Оглавление

Аннотация

В работе рассмотрены и изучены основные проблемы, возникающие при генерации конечных автоматов для проверки моделей с большим числом состояний, не умещающимся в памяти одного компьютера. Предложен способ решения это проблемы путем хранения состояний конечного автомата на различных узлах вычислительной системы с разделенной памятью.

В рамках решения поставленной задачи рассмотрены методы построения конечного автомата модели по графу формального описания модели, изучены основные методы хеширования, а также способы хранения таблицы переходов конечного автомата. Разработаны и реализованы два алгоритма распределенной генерации и хранения конечного автомата модели на узлах вычислительной системы с разделенной памятью, проведены исследования временных характеристик разработанной системы.

1 Введение

В настоящее время большое значение приобретают средства и методы для проверки правильности систем. Для проверки правильности проектируемой системы используются методы имитационного моделирования и тестирования, а также методы экспертной оценки, когда процесс разработки системы контролируется по большей части людьми и динамического тестирования (контроль на этапе разработки, испытание системы, эксплуатационные испытания и т.д.). Однако, в связи с ростом сложности проверяемых систем данные методы становятся неэффективными.

Альтернативой данным методам является *формальная верификация моделей* (или проверка моделей, model checking). Наиболее известным из средств формальной верификации является верификатор SPIN, разработанный Д. Хольцманом в одной из лабораторий NASA – *NASA JPL Laboratory for Reliable Software*. В существующих средствах верификации используется способ проверки конечных моделей, при котором по формальному описанию модели строится множество состояний модели, а также переходов между ними (т.е. строится конечный автомат), и далее для проверки выполнения (или невыполнении) некоторого условия происходит перебор в худшем случае всех состояний модели. При этом число состояний модели может быть очень велико, и, как следствие, не умещаться в оперативной памяти компьютера. Но оно обязательно должно быть конечно, в противном случае придется использовать другие средства для проверки модели. Таким образом, возникает *проблема хранения большого количества состояний модели*.

Для решения этой проблемы можно хранить состояния, например, на жестком диске. Однако, экспериментально установлено, что использование для хранения множества состояний модели памяти, более медленной чем ОЗУ, приводит к замедлению работы алгоритма на 2-3 порядка. Поэтому в качестве решения проблемы хранения большого множества состояний модели предлагается использовать кластерные системы с разделенной памятью. Это позволяет хранить достаточно большое количество состояний модели в оперативной памяти узлов кластера.

Целью данной работы является реализация алгоритмов распределенной генерации конечных автоматов для проверки моделей, называемых также автоматами Бюхи. Для достижения поставленной цели необходимо:

1. Изучить существующие подходы к созданию параллельных программ, оперирующих с большими дискретными структурами.
2. Разработать и реализовать представление конечных автоматов моделей систем.
3. Разработать алгоритм распределенной генерации и хранения конечного автомата модели системы.
4. Программно реализовать разработанные алгоритмы.
5. Исследовать временные характеристики разработанной вычислительной системы.

2 Аналитический раздел

Если для системы можно построить модель, имеющую конечное число состояний, то полное поведение модели описывается графом, в вершинах которого содержится полный список значений переменных, полностью описывающих конкретное состояние модели системы.

Поэтому необходимо:

1) изучить основные способы построения конечного автомата модели;
2) изучить основные способы хеширования с целью уменьшения времени поиска в пространстве состояний;
3) изучить способы хранения таблицы переходов конечного автомата.

2.1 Построение конечного автомата модели

В данной работе рассматриваются системы, модели которых имеют конечное число состояний и переходов между ними и, соответственно, описываются конечными автоматами. Однако, данный автомат является конечной формой описания поведения модели и не может задаваться человеком непосредственно, поскольку интерес представляют модели с очень большим числом состояний.

В качестве описания подобной модели специалисты по моделированию обычно используют средство, позволяющее компактно описать модели системы с большим числом состояний в виде, удобном для некоторой предметной области. Например, рассмотрим следующее формальное описание модели на языке Promela.

```
#define N    4
int x = N;

active proctype A()
{
        do
        :: x%2 -> x = 3*x+1
        od
}
```

```
active proctype B()
{
        do
        :: !(x%2) -> x = x/2
        od
}
```

Это описание изоморфно конечному автомату с выходом.

Автоматы A_1 и A_2 соответствующие процессам A и B, описывающим модель системы, выглядят следующим образом (см. рис. 2-1):

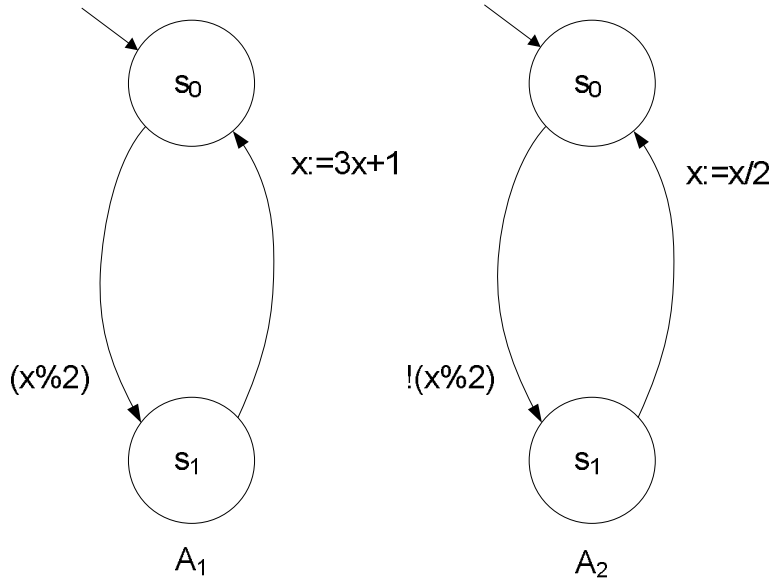


Рисунок 2-1. Конечные автоматы, соответствующий процессам *A* и *B* модели

Для построения автомата модели необходимо построить автомат, являющийся произведением (асинхронным) автоматов A_1 и A_2. Этот автомат изображен на рис. 2-2.

Состояния этого автомата помечены парами меток p и q, где p – метка соответствующего состояния в автомате A_1, а q – метка соответствующего состояния в автомате A_2. Состояние s_0, s_0 является начальным состоянием. Поскольку ни автомат A_1 ни автомат A_2 не имеют допускающих состояний, то и произведение этих автоматов также не имеет допускающих состояний.

При внимательном рассмотрении автомата, и интерпретации меток переходов, можно заметить, что все пути, ведущие в состояние s_1, s_1, недостижимы, поскольку они требуют выполнения как условия $(x\%2)$, так и его отрицания $!\,(x\%2)$ без промежуточного изменения значения переменой x. Поэтому состояние s_1, s_1 фактически недостижимо.

Приведенный на рис. 2-2 автомат, однако, не является конечным автоматом, соответствующим приведенной выше системе. Необходимо построить *расширенный*

конечный автомат (автомат Бюхи), полностью раскрывая все значения, которые может принимать целочисленная переменная x.

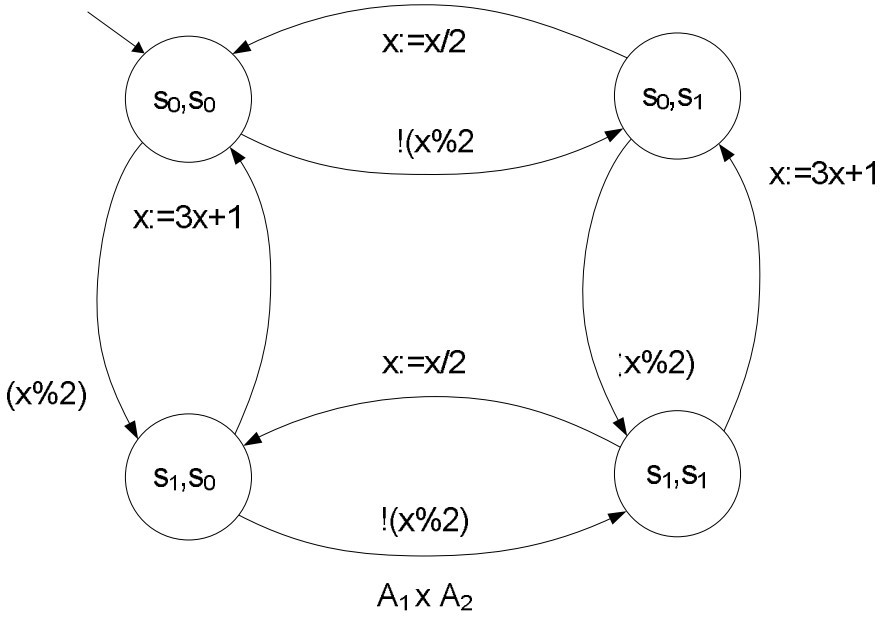


Рисунок 2-2. Асинхронное произведение базовых конечных автоматов

Расширенный конечный автомат и является конечным автоматом, соответствующим заданной выше модели. Этот конечный автомат изображен на рис. 2-3:

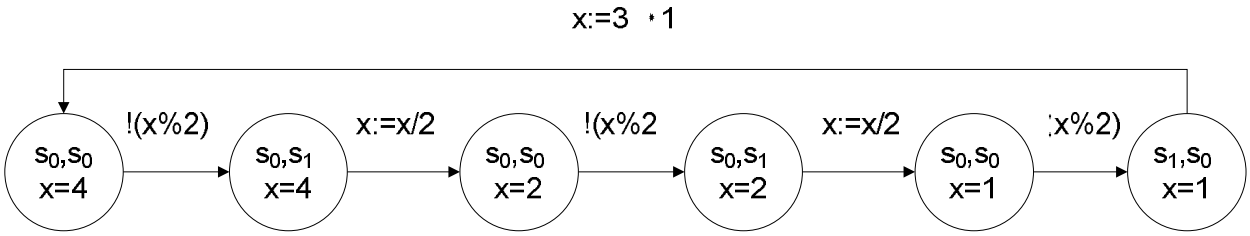


Рисунок 2-3. Расширенный конечный автомат модели

Состояния расширенного конечного автомата помечены тройками p, q и v, где p и q – метки соответствующих состояний в исходном автомате, а v – целочисленное значение переменной x в каждом состоянии. Исходный конечный автомат, построенный по формальному описанию модели (см. рис. 2-2), по которому строится расширенный конечный автомат, будем называть *базовым конечным автоматом.* В дальнейшем часто будем обозначать базовый конечный автомат аббревиатурой БКА, а расширенный – аббревиатурой РКА.

Рассмотрим процесс построения РКА по заданному БКА более подробно. При формальном описании модели используется единственная переменная x с начальным значением, равным 4. Начальное состояние БКА обозначено как s_0, s_0. Таким образом, получаем начальное состояние РКА $s_0, s_0, x = 4$. Далее, по заданной метке s_0, s_0 без значения переменной начального состояния РКА, определяем, какие состояния достижимы из начального состояния БКА при заданном значении $x = 4$. В результате по переходу с меткой $!(x\%2)$ переходим в состояние s_0, s_1 БКА. При этом значение $x = 4$. Следовательно, получили следующее состояние РКА $s_0, s_1, x = 4$, достижимое из начального состояния РКА по переходу с меткой $!(x\%2)$. Далее, по заданной метке s_0, s_1, определяем, какие состояния достижимы из соответствующего состояния в БКА. В результате переходим по метке $x = x/2$ в состояние s_0, s_0 БКА, при этом значение переменной $x = 2$. Таким образом, получаем очередное состояние РКА с меткой $s_0, s_0, x = 2$. Этот процесс продолжается до тех пор, пока не будет обнаружен цикл, либо пока не закончится память, доступная процессу генератору. Заметим, что даже при небольшом числе состояний исходного конечного автомата, число состояний расширенного автомата достаточно велико, и зависит нескольких факторов, в частности от количества переменных и количества возможных значений этих переменных.

Таким образом, для генерации РКА необходимы:

1) базовый конечный автомат (строится по формальному описанию модели);
2) список переменных модели (в данном примере список состоял из одной целочисленной переменной x).

Рассмотрим подробнее базовый конечный автомат модели. Фактически, БКА является *конечным автоматом с выходом*. Конечным автоматом с выходом называется шестерка вида:

$$M = (V, W, S, s_0, F, \delta, \mu),$$

где V - входной алфавит, W – выходной алфавит, S - конечное множество состояний; $s_0 \in S$ - начальное состояние; $F \subseteq S$ - множество заключительных состояний; $\delta: Q \times V \to Q$ - функция переходов; $\mu: Q \times V \to W$ - функция выходов.

Входным алфавитом БКА является множество предикатов: каждому символу входного алфавита ставится в соответствие предикат из множества предикатов, которое строится по формальному описанию модели. Каждый предикат может принимать два значения – *“истина”* или *“ложь”*. *Выходным алфавитом БКА* является множество действий: каждому символу выходного алфавита ставится в соответствие действие из

множества действий, которое также строится по формальному описанию модели. На рис. 2-4 приведен пример конечного автомата с выходом, у которого:

- входной алфавит $V = \{x < 4, x == 4\}$;
- выходной алфавит $W = \{x = x + 1, x = 0\}$;
- множество состояний $Q = \{s_0, s_1\}$ при начальном состоянии s_0 и заключительном состоянии s_1.

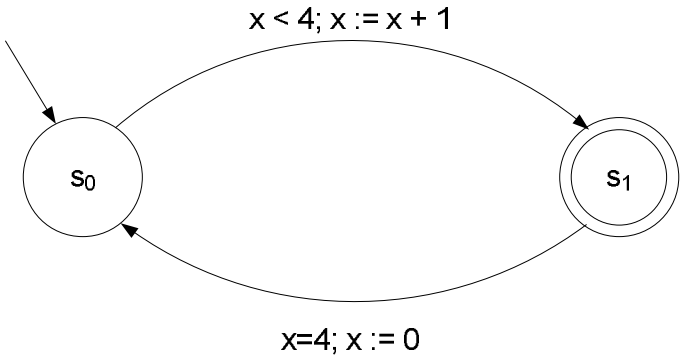


Рисунок 2-4. Пример конечного автомата модели с выходом

Для базового конечного автомата A_1, соответствующего процессу A приведенной выше модели (см. рис. 2-5) не на всех переходах предикаты и действия указаны явно.

Пусть:

- λp – пустой предикат (пустой символ входного алфавита БКА), значение предиката λp всегда равно *"истина"*;
- λa – пустое действие (пусто символ входного алфавита БКА), что соответствует пустому оператору в языках программирования (как правило, его обозначают nop).

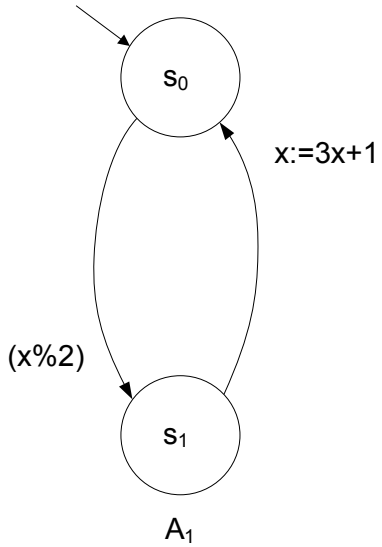


Рисунок 2-5. Конечный автомат процесса *A*

Тогда автомат A_1 можно изобразить следующим образом (см. рис. 2-6):

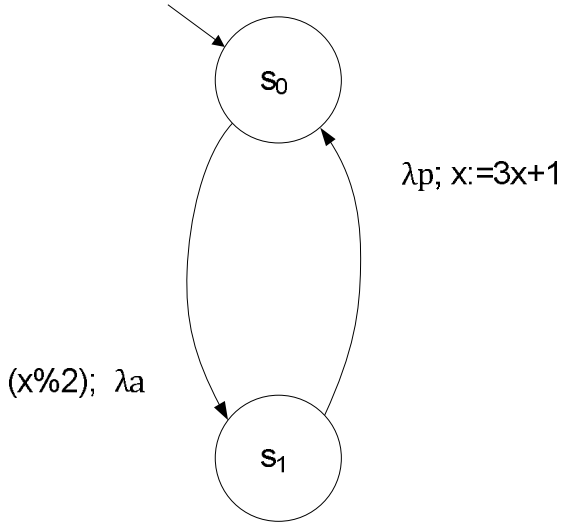


Рисунок 2-6. Конечный автомат процесса A с указанными метками предикатов и действий

Подобный подход связан с тем, что как правило, часто строятся модели параллельных систем (в том числе и в приведенном выше примере формального описания модели). В этом случае каждый оператор должен быть атомарным (т.е. неделимым), соответственно переход из одного состояния модели в другое также должен быть неделимым. Только таким образом можно получить все возможные состояния модели. Ниже подробно рассмотрен псевдокод алгоритма генерации РКА по заданному БКА.

2.2 Алгоритм генерации расширенного конечного автомата по заданному базовому конечному автомату

Алгоритм построения расширенного автомата по заданному автомату модели основан на алгоритмах поиска на графах - поиске в глубину или в ширину. Ниже приводится псевдокод генерации расширенного конечного автомата с использованием поиска в глубину. Для простоты список переменных не указывается явно в псевдокоде, однако следует помнить, что все переходы осуществляются исходя из значений переменных в списке в данный момент времени. Процедура ***Generator*** имеет следующие параметры:

1) $baseFsm$ – исходный базовый конечный автомат;
2) $expandedFsm$ – получаемый в результате генерации расширенный конечный автомат.

```
procedure Generator(baseFsm, expandedFsm)
begin
    queue ≔ ∅;          /* очередь неисследованных состояний */
    expandedFsm.states ≔ ∅;
    s ≔ CreateStartState(baseFsm.startState);
    push(queue, s);
    while nonempty( queue) do
        s ≔ front(queue); pop(queue);
        for each t in EnabledTranstions(s, baseFsm) do
            state ≔ CreateState(s, t, basefsm);
            if not found(state, expandedFsm.states) then
                push(queue, state);
                AddState(expandedFsm, state);
            endif;
            t.indexFrom ≔ s.index; t.indexTo ≔ state.index;
            AddTransition(expandedFsm, t, s.index, state.index);
        endfor;
    endwhile;
end.
```

Заметим, что вне зависимости от выбранного алгоритма поиска на графах, особенностью при реализации алгоритма генерации расширенного конечного автомата является необходимость осуществлять поиск состояний в исходном автомате, в результате которого генерируются состояния расширенного конечного автомата. Эта особенность создает некоторые трудности в реализации алгоритма генерации расширенного конечного автомата.

Основные этапы алгоритма генерации расширенного конечного автомата:

1) генерация нового состояния расширенного конечного автомата;
2) поиск состояния во множестве состояний расширенного конечного автомата;
3) добавление состояния и перехода в расширенный конечный автомат.

Как уже отмечалось выше, даже при достаточно небольшом исходном конечном автомате модели число состояний расширенного автомата может быть очень велико, причем настолько, что объема памяти вычислительной машины может быть недостаточно

для хранения всего множества состояний. Поэтому возникает необходимость решения следующих задач:

1) уменьшение времени поиска состояния в расширенном конечном автомате;
2) обеспечение возможности хранения большого (не умещающегося в памяти одного компьютера) множества состояний.

Наиболее очевидным способом решения первой задачи является хеширование. Ниже рассмотрены основные способы хеширования.

2.3 Хеширование

Проблема поиска состояний в пространстве состояний является одной из основных проблем, возникающих при построении расширенного конечного автомата модели. Для сложных моделей с большим числом состояний поиск конкретного состояния может занимать достаточно большое время.

Для решения проблемы поиска в пространстве состояний применяется *хеширование*.

Хеш-таблица представляет собой эффективную структуру данных для реализации словарей. Хотя на поиск элемента в хеш-таблице может в наихудшем случае потребоваться столько же времени, что и в связанном списке, а именно $\Theta(n)$, на практике хеширование исключительно эффективно. При вполне обоснованных допущениях математическое ожидание времени поиска элемента в хеш-таблице составляет $O(1)$.

Хеш-таблица представляет собой обобщение обычного массива. Возможность прямой индексации элементов обычного массива обеспечивает доступ к произвольной позиции в массиве за время $O(1)$. Если количество реально хранящихся в массиве ключей мало по сравнению с количеством возможных значений ключей, эффективной альтернативой массива с прямой индексацией становится хеш-таблица, которая обычно использует массив с размером, пропорциональным количеству реально хранящихся в нем ключей. Вместо непосредственного использования ключа в качестве индекса массива, индекс вычисляется по значению ключа.

Ниже рассмотрены основные идеи и особенности хеширования, а также способы разрешения коллизий (когда несколько ключей отображается в один и тот же индекс массива).

Основными операциями для хеш-таблиц (как и для словарей) являются:

- Search (T, k) – поиск элемента в хеш-таблице T по ключу k,
- Insert (T, x) – Вставка элемента x в хеш-таблицу T,
- Delete(T, x) - удаление элемента x из хеш-таблицы T.

2.3.1 Таблицы с прямой адресацией

Прямая адресация представляет собой простейшую технологию, которая хорошо работает для небольших множеств ключей. Предположим, что приложению требуется динамическое множество, каждый элемент которого имеет ключ из множества $U = \{0, 1, \ldots m - 1\}$, где m не слишком велико. Кроме того, предполагается, что никакие два элемента не имеют одинаковых ключей.

Для представления динамического множества мы используем массив, или таблицу с прямой адресацией, который обозначим как $T \in [0 \ldots m — 1]$, каждая позиция, или ячейка (position, slot), которого соответствует ключу из пространства ключей U. На рис. 2-7 представлен данный подход.

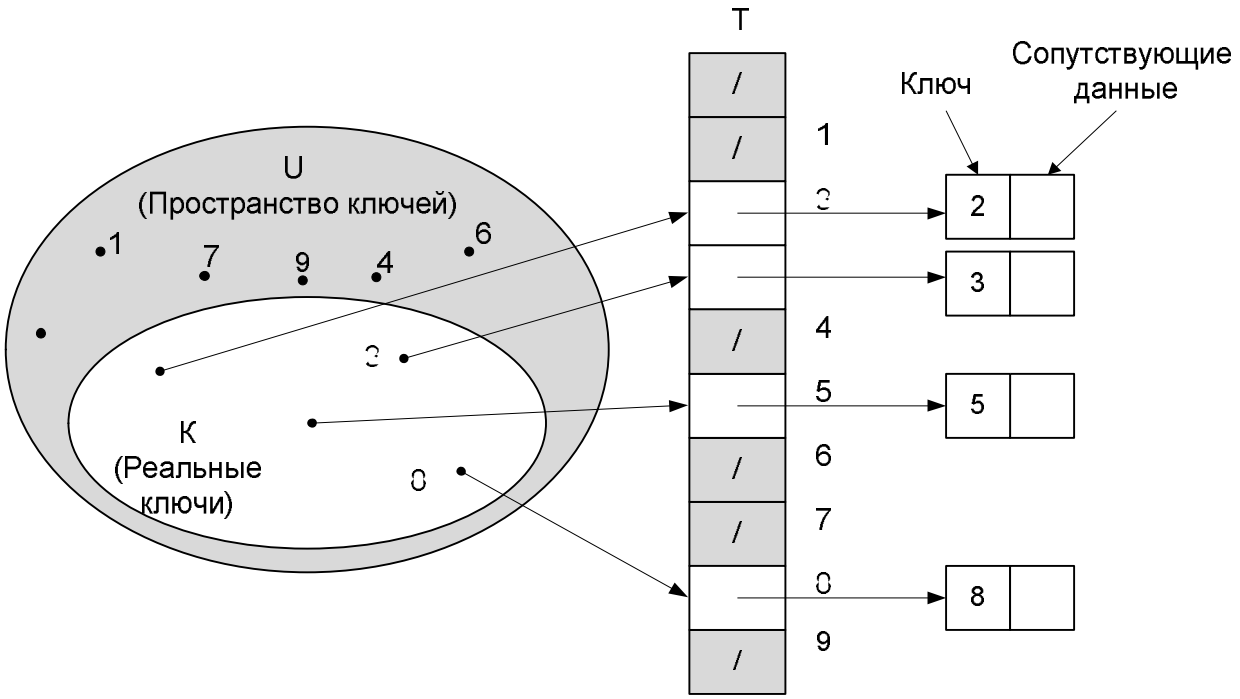


Рисунок 2-7. Пример таблицы с прямой адресацией

Ячейка k указывает на элемент множества с ключом k. Если множество не содержит элемента с ключом k, то $T[k] = \textbf{NIL}$. На рисунке каждый ключ из пространства $U = \{0, 1, \ldots, 9$ соответствует индексу таблицы. Множество реальных ключей $K = \{2, 3, 5, 8\}$ определяет ячейки таблицы, которые содержат указатели на элементы. Остальные ячейки (закрашенные темным цветом) содержат значение **NIL**.

Реализация основных операций тривиальна:

Direct_Address_Search(T, k)

return $T[k]$;

Direct_Address_Insert(T, x)

$T[key[x]] := x$;

Direct_Address_Delete(T, x)

$T[key[x]] :=$ **NIL**;

Каждая их приведенных операций очень быстрая, время их работы равно $O(1)$.

В некоторых приложениях элементы динамического множества могут храниться непосредственно в таблице с прямой адресацией. То есть вместо хранения ключей и сопутствующих данных элементов в объектах, внешних по отношению к таблице с прямой адресацией, а в таблице — указателей на эти объекты, эти объекты можно хранить непосредственно в ячейках таблицы (что тем самым приводит к экономии используемой памяти). Кроме того, зачастую хранение ключа не является необходимым условием, поскольку если известен индекс объекта в таблице, то тем самым известен и его ключ. Однако если ключ не хранится в ячейке таблицы, то необходим какой-то иной механизм для того, чтобы помечать пустые ячейки.

2.3.2 Хеш–таблицы

Недостаток прямой адресации очевиден: если пространство ключей U велико, хранение таблицы T размером $|U|$ непрактично, а то и вовсе невозможно — в зависимости от количества доступной памяти и размера пространства ключей. Кроме того, множество K реально сохраненных ключей может быть мало по сравнению с пространством ключей U, а в этом случае память, выделенная для таблицы T, в основном расходуется напрасно.

Когда множество K хранящихся в словаре ключей гораздо меньше пространства возможных ключей U, хеш-таблица требует существенно меньше места, чем таблица с прямой адресацией. Точнее говоря, требования к памяти могут быть снижены до $\Theta(|U|)$, при этом время поиска элемента в хеш-таблице остается равным $O(1)$. Следует заметить, что это граница среднего времени поиска, в то время как в случае таблицы с прямой адресацией эта граница справедлива для наихудшего случая.

В случае прямой адресации элемент с ключом k хранится в ячейке k. При хешировании этот элемент хранится в ячейке $h(k)$, т.е. мы используем *хеш-функцию* h для вычисления ячейки для данного ключа k. Функция h отображает пространство ключей U на ячейки хеш-таблицы $T[0 \ldots m-1]$:

$$h: U \to \{0, 1, \ldots m-1\}.$$

Говорят, что элемент с ключом к *хешируется* в ячейку $h(k)$. Величина $h(k)$ называется *хеш-значением* ключа k. На рис. 2-8 представлена основная идея хеширования. Цель хеш-функции состоит в том, чтобы уменьшить рабочий диапазон индексов массива,

и вместо $|U|$ значений мы можем обойтись всего лишь m значениями. Соответственно снижаются и требования к количеству памяти.

