РАЗРАБОТКА ПАРАЛЛЕЛЬНОГО АЛГОРИТМА ГЕНЕРАЦИИ СОСТОЯНИЙ ПРИ ПРОВЕРКЕ МОДЕЛЕЙ ДЛЯ СИСТЕМ С НЕРАЗДЕЛЯЕМОЙ ПАМЯТЬЮ

И. А. Коротков
Московский государственный технический университет им. Н. Э. Баумана
Россия, 105005, Москва, 2-ая Бауманская, 5
E-mail: korotkov2@mail.ru

Ключевые слова: формальная верификация, проверка моделей, генерация состояний, параллельные вычисления, Promela
Key words: model checking, state generation, parallel statespace, parallelism, Promela

Основной проблемой верификации конечных моделей является комбинаторный «взрыв» числа состояний, которые с ростом размера модели становится затруднительно хранить в ОЗУ одной машины. Рассматривается подход к верификации конечных моделей на основе параллельной генерации состояний и их распределенного хранения. Предлагается схема распределенного хранения состояний, позволяющая уменьшить число удаленных вызовов между узлами в процессе генерации. Приводятся результаты экспериментов, полученные при помощи разработанного программного средства.

Parallel statespace generation during model-checking for systems with distributed memory / I. A. Korotkov (Moscow State Technical Univercity, 5 Baumanskay 2-aya ul., Moscow 105005, Russia, e-mail: korotkov2@mail.ru).
A major limitation of model checking is statespace combinatorial explosion, which makes even medium-sized model inappropriate for that kind of verification. In this paper, parallel statespace generation with distributed state storage is proposed as a possible solution. State partitioning scheme that allows to reduce number of remote calls during generation process is developed. Experimental results, produced by developed verification tool, are given and prove that proposed partitioning scheme is better than random uniform distribution.

1 Введение

При разработке сложных параллельных систем (систем, состоящих из нескольких асинхронно работающих компонент) с высокой степенью надежности традиционных подходов к тестированию зачастую бывает недостаточно, поскольку они позволяет выявить лишь легко воспроизводимые ошибки. В некоторых случаях, например, в программном обеспечении для бортовых систем, определенные классы ошибок требуется полностью исключить.

Для таких случаев применяется проверка модели (model checking) — автоматический формальный подход, при котором на основе дискретной детерминированной модели программы или комплекса программ строится полное пространство состояний и на нем проверяется набор интересующих нас утверждений — спецификация [1]. Проверку моделей можно использовать для поиска взаимоблокировок в параллельных алгоритмах и ошибок в спецификациях сетевых протоколов. В качестве примера можно привести протокол маршрутизации RIP: проверка модели сети из четырех маршрутизаторов, соединенных

четыремя сетевыми интерфейсами, на предмет возникновения циклов в маршрутных таблицах позволяет убедиться, что существуют сценарии, при которых такие циклы возникают, и необходимо использование специальных мер (расщепленный горизонт) для их избежания [2].

2 Формальное описание проверки конечных моделей

Пространство состояний моделируемой программы или программного комплекса можно формализовать, как модель Крипке (структуру Крипке, Kripke structure) [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 $R(s, s')$;
- $L\colon S \to 2^{AP}$ — 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 $R(s_i, s_{i+1})$.

Моделируемый программный комплекс в каждом своем состоянии описываются набором значений переменных $V = \{v_0, v_1, \ldots\}$, принимающих значения на конечном множестве D (домене интерпретации) и описывающих отдельные компоненты 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 — LTL (linear time logic, логика линейного времени), CTL (computation tree logic, логика ветвящегося времени), CTL* (объединение LTL и CTL) [1, 3]. Формулы в CTL* составляются из атомарных утверждений относительно значений переменных v_i и кванторов: **A** (all) — «для всех путей, выходящих из данного состояния», **E** (exists) — «существует такой путь, выходящий из данного состояния», **F** (finally) — «рано или поздно в пути встретится состояние, в котором выполняется ...», **G** (globally) — «во всех состояниях пути выполняется ...», **X** (next) — «в следующем состояния на данном пути выполняется ...», **U** (until) — «пока в пути не появится состояние, в котором выполняется y, во всех состояниях должно выполняться x».

