электронное научно-техническое издание

НАУКА и ОБРАЗОВАНИЕ

Эл № ФС77 - 30569. Государственная регистрация №0421100025. ISSN 1994-0408

# Выбор функции распределения состояний при параллельной проверке модели

77-30569/??????

# 0?, декабрь 2011

И. А. Коротков, В. А . Крищенко

УДК 004.021

МГТУ им. Н.Э. Баумана

kva@bmstu.ru

## Введение

Проверка модели — автоматический формальный подход, при котором на основе конечной дискретной модели строится полное пространство состояний и на нем проверяется набор интересующих нас утверждений [1]. Основной проблемой данного метода является быстрый рост числа состояний при росте сложности модели.

В настоящий момент на практике применяется ряд оптимизаций, позволяющих сократить как число состояний, так и требуемый для их хранения объем ОЗУ: сокращение частных порядков [1], битовое хэширование без обнаружения коллизий [2], сжатие состояний. Однако, данные меры либо дают небольшой и плохо масштабируемый прирост, либо приводят к потенциальным потерям состояний при обходе. Альтернативным подходом является параллельная генерация состояний с их распределенным хранением по узлам вычислительного кластера. При этом от выбора функции распределения состояний между узлами зависит число обменов информации между узлами и общее время проверки модели. В данной работе рассматривается вопрос выбора такой функции при разработке средства проверки моделей, использующего для своей работы MPI-кластер.

Статья организована следующим образом. В разд. 1 дано формальное определение модели и проверяемой спецификации. В разд. 2 приведено описание языка описания моделей в создаваемой системе. В разд. 3 описан общий алгоритм параллельной генерации состояний, а в разд. 4 приведена предлагаемая функции разбиения пространства состояний. Наконец, в разд. 5 описан используемая для экспериментов система, а в разд. 6 представлены результаты экспериментов.

## 1. Проверка конечных моделей

Пространство состояний детерминированной системы, например, моделируемой программы, можно формализовать в виде модели Крипке [1]. Моделью Крипке $M$ над множеством атомарных высказываний $AP$ называют четверку $(S, S_0, R, L)$, где:

- $S$ — конечное множество состояний;
- $S_0 \in S$ — множество начальных состояний;
- $R \in S \times S$ — отношение переходов, которые обязано быть тотальным, т.е. для каждого состояния $s \in S$ должно существовать такое состояние $s' \in S$, что имеет место $R(s, s')$;
- $L: S \to 2^{AP}$ — функция, которая помечает каждое состояние множеством атомарных высказываний, истинных в этом состоянии.

Путь в модели $M$ из состояния $s$ — это бесконечная последовательность состояний $\pi = s_0 s_1 \ldots$, такая, что $s_o = s$ и для всех $i \geq 0$ выполняется $R(s_i, s_{i+1})$.

Моделируемая программа в каждом своем состоянии описывается набором значений переменных $V = \{v_0, v_1, \ldots\}$, принимающих значения на конечном множестве $D$ (домене интерпретации) и описывающих отдельные компоненты модели и взаимодействие между ними. Множество $AP$ состоит из утверждений вида $v_i = d_i$, где $d_i \in D$. Таким образом, каждое состояние $s$ в $M$ представляет собой отображение $V \to D$.

Отношение $R$ определяется следующим образом. Пусть имеются два состояния, $s_1$ и $s_2$. Если в $s_1$ имеется компонент, который может выполнить атомарный переход (изменение значений своих переменных), в результате выполнения которого система будет находиться в состоянии $s_2$, тогда состояния $s_1$ и $s_2$ связаны отношением перехода: $(s_1, s_2) \in R$. В случае, если нет такого состояния $s_2$, для которого бы выполнялось $R(s_1, s_2)$, полагается $R(s_1, s_1)$, т.е. «тупиковое» состояние связано отношение перехода само с собой.