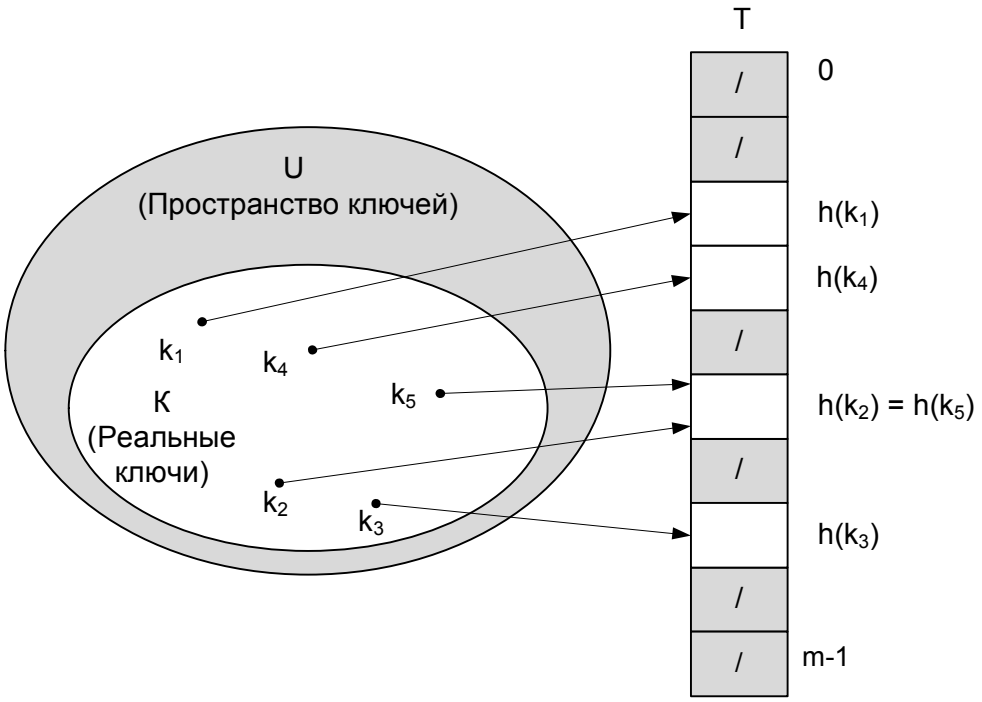


Рисунок 2-8. Пример хеш-таблицы

Основной проблемой при использовании хеширования является возникновение *коллизий*, когда два и более ключа хешируются в одну и ту же ячейку. Заметим, что количество коллизий во многом зависит от выбранной хеш-функции. Однако, доказано, что полностью избавиться от коллизий невозможно в принципе, хорошая хеш-функция способна лишь минимизировать количество коллизий.

На данный момент существует несколько достаточно эффективных способов разрешения коллизий, наиболее распространенные из них рассмотрены ниже.

2.3.3 Хеш-функции

Хеш-функция – это некоторая функция $h(k)$, которая по некоторому ключу k и возвращает адрес, по которому производится поиск в хеш-таблице, чтобы получить информацию, связанную с ключом k.

Выбор хеш-функции во многом определяет эффективность хеширования. Хорошая хеш-функция должна удовлетворять следующим требованиям:

- хеш-функция должна вычисляться быстро;
- хеш-функция должна минимизировать количество коллизий.

Можно выделить следующие основные схемы построения хеш-функций:

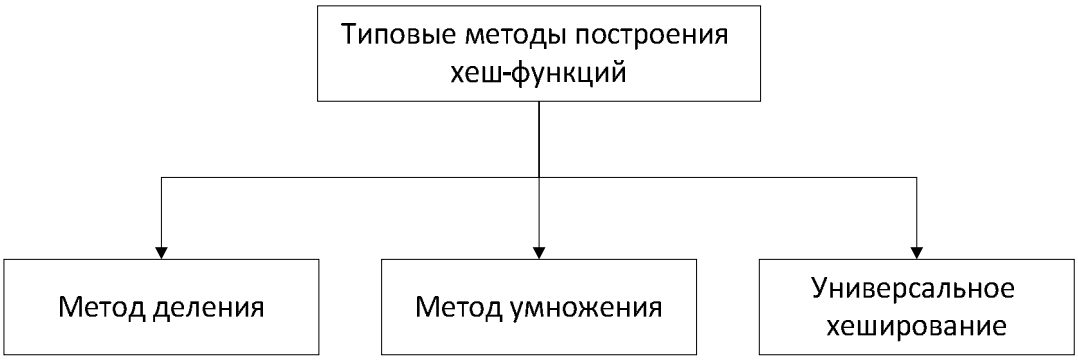


Рисунок 2-9. Способы построения хеш-функций

Метод деления

Метод деления – наиболее простой и распространенный из всех методов. Он заключается отображении ключа k в одну из ячеек путем получения остатка от деления на m, т.е. хеш-функция имеет вид:

$$h(k) = k \; mod \; m.$$

Поскольку для вычисления хеш-функции требуется только одна операция деления, хеширование методом деления считается достаточно быстрым.

Например, при размере хеш-таблицы $m = 12$ и значении ключа $k = 100$ получим $h(k) = 4$.

Основной проблемой при использовании метода деления является выбор значения m. Например, если взять $m = 100$, а ключом k будет служить год рождения, то распределение будет очень неравномерным для ряда задач (идентификация игроков юношеской бейсбольной лиги, например). Более того, при четном значении m значение функции будет четным при четном ключе k и нечетным - при нечетном ключе, что приведет к нежелательному результату. Также m не должно быть степенью 2, поскольку, если $m = 2^p$, то $h(k)$ представляет собой p младших битов числа k. Желательно выбирать хеш-функцию таким образом, чтобы ее результат зависел от всех битов числа. Выбор $m = 2^p - 1$ также неудачен, поскольку если ключи представляют собой строки символов, интерпретируемые как числа в системе счисления с основанием 2^p, поскольку перестановка символов ключа не приводит к изменению его хеш-значения. Хорошие результаты часто можно получить, выбирая в качестве значения m простое число, достаточно далекое от степени двойки.

Метод умножения

Построение хеш-функции методом умножения выполняется в два этапа. Сна-Сначала мы умножаем ключ k на константу $0 < A < 1$ и получаем дробную часть полученного произведения. Затем мы умножаем полученное значение на m и при-применяем к нему функцию "пол" т.е.

$$h(k) = [\, m\ (kA\ mod\ 1)\ ],$$

где выражение "$kA\ mod\ 1$" означает получение дробной части произведения kA. Оператор [ ] возвращает наибольшее целое, которое меньше аргумента.

Достоинство метода умножения заключается в том, что значение m перестает быть критичным. Обычно величина m из соображений удобства реализации функции выбирается равной степени 2.

Основной проблемой при использовании метода умножения является выбор константы A. Пусть, например, имеется компьютер с размером слова w битов и k помещается в одно слово. Ограничим возможные значения константы A видом $s/2^w$, где s — целое число из диапазона $0 < s < 2^w$. Тогда мы сначала умножаем k на w-битовое целое число $s = A * 2^w$. Результат представляет собой $2w$-битовое число $r_1 2^w + r_0$, где r_1 — старшее слово произведения, а r_0 — младшее. Старшие p битов числа r_0 представляют собой искомое p-битовое хеш- значение (см. рис. 2-10).

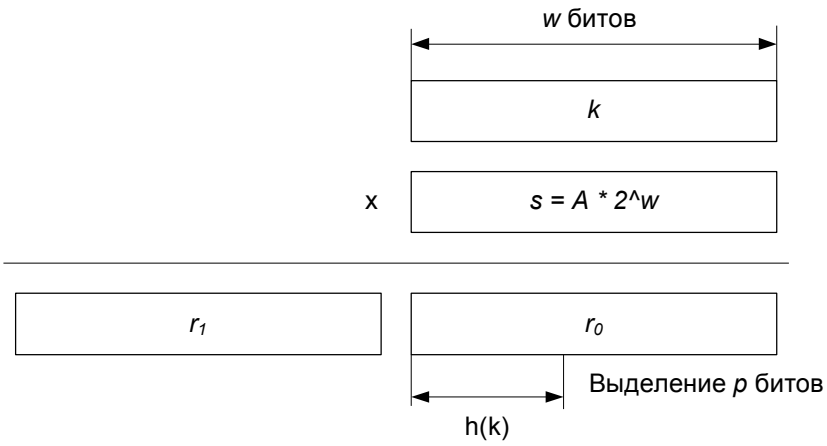


Рисунок 2-10. Вычисления хеш-значения в методе умножения

Хотя описанный метод работает с любыми значениями константы A, некоторые значения дают лучшие результаты по сравнению с другими. Оптимальный выбор зависит от характеристик хешируемых данных. Д. Кнут предложил использовать в качестве значения константы A значение «золотого сечения»:

$$A \approx \frac{\sqrt{5}-1}{2} = 0.6180339887\ldots$$

Возьмем в качестве примера $k = 123456,\ p = 14,\ m = 2^{14} = 16384,\ w = 32.$ Принимая предложение Кнута, выбираем значение A в виде $\frac{s}{2w}$, ближайшее к величине золотого сечения, так что $A = 2654435769/2^{32}$. Тогда

$$k * s = 327\,706\,022\,297\,664 = (76\,300 * 2^{32}) + 17\,612\,864,$$

и, соответственно, $r_1 = 76\,300$ и $r_0 = 17\,612\,864$. Старшие 14 битов числа r_0 дают хеш-значение $h(k) = 67$.

Универсальное хеширование

Если умышленно выбирать ключи для хеширования при помощи конкретной хеш-функции, то можно подобрать n значений, которые будут хешироваться в одну и ту же ячейку таблицы, приводя к среднему времени выборки $\Theta(n)$. Таким образом, любая фиксированная хеш-функция становится уязвимой, и единственный эффективный выход из ситуации — случайный выбор хеш-функции, не зависящий от того, с какими именно ключами ей предстоит работать. Такой подход, который называется универсальным хешированием, гарантирует хорошую производительность в среднем, независимо от того, какие данные будут выбраны. Главная идея универсального хеширования состоит в случайном выборе хеш-функции из некоторого тщательно отобранного класса функций в начале работы программы. Рандомизация гарантирует, что одни и те же входные данные не могут постоянно давать наихудшее поведение алгоритма. В силу рандомизации алгоритм будет работать всякий раз по-разному, даже для одних и тех же входных данных, что гарантирует высокую среднюю производительность для любых входных данных. Например, если имеется таблица символов компилятора, мы обнаружим, что никакой выбор программистом имен идентификаторов не может привести к постоянному снижению производительности хеширования. Такое снижение возможно только тогда, ко-когда компилятором выбрана случайная хеш-функция, которая приводит к плохому хешированию конкретных входных данных; однако вероятность такой ситуации очень мала и одинакова для любого множества идентификаторов одного и того же размера.

Пусть H — конечное множество хеш-функций, которые отображают пространство ключей U в диапазон $\{0, 1, 2, \ldots, m-1\}$. Такое множество называется ***универсальным***, если для каждой пары различных ключей $k, l \in U$ количество хеш-функций $h \in H$, для которых $h(k) = h(l)$, не превышает $|H|/m$. Другими словами, при случайном выборе

хеш-функции из Л вероятность коллизии между различными ключами k и l не превышает вероятности совпадения двух случайным образом выбранных хеш-значений из множества $\{0, 1, 2, \ldots, m-1\}$, которая равна $1/m$. Существуют несколько теорем, доказывающих, что универсальное хеширование показывает хорошую среднюю производительность (см. [11]).

Построить универсальное множество хеш-функций можно следующим образом. Выберем простое число p, достаточно большое, чтобы все ключи находились в диапазоне от 0 до $p-1$ включительно. Пусть Z_p обозначает множество $0, 1, \ldots, p-1\}$, а Z^* - множество $\{1, 2, \ldots, p-1\}$. Поскольку p - простое число, мы можем решать уравнения по модулю p при помощи методов теории чисел (см [11]). Из предположения о том, что пространство ключей больше, чем количество ячеек в хеш-таблице, следует, что $p > m$. Теперь определим хеш-функцию $h_{a,b}$ для любых $a \in Z^*$ и $b \in Z_p$ следующим образом:

$$h_{a,b}(k) = \big((ak + b)\ mod\ p\big)\ mod\ m.$$

Например, при $p = 17$ и $m = 6$ $h_{a,b}(8) = 5$. Семейство всех таких функций образует множество:

$$H_{p,m} = \{h_{a,b} : a \in Z_p^* \text{ и } b \in Z_p\}$$

Каждая хеш-функция $h_{a,b}$ отображает Z_p на Z_m. Этот класс хеш-функций обладает тем свойством, что размер m выходного диапазона произволен и не обязательно представляет собой простое число. Поскольку число a можно выбрать $p-1$ способом, и p способами - число b, всего во множестве $H_{p,m}$ содержится $p\ (p-1)$ хеш-функций.

2.3.4 Методы разрешения коллизий

Коллизия – это ситуация, когда для двух (или более) ключей k_1 *и* k_2 *и* $k_1 \neq k_2$ *и хеш − функции* h *выполняется* $h(k_1) = h(k_2)$. В этом случае, необходимо найти новое место для хранения данных. Очевидно, что количество коллизий необходимо минимизировать. Как уже отмечалось выше, число коллизий зависит от выбора хеш-функции, хорошая хеш-функция позволяет минимизировать количество коллизий, но никакая хеш-функция не позволяет полностью избавиться от коллизий. Устранить возникновение коллизий невозможно в принципе. Поэтому для разрешения коллизий используются специальные методы. Основными методами разрешения коллизий являются метод цепочек и метод открытой адресации.

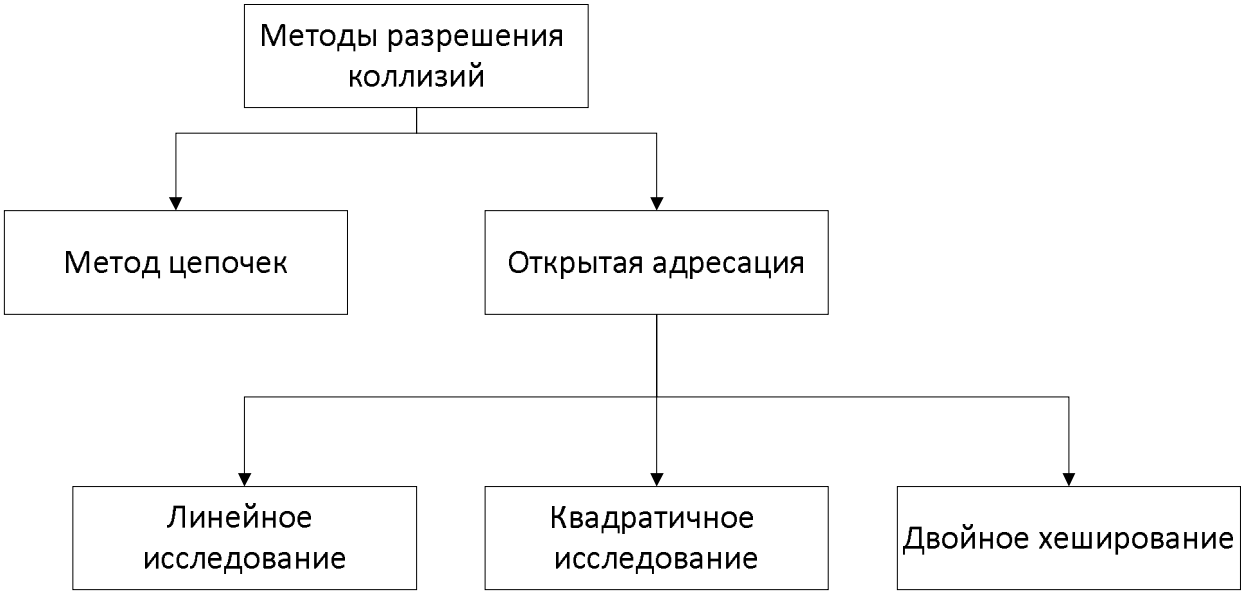


Рисунок 2-11. Методы разрешения коллизий

Метод разрешения коллизий при помощи цепочек

Это самый простой метод разрешения коллизий. При использовании данного метода все элементы, хешированные в одну и ту же ячейку объединяются в связный список (см. рис. 2-12).

Ячейка j содержит указатель на заголовок списка всех элементов, хеш-значение ключа которых равно j; если таких элементов нет, ячейка содержит значение NIL. На рис. 2-12 показано разрешение коллизий, возникающих из-за того, что $h(k_1) = h(k_4),\ h(k_5) = h(k_2) = h(k_7), h(k_8) = h(k_6)$. Основные операции реализуются достаточно просто:

1) Insert(T, x)

   Вставить x в заголовок списка $T[h(key[x])]$

2) Search(T, k)

   Поиск элемента с ключом k в списке $T[h(k)]$

3) Delete(T, x)

   Удаление x из списка $T[h(key[x])]$

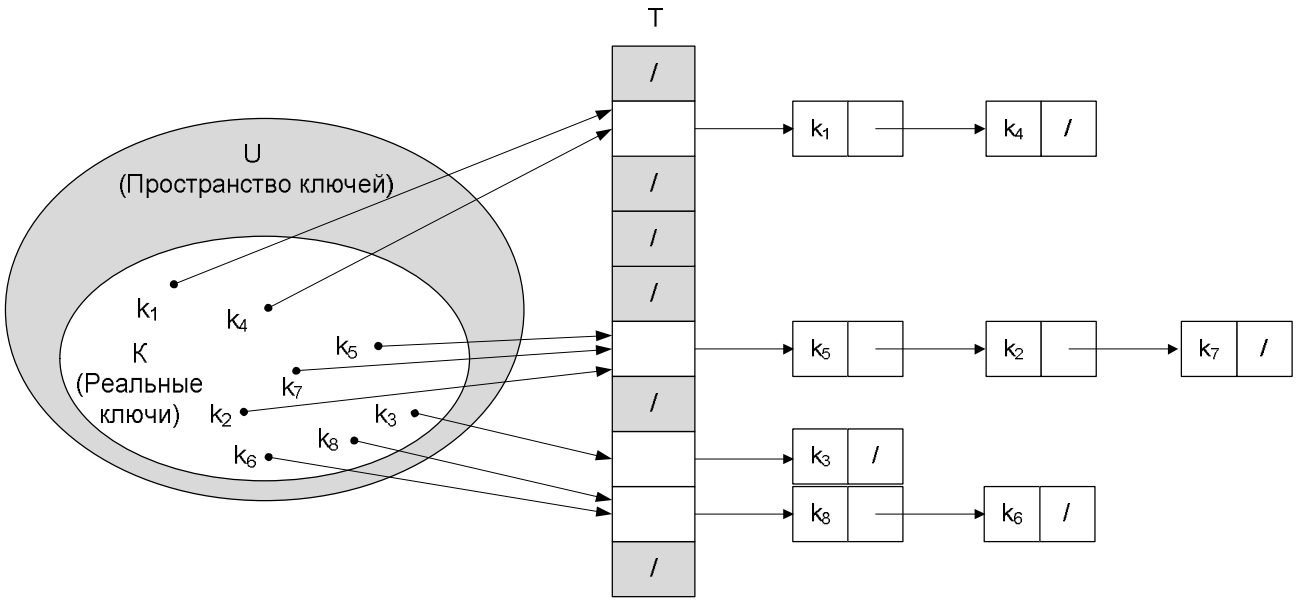


Рисунок 2-12. Разрешение коллизий методом цепочек

Недостатком хеширования с цепочками является необходимость поиска элемента в списке, соответствующем ячейке $h(key[x])$. Эффективность такого метода зависит от того, на сколько равномерно хеш-функция h распределяет множество сохраняемых ключей по m ячейкам. В наихудшем случае все n ключей хешируются в одну и ту же ячейку, создавая список длиной n. В этом случае время выполнения всех операций будет напрямую зависеть от длины списка, и, таким образом, выигрыша при использовании хеш-таблиц в сравнении со связными списками не будет.

Также доказано, что использование универсального хеширования и разрешения коллизий методом цепочек в хеш-таблице с m ячейками дает математическое ожидание времени выполнения любой последовательности из n вставок, поисков и удалений, в которой содержится $O(m)$ вставок, равное $\Theta(n)$ (см. [11]).

Метод открытой адресации разрешения коллизий

При использовании метода *открытой адресации* все элементы хранятся непосредственно в хеш-таблице, т.е. каждая запись таблицы содержит либо элемент динамического множества, либо значение NIL. При поиске элемента систематически проверяются ячейки таблицы до тех пор, пока не будет найден искомый элемент или пока не убедимся в его отсутствии в таблице. Здесь, в отличие от метода цепочек, нет ни списков, ни элементов, хранящихся вне таблицы. Таким образом, ***в методе открытой адресации хеш-таблица может оказаться заполненной, делая невозможной вставку***

новых элементов; *коэффициент заполнения* α не может превышать 1. Конечно, при хешировании с разрешением коллизий методом цепочек можно использовать свободные места в хеш-таблице для хранения связанных списков, но преимущество открытой адресации заключается в том, что она позволяет полностью отказаться от указателей. Вместо того чтобы следовать по указателям, *вычисляется* последовательность проверяемых ячеек. Дополнительная память, освобождающаяся в результате отказа от указателей, позволяет использовать хеш-таблицы большего размера при том же общем количестве памяти, потенциально приводя к меньшему количеству коллизий и более быстрой выборке.

Для выполнения вставки при открытой адресации последовательно проверяются, или *исследуются* (probe), ячейки хеш-таблицы до тех пор, пока не будет найдена пустая ячейка, в которую и помещается вставляемый ключ. Вместо фиксированного порядка исследования ячеек $0, 1, \dots, m-1$ (для чего требуется $\Theta(n)$ времени), последовательность исследуемых ячеек *зависит от вставляемого в таблицу ключа*. Для определения исследуемых ячеек мы расширим хеш-функцию, включив в нее в качестве второго аргумента номер исследования (начинающийся с 0). В результате хеш-функция имеет вид:

$$h: U \times \{0, 1, \dots, m-1\} \to \{0, 1, \dots, m-1\}.$$

В методе открытой адресации требуется, чтобы для каждого ключа k последовательность исследований $\langle h(k,0), h(k,1), \dots h(k, m-1)\rangle$ представляла собой перестановку множества $\langle 0, 1, \dots, m-1\rangle$, чтобы, в конечном счете, могли быть просмотрены все ячейки хеш-таблицы. В приведенном далее псевдокоде предполагается, что элементы в таблице T представляют собой ключи без сопутствующей информации; ключ к тождественен элементу, содержащему ключ k. Каждая ячейка содержит либо ключ, либо значение **NIL** (если она не заполнена):

```
procedure Hash_Insert(T, k)
begin
    i := 0;
    repeat j: = h(k, i)
        if T[j] = NIL then
            T[j] := k;
            return j;
        else i: = i + 1;
```

```
            endif;
        until i = m;
        error Хеш-таблица переполнена
end.
```

Алгоритм поиска ключа k исследует ту же последовательность ячеек, что и алгоритм вставки ключа k. Таким образом, если при поиске встречается пустая ячейка, поиск завершается неуспешно, поскольку ключ k должен был бы быть вставлен в эту ячейку в последовательности исследований, и никак не позже нее. (Здесь предполагается, что удалений из хеш-таблицы не было.) Процедура Hash_Search получает в качестве входных параметров хеш-таблицу T и ключ k и возвращает номер ячейки, которая содержит ключ k (или значение **NIL**, если ключ в хеш - таблице не обнаружен):

```
procedure Hash_Search(T, k)
begin
        i: = 0;
        repeat j := h(k, i)
                if T[j] = k then
                        return j;
                i := i + 1;
        until T[j] = NIL or i = m;
        return NIL;
end.
```

Процедура удаления из хеш-таблицы с открытой адресацией достаточно сложна. При удалении ключа из ячейки i нельзя просто пометить ее значением **NIL**, поскольку при этом может стать невозможной выборку ключа k, в процессе вставки которого исследовалась и оказалась занятой ячейка i. Одно из решений состоит в том, чтобы помечать такие ячейки специальным значением $deleted$ вместо **NIL**. В этом случае нужно изменить процедуру Hash_Insert так, чтобы она рассматривала такую ячейку как пустую и могла вставить в нее новый ключ. В процедуре поиска Hash_Search никакие изменения не требуются, поскольку такие ячейки при поиске просто пропускаются, и исследуются следующие ячейки в последовательности. Однако при использовании специального значения $deleted$ время поиска перестает зависеть от коэффициента заполнения α, и по

этой причине, как правило, при необходимости удалений из хеш-таблицы в качестве метода разрешения коллизий выбирается метод цепочек.

При описании метода цепочек для разрешения коллизий отмечалось, что эффективность хеширования цепочками зависит от того, насколько равномерно хеш-функция распределяет множество ключей по ячейкам хеш-таблицы. Это утверждение также верно и для метода открытой адресации. То есть, для каждого ключа в качестве последовательности исследований равновероятны все $m!$ перестановок множества $\{0, 1, \dots, m-1\}$. В случае метода открытой адресации хеш-функция возвращает не одно значение, а целую последовательность исследований. На практике реализация истинно равномерного хеширования достаточно трудна, и поэтому используются его аппроксимации (например, двойное хеширование).

Для вычисления последовательности исследований для открытой адресации обычно используются три метода:

- линейное исследование,
- квадратичное исследование,
- двойное хеширование.

Эти методы гарантируют, что для каждого ключа k $\langle h(k,0), h(k,1), \dots h(k,m-1)\rangle$ является перестановкой $\langle 0, 1, \dots, m-1\rangle$. Эти методы, однако, не дают действительно равномерного распределения, поскольку могут сгенерировать не более m^2 различных последовательностей исследований вместо $m!$, требующихся для равномерного хеширования. Наибольшее количество последовательностей исследований (и, следовательно, наилучшие результаты) дает равномерное хеширование.

Перед тем, как перейти непосредственно к рассмотрению вышеперечисленных методов, заметим, что обычную хеш-функцию $h' : U \to \{0, 1, \dots, m-1\}$ в дальнейшем при рассмотрении вышеуказанных методов будем называть *вспомогательной хеш-функцией*.

Линейное исследование

Метод линейного исследования для вычисления последовательности исследований использует хеш-функцию вида:

$$h\,(k,i) \;=\; (h'(k) \;+\; i) \bmod m,$$

где $0 \le i \le m-1$. Для данного ключа k первой исследуемой ячейкой является $T[h'(k)]$ т.е. ячейка, которую дает вспомогательная хеш-функция. Далее исследуется ячейка $T[h'(k)+1]$ и далее последовательно все до ячейки $T[m-1]$, после чего осуществляется переход в начало таблицы и последовательно исследуются ячейки $T[0], T[1], \dots, T[h'(k)-1]$. Поскольку начальная исследуемая ячейка однозначно

определяет всю последовательность исследований целиком, всего имеется m различных последовательностей.

Линейное исследование легко реализуется, однако с ним связана проблема *первичной кластеризации* из-за создания длинных последовательностей занятых ячеек, что увеличивает среднее время поиска. Кластеры возникают в связи с тем, что вероятность заполнения пустой ячейки, которой предшествуют i заполненных ячеек, равна $(i + 1)/m$. Таким образом, длинные серии заполненных ячеек имеют тенденцию ко все большему удлинению, что приводит к увеличению среднего времени поиска.

Квадратичное исследование

Квадратичное исследование использует хеш-функцию вида

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m,$$

где h' — вспомогательная хеш-функция, c_1 и $c_2 \neq 0$ — вспомогательные константы, и $0 \leq i \leq m - 1$. Начальная исследуемая ячейка – $T[h'(k)]$; остальные исследуемые позиции смещены относительно нее на величины, которые описываются квадратичной зависимостью от номера исследования i. Этот метод работает существенно лучше линейного исследования, но для того, чтобы исследование охватывало все ячейки, необходим выбор специальных значений c_1, c_2 и m. Кроме того, если два ключа имеют одну и то же начальную позицию исследования, то одинаковы и последовательности исследования в целом, так как из $h_1(k, 0) = h_2(k, 0)$ следует $h_i(k, 0) = h_i(k, 0)$. Это свойство приводит к более мягкой *вторичной кластеризации*. Как и в случае линейного исследования, начальная ячейка определяет всю последовательность, поэтому всего используется m различных последовательностей исследования.