Например, формула **AFG** x означает «во всех путях, идущих из начального состояния, с некоторого состояния на протяжении всего пути выполняется x», а формула **AGEF** x означает «во всех путях, идущих из начального состояния, из каждого состояния есть хотя бы один путь, в котором рано или поздно встретится состояние, в котором выполняется x».

3 Средство проверки конечных модели SPIN

Наиболее распространенным средством проверки конечных моделей является ПО SPIN, использующее для описания исходной модели язык *Promela* (PROtocol Meta Language) [4].

Модель на языке Promela описывается в виде набора процессов, 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 каналы (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 [5] и трех захватывающих его процессов в нотации Promela привен ниже:

```
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
}
```

В ходе верификации SPIN выполняет исчерпывающий поиск в глубину по графу состояний (модели Крипке) и, при обнаружении пути, на котором нарушается проверяемое утверждение, сохраняет его в качестве контрпримера [1]. Если контрпример обнаружить не удается, верификация успешна. Функциональная схема процесса проверки модели изображена на рис. 1.

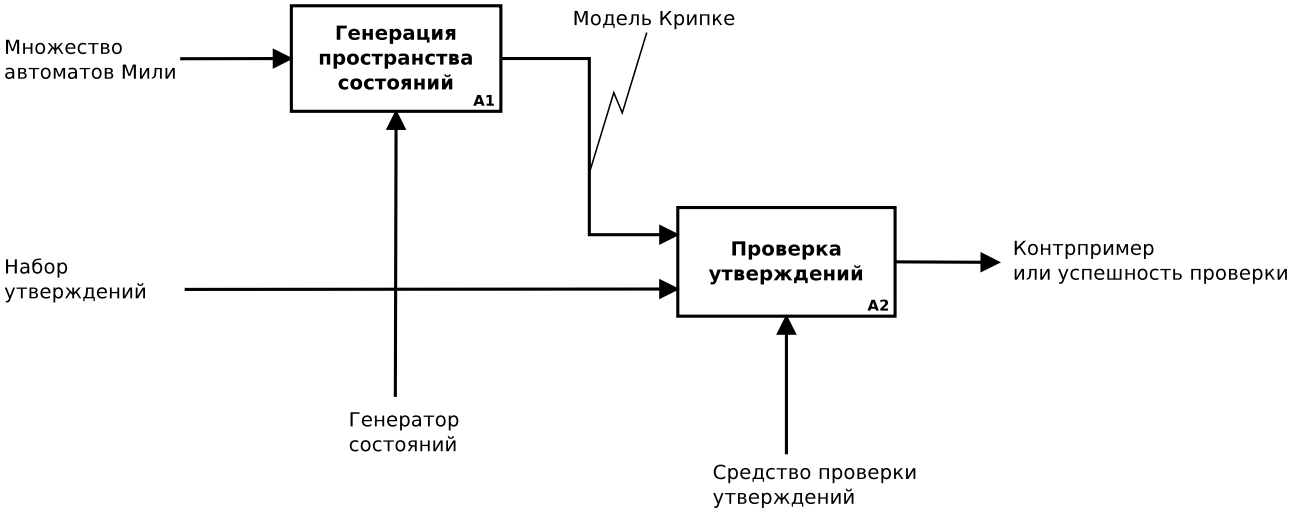

Рис. 1: Функциональная схема проверки модели

4 Параллельная генерация состояний

При росте числа и сложности компонент моделируемого программного комплекса наблюдается комбинаторный рост числа возможных состояний, поэтому проверка модели требует больших затрат вычислительных ресурсов. Приведенная в предыдущем примере модель сети из четырех RIP-маршрутизаторов имеет более 10^9 состояний. Поскольку граф состояний в общем случае имеет циклы, необходимо хранить множество посещенных состояний, которое при таком числе состояний не помещается в ОЗУ одной машины, тогда как использование внешней памяти приведет к увеличению времени проверки на 3–4 порядка.