Для формализации проверяемых на модели $M$ утверждений обычно используются временные логики [1]. Однако, для проверки моделей программных систем наиболее актуальны два вида высказываний:

- утверждение о постоянном выполнении некоего условия в заданной точке компонента, что является аналогом конструкции **assert** при программировании;
- утверждение о завершении работы модели или её функционировании в вечном цикле, что важно при поиске тупиков в программной системе.

## 2. Язык описания модели

Наиболее распространенным средством проверки конечных моделей является система SPIN, использующее для описания исходной модели язык Promela [3]. Разрабатываемая в рамках данной работы система так же использует подмножество языка Promela для описания моделей.

Модель на языке Promela описывается в виде набора процессов (компонент модели), состоящих из последовательности команд. Каждый процесс имеет свой набор локальных переменных, в том числе счетчик команд. Для взаимодействия между процессами могут использоваться глобальные переменные и очереди сообщений. Каждая команда имеет свое условие выполнимости, и процесс считается заблокированным, если условие выполнимости его текущей команды не выполнено.

Пример описания модели семафора Дейкстры и трех захватывающих его процессов в нотации языка Promela приведен ниже. Для данной модели представляет интерес утверждение о «вечном» выполнении модели, что будет свидетельствовать об отсутствии тупиков.

```
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 concurent_user()
{        do
         :: enter: sema?p;  /* вход в критическую секцию */
             crit: skip;    /* критическая секция */
            leave: sema!v;  /* выход из критической секции */
         od
}
```

## 3. Параллельная генерация состояний

При росте числа и сложности компонент моделируемого программной системы в случае наблюдается комбинаторный рост числа возможных состояний. Поскольку граф состояний в общем случае имеет циклы, необходимо хранить множество посещенных состояний, которое быстро перестаёт помещаться в ОЗУ одной машины, а использование внешней памяти приведет к увеличению времени проверки на 3–4 порядка. В силу этого для проверки больших моделей лучше использовать вычислительный кластер.

Возможны два подхода к использованию ресурсов вычислительного кластера при генерации состояний. При распределенном хранилище состояний состояния генерирует только один узел, а для хранения используются все узлы. Каждое состояние имеет свой однозначно вычислимый номер узла и для проверки, принадлежит ли следующее состояние множеству посещенных, делается синхронный удаленный вызов хранящего это состояние узла.

При параллельной генерации состояний каждый узел кластера является одновременно хранилищем и генератором. Если новое состояние принадлежит другому узлу (переход — внешний), то оно высылается ему асинхронным удаленным вызовом. На рис. 1 показан пример обхода графа при данном подходе. Цифры рядом с состояниями обозначают локальный порядок генерации (в пределах данного узла).