Двойное хеширование

Двойное хеширование представляет собой один из наилучших способов использования открытой адресации, поскольку получаемые при этом перестановки обладают многими характеристиками случайно выбираемых перестановок. Двойное хеширование использует хеш-функцию вида:

$$h\,(k, i) \;=\; (h_1(k) \;+\; i h_2(k)) \bmod m,$$

где h_1 и h_2 - вспомогательные хеш-функции. Начальное исследование выполняется в позиции $T\,[h_1(k)]$, а смещение каждой из последующих исследуемых ячеек относительно предыдущей равно $h_2(k)$ по модулю k. Следовательно, в отличие от линейного и квадратичного исследования, в данном случае последовательность исследования зависит от ключа k по двум параметрам — в плане выбора начальной исследуемой ячейки и

расстояния между соседними исследуемыми ячейками, так как оба эти параметра зависят от значения ключа. На рис. 2-13 показан пример вставки при двойном хешировании. Хеш-таблица на этом рисунке имеет размер 13 ячеек; используются вспомогательные хеш-функции $h_1(k) = k\ mod\ 13$ и $h_2(k) = 1 + (k \bmod 11)$. Так как $14 \equiv 1 \pmod{13}$ и $14 \equiv 3 \pmod{11}$, ключ 14 вставляется в пустую ячейку 9, после того как при исследовании ячеек 1 и 5 выясняется, что эти ячейки заняты. Для того чтобы последовательность исследования могла охватить всю таблицу, значение $h_2(k)$ должно быть взаимно простым с размером хеш-таблицы m.

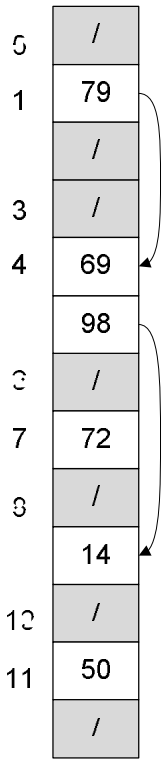


Рисунок 2-13. Вставка в хеш-таблицу при двойном хешировании

Удобный способ обеспечить выполнение этого условия состоит в выборе числа m, равного степени 2, и выборе хеш-функции h_2 так, чтобы она возвращала только нечетные значения. Другой способ состоит в использовании в качестве m простого числа и построении хеш-функции h_2 такой, чтобы она всегда возвращала натуральные числа, меньшие m.

Двойное хеширование превосходит линейное или квадратичное исследования в смысле количества $\Theta(m^2)$ последовательностей исследований, в то время как у упомянутых методов это количество равно $\Theta(m)$. Это связано с тем, что каждая возможная пара $(h_1(k),\ h_2(k))$ дает свою, отличающуюся от других, последовательность исследований. В результате производительность двойного хеширования достаточно близка к производительности "идеальной" схемы равномерного хеширования.

2.3.5 Идеальное хеширование

В данной работе этот метод хеширования не использовался в связи с тем, что для его реализации необходимо решать ряд специфических задач (например, необходимо гарантировать отсутствие коллизий в хеш-таблицах второго уровня). Поэтому в данном разделе приводится лишь краткое описание метода идеального хеширования.

Хотя чаще всего хеширование используется из-за превосходной средней производительности, возможна ситуация, когда реально получить превосходную производительность хеширования в наихудшем случае. Такой ситуацией является статическое множество ключей, т.е. после того как все ключи сохранены в таблице, их множество никогда не изменяется. Ряд приложений в силу своей природы работает со статическими множествами ключей. В качестве примера можно привести множество зарезервированных слов языка программирования или множество имен файлов на компакт-диске. Идеальным хешированием называют методику, которая в наихудшем случае выполняет поиск за $O(1)$ обращений к памяти. Основная идея идеального хеширования достаточно проста, и заключается в использовании двухуровневой схемы хеширования с универсальным хешированием на каждом уровне. Первый уровень по сути тот же, что и в случае хеширования с цепочками: n ключей хешируются в m ячеек с использованием хеш-функции h, тщательно выбранной из семейства универсальных хеш-функций. Однако вместо того, чтобы создавать список ключей, хешированных в ячейку j, используют маленькую вторичную хеш-таблицу S_j со своей хеш-функцией h_j. Путем точного выбора хеш-функции h_j можно гарантировать отсутствие коллизий на втором уровне. Для того чтобы гарантировать отсутствие коллизий на втором уровне, требуется, чтобы размер m_j хеш-таблицы S_j был равен квадрату числа n_j ключей, хешированных в ячейку j. Такая квадратичная зависимость m_j от n_j может показаться чрезмерно расточительной, однако можно показать, что при корректном выборе хеш-функции первого уровня ожидаемое количество требуемой для хеш-таблицы памяти остается равным $O(n)$. Хеш-функция первого уровня выбирается из множества $H_{p,m}$ универсальных хеш-функций, где p является простым числом, превышающим значение любого из ключей. Ключи, хешированные в ячейку j, затем повторно хешируются во вторичную хеш-таблицу S_j размером m_j с использованием хеш-функции h_j, выбранной из класса универсальных хеш-функций H_{p,m_j}.

2.4 Определения признака завершения работы распределенных вычислений

Одним из способов решения проблемы генерации и хранения большого множества состояний РКА является хранение состояний на разных узлах. Кроме того, генерацию состояний можно также осуществлять параллельно, что может существенно снизить временные затраты на генерацию РКА с большим числом состояний. Для распределенных вычислений встает задача определения признака завершения вычислений. Для решения этой задачи существуют различные алгоритмы, некоторые из них рассмотрены ниже. При этом предполагается, что взаимодействия в системе в общем случае носят асинхронный характер. Распределенные вычисления полагаются законченными, если каждый процесс завершился и все коммуникационные каналы пусты.

В основе большинства алгоритмов, определяющих, завершились ли распределенные вычисления, лежит обмен сообщениями, содержащих данные о количестве отправленных и полученных в системе сообщений. С этой целью каждый процесс подсчет отправленных и полученных им сообщений (то есть имеется два соответствующих счетчика). При этом следует учитывать, что пересылка данных занимает некоторое время, и прежде чем отправленные данные будут получены процессом получателем, ситуация в системе может измениться.

Пусть $S(t) \coloneqq \sum_i s_i(t)$ - количество полученных в системе сообщений, а $R(t) \coloneqq \sum_i r_i(t)$ – количество полученных в системе сообщений, и пусть в системе имеется процесс инициатор, которому необходимо знать, завершены ли распределенные вычисления. Формально, система процессов P_i (где $i \in I$ – множеству индексов), завершена в момент времени t тогда и только тогда, тогда, когда $S(t) = R(t)$. Но, поскольку контрольные сообщения будут получены процессами P_i в различные моменты времени t_i, инициатор ошибочно сравнивает $S^* \coloneqq \sum_i s_i(t_i)$ с $R^* \coloneqq \sum_i r_i(t_i)$, а не значения $S(t)$ и $R(t)$.

Поэтому, необходимо выполнять синхронизацию часов, например, при помощи алгоритма Лампорта (Lamport), либо использовать другие средства для синхронизации при обмене сообщениями. На практике для решения этой проблемы используются несколько алгоритмов, которые подробно описаны в [4]. Ниже подробно рассмотрен наиболее простой и при этом достаточно эффективный алгоритм для определения признака завершения вычислений, и основанный на использовании четырех счетчиков (*four counter method*, см. [4]).

Алгоритм основан на повторной рассылке контрольных сообщений («контрольных волн»). После того, как процесс-инициатор получил ответ от последнего процесса в системе и увеличил значения счетчиков R^* и S^*, он вновь рассылает контрольные сообщения, сохраняя количество отправленных и полученных им сообщений в счетчиках S'^* и R'^*. Вычисления полагаются законченными, если $R^* = S^* = S'^* = R'^*$.

Фактически, если $R^* = S'^*$, то система завершила вычисления еще в конце первой контрольной волны. Для лучшего понимания этого факта рассмотрим *лемму*. Пусть t_2 - момент времени, в который завершилась первая контрольная волна, и $t_3 \geq t_2$ - момент времени, в который начинается вторая волна. Тогда:

1. Значения локальных счетчиков сообщений монотонны, $t \leq t' => s_i(t) \leq s_i(t')$ и $r_i(t) \leq r_i(t')$. Доказательство: следует из определения.
2. Суммарное число отправленных или полученных сообщений монотонно, $t \leq t' => S(t) \leq S(t')$ и $R(t) \leq R(t')$. Доказательство: следует из определения и из пункта 1.
3. $R^* \leq R(t_2)$. Доказательство: следует из пункта 1 и из того факта, что r_i вычисляется до ($\leq$) момента времени t_2.
4. $S'^* \geq S(t_3)$. Доказательство: следует из пункта 1 и из того факта, что все значения s_i вычисляются после ($\geq$) момента времени t_3.
5. Для всех моментов времени $t: R(t) \leq S(t)$. Доказательство: $D(t) \coloneqq S(t) - R(t)$ – количество передаваемых сообщений; $D(t) \geq 0$ – выполняется всегда (логически следует из количества действий).

Теперь докажем вышеприведенное утверждение:

$$R^* = S'^* => R(t_2) \geq S(t_3)$$
$$=> R(t_2) \geq S(t_2)$$
$$=> R(t_2) = S(t_3).$$

Таким образом, системе завершает работу в момент времени t_2.

Если система завершает работу еще до начала первой контрольной волны, это означает, что все сообщения получены, и, следовательно, значения счетчиков полученных и отправленных сообщений совпадают. Поэтому, завершение вычислений будет определено после его возникновения за два «раунда».

Необходимо заметить, что вторая контрольная волна в случае неудачного теста на завершенность вычислений может быть использована как первая волна для следующей попытки. Однако, проблема данного метода заключается в том, как определить в какой

момент времени инициировать вторую волну после неудачного теста на завершенность вычислений, поскольку это может привести к бесконечному контрольному циклу, в котором будут инициироваться контрольные волны. Для решения это проблемы существуют специальные методы (см. [4]).

Намного лучшей особенностью алгоритма является *симметричность* в случае, если процессы объединены в *виртуальную кольцевую топологию*. В этом случае, любой процесс может начать определение признака завершения посредством рассылки контрольного сообщения, циркулирующего по кольцу и накапливающего значения счетчиков; синхронизация или взаимное исключение в данном случае не требуются, и несколько контрольных волн могут быть активными одновременно. Кроме того, процессу нет необходимости знать общее число процессов в кольце, а достаточно знать только номер следующего процесса.

2.5 Способы хранения таблицы переходов конечных автоматов

При реализации конечных автоматов используются два основных формата (способа) хранения состояний и переходов (точнее, таблицы переходов, или, как еще говорят, матрицы инцидентностей)[6]:

- формат хранения таблицы переходов по столбцам (*column-wise format*) - сохраняются состояния и входящие в них дуги переходов;
- формат хранения таблицы переходов по строкам (*row-wise format*) - сохраняются состояния и исходящие из них дуги переходов.

Пример. Пусть имеется следующий конечный автомат:

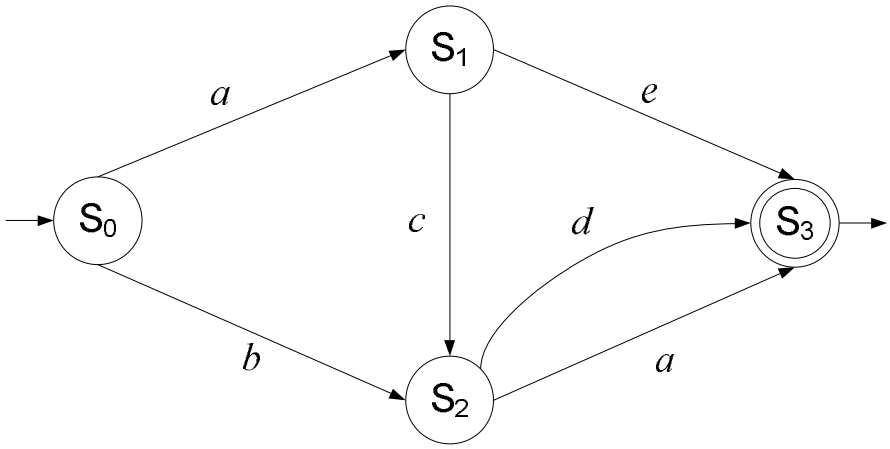


Рисунок 2-14. Пример конечного автомата

Данный конечный автомат можно описать следующей системой команд:

$$S_0 a \to S_1$$
$$S_0 b \to S_2$$
$$S_1 c \to S_2$$
$$S_1 e \to S_3$$
$$S_2 a \to S_3$$
$$S_2 d \to S_3$$

Тогда таблица (матрица) T_{column} переходов при хранении по столбцам будет выглядеть следующим образом:

	S_0	S_1	S_2	S_3
S_0	$\{\emptyset\}$	$\{\emptyset\}$	$\{\emptyset\}$	$\{\emptyset\}$
S_1	$\{a\}$	$\{\emptyset\}$	$\{\emptyset\}$	$\{\emptyset\}$
S_2	$\{b\}$	$\{c\}$	$\{\emptyset\}$	$\{\emptyset\}$
S_3	$\{\emptyset\}$	$\{e\}$	$\{a, d\}$	$\{\emptyset\}$

Таблица 2-1. Таблица переходов конечного автомата при хранении по столбцам

В первом столбце указаны состояния, в которые входят дуги переходов (преемники), в первой строке – состояния, из которых исходят дуги переходов (предшественники).

Здесь и далее элементы таблицы (матрицы) переходов будем обозначать как $t[i, j]$, где i, j -- состояния конечного автомата, $i \in \{S_0, S_1, S_2, S_3\}$, $j \in \{S_0, S_1, S_2, S_3\}$.

Элемент таблицы переходов при хранении по столбцам представляет собой множество переходов, ведущих из состояния j в состояние i.

Поскольку в исходном конечном автомате нет переходов, входящих в состояние S_0, $t[S_0, j] = \{\emptyset\}$ $\forall j \in \{S_0, S_1, S_2, S_3\}$. Далее, так как в автомате есть переход с меткой "c" из состояния S_1 в состояние S_2, то $t[S_2, S_1] = \{c\}$. Кроме того, в автомате есть два перехода из состояния S_2 в состояние S_3 с метками "a" и "d" соответственно. Поэтому $t[S_3, S_2] = \{a, d\}$. И, наконец, поскольку в исходном конечном автомате нет переходов, исходящих из состояния S_3, то $t[i, S_3] = \{\emptyset\}$ $\forall i \in \{S_0, S_1, S_2, S_3\}$.

Таблица (матрица) T_{row} переходов при хранении по строкам будет выглядеть следующим образом:

	S_0	S_1	S_2	S_3
S_0	$\{\emptyset\}$	$\{a\}$	$\{b\}$	$\{\emptyset\}$
S_1	$\{\emptyset\}$	$\{\emptyset\}$	$\{c\}$	$\{e\}$
S_2	$\{\emptyset\}$	$\{\emptyset\}$	$\{\emptyset\}$	$\{a, d\}$
S_3	$\{\emptyset\}$	$\{\emptyset\}$	$\{\emptyset\}$	$\{\emptyset\}$

Таблица 2-2. Таблица переходов конечного автомата при хранении по строкам

В первом столбце указаны состояния, из которых исходят дуги переходов (предшественники), в первой строке – состояния, в которые ведут дуги переходов (преемники).

Элемент таблицы переходов при хранении по строкам представляет собой множество переходов, ведущих из состояния j в состояние i.

Поскольку в исходном конечном автомате нет переходов, исходящих из состояния S_3, $t[S_3, j] = \{\emptyset\}\ \forall\, j \in \{S_0, S_1, S_2, S_3\}$. Далее, так как в автомате есть переход с меткой "c" из состояния S_1 в состояние S_2, то $t[S_1, S_2] = \{c\}$. Кроме того, в автомате есть два перехода из состояния S_2 в состояние S_3 с метками "a" и "d" соответственно. Поэтому $t[S_2, S_3] = \{a, d\}$. И, наконец, поскольку в исходном конечном автомате нет переходов, исходящих из состояния S_3, то $t[i, S_3] = \{\emptyset\}\ \forall\, i \in \{S_0, S_1, S_2, S_3\}$. Наконец, поскольку в автомате нет ни одного перехода, ведущего в состояние S_0, то $t[i, S_0] = \{\emptyset\}\ \forall\, i \in \{S_0, S_1, S_2, S_3\}$.

Таким образом, $T_{column} = (T_{row})^T$.

Выбор формата хранения таблицы переходов конечного автомата зависит от конкретных задач и алгоритмов, используемых для их решения. Так, например, реализацию поиска в ширину либо поиска в глубину проще выполнить, используя формат хранения таблицы переходов по строкам. В этом случае, для начального состояния сразу известны все его преемники. С другой стороны, в рассмотренном примере начальное состояние S_0 является недостижимым, поскольку нет ни одного перехода, ведущего в это состояние. В этом случае, при хранении таблицы переходов по столбцам реализация поиска в ширину или в глубину весьма сложна, если возможна вообще.

Если же стоит, например, задача поиска всех возможных состояний, из которых достижимо заданное состояние, то предпочтительнее использовать формат хранения таблицы переходов по столбцам. В таком случае для заданного состояния сразу известны все его предшественники. В рассмотренном примере, из заключительного состояния S_3 нет переходов ни в одно состояние конечного автомата. Соответственно, хранение таблицы по строкам приводит к сложностям при решении задачи, а также может привести к снижению эффективности работы программы. Если же хранить таблицы переходов по

столбцам, то для состояния S_3 сразу известны его предшественники – состояния S_1 и S_2, для которых также сразу можно определить предшественника – состояние S_0.

2.6 Заключение и выводы по аналитическому разделу

В аналитическом разделе рассмотрены основы генерации РКА для проверки моделей. Примерами алгоритма генерации РКА являются алгоритмы, основанные на поиска в ширину и поиске в глубину. При этом число состояний РКА может быть достаточно велико, поэтому необходимо сохранять состояния на разных узлах вычислительной системы с разделенной памятью, а в рамках одного узла системы использовать методы снижения временных затрат на поиск состояний. Для этой цели были рассмотрены различные методы хеширования. Поскольку используется распределенное хранение состояний РКА, то возможна также и распределенная генерация. При этом возникает задача определения признака завершения распределенных вычислений, которую можно решить при помощи рассмотренного простой алгоритм, основанного на периодическом обмене процессов сообщениями, включающими данные о количестве полученных и отправленных в системе сообщений. Кроме того, рассмотрены основные форматы хранения таблицы переходов конечных автоматов.

3 Конструкторский раздел

Необходимо разработать алгоритм для распределенного хранения (на нескольких компьютерах) большого (не умещающегося в памяти одного компьютера) множества состояний и переходов расширенного конечного автомата на множестве бесконечных слов, предназначенного для проверки моделей систем (автомата Бюхи). Для решения этой задачи необходимо:

1) разработать основные структуры данных, составляющих конечный автомат;
2) выбрать способ организации множества состояния РКА;
3) выбрать формат хранения таблицы переходов РКА;
4) разработать алгоритм распределенной генерации РКА по заданному БКА.
5) выбрать топологию вычислительной системы.

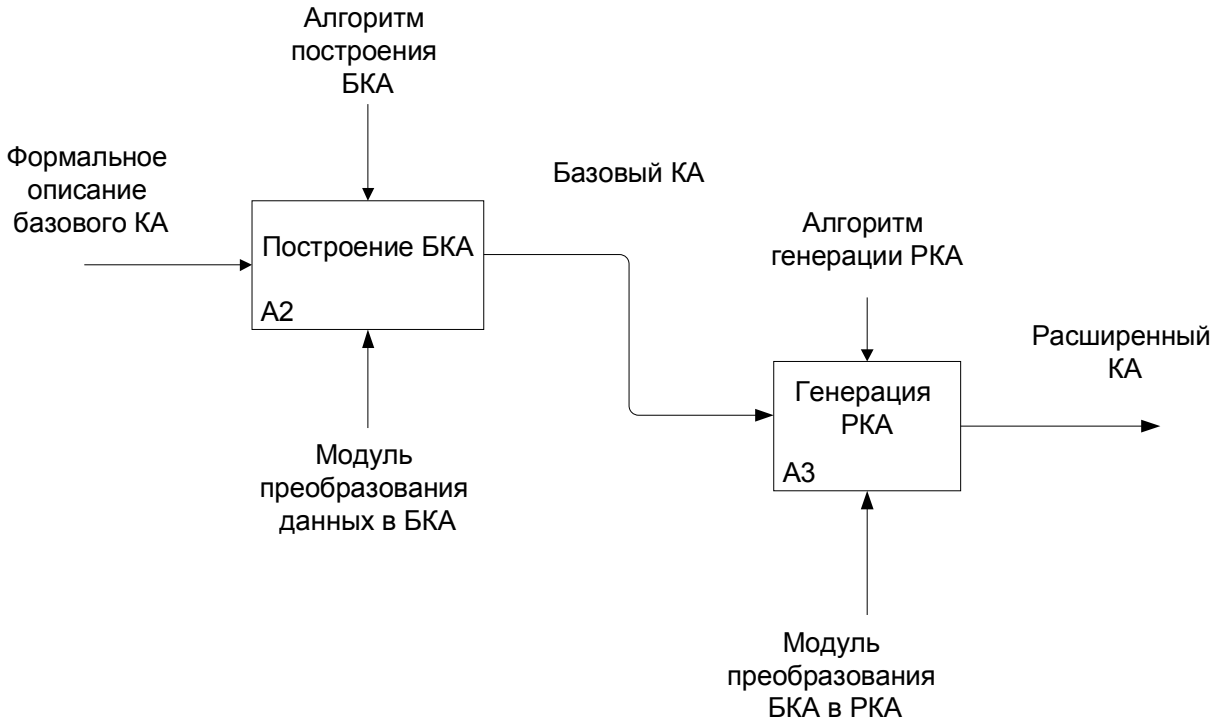


Рисунок 3-1. Процесс генерации расширенного конечного автомата и проверки утверждения

3.1 Входные и выходные данные алгоритма распределенной генерации расширенного конечного автомата

Входным данным для алгоритма распределенной генерации расширенного конечного автомата является базовый конечный автомат, который строится по формальному описанию модели. В дальнейшем часто будем обозначать базовый конечный автомат аббревиатурой БКА, а расширенный – аббревиатурой РКА.

На рис. 3-2 приведен пример конечного автомата с выходом, у которого:

- входной алфавит $V = \{x < 4, x == 4\}$;
- выходной алфавит $W = \{x = x + 1, x = 0\}$;
- множество состояний $Q = \{s_0, s_1\}$ при начальном состоянии s_0 и заключительном состоянии s_1.

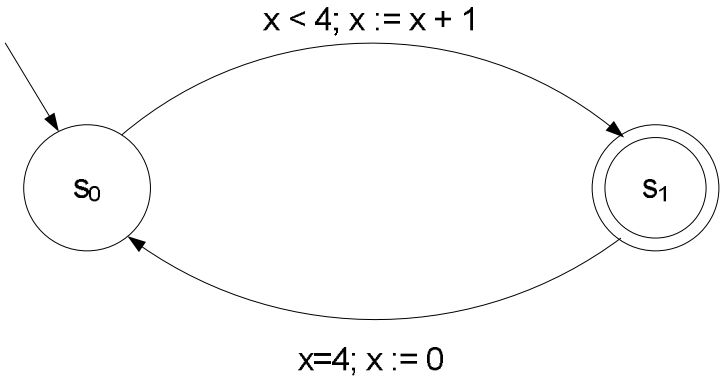


Рисунок 3-2. Пример конечного автомата с выходом

Выходными данными алгоритма распределенной генерации является РКА.

В разделе 2.1 подробно рассмотрены предикаты и действия, представляющие входной и выходной алфавит БКА. При реализации предикатов и действий, они задаются как строковые метки, которые в дальнейшем преобразуются в выражения, затем вычисляются значения выражений. Для обозначения пустых предикатов (*pempty*) и действий (*aempty*) используются строковые значения «*TRUE»* и *«NULL»* соответственно. Реализация списка переменных совпадает с описанным в разделе 2.1.

Вновь рассмотрим РКА, приведенный в качестве примера в разделе 2.1 (см. рис. 3-3) .

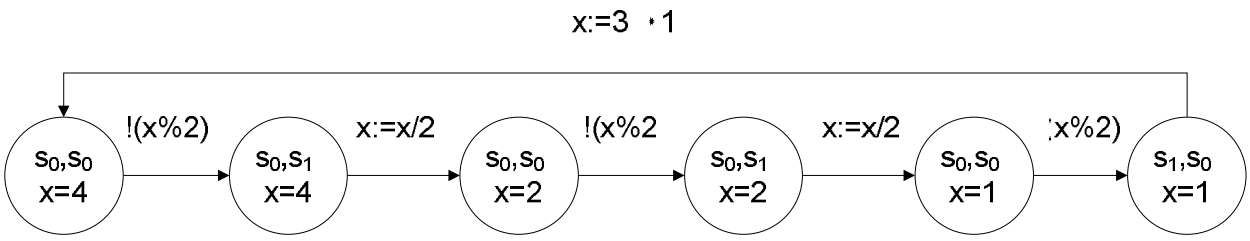


Рисунок 3-3. Пример расширенного конечного автомата

Тогда результат работы алгоритма упрощенно можно изобразить следующим образом (см. рис. 3-4):

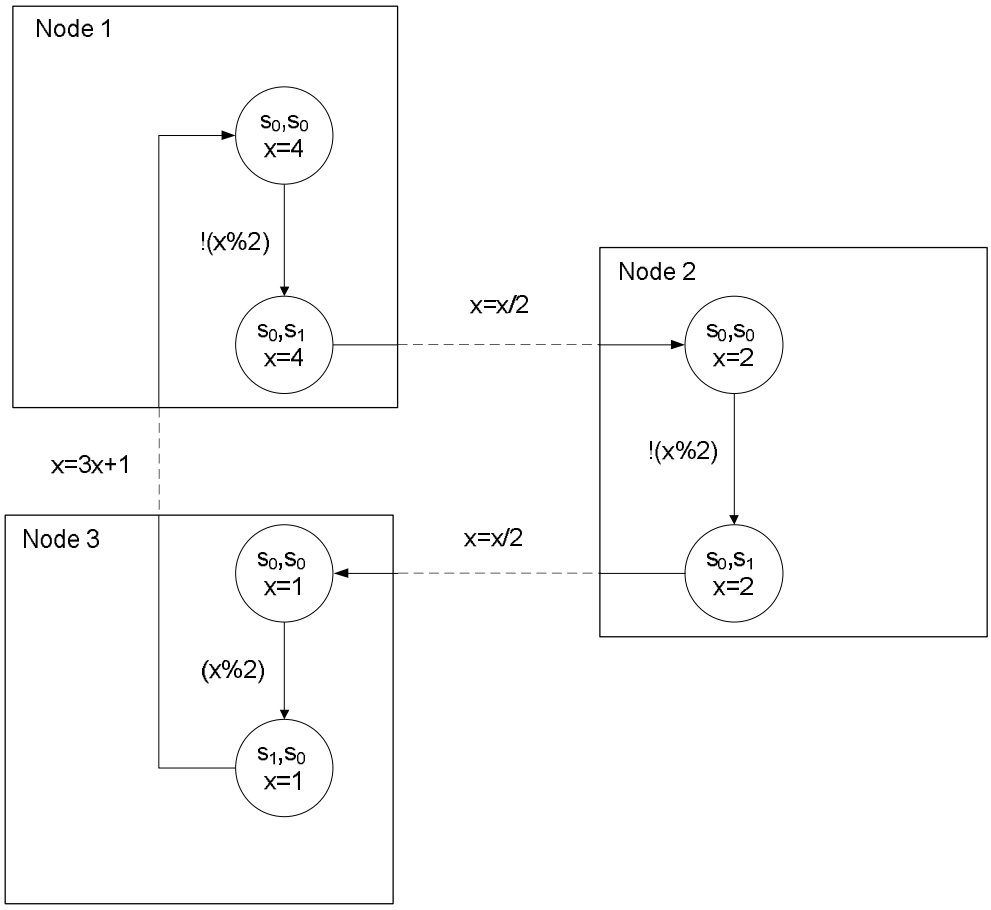


Рисунок 3-4. Упрощенная схема хранения РКА на нескольких узлах

3.2 Представление состояний и переходов КА, хеширование, выбор способа хеширования для хранения на одной машине

Прежде чем перейти к разработке и реализации распределенного алгоритма генерации и хранения РКА, необходимо разработать представление БКА и РКА на одной машине (одном узле кластера) с учетом следующих особенностей РКА: количество состояний РКА может быть очень велико, что приводит к необходимости решения проблемы эффективного поиска состояний во множестве состояний КА. Для этих целей используется хранение множество состояний РКА в виде хеш-таблицы. Отметим, что для БКА такой проблемы не возникает, поскольку построенный по формальному описанию модели БКА, как правило, обладает небольшим числом состояний. Поэтому для реализации множества состояний БКА используется список.