Применяется ряд оптимизаций, позволяющих сократить как число состояний, так и требуемый для их хранения объем ОЗУ: *сокращение частных порядков* (уменьшает размер графа состояний [6]), *битовое хэширование* (уменьшает объем требуемоей памяти за счет того, что сами состояния не хранятся и коллизии в хэш-таблице не отслеживаются [7, 8]), *сжатие состояний* (уменьшает объем требуемой памяти, однако незначительно и ценой существенного увеличения времени проверки [9]).

Однако, приведенные меры либо дают небольшой, плохо масштабируемый прирост, либо приводят к потенциальным потерям состояний при обходе. Альтернативным подходом является *параллельная генерация состояний* с распределенным хранением по различным узлам вычислительной сети.

Возможны два подхода к 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 и для проверки, принадлежит ли следующее состояние множеству посещенных, делается синхронный удаленный вызов хранящего это состояние узла.

2. Распределенная генерация состояний. Каждый узел является одновременно хранилищем и генератором. Если новое состояние принадлежит другому узлу, оно высылается ему асинхронным удаленным вызовом. На рис. 2 показан пример обхода графа при данном подходе. Цифры рядом с состояниями обозначают локальный порядок генерации (в пределах данного узла).

Несмотря на очевидные преимущества (использование вычислительной мощности всех узлов, асинхронные вызовы вместо синхронных), второй подход имеет свой недостаток: отсутствие какого-либо глобального порядка обхода состояний. Проверка определенных классов утверждений (например, LTL-формул) требует поиска циклов в графе состояний, то есть обхода в глубину. В данной статье рассматривается лишь генерация состояний; вопрос нахождения таких циклов при распределенной генерации выходит за ее рамки и подробно рассмотрен в [10, 11].

Одной из основных проблем распределенной генерации состояний является выбор функции распределения состояний между узлами.

5 Распределение состояний между узлами

Функция распределения ставит в соответствие каждому состоянию индекс узла, отвечающего за хранение данного состояний. Эта функция должна обладать следующими свойствами:

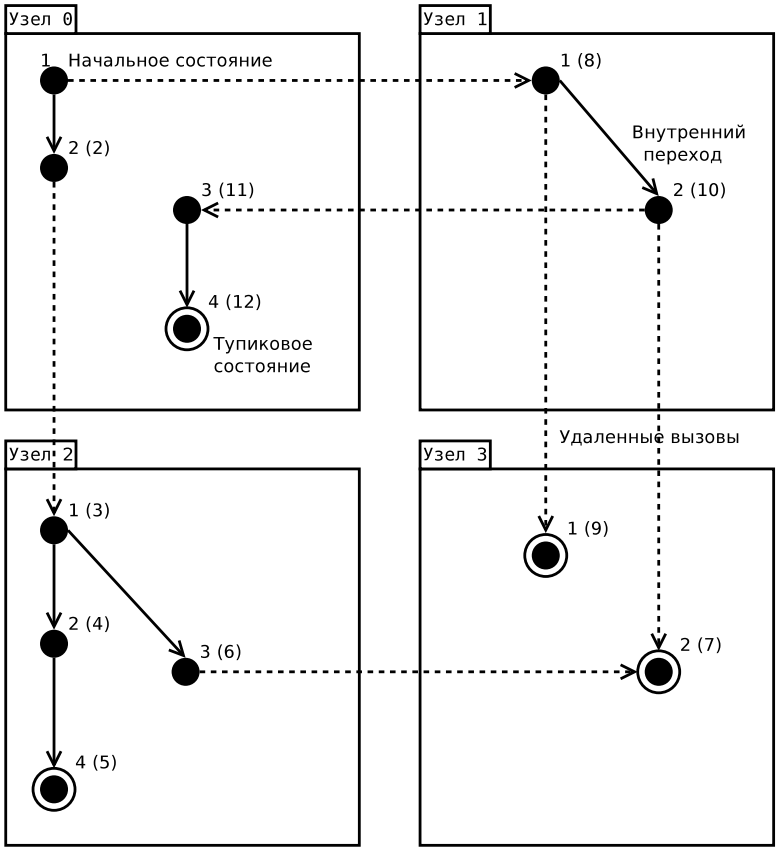

Рис. 2: Пример работы распределенной генерации состояний

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

Последнее условие имеет смысл лишь при втором подходе, «распределенной генерации состояний», и позволяет уменьшить число асинхронных удаленных вызовов между узлами.