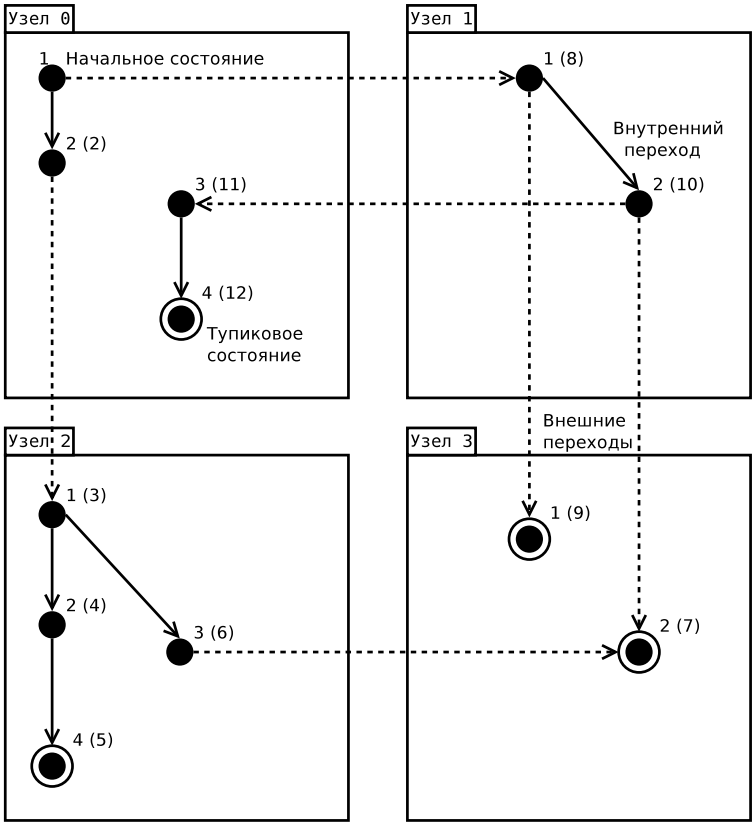


**Рис. 1**. Пример работы параллельной генерации состояний

Несмотря на очевидные преимущества, второй подход имеет свой недостаток: отсутствие какого-либо глобального порядка обхода состояний. В работе была выбрана параллельная генерации состояний. Каждый узел, кроме генерации состояний, также проверяет утверждения.

## 4. Функция распределения состояний между узлами

Одной из основных проблем параллельной генерации состояний является выбор функции распределения состояний между узлами. Эта функция ставит в соответствие

каждому состоянию индекс узла, отвечающего за хранение данного состояний. В случае параллельной генерации состояний она должна иметь следующие свойства:

- зависеть только от битового представления самого состояния, поскольку одно и то же состояние может генерироваться различными узлами в результате различных переходов;
- распределять состояния между узлами достаточно равномерно, в противном случае часть памяти у некоторых узлов будет простаивать;
- обладать свойством локальности относительно переходов между состояниями — по возможности новые состояния должны принадлежать тому же узлу, что и исходное.

Наиболее простым подходом является использование хэш-функции от битового представление состояния $s$ в качестве индекса хранящего его узла. Это обеспечит первые два условия: если выбрана подходящая хэш-функция, распределение будет достаточно равномерным. Однако, третье условие при этом не соблюдается: 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

$$Q = T(1 - \frac{1}{N}), \tag{1}$$

что при больших значениях стремится к $T$.

Высокое число удаленных вызовов негативно отражаются на производительности, поэтому необходимо найти более удачную функцию распределения состояний, которая бы удовлетворяла условию локальности. Одна из возможных идей заключается в использовании хэш-код не от всего состояния $s$, а от некоторой его части $\tilde{s}$.

Битовое представление состояния в общем случае представляет собой набор значений переменных, описывающих состояние отдельных компонент моделируемой системы и значения общих переменных, описывающих взаимодействие между ними.

Пусть $P$ — число таких компонент (процессов в нотации Promela) в модели, $k$ — среднее число компонент, состояние которых меняется при переходе. Для языка Promela $1 \approx k < 2$ , поскольку взаимодействие между более чем двумя процессами (компонентами моделируемой системы) нереализуемо, но для двух процессов есть возможность синхронной передачи сообщения, при которой оба меняют свое состояние. Последняя возможность используется обычно редко, поэтому для большинства моделей $k$ достаточно близко к 1.

Таким образом, битовое представления состояния естественным образом разделяется на $(P+1)$ область, с учетом области глобальных переменных. При этом $P$ из них меняются почти независимо друг от друга при условии $k \approx 1$, и в качестве хэшируемого подсостояния $\tilde{s}$ можно выбрать первые (или произвольные) $\rho$ областей, хранящих локальные состояния первых $\rho$ процессов.

Если предположить, что каждый процесс $p_i$ участвует примерно в равной доле переходов, то для произвольного наперед заданного процесса вероятность участия в данном переходе составит $k/\rho$, а для $\rho$ процессов при $k \approx 1$ либо небольшом $\rho$ — $\frac{k\rho}{P}$. При условии, что множество возможных локальных состояний процесса отображается на множество узлов равномерным образом, вероятность удаленного вызова при изменении локального состояния процесса (т.е. при его участии в переходе), по аналогии с предыдущими рассуждениями, составит $1 - 1/N$. Количество удаленных вызовов во всей модели, таким образом, равняется

$$Q_\rho = \frac{k\rho}{P} T (1 - \frac{1}{N}) \tag{2}$$

и с ростом $N$ стремится к $\frac{k\rho}{P}T$. При количестве процессов $P = 10$, $k = 1.1$ и $\rho = 2$, число удаленных вызовов уменьшается примерно в 4 раза в сравнении с «наивным» подходом.