3.2.1 Описание основных структур данных

Прежде, чем рассмотреть основные структуры данных, используемые непосредственно для представления конечного автомата, рассмотрим ряд вспомогательных структур. Для графической иллюстрации структур данных использовались средства языка UML.

Индекс состояния (Index). Включает в себя (см. рис. 3-5):

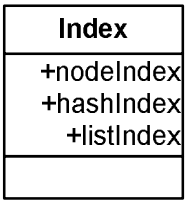


Рисунок 3-5. Структура индекса состояния

- Индекс состояния во множестве состояний (listIndex) – идентификатор, связанный с состоянием для облегчения поиска состояния во множестве состояний конечного автомата. Значение индекса устанавливается при добавлении состояния в конечный автомат.
- Индекс состояния в хеш-таблице (hashIndex) – используется для ускорения поиска состояния в хеш-таблице (для расширенного конечного автомата). Вычисляется при добавлении состояния во множество состояний расширенного конечного автомата, которое организовано в виде хеш – таблицы.
- Индекс узла (nodeIndex) – индекс узла, на котором хранится состояние. Используется при распределенной генерации расширенного конечного автомата.

Все вышеперечисленные индексы являются целыми числами. Индекс состояния, также как и метка состояния, является основным признаком, по которому производится поиск состояний во множестве состояний. При поиске состояний в БКА используется только индекс состояния во множестве состояний. При поиске состояний в РКА в рамках одного узла используется также индекс состояния в хеш-таблице. Наконец, индекс узла используется при распределенной генерации расширенного конечного автомата, а также при поиске состояний, находящихся на разных узлах, либо при обходе расширенного конечного автомата.

Переменная (Variable). Переменная включает (см. рис. 3-6):

- имя переменной (varName);
- значение переменной (varValue).

Имя переменной имеет строковый тип. Для генерации РКА достаточно рассмотреть значение переменных какого-либо типа, например целого или вещественного. При необходимости модификации с целью использования также и других типов для значений переменных основная логика не изменится, изменятся лишь диапазон допустимых значений и операции.

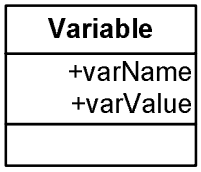


Рисунок 3-6. Структура переменной

В данной работе значение переменной – целое число, соответственно диапазон операций ограничен операциями, допустимыми для целочисленных переменных.

Список переменных. Включает в себя (см. рис. 3-7):

- собственно список переменных (variables);
- количество переменных в списке (varCount).

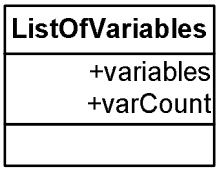


Рисунок 3-7. Структура списка переменных

Список переменных реализован в виде списка, поскольку количество переменных, как правило, невелико и поиск нужной переменной в списке не требует больших временных затрат.

Теперь рассмотрим основные структуры данных, которые используются собственно для реализации конечного автомата. Отметим, что для реализации БКА и РКА используются одни и те же структуры данных.

Переход. Включает в себя (см. рис. 3-8):

- предикат (символ входного алфавита);
- действие (символ выходного алфавита);
- индекс состояния, из которого исходит дуга;
- индекс состояния, в которое ведет дуга;

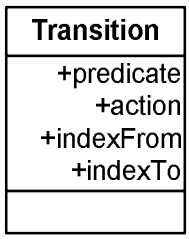


Рисунок 3-8. Структура перехода

Переход конечного автомата содержит предикат (символ входного алфавита) и действие (символ выходного алгоритма). Для того чтобы осуществить переход из одного состояния в другое, необходимо, чтобы выполнялся предикат. Для проверки предиката либо выполнения действия в соответствующий обработчик передается список переменных. Если предикат выполняется, то осуществляется действие.

Состояние. Состояние включает (см. рис. 3-9):

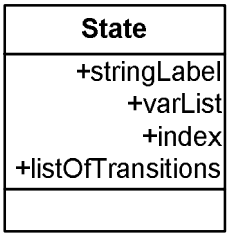


Рисунок 3-9. Структура состояния

- метку состояния (строковое значение);
- список переменных и их значений, соответствующих данному состоянию;
- индекс состояния;
- список переходов – список входящих / исходящих из состояния переходов (в зависимости от выбранного формата хранения таблицы переходов).

Метка состояния (строковая) – одна из основных характеристик, по которым производится поиск состояний во множестве состояний БКА. Простейшими примерами строковых меток состояний являются: "s_0", "s_2" и т.д. При поиске состояний в РКА также учитываются значения переменных в списке переменных, соответствующем данному состоянию. Другой важной характеристикой при поиске состояния во множестве состояний БКА или РКА является его индекс. Именно по индексам, как правило, производится поиск состояний при генерации РКА.

Множество состояний (Set of states). Включает в себя (см. рис. 3-10):

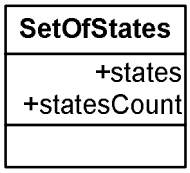


Рисунок 3-10. Структура множества состояний

Как уже отмечалось выше, способ организации множества состояний (его структура) зависит от зависит от решаемых задач. Так, для базового конечного автомата наиболее подходящей структурой является *список (list)* состояний.

Хранение множества состояний расширенного конечного автомата, число состояний которого может быть очень велико, в виде списка нерационально, поскольку в худшем случае для поиска состояния в списке потребуется полный перебор всех состояний, что значительно снижает производительность системы. Поэтому множество состояний расширенного автомата целесообразно хранить в виде *хеш-таблицы (hash table)*. Главное требование, которому должно удовлетворять множество состояний независимо от организации - обеспечивать удобные добавление и быстрый поиск элементов (состояний) по различным признакам (индекс, метка).

Конечный автомат (Finite state machine, FSM).

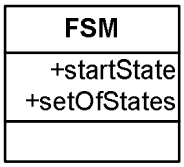


Рисунок 3-11. Структура конечного автомата

Конечный автомат состоит из (см. рис. 3-11):

- множества состояний (setOfStates) - списка в случае базового конечного автомата и хеш-таблицы в случае расширенного конечного автомата;
- начального состояния (startState).

Конечный автомат можно представить в виде множества состояний. Способ конкретной реализации множества (список или хеш-таблица) состояний зависит от решаемых задач. Отметим также, что таблица переходов хранится неявно: конечный автомат не содержит таблицы переходов, но каждое состояние включает список связанных с ним переходов (входящих или исходящих в зависимости от формата хранения таблицы – по строкам или по столбцам).

В заключение приводится полная схема взаимодействия вышеописанных структур данных (см. рис. 3-12).

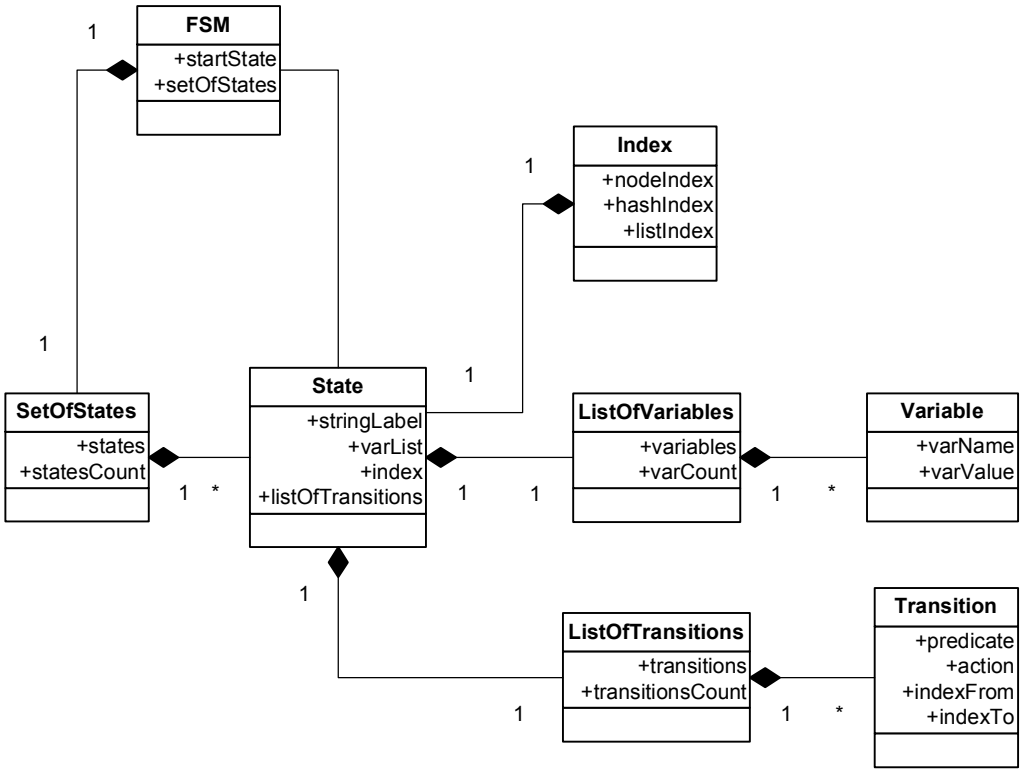


Рисунок 3-12. Схема взаимодействия структур данных

3.2.2 Хеширование, выбор хеш-функции для хранения РКА на одной машине, выбор способа разрешения коллизий

Как уже отмечалось, одной их основных проблем при генерации расширенного конечного автомата с большим числом состояний является поиск в пространстве состояний уже сгенерированных состояний с.

При генерации состояний и переходов в РКА необходимо производить поиск состояний во множестве состояний РКА. целью избежать дублирования состояний и обнаружения циклов. Последнее условие является ключевым – в противном случае алгоритм генерации будет работать бесконечно. Однако, множество состояний РКА может быть очень большим, что приводит к необходимости использования хеширования для снижения временных затрат на поиск состояний. Различные способы хеширования были рассмотрены в разделе 2.3. Выбор способа хеширования сводится к:

1) выбору способа разрешения коллизий;
2) выбору хеш-функции.

Как отмечалось выше, множество состояний РКА хранится в виде хеш-таблицы. Для разрешения коллизий используется ***метод цепочек***, т.к. необходимо сохранить все состояния, а заранее невозможно сказать, каков будет размер множества состояний РКА (что не позволяет использовать для разрешения коллизий методы открытой адресации). Выбор хеш-функции следует осуществлять исходя из следующих требований:

1) хеш-функция должна быть равномерной (что позволяет минимизировать количество коллизий и, следовательно, сократить временные затраты на поиск состояний);
2) вычисление хеш-функции должно осуществляться быстро.

Ключом хеш-функции в данном случае является состояние РКА. При вычислении значения хеш-функции состояния учитываются:

1) метка состояния;
2) значения переменных, соответствующие данному состоянию;
3) размер хеш-таблицы.

Основной проблемой является выбор хеш-функции. Подобрать хеш-функцию, дающую равномерное распределения для всех возможных значений переменных и меток состояний, достаточно трудно. Опытным путем была подобрана хеш-функция, дающая относительно равномерное распределение состояний по ячейкам в хеш-таблице для

большинства тестовых случаев. Значение хеш-функции зависит от кодов символов метки состояния и значений переменных в списке в данном состоянии.

3.3 Алгоритм GeneratorCollectors генерации расширенного конечного автомата

Алгоритм ***GeneratorCollectors*** основан на MIMD – модели, т.е. процессы выполняют различный код. В процессе выполнения все процессы обмениваются сообщениями 2-х различных типов:

- Arc – используется при передаче состояний и переходов от генератора к коллекторам;
- $Indx$ - используется для передачи индексов состояний в РКА от коллекторов к генератору;
- Trm - используется для определения признака глобального завершения распределенных вычислений.

Основная идея алгоритма заключается в следующем. Пусть для генерации РКА используется N процессов, выполняющихся на N узлах. Один из этих процессов является процессом-генератором (*generator*), а остальные – процессами-коллекторами (collectors), или хранителями.

Узлы, на которых выполняются процессы-коллекторы, осуществляют единственную функцию–хранение состояний РКА.

Процесс-генератор генерирует новые состояния. Для каждого нового сгенерированного состояния s вычисляется хеш-функция $Partition$(s, N), определяющая номер узла процесса-коллектора, на котором должно быть сохранено состояние s в зависимости от метки состояния, списка переменных и числа коллекторов, после чего состояние s и ведущий в него переход пересылаются на узел процесса-коллектора (при этом коллектору отсылается сообщения типа Arc) . После этого процесс-генератор блокируется в ожидании от процесса-коллектора индекса состояния s в РКА, поскольку индекс состояния s в РКА необходим для исследования состояния s.

В свою очередь, процесс-коллектор добавляет состояние s и в РКА и отправляет процессу-генератору индекс состояния s в РКА (генераторы отсылается сообщения типа $Indx$).

Процесс-генератор получает от процесса-коллектора индекс состояния s и начинает исследовать это состояние (то есть генерировать новые состояния, достижимые

из состояния s). Таким образом, при исследовании состояния s процесс-генератор взаимодействует только с одним процессом-коллектором. Узел, на котором выполняется процесс-генератор, не хранит состояний РКА.

Завершив вычисления, процесс-генератор рассылает всем процессам коллекторам сообщение Trm, сигнализируя, что вычисления закончены. Получив сообщение Trm, коллектор завершает работу.

Псевдокод алгоритма ***GeneratorCollectors*** приводится ниже. Процесс-генератор вызывает процедуру ***Generator***, принимающую следующие параметры:

1) $baseFsm$ – базовый конечный автомат;
2) i – номер процесса-генератора;
3) N – количество коллекторов, осуществляющих хранение РКА.

Узел-генератор не хранит состояний РКА, поэтому процедура ***Generator*** не принимает РКА в качестве параметра.

```
procedure Generator(baseFsm, i, N)
begin
    queue := ∅;
    s := CreateStartState(baseFsm.startState);
    push(queue, s);
    while nonempty( queue) then
        s := front(queue); pop(queue);
        for each t in EnabledTranstions(s, baseFsm) do
            state := CreateState(s, t, basefsm);
            j := Partition(N);
            t.indexFrom := s.index;
            SendArc(j, state, t);
            RecvIndex(j, sIndex);
            state.index = sIndex;
            push(queue, state);
        endfor;
    endwhile;
end.
```

Процессы-коллекторы вызывают процедуру ***Collector***, принимающую следующие параметры:

1) $baseFsm$ – базовый конечный автомат;
2) $expandedFsm$ – расширенный конечный автомат (результат выполнения процедуры);
3) i – номер текущего процесса-коллектора;
4) $generator$ – номер процесса-генератора.

procedure Collector($baseFsm, expandedFsm, i, generator$)
begin
    $expandedFsm.states \coloneqq \emptyset$;
    $terminated_i \coloneqq$ **false**;
    while not $terminated_i$ **do**
        if $Receive\ (m)$ **then**
            if $m = Arc(s,t)$ **then**
                if not $found(s, expandedFsm.states)$ **then**
                    $AddState(expandedFsm, s)$;
                endif;
                $t.indexTo \coloneqq s.index$;
                $AddTransition(expandedFsm, t.indexFrom, t.indexTo)$;
                $SendIndex(generator, s.index)$;
            elseif $m = Trm$ **then**
                $terminated_i \coloneqq$ **true**;
            endif;
        endif;
    endwhile;
end.

В алгоритме ***GeneratorCollectors*** формат хранения таблицы переходов РКА не существен. При реализации алгоритма использовался формат хранения таблицы переходов РКА по столбцам.

Взаимодействие процесса-генератора и одного из процессов-коллекторов при обмене сообщениями схематически изображено на рис. 3-13.

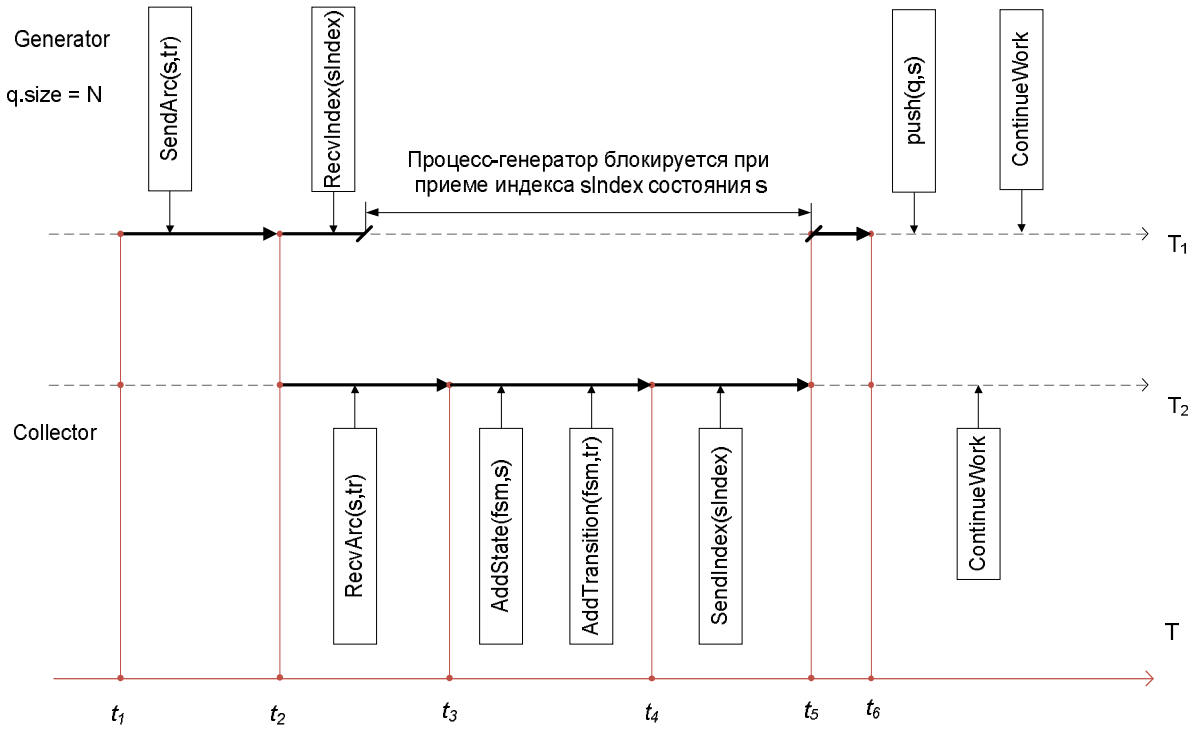


Рисунок 3-13. Взаимодействие процесса-генератора и процесса-коллектора

Поскольку процесс-генератор взаимодействует всего с одним процессом-коллектором, в вызовах $SendArc$, $RecvArc$, $SendIndex$, $RecvIndex$ не указаны номера процессов-отправителей и процессов-получателей.

Рассмотрим процесс взаимодействия подробнее. Процесс-генератор посылает процессу-коллектору состояние s и ведущий в него переход tr посредством вызова $SendArc$ (при этом процессу-коллектору отправляется сообщение типа Arc). После этого процесс-генератор инициирует вызов

3.3.1 Схема взаимодействия генератора и коллекторов

Пусть в системе имеется N узлов с номерами от 1 до N, узлы с номерами от 1 до $N-$ являются узлами-коллекторами, а узел с номером N - узлом-генератором. Генератор в процессе вычислений поочередно извлекает состояния из очереди неисследованных состояний. Для состояния s_0 значение хеш-функции разбиения $Partition(s_i) = i$. Поэтому процесс-генератор пересылает посредством вызова $SendArc(C_i, s_2, t_2)$ состояние s_0 и ведущий в него переход t_2 на узел с номером 1, на котором выполняется процесс-коллектор C_i . Коллектор помещает состояние s_2 во множество состояний РКА, а переход t_2 – в таблицу переходов соответственно. После помещения состояния s_2 во множество состояний РКА, коллектор отсылает индекс $indexS_2$ состояния s_2 в РКА процессу-генератору G_N посредством вызова

$SendIndex(G_N, indexS_2)$. Получив индекс $indexS_2$ состояния s_2 в РКА, процесс-генератор G_N продолжает исследовать состояние s_2.

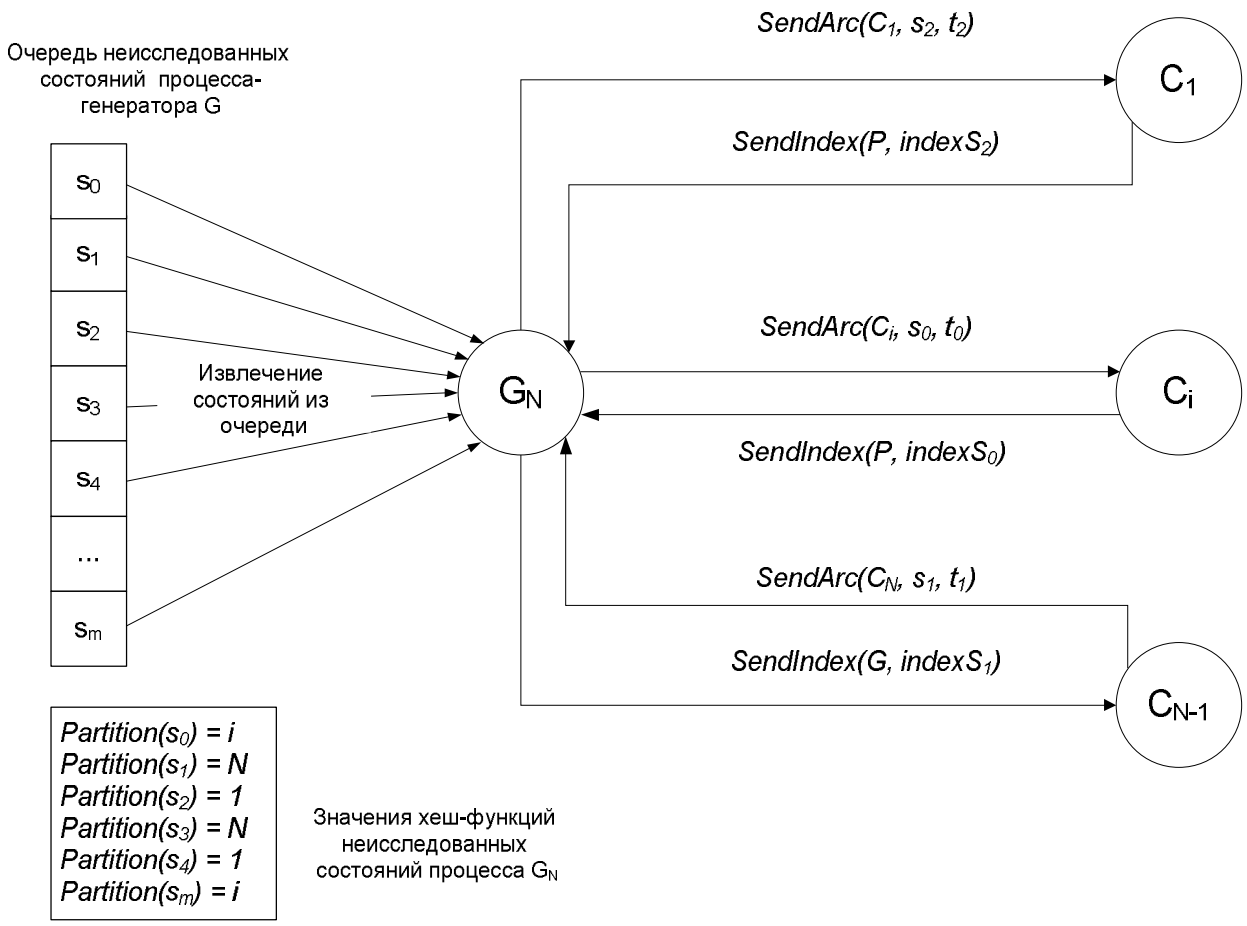


Рисунок 3-14. Схема взаимодействия генератора и коллекторов

3.4 Алгоритм Distributor распределенной генерации расширенного конечного автомата

Алгоритм ***Distributor*** распределенной генерации основан на SMPD – модели, т.е. все процессы выполняют идентичный код. В процессе выполнения все процессы обмениваются сообщениями 4-х различных типов:

- Arc – используется при передаче состояний и переходов;
- Snd - используется для передачи количества сообщений, отправленных процессом;
- Rec - используется для передачи количества сообщений, полученных процессом;
- Trm - используется для определения признака глобального завершения распределенных вычислений;

Сообщения типа Arc используются для генерации и хранения состояний на различных узлах, сообщения остальных типов используются для определения признака завершения распределенных вычислений.

Псевдокод алгоритма выглядит следующим образом. Процедура ***Distributor*** принимает на входе следующие параметры:

1) $baseFsm$ – базовый конечный автомат;
2) $expandedFsm$ – расширенный конечный автомат (результат выполнения процедуры);
3) i – номер текущего процесса;
4) N – количество процессов, осуществляющих генерацию.

procedure Distributor$(baseFsm, expandedFsm, i, N)$
begin
$queue \coloneqq \emptyset$;
$expandedFsm.states \coloneqq \emptyset$;
$s \coloneqq CreateStartState(baseFsm.startState)$;
$initiator_i \coloneqq (Partition(s) = i)$;
if $initiator_i$ **then** $terminit \coloneqq$ **false endif**;
$terminated_i \coloneqq$ **false**;
$nbsent_i \coloneqq 0; nbrecd_i \coloneqq 0;\ k \coloneqq 0; nextRank \coloneqq (i + 1)$ **mod** N;
$push(queue, s)$;
while not $terminated_i$ **do**
if $nonempty(queue)$ **then**
$s \coloneqq front(queue);\ pop(queue)$;
for each t **in** $EnabledTranstions(s, baseFsm)$ **do**
$state \coloneqq CreateState(s, t, basefsm)$;
$j \coloneqq Partition(state)$;
if not $j = i$ **then**
$SendArc(j, state, t)$;
$nbsent_i \coloneqq nbsent_i + 1$;
else
if not $found(state, expandedFsm.states)$ **then**
$push(queue, state)$;
$AddState(expandedFsm, state)$;

```
                        endif;
                        t.indexFrom := s.index; t.indexTo := state.index;
                        AddTransition(expandedFsm, t, s.index, state.index);
                    endif;
                endfor;
            endif;
            if Receive (m) then
                case m is
                    Arc(s,t) →  push(queue, s); nbrecd_i := nbrecd_i + 1;
                    Rec(k) →    if not initiator_i then
                                    Send(nextRank, Rec(k + nbrecd_i));
                                elseif terminit then
                                    totalrecd := k;
                                    Send(nextRank, Snd(nbsent_i));
                                endif;
                    Snd(k) →    if not initiator_i then
                                    Send(nextRank, Snd(k + nbsent_i));
                                elseif terminit and totalrecd = k then
                                    Send(nextRank, Trm);
                                else
                                    terminit := false;
                                endif;
                    Trm →       if not initiator_i then Send(nextRank, Trm);
                                endif;
                                terminated_i := true;
                endcase;
            endif;
            if empty(queue) and initiator_i and not terminit then
                terminit := true; Send(nextRank, Rec(nbrecd_i));
            endif;
        endwhile;
end.
```

Рассмотрим алгоритм более подробно. Для того чтобы определить, на каком узле из N узлов должно быть сохранено и исследовано состояние s, необходимо вычислить функцию разбиения $Partition(s, N)$ состояния. Эта функция в зависимости от метки состояния и значений переменных, соответствующих данному состоянию, а также количества доступных узлов, возвращает номер узла, на котором сохранено и исследовано состояние. Таким образом, функция $Partition(s, N)$ является хеш-функцией, следовательно, основные требования к ней аналогичны требованиям к хеш-функциям, используемым при вычислении хеш-кода состояний для хранения на одном узле – равномерность и малые временные затраты на вычисление. Заметим, что в данном случае (как и в алгоритме ***GeneratorCollectors***) не нужно разрешать коллизии, а требование равномерности обусловлено необходимостью обеспечить равномерную нагрузку на процессоры.

Другой особенностью является то, что при генерации нет необходимости осуществлять поиск состояний по всем узлам. После того, как состояние s получено узлом P_i, номер i которого соответствует значению функции $Partition$ для данного состояния, в дальнейшем все действия с этим состоянием (в том числе и поиск) производятся на этом узле. Таким образом, использование в качестве функции $Partition$ хеш-функции значительно упрощает распределение состояний по узлам и их поиск.

