В.Н. Касьянов, И.Л. Мирзуитова

УПОРЯДОЧЕННЫЕ ДИАГРАММЫ БИНАРНЫХ РЕШЕНИЙ*

ВВЕДЕНИЕ

Многие проблемы в таких областях, как проектирование цифровых систем, комбинаторная оптимизация, математическая логика и искусственный интеллект, могут быть сформулированы в терминах операций над небольшими конечными областями. Посредством введения бинарного кодирования элементов этих областей, упомянутые проблемы могут быть сведены к операциям над булевыми значениями. Используя символьное представление булевых функций, мы можем выражать проблемы в обобщенной форме. Решение такой обобщенной проблемы с помощью символьной обработки булевых функций дает возможность разрешить большое количество конкретных частных случаев. Следовательно, эффективный метод представления и обработки булевых функций может привести к решению большого класса сложных проблем.

Упорядоченные бинарные диаграммы решений (OBDD) представляют булевы функции в виде ориентированных ациклических графов. Они образуют каноническое представление, с помощью которого проверка таких функциональных характеристик, как выполнимость и эквивалентность, может быть произведена простым образом. Множество операций на булевых функциях может быть реализовано в виде графовых алгоритмов на OBDD-структурах данных. Используя упорядоченные бинарные диаграммы решений, мы можем решать проблемы различных классов с помощью *символьного анализа,* если все возможные вариации параметров системы и технических условий можно закодировать с помощью булевых переменных.

Данная работа представляет собой обзор символьных булевых операций с упорядоченными бинарными диаграммами решений. Описываются OBDD-диаграммы и алгоритмы их построения и обработки. Рассматриваются некоторые базовые техники для представления и обработки таких математических структур, как функции, множества и отношения, с помощью символьных булевых операций. Эти техники проиллюстрированы описани-

* Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (грант № 01-01-794) и Министерства образования РФ.

ем некоторых приложений, использующих OBDD, а также рассмотрены возможные направления дальнейших исследований. Хотя большая часть описанных приложений относится к области проектирования цифровых систем, похожие методы могут быть применены и в других областях.

1. УПОРЯДОЧЕННЫЕ ДИАГРАММЫ БИНАРНЫХ РЕШЕНИЙ И ЛОГИЧЕСКИЕ ФРАГМЕНТЫ

Бинарная диаграмма решений (BDD) представляет булеву функцию в виде корневого ориентированного ациклического графа — дэга с одной выделенной вершиной, называемой корнем, которая не имеет предшественников и из которой достижимы все его вершины.

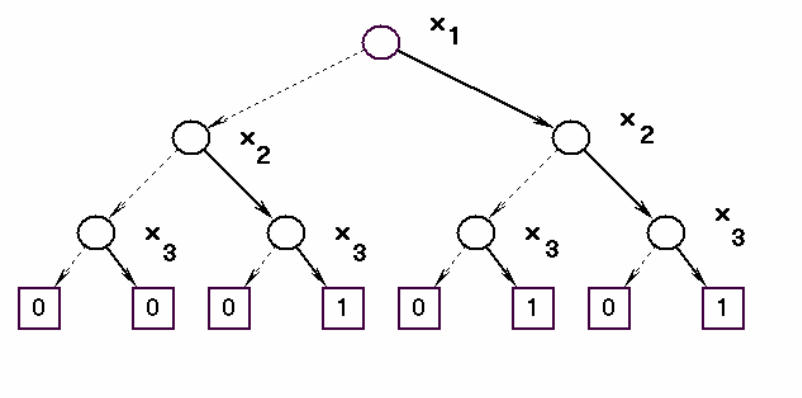

Рис. 1. Бинарная диаграмма решений булевой функции *f* (табл. 1), являющейся деревом. Пунктирные и сплошные дуги соответствуют случаям, когда значение разрешающей переменной равно 0 или 1 соответственно

На рис. 1 показан пример представления функции $f(x_1,x_2,x_3)$, определяемой табл. 1, когда указанный граф является деревом. Каждая его нетерминальная (не висячая) вершина графа v помечена переменной ПЕР(v) и имеет две исходящих дуги к двум сыновьям: ЛЕВ(v) (на рисунке показана пунктирной линией) в случае, когда переменной присваивается значение 0, и ПРАВ(v) (показана сплошной линией) в случае, когда переменной присваивается значение 1. Каждая терминальная (висячая) вершина графа помечена либо 0, либо 1. Для заданного распределения значений истинности логических переменных значение функции, реализованной диаграммой, определя-

ется с помощью продвижения по пути от корня к терминальной вершине по ветвям, соответствующим присваиваемым переменным значениям. В качестве значения функции берется метка найденной терминальной вершины. Дуги на данном рисунке упорядочены таким образом, что значения терминальных вершин (слева направо) соответствуют их вхождениям в таблицу истинности (сверху вниз).

Таблица 1

Табличное задание функции f

X_1	X_2	X_3	F
0	0	0	0
0	0	1	0
0	0	0	0
0	1	1	1
1	1	0	0
1	0	1	1
1	0	0	0
1	1	1	1

В случае упорядоченной бинарной диаграммы решений (OBDD) рассматривается некоторый линейный порядок < на множество переменных и требуется, чтобы для любой вершины *u* и любого ее преемника *v,* не являющегося терминальной вершиной, соответствующие им переменные были упорядочены: ПЕР*(u)* < ПЕР*(v)*. К примеру, в дереве решений на рис. 1 переменные упорядочены следующим образом: $x_1 < x_2 < x_3$.

Теоретически порядок переменных может быть выбран произвольно — алгоритмы будут работать корректно при любом выборе порядка. Но на практике выбор подходящего упорядочения является критичным для эффективности символьной обработки. Этот вопрос подробнее рассматривается ниже.

Определяются три правила преобразования описанных графов, не изменяющих значение представляемой функции.

Удаление дублирующих терминальных переменных. Удаляются все терминальные вершины с заданной меткой, кроме одной, а все дуги, ведущие к удаленным вершинам, перенаправляются в оставшуюся вершину.

Удаление дублирующих нетерминальных вершин. Если для нетерминальных вершин *u* и *v* выполняются следующие условия: ПЕР(*u*)=ПЕР(*v*), ЛЕВ(*u*)=ЛЕВ(*v*) и ПРАВ(*u*)=ПРАВ(*v*), то одна из этих вершин удаляется, а все дуги, входящие в нее, перенаправляются во вторую вершину.

Удаление избыточных проверок Если для нетерминальной вершины *v* верно ЛЕВ(*v*)=ПРАВ(*v*), то *v* удаляется, а все дуги, входящие в нее, перенаправляются в ЛЕВ(*v*).

Начиная с любой бинарной диаграммы решений, обладающей порядком, можно пытаться сократить ее размер путем последовательного применения правил преобразования. Термин «OBDD» используется для обозначения максимально приведенного графа, обладающего некоторым фиксированным порядком на множестве переменных. Пример на рис. 2 иллюстрирует процесс приведения дерева решений, представленного на рис. 1, к OBDD. Применение первого правила преобразования сокращает количество терминальных вершин с 8 до 2. Применение второго правила удаляет две вершины с переменной x_3 и дуги к терминальным вершинам с метками 0 (ЛЕВ) и 1 (ПРАВ). Применение третьего правила удаляет две вершины: одну с переменной x_3 и одну с переменной x_2. В общем случае правила преобразования должны применяться неоднократно, поскольку каждое применение преобразования может привести к появлению новых возможностей для других преобразований.

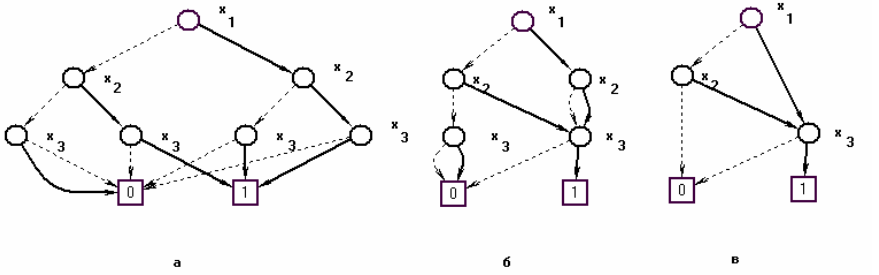

Рис. 2. Приведение дерева решений к OBDD.
Применение трех правил преобразования к дереву на рис. 1 дает в результате каноническое представление функции в виде OBDD: удаление дублирующих терминальных переменных (а), удаление дублирующих нетерминальных переменны (б), удаление избыточных проверок (в)

Представление функции в виде OBDD является универсальным и каноническим.

Свойство А. *Любая булева функция при любом заданном порядке на переменных имеет OBDD-представление.*

Свойство Б. *Для любого заданного порядка на переменных две упорядоченные бинарные диаграммы решений одной и той же функции изоморфны.*

Из этого вытекают несколько важных следствий. Легко может быть проверена функциональная эквивалентность. Функция является выполнимой в том и только том случае, когда ее OBDD-представление не сводится к одиночной терминальной вершине с меткой 0. Любая тавтологическая функция должна иметь терминальную вершину с меткой 1 в качестве своего OBDD-представления. Если функция не зависит от переменной *x,* то ее OBDD-представление не может иметь вершин, помеченных *x*. Таким образом, как только построены представления функций в виде OBDD, многие их характеристики становятся легко проверяемыми.

Как демонстрируют рис. 1 и 2, можно строить OBDD-представление для функции, заданной таблично, построив и затем сократив ее дерево решений. Однако этот подход является практичным только для функций с небольшим количеством переменных, поскольку и таблица истинности, и дерево решений имеют экспоненциальный по отношению к числу переменных размер. Поэтому OBDD обычно строятся при помощи «символьного выполнения» логического выражения или сети логических элементов с применением операции APPLY, описанной в п. 1.2.

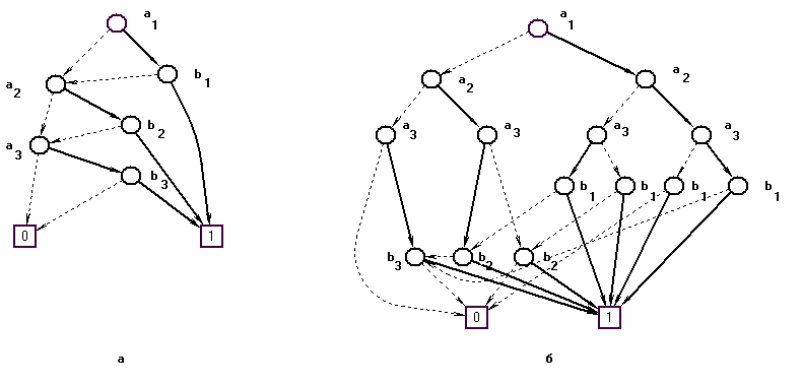

Рис. 3. OBDD-представления одной и той же функции для двух различных способов упорядочения переменных

Форма и размер OBDD, представляющей функцию, зависит от способа упорядочения переменных. На рис. 3 приведен пример двух OBDD-представлений функции, заданной булевым выражением

$$a_1*b_1 + a_2*b_2 + a_3*b_3,$$

где «*» обозначает операцию AND, а «+» — операцию OR. Для примера на рис. 3,а был задан следующий порядок переменных:

$$a_1 < b_1 < a_2 < b_2 < a_3 < b_3,$$

тогда как на рис. 3, б порядок был таким:

$$a_1 < a_2 < a_3 < b_1 < b_2 < b_3.$$

Можно обобщить эту функцию следующим образом:

$$a_1*b_1 + a_2*b_2 + \ldots + a_n*b_n.$$

Обобщение первого способа упорядочения переменных до

$$a_1 < b_1 < \ldots < a_n < b_n$$

дает OBDD-представление с количеством нетерминальных вершин, равным $2n$. Обобщение второго способа до

$$a_1 < \ldots a_n < b_1 < \ldots < b_n,$$

с другой стороны, дает в результате OBDD-представление с $2(2^n - 1)$ числом нетерминальных вершин. Для больших значений n разница между линейным возрастанием первого представления и экспоненциальным ростом второго будет иметь существенное влияние на объем требуемой памяти и эффективность алгоритма обработки.

