4] желаемые свойства системы формулируются в терминах событий модели; в отличие от [12] метод не подразумевает внесения изменений в постусловия переходов исходной модели. Заметим так же, что верификация в системах [9, 10, 12] выполняется с фиксированным ограничением на длину трассы, таким образом, отсутствие результата не означает, что модель не допускает тестовый сценарий. Описываемый метод подразумевает накладывание ограничений на тестовый сценарий, что дает возможность проверить его допустимость. Еще одним важным отличием является то, что такие сценарии накладывают дополнительные ограничения на поиск, отсекая ветви поведения модели, не удовлетворяющие тестовому сценарию. Метод можно использовать в качестве определяемой пользователем эвристики при решении задач достижимости [13, 14].

Далее предложено расширение способов описания цели теста путем задания специальных регулярных выражений над алфавитом имен параметризированных переходов модели (далее образцов), определяющих условие покрытия и направление поиска.

Операционная семантика спецификаций может быть определена в терминах транзиционных систем. Транзиционная система – это тройка $T = (Q, q_0, \to)$, где Q – множество состояний, $q_0 \in Q$ – начальное состояние, $\to \subseteq Q \times Q$ – множество переходов, записывается $q \to q'$. Путь из состояния q_0 в q_n – (конечная) последовательность состояний $q_0 \to q_1 \to ... \to q_n$. Состояние q достижимо, если существует путь из q_0 в q. Размеченная транзиционная система $TL = (T, A)$ где $T = (Q, q_0, \to)$ – транзиционная система, в которой множество переходов имеет разметку: $\to = \bigcup_{a \in A} \xrightarrow{a}$. Трасса в TL это последовательность $a_1, a_2, ...a_n, ...$ такая, что существует путь $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} ... \xrightarrow{a_n} q_n ...$ Язык, ассоциированный с TL обозначается $\mathcal{L}(TL)$ – это набор всех трасс, выходящих из начального состояния. Как правило, события формальной модели ассоциируются с именами ее переходов, поэтому в данной работе множеством меток A является множество параметризированных имен переходов. Каждый переход на трассе параметризирован идентификатором инстанции (процесса). Цели тестирования задаются пользователем в виде образцов следующего вида:

$a.n$ – переход a с любым параметром на максимальной дистанции n;

$a(i).n$ – переход a с параметром i на максимальной дистанции n;

$\sim a$ – запрет перехода a с любым параметром;

$\sim a(i)$ – запрет перехода a с параметром i;

$a; b$ – операция конкатенации образцов;

$a \vee b$ – недетерминированный выбор образцов;

$a \parallel b$ – параллельная композиция образцов;

$a*$ – итерация образца a.

Назовем переходы, входящие в образец, наблюдаемыми. Два наблюдаемых перехода на трассе расположены на дистанции x, если между ними расположены $x-1$ переходов. Дистанция первого перехода в образце является расстоянием от начального состояния. Запреты действуют до обнаружения следующего допустимого наблюдаемого перехода и не

могут быть последними в образце (за исключением использования внутри итерации). Параллельная композиция образцов используется для случаев, когда модель имеет несколько параллельно работающих компонент (процессов), причем дистанция подсчитывается для каждого параллельного участка образца отдельно. Построенная трасса удовлетворяет образцу, если она содержит переходы в непротиворечащей заданному образцу последовательности, на дистанциях, не превышающих указанные. Например, образец

$$trA(ms1).2;(trB.2\vee \sim trZ(ms1);trC.4)$$

имеет примеры удовлетворяющих трасс:

$$trS(ms1),trA(ms1),trZ(ms1),trB(ms2),$$

$$trA(ms1),trX(ms1),trC(ms2)$$

пример не удовлетворяющей трассы:

$$trA(ms1),trX(ms1),trZ(ms1),trC(ms2),trB(ms1)$$

Как видно, выражения, не содержащие итерации, всегда имеют максимальную длину трассы, удовлетворяющей образцу; более того, в отличие от [9], определить, что трасса не удовлетворяет образцу, можно не достигая этого максимума, так как всегда определена максимальная дистанция между событиями. В приведенном примере максимальная длина трассы 6, в то время как трасса

$$trA(ms1),trZ(ms1),trC(ms1)$$

уже не имеет допустимого образцом продолжения и может быть прервана.