Помимо вышеперечисленных особенностей алгоритма, следует разрешить ряд проблем, основной из которых является задача определения глобального признака завершения вычислений. Эти проблемы и способы их решения рассмотрены ниже.

3.4.1 Выбор формата хранения матрицы переходов расширенного конечного автомата

В данной работе для таблицы переходов БКА используется формат хранения по строкам, поскольку для генерации РКА по БКА используется поиск в ширину, который при таком подходе гораздо проще осуществлять, так как для каждого состояния БКА сразу известны все состояния-приемники. При использовании последовательного алгоритма генерации, для генерации и хранения РКА на одном узле также можно использовать формат хранения таблицы переходов по строкам.

В случае же распределенной генерации РКА, хранящегося на нескольких узлах, используется формат хранения таблицы переходов по столбцам по ряду причин. Во-первых, при проверке модели использование формата хранения по столбцам позволяет сразу определить путь, ведущий в состояние РКА, в котором выполняется (или не

выполняется) заданное условие. Во-вторых, использование формата хранения по столбцам имеет ряд преимуществ при выполнении распределенной генерации состояний, тогда как использование формата хранения по строкам, наоборот, приводит к возникновению ряда сложностей при реализации алгоритма.

Однако при реализации распределенного (параллельного) алгоритма генерации РКА использование формата хранения таблицы переходов РКА по строкам может привести к дополнительным сложностям при реализации.

Рассмотрим псевдокод алгоритма ***Distributor*** распределенной генерации конечного автомата (как отмечалось, все процессы, выполняют этот код). Пусть таблица переходов РКА хранится по строкам. Предположим, что узел P_i, индекс которого равен i, отправляет узлу P_j состояние s_{new} такое что $Partition(s_{new}) = j$ (то есть состояние s_{new} должно быть исследовано и сохранено на узле с индексом j) посредством вызова некоторого вызова $SendState(j, s_{new})$. Пусть s_{new} является преемником некоторого состояния s, из которого осуществляется переход t в состояние s_{new}. Чтобы добавить переход в таблицу переходов РКА, узлу P_i необходимо знать локальный индекс k состояния s_{new} на узле P_j. Однако, к моменту вызова $SendState(j, s_{new})$ этот индекс неизвестен. Таким образом, для добавления перехода необходимо чтобы:

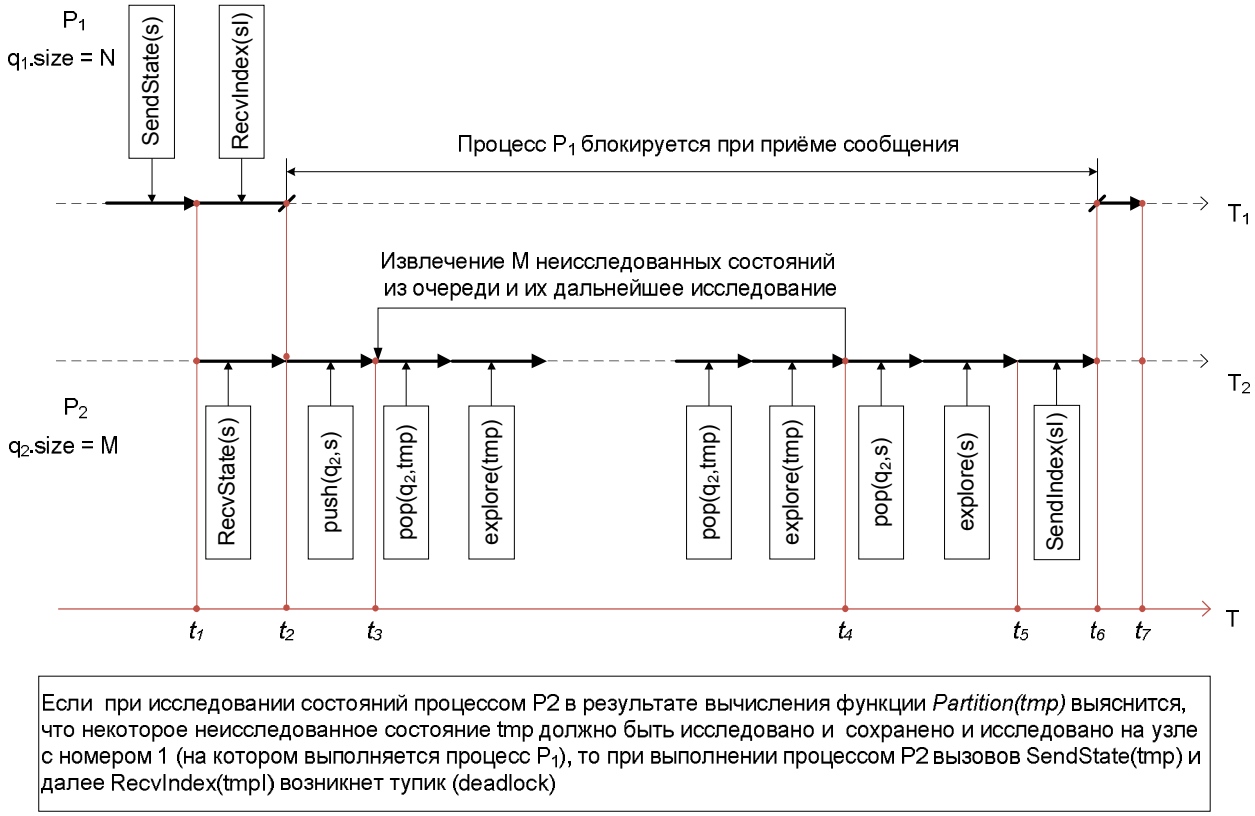
1) узел P_j, получив состояние s_{new} посредством некоторого вызова $ReceiveState(i, s_{new})$ и добавив его во множество состояний РКА, отправил полученный при добавлении локальный индекс k состояния s_{new} узлу P_i посредством некоторого вызова $SendIndex(i, k)$;
2) узел P_i, выполнив отправку сообщения посредством вызова $SendState(j, s_{new})$, осуществил вызов $ReceiveIndex(j, k)$ для получения локального индексы k состояния s_{new} на узле P_j.

Отметим также, что вызовы $SendState$ и $ReceiveState$ являются неблокирующими, поскольку к приему сообщений, включающих состояния каждый процесс готов всегда (то есть к процессу выстраивается очередь сообщений, которые далее извлекаются из нее), тогда как прием индексов следует инициировать только в случае, если состояние отправлено, и поэтому $SendIndex$ и $ReceiveIndex$ являются блокирующими. Таким образом, процесс на узле P_i будет заблокирован на вызове $ReceiveIndex$ до тех пор, пока не придет сообщение, включающее локальный индекс k состояния s_{new} на узле P_j.

Проблема заключается в том, что, даже получив состояние s_{new}, процесс, выполняемый на узле P_j, добавляет его в очередь неисследованных состояний. В результате, процесс узла P_i может быть заблокирован при выполнении вызова

$ReceiveIndex$ достаточно долгое время. При этом, если например процесс P_l отправил некоторое состояние процессу P_i, то он также будет заблокирован при вызове $ReceiveIndex(i,k)$, поскольку процесс P_i также заблокирован в этот момент. Таким образом, может возникнуть цепочка заблокированных процессов. Если же в эту цепочку попадет и процесс P_j, то может возникнуть тупик.

Упрощенная схема взаимодействия процессов P_1 и P_2 в процессе обмена сообщениями при формате хранения таблицы переходов по строкам изображена на рис. 3-15. Предполагается, что имеется всего два процесса, которые могут обмениваться сообщениями только друг с другом. Поэтому в вызовах типа $SendState(s)$ ($RecvState(s)$) не указывается номер процесса-получателя (процесса-отправителя) состояния s.



Если при исследовании состояний процессом P2 в результате вычисления функции *Partition(tmp)* выяснится, что некоторое неисследованное состояние tmp должно быть исследовано и сохранено и исследовано на узле с номером 1 (на котором выполняется процесс P_1), то при выполнении процессом P2 вызовов SendState(tmp) и далее RecvIndex(tmpI) возникнет тупик (deadlock)

Рисунок 3-15. Взаимодействие двух процессов при формате хранения таблицы переходов РКА по строкам

Рассмотрим взаимодействие процессов P_1 и P_2 подробнее. Время выполнения (или линия жизни) процессов P_1 и P_2 отображено на осях T_1 и T_2 соответственно. На глобальной временной оси T отмечены этапы $t_1, t_2, t_3, t_4, t_5, t_6$ и t_7 завершения выполнения операций (вызовов) процессами P_1 и P_2.

Процесс P_1:

1) осуществляет вызов $SendState(s)$, который завершается в момент времени t_1;
2) в момент времени t_1 инициирует вызов $RecvIndex(sI)$ для получения индекса состояния s на узле с номером 2, и блокируется в ожидании прихода соответствующего сообщения.

Процесс P_2:

1) для получения состояния s от процесса P_1 в момент времени t_1 инициирует вызов $RecvState(s)$;
2) в момент времени t_2 посредством вызова $push(q_2, s)$ помещает полученное состояние в очередь неисследованных состояний;
3) в интервале времени с t_3 по t_4 извлекает посредством вызова $pop(q_2, tmp)$ и исследует (то есть вычисляет всех приемников данного состояния tmp и при необходимости добавляет tmp во множество состояний РКА, а также добавляет переход, ведущий в состояние tmp, в таблицу переходов РКА) посредством вызова $explore(tmp)$ каждое из M неисследованных состояний, находившихся в очереди q_2 к моменту времени t_1;
4) в интервале времени с t_4 по t_5 извлекает посредством вызова $pop(q_2, s)$ и исследует посредством вызова $explore(s)$ состояние s, полученное от узла с номером 1;
5) в момент времени t_6 завершает отправку процессу P_1 индекса состояния s, полученного в результате исследования состояния s, посредством вызова $SendIndex(sI)$ и продолжает работу.

После отправки индекса sI состояния s процессом P_2, процесс P_1 выходит из состояния блокировки, завершает вызов $RecvIndex(sI)$ в момент времени t_7, добавляет в таблицу переходов очередной переход, ведущий в состояние s, и продолжает работу. Заметим, что если в процессе исследования процессом P_2 состояний из очереди неисследованных состояний q_2 выяснится, что очередное исследуемое состояние tmp должно быть исследовано и сохранено на узле с номером 1 (то есть $Partition(tmp) = 1$), то при попытке осуществления процессом P_2 вызова $SendState(tmp)$ с целью отправки

состояния tmp процессу P_1 и дальнейшего вызова $RecvIndex(tmpI)$ для получения индекса состояния tmp на узле с номером 1, процесс P_2 будет заблокирован, так как в этот момент процесс P_1 блокирован в ожидании сообщения от процесса P_2. Таким образом, возникает тупик (*deadlock*).

Конечно, можно сделать вызовы $SendIndex$ и $ReceiveIndex$ неблокирующими. Но в этом случае при отправке локального индекса k состояния s_{new} на узле P_j также необходимо отправлять локальный индекс состояния s (из которого было достигнуто состояние s_{new} узле P_i и который был отправлен на узел P_j вместе с состоянием s_{new}) Затем, на узле P_i необходимо заново производить поиск состояния s (по индексу) и добавлять соответствующий переход к списку переходов состояния s с учетом полученного индекса состояния s_{new} на узле P_j. Однако, этот путь сложнее в реализации и кроме того, потребует дополнительных временных затрат на поиск состояний.

Теперь предположим, что таблица переходов хранится по столбцам. В этом случае, процесс P_i отправляет процессу P_j состояние s_{new} и ведущий в него переход t, а также локальный индекс k состояния s (из которого достигнуто состояние s_{new} по переходу t) на узле P_i – он, разумеется, известен к моменту отправки состояния s_{new} на процессу P_j. Отправку состояния s_{new} и перехода t можно осуществить посредством всего одного вызова $SendArc(j, t, s_{new})$. Получив соответствующее сообщение, процесс P_j сначала добавляет во множество состояний РКА состояние s_{new} (получая в результате добавления локальный индекс состояния s_{new}), а затем переход t - поскольку к этому моменту уже известны индексы состояния, из которого исходит дуга перехода, и индекс состояния, в которое ведет дуга перехода, добавление перехода осуществляется достаточно просто.

Упрощенная схема взаимодействия процессов P_1 и P_2 в процессе обмена сообщениями при формате хранения таблицы переходов по столбцам изображена на рис. 3-16. Процесс P_1 отправляет дугу (состояние s_1 и ведущий в него переход tr_1) процессу P_2 посредством вызова $SendArc(s_1, tr_1)$, который завершается в момент времени t_2, и продолжает выполнять вычисления (исследовать текущие неисследованные состояния, при необходимости отправляя их процессу P_2, либо добавляя во множество состояний РКА). Процесс P_2 получает дугу посредством инициирования в момент времени t_2 вызова $RecvArc(s_1, tr_1)$, далее полученное состояние s_2 помещается в очередь неисследованных состояний, и процесс P_2 продолжает выполнение. Аналогичные действия выполняются при отправке процессом P_2 дуги (s_2, tr_2) процессу P_1. Отметим, что при таком подходе не используются блокирующие вызовы, и следовательно, не возникает тупиковых ситуаций.

Рисунок 3-16. Взаимодействие процессов при формате хранения таблицы переходов по столбцам

Таким образом, в данном случае использование формата хранения таблицы переходов по столбцам позволяет разрешить некоторые проблемы, возникающие при распределенной генерации РКА.

3.4.2 Схема взаимодействия узлов при передаче состояний и переходов

В процессе генерации состояний РКА процессы обмениваются сообщениями 4-х типов. Для пересылки состояний и переходов используется сообщения типа Arc. Отметим некоторые особенности касающиеся пересылки состояний и переходов.

Во-первых, для уменьшения количества вызовов передачи и приема ($Send$, $Receive$) состояние s и ведущий в него переход t передаются вместе, посредством одного вызова $SendArc(i, s, t)$, где i – номер узла-получателя. Это возможно в связи с использованием формата хранения по столбцам, т.к. в этом случае к моменту пересылки состояния s_i уже известно из какого состояния s_j и по какому переходу t_{ij} было достигнуто данное состояние s_i.

Во-вторых, схема взаимодействия процессов при пересылке состояний зависит от вида выбранной хеш-функции $Partition$, которая в зависимости от метки состояния s_i (строковой метки и соответствующих значений переменных для РКА) и количества узлов,

осуществляющих генерацию, вычисляет индекс k узла, на котором должно быть исследовано и сохранено данное состояние s_i. После этого состояние и ведущий в него переход t пересылаются на Таким образом, при пересылке сообщений процессы не объединяются в виртуальную топологию.

Схематически взаимодействие процессов при отправке сообщений изображено на рис. 3-17. На рисунке процесс P_j поочередно извлекает состояния из полученной в процессе вычислений очереди неисследованных состояний. Для каждого состояния вычисляется значение хеш-функции $Partition(s)$. Например, значение $Partition(s_{0j}) = j$, поэтому состояние s_{0j} исследуется процессом P_j. Значение $Partition(s_{1j}) = N$, поэтому состояние s_{1j} пересылается на узел N, где помещается в очередь неисследованных состояний и будет в дальнейшем извлечено из очереди и исследовано в процессе выполнения вычислений процессом P_N.

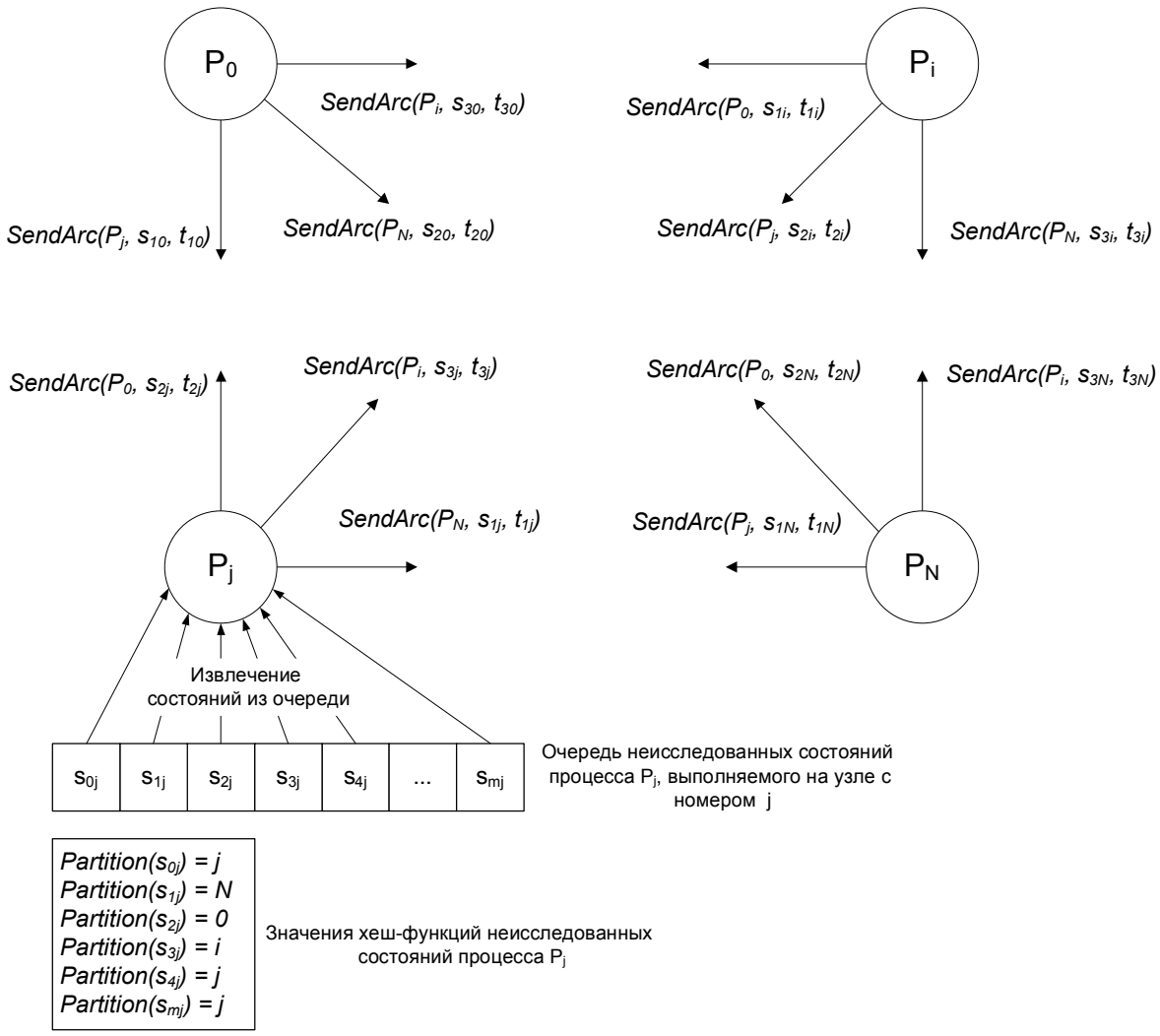


Рисунок 3-17. Схема взаимодействия процессов при пересылке состояний

3.4.3 Схема взаимодействия узлов при определении признака завершения параллельных вычислений

Одной из основных задач, которые приходится решать при реализации параллельных алгоритмов, является задача определения признака глобального окончания распределенных вычислений. Согласно основному определению, окончание работы достигается, когда все локальные вычисления закончены (т.е. каждая машина *i* не имеет состояний, которые необходимо исследовать), и все каналы передачи сообщений пусты.

Для решения этой задачи существуют различные алгоритмы, большинства которых основаны на подсчете количества отправленных и полученных каждым процессом во время выполнения вычислений сообщений и последующей рассылкой этой информации все остальным процессам.

В данной работе для определения окончания распределенных вычислений используется алгоритм, основанный на периодической рассылке сообщений о числе полученных и отправленных в системе сообщений. Такие алгоритмы просты в реализации, надежны, а также не требуют выполнения сложных вычислений.

Рассмотрим алгоритм более подробно. Пусть имеется N узлов, и, соответственно, N исполняемых процессов – по одному на каждый узел. Все узлы полагаются объединенными в виртуальное однонаправленное кольцо, которое соединяет каждый узел i с его узлом - приемником $(i+1)\ mod\ N$. В процессе генерации узлы обмениваются сообщениями, включающими данные, необходимые для выполнения вычислений (в данном случае, это состояния и переходы), а также специальными сообщениями, необходимыми для определения признака окончания распределенных вычислений.

Каждый раз, когда узел-инициатор заканчивает свои локальные вычисления, он проверяет, не достигнуто ли глобальное окончание вычислений, путем генерации двух последовательных сигналов, включающих специальные сообщения *Rec* и *Snd* в виртуальном кольце для получения количества сообщений, полученных и отправленных всеми машинами, соответственно. Сообщение $Rec(k)$ ($Snd(k)$), полученное узлом i, означает, что k сообщений было получено (отправлено) всеми машинами, начиная с машины инициатора и вплоть до машины $(i-1)\ mod\ N$. Каждая машина i *подсчитывает* количество полученных и отправленных ею сообщений, необходимых для выполнения вычислений, используя две переменных счетчика $nbrecd_i$ и $nbsent_i$, и прибавляет их к значениям, полученным в сообщениях Rec и Snd. При получении сообщения $Snd(k)$, которым заканчивался второй сигнал, узел-инициатор проверяет, равно ли общее число отправленных сообщений k общему числу полученных сообщений $totalrecd$ (результат

сигнала Rec). Если вышеописанное условие выполняется, узел-инициатор оповещает остальные узлы о том, что конец достигнут, при помощи отправки сообщения Trm по кольцу. В противном случае, инициатор заключает, что окончание работы еще не достигнуто, и позднее сгенерирует сигнал для определения окончания работы повторно.

На рис. 3-18 изображена схема взаимодействия (топология) узлов, объединенных в виртуальное однонаправленное кольцо. Каждый процесс i посылает сообщения m_{i+1} только следующему процессу в кольце и принимает сообщения m_i только от предыдущего процесса в кольце, и при этом $m_i \in \{Snd, Rec, Trm\}$ $\forall i: 0 \leq i \leq N$. (То есть обмен состояниями и переходами не производится в рамках этого алгоритма).

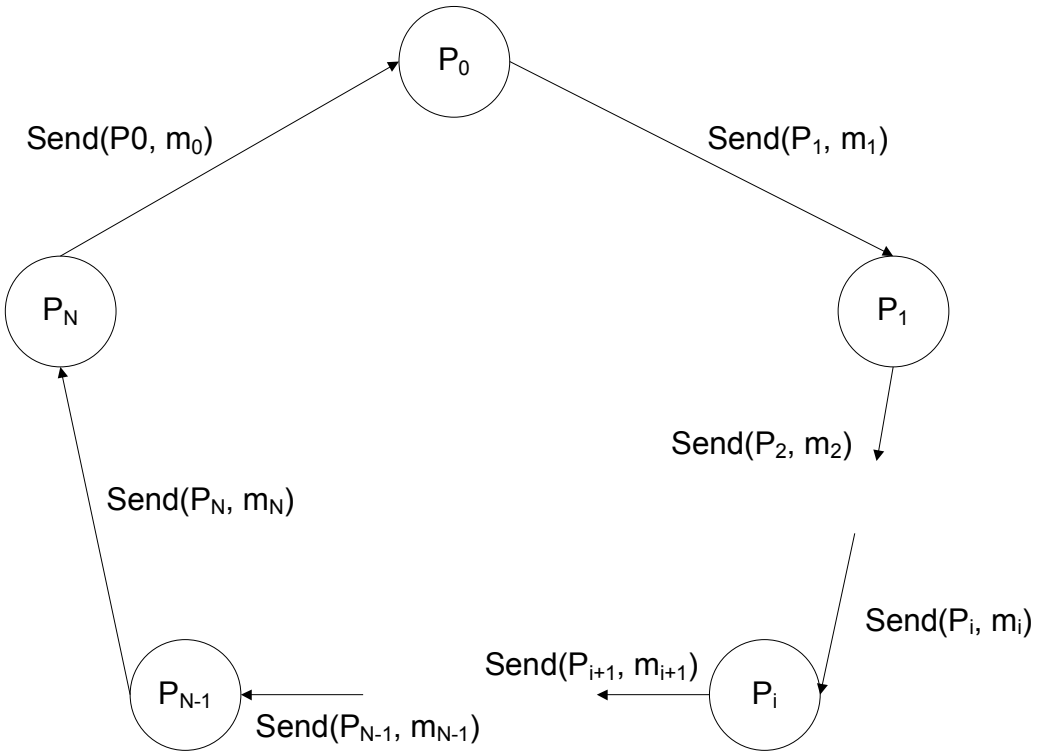


Рисунок 3-18. Взаимодействие процессов, объединенных в однонаправленное кольцо

3.5 Заключение и выводы по конструкторскому разделу

В рамках конструкторского раздела решены следующие задачи:

1. Разработаны основные структуры, необходимые для реализации и хранения БКА и РКА. При этом одной из основных структур является индекс (***Index***) – основная характеристика состояния (после метки и списка переменных), по которой производится поиск состояний во множестве состояний. Также индекс используется для организации переходов между состояниями.
2. Для снижения временных затрат на поиск состояний множество состояний РКА на одном узле реализовано в виде хеш-таблицы. Выбрана равномерная хеш-функция, вычисляющая хеш-значение состояния РКА в зависимости от метки и значений переменных в этом состоянии. Для разрешения коллизий используется метод цепочек.
3. Разработаны два алгоритма распределенной генерации и хранения РКА – ***GeneneratorCollectors*** (один процесс-генератор, осуществляющий генерацию состояний и множество процессов-коллекторов, осуществляющих хранение состояний РКА), и ***Distributor*** (все процессы осуществляют генерацию и хранение состояний). Состояния РКА распределяются по узлам в соответствии с хеш-функцией разбиения *Partition*, вычисляющая номер узла, на котором должно быть сохранено состояние в зависимости от метки и значений переменных в этом состоянии.
4. Таблица переходов РКА хранится по столбцам, что позволяет разрешить трудности, связанные с блокировкой процессов, возникающие при реализации алгоритма ***Distributor***.
5. Для определения завершения вычислений в алгоритме ***Distributor*** используется алгоритм, основанный на отслеживании каждым процессом количества отправленных и полученных в системе сообщений.

4 Технологический раздел

В данном разделе проводится обоснование выбора средств разработки, рассматриваются особенности реализации, тестирования и отладки программной реализации.

4.1 Выбор средств разработки

4.1.1 Система программирования MPI

В настоящее время существует множество систем параллельного программирования, как основанных на традиционных последовательных языках (OpenMP, DVM, mpC), так и основанных на передаче сообщений (MPI, Linda). В данной работе необходима система программирования с поддержкой распределенной памяти, к которым относятся MPI и Linda. Для создания программной реализации алгоритма распределенной генерации расширенных конечных автоматов для проверки темпоральных свойств систем в рамках данной работы использовалась технология MPI. Выбор был определен, в первую очередь, тем, что именно библиотека MPI (реализация MPICH) установлена на доступном для выполнения параллельных вычислений кластере МГТУ им. Н.Э. Баумана. В пользу MPI также говорит то, что на данный момент эта технология наиболее популярна среди технологий программирования параллельных компьютеров с распределенной памятью. Поэтому, в случае необходимости можно достаточно легко найти необходимую для решения проблемы информацию, либо получить квалифицированную помощь на форумах (например, форум parallel.ru). Среди основных особенностей MPI следует выделить:

- поддержку создания параллельных программ как в стиле MIMD (что подразумевает объединение процессов с различными исходными текстами), так и в стиле SPMD (для всех параллельных процессов используется один и тот же исходный код), который на практике используется более часто;
- подробно и четко специфицированный интерфейс;
- наличие большого количества высокоуровневых функций для решения различных задач;
- возможность создания пользовательских виртуальных топологий;
- возможность создания пользовательских типов данных.

Из недостатков следует выделить то, что различные реализации стандарта MPI могут существенно отличаться на низком уровне, и как следствие, эффективность выполнения программ может быть различной при переходе от одной реализации к другой. Кроме того,

разработчики также могут добавлять свои функции в конкретную реализацию, использование которых привести к необходимости внесения существенных изменений в код при переходе от реализации, в которой поддерживаются эти функции, к реализации, в которой такой поддержки нет. Поэтому рекомендуется при написании программ использовать только те функции, которые определены в стандарте (см. []).