Исследуя структуру двух графов на рис. 3, мы видим, что в первом случае переменные объединены в пары в соответствии с их вхождением в булево выражение

$$a_1*b_1 + a_2*b_2 + a_3*b_3.$$

Таким образом, для любого второго уровня в графе имеются только два возможных пункта назначения исходящих ветвей: одна ведет к терминальной вершине, помеченной 1, в случае, когда соответствующее произведение дает 1; вторая ведет к следующему уровню в случае, если все произведения переменных до этого момента были равны 0. С другой стороны, все три уровня во втором случае образуют полное бинарное дерево, описывающее все возможные присваивания переменным a. В общем случае при каждом

присваивании переменным *a* значение функции единственным образом зависит от присваивания переменным *b*. Поскольку мы обобщаем эту функцию и порядок на более чем 2*n* переменных, первые *n* уровней OBDD образуют полное бинарное дерево.

Большинство приложений, использующих OBDD, выбирают некоторый порядок переменных с самого начала и строят все графы в соответствии с этим порядком. Поскольку проблема нахождения оптимального упорядочения переменных для OBDD является *NP*-трудной, порядок, как правило, выбирается либо вручную, либо с помощью эвристического анализа представляемой системы. Например, были разработаны некоторые эвристические методы, которые, получив на входе сеть логических элементов, обычно выводят хороший способ упорядочения переменных, представляющих первичные входные данные. Другие методы были разработаны для анализа последовательных систем. Заметим, что упомянутые эвристики не обязаны находить оптимальное упорядочение — выбор порядка не влияет на корректность результатов. Как только обнаружен порядок, не приводящий к экспоненциальному росту, операции на OBDD показывают приемлемую эффективность.

OBDD представляет собой практичный подход к символьной булевой обработке только в том случае, когда размер графа остается значительно меньше экспоненциального относительно количества переменных. Как показано в предыдущем примере, некоторые функции чувствительны к упорядочению переменных, но остаются вполне компактными, если выбран хороший порядок. Далее, имеется достаточный эмпирический опыт, чтобы утверждать, что многие функции из реальных приложений могут быть эффективно представлены в виде OBDD. Одним из способов более полно представить сильные и слабые стороны OBDD-представления может быть вывод нижних и верхних границ для важнейших классов булевых функций.

В табл. 2 сведены воедино асимптотические оценки степени возрастания для нескольких классов булевых функций и их чувствительность к упорядочению переменных. Симметричные функции, у которых значение функции зависит только от количества аргументов, равных 1, нечувствительны к порядку переменных. За исключением тривиального случая константных функций, графы симметричных функций попадают в промежуток между линейными (например, четность) и квадратичными (например, не менее половины входных данных равны 1).

Таблица 2

Сложность OBDD-представления для широко распространенных классов функций

Класс функций	Сложность	
	Лучший случай	Худший случай
Симметричные	Линейная	Квадратичная
Целочисленное сложение (любой бит)	Линейная	Экспоненциальная
Целочисленное умножение (средние биты)	Экспоненциальная	Экспоненциальная

Мы можем рассматривать каждое выходное значение n-битового сумматора как булеву функцию на переменных

$$a_0 , a_1 , ..., a_{n-1},$$

представляющих один операнд, и

$$b_0 , b_1 , ..., b_{n-1},$$

представляющих второй операнд. Функция для любого бита имеет OBDD-представление линейной сложности для порядка

$$a_0 < b_0 < a_1 < b_1 < ... < a_{n-1} < b_{n-1}$$

и экспоненциальной сложности для порядка

$$a_0 < a_1 < ... < a_{n-1} < b_0 < b_1 < ... < b_{n-1}.$$

Фактически представление таких функций похоже на представление функции, изображенной на рис. 3.

Булевы функции, представляющие целочисленное умножение, являют собой особенно сложный случай для OBDD.

Свойство В. *Вне зависимости от способа упорядочения булева функция, представляющая любое из двух средних выходных значений n-битного умножителя, имеет экспоненциальное OBDD-представление.*

Верхние границы для других классов булевых функций могут быть выведены на основе структурных свойств их логических сетей. Рассмотрим

сеть с *n* первичными входными значениями и одним первичным выходным значением, состоящую из m «логических блоков». Каждый блок может иметь несколько входов и выходов. Первичные данные представлены «исходными» блоками без входа и с одним выходом. В качестве примера на рис. 4 приведена сеть, дающая в качестве выходного значения самый важный бит суммы *n*-битного сумматора. Эта сеть состоит из цепи переноса, последнее значение которой вычисляется из c_{n-1}. Блоки, помеченные «2/3», вычисляют функцию MAJORITY, которая дает на выходе 1, если по меньшей мере два входных значения равны 1. Выходное значение вычисляется как EXCLUSIVE-OR самых важных битов входных значений и c_{n-1}.

Определим *линейное расположение* сети как нумерацию блоков от 1 до *m* таким образом, что блок, вычисляющий выходное значение, пронумерован последним. Определим *прямое поперечное сечение* в блоке *i* как общее количество проводов, ведущих из выхода блока *j (j < i)* ко входу блока *k (i ≤ k)*. Определим прямое поперечное сечение цепи w_f (относительно расположения) как максимальное прямое поперечное сечение среди всех блоков. Как видно на рис. 4, прямое поперечное сечение нашей цепи сумматора равно 3 (обозначено пунктирной линией). Подобным же образом определим *обратное поперечное сечение* в блоке i как общее количество проводов, ведущих из выхода блока *j (j > i)* ко входу блока *k (i ≥ k)*. В тех расположениях, где блоки упорядочены *топологически*, обратное поперечное сечение равно 0. На рис. 4 мы имеем именно такую картину. Определим прямое поперечное сечение цепи w_r (относительно расположения) как максимальное прямое поперечное сечение среди всех блоков. Можно показать, что существует OBDD-представление функции цепи с не более чем $n2^{w_f 2^{w_r}}$ вершинами. Более того, нахождение расположения с низкими поперечными сечениями означает нахождение хорошего способа упорядочения переменных функции — таким порядком будет инверсия порядка соответствующих исходных блоков в их расположении.

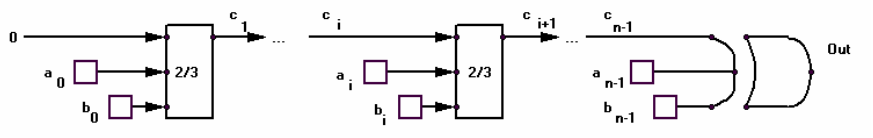

Рис. 4. Линейное расположение цепи,
вычисляющей самый главный бит целочисленного сложения

С помощью сетевых реализаций можно найти подходящие границы для различных булевых функций. Например, функции, имеющие реализацию с постоянным прямым поперечным сечением и нулевым обратным поперечным сечением (такие, как цепь сумматора на рис. 4), имеют линейное OBDD-представление. Симметричная функция от n переменных может быть реализована в виде цепи с прямым поперечным сечением 2 + log *n* и обратным поперечным сечением 0. Эта цепь состоит из серии каскадов, вычисляющих общее количество входных значений, равных 1, кодируя их в виде $[\log_2 n]$-битного двоичного числа. Эта реализация приводит к квадратичной верхней границе OBDD-представления, указанной в табл. 1.

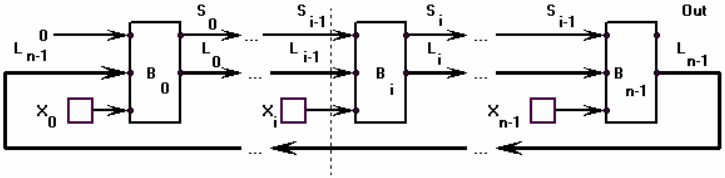

Рис. 5. Линейное расположение кольцевой цепи для *Within-K*. Пунктирной линией показано прямое поперечное сечение $2 + [\log_2 K]$ и обратное поперечное сечение $[\log_2 K]$

На рис. 5 изображено применение этого результата к цепи с ненулевым обратным поперечным сечением. Эта цепь представляет общую реализацию функции *Within-K*, где *K* — некоторая константа, такая, что $0 < K < n$. Для входных значений

$$x_0, x_1, \ldots, x_{n-1}$$

функция выдает значение 1 в случае, когда имеются входные значения x_i и $x_{i'}$, равные 1, причем $i' = i + j \bmod n$ для некоторого значения j, $0 < j < K$. Как показано на рис. 5, эта функция может быть вычислена с помощью серии блоков, расположенных в виде кольца, где каждый блок B_i выдает на выходе 1-битное значение s_i и k-битное целочисленное значение L_i, где $k = [\log_2 K]$:

$$s_i = \begin{cases} 1, \text{ если } x_i = 1 \text{ и } L_{i-1} \neq 0, \\ s_{i-1}, \text{ в противном случае,} \end{cases}$$

$$L_i = \begin{cases} K-1, \text{если } x_1 = 1, \\ L_{i-1} - 1, \text{ если } x_1 = 0 \text{ и } L_{i-1} > 0, \\ 0, \text{ в противном случае.} \end{cases}$$

В данной реализации каждый сигнал L_i кодирует количество оставшихся позиций, с которыми может составить пару самое последнее входное значение 1, в то время как каждый сигнал s_i показывает, встретилась ли уже пара входных значений, равных 1, внутри дистанции K. Чтобы удовлетворять условию близости модулей, выход L_{i-1} последнего каскада направляется на вход начального каскада. Заметим, что, хотя эта цепь имеет циклическую структуру, ее выходное значение однозначно определяется входными значениями. Как отмечено пунктирной линией, эта кольцевая структура может быть «сплющена» до линейного расположения с прямым поперечным сечением $k + 2$ и обратным поперечным сечением k. Эта конструкция дает верхнюю границу OBDD-представления, равную $(8K4^K)n$. Для константных значений K размер OBDD будет линейным, хотя постоянный коэффициент будет быстро возрастать с ростом K. Существует обобщение этой техники до *древовидного расположения*, в котором сеть организована в виде дерева логических блоков с коэффициентом ветвления b и первичным выходным значением, производимым корневым блоком. При таком расположении прямое поперечное сечение соответствует количеству проводов, направленных к корню (для обратного сечения — соответственно, из корня). Эта конструкция дает верхнюю границу размера OBDD-представления

$$n[2^b n^{b-1}]^{w_f 2^{w_r}}.$$

Верхняя граница для линейного расположения получается подстановкой $b = 1$ в эту формулу. Заметим, что для константных значений b, w_f и w_r размер OBDD будет полиномиальным относительно n.

Эти оценки верхней границы дают некоторое представление о том, какое количество функций, встречающихся в приложениях в области логического проектирования, имеют эффективные OBDD-представления. Они также позволяют предложить стратегию обнаружения хороших способов упорядочения переменных с помощью нахождения сетевых реализаций с низкими поперечными сечениями. Результаты, полученные для этой формы представления булевых функций, могут оказаться полезными при оценке потенциала OBDD-представления в других областях применения.

