Параллельное построение пространства состояний конечной системы

Коротков Иван Андреевич
Кафедра ИУ7 МГТУ им. Баумана
e-mail: korotkov2@mail.ru

Введение

При разработке параллельных алгоритмов и протоколов взаимодействия часто возникает необходимость их верификации на соответствие определенным условиям целостности. Например, параллельного алгоритма не должен иметь взаимоблокировок, а в протоколе взаимодействия все участвующие стороны должны всегда достигать конечного состояния.

Функциональное тестирование системы позволяет выявить не все ошибки, а лишь наиболее часто встречающие. В то же время, определенные классы ошибок требуется полностью исключить. Например, если новое расширение сетевого протокола с улучшенным контролем трафика приводит к потере данных только в 10^{-3}% случаев, это может остаться не выявленным при тестировании, однако при использовании на 100 млн. машин приведет к потерям на 1000 из них, что уже довольно много. Для их выявления и подходит верификация самой модели, которая путем перебора всех состояний системы проверит, возможен ли такой сценарий функционирования системы, при котором она приходит в недопустимое состояние.

Основным программным средством для подобной проверки конечных моделей является SPIN. Для описания модели в нем используется язык описания Promela (PROtocol Meta Language).

При верификации модели SPIN выполняет исчерпывающий поиск в глубину по пространству состояний и, при достижении недопустимого состояния, сохраняет сценарий, приведший к приходу в это состояние.

Поскольку размер пространства состояний модели растет экспоненциально, даже для среднего размера моделей оно насчитывает до 10^9-10^{10} состояний, требующих десятки гигабайт памяти для хранения множества пройденных состояний. Это делает проверку больших моделей нереализуемой практически. Для сокращения объема требуемой памяти используются различные оптимизации — сжатие хранимых состояний, битовое хэширование пространства состояний (описано далее) и т.д. В данной работе предлагается другой подход — параллельное выполнение на нескольких узлах вычислительной сети с распределением хранимых состояний между ними.

Последовательная генерация состояний

Процесс обычного (последовательного) построения пространства состояний можно рассмотреть на примере упомянутого ранее верификатора SPIN.

Модель конечной системы на языке Promela описывается в виде ряда взаимодействующих процессов, обменивающихся сообщениями через каналы ограниченной емкости. Состояние системы представляет собой набор значений глобальных переменных и локальных состояний каждого из процессов. Состояние процесса включает в себя его «счетчик инструкций» и набор локальных переменных. Множество переходов из текущего состояния представляет собой объединение возможных переходов для каждого процесса из его локального состояния.

Пример модели на на этом языке, описывающей семафор Дейкстры и 3 процесса, захватывающих его:

```
mtype { p, v };
chan sema = [0] of { mtype };
active proctype Dijkstra()
{       byte count = 1;
        do
        :: (count == 1) ->
                sema!p; count--
        :: (count == 0) ->
                sema?v; count++
        od
}
active [3] proctype user()
{        do
         :: enter: sema?p; /* enter critical section */
             crit: skip;    /* critical section */
                   sema!v; /* leave critical section */
         od
}
```

Последовательный алгоритм построения пространства состояний, используемый верификатором SPIN, можно представить следующим псевдокодом:

```
Visited = []
def StateSpaceDFS(state):
      if not state in Visited:
            Visited += state
            for each transition t from state:
                  next_state = next state after t
```

```
                    StateSpaceDFS(next_state)
StateSpaceDFS(start)
```

Последовательное построение пространства состояний с ростом размера модели быстро становится невозможным из-за нехватки памяти для хранения множества достигнутых состояний (**Visited**). Можно предложить два решения проблемы: использование оптимизаций, сокращающих расход памяти, и параллельная генерация состояний с распределением хранимых состояний между различными узлами вычислительной сети.

Битовое хэширование состояний

Одной из оптимизаций, позволяющей сократить расходуемый объем памяти, является так называемое битовое хэширование состояний (*bit-state hashing*). Суть его заключается в том, что вместо традиционной хэш-таблицы, с открытой индексацией или со списками, для хранения состояний используется битовая таблица. Каждое состояние S представлено в ней битом с номером i = hash(S). Нулевой бит означает, что состояние еще не было достигнуто, единичный — было. В чистом виде битовое хэширование не применяется из-за высокой вероятности коллизий [1]. Возможные решения включают в себя:

1. Использование нескольких независимых хэш-функций. Каждое состояние имеет, таким образом, несколько индексов $\{i_1 = hash_1(S), i_2 = hash_2(S), ... i_n = hash_n(S)\}$ и считается найденным в хэш-таблице только в том случае, если все биты $\{i_1, i_2, ... i_n\}$ установлены.
2. Использование традиционной хэш-таблицы, в которой вместо самих состояний хранятся их хэш-коды, полученные при помощи хэш-функции hash*(S) большей разрядности (например, 64 бита). Функция **hash*** должна быть независима от **hash**, используемой для индексации в этой таблице [1].