4.1.2 Выбор языка программирования и среды разработки

Для реализации распределенной генерации использовалась библиотека MPI 1.1. Эта версия библиотеки поддерживает языки C/C++ и FORTRAN. Поэтому в качестве основного языка для написания программы был выбран язык C++ по следующим причинам:

- поддержка объектно-ориентированного программирования;
- позволяет производить оценку быстродействия и отладку приложений большой сложности, поддерживает объектную и модульную структуры приложений;
- стандартная библиотека STL языка C++ включает большое количество типов данных и функций, позволяющих не реализовывать пользовательские типы данных, например, списки, словари, множества.

В качестве среды разработки приложения использовалась Microsoft Visual Studio 2005 по следующим причинам. Интегрированная среда разработки Microsoft Visual Studio 2005 имеет в своем составе отладчик, предоставляющий широкие возможности для поиска и устранения ошибок, в том числе и для отладки MPI приложений. Заметим, что по умолчанию библиотека MPI в состав платформы .NET не входит. Поэтому необходимо предварительно установить пакет Microsoft Compute Cluster Pack (MS CCP). Этот пакет можно бесплатно загрузить с сайта корпорации Microsoft. Соответственно, при этом используется реализация библиотеки MPI от Microsoft, поэтому в библиотеку могут быть включены некоторые дополнительные функции или константы, не описанные в стандарте (например, функция MPI_Request_get_status). Поэтому, во избежание проблем при переносе программы на кластер, необходимо осторожно использовать функции, предоставляемые библиотекой MS MPI. Наиболее надежным является использование только тех функций, которые описаны в стандарте MPI. Ниже будут рассмотрены основные особенности отладки MPI –программ, в том числе и в среде Microsoft Visual Studio 2005.

Помимо отладчика, Microsoft Visual Studio 2005 обладает и другими преимуществами: полная и подробная документация, дружественный интерфейс пользователя, большое количество функций, сокращающих время написания кода.

4.2 Описание основных функций MPI, используемых в программе

MPI-программа - это множество параллельных взаимодействующих процессов. Все процессы порождаются один раз, образуя параллельную часть программы. В ходе выполнения MPI-программы порождение дополнительных процессов или уничтожение существующих не допускается (это верно для MPI 1.1, в MPI 2 такая возможность появилась). Каждый процесс работает в своем адресном пространстве, никаких общих переменных или данных в MPI нет. Основным способом взаимодействия между процессами является явная посылка сообщений. Для локализации взаимодействия параллельных процессов программы можно создавать *группы процессов,* предоставляя им отдельную среду для общения - *коммуникатор.* Состав образуемых групп произволен. Группы могут полностью совпадать, входить одна в другую, не пересекаться или пересекаться частично. Процессы могут взаимодействовать только внутри некоторого коммуникатора, сообщения, отправленные в разных коммуникаторах, не пересекаются и не мешают друг другу. Коммуникаторы имеют в языке С – предопределенный тип `MPI_Comm`. При старте программы всегда считается, что все порожденные процессы работают в рамках всеобъемлющего коммуникатора, имеющего предопределенное имя `MPI_COMM_WORLD`. Этот коммуникатор существует всегда и служит для взаимодействия всех запущенных процессов MPI-программы. Кроме него при старте программы имеется коммуникатор `MPI_COMM_SELF`, содержащий только один текущий процесс, а также коммуникатор `MPI_COMM_NULL`, не содержащий ни одного процесса. Все взаимодействия процессов протекают в рамках определенного коммуникатора, сообщения, переданные в разных коммуникаторах, никак не мешают друг другу.

Каждый процесс MPI-программы имеет в каждой группе, в которую он входит, уникальный атрибут *номер процесса,* который является целым неотрицательным числом. С помощью этого атрибута происходит значительная часть взаимодействия процессов между собой. Ясно, что в одном и том же коммуникаторе все процессы имеют различные номера. Но поскольку процесс может одновременно входить в разные коммуникаторы, то его номер в одном коммуникаторе может отличаться от его номера в другом. Отсюда становятся понятными *два основных атрибута процесса: коммуникатор и номер в*

коммуникаторе. Если группа содержит n процессов, то номер любого процесса в данной группе лежит в пределах от 0 до $n - 1$.

Основным способом общения процессов между собой является явная посылка сообщений. *Сообщение* — это набор данных некоторого типа. Каждое сообщение имеет несколько *атрибутов,* в частности, номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения и другие. Одним из важных атрибутов сообщения является его идентификатор или тэг. По идентификатору процесс, принимающий сообщение, например, может различить два сообщения, пришедшие к нему от одного и того же процесса. Сам идентификатор сообщения является целым неотрицательным числом, лежащим в диапазоне от 0 до `MPI_TAG_UP`, причем гарантируется, что `MPI_TAG_UP` не меньше 32767. Для работы с атрибутами сообщений в языке С введена структура `MPI_Status`, поля которой дают доступ к их значениям.

В языке С в возвращаемом значении функции большинство процедур MPI возвращают информацию об успешности завершения. В случае успешного выполнения возвращается значение MPI_SUCCESS, иначе - код ошибки. Вид ошибки, которая произошла при выполнении процедуры, можно будет определить из ее описания. Предопределенные значения, соответствующие различным ошибочным ситуациям, перечислены в файле mpi.h.

Все дополнительные объекты: имена процедур, константы, предопределенные типы данных и т.п., используемые в MPI, имеют префикс MPI_. Если пользователь не будет использовать в программе имен с таким префиксом, то конфликтов с объектами MPI заведомо не будет. В языке Си, кроме того, является существенным регистр символов в названиях функций. Обычно в названиях функций MPI первая буква после префикса MPI_ пишется в верхнем регистре, последующие буквы – в нижнем регистре, а названия констант MPI записываются целиком в верхнем регистре. Все описания интерфейса MPI собраны в файле mpi.h**,** поэтому в начале MPI-программы должна стоять директива #include “mpi.h”.

Стандарт MPI 1.1 включает более 120 функций. В данном разделе рассмотрены основные функции MPI, используемые в программе.

4.2.1 Общие функции MPI

В данном разделе рассмотрены общие функции MPI, не связанных с пересылкой данных. Большинство функций этого раздела необходимо практически в каждой содержательной параллельной программе.

int MPI_Init(int *argc, char ***argv)

Инициализация параллельной части программы. Все другие процедуры MPI могут быть вызваны только после вызова `MPI_Init`. Инициализация параллельной части для каждого приложения должна выполняться только один раз. В качестве параметров функции `MPI_Init` передаются указатели на аргументы командной строки программы `argc` и `argv`, из которых системой могут извлекаться и передаваться в параллельные процессы некоторые параметры запуска программы.

int MPI_Finalize(void)

Завершение параллельной части приложения. Все последующие обращения к любым процедурам MPI, в том числе к `MPI_Init`, запрещены. К моменту вызова `MPI_Finalize` каждым процессом программы все действия, требующие его участия в обмене сообщениями, должны быть завершены.

MPI_Initialized(int *flag)

Процедура возвращает в аргументе `flag` значение 1, если вызвана из параллельной части приложения, и значение 0 - в противном случае. Это единственная процедура MPI, которую можно вызвать до вызова `MPI_Init`.

int MPI_Comm_size(MPI_Comm comm, int *size)

В аргументе `size` процедура возвращает число параллельных процессов в коммуникаторе `comm`.

MPI_Comm_rank(MPI_Comm comm, int *rank)

В аргументе `rank` процедура возвращает номер процесса в коммуникаторе `comm`. Если процедура `MPI_Comm_size` для того же коммуникатора `comm` вернула значение `size`, то значение, возвращаемое процедурой `MPI_Comm_rank` через переменную `rank`, лежит в диапазоне от `0` до `size-1`.

double MPI_Wtime()

Эта функция возвращает на вызвавшем процессе астрономическое время в секундах (вещественное число), прошедшее с некоторого момента в прошлом. Если некоторый участок программы окружить вызовами данной функции, то разность возвращаемых значений покажет время работы данного участка. Гарантируется, что момент времени, используемый в качестве точки отсчета, не будет изменен за время существования

процесса. Заметим, что эта функция возвращает результат своей работы не через параметры, а явным образом.

```
int MPI_Get_processor_name(char *name, int *resultlen)
```

Процедура возвращает в строке `name` имя узла, на котором запущен вызвавший процесс. В переменной `len` возвращается количество символов в имени, не превышающее значения константы `MPI_MAX_PROCESSOR_NAME`. С помощью этой процедуры можно определить, на какие именно физические процессоры были спланированы процессы MPI - приложения.

4.2.2 Функции для приема и передачи сообщений между отдельными процессами

В MPI можно выделить функции для взаимодействия двух процессов (индивидуальные операции, или операции «точка-точка») и функции, которые вовлекают во взаимодействие все процессы некоторого коммуникатора (коллективные операции). При реализации программы использовались функции первого типа. Во взаимодействиях типа «точка - точка» участвуют два процесса, причем один процесс является отправителем сообщения, а другой - получателем. Процесс-отправитель должен вызвать одну из процедур передачи данных и явно указать номер в некотором коммуникаторе процесса-получателя, а процесс-получатель должен вызвать одну из процедур приема с указанием того же коммуникатора, причем в некоторых случаях он может не знать точный номер процесса-отправителя в данном коммуникаторе. Все процедуры данной группы, в свою очередь, так же делятся на два класса: процедуры с блокировкой (с синхронизацией) и процедуры без блокировки (асинхронные). Процедуры обмена с блокировкой приостанавливают работу процесса до выполнения некоторого условия, а возврат из асинхронных процедур происходит немедленно после инициализации соответствующей коммуникационной операции. Неаккуратное использование процедур с блокировкой может привести к возникновению тупиковой ситуации, поэтому при этом требуется дополнительная осторожность. Использование асинхронных операций к тупиковым ситуациям не приводит, однако требует более аккуратного использования массивов данных.

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

- `buf` – адрес начала буфера с посылаемым сообщением;
- `count` – число передаваемых элементов в сообщении;

- `datatype` – тип передаваемых элементов;
- `dest` – номер процесса – получателя;
- `tag` – идентификатор сообщения;
- `comm` – идентификатор коммуникатора.

Блокирующая посылка массива `buf` с идентификатором `tag` состоящего из `count` элементов типа `datatype`, процессу с номером `dest` в коммуникаторе `comm`. Все элементы посылаемого сообщения должны быть расположены подряд в буфере `buf`. Операция начинается независимо от того, была ли инициализирована соответствующая процедура приема. При этом сообщение может быть скопировано как непосредственно в буфер приема, так и помещено в некоторый системный буфер (если это предусмотрено в MPI). Значение `count` может быть нулем. Процессу разрешается передавать сообщение самому себе, однако это небезопасно и может привести к возникновению тупиковой ситуации. Тип передаваемых элементов `datatype` должен указываться с помощью предопределенных констант типа, например, `MPI_INT`, `MPI_LONG`, `MPI_CHAR`, `MPI_FLOAT` и других, либо с помощью производных типов. Для каждого типа данных языка С есть своя константа. Полный список предопределенных имен типов можно посмотреть в файле mpi.h. Блокировка гарантирует корректность повторного использования всех параметров после возврата из процедуры. Это означает, что после возврата из `MPI_Send` можно использовать любые присутствующие в вызове данной процедуры переменные без опасения испортить передаваемое сообщение. Выбор способа осуществления этой гарантии: копирование в промежуточный буфер или непосредственная передача процессу `dest`, остается за разработчиками конкретной реализации MPI. Следует специально отметить, что возврат из процедуры `MPI_Send` не означает ни того, что сообщение получено процессом `dest`, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший данный вызов. Предоставляется только гарантия безопасного изменения переменных, использованных в вызове данной процедуры. Подобная неопределенность далеко не всегда устраивает пользователя. Чтобы расширить возможности передачи сообщений, в MPI введены дополнительные три процедуры, однако их рассмотрение выходит за рамки данного раздела.

```
int MPI_Recv_init(void *buf, int count, MPI_Datatype datatype, int
                  source, int tag, MPI_Comm comm, MPI_Request
                  *request)
```

- `buf` - адрес начал буфера для приема сообщения;
- `count` – максимальное число элементов в принимаемом сообщении;

- `datatype` – тип элементов принимаемого сообщения;
- `source` – номер процесса - отправителя или `MPI_ANY_SOURCE`;
- `tag` – идентификатор принимаемого сообщения или `MPI_ANY_TAG`;
- `comm` – идентификатор коммуникатора;
- `request` - выходной параметр, идентификатор операции асинхронного приема сообщения.

Формирование отложенного запроса на прием сообщения. Сама операция приема при этом не начинается до вызова `MPI_Start` или его аналогов! По завершении приема данных параметр запроса `request` может быть использован повторно. При приеме сообщения вместо аргументов `source` и `msgtag` можно использовать следующие предопределенные константы: `MPI_ANY_SOURCE` - признак того, что подходит сообщение от любого процесса; `MPI_ANY_TAG` - признак того, что подходит сообщение с любым идентификатором. При одновременном использовании этих двух констант будет принято сообщение с любым идентификатором от любого процесса. (Реальные атрибуты принятого сообщения всегда можно определить по соответствующим значениям полей структуры `status`, который передается как параметр в функции `MPI_Recv` и `MPI_Irecv`. При использовании же функции `MPI_Recv_init` для определения значений `status` можно использовать `MPI_Wait`, см. ниже. В языке С параметр `status` является структурой предопределенного типа MPI_Status с полями `MPI_SOURCE`, `MPI_TAG` и `MPI_ERROR`). Обратим внимание на некоторую несимметричность операций посылки и приема сообщений. С помощью константы `MPI_ANY_SOURCE` можно принять сообщение от любого процесса. Однако в случае посылки данных требуется явно указать номер принимающего процесса.

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
               MPI_Status *status)
```

- `source` – номер процесса-отправителя или `MPI_ANY_SOURCE`;
- `tag` - идентификатор ожидаемого сообщения или `MPI_ANY_TAG`;
- `comm` – идентификатор коммуникатора;
- `flag` – выходной параметр, признак завершенности операции обмена;
- `status` - выходной параметр, параметры приходящего сообщения.

Получение информации о поступлении и структуре ожидаемого сообщения без блокировки. В параметре `flag` возвращается значение 1, если сообщение с подходящими атрибутами уже может быть принято, и значение 0, если сообщения с указанными

атрибутами еще нет. Атрибуты доступного сообщения можно определить по значению параметра `status`.

`int MPI_Start(MPI_Request *request)`

- `request` – отложенный запрос на взаимодействие.

Инициализация отложенного запроса на выполнение операции обмена, соответствующей значению параметра `request`, инициированного `MPI_Send_init` или `MPI_Recv_init`. Операция запускается как неблокирующая.

`int MPI_Wait(MPI_Request *request, MPI_Status *status)`

- `request` – идентификатор операции асинхронного приема или передачи;
- `status` – выходной параметр, параметры сообщения.

Ожидание завершения асинхронной операции приема или передачи с идентификатором `request`. Пока асинхронная операция не будет завершена, процесс, выполнивший вызов функции `MPI_Wait`, будет заблокирован. Если речь идет о приеме, то атрибуты и длину принятого сообщения можно определить с помощью параметра `status`.

4.3 Основные ошибки, допускаемые при написании параллельных MPI программ

Все ошибки, которые встречаются при последовательном программировании, характерны также и для параллельного программирования. Однако кроме них есть ряд специфических типов ошибок, обусловленных наличием в MPI-программе нескольких взаимодействующих процессов. Дополнительные сложности в разработке параллельных программ обуславливают повышенные требования к квалификации программистов, реализующих параллельные версии алгоритмов. И хотя перечислить все типы ошибок, которые могут возникнуть при программировании с использованием технологии MPI, крайне затруднительно, дадим краткую характеристику ряду наиболее характерных ошибок, преследующих начинающих разработчиков. К числу наиболее типичных ошибок при параллельном программировании с использованием технологии MPI следует отнести:

- ***Взаимная блокировка при пересылке сообщений.*** Предположим, что n процессов передают информацию друг другу по цепочке так, что процесс с индексом i передает информацию процессу с индексом $i+1$ (для индексов $i = 0, \dots, n-2$), а процесс с индексом $n-1$ передает информацию процессу с

индексом 0 . Передача осуществляется с использованием функции отправки сообщений `MPI_Send` и функции приема сообщений MPI_Recv, при этом каждый процесс сначала вызывает `MPI_Send`, а затем `MPI_Recv`. Функции являются блокирующими, то есть `MPI_Send` возвратит управление только тогда, когда передача будет завершена, или когда сообщение будет скопировано во внутренний буфер. Таким образом, стандарт допускает возможность, что функции отправки сообщения вернут управление только после того, как передача будет завершена. Но передача не может завершиться, пока принимающий процесс не вызовет функцию `MPI_Recv`. А функция `MPI_Recv` будет вызвана, в свою очередь, только после того, как процесс завершит отправку своего сообщения. Так как цепочка передачи сообщений замкнута, отправка сообщений может не завершиться никогда, потому что каждый процесс будет ожидать завершения отправки своего сообщения, и никто не вызовет функцию приема сообщения. Избежать подобной блокировки можно, например, вызывая на процессах с четными индексами сначала `MPI_Send`, а затем `MPI_Recv`, а на процессах с нечетными индексами – в обратной порядке. Несмотря на очевидность ошибки, ее часто допускают, так как при небольших сообщениях высока вероятность, что сообщения уберутся во внутренние структуры библиотеки, функция отправки сообщения вернет управление, и ошибка не даст о себе знать. Кроме того, тупиковые ситуации можно разрешать, используя функции неблокирующего приема и передачи (`MPI_Isend`, `MPI_Irecv`), или совмещенные функции приема и передачи.

- ***Освобождение или изменение буферов, используемых при неблокирующей пересылке***. При вызове неблокирующих функций возврат из них происходит немедленно, а адреса массивов с сообщениями запоминаются во внутренних структурах библиотеки MPI. При непосредственной передаче сообщений по сети, которая будет выполнена в отдельном потоке, происходит обращение к исходному массиву с сообщением, поэтому важно, чтобы этот участок памяти не был удален или изменен до окончания передачи (установить, что передача завершена можно вызовом специальных функций, некоторые из которых рассмотрены выше). Однако часто об этом правиле забывают, что приводит к непредсказуемому поведению программы. Такого рода ошибки являются одними из самых сложных для отладки.

- ***Несоответствие вызовов функций передачи и приема сообщений.*** Важно следить, чтобы каждому вызову функции отправки сообщения соответствовал вызов функции приема и наоборот. Однако в том случае, когда число передач заранее неизвестно, а определяется в ходе вычислений, бывает легко ошибиться и не обеспечить нужных гарантий выполнения данного условия. К этому же типу ошибок можно отнести вызов функций передачи и приема сообщений с несоответствующими тэгами идентификации сообщений.

Это далеко не полный список ошибок, допускаемых при написании программ с использованием технологии MPI, однако это одни из наиболее часто допускаемых начинающими разработчиками ошибок, на которые следует обратить внимание при отладке программы.

4.4 Особенности отладки параллельных MPI программ

Отладка (поиск и устранение ошибок) – один из важнейших этапов в написании программных систем, часто занимающий у разработчика больше времени, чем написание отлаживаемого кода. Для того, чтобы максимально снизить время, затрачиваемое на отладку, необходимо придерживаться рекомендаций крупнейших производителей программного обеспечения и ведущих исследователей в области компьютерных наук еще на этапах проектирования и программирования. Например, подобные рекомендации обычно требуют подготавливать автоматические тесты для всех участков разрабатываемой системы и контролировать корректность значений внутренних переменных с использованием утверждений (assert). Но и сама процедура отладки должна проводиться эффективно, в соответствии с рекомендациями ведущих экспертов в этой области. Огромное значение здесь имеет правильный выбор отладчика – специальной программы, существенно упрощающий процесс поиска ошибок, позволяющий выполнять программу в пошаговом режиме, отслеживать значения переменных, устанавливать точки остановки и т.д.

4.4.1 Проблемы, возникающие при отладке MPI-программ

Отладка параллельных MPI программ имеет ряд особенностей, обусловленных природой программирования для кластерных систем. В параллельной программе над решением задачи работают одновременно несколько процессов, каждый из которых, в свою очередь, может иметь несколько потоков команд. Это обстоятельство существенно усложняет отладку, так как помимо ошибок, типичных для последовательного

программирования, появляются ошибки, совершаемые только при разработке параллельных программ. К числу таких ошибок можно отнести, например, сложно контролируемую ситуацию *гонки процессов*, когда процессы параллельной программы взаимодействуют между собой без выполнения каких-либо синхронизирующих действий. В этом случае, в зависимости от состояния вычислительной системы, последовательность выполняемых действий может различаться от запуска к запуску параллельной программы. Как результат, при наличии гонки процессов сложным становится применение одного из основных принципов отладки – проверка работоспособности при помощи тестов (однократное выполнение теста для параллельной программы может не выявить ситуации гонки процессов). Это еще одна причина, по которой необходимо иметь эффективные способы для отладки параллельных программ.

4.5 Основные способы отладки параллельных MPI-программ

Основными используемыми способами отладки MPI-программ являются:

- Использование текстовых сообщений, выводимых в файл или на экран, со значениями интересуемых переменных и/или информацией о том, какой именно участок программы выполняется. Такой подход часто называют “printf отладкой” (“printf debugging”), так как чаще всего для вывода сообщений на консольный экран используется функция “printf”. Данный подход хорош тем, что он не требует от программиста специальных навыков работы с каким-либо отладчиком, и при последовательном применении позволяет найти ошибку. Однако для того, чтобы получить значение очередной переменной, приходится каждый раз писать новый код для вывода сообщений, перекомпилировать программу и производить новый запуск. Это занимает много времени. Кроме того, добавление в текст программы нового кода может приводить к временному исчезновению проявлений некоторых ошибок. Существенным недостатком является то, что вывод отладочных сообщений может осуществляться разными процессами в один и тот же поток, что приводит к трудностям при определении причин возникновения ошибки.
- Использование утверждений (assert) также позволяет достаточно эффективно обнаруживать ошибки. Однако, для эффективного использования утверждений надо уметь четко формулировать условия, проверяемые в утверждениях. Кроме

того, использование утверждений также увеличивает объем кода и приводит к необходимости перекомпилировать программу каждый раз при добавлении нового утверждения.

- Использование последовательного отладчика. Так как MPI задача состоит из нескольких взаимодействующих процессов, то для отладки можно запустить несколько копий последовательного отладчика, присоединив каждую из них к определенному процессу MPI задания. Данный подход позволяет в некоторых случаях более эффективно производить отладку, чем при использовании текстовых сообщений. Главным недостатком подхода является необходимость ручного выполнения многих однотипных действий (например, необходимость приостановки 32 процессов MPI задания: в случае использования последовательного отладчика придется переключиться между 32 процессами отладчика и вручную дать команду приостановки).
- Использование функции MPI_Pcontrol, позволяющей создавать трассировочные файлы, и коммерческого отладчика Trace View, позволяющего отображать трассировочные файлы в удобном пользователю виде.

Заметим, что, несмотря на все недостатки, вышеперечисленные способы отладки являются основными наиболее часто используемыми при создании MPI-программ в силу своей универсальности. Кроме того, встроенные отладчики (например, рассмотренный ниже отладчик среды Microsoft Visal Studio 2005) также не всегда позволяют точно локализовать ошибку и определить ее причины. Но, тем не менее, использование отладчиков может значительно упростить поиск ошибок при создании параллельных MPI-программ. Поэтому правильный выбор отладчика достаточно важен. Важнейшими критериями выбора отладчика являются богатство предоставляемого программисту инструментария и удобство использования. В ходе выполнения работы использовался отладчик MPI-программ среды Microsoft Visal Studio 2005.

4.5.1 Отладчик параллельных MPI - программ в среде Microsoft Visual Studio 2005

Отладчик параллельных MPI - программ в Microsoft Visual Studio появился впервые в версии Visual Studio 2005. В данном разделе рассмотрены основные возможности предоставляемые отладчиком MPI-программ среды Visual Studio 2005. Прежде всего отметим, что среда рассматривает все процессы одной MPI задачи как единую параллельно выполняемую программу, максимально приближая отладку к

отладке последовательных программ. При запуске MPI-программы открывается основное окно (для главного процесса mpiexec.exe) и кроме того, по одному окну на каждый процесс, число которых указано в параметрах командной строки для запуска MPI-программы. Таким образом, сообщения выводимые различными процессами, не будут перемешаны. Есть возможность переключения между параллельными процессами. В остальном отладка параллельных в Visual Studio 2005 программ схожа с отладкой последовательных программ. Наиболее частым приемом отладки является приостановка работы программы в заданный момент времени и анализ значений ее переменных. Момент, в который программа будет приостановлена, определяется выбором, так называемых, точек остановки, то есть указанием строк кода исходной программы, по достижении которых выполнение останавливается до получения соответствующей команды пользователя. Помимо простого указания строк кода, где произойдет приостановка, возможно также указание условий, которые должны при этом выполняться. Например, можно дать указание приостановить выполнение программы на заданной строке только в том случае, если значение некоторой переменной программы превысило заданную константу. Правильный выбор момента остановки имеет решающее значение для успеха отладки. Так, зачастую анализ значений переменных непосредственно перед моментом падения программы дает достаточно информации для определения причин некорректной работы. Другим исключительно полезным инструментом является возможность выполнять программу в пошаговом режиме – по одной строке исходного кода отлаживаемой программы при каждом нажатии пользователем кнопки F10. При этом пользователь также имеет возможность заходить внутрь функции, вызов которой происходит на данной строке, или просто переходить к следующей строке. Таким образом, пользователь всегда находится на том уровне детализации, который ему необходим. Для того, чтобы в полной мере оценить преимущества отладчика, необходимо предварительно скомпилировать программу в специальной отладочной конфигурации (чаще всего такая конфигурация называется “Debug”), особенностью которой является добавление в генерируемые бинарные файлы специальной отладочной информации, которая позволяет видеть при отладке исходный код выполняемой программы на языке высокого уровня. К числу других часто используемых инструментов отладки Microsoft Visual Studio 2005 относятся:

- Окно **“Call Stack”** - окно показывает текущий стек вызова функций и позволяет переключать контекст на каждую из функций стека (при переключении контекста программист получает доступ к значениям локальных переменных функции);

- Окно **“Autos”** - окно показывает значения переменных, используемых на текущей и на предыдущих строках кода. Кроме того, это окно может показывать значения, возвращаемые вызываемыми функциями. Список отображаемых значений определяется средой автоматически;
- Окно **“Watch”** - окно позволяет отслеживать значения тех переменных, которых нет в окне **“Autos”**. Список отслеживаемых переменных определяется пользователем;
- Окно **“Threads”** - окно позволяет переключаться между различными потоками команд процесса;
- Окно **“Processes”** - окно позволяет переключаться между различными отлаживаемыми процессами (например, в случае отладки MPI – программы).

Таким образом, инструменты и приемы, используемые в Microsoft Visual Studio 2005 для отладки как последовательных, так и параллельных программ схожи, поэтому для разработчика, имеющего хороший опыт отладки последовательных программ, отладка параллельных программ также будет не слишком сложной.

4.6 Определение требований к вычислительной системе

Использование распределенных вычислений и системы параллельного программирования MPI, а также большое количество состояний РКА, которые необходимо хранить в оперативной памяти одновременно, выдвигают ряд требований к вычислительной системе:

1. В связи с использованием системы MPI, необходимо, чтобы на вычислительной системе была установлена конкретная реализация MPI (например, MPICH или LAM), а также компилятор языка C++ и стандартная библиотека STL языка C++;
2. Для эффективного выполнения параллельного алгоритма генерации, использующего библиотеку MPI, необходимо иметь более одного процессора, либо многоядерный процессор(ы). Конфигурация с одноядерным процессором позволяет запускать и тестировать MPI-программу, но реально параллельного выполнения не происходит.
3. Для хранения столь большого числа состояний необходима система с разделенной памятью (в идеале кластер).

При разработке и тестировании программы использовалась следующая конфигурация:

- Ноутбук Dell Latitude D630
- Процессор: Intel Core2 Duo T8340 (2 ядра, 2400 MHz, кэш L1 64 kB, кэш L2 3MB)
- ОЗУ: 2045 Мб
- Жесткий диск: Seagate Momentus ST9160823ASG, 160 Gb, 7200 rpm
- Видеокарта: NVidia Quadro NVS 135M
- Операционная система: Windows XP Service Pack 3
- В качестве конкретной реализации MPI использовалась реализация библиотеки фирмы Microsoft, входящая в состав Microsoft Compute Cluster Pack.

4.7 Основные классы, реализованный в программе

Всего в программе реализовано более 30 классов. В этом разделе кратко описаны основные классы и основные методы классов, реализованные в программе. Для удобства классы разбиты на группы. Приводится диаграмма основных классов.

4.7.1 Описание основных классов, реализующих конечный автомат

Класс **Index** реализует индекс состояния. Все методы класса предназанчены для задаиния / получения значений полей экземпляра класса, и поэтому здесь не описываются.