Есть целый ряд различных усовершенствований базовой структуры OBDD, дающих значительную экономию при выделении памяти — как правило, наиболее критичного ресурса при определении сложности решаемых проблем (приложения, требующие генерации более чем миллиона вершин OBDD, не так уж легко исполнить на обычных рабочих станциях). К ним относятся, например, использование единственного многокорневого

графа для представления всех требуемых функций, снабжение дуг метками для обозначения булева вычитания и обобщение понятия областей с конечным числом состояний.

2. КОНСТРУИРОВАНИЕ И МАНИПУЛЯЦИЯ

Введем систему обозначений для описания операций на булевых функциях. Мы будем использовать стандартные операции булевой алгебры: + для OR, · для AND, $\oplus$ для EXCLUSIVE-OR и верхний подчерк для NOT. Дополнительно мы будем использовать символ $\overline{\oplus}$, обозначающий дополнение к операции EXCLUSIVE-OR (иногда называемое EXCLUSIVE-NOR). Мы также используем понятия суммы (Σ) и произведения (Π), подразумевая под ними булеву сумму (OR) и произведение (AND). Заметим, что эти операции определяются над *функциями*, также как и над булевыми значениями 0 и 1. К примеру, если f и g — функции над некоторым множеством переменных, то $f + g$ также является функцией (скажем, h) над этими же переменными. Для некоторого присваивания значениям переменных $\vec{a}$, $h(\vec{a})$ выдает значение 1 в том и только том случае, когда либо $f(\vec{a})$, либо $g(\vec{a})$ выдает значение 1. Константные функции, значение которых равно 1 или 0 для любых присваиваний, обозначаются как **1** и **0** соответственно.

Функция, которая получается, когда некоторому аргументу x функции f присваивается константное значение k (0 либо 1), называется *рестрикцией* функции f (иногда ее также называют «кофактором» f) и обозначается $f|_{x\leftarrow k}$. Если даны две рестрикции функции относительно одной переменной, функция может быть реконструирована следующим образом:

$$f = \overline{x} \cdot f|_{x\leftarrow 0} + x \cdot f|_{x\leftarrow 1}.$$

Это тождество часто называют *Шенноновым разложением f* по переменной x, хотя впервые оно было упомянуто Булем.

Многие другие полезные операции могут быть определены в терминах алгебраических операций и операции рестрикции. Операция *композиции*, при которой функция g заменяет переменную x функции f, задается следующим тождеством:

$$f|_{x\leftarrow g} = \overline{g} \cdot f|_{x\leftarrow 0} + g \cdot f|_{x\leftarrow 1}.$$

Операция *квантификации переменной*, при которой некоторая переменная x экзистенциально или универсально оценивается относительно функции f, задается тождествами

$$\exists x\ f = f\big|_{x\leftarrow 0} + f\big|_{x\leftarrow 1},$$
$$\forall x\ f = f\big|_{x\leftarrow 0} \cdot f\big|_{x\leftarrow 1}.$$

Некоторые исследователи предпочитают называть эти операции *сглаживанием* (экзистенциальная квантификация) и *консенсусом* (универсальная квантификация), чтобы подчеркнуть тот факт, что они являются операциями над булевыми функциями, а не над истинностными значениями.

Большое количество символьных операций над булевыми функциями может быть реализовано в виде графовых алгоритмов, применяемых к упорядоченным бинарным диаграммам решений. Эти алгоритмы обладают важным свойством замыкания — если аргументы функции представляют собой OBDD с некоторым порядком, то результатом функции будет OBDD с тем же порядком. Следовательно, мы можем реализовать сложную обработку с помощью последовательности простых манипуляций, всегда работая с OBDD с привычным порядком. Пользователи могут рассматривать библиотеку подпрограмм обработки бинарных диаграмм решений как реализацию абстрактного типа данных — булевой функции. За исключением выбора способа упорядочения переменных, все операции реализуются полностью автоматически. Пользователю нет необходимости вдаваться в детали представления или реализации.

2.1. Операция APPLY

Операция APPLY генерирует булевы функции путем применения алгебраических операций к другим функциям. Если в качестве аргументов заданы функции f и g и бинарный булев оператор <op> (например, AND или OR), то функция APPLY выдает в результате функцию f <op> g. Эта операция является важнейшей для символьной булевой обработки. С ее помощью мы можем находить дополнение функции f, вычисляя $f \oplus 1$. Пусть даны функции f и g и состояние безразличности, выраженное функцией d (то есть $d(\vec{x})$ равно 1 для тех присваиваний переменным $\vec{x}$, для которых значения функции не важны), тогда мы можем проверить, являются ли f и g эквивалентными для всех «небезразличных» состояний, вычисляя значение $(f \overline{\oplus}\ g) + d$ и проверяя, равен ли результат функции **1**. Мы также можем построить OBDD-представление выходных функций сети комбинационно-логических элементов путем «символической интерпретации» сети. Это

производится следующим образом. Вначале мы представляем функцию на каждом первичном входе в виде OBDD, состоящего из проверки одной переменной. Затем в порядке, определяемом структурой сети, мы применяем операцию APPLY для построения OBDD-представления каждого выходного элемента в соответствии с операцией, выполняемой элементом, и OBDD, построенными для входных значений этого элемента.

Алгоритм APPLY производит обход графов аргументов в глубину, поддерживая две хэш-таблицы: одну — для улучшения производительности вычисления, вторую — для построения максимально приведенного графа. Заметим, что в отличие от более ранних алгоритмов, которым требовался отдельный шаг для приведения сокращаемого графа к канонической форме, описанный ниже алгоритм строит приведенную форму сразу. Чтобы проиллюстрировать эту операцию, мы рассмотрим применение операции + к функциям $f(a,b,c,d) = (a + b) \cdot c + d$ и $g(a,b,c,d) = (a \cdot \bar{c}) + d$, OBDD-представление которых показано на рис. 6.

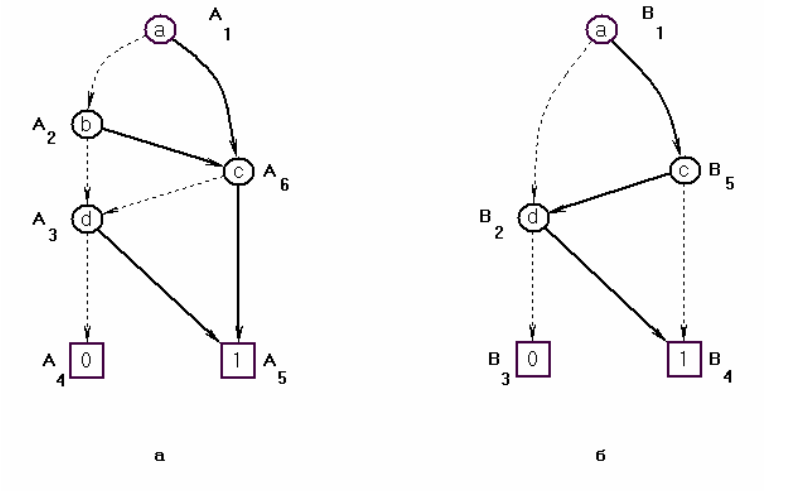

Рис. 6. Аргументы операции APPLY.
Вершины помечены для идентификации их в следе исполнения

Реализация операции APPLY опирается на то, что алгебраические операции связаны с Шенноновым разложением по любой переменной x:

$$f \langle op \rangle g = \bar{x} \cdot (f|_{x \leftarrow 0} \langle op \rangle g|_{x \leftarrow 0}) + x \cdot (f|_{x \leftarrow 1} \langle op \rangle g|_{x \leftarrow 1}). \qquad (1)$$

Заметим, что для функции f, представленной диаграммой с корневой вершиной r_f, рестрикция относительно переменной x, такой, что $x \le \text{ПЕР}(r_f)$, может быть вычислена следующим простым способом:

$$f|_{x\leftarrow 0} = \begin{cases} r_f, & \text{если } x < \text{ПЕР}(r_f), \\ \text{ЛЕВ}(r_f), & \text{если } x = \text{ПЕР}(r_f) \text{ и } b = 0, \\ \text{ПРАВ}(r_f), & \text{если } x = \text{ПЕР}(r_f) \text{ и } b = 1. \end{cases}$$

Т. е. рестрикция представлена либо тем же самым графом, либо одним из подграфов корневой вершины.

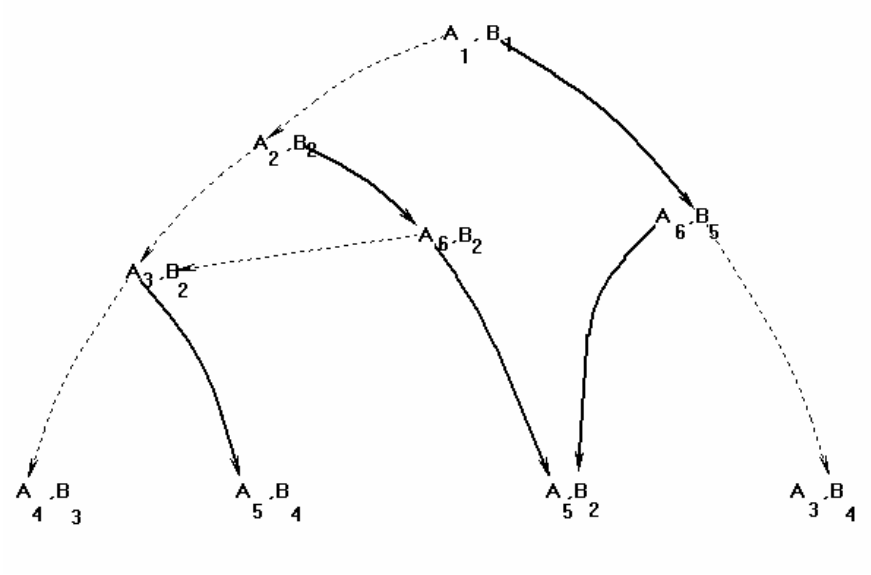

Рис. 7. След исполнения операции APPLY с операцией +.
Каждый шаг выполнения имеет в качестве операндов вершину
из каждого графа-аргумента

Равенство 1 образует базис для рекурсивной процедуры вычисления OBDD-представления f <op> g. Структура рекурсивного вычисления для нашего примера представлена на рис. 7. Заметим, что каждый шаг вычисления идентифицируется вершиной каждого из графов-аргументов. Предположим, что функции f и g представлены OBDD-диаграммами с корневыми вершинами r_f и r_g, соответственно. В случае, когда и r_f и r_g являются терми-

нальными вершинами, рекурсия завершается, возвращая подходящим образом помеченную терминальную вершину. В нашем примере это происходит при вычислении A_4,B_3 и A_5,B_4. В противном случае будем считать *x расщепляющей переменной*, определяя ее как минимум из переменных ПЕР(r_f) и ПЕР(r_g). OBDD-диаграммы для функций

$$f\big|_{x\leftarrow 0} <\text{op}> g\big|_{x\leftarrow 0} \text{ и } f\big|_{x\leftarrow 1} <\text{op}> g\big|_{x\leftarrow 1}$$

строятся путем рекурсивного выполнения рестрикций f и g для значения 0 (обозначенных пунктирными линиями на рис. 7) и для значения 1 (обозначенных сплошными линиями). В нашем примере начальное вычисление для вершин A_1,B_1 вызывает рекурсивное вычисление для вершин A_2,B_2 и A_6,B_5.