Семантически такие образцы задают «контрольные точки» поведения модели и так же определяют критерий выбора построенных трасс (вариантов конкретного поведения системы) для последующего их использования в качестве тестовых сценариев [14]. Так, предполагаемые поведения моделируемой системы, описанные пользователем, проверяются на допустимость, одновременно накладывая ограничения на поиск. По заданным образцам строится соответствующий детерминированный конечный автомат. Для различных видов регулярных выражений существуют алгоритмы различной вычислительной сложности [15, 16]. Заметим, что на практике удовлетворительный результат можно получить, используя жадные алгоритмы линейной сложности.

Обход пространства поведения, реализующий адаптированный алгоритм Тарьяна [17], модифицируется путем добавления перед шагом вглубь вызова специальной функции, реализующей автомат, распознающий образцы, при этом *полным* состоянием модели будет синхронное произведение состояний модели (значений атрибутов) и состояний автомата, распознающего образцы. Если текущая трасса удовлетворяет образцу, она будет сохранена на жесткий диск, если не исчерпан соответствующий лимит (пользователь определяет нужное количество трасс по каждому образцу). Если лимит исчерпан, образец исключается из списка актуальных. Если данная функция возвращает на выходе `false`, это означает, что достигнута ситуация, из которой любое продолжение текущей трассы не приведет обнаружению вхождения очередного символа (имени наблюдаемого перехода) хотя бы одного актуального образца. Такая ситуация является дополнительным критерием завершения генерации текущей трассы, таким образом, алгоритм выполнит шаг назад. При этом, если трасса содержит исторически максимальной длины префикс хотя бы одного образца, она будет сохранена на диск с соответствующей пометкой. Таким образом, если полного вхождения какого-либо образца не обнаружено, будет построена трасса, удовлетворяющая максимальному префиксу этого образца. Дополнительным для сохранения трассы является условие вхождения в трассу перехода, не входящего ни в одну из уже сохраненных трасс. В итоге будет построен набор трасс, включающий как предполагаемые пользователем поведения модели, так и все достигнутые переходы модели.

Использование итерации

Для обнаружения бесконечных циклов используется итерация (замыкание Клини). Допускается использование не более одной итерации и только в конце образца. Необходимо отметить, что под циклом в данном случае понимается цикл полных состояний, т.е. состояний автомата регулярного выражения и значений атрибутов модели.

Например, используя итерацию, можно проверить, приходит ли модель системы в устойчивое состояние при отсутствии внешних сигналов. Пусть $trE1$ и $trE2$ – единственные воздействия (переходы) внешней среды, а trA – переход некоторого единственного процесса ms. Пусть так же задан образец

$$trE1.2; trA(ms).4; (\sim trE1 \parallel \sim trE2)*$$

Тогда трасса, удовлетворяющая образцу должна иметь цикл, в котором нет переходов $trE1$ и $trE2$, а префикс этой трассы должен включать переходы $trE1$ и $trA(ms)$ на соответствующих дистанциях. Таким образом, после воздействия среды событием, размеченным переходом $trE1$ и последующей реакцией переходом trA, процесс ms не приходит в устойчивое состояние (в цикл не входит ни одно событие от внешней среды).

Стратегии

Одним из подходов к решению проблемы комбинаторного взрыва является привлечение пользователя (эксперта предметной области) к процессу верификации. Большинство из верификационных систем использует заданные пользователем абстракции [18] и специальные состояния, описанные в виде формул темпоральной логики для отсечения ветвей поведения («hints», «restricted states», «fairness constraints») [19, 20]. В работе [21] предложен метод автоматического направления поиска нарушения свойств модели, в основе которого лежит структурный анализ переходов модели и проверяемых свойств. В работе [22] описан верификатор, использующий эвристические методы для направления поиска ошибок модели. В отличие от упомянутых, данная работа предлагает интерфейс к правилам и инструкциям обхода пространства поведения модели, посредством которого пользователь может задавать свои эвристики в терминах переходов (событий) модели. По заданным правилам строится автомат, работающий синхронно с автоматом распознавания образцов, динамически изменяющий ограничения пространства поиска (длина трассы, терминальные состояния), приоритеты для инстанций (процессов) и переходов, а так же реализующий механизмы включения и выключения инстанций и переходов на некоторых участках поведения модели. Тело основного цикла поиска в глубину модифицируется путем добавления предварительной сортировки переходов в соответствии с приоритетами и исключения выключенных переходов.