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, 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

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

что при больших значениях стремится к T.

Столь 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 предложена в [12]: использовать 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[1] нереализуемо, но для двух процессов есть возможность синхронной передачи сообщения, при которой оба меняют свое состояние. Последняя возможность используется нечасто, 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) ρ областей, хранящих локальные состояния первых ρ 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/ρ, а для ρ процессов при $k \approx 1$ либо небольшом ρ — $\frac{k\rho}{P}$. 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 ρ приводит к меньшему числу удаленных вызовов, однако увеличивает неравномерность распределения, поэтому его значение следует выбирать из баланса между требуемоей 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 ρ процессов тогда имеет не более $W_\rho = \prod_{i=1}^{\rho} w_i$ возможных значений. Число значений может быть меньше W_ρ, поскольку в общем случае не все комбинации являются допустимыми. Значение ρ 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.

[1]Далее под словом «процесс» будет подразумеваться процесс в понимании Promela, т.е. компонент моделируемой системы.

6 Результаты

Создано программное средство для параллельной проверки состояний с распределенной генерацией, поддерживающий подмножество языка Promela для описания модели. Для задания проверямых утверждений поддерживается подмножество LTL, допускающее формулы **AG** x и **AF** x, где x может содержать локальные переменные процессов (включая счетчик команд) и глобальные переменные. На практике данное подмножество LTL реализовано при помощи встроенной в Promela функции `assert` и поиска тупиковых состояний.

В разработанном средстве поддерживаются основные возможности языка Promela, такие как асинхронные каналы, пользовательские типа данных, `atomic` и `d_step` [4].

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 SPIN, по полученным графам команд генерируется код на языке C, выполняющий вычисление функции переходов модели $Next(s) = \{s' : R(s, s')\}$ и функции проверки состояний $Assert(s) : S \to \{0, 1\}$. Функциональная схема данного процесса приведена на рис. 3.

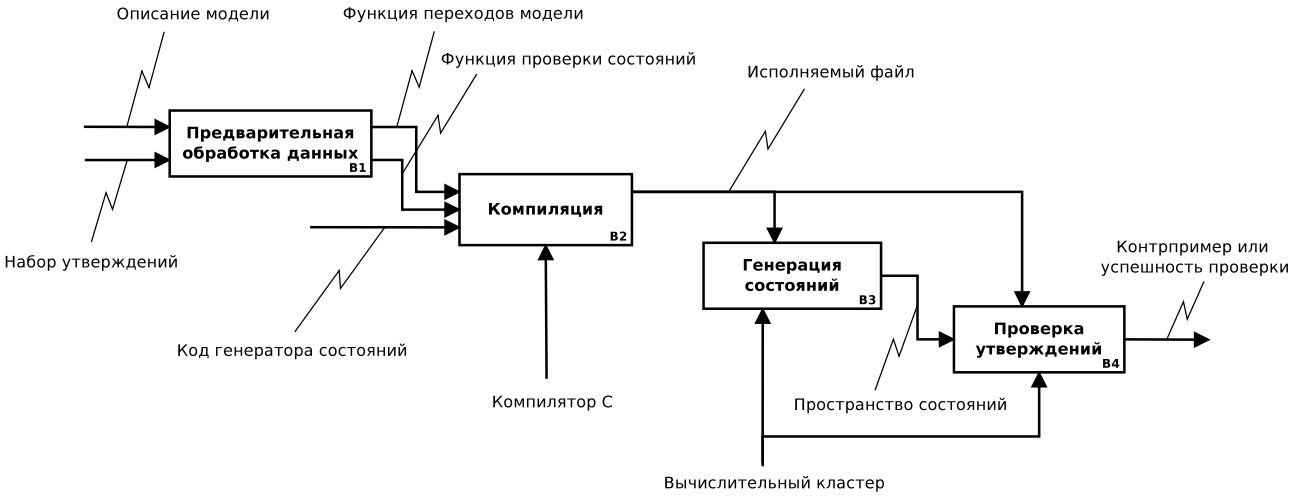

Рис. 3: Функциональная схема процесса генерации кода по описанию модели

В качестве платформы для параллельных вычислений использовался стандарт MPI, выбранный в силу его распространенности среди кластерного ПО и поддержки широкого набора языков.