Битовое хэширование состояний используется в оригинальном верификаторе SPIN и далее не рассматривается.

Параллельная генерация состояний

Возможны два варианта параллельной генерации состояний:

1. Распределенное хранилище состояний. Состояния генерирует только один узел, а для хранения используются все узлы. Каждое состояние имеет свой однозначно вычислимый номер узла. Преимуществом данного подхода является возможность по-прежнему использовать поиск в глубину, необходимый для поиска циклов. Недостаток заключается в том, что каждое новое состояние требует удаленного вызова (обращения к другому узлу) для проверки, было ли он уже достигнуто.
2. Распределенная генерация состояний. Каждый узел является одновременно

хранилищем и генератором. Этот вариант предпочтителен, поскольку используется вычислительная мощность всех узлов и сокращается число удаленных вызовов за счет тех случаев, когда новое состояние принадлежит тому же самому узлу. Хотя каждый узел может выполнять с локальной точки зрения поиск в глубину, глобальный порядок достижения состояний нарушается, и поиск циклов становится затруднителен. В работах [2,3] было предложено несколько различных алгоритмов, совмещающих поиск в ширину с последующим нахождением циклов за счет определенной потери производительности. Зачастую верификация модели требует лишь проверки корректности конечных состояний и выполнения инвариантов, а в этом случае поиск циклов не требуется, что позволяет использовать в данном варианте обычный поиск в ширину.

Параллельная версия алгоритма для второго случая выглядит следующим образом:

```
Visited = []
Queue = []
if NodeId = 0:
      Queue += start_state
def ParStateSpaceBFS():
      while not empty(Queue):
            state <- Queue
            node = StateNode(state)
            if NodeId = node:
                  if not state in Visited:
                        Visited += state
                        for each transition t from state:
                              next_state = next state after t
                              Queue <- next_state
            else:
                  node.Queue <- state
ParStateSpaceBFS()
```

Распределение состояний между узлами

В приведенном коде для определения номера узла, хранящего сообщения, используется функция **StateNode**. Эта функция должна обладать следующими характеристиками:

- она должна зависеть только от самого состояния, поскольку одно и то же состояние может генерироваться различными узлами в результате различных переходов;
- она должна по возможности распределять состояния между узлами равномерно;
- она должна обладать определенной локальностью — по возможности новые состояния должны принадлежать тому же узлу, что и исходное, для уменьшения числа удаленных

вызовов.

Наиболее простым подходом является в качестве **StateNode** использовать хэш-функцию от всего состояния. Это обеспечит первые 2 условия: если выбрана подходящая хэш-функция, распределение будет достаточно равномерным. Однако, третье условие при этом не соблюдается, поскольку все новые состояния имеют равные шансы принадлежать любому узлу независимо того, на каком узле они были сгенерированы.

Пусть число узлов — N, состояний — S, переходов между ними — T. В случае равномерного распределения состояний между узлами вероятность того, что следующее состояние будет принадлежать текущему узлу равняется 1/N. Следовательно, вероятность того, что потребуется удаленный вызов, равна $1-1/N$, а среднее число удаленных вызовов в течение всей генерации составит $T(1-1/N)$, что при больших значениях N стремится с T. Как показывают эксперименты в [4], столь частые пересылки состояний негативно отражаются на производительности, поэтому необходимо найти более удачное распределение состояний, которое бы удовлетворяло условию локальности.

В [4] предлагается для распределения состояний по узлам хэшировать не все состояние целиком, а лишь локальное состояние одного процесса. В этом случае удаленный вызов будет происходить лишь при переходах, изменяющих локальное состояние этого процесса.

Пусть P — число процессов в модели, k — среднее число процессов, затрагиваемых переходом (состояние которых меняется при переходе). k больше 1, так как возможна (атомарная) передача сообщений между процессами, при которой сообщение меняют одновременно 2 процесса. k меньше 2, так как взаимодействие трех или более процессов в языке Promela нереализуемо.

Если предположить, что каждый процесс участвует примерно в равной доле переходов, то для произвольного наперед заданного процесса вероятность участия в данном переходе составит k/P. Если **StateNode** отображает множество локальных состояний процесса на множество узлов равномерным образом, то, по аналогии с предыдущими рассуждениями, вероятность удаленного вызова при изменении состояния процесса составит $1-1/N$. Количество удаленных вызовов во всей модели, таким образом, равняется $T(1-1/N)k/P$ и с ростом N стремится к Tk/P. При количестве процессов P = 6-8 и k = 1.5, т.е. когда половина переходов приходится на взаимодействия процессов, получаем в P/k = 4-5 раз меньше удаленных вызовов, чем при хэшировании состояния целиком.

Эксперименты

Для экспериментов был создан прототип системы, последовательным образом имитирующий параллельную генерацию состояний с распределенных хранением. Данный

прототип состоит из транслятора, разбирающего модель на языке Promela, и генератора состояний на языке C. Схема его работы представлена на рис. 1.