Чтобы более эффективно реализовать операцию APPLY, можно ввести два усовершенствования в описанную выше процедуру.

Во-первых, если встречается условие, в котором один из аргументов является терминальной вершиной, представляющей «доминантное» значение для операции <op> (например, 1 для OR или 0 для AND), то можно остановить рекурсию и вернуть подходящим образом помеченную терминальную вершину. В нашем примере это происходит при вычислении для A_5,B_2 и A_3,B_4.

Во-вторых, можно избегать любых множественных рекурсивных вызовов одной и той же пары аргументов путем поддержки хэш-таблицы, в которой каждая запись в качестве ключа содержит пару вершин двух графов-аргументов, а в качестве элемента данных — вершину генерируемого графа. В начале выполнения для аргументов u и v проверяется наличие записи с ключом $<u, v>$ в хэш-таблице. Если такая запись обнаружена, возвращается элемент данных этой записи, что позволяет прекратить дальнейшую рекурсию. Если записи не найдено, производятся описанные выше действия по созданию новой записи перед возвратом результата. В нашем примере с помощью этого усовершенствования мы избегаем множественного исполнения для аргументов A_3,B_2 и A_5,B_2. Заметим, что при использовании указанного усовершенствования структура вычисления представлена ориентированным ациклическим графом, а не более привычной для рекурсивных процедур древовидной структурой.

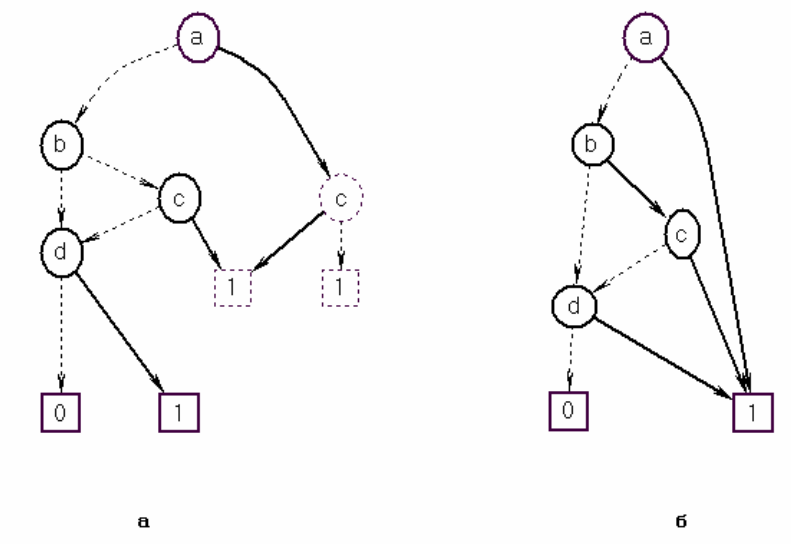

Рис. 8. Генерация результата для операции APPLY.
Структура рекурсивных вызовов по своей природе определяет неприведенный граф (слева). С помощью применения правил сокращения в конце каждого рекурсивного вызова сразу же строится приведенный граф (справа)

Каждый шаг вычисления возвращает в качестве результата вершину генерируемого графа. Структура рекурсивной оценки по своей природе определяет неприведенный граф, в котором каждый шаг вычисления задает вершину, помеченную расщепляющей переменной и имеющую в качестве потомков результаты рекурсивных вызовов. Такой граф для нашего примера показан на рис. 8. Чтобы сразу же генерировать приведенный граф, на каждом шаге выполнения мы пытаемся «уклониться» от создания новой вершины с помощью применения тестов, соответствующих описанным ранее правилам преобразования. Предположим, что шаг выполнения содержит расщепляющую переменную x, и рекурсивные вычисления возвращают вершины v_0 и v_1. Вначале мы проверяем, выполняется ли условие $v_0 = v_1$, и, если выполняется, то мы возвращаем эту вершину как результат процедуры. Затем мы проверяем, не содержит ли уже сгенерированный граф какую-нибудь вершину v, такую, что ПЕР(v) = x, ЛЕВ(v) = v_0 и ПРАВ(v) = v_1. Эта проверка сопровождается поддержкой хэш-таблицы, содержащей запись для каждой нетерминальной вершины v в сгенерированном файле с ключом

<ПЕР(v), ПРАВ(v), ЛЕВ(v)>.

Если искомая вершина найдена, она возвращается как результат процедуры. В противном случае вершина добавляется в граф, ее запись добавляется в хэш-таблицу, и вершина возвращается как результат процедуры. Подобным же образом в хэш-таблице оказываются терминальные вершины, их ключами являются метки. Новая терминальная вершина генерируется только в том случае, если не имеется вершины с нужной меткой. В нашем примере этот процесс не создает вершины, изображенные пунктиром в левой части рис. 8. Напротив, граф в правой части рис. 8 генерируется непосредственно. Заметим, что этот граф представляет функцию

$$a + b*c + d,$$

которая и является результатом применения операции OR к двум функциям-аргументам.

Использование таблицы для устранения множественного исполнения данной пары вершин ограничивает сложность операции APPLY и одновременно задает границу размера результата. А именно, пусть функции f и g представлены OBDD-диаграммами с количеством вершин m_f и m_g, соответственно. Тогда может быть не более чем $m_f m_g$ уникальных аргументов для вычисления, и каждое вычисление добавляет не более одной вершины к генерируемому результату. Имея хорошую реализацию хэш-таблицы, мы можем провести каждый шаг вычисления в среднем за константное время. Таким образом, и сложность алгоритма, и размер генерируемого результирующего графа должны оцениваться как $O(m_f m_g)$.

2.2. Операция RESTRICT

Вычисление рестрикции функции, представленной любым видом бинарной диаграммы решений, довольно просто. Чтобы выполнить рестрикцию переменной x до значения k, мы перенаправим любую дугу, ведущую к вершине v, для которой ПЕР(v) = x, либо в ЛЕВ(v) для k = 0, либо в ПРАВ(v) для k=1. На рис. 9 представлена рестрикция переменной b в функции

$$b*c + a*\overline{b} *\overline{c}$$

до значения 1. Исходная функция задана OBDD в левой части. После перенаправления дуг вершина b исключается из обхода, что показано в центральной части.

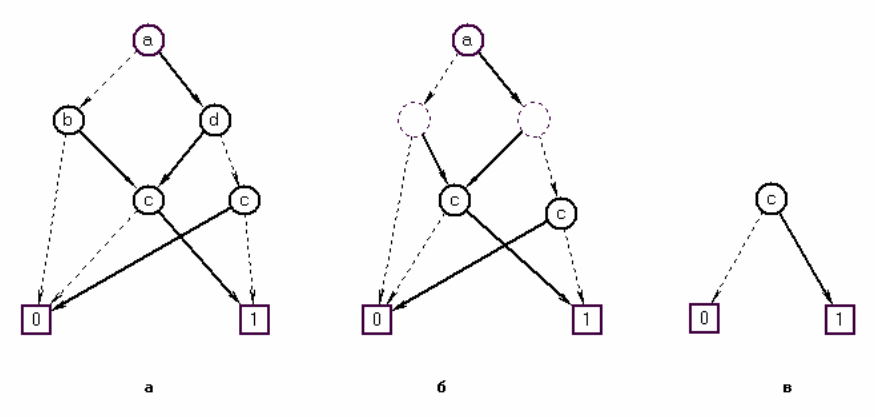

Рис. 9. Пример операции RESTRICT.
Рестрикция переменной b (слева) до значения 1 приводит к исключению из обхода вершин, помеченных b (в центре) и к получению приведенного графа (справа)

Как видно из этого примера, прямая реализация техники рестрикции может дать в итоге неприведенный граф. Взамен этого вначале производится обход исходного графа в глубину. Каждый рекурсивный вызов имеет своим аргументом вершину в исходном графе и возвращает в виде результата вершину генерируемого графа. Чтобы гарантировать, что генерируемый граф является приведенным, процедура поддерживает хэш-таблицу, содержащую запись для каждой вершины генерируемого графа, применяя те же правила сокращения, что и при операции APPLY. В нашем примере результат может быть OBDD-представлением функции *c*, как показано в правой части рис. 9.

Вычисление рестрикции функции *f*, OBDD-представление которой имеет m_f вершин, содержит не более чем m_f рекурсивных вызовов, каждый из которых генерирует не более одной вершины результирующего графа. Имея хорошую реализацию хэш-таблицы, каждый шаг рекурсии требует в среднем константного времени. Таким образом, и сложность алгоритма, и размер генерируемого результирующего графа можно оценить как $O(m_f)$.

2.3. Производные операции

Как было описано выше, значительное количество операций над булевыми функциями может быть выражено в терминах алгебраических опера-

ций и операции рестрикции. Алгоритмы APPLY и RESTRICT дают способ реализации этих операций. Далее, для каждой из этих операций ее сложность и размер генерируемого графа ограничены некоторой полиномиальной функцией относительно функций-аргументов. Обозначим размер OBDD-представления функции f как m_f. Пусть даны функции f и g и «состояния безразличности», выраженные функцией d, тогда мы можем вычислить эквивалентность f и g для «небезразличных» состояний за время $O(m_f m_g m_d)$. Мы можем вычислить композицию функций f и g с двумя рестрикциями и тремя вызовами APPLY. Этот подход дает временную сложность $O(m_f^2 m_g^2)$. Реализуя все вычисление за один проход, можно сократить эту оценку до $O(m_f m_g^2)$. Наконец, можно провести квантификацию переменной функции f за время $O(m_f^2)$.

2.4. Характеристики производительности

Использование OBDD для решения проблемы требует выражения задания в виде серии операций на булевых функциях, подобных описанным выше. Как мы видели, все эти операции могут быть реализованы с помощью алгоритмов, полиномиальных по сложности относительно размера диаграмм, представляющих аргументы. В результате основанные на OBDD символьные булевы операции имеют два преимущества по сравнению с другими общепринятыми подходами.

Во-первых, до тех пор, пока размеры графов остаются приемлемыми, все вычисление в целом оказывается контролируемым.

Во-вторых, хотя размеры графов могут возрастать с каждой успешной операцией, любая отдельная операция имеет разумную производительность в худшем случае.

Напротив, многие другие представления булевых функций не обладают этим свойством «постепенного снижения производительности». Например, справедливо следующее свойство.

Свойство Г. *Даже если функция имеет умеренно компактное представление суммы произведений, ее дополнение может быть экспоненциального размера.*

2.5. Технология реализации

С точки зрения реализации, основанные на OBDD символьные операции имеют характеристики, отличающие их от многих других вычислительных задач. Во время вычисления конструируются и уничтожаются тысячи гра-

фов, содержащие тысячи вершин каждый. Информация представлена при использовании OBDD-представлений скорее самой их структурой, чем ассоциированными с ними данными, и в силу этого к каждой отдельной вершине прилагаются очень небольшие вычислительные действия. Таким образом, вычисление носит высоко динамичный характер, не позволяющий делать предварительных предположений об обращениях к памяти. В настоящее время наиболее успешные варианты реализации были получены на рабочих станциях с большой физической памятью, на которых уделялось много внимания программированию процедур управления памятью.