Исходными данными служат две модели: алгоритм выбора лидера и «обедающие философы» с числом компонент $P = 6$. Для проведения экспериментов использовался 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. Приведены следующие величины:

- доля вызовов среди 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.

ρ	Доля вызовов среди 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)

ρ	Доля вызовов среди 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.
Из 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 ρ процессам позволяет уменьшить число удаленных вызовов и время выполнения в сравнении с «наивным» подходом в соответствии с (2);

4. необходим подбор параметра ρ в соответствии со свойствами проверямой 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 вообще.

7 Заключение

Использование параллельной генерации состояний дискретных детерминированных моделей при проверке их соответствия спецификациям позволяет выполнять верификацию моделей на несколько порядков большего размера, чем позволяют аналогичные подходы с последовательной генерацией. Одним из наиболее важных факторов является функция распределения хранимых состояний между узлам, правильный выбор которой, как подтверждают эксперименты, позволяет в несколько раз ускорить верификацию.

Список литературы

1. *Кларк, Э. М.* Верификация моделей программ. Model Checking / Э. М. Кларк, О. Грамберг, Д. Пелед. — МЦНМО, 2002.

2. *Krishchenko, Vsevolod.* Model Checking of RIP Count-to-Ininity Problem / Vsevolod Krishchenko. — Москва, 2008.

3. *Вельдер, С. Э.* Введение в верификацию автоматных программ / С. Э. Вельдер, А. А Шалыто // XXXVI научная и учебно-методическая конференция профессорско-преподавательского и научного состава / СПбГУ ИТМО. — 2007.

4. *Holzmann, Gerard J.* The SPIN Model Checker : Primer and Reference Manual / Gerard J. Holzmann. — Addison-Wesley Professional, 2003.

5. *Holzmann, Gerard J.* The model checker SPIN / Gerard J. Holzmann // *IEEE Transactions on Software Engineering*. — 1997. — Vol. 23. — Pp. 279–295.

6. *Holzmann, Gerard J.* The Verification of Concurrent Systems, unpublished manuscript / Gerard J. Holzmann. — AT&T Inc.

7. *Holzmann, G. J.* An Analysis of Bitstate Hashing / G. J. Holzmann // *Formal Methods in System Design*. — 1998. — Vol. 13, no. 3. — Pp. 287–305. — extended and revised version of Proc. PSTV95, pp. 301-314.

8. *Wolper, Pierre*. Reliable Hashing without Collision Detection. — 1993. — Januar.

9. *Holzmann, Gerard J.* State Compression in SPIN: Recursive Indexing and Compression Training Runs. — 1997. — Aprtil.

10. *Barnat, J.* Distributed LTL Model-Checking in SPIN / J. Barnat, L. Brim, J. Stříbrná // Proc. SPIN Workshop on Model Checking of Software. — Vol. 2057 of *LNCS*. — Springer, 2001. — Pp. 200–216.

11. *Barnat, J.* Parallel Breadth-First Search LTL Model-Checking / J. Barnat, L. Brim, J. Chaloupka // 18th IEEE International Conference on Automated Software Engineering (ASE'03). — IEEE Computer Society, 2003. — Oct. — Pp. 106–115.

12. *Lerda, Flavio*. Distributed-Memory Model Checking with SPIN / Flavio Lerda, Riccardo Sisto // Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Trento, Italy, July 5, 1999, Toulouse, France, September 21 and 24 1999, Proceedings / Ed. by Dennis Dams, Rob Gerth, Stefan Leue, Mieke Massink. — Vol. 1680 of *Lecture Notes in Computer Science*. — Springer, 1999. — Pp. 22–39.