```
class Index
{
private:
	// Индекс процессора, на котором хранится состояние
	int _procIndex;
	// Индекс ячейки в хеш-таблице
	int _hashIndex;
	// Индекс состояния в списке, соответствующем ячейке хеш-таблицы
	// с индексом _hashIndex
	int _listIndex;

	// . . .

public:
	// Задать индекс процессора
	void SetProcIndex(int procIndex);
public:
	// Возвращает индекс процессора
	int GetProcIndex(void) const;

	// . . .
};
```

Класс **Variable** реализует переменную. Класс **VarList** реализует список переменных с использованием класса list библиотеки STL и здесь не описывается

```
class Variable
{
public:
	// Имя переменной
	TagPtr _tag;
	// Значение переменной
	int _value;

	// . . .
};
```

Класс **Transition** реализует переход конечного автомата. Помимо методов запроса и модификации полей включает методы проверки предиката и выполнения действия. Класс **TransList** реализует список переменных с использованием класса list библиотеки STL и здесь не описывается.

```
class Transition
{
private:
	// Метка перехода
	TransitionLabel _label;
	// Индекс состояния, из которого исходит дуга (переход)
	Index _from;
	// Индекс состояния, в которое входит дуга (переход)
	Index _to;
	// Обработчик предиката перехода
	PredicateHandler _predicateHandler;
	// Обработчик действия перехода
	ActionHandler _actionHandler;

	// . . .

public:
	// Проверить предикат
	bool Predicate(const VarList& varList) const;
public:
	// Выполнить действие перехода
	VarList Action(const VarList& varList) const;

	// . . .
};
```

Класс **State** реализует состояние конечного автомата. Метка и список переменных состояния реализуются классом **StateLabel**. Также класс **State** включает список переходов, входящих или исходящих из состояния в зависимости от выбранного формата хранения. Основные методы класса состояния – запрос и модификация данных (например, значений переменных), а также добавление переходов в список переходов состояния. Класс **StateList** реализует список переменных с использованием класса list

библиотеки STL и здесь не описывается. Класс **StateHash** реализует хеш-таблицу состояний, здесь не рассматривается. Работа с классом State осуществляется через «умный указатель», реализуемый классом **StatePtr**.

```
class State
{
public:
	// Метка и список переменных состояния
	StateLabel _label;
	// Индекс состояния
	Index _index;
	// Список исходящих из состояния переходов
	TransList _transitions;

	// . . .
public:
	// Задать метку состояния
	void SetStringLabel(string label);
public:
	// Вернуть метку состояния
	string GetStringLabel(void) const;
public:
	// Задать индекс состояния
	void SetIndex(const Index& index);
public:
	// Вернуть индекс состояния
	const Index& GetIndex(void) const;

	// . . .

public:
	// Вернуть список переменных состояния
	const VarList& GetVarList(void) const;
public:
	// Добавить переход в список исходящих переходов состояния
	void AddTransition(const Transition& trans);
public:
	// Сравнение состояний по меткам
	bool IsEqualByStringLabel(const State& state);
public:
	// Сравнение состояний по меткам и спискам переменных
	bool IsEqual(const State& state);

	// . . .
};
```

Класс **FSM** реализует интерфейс для классов базового и расширенного конечного автомата, поэтому здесь приводится полное описание методов и полей класса **FSM**.

```
class FSM
{
protected:
	// Начальное состояние конечного автомата
	StatePtr _startState;
	// Формат хранения состояний и переходов конечного автомата
	FSMStorageFormat* _storageFormat;
```

```
public:
      // Добавление состояния в список состояний
      virtual void AddState(StatePtr state) = 0;
public:
      // Задать начальное состояние конечного автомата
      virtual void SetStartState(const StateLabel& label);
public:
      // Вернуть начальное состояние
      virtual const StatePtr GetStartState(void) const;
public:
      // Проверка, есть ли состояние с указанной меткой в списке состояний
      // автомата. При поиске в базовом конечном автомате сравнения происходит
      // только по строковым меткам
      virtual bool ContainsState(const StateLabel& label) const;
public:
      // Добавить переход в список переходов конечного автомата
      virtual void AddTransition(const Transition& trans);
public:
      // Добавить переход в список переходов конечного автомата
      virtual void AddTransition(const StateLabel& labelFrom,
                                 const StateLabel& labelTo,
                                 const TransitionLabel& transLabel);
public:
      // Добавить переход в список переходов конечного автомата
      virtual void AddTransition(const Index& indexFrom,
                                 const Index& indexTo,
                                 const TransitionLabel& transLabel);
public:
      // Возвращает индекс состояния с заданной меткой в конечном автомате
      virtual Index GetIndexOfState(const StateLabel& label) const;
public:
      // Печать базового конечного автомата. Используется при отладке
      virtual void Print(void);
public:
      // Вернуть состояние с заданной меткой из списка состояний
      // При поиске в базовом конечном автомате сравнения происходит
      // только по строковым меткам
      virtual StatePtr GetState(const StateLabel& label) const = 0;
public:
      // Вернуть состояние с заданным индексом из списка состояний
      virtual StatePtr GetState(const Index& index) const = 0;

};
```

Класс **BaseFSM** реализует интерфейс для классов базового и расширенного конечного автомата, поэтому здесь приводится полное описание методов и полей класса **FSM**. Интерфейс полностью совпадает с интерфейсом класса FSM. Для хранения множества состояний использует список. Таблица переходов хранится по строкам.

```
class BaseFSM : public FSM
{
private:
      // Список состояний конечного автомата
      StateList _states;
      // . . .
};
```

Класс **ExpandedFSM** реализует интерфейс для классов базового и расширенного конечного автомата, поэтому здесь приводится полное описание методов и полей класса **FSM**. Интерфейс полностью совпадает с интерфейсом класса FSM. Для хранения множества состояний использует список. Таблица переходов хранится по столбцам.

```
class ExpandedFSM : public FSM
{
private:
	// Хеш-таблица состояний
	StateHash _states;

	// . . .
};
```

При распределенной генерации и хранении подмножества множества состояний РКА хранятся на разных узлах.

Таким образом, реализованный классы почти полностью соответствуют основным структурам, описанным в разделе 3.2.1.

4.7.2 Описание основных классов, реализующих генерацию расширенного конечного автомата

Класс **BFS_EFSMGenerator**. Класс реализует генерацию расширенного конечного автомата методом поиска в ширину. Почти все методы и поля данного класса используются в производных классах.

```
class BFS_EFSMGenerator : public EFSMGenerator
{
	// . . .
	// Поля класса
	// . . .

public:
	// Генерация расширенного конечного автомата по заданному
	// базовому конечному автомату
	virtual ExpandedFSM GenerateExpandedFSM(const BaseFSM& baseFSM,
                                            const VarList& varList);
protected:
	// По данному состоянию и списку переменных вычислить всех потомков
	// и соответствующие им данные и занести в очередь
	virtual void PushNextQueueElements(const BaseFSM& baseFSM,
                                       const Index& indexFrom);
protected:
	// Извлекает с головы очереди очередное состояние для исследования
	// в поле _currentQE
	virtual void PopCurrentQueueElement(void);
protected:
	// Добавить состояние и переход в расширенный конечый автомат
	// Если состояние добавлено в расширенный конечный автомат,
	// то возвращается его индекс в автомате,
	// в противном случае возвращается некорректный индекс (InvalidIndex)
	virtual Index AddStateAndTrans(ExpandedFSM& expFSM);
protected:
```

```
        // Исследовать очередное состояние из очереди
        // Возвращает true, если состояние уже было исследовано
        virtual void ExploreState(ExpandedFSM& expFSM, const BaseFSM& baseFSM);
protected:
        // Создание состояния расширенного КА по данным первого элемента очереди
        virtual StatePtr CreateCurrentExpState(void);
};
```

Класс **MIMD_EFSMGenerator** реализует алгоритм ***GeneratorCollectors*** распределенной генерации РКА по заданному БКА. Описаны только основные методы.

```
class MIMD_EFSMGenerator : public BFS_EFSMGenerator
{
        // . . .
        // Поля класса
        // . . .

public:
        // Параллельная генерация расширенного конечного автомата по заданному
        // базовому конечному автомату
        virtual ExpandedFSM GenerateExpandedFSM(const BaseFSM& baseFSM,
                                                const VarList& varList);
        // Запусить процесс-генератор
        void RunGenerator(ExpandedFSM& expFSM, const BaseFSM& baseFSM,
                          const VarList& varList);
private:
        // Запусить процесс-генератор
        void RunCollector(ExpandedFSM& expFSM, const BaseFSM& baseFSM,
                          const VarList& varList);
private:
        // Вычислить хеш-функцию состояния для определения узла, на котором
        // это состояние необходимо исследовать
        virtual int Partition(StatePtr state);
private:
        // Отправить состояние и ведущий в него переход на другой узел
        // Номер узла должен быть записан в state->ProcIndex
        void SendArc(int procIndex);
private:
        // Принять очередное сообщение, пришедшее на текущий узел.
        // Возвращает тег принятого сообщения
        bool Receive(int* tag);

        // . . .
};
```

Класс **SIMD_EFSMGenerator** реализует алгоритм ***Distributor*** распределенной генерации РКА по заданному БКА. Описаны только основные методы. Класс реализует параллельную генерацию расширенного конечного автомата по заданному базовому конечному автомату. В качестве основы используется поиск в ширину. Процессы организованными в топологию "однонаправленное кольцо". Указаны только основные методы и поля, используемые для определения признака завершения выполнения распределенных вычислений.

```
class SIMD_EFSMGenerator : public BFS_EFSMGenerator
{
private:

      // Следующие поля используются для определения завершения параллельных
      // распределенных вычислений
      bool _initiator;  // Флаг, указывающий, является ли текущий процесс
                        // инициатором
      bool _terminit;   // Флаг, указывыющий, завершился ли процесс-инициатор
      bool _terminated; // Флаг, указывающий, завершились ли вычисления
      unsigned long _nbsent;  // Количество сообщений отправленных
                              // текущим процессом
      unsigned long _nbrecd;  // Количество сообщений принятых текущим
                              // процессом
      unsigned long _totalrecd;     // Количество принятых в системе сообщений
      int _myRank;            // Номер текущего процесса в кольце
      int _nextRank;          // Номер следующего процесса в кольце
      int _commSize;          // Размер коммуникатора
      int _k;                 // Принимаемое сообщение

      // . . .

public:
      // Параллельная генерация расширенного конечного автомата по заданному
      // базовому конечному автомату
      virtual ExpandedFSM GenerateExpandedFSM(const BaseFSM& baseFSM,
                                              const VarList& varList);
public:
      // Вычислить хеш-функцию состояния для определения узла, на котором
      // это состояние необходимо исследовать
      virtual int GetStateHash(StatePtr state);
private:
      // Отправить состояние и ведущий в него переход на другой узел
      // Номер узла должен быть записан в state->ProcIndex
      void SendArc(int procIndex);
private:
      // Принять очередное сообщение, пришедшее на текущий узел.
      // Возвращает тег принятого сообщения
      bool Receive(int* tag);
private:
      // Обработчик полученного сообщения
      void ReceiveMsgHandler(int tag, const BaseFSM& baseFSM);
private:
      // Обработчик, вызываемый при получении сообщения MPI_ARC_TAG
      void RecvArcHandler(const BaseFSM& baseFSM);
private:
      // Обработчик, вызываемый при пролучении сообщения MPI_REC_TAG
      void RecvRecHandler(void);
private:
      // Обработчик, вызываемый при пролучении сообщения MPI_REC_SND
      void RecvSndHandler(void);
private:
      // Обработчик, вызываемый при пролучении сообщения MPI_REC_TRM
      void RecvTrmHandler(void);