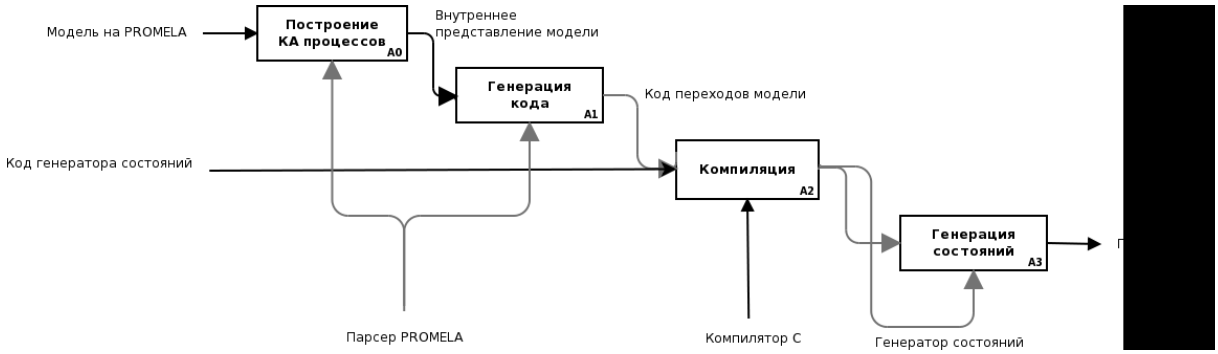

Рис.1. Схема работы генератора состояний

В качестве модели использовался алгоритм выбора лидера из набора примеров к пакету SPIN. Используемый объем памяти растет экспоненциально с количеством участвующих в алгоритме процессов. В качестве функции NodeState использовалась как хэш-функция от всего состояния, так и хэш-функция от состояния первого процесса.

Результаты экспериментов для имитации ЛВС из 4 узлов при P = 5, 6 и 7 представлены на табл. 1. В столбцах «Состояния» и «Переходы» указано общее число состояний и переходов модели. В столбце «Удаленные вызовы» — число (суммарное по всем «узлам») удаленных вызовов при использовании обоих вариантов **StateNode** (слева от «/» — от всего состояния, справа — от состояния первого процесса), в столбце «Загруженность узлов» в качестве меры «равномерности» распределения состояний между узлами показано отношение минимального и максимального объема используемой памяти среди всех узлов.

P	Состояния	Переходы	Удаленные вызовы	Загруженность узлов
5	41692	169690	154418 / 44782	0.92 / 0.65
6	341316	1667887	1492092 / 453512	0.96 / 0.71
7	2801653	15976645	14219214 / 4976725	0.99 / 0.74

Таблица 1. Результаты экспериментов

Из результатов можно сделать следующие выводы:

1. При использовании в качестве **StateNode** хэша всего состояния число удаленных вызовов действительно приближается к числу всех переходов.
2. Использование вместо него хэша от состояния первого процесса приводит к значительному сокращению числа удаленных вызовов.
3. Равномерность распределения состояний между узлами падает в последнем случае, что приведет к простою части памяти (до 30%) на некоторых узлах, поэтому функция распределения нуждается в дальнейшем улучшении.

Выводы

В данной работе описана одна из возможных оптимизаций процесса верификации конечных моделей — параллельное построение пространства состояний с использованием памяти и вычислительной мощности нескольких машин. Она может быть использована для разработки параллельной версии верификатора SPIN, возможности которой по проверке больших моделей были бы значительно больше, чем у оригинала.

Одним из наиболее важных аспектов, влияющих на производительность получаемого верификатора, является распределение состояний между узлами. В работе показано, что равномерное распределение, обеспечиваемое обычной хэш-функцией от всего состояния, является неэффективным из-за большого числа удаленных вызовов. Также показано, что распределение может быть значительно улучшено, если использовать в качестве номера узла хэш-код части состояния, соответствующей одному из его процессов.

Данная функция распределения не является лучшей и задача поиска более оптимальных альтернатив остается. В частности, можно предложить «интеллектуальный» подбор функции для конкретной модели и адаптацию используемого распределения во время работы алгоритма.

Список литературы

1. Gerard J. Holzman. An Analysis of Bit-state Hashing. In Proc. 15th Int. Conf on. Protocol Specification, Testing, and Verification, pp. 301–314.
2. Jiri Barnat, Lubos Brim. Parallel Breadth-First Search LTL Model-Checking. 18th IEEE International Conference on Automated Software Engineering (ASE'03), pp. 106-115.
3. Jiri Barnat, Lubos Brim. Distributed LTL Model-Checking in SPIN. Lecture Notes in Computer Science, vol. 2057/2001, pp 200-216.
4. Flavio Lerda, Riccardo Sisto. Distributed-Memory Model Checking with SPIN. Lecture Notes In Computer Science, vol. 1680, pp 22-39.