Ослабление эквивалентности трасс

В данном случае ослабление трассовой эквивалентности подразумевает разделе-ние модели на тестируемую и тестирую-щую инстанции. Такой подход применим в случае, когда переходы модели представ-ляют собой тройки вида $A \xrightarrow{p} B$, где A и B пред- и предусловия, а p описывает процесс – наблюдаемые события перехода. В работах [14, 23] процесс описывается MSC диаграммой, а результирующая трас-са записывается в виде MSC сценария, состоящего из последовательности событий соответствующих трассе переходов. Так, для ослабления эквивалентности, пользователь выделяет два набора инстанций, после чего трассовая эквивалентность рассматривается с точностью до событий тестируемых инстанций. Описанная далее процедура `Save_Abstract_Trace` строит абстрактные трассы, используя заданную эквивалентность. Трасса состоит из переходов модели, которые в свою очередь описывают события, причем не обязательно одной инстанции (процесса). Функция `instance` идентифицирует инстанцию очередного события на трассе.

Вход: полная трасса Trace, множество тестируемых инстанций Tested_Instances и идентификатор теста Test_ID.
Выход: абстрактная трасса Abstract_ Trace.

```
Save_Abstract_Trace := proc(Trace, Tested_Instances, Test_ID)
                       local(Abstract_Trace, Transition, Event) begin
   Abstract_Trace ← ∅;
   for_each Transition ∈ Trace do
      for_each Event ∈ Transition do
         if(instance(Event) ∈ Tested_Instances) do
         Abstract_Trace ← {Abstract_Trace ∪Event};
   Save_Trace(Abstract_Trace, Test_ID)
end
```

На практике актуальна минимизация числа тестов. Процедура Save_Trace обеспечивает поиск вхождения одного теста в другой. В случае обнаружения, новая трасса либо не добавляется к тестовым сценариям, либо заменяет ранее добавленный.

Вход: абстрактная трасса Abstract_Trace и идентификатор теста Test_ID.
Выход: обновленное множество Stored.

```
Save_Trace := proc(Abstract_Trace, Test_ID)local(T) begin
   for_each T ∈ Stored do begin
      if(Abstract_Trace ⊆ T.trace) do return;
      if (Abstract_Trace ⊃ T.trace) do begin
         *T.trace ← Abstract_Trace;
         *T.test_id ← {T.test_id ∪Test_ID};
         return
      end
   end
   Stored ← {Stored ∪ (Abstract_Trace, Test_ID)}
end
```

В результате множество Stored будет содержать минимизированный набор пар состоящих из абстрактной трассы и списка идентификаторов тестовых целей.

Ослабление эквивалентности состояний

Ослабление эквивалентности состояний в данном случае является дополнительной возможностью для пользователя задавать стратегию поиска путем определения правил абстракции состояний. Такие правила представляют собой условия для игнорирования либо изменения значений некоторых атрибутов пройденных состоя-ний. Например, для системы, в которой после заполнения информации в полях некоторого интерфейсного диалогового окна была выполнена команда «отменить», введенные значения не играют роли, и в следующий раз при выполнении такой ко-манды состояние модели будет идеентифи-цировано как пройденное, если найдется пройденное ранее состояние, эквивалент-ное данному без учета значений полей диа-лога. Далее приведен пример такой функ-ции user_abstraction, определяемой пользователем.