Выбор меньших значений $\rho$ приводит к меньшему числу удаленных вызовов, однако увеличивает неравномерность распределения, поэтому его значение следует выбирать из баланса между требуемой равномерностью распределения состояний и выигрышем во времени за счет уменьшения числа вызовов.

Пусть $i$-ый процесс $p_i$ имеет $w_i$ возможных значений локального состояний, т.е. число допустимых комбинаций значений его переменных составляет $w_i$. Объединение локальных состояний $\rho$ процессов тогда имеет не более $W_\rho = \prod_{i=1}^{\rho} w_i$ возможных значений. Число значений может быть меньше $W_\rho$, поскольку в общем случае не все комбинации являются допустимыми. Значение $\rho$ должно обеспечивать условие $W_\rho \gg N$, иначе, особенно при $W_\rho \approx N$, распределение будет неравномерным даже при удачном выборе хэш-функции, а при $W_\rho < N$ память некоторых узлов не будет использоваться вообще, так как число возможных значений хэш-функции будем меньше числа узлов.

## 5. Реализация экспериментальной системы

Было создано программное средство для параллельной проверки состояний с распределенной генерацией, поддерживающее основные возможности языка Promela для описания модели и проверку утверждений с функцией **assert** и поиском тупиковых состояний.

Разработанное ПО работает следующим образом (рис. 2). Исходная модель на Promela считывается и транслируется во внутренний граф команд для каждого процесса, который затем минимизируется. С целью достижения скорости генерации состояний, сравнимой со скоростью системы SPIN, по полученным графам команд генерируется код на

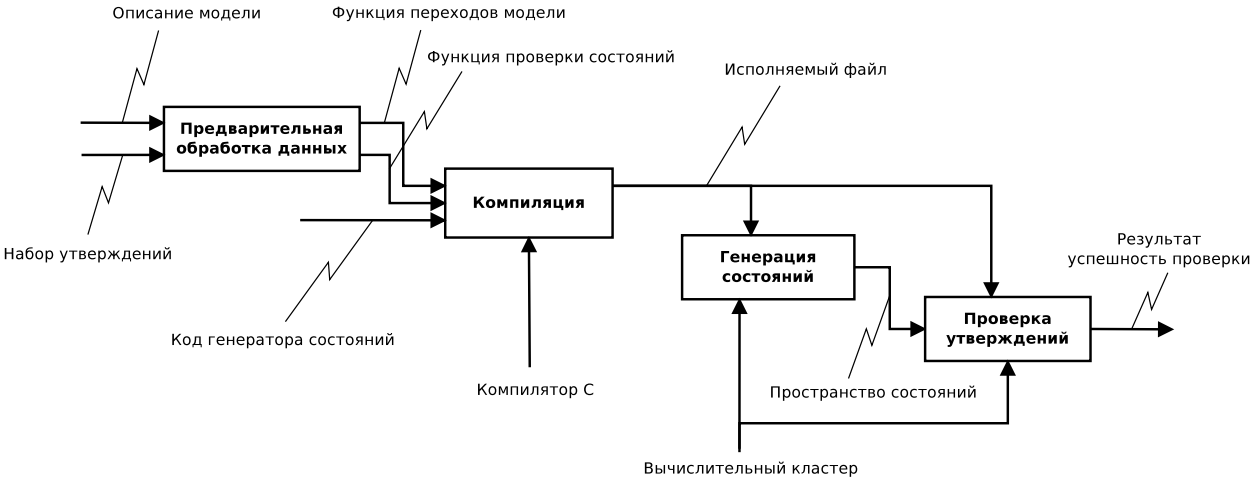


**Рис. 2**. Функциональная схема процесса генерации кода по описанию модели

языке C, выполняющий вычисление функции переходов модели $Next(s) = \{s' : R(s, s')\}$ и функции проверки состояний $Assert(s) : S \to \{0, 1\}$. В качестве платформы для параллельных вычислений в системе используется вычислительный кластер, поддерживающий стандарт интерфейса параллельного программирования MPI.