Чтобы достичь максимальной производительности, было бы желательно использовать потенциал конвейерных и параллельных компьютеров. В задачах символьного анализа параллелизм может существовать на *макроуровне*, когда многие операции могут быть исполнены одновременно, и на *микроуровне*, когда многие вершины внутри заданного OBDD-представления могут быть обработаны одновременно. В сравнении с другими задачами, которые были успешно реализованы на векторных и параллельных компьютерах, обработка с использованием OBDD требует заметно большей коммуникации и синхронизации между вычисляющими элементами и заметно меньших локальных вычислений. Эта задача пока остается вызовом для разработчиков архитектур параллельных компьютеров, программных моделей и языков программирования. Тем не менее, некоторые из первых попыток оказались многообещающими. Исследователи успешно использовали векторную обработку и показали хорошие результаты при исполнении на мультипроцессорах с разделенной памятью. Оба подхода используют параллелизм на микроуровне с помощью реализации операции APPLY в виде обхода графов-аргументов в ширину, в противоположность общепринятому для большинства других реализаций обходу в глубину.

3. ПРЕДСТАВЛЕНИЕ МАТЕМАТИЧЕСКИХ ОБЪЕКТОВ

Некоторые приложения, чаще всего в области разработки цифровых систем связи, требуют прямой реализации и обработки булевых функций. Однако в общем случае мощность символьной булевой обработки состоит в способности бинарных значений и булевых операций представлять и воплощать разнообразные области математики.

Таблица 3

**Примеры систем,
которые могут быть представлены булевыми функциями**

Класс	**Типичные операции**	**Типичные тесты**
Логика	$\wedge$, $\vee$, $\neg$, $\forall$, $\exists$	выполнимость, импликация
Конечные области	*Проблемно-зависимые*	эквивалентность
Функции	Аппликация, композиция	эквивалентность
Множества	$\cup$, $\cap$, –	подмножество
Отношения	композиция, замыкание	симметричность, транзитивность

В табл. 3 приводятся примеры некоторых разделов математики, в которых объекты могут быть представлены, обработаны и проанализированы с помощью символьных булевых операций, так как лежащие в их основе области являются конечными. Предоставляя унифицированную структуру для различных математических систем, символьная булева обработка дает возможность решать не только проблемы в отдельных областях, но и проблемы, затрагивающие различные области математики. Например, разработаны программы для анализа последовательного поведения цифровых схем, которые включают операции над всеми перечисленными в табл. 3 областями. Требуемые свойства системы выражаются в виде логических формул. Поведение системы задается функциями следующего состояния схемы. Анализатор вычисляет множества состояний, имеющих определенные свойства. Структура переходов в системе конечного автомата представлена в виде отношения. Во время исполнения анализатор может быстро перемещаться между представлениями, используя только OBDD как базисную структуру данных. Более того, свойство каноничности OBDD позволяет с легкостью распознавать такие условия, как сходимость, и определять, существует ли решение заданной проблемы.

Ключом к использованию возможностей символьной булевой обработки является выражение исходной проблемы в форме, в которой все объекты представлены булевыми функциями. В оставшейся части данной главы мы опишем некоторые стандартные техники, которые были разработаны для этой цели. Имея достаточный опыт и практику, можно представить в таком виде удивительно широкий класс проблем. Математические основы для

этого представления были разработаны уже давно. Ни одна из рассматриваемых техник не опирается на особенности OBDD-представления — они могут быть реализованы для любого из множества представлений. Разработка OBDD-представления просто расширила спектр проблем, которые могут быть разрешены практически. Однако тем самым она усилила мотивацию к представлению различных проблем в терминах символьных булевых операций.

3.1. Кодирование конечных областей

Рассмотрим конечное множество элементов A, $|A| = N$. Мы можем кодировать элемент из A с помощью вектора из n бинарных значений, где $n = \lceil \log_2 N \rceil$. Это кодирование обозначается функцией $\sigma: A \to \{0, 1\}^n$, которая сопоставляет элементам из A различные битовые n-вектора. Пусть $\sigma_i(a)$ обозначает i-й бит этого вектора. Функция $f: A \to A$, отображающая элементы A на элементы A, представлена вектором из n булевых функций $\vec{f}$, где каждая функция f_i: $\{0, 1\}^n \to \{0, 1\}$ определяется как

$$f_i(\sigma(a)) = \sigma_i(f(a)).$$

Во многих реальных приложениях области обладают «естественным» порядком (пример — бинарное кодирование целых чисел); для других областей порядок строится искусственно.

Например, символьный симулятор COSMOS использует OBDD для символьного расчета поведения транзисторной схемы. Такой симулятор может быть использован для автоматической генерации тестов для поиска ошибок в схеме и для формальной проверки того, отвечает ли поведение схемы некоторым заданным критериям. В модели схемы напряжение в узлах представлено трехзначным сигнальным набором, в котором значения 0 и 1 обозначают низкое и высокое напряжение, а третье значение, X, обозначает неизвестное или потенциально нецифровое напряжение. Во время символьного моделирования состояния вершин должны быть вычислены как трехзначные функции над множеством булевых переменных, введенных пользователем для представления первичных входных данных или начального состояния. COSMOS представляет состояние вершины парой OBDD-диаграмм. А именно, он кодирует каждый из $N = 3$ элементов сигнального набора вектором из $n = 2$ бинарных значений в соответствии с кодирующей функцией:

$$\sigma(0) = [0, 1], \sigma(1) = [1, 0], \sigma(X) = [1, 1].$$

Функции следующего состояния, вычисляемые симулятором, полностью определяются в соответствии с этим булевым кодированием, что дает возможность точно описать булевыми функциями поведение схемы. В табл. 4 приведен пример трехзначного расширения операций AND, OR и NOT. Заметим, что операции дают значение X в каждом случае, когда неизвестный аргумент может вызвать неопределенность в значении функции. Пусть $[a_1, a_0]$ обозначает кодировку трехзначного сигнала a, тогда трехзначная операция может быть полностью выражена в терминах булевых операций:

$$[a_1, a_0] \cdot_t [b_1, b_0] = [a_1 \cdot b_1, a_0 + b_0],$$
$$[a_1, a_0] +_t [b_1, b_0] = [a_1 + b_1, a_0 \cdot b_0],$$
$$\overline{[a_1, a_0]}^t = [a_0, a_1].$$

Во время функционирования симулятор действует подобно обычному логическому симулятору, управляемому событиями. Вначале каждый внутренний узел инициализируется значением [1, 1], означающим, что значение узла неизвестно независимо от обстоятельств. В процессе моделирования состояния узлов модифицируются с помощью вычисления булевого представления функции следующего состояния с применением операции APPLY. Каждый раз, когда состояние узла перевычисляется, старое состояние сравнивается с новым, и в случае, когда они неэквивалентны, создается событие для каждого разветвления этого узла. Процесс продолжается до тех пор, пока список событий не становится пустым, указывая на то, что сеть находится в стабильном состоянии. Этот метод обработки событий требует наличия эффективного способа проверки эквивалентности.

Таблица 4

Трехзначное расширение операций AND, OR и NOT. Третье значение X обозначает неизвестное или потенциально нецифровое напряжение

$\cdot_t$	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

$+_t$	0	1	X
0	0	1	X
1	1	1	1
X	X	1	X

a	$\overline{a}^t$
0	1
1	0
X	X

3.2. Множества

Пусть задано базисное множество A. Мы можем представлять и обрабатывать его подмножества, используя «характеристические функции». Множество $S \subseteq A$ обозначается функцией χ_S: $\{0, 1\}^n \to \{0, 1\}$,

$$\chi_S(\vec{x}) = \sum_{a \in S} \prod_{1 \le i \le n} x_i \,\overline{\otimes}\, \sigma_i(a),$$

где $\overline{\oplus}$ представляет собой дополнение операции EXCLUSIVE-OR. Теперь операции над множествами (подмножествами) могут быть реализованы с помощью булевых операций на их характеристических функциях, например:

$$\chi_\varnothing := \mathbf{0},$$
$$\chi_{S \cup T} := \chi_S + \chi_T,$$
$$\chi_{S \cap T} := \chi_S \cdot \chi_T,$$
$$\chi_{S-T} := \chi_S \cdot \overline{\chi_T}.$$

Множество S является подмножеством T в том и только том случае, если $\chi_S \cdot \overline{\chi_T} = \mathbf{0}$. Многие приложения, использующие OBDD, строят и обрабатывают множества даже без явной нумерации их элементов. Вместо этого можно представить (непустое) множество как множество возможных выходных значений вектора функций, т.е. можно полагать, что функция $\vec{f}$ обозначает множество

$$\{a \mid \sigma(a) = \vec{f}(\vec{b}) \text{ для некоторого } \vec{b} \in \{0,1\}^n\}.$$

Это представление может оказаться удобным для приложений, в которых анализируемая система представлена вектором функций. Модифицируя эти функции, мы можем также представлять подмножества состояний системы.

3.3. Отношения

Мы можем определить k-арное отношение как множество упорядоченных кортежей размерности k. Следовательно, отношения также могут быть представлены и обработаны с помощью характеристических функций. Рассмотрим в качестве примера бинарное отношение $R \subseteq A \times A$, обозначаемое булевой функцией χ_R, которая определяется следующим образом:

$$\chi_R(\vec{x}, \vec{y}) = \sum_{a \in A} \sum_{\substack{b \in A \\ aRb}} \left[\prod_{1 \le i \le n} x_i \,\overline{\oplus}\, \sigma_i(a) \right] \left[\prod_{1 \le i \le n} y_i \,\overline{\oplus}\, \sigma_i(b) \right].$$

С помощью этого представления мы можем производить такие действия, как пересечение, объединение и разность отношений, применяя булевы операции к их характеристическим функциям.

Комбинируя функциональную композицию и квантификацию переменных, мы можем также получать составные отношения:

$$\chi_{R^\circ S} := \exists\ \vec{x}\ [\chi_R(\vec{x},\vec{z}) \cdot \chi_S(\vec{z},\vec{y})],$$

где $R \circ S$ обозначает композицию отношений R и S. Квантификация над вектором переменных включает квантификацию каждого элемента вектора в любом порядке. Далее, мы можем вычислить транзитивное замыкание отношения, используя метод неподвижной точки. Функция χ_{R^*} вычисляется как предел последовательности функций χ_{Ri}, каждую из которых определяет отношение

$$R_0 = I,$$
$$R_{i+1} = I \cup R \circ R_i,$$

где I обозначает тождественное отношение. Процесс вычисления сходится, когда он достигает такой итерации i, что $\chi_{Ri} = \chi_{Ri+1}$, для чего вновь потребуется эффективная проверка эквивалентности. Если мы предположим, что R представляет граф, с вершиной для каждого элемента A и дугой для каждого элемента R, тогда отношение R_i обозначает пары вершин, достижимые по путям длиной не более чем i дуг. Следовательно, вычисление должно сходиться максимум за N-1 итераций, где $N = |A|$. Техника, известная как «метод итеративных квадратур», сокращает максимальное количество итераций до $n = \lceil \log_2 N \rceil$. Каждая итерация вычисляет отношение $R_{(i)}$, указывающее на пары вершин, достижимых по путям не длиннее 2^n:

$$R_{(0)} = I \cup R,$$
$$R_{(i+1)} = R_{(i)} \circ R_{(i)}.$$

Многие приложения OBDD содержат обработку отношений над очень большими множествами, и поэтому сокращение числа итераций с N (например, 10^9) до n (например, 30) может быть очень существенным.

4. ДРУГИЕ ПРИМЕНЕНИЯ

OBDD широко используются в разработке, верификации и тестировании цифровых систем связи. Ниже будут описаны методы их применения в некоторых областях.