Вход: конкретное состояние модели
Выход: абстрактное состояние, построенное согласно правилам пользователя

```
user_abstraction := proc(S)local(A)begin
```

```
   A ← S;
   if(S.command = “cancel”) do ignore(A, Dialog.data);
   if(S.command = “send_sms” & S.sms_size > S.max_sms_size) do
      A.sms_size ← S.max_sms_size + 1
   return A
end
```

Заметим, что в данном примере второе правило абстрагирует реальное значение атрибута, хранившего размер сообщения. Таким образом пользователь сообщает системе, что нет смысла различать размер сообщения, превосходящего максимально допустимый, и что в таком случае достаточно лишь зафиксировать факт превышения допустимого значения. Функция `ignore` исключает атрибут (`Dialog. .data`) из состояния (`A`).

Описанная далее процедура исполь-зуется для сохранения и проверки прой-денных состояний с учетом пользователь-ских абстракций; применяется в процессе обхода поведения модели.

Вход: конкретное состояние модели
Выход: обновленное абстрактным состоя-нием множество `Visited`

```
visited := proc(S)local(v)begin
   for_each v ∈ Visited do begin
      flag ← true;
      for_each a ∈ v do
         if(v->a.value ≠ S->a.value) do begin
            flag ← false;
            break
         end
      if(flag = true) do return true
   end
   v ← user_abstraction(S);
   Visited ← {Visited ∪ v};
   return false
end
```

Пример

Далее приведена типичная тестовая процедура и соответствующий тестовый сценарий из промышленного телекоммуникационного проекта [14].

Вход: тестовая процедура (последовательность действий): 1) X создает Y; 2) X проверяет, что Y создан; 3) X удаляет Y; 4) X и Y получают уведомление об удалении; 5) X проверяет, что Y больше нет в списке контактов; 6) Y проверяет, что X больше нет в списке контактов.

В результате по тестовой процедуре было построено такое выражение, задающее цель тестирования:

```
sec_T1943 = (
   create_contact(ms1).30 ;
   PTT_Contacts04(ms1).10 \/ PTT_Contacts05(ms1).10 ;
   PTT_Accept_Invitation(ms2).15 ;
   show_indiv_cont(ms1).10 ;
   delete_indiv_cont(ms1).3 ;
   PTT_Receive_Notice06(ms1).7 || PTT_Receive_Notice07(ms2).7 ;
   show_empty_indiv_cont(ms1).10 || show_ empty_indiv_cont(ms2).10
)
```

Выход: трасса (тестовый сценарий): полная последовательность событий модели, соответствующая тестовой процедуре.

Применение технологии направленного поиска дало возможность не только выявить ошибки в спецификациях, но и создать набор тестовых сценариев (~100 трасс), обеспечивающих необходимое функциональное покрытие. События MSC трассы были транслированы в последовательности команд тестируемого устройства, что дало возможность автоматизировать исполнение построенных таким методом тестов [14].

Выводы

Предложенный метод является разновидностью ограниченной проверки модели (bounded model checking), в качестве ограничителей использует цели тестирования, определяемые пользователем. Метод проверяет допустимость предполагаемых поведений, по сути осуществляя валидацию модели. Использование коротких дистанций и запретов между контрольными точками образцов позволяет эффективно находить труднодостижимые состояния. Методы ослабления эквивалентности трасс и состояний существенно сокращают количество тестовых сценариев обеспечивающих необходимое покрытие и время поиска.