      // . . .
};
```

4.7.3 Диаграмма классов

Ниже приводится диаграмма вышеописанных классов.

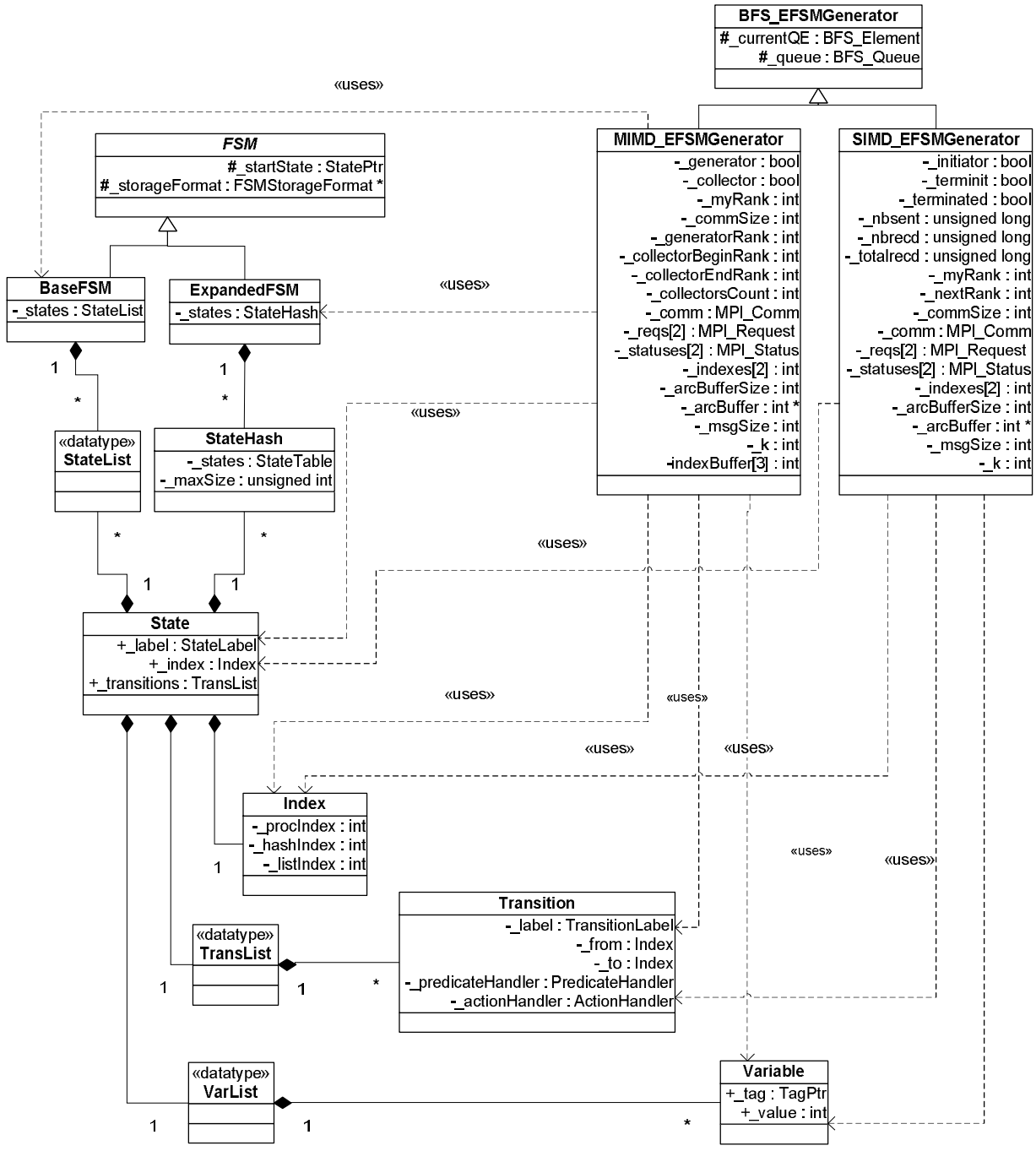


Рисунок 4-1. Диаграмма основных классов

4.8 Взаимодействие с библиотекой MPI на примере процесса-генератора

В разработанных алгоритмах параллельной генерации множества состояний ***GeneratorCollectors*** и ***Distributor*** для все взаимодействия между процессами выполняются посредством вызовов функций библиотеки MPI. Рассмотрим взаимодействие процесса с библиотекой MPI на примере процесса-генератора из алгоритма ***GeneratorCollectors*** (см. рис. 4-2). Для описания подобного взаимодействия подходит диаграмма последовательностей UML. Для процессов-коллекторов, а также для алгоритма ***Distributor*** диаграммы будут выглядеть аналогично, и здесь не приводятся в виду большого размера.

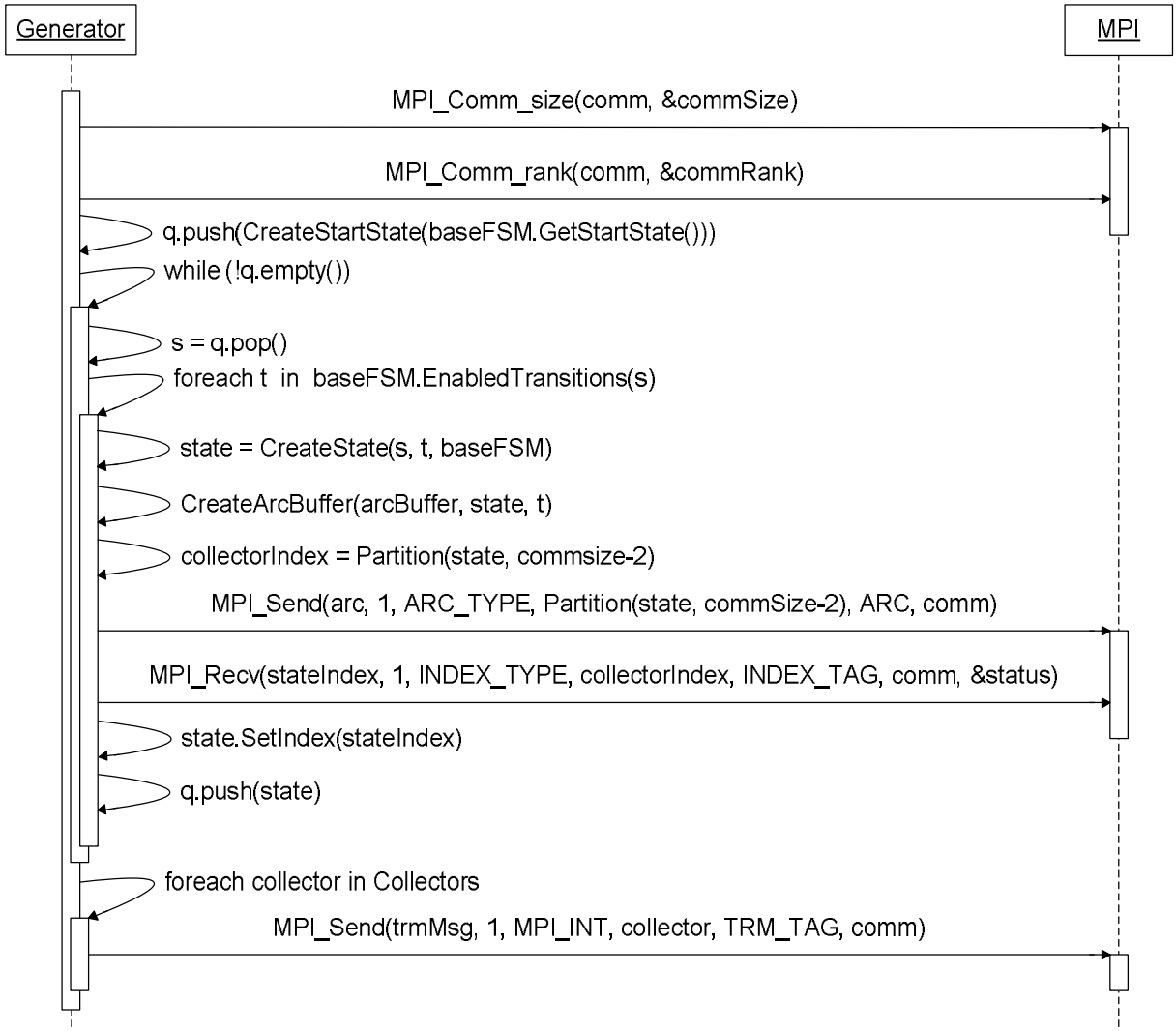


Рисунок 4-2. Взаимодействие процесса-генератора с библиотекой MPI

4.9 Тестирование разработанного программного обеспечения

В целях тестирования программы были произведены модульное, интеграционное и системное тестирование. Модульное тестирование — тестирование на уровне отдельно взятых модулей, функций или классов. Интеграционное тестирование — тестирование, при котором отдельные программные модули объединяются и тестируются в группе. Системное тестирование программного обеспечения — это тестирование программного обеспечения, выполняемое на полной, интегрированной системе, с целью проверки соответствия системы исходным требованиям.

При тестировании разработанного программного обеспечения возникли следующие трудности:

1. Значительный объем выходной информации затрудняет тестирование, в том числе и автоматическое. Поэтому для тестирования использовались БКА и РКА с небольшим числом состояний, а также введение утверждений (assert).
2. Параллельное выполнение процессов также приводит к сложностям при тестировании. Наиболее эффективным в данном случае оказалось системное тестирование взаимодействия небольшого количества процессов с обязательным использованием утверждений (assert).

Для модульного и интеграционного тестирования был написан класс **Tester**, создающий различные экземпляры классов (например, состояния и переменные), а также генерирующий РКА с небольшим числом состояний, после чего правильность полученных данных проверялась посредством анализа выводимой информации, либо при необходимости, вводились утверждения.

Учитывая сложность тестирования из-за большого объема информации, выводимой при в качестве результата работы программы при большом числе состояний, а также изменчивость результатов зависимости от количества процессов, было решено при системном тестировании анализировать небольшой объем выходной информации, не зависящей от количества процессов. Такими данными является количество состояний сгенерированного РКА. Эта величина вычислялась сначала при генерации на одном узле, а затем с полученным значением сравнивались результаты генерации на числе узлов больше одного.

5 Исследовательский раздел

Целью исследований является выявление временных характеристик разработанных алгоритмов для выбора наиболее эффективного из них для генерации конкретного тестового РКА, а также. Для исследований использовался тестовый БКА, изображенный на рис. 5-1.

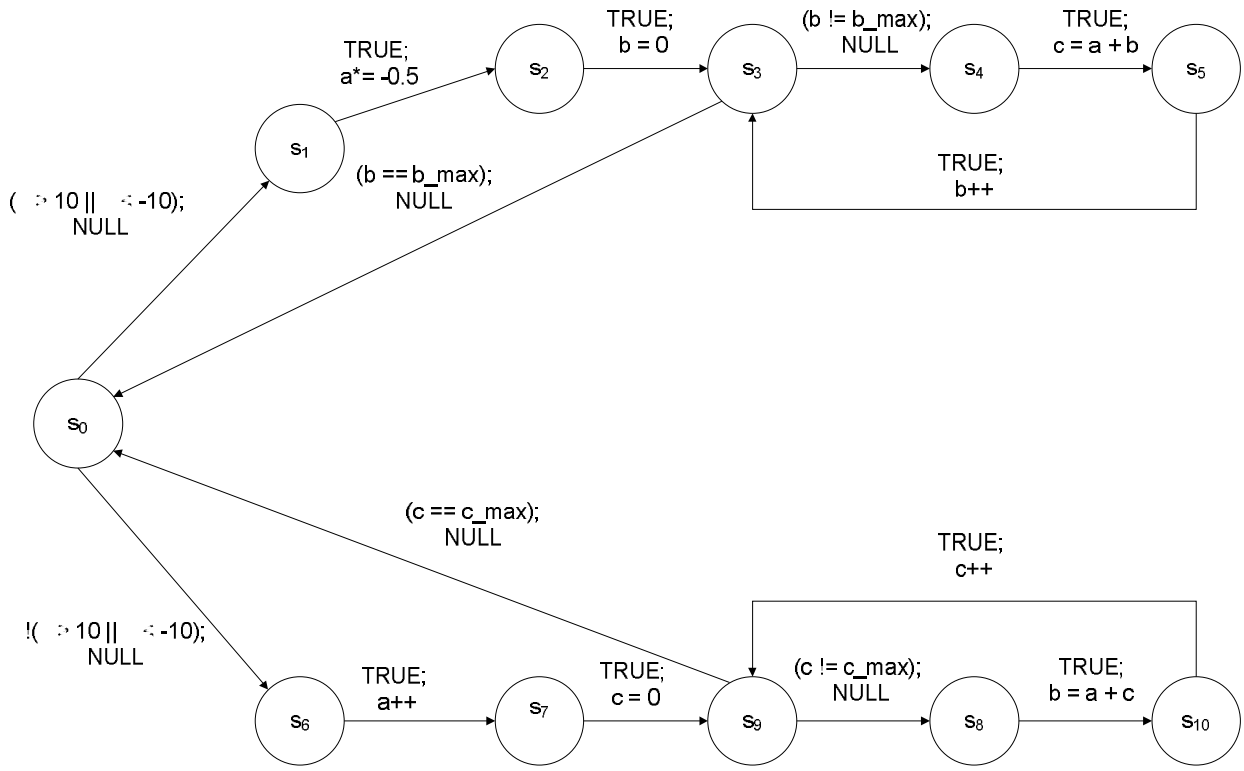


Рисунок 5-1. Тестовый базовый конечный автомат

Рассмотрим тестовый БКА подробнее. В данном БКА имеется четыре цикла (два по верхней ветви и два – по нижней). Меняя допустимые диапазоны изменения переменных a, b, c можно уменьшать или увеличивать количество состояний генерируемого РКА. В результате проведенных экспериментов выяснилось, что увеличение диапазона значение переменной a оказывает достаточно слабое влияние на количество состояний генерируемого РКА по сравнению с изменением диапазонов изменений переменных b и c. Поэтому далее при проведении экспериментов изменялись только диапазоны значений переменных b и c (для этих целей достаточно изменить значения переменных b_max и c_max), причем, для простоты диапазоны всегда задавались одинаковыми для обеих переменных (например, $b_{max} = 100$, $c\_max = 100$, что дает 5364 состояния РКА).

Используемые при проведении экспериментов значения переменных b_max и c_max и соответствующие им количества состояний РКА приведены в таблице ниже.

b_max	200	400	600	800	1000	1200	1400	1600	1800	2000
c_max	200	400	600	800	1000	1200	1400	1600	1800	2000
Число состояний РКА	10664	21264	31864	42464	53064	63664	74264	84864	95464	106064

Таблица 5-1. Количество состояний РКА в зависимости от b_max и c_max

Для наглядности эта зависимость приведена в виде графика на рис. 5-2.

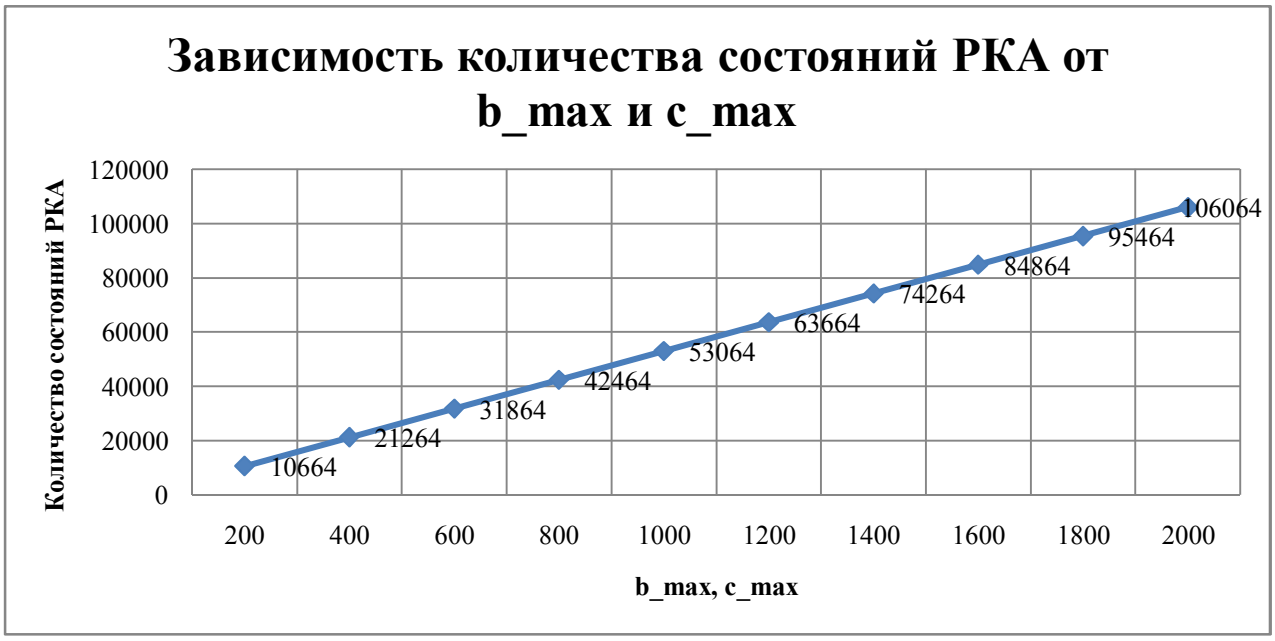


Рисунок 5-2. Зависимость количества состояний РКА от b_max и c_max

5.1 Исследование зависимости времени генерации РКА от размера хеш-таблицы

Ниже приводятся графики зависимости времени генерации РКА с использованием алгоритма ***GeneratorCollectors*** в зависимости от количества состояний при различном количестве узлов и при фиксированном размере хеш-таблицы, равном 7 (см. таблицу 5-2). Графически эта зависимость представлена на рис. 5-3. Здесь и далее время генерации всегда измеряется в секундах.

Несмотря на то, что генерацией состояний занимается всего лишь один процесс-генератор, на графике видно, что с ростом количества узлов время генерации существенно уменьшается в сравнении с генерацией состояний на одном узле. Это связано с тем, что в

увеличением количества узлов увеличивается размер хеш-таблицы РКА. Пусть размер хеш-таблицы на каждом узле равен m ячеек. Тогда при генерации состояний на одном узле размер хеш-таблицы равен m, и при числе состояний N, таком, что $N \gg m$, число коллизий будет очень велико, соответственно увеличивается время поиска состояний во множестве состояний РКА.

b_max, c_max / число узлов	10 (594)	50 (2714)	100 (5364)	150 (8014)	200 (10664)	250 (13314)	300 (15964)	400 (21264)	500 (26564)
1	3	57	215	467	839	1329	1873	3351	5205
4	1	22	86	176	310	499	661	1286	1851
8	3	60	240	528	904	1452	1985	3807	6087
16	1	9	26	30	82	116	158	295	431
24	1	7	21	40	58	85	115	206	309
32	1	7	20	35	51	70	97	155	277

Таблица 5-2. Зависимость времени генерации (в секундах) от числа состояний при малом размере хеш-таблицы (в скобках указано количество состояний при заданных b_max и c_max)

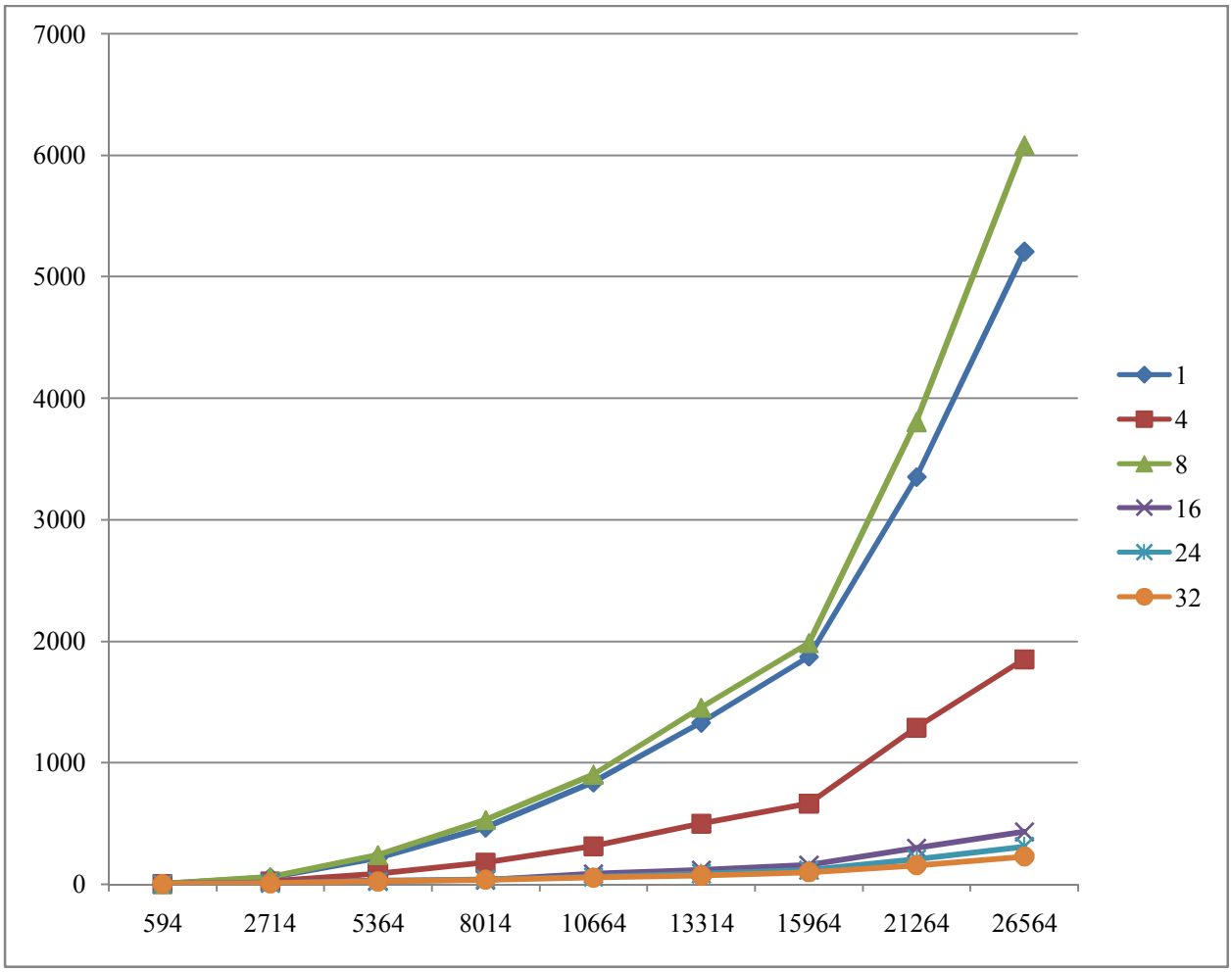


Рисунок 5-3. Зависимость времени генерации от числа узлов при малом размере хеш-таблицы

При генерации и хранении состояний на k $(k > 1)$ узлах общий размер хеш-таблицы РКА будет равен $k * m$, что при использовании равномерной хеш-функции распределения состояний по узлам приводит к значительному уменьшению числа коллизий, и, следовательно, к снижению времени на поиск состояний во множестве состояний РКА.

Приведенные графики также позволяют сделать вывод о том, что поиск состояний во множестве состояний РКА является одной из наиболее затратных операций в реализованных алгоритмах.

Отметим, что при числе узлов, равном 8, эффективность работы ниже чем даже при генерации на одном узле. Это связано с особенностями работы хеш-функции, и при 8 узлах, на каждом узле все состояния хешируются в одну и ту же ячейку хеш-таблицы, номер которой совпадает с номером узла.

Результаты проведенных экспериментов показывают, что увеличение размера хеш-таблицы приводит к значительному уменьшению времени генерации. Поэтому решено было исследовать зависимость времени генерации от размера хеш-таблицы на одном узле. Для этого были выбраны значения b_max=300 и c_max=300, что дает нам РКА с 15964 состояниями. Ниже представлены результаты этого исследования в виде таблицы и в графическом виде.

Размер хеш-таблицы	Время генерации
7	5205
28	1359
56	667
112	351
224	183
448	105
896	65
1792	45
2048	43

Таблица 5-3. Время генерации (в секундах) в зависимости от размера хеш-таблицы

Из графика на рис. 5-4 следует, что увеличение размера хеш-таблицы приводит к существенному снижению временных затрат на генерацию большого числа состояний. В дальнейшем для исследований зависимости времени генерации РКА от количества узлов кластера и числа состояний РКА использовалась хеш-таблица размером 4097 ячеек.

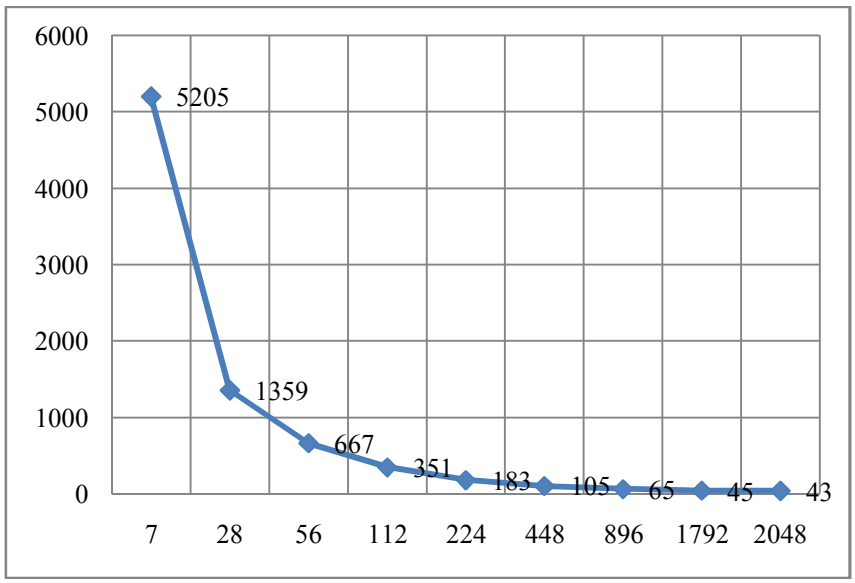


Рисунок 5-4. Зависимость времени генерации от размера хеш-таблицы (число состояний равно 15964)

5.2 Исследование хеш-функции распределения состояний по ячейкам хеш-таблицы и по узлам кластера

Основным требованием к хеш-функции, осуществляющей распределение состояний по ячейкам хеш-таблицы и по узлам кластера является равномерность. Для исследования равномерности используемой в работе хеш-функции было исследовано распределение 15964 состояний по ячейкам хеш-таблицы размером 256 на одном узле (см. рис. 5-5) и по 16 узлам (см. рис. 5-6).

Из приведенной на рис. 5-5 диаграммы видно, что состояния распределены по ячейкам хеш-таблицы достаточно равномерно в данном случае. Это позволяет предположить, что при использовании выбранной хеш-функции подобное распределение также будет иметь место и в случае других размеров хеш-таблицы и при другом количестве состояний.

Из рисунка 5-6 видно, что выбранная хеш-функция Partition также обеспечивает относительно равномерное распределение состояний по узлам кластера в данном конкретном случае, что позволяет предположить, что аналогичное распределение будет иметь место и с большим числом узлом кластера и состояний РКА.

При этом следует учитывать, что данные результаты получены для конкретного тестового БКА, и для других БКА выбранная хеш-функция может быть менее эффективной. Вообще, возможно, что наиболее эффективным будет тщательный выбор хеш-функции и размера хеш-таблицы для каждого конкретного случая.

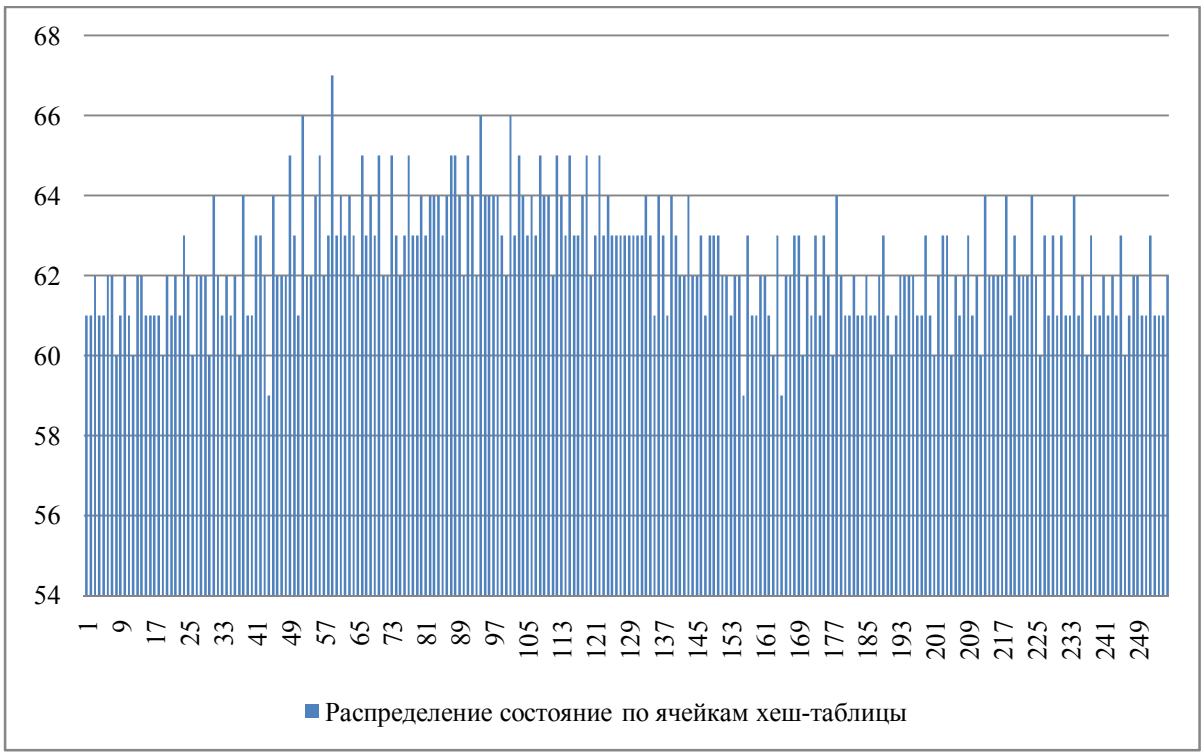


Рисунок 5-5. Диаграмма распределения состояний по ячейкам хеш-таблицы размером 256 ячеек

Рисунок 5-6. Распределение состояний по 16 узлам кластера

5.3 *Исследование зависимости времени генерации РКА от количества узлов кластера*

Для исследований зависимости времени генерации РКА от количества узлов кластера использовались значения b_max и c_max, приведенные в таблице 5-1. Приведем ее еще раз для удобства (см. таблицу 5-4).

b_max	200	400	600	800	1000	1200	1400	1600	1800	2000
c_max	200	400	600	800	1000	1200	1400	1600	1800	2000
Число состояний РКА	10664	21264	31864	42464	53064	63664	74264	84864	95464	106064

Таблица 5-4. Значения b_max и c_max и соответствующие им количества состояний РКА, используемые для исследований

Исследования проводились на 1, 4, 8, 16, 24 и 32 узлах кластера.

5.3.1 Исследование зависимости времени генерации РКА от количества узлов кластера с использованием алгоритма GeneratorCollectors

Для алгоритма ***GeneratorCollectors*** производительность мало зависит от числа узлов, поскольку генерацию осуществляет только один процесс. Поэтому влияние на производительность оказывает в первую очередь число генерируемых состояний, и, в меньшей степени, накладные расходы на передачу состояний. Зависимость времени генерации от числа состояний на 4 узлах приводится на рисунке ниже:

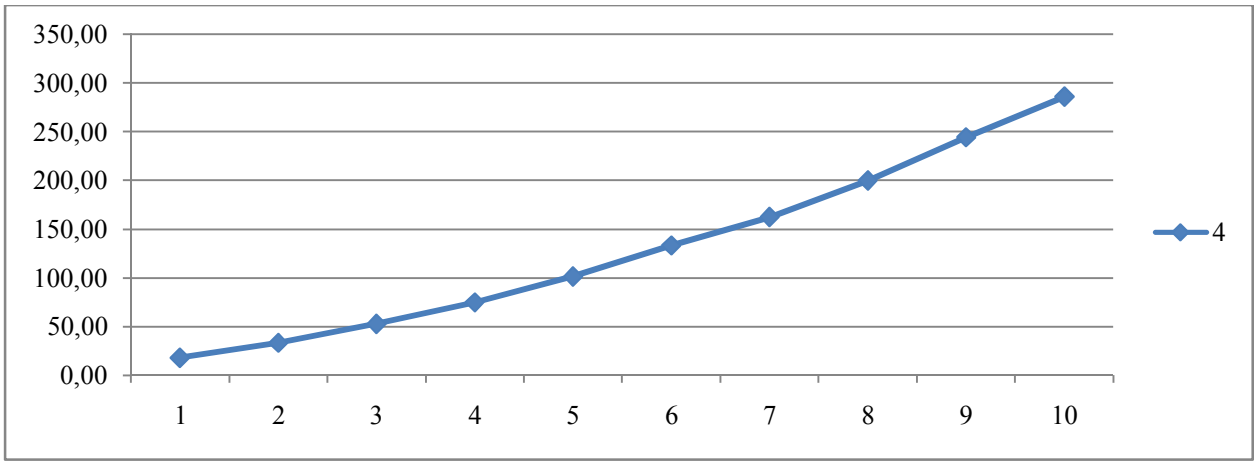


Рисунок 5-7. Зависимость времени генерации от количесвта состояний на 4 узлах

Таким образом, использовать алгоритм ***GeneratorCollectors*** имеет смысл только в случае острой нехватки памяти одного узла, в противном случае следует обойтись генерацией на одном узле.

5.3.2 Исследование зависимости времени генерации РКА от количества узлов кластера с использованием алгоритма Distributor

Теоретически, алгоритм ***Distributor*** обладает рядом преимуществ перед алгоритмом ***GeneratorCollectors***. Во-первых, в данном алгоритме генерация состояний производится параллельно несколькими процессами, а во-вторых, среднее количество вызовов функций библиотеки MPI в нем ниже. В алгоритме ***Distributor*** на каждой итерации в худшем случае происходит вызов трех функций MPI (отправка состояния, проверка прихода состояния или сообщения, отправка сообщения), но в лучшем случае вообще не выполняется ни одного вызова MPI. В алгоритме ***GeneratorCollectors*** на каждой итерации гарантировано выполняются два MPI-вызова. Поэтому, в теории, алгоритм ***Distributor*** выполнять генерацию состояний намного быстрее, чем алгоритм ***GeneratorCollectors***.

Ниже приводятся результаты исследования зависимости времени генерации РКА от числа состояний РКА и количества узлов. Результаты приведены как в табличном (см. таблицу 5-5), так и в графическом виде (см. рис. 5-8).

ЧС / ЧУ	10664	21264	31864	42464	53064	63664	74264	84864	95464	106064
1	17,33	32,00	50,67	72,00	97,33	128,00	156,00	192,00	234,67	274,67
4	12,50	22,00	32,50	43,00	54,00	64,50	75,50	92,00	102,00	124,00
8	6,50	11,75	17,00	22,50	27,75	33,50	38,75	44,75	52,00	63,25
16	3,38	6,25	9,25	12,38	15,38	22,38	21,13	23,38	28,38	32,50
24	2,50	4,17	6,50	8,75	10,67	12,58	14,50	16,58	18,25	21,83
32	1,75	3,56	5,06	7,13	8,69	10,13	11,56	13,31	14,75	17,13

Таблица 5-5. Зависимость времени генерации от количества узлов и числа состояний

По результатам исследований видно, что алгоритм ***Distributor*** значительно эффективнее алгоритма ***GeneratorCollectors***. Использование алгоритма ***Distributor*** на более чем одном узле при большом числе состояний приводит к значительному снижению временных затрат на генерацию РКА. Вместе с тем, очевидно, что для данного случая использование более 16 узлов для параллельной генерации узлов не приводит к значительному снижению временных затрат на генерацию. Таким образом, при использовании алгоритма ***Distributor*** желательно, чтобы число узлов, задействованных в

генерации РКА, было оптимальным, в противном случае временные расходы на обмен данными между узлами могут быть достаточно высоки в сравнении с выигрышем времени от параллельной генерации состояний. Исключением являются случаи, когда оптимальное число узлов не дает необходимого объема памяти для хранения РКА.

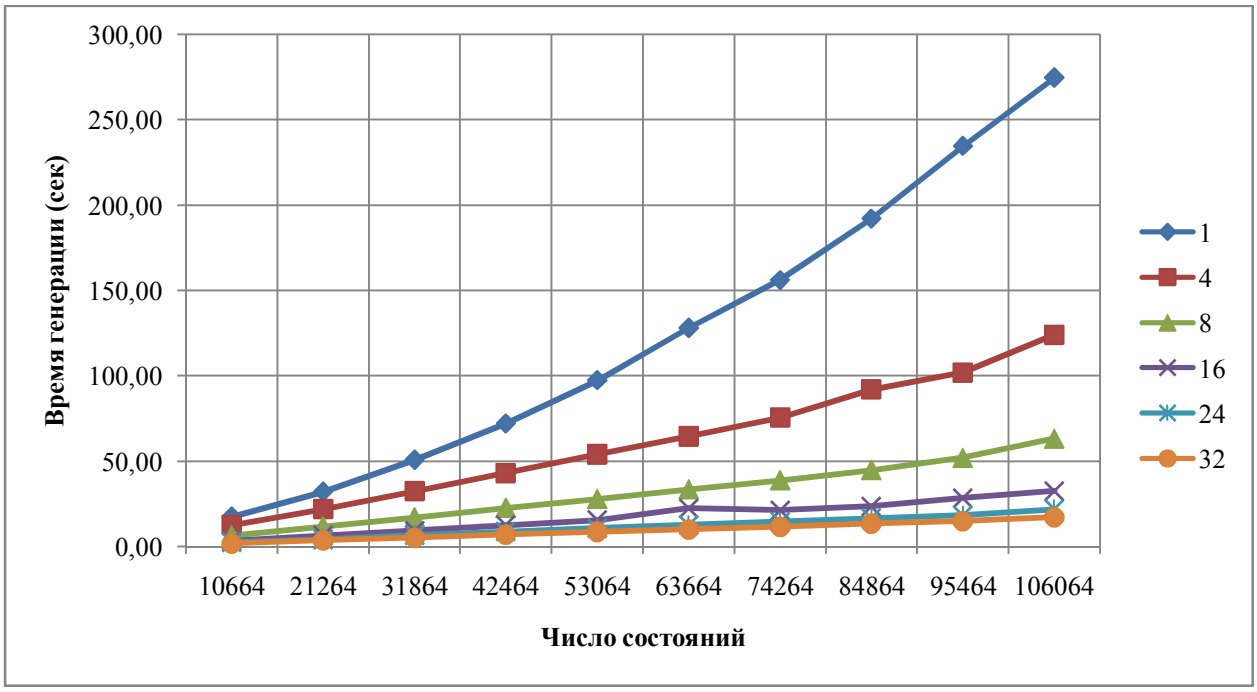


Рисунок 5-8. Зависимость времени генерации от количества узлов и числа состояний

В заключение приведем в табличном виде значения ускорения и эффективности генерации РКА в зависимости от числа узлов.

Ускорение решения задачи вычисляется по формуле:

$$A(n) = t(1)/t(n),$$

где A – ускорение решения задачи, $t(n)$ – время решения задачи n процессорами. Ускорение приведено в табличном виде (см. таблицу 5-6).

	10664	21264	31864	42464	53064	63664	74264	84864	95464	106064
1	1	1	1	1	1	1	1	1	1	1
4	0,7211	0,6875	0,6414	0,5972	0,5547	0,5039	0,4839	0,4791	0,4346	0,4514
8	0,375	0,3671	0,3355	0,3125	0,2851	0,2617	0,2483	0,2330	0,2215	0,2302
16	0,1947	0,1953	0,1825	0,1718	0,1579	0,1748	0,1354	0,1217	0,1209	0,1183
24	0,1442	0,13	0,1282	0,1215	0,1095	0,0983	0,0929	0,0863	0,0777	0,0794
32	0,1009	0,111	0,0999	0,0989	0,0892	0,0791	0,0741	0,0693	0,0628	0,0623

Таблица 5-6. Зависимость ускорения генерации РКА от числа узлов

Используя полученные значения ускорений можно получить эффективность решения задачи по формуле:

$$E(n) = A(n)/n,$$

где E – эффективность, $A(n)$ – ускорение, n – число используемых процессоров.

Эффективность генерации РКА с использованием алгоритма Distributor приведена в таблице 5-7.

	10664	21264	31864	42464	53064	63664	74264	84864	95464	106064
1	1	1	1	1	1	1	1	1	1	1
4	0,1802	0,1718	0,1603	0,1493	0,1386	0,1259	0,1209	0,1197	0,1086	0,1128
8	0,0468	0,0458	0,0419	0,0390	0,0356	0,0327	0,0310	0,0291	0,0276	0,0287
16	0,0121	0,0122	0,0114	0,0107	0,0098	0,0109	0,0084	0,0076	0,0075	0,0073
24	0,0060	0,0054	0,0053	0,0050	0,0045	0,0040	0,0038	0,0035	0,0032	0,0033
32	0,003	0,003	0,003	0,003	0,002	0,002	0,00231	0,0022	0,002	0,002

Таблица 5-7. Эффективность генерации РКА в зависимости от числа узлов

6 Заключение и выводы

В результате выполнения работы:

- изучены основные подходы к построению конечных автоматов для проверки темпоральных свойств моделей с конечным числом состояний;
- изучены проблемы возникающие при генерации конечных автоматов моделей с большим числом состояний, и предложен способ решения этих пробел путем хранения множества состояний конечного автомата модели на нескольких узлах вычислительной системы с разделенной памятью;
- изучены основные методы хеширования;
- разработаны структуры данных для хранения множества состояний конечных автоматов моделей на нескольких узлах вычислительной системы;
- разработаны и реализованы два алгоритма, осуществляющие распределенную генерацию и хранение расширенного конечного автомата модели;
- исследованы временные характеристики реализованной системы.

Проведенные исследования показывают, что распределение состояний по узлам кластера может быть достаточно равномерным, а накладные расходы на взаимодействие узлов – низкими, чтобы считать использование кластеров для поставленной задачи достаточно эффективным.

7 Список использованной литературы

1. J-P Katoen. Concepts, algorithms, and tools for model checking. Lecture notes of the course “Mechanised Validation of Parallel Systems”, 1998-1999.
2. F. Lerda, R. Sisto. Distributed-memory model checking with SPIN.
3. H. Garavel, R. Mateescu, I. Smarandache. Parallel state space construction for model checking, 2001.
4. F. Mattern. Algorithms for distributed termination detection, 1987.
5. W.M. Zuberek, I. Rada. Modeling and analysis of distributed state space generation for timed Petri nets.
6. G. Ciardo, J. Gluckman, D. Nicol. Distributed state space generation of discrete-state stochastic models.
7. G.J. Holzmann. Spin model checker, The: Primer and Reference Manual, Addison Wesley, 2003.
8. Э.М. Кларк, О. Грамберг, Д. Пелед. Верификация моделей программ: Model Checking, МЦНМО, 2002.
9. А.С.Антонов. Параллельное программирование с использованием технологии MPI, Изд-во МГУ, 2004.
10. Э.С. Реймонд. Искусство программирования для Unix, Вильямс, 2005
11. Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. Алогритмы: построение и анализ, 2-ое издание, Издательский дом "Вильямс", 2005.
12. А.И. Белоусов, СБ. Ткачев. Дискретная математика, Издательство МГТУ им. Н.Э. Баумана, 2004.
13. В.В. Воеводин, Вл. В. Воеводин. Параллельные вычисления, Издательство «БХВ-Петербург», 2004.
14. Стандарт MPI 1.1. http://www.mpi-forum.org