4.1. Верификация

OBDD могут быть использованы непосредственно для установления эквивалентности двух комбинаторно-логических схем. Эта проблема возникает в случаях, когда необходимо сравнить схему и сеть, полученную из спецификаций системы, либо когда требуется проверить, не изменил ли логический оптимизатор функциональность схемы. Вначале с помощью операции APPLY вычисляются функциональные представления обеих сетей, и затем проверяется их эквивалентность. Этот метод позволяет также сравнивать две последовательные системы, если они используют одно и то же кодирование состояний, а именно — если они имеют идентичные выходные данные и функции перехода (к следующему состоянию).

4.2. Коррекция ошибок схемы

Не довольствуясь простым обнаружением ошибок в логической схеме, можно создавать методы автоматической коррекции дефектных схем. Существующие методы коррекции включают рассмотрение относительно небольшого класса потенциальных ошибок, таких, как одиночный некорректный логический элемент, и определение того, может ли какой-либо вариант данной цепи достичь требуемой функциональности. Этот анализ демонстрирует мощь операций квантификации при вычислении проекций, в данном случае производя проекцию первичных входных значений посредством универсальной квантификации.

Такой анализ может быть проведен символически, с помощью кодирования возможных функций элементов булевыми переменными. Как показано на примере (рис. 10), произвольный *k*-входовой канал может быть моделирован 2^k-входовым мультиплексором, где операция элемента определена входными данными мультиплексора $\vec{a}$. Рассмотрим контур N с одним входом, в котором один из логических элементов заменен таким блоком, это дает в итоге функциональность сети $N(\vec{x},\vec{a})$, где $\vec{x}$ представляет множество первичных входных значений. Предположим, что желаемая функциональность есть $S(\vec{x})$. Наша задача — определить, могут ли (и если могут, то каким образом) две функции быть сделаны идентичными для всех значений первичных данных посредством подходящего «программирования» элемента. Это включает вычисление функции $C(\vec{a})$, определенной следующим образом:

$$C(\vec{a}) = \forall \vec{x} \ \ [N(\vec{x},\vec{a}) \ \overline{\oplus} \ S(\vec{x})].$$

Хотя серьезные ошибки схемы не могут быть исправлены таким образом, он позволяет избежать утомительной коррекции часто встречающихся ошибок, таких, как неправильное расположение инверторов или использование некорректного типа логического элемента. Эта задача также оказывается полезной в логическом синтезе, когда требуется изменить схему таким образом, чтобы она соответствовала измененной спецификации.

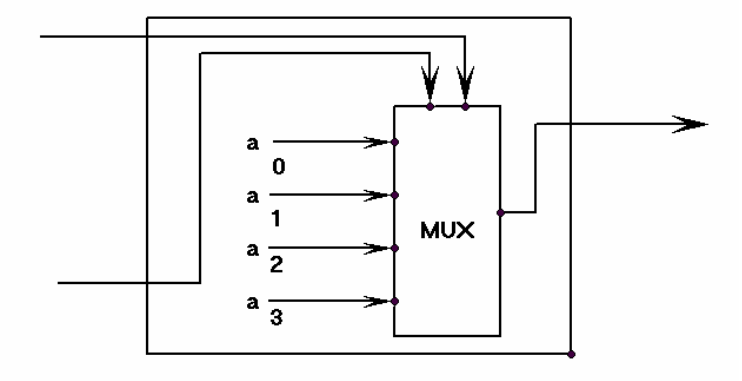

Рис. 10. Блок функции всеобщности.
Присваивая различные значения переменным $\vec{a}$,
можно реализовать произвольную 2-входовую операцию

4.3. Анализ чувствительности

Второй класс приложений включает составление характеристик того влияния, который оказывают изменения значений сигналов различных проводов комбинаторной схемы. А именно, для каждого значения сигнала *s* мы хотим вычислить булеву разницу для каждого первичного выходного значения относительно *s*. Этот анализ может быть проведен символически, с помощью введения в сеть «преобразователей сигнального провода», как показано на рис. 11. Для каждого провода, в обычной ситуации передающего сигнал *s*, мы избирательно изменяем значение сигнала, которое теперь равно *s'*, при помощи булевого значения P_s: $s' = s \oplus P_s$.

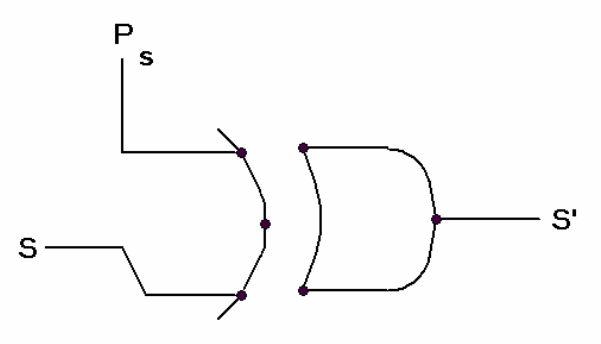

Рис. 11. Преобразователь сигнального провода.
Ненулевое значение P_s изменяет значение, передаваемое по этому проводу

Мы можем определить условия, при которых некоторое выходное значение схемы оказывается чувствительным к значению сигнального провода, сравнивая выходные значения исходной и измененной схем, как показано на рис. 12. Как видно из этого рисунка, мы можем даже вычислить влияние каждого однопроводного преобразования в схеме за одно символическое вычисление. А именно, пронумеруем каждый сигнальный провод числами от 1 до m-1 и введем множество $\lceil \log m \rceil$ «перестановочных переменных» $\vec{r}$. Каждый перестановочный сигнал P_s затем определяется как функция, принимающая значение 1 в случае, когда перестановочные переменные являются бинарным представлением номера, присвоенного сигналу s. В терминах построения логических схем это эквивалентно генерации перестановочных сигналов декодером с $\vec{r}$ на входе. Полученная функция $T(\vec{x}, \vec{r})$ принимает значение 1, если исходная схема и схема, полученная с помощью перестановки $\vec{r}$, выдают одни и те же выходные значения на входных значениях $\vec{x}$.

Одной областью приложения такого анализа чувствительности является автоматическая генерация тестов. Функция чувствительности описывает множество всех тестов для каждой одиночной ошибки. Предположим, что сигнальный провод, помеченный бинарным номером $\vec{b}$, в обычной ситуации имеет в схеме функцию $s(\vec{x})$. Тогда входной шаблон $\vec{a}$ будет обнаруживать ошибку типа «отсутствие сигнала на проводе 1» в том и только том случае, если $\overline{T(\vec{a},\vec{b})} \bullet \overline{s(\vec{a})} = 0$. Подобным же образом, шаблон $\vec{a}$ будет

обнаруживать ошибку типа «отсутствие сигнала на проводе 0» в том и только том случае, если

$$\overline{T(\vec{a},\vec{b})} \bullet \overline{s(\vec{a})} = 1.$$

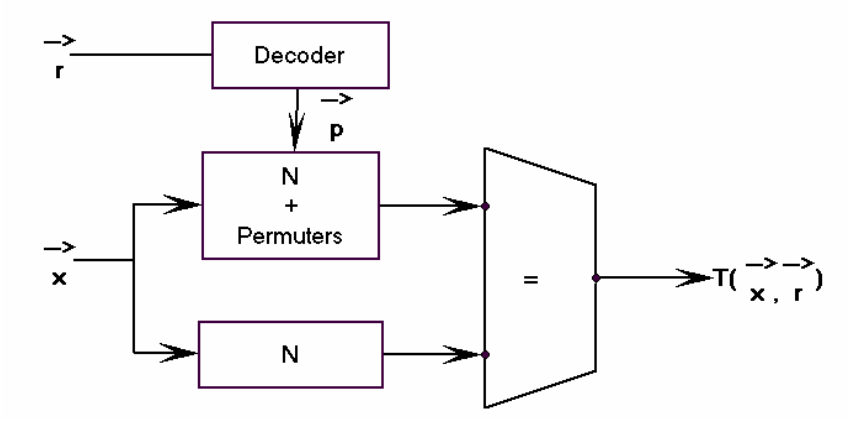

Рис. 12. Вычисление чувствительных к однопроводным преобразованиям участков. Каждое присваивание переменной $\vec{r}$ вызывает изменение значения ровно одного провода

Этот метод также может быть распространен на последовательные схемы и на схемы, представленные на уровне переключателей.

Второй областью приложения является комбинационно-логическая оптимизация. Для сигнального провода, имеющего бинарный номер $\vec{b}$, функция $T(\vec{x},\vec{y})$ представляет набор «безразличностей» для каждого провода схемы, то есть таких случаев, когда выходные значения схемы не зависят от сигнального значения этого провода. Руководствуясь этой информацией, оптимизатор схемы может применять такие преобразования, как удаление сигнального провода или перемещение провода на другой выходной элемент. Недостаток этого подхода состоит в том, что функция чувствительности должна вычисляться заново каждый раз, как оптимизатор изменяет схему. Альтернативный подход дает более ограниченный, но и более компактный набор функций «безразличностей», в котором наборы «безразличностей» остаются действительными даже в случаях, когда меняется структура схемы.

4.4. Вероятностный анализ

Существует метод для статистического анализа влияния меняющейся задержки сигнала в цифровой схеме. Это приложение OBDD выглядит особенно интригующим, поскольку здравый смысл заставляет считать, что такой анализ требует вычисления реальных вариаций параметров и в силу этого не может быть закодирован булевыми переменными.

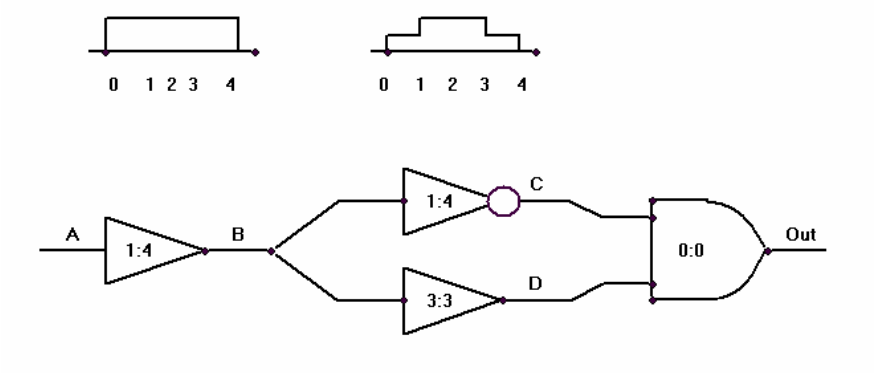

Рис. 13. Схема с неопределенными задержками.
Логические элементы помечены своими минимальными и максимальными задержками.
Инверторы имеют распределение задержек

Рассмотрим сеть логических элементов, в которой каждый элемент имеет задержку, задаваемую некоторым вероятностным распределением. Эта схема демонстрирует диапазон вариантов поведения, некоторые из которых классифицируются как нежелательные. «Выход» определяется как вероятность того, что нежелательное поведение не встретится. На рис. 13 приведен пример простой схемы, где два из логических элементов имеют переменное распределение задержек; мы хотим оценить вероятность появления сбоя в вершине Out, в то время как входной сигнал A совершает переход из 0 в 1. Нужно провести анализ ситуации, когда сигнал A изменяет свое значение на 1 в момент времени 0. Сигналы C и D совершают переход, для которого известно вероятностное распределение времени перехода. Необходимо провести однократный простой анализ для расчета вероятной формы волны для всех сигналов, как если бы они распределялись независимо. Затем мы сможем легко вычислить поведение каждого выходного значения схемы в соответствии с функцией элемента и формой входной волны. Например, если мы рассматриваем сигналы C и D как независимые, мы можем

вычислить вероятность приближающегося перехода в вершине Out в момент времени t как произведение (1) вероятности того, что C совершает переход в момент t и (2) вероятности того, что D не совершает перехода во время ≤ t. Это приводит к вероятностному распределению перехода, помеченному «Out (Independent)». Конечная вероятность появления перехода (то есть сбоя) оказывается равной 30%. В действительности, разумеется, время перехода сигналов C и D в значительной степени коррелирует друг с другом — оба подвержены влиянию задержки начального буферного элемента. В силу этого более тщательный анализ даст вероятностное распределение времени перехода, помеченное «Out (Actual)», в результате конечная вероятность появления перехода оказывается равной 12.5%. Таким образом, упрощенный анализ недооценивает выход схемы. В других случаях упрощенный анализ может переоценивать выход схемы.