1. http://en.wikipedia.org/wiki/Model_checking
2. *Ben-Ari. M.* Principles of Spin. // Springer Verlag. – 2008. – P. 216.
3. *Burch J., Clarke E., McMillan K., Dill D., and Hwang L.* Symbolic model checking: 10^20 states and beyond // Information and Computation. – 1992. – Vol. 98, N 2. – P. 142–170.
4. *Cimatti A., Clarke E., Giunchiglia E., and others.* NuSMV 2: An OpenSource Tool for Symbolic Model Checking // In Proceeding of International Conf. on Computer-Aided Verification, Copenhagen, Denmark. – 2002. – P. 359–364.
5. *Godefroid P.* Partial-order methods for the verification of concurrent systems – an approach to the state-explosion problem // Lecture notes in computer science, Springer-Verlag. – 1996. – Vol. 1032. – P. 143.
6. *ITU*-T Recommendation Z.100 – Specification and Description Language (SDL), ITU. – 2002.
7. ISO/IEC. Information Processing Systems - Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807:1989, International Organization for Standardization, Geneva, Switzerland. – 1989.
8. ISO/TC97/SC21. Information Processing Systems – Open Systems Interconnection – Estelle – A Formal Description Technique based on an Extended State Transition Model. ISO 9074:1997, International Organization for Standardization, Geneva, Switzerland. –1997.
9. *Fernandez J., Jard C., Jeron T., Nedelka L., and Viho C.* An experiment in automatic generation of test suites for protocols with verification technology // Science of Computer Programming. – 1997. – Vol. 29, N 1–2. – P. 123–146.
10. *Grabowski J., Hogrefe D., Nahm R.* Test case generation with test purpose specification by MSCs // In Elsevier Science B.V. (North-Holland), editor, 6th SDL Forum. – 1993. – P. 253–266.
11. ITU-T Recommendation Z.120 – Message Sequence Chart (MSC), ITU. – 1996.
12. *Bourdonov I., Kossatchev A., Kuliamin V., and Petrenko A.* UniTesK Test Suite Architecture // In Proc. of FME 2002, LNCS 2391, Springer-Verlag. – 2002. – P. 77–88.
13. *Колчин А.В.* Направленный поиск в верификации формальных моделей // Тези доп. Міжнар. конф. «Теоретичні та прикладні аспекти побудови програмних систем TAAPSD'2007». – К.: НаУКМА, Національний ун-т ім. Т.Г. Шевченка, Інститут програмних систем НАН України. – 2007. – С. 256–258.
14. *Баранов С.Н., Котляров В.П., Летичевский А.А.* Индустриальная технология автоматизации тестирования мобильных устройств на основе верифицированных поведенческих моделей проектных спецификаций требований // Труды Междунар. науч. конф. «Космос, астрономия и программирование». – С-Пб: СПбГУ, 2008. – С. 134–145.
15. *Aho A.* Algorithms for finding patterns in strings // Handbook for theoretical computer science, MIT Press. – 1990. – Vol. A. – P. 257–300.
16. *Smyth B.* Computing Patterns in Strings // ACM Press Books. – 2003. – P. 440.
17. *Колчин А.В.* Автоматический метод оперативного построения абстракций при верификации формальных моделей асинхронных систем // Искусственный интеллект. – 2008. – № 4. – С. 690–705.
18. *Clarke E.* Model checking and abstraction // In ACM Transactions on programming languages and systems. – 1994. – Vol. 16, N 5. – P. 1512–1542.
19. *Bloem R., Ravi K., and Somezi F.* Symbolic guided search for CTL model checking // In Design Automation Conference. – 2004. – P. 29–34.
20. *Barner S., Glazberg Z., and Rabinovitz I.* Wolf - bug hunter for concurrent software using formal

methods // In Computer Aided Verification. – 2005. – P. 153–157.
21. *Peranandam P., Weiss R., Ruf J., Kropf T. and Rosenstiel W.* Dynamic guiding of bounded property checking // In IEEE International High Level Design Validation and Test Workshop. – 2004. – P. 15–18.
22. *Edelkamp, S., Leue S., and Lafuente A. L.* Directed explicit-state model checking in the validation of communication protocols // International journal on software tools for technology transfer. – 2003. – N 5. – P. 247–267.
23. *Baranov S.N., Kapitonova J.K., Kolchin A.V., and others.* Tools for Requirements Capturing Based on the Technology of Basic Protocols // Proc. of St. Petersburg IEEE Chapter. – 2005. – P. 92–97.

Получено 04.06.2008

Об авторе:
Колчин Александр Валентинович,
младший научный сотрудник.

Место работы автора:
Институт кибернетики
им. В.М. Глушкова НАН Украины.
Тел.: (044) 200 8423.
e-mail: kolchin_av@yahoo.com