## 6. Результаты экспериментов

В качестве исходных данных для экспериментов были взяты две модели: выбора лидера и «обедающие философы» с числом компонент $P = 6$. Для проведения экспериментов использовался MPI-кластер из 20 узлов, имеющих 4 Гб ОЗУ и 4 ЦПУ Intel Xeon 5120 1.86 ГГц каждый.

Результаты экспериментов по сравнению предлагаемого распределения с $\rho = 1$ и $\rho = 2$ с наивным представлены на табл. 1 и 2.

| $\rho$ | Доля внешних переходов, % | Неравномерность распределения, % | Время простоя, сек | Общее время работы, сек |
|---|---|---|---|---|
| 1 | 16 | **66.3** | 29 | 43 |
| 2 | 36 | 12.8 | 65 | 84 |
| — | **87** | 0.1 | **127** | **164** |

Таблица 1. Сравнение распределений (алгоритм выбора лидера)

| $\rho$ | Доля внешних переходов, % | Неравномерность распределения, % | Время простоя, сек | Общее время работы, сек |
|---|---|---|---|---|
| 1 | 17 | **89.4** | 3 | 14 |
| 2 | 35 | 29.6 | 7 | 21 |
| — | **88** | 0.1 | **50** | **75** |

Таблица 2. Сравнение распределений («обедающие философы»)

Проблемные значения выделены жирным начертанием. В таблицах приведены следующие величины:

- доля внешних переходов — отношение числа удаленных вызовов (суммарно на всех узлах) к числу переходов $T$;
- неравномерность распределения — отношение среднеквадратичного отклонения к среднему для последовательности $m_1 m_2 \dots m_N$, где $m_i$ — число состояний, хранимых узлом $i$;
- время простоя при ожидании сообщений от других узлов (сетевые задержки);
- общее время работы.

Из приведённых результатов можно сделать следующие выводы.

1. Выбор распределения между узлами важен, поскольку время простоя за счет удаленных вызовов составляет существенного часть от времени выполнения.
2. При «наивном» подходе к распределению состояний число удаленных вызовов близко к числу всех переходов, как и следует из (1).
3. Предлагаемый способ распределения состояний по первым $\rho$ процессам позволяет уменьшить число удаленных вызовов и время выполнения по сравнению с «наивным» подходом в соответствии с (2).
4. Необходим подбор параметра $\rho$ в соответствии со свойствами проверяемой модели $(P, w_i)$ для обеспечения требуемого уровня равномерности распределения состояний. В частности, значения $\rho = 1$ в приведенных экспериментах оказалось недостаточно, поскольку неравномерность до 90% означает, что большая часть памяти некоторых узлов не используется.

## Заключение

Предложена функция разбиения пространства состояний, позволяющая уменьшить число удаленных вызовов между узлами при сохранении равномерности распределения. Эксперименты, проведенные при помощи прототипа средства параллельной проверки моделей, показывают, что предложенное разбиение в несколько раз сокращает время проверки в сравнении с тривиальным. Полученный алгоритм можно использовать для параллельной проверки моделей большего размера, чем позволяют традиционные последовательные подходы.

## Список литературы


1. *Э. М. Кларк, О. Грамберг, Д. Пелед.* Верификация моделей программ. Москва: МЦНМО, 2002. 416 с.
2. *Holzmann, G. J.* An Analysis of Bitstate Hashing // Formal Methods in System Design. 1998. Vol. 13, N. 3. Pp. 287–305.
3. *Holzmann, G. J.* The model checker SPIN // IEEE Transactions on Software Engineering. 1997. Vol. 23. Pp. 279–295.