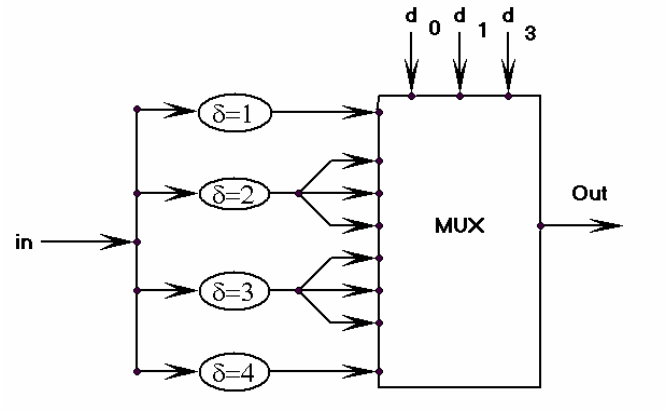

Рис. 14. Моделирование неопределенных задержек.
Булевы переменные управляют выбором задержки.
Сигналы повторяются в соответствии с распределением задержек

Чтобы решить эту проблему с помощью символьного булева анализа, мы должны принять два ограничения. Во-первых, все задержки схемы должны иметь целочисленные значения (с помощью подходящего выбора единицы времени), в силу чего переходы будут совершаться в дискретных временных точках. Во-вторых, вероятность задержки должна быть кратной значению $1/k$, где k — некоторая степень двойки. Например, оба переменных элемента на рис. 13 имеют задержки от 1 до 4. Один имеет равномерное распределение задержек [1/4, 1/4, 1/4, 1/4], тогда как у другого распре-

деление задержек приближается к нормальному [1/8, 3/8, 3/8, 1/8]. Затем значение задержки для элемента может быть закодировано множеством из log k булевых переменных, как показано на рис. 14. А именно, мы моделируем компонент схемы k-входовым мультиплексором, в котором значение задержки, имеющее вероятность c/k, подается на с входов. Затем схема рассчитывается с использованием символьного расширения обычного алгоритма симуляции логических схем. Значение сигнала в вершине N в каждый момент времени t является булевой функцией $N(t)$ от переменных задержки.

Таблица 4

Условия задержки для схемы на рис. 14

$A \rightarrow B$

Задержка	*Условия*
1	$\overline{e_1} \cdot \overline{e_0}$
2	$\overline{e_1} \cdot e_0$
3	$e_1 \cdot \overline{e_0}$
4	$e_1 \cdot e_0$

$B \rightarrow C$

Задержка	*Условие*
1	$\overline{d_2} \cdot \overline{d_1} \cdot \overline{d_0} \cdot$
2	$\overline{d_2} \cdot (d_1 + d_0)$
3	$d_2 \cdot (\overline{d_1} + \overline{d_0})$
4	$d_2 \cdot d_1 \cdot d_0$

Предположим, что в примере на рис. 15 переменные $[e_1, e_0]$ кодируют задержку между A и B, тогда как переменные $[d_2, d_1, d_0]$ кодируют задержку между B и C, как показано в табл. 4. Для времени $t < 0$ функции вершин будут следующими: $A(t) = B(t) = D(t) = Out(t) = \mathbf{0}$, $C(t) = \mathbf{1}$. Для времени $t \geq 0$ вершина A имеет функцию $A(t) = \mathbf{1}$, остальные функции вычисляются следующим образом:

$$\begin{aligned} B(t) = {} & \overline{e_1} \cdot \overline{e_0} \cdot A(t\text{-}1) \\ & + \overline{e_1} \cdot e_0 \cdot A(t\text{-}2) \\ & + e_1 \cdot \overline{e_0} \cdot A(t\text{-}3) \\ & + e_1 \cdot e_0 \cdot A(t\text{-}4), \end{aligned}$$

$$C(t) = \overline{d_2} \cdot \overline{d_1} \cdot \overline{d_0} \cdot \overline{B(t-1)}$$
$$+ \overline{d_2} \cdot (d_1 + d_0) \cdot \overline{B(t-2)}$$
$$+ d_2 \cdot (\overline{d_1} + \overline{d_0}) \cdot \overline{B(t-3)}$$
$$+ d_2 \cdot d_1 \cdot d_0 \cdot \overline{B(t-4)},$$
$$D(t) = B(t - 3),$$
$$Out(t) = C(t) \cdot D(t).$$

Значение выходного сигнала мы можем получить из этих уравнений как Out(t) = **0** для t ≤ 3 и t ≥ 8. Для других значений времени значение сигнала будет вычисляться следующим образом:

$$Out(4) = d_2 \cdot d_1 \cdot d_0 \cdot \overline{e_1} \cdot \overline{e_0},$$
$$Out(5) = d_2 \cdot d_1 \cdot d_0 \cdot \overline{e_1} \cdot e_0,$$
$$Out(6) = d_2 \cdot d_1 \cdot d_0 \cdot e_1 \cdot \overline{e_0},$$
$$Out(7) = d_2 \cdot d_1 \cdot d_0 \cdot e_1 \cdot e_0.$$

Мы можем вычислить булеву функцию, показывающую условия задержки, при которых встречается некоторое нежелательное поведение. Например, мы можем вычислить вероятность того, что сбой происходит в вершине *Out*, как $G = \Sigma Out(t)$. В этом случае мы можем вычислить $G = d_2 \cdot d_1 \cdot d_0$, то есть вероятность того, что сбой происходит в том и только том случае, когда задержка между B и C равна 4.

Пусть дана булева функция, представляющая условия, при которых происходит некоторое событие. Мы можем вычислить вероятность события путем вычисления плотности функции, то есть доли присваиваний переменным, на которых функция принимает значение 1. С помощью Шенноновского раскрытия мы можем задать плотность $\rho(f)$ функции f рекурсивным образом:

$$\rho(\mathbf{1}) = 1$$
$$\rho(\mathbf{0}) = 0$$
$$\rho(f) = 1/2\ [\rho(f|_{x\leftarrow 0}) + \rho(f|_{x\leftarrow 1})].$$

Таким образом, при наличии OBDD-представления f мы можем вычислить плотность за линейное время путем обхода графа в глубину, помечая каждую вершину плотностью функции, обозначающей ее подграф. На

рис. 15 изображено такое вычисление для OBDD, представляющего условия, при которых вершина C на рис. 14 содержит переход в момент времени 6, помечая ее вероятностью этого события, равной 7/32.

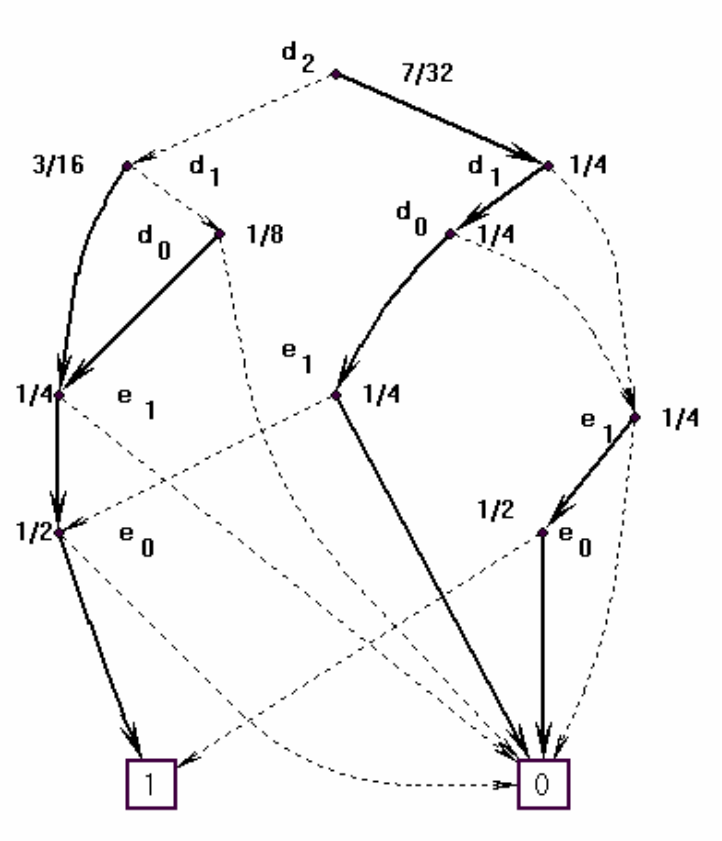

Рис. 15. Вычисление плотности функции.
Каждая вершина помечена долей присваиваний переменным, дающих в результате значение 1

Как демонстрирует это приложение, основанный на OBDD символьный анализ может быть применен к системам со сложными вариациями параметров. Хотя при этом требуется упростить проблему таким образом, чтобы было возможно рассматривать только дискретные вариации, все-таки могут быть получены хорошие результаты. Основным достоинством этого подхо-

да по сравнению с другими упрощенными методами вероятностного анализа (например, измерение управляемости и наблюдаемости) является то, что он точно вычисляет влияние корреляций между стохастическими значениями.

4.5. Анализ конечных автоматов

Многие задачи в таких областях, как верификация цифровых систем, проверка протоколов и оптимизация последовательных систем, требуют детальной характеризации конечного автомата над последовательностью переходов. Классические алгоритмы для задачи характеризации осуществляют явное представление автомата в виде диаграммы переходов и затем анализируют структуру его путей и циклов. Для этого применения также были разработаны более специальные методы, использующие характеристики верифицируемой системы, например, того, что схема является синхронной и детерминированной и что спецификация требует анализировать только ограниченное количество тактов. Однако эти методы непрактичны, если количество состояний становится большим. К сожалению, даже относительно маленькие цифровые системы могут иметь очень большие пространства состояний. К примеру, один 32-битный регистр может иметь около $4 * 10^9$ состояний.

Существуют «символьные» методы описания автоматов, в которых структура переходов представлена в виде булевой функции. Вначале выбираются бинарное кодирование состояний системы и входной алфавит. Поведение, связанное со следующим состоянием, описывается как отношение, задаваемое характеристической функцией $\delta(\vec{x},\vec{o},\vec{n})$, которая принимает значение 1 в случае, когда входное значение $\vec{x}$ может вызвать переход из состояния $\vec{o}$ в состояние $\vec{n}$.

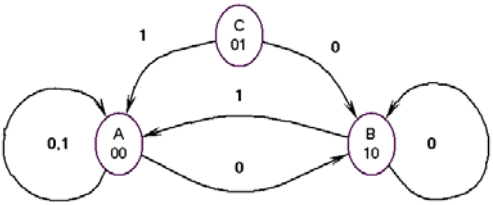

Рис. 16. Явное представление недетерминированного конечного автомата. Размер представления растет линейно относительно количества состояний

На рис. 17 приведен пример OBDD-представления недетерминированного автомата, диаграмма переходов которого изображена на рис. 16. В этом примере три возможных состояния представлены с помощью двух бинарных значений посредством кодирования: $\sigma(A) = [0, 0]$, $\sigma(B) = [1, 0]$, $\sigma(C) = [0, 1]$. Заметим, что неиспользованное значение [1, 1] может рассматриваться как «безразличное» для аргументов $\vec{o}$ и $\vec{n}$ функции δ. В OBDD на рис. 17 эта комбинация используется как альтернативный код состояния C для упрощения OBDD-представления.

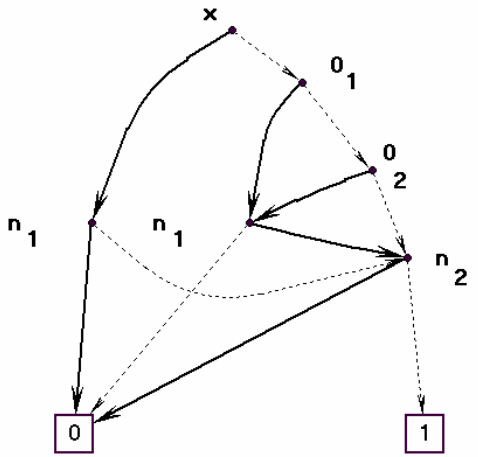

Рис. 17. Символьное представление недетерминированного конечного автомата. Количество переменных растет логарифмически относительно количества состояний

Для такого маленького автомата OBDD-представление не является улучшением по сравнению с явным представлением. Но для более сложных систем OBDD-представление может быть значительно меньшим. Основываясь на оценках сложности сетей с ограниченной шириной, описанных выше, можно охарактеризовать некоторые условия, при которых OBDD-диаграмма, кодирующая отношение переходов системы, растет линейно относительно количества компонентов системы, тогда как количество состояний растет экспоненциально. В частности, это свойство имеет место в тех случаях, когда справедливы следующие два условия:

(1) компоненты системы связаны в линейную или древесную структуру,

(2) каждый компонент поддерживает только ограниченное количество информации о состоянии других компонентов.

Как показывает пример на рис. 5, это ограничение выполняется также для кольцевых структур, так как кольцо может быть «сплющено» в линейную цепь двунаправленных ссылок.

Если имеется OBDD-представление, то свойства системы с конечным числом состояний могут быть выражены в терминах уравнений с неподвижной точкой над функцией перехода, и эти уравнения могут быть решены с использованием итеративных методов, подобных тем, которые использовались для нахождения транзитивного замыкания отношения. Рассмотрим, например, задачу определения множества состояний, достижимых из начального состояния с бинарной кодировкой $\vec{q}$ после некоторой последовательности переходов. Пусть отношение S обозначает условия, при которых для некоторого начального значения $\vec{x}$ может существовать переход из состояния $\vec{o}$ в состояние $\vec{n}$. Это отношение имеет характеристическую функцию

$$\chi_S\ (\vec{o},\vec{n}) = \exists\ \vec{x}\ [\ \delta(\vec{x},\vec{o},\vec{n})\ ].$$

Множество состояний, достижимых из состояния $\vec{q}$, имеет характеристическую функцию

$$\chi_R\ (\vec{s}\ ) = \chi_S\ (\vec{q},\vec{s}\ ).$$

С помощью этого метода были проанализированы системы с числом состояний более 10^{20}, это намного больше, чем было возможно анализировать с помощью методов, основанных на явных представлениях диаграмм переходов. Известны также усовершенствования, способствовавшие ускорению сходимости и уменьшению размера промежуточных OBDD.

К сожалению, те характеристики системы, которые гарантируют эффективность OBDD-представления отношения перехода, не дают возможности найти подходящие верхние границы для результатов, генерируемых символьным анализатором конечных автоматов. Например, существуют примеры систем, имеющих линейную структуру внутренних соединений, для которых характеристическая функция множества достижимых состояний требует OBDD-представления экспоненциального размера. С другой стороны, на практике было обнаружено, что большое количество реальных систем могут анализироваться рассмотренными методами.

Одним из приложений анализа конечных автоматов является проверка корректности последовательной цифровой схемы. Например, можно доказывать, что конечный автомат, выведенный из спецификаций системы, эквивалентен другому автомату, выведенному из схемы, даже если его кодирование состояний отличается от первого. Для этого применения также были разработаны более специальные методы, использующие характеристики верифицируемой системы, например такие, что схема является синхронной и детерминированной и что спецификация требует анализировать только ограниченное количество тактов. С их помощью, например, были верифицированы конвейерные маршруты данных, содержащие более 1000 бит состояния регистра. Такая система, имеющая более чем 10^{300} состояний, превосходит возможности современных символьных методов, основанных на диаграммах переходов.

4.6. Другие области применения

Исторически сложилось так, что вначале OBDD применялись в основном в задачах разработки цифровых систем, верификации и тестирования. Однако в последнее время они получили применение в других областях. Например, техники с использованием неподвижной точки, применяемые при символьном анализе конечных автоматов, могут быть полезными при решении множества проблем математической логики и формальных языков, если предметные области в них являются конечными. Также было обнаружено, что проблемы многих приложений могут быть сформулированы в виде набора уравнений над булевыми алгебрами, которые поддаются решению некоторым унифицированным образом.

В области искусственного интеллекта была разработана система поддержки истинности, основанная на OBDD. Она использует OBDD для представления «базы данных», то есть известных отношений между элементами. Было обнаружено, что с помощью кодирования базы данных в этой форме система может делать заключения быстрее, чем при традиционном подходе, когда она просто ведет список «известных фактов». Например, распознавание того, согласуется ли новый факт с множеством существующих фактов или следует ли он из них, представляет собой простую проверку импликации.

4.7. Возможности улучшения

Хотя существует широкое множество различных проблем, которые были успешно решены с помощью основанной на OBDD символьной обработ-

ки, имеется немало задач, для решения которых требуются более адекватные методы. Конечно, большинство этих задач являются NP-трудными и иногда даже PSPACE-трудными. Следовательно, мало вероятно, что для их решения может быть найден метод с полиномиальной сложностью. Лучшее, что можно сделать, — это разработать методы, дающие приемлемую производительность для большинства интересующих нас задач.

Одна из возможностей на пути к этому состоит в улучшении самого представления. При работе с цифровыми системами, содержащими умножители и другие функции, включающие сложные отношения между управляющими сигналами и сигналами данных, OBDD-представления быстро становятся нереально большого размера. Было предложено несколько методов, основанных на тех же общих принципах, что и OBDD, но с менее строгими ограничениями на структуры данных. Среди них IFE-представления (или «If-Then-Else дэги»), где условие проверки в каждой вершине может быть более сложной функцией, чем обычная проверка переменной, «свободные бинарные диаграммы решений», в которых ограничение на порядок переменных заменяется на условие, что ни один путь из корневой вершины в терминальную не проверяет одну и ту же переменную более одного раза, «однократно ветвящиеся программы», которые обладают многими полезными свойствами OBDD, включая эффективный (хотя и вероятностный) метод проверки эквивалентности. В каждом из этих расширений метода, основанного на OBDD, делается попытка найти компромисс между компактностью представления и трудностью его построения или проверки его свойств.

Во многих задачах комбинаторной оптимизации символьные методы, использующие OBDD, не показали столь же высокую производительность, как более традиционные методы. Это связано с тем, что в этих задачах обычно требуется найти только одно решение, удовлетворяющее некоторому критерию оптимальности. Напротив, большинство основанных на OBDD подходов вычисляют все возможные решения и затем выбирают лучшее из них. К сожалению, многие задачи имеют слишком много решений, чтобы можно было их символически закодировать. Более традиционные методы поиска, такие, как метод ветвей и границ, часто оказываются более эффективными и способными решать большие задачи. Одна из возможностей улучшения состоит в использовании идеи «ленивого» или «отложенного» исполнения при обработке с использованием OBDD. Вместо того, чтобы немедленно создавать полное представление каждой функции в процессе выполнения последовательности операций, следует пытаться

строить только те OBDD, которые нужны для вывода окончательной информации.

5. ЗАМЕЧАНИЯ

Представление в виде упорядоченных бинарных диаграмм определяется путем наложения ограничений на бинарные диаграммы решений, введенные Ли [26] и Экерсом [6]. Эти ограничения и получаемая в итоге каноничность представления были впервые описаны Фортуном и др. [20]. Бинарные диаграммы решений много лет использовались в качестве абстрактного представления булевых функций. Под названием «программы ветвления» они были тщательно исследованы специалистами по теории сложности вычислений (см., например, Вегенер [37], Мейнел [30]). В своем изложении мы базируемся на работах Брянта [13-15], который ввел понятие OBDD в 1986 году. Дополнительная библиография по этой тематике может быть найдена в обзоре Мейнела и др. [31]. Более общее понятие логического фрагмента [3], которое позволяет представлять частичные функции, а также не только логические, но и другие функции от логических переменных, использовалось при исследовании схем Янова [2, 4].

Проблема нахождения оптимального упорядочения переменных для OBDD является NP-трудной [10, 36]. Эвристические методы, позволяющие находить хорошие способы упорядочения переменных для OBDD, предложены в работах Фуджита и др. [21], Малика и др. [28], Джонга и др. [22]. Свойство A доказано в работе [14], а свойство B — в работе [12], символьный симулятор COSMOS описан в [17]. Берман [8] и Макмиллан [29] вывели границы для нескольких классов сетей с «ограниченной шириной». Макмиллан [29] обобщил эту технику до *древовидного расположения*, в котором сеть организована в виде дерева логических блоков с коэффициентом ветвления b и первичным выходным значением, производимым корневым блоком.

Вопросы реализации OBDD на рабочих станциях с большой физической памятью рассматривались Брэйсом и др. в работе [11]. Другие исследователи [25, 33] успешно использовали для их реализации векторные операции и достигли хороших результатов при обработке OBDD на мультипроцессорах с разделенной памятью.

Усовершенствованиям базовой структуры OBDD, которые направлены на экономию памяти, посвящен целый ряд работ. Среди них — использование единственного многокорневого графа для представления всех требуе-

мых функций [13, 23, 32, 34], снабжение дуг метками для обозначения булевого вычитания [11, 23, 27, 32] и обобщение понятия областей с конечным числом состояний [35]. Ряд авторов рассматривал методы, основанные на тех же общих принципах, что и OBDD, но с менее строгими ограничениями структуры данных [7, 9, 16, 19, 23, 24, 37]. Возможности улучшения процессов обработки с использованием OBDD на основе идеи «ленивого» (или «отложенного») исполнения рассматривались в [5, 18].

СПИСОК ЛИТЕРАТУРЫ

1. **Евстигнеев В.А., Касьянов В.Н.** Теория графов: алгоритмы обработки бесконтурных графов — Новосибирск: Наука, 1998.
2. **Ершов А.П.** Введение в теоретическое программирование. — Новосибирск: Наука, 1977.
3. **Касьянов В.Н., Сабельфельд В.К.** Сборник заданий по практикуму на ЭВМ. — М.: Наука, 1986.
4. **Котов В.Е.** Введение в теорию схем программ. — Новосибирск: Наука, 1978.
5. **Abelson H., Sussman G. J., Sussman J.** Structure and interpretation of computer programs. — MIT Press, Cambridge, Mass, 1988. — P. 261–264.
6. **Akers S.** B. Binary decision diagrams // IEEE Trans. Comput. — 1978. — Vol. C-27, N 6. — P. 509–516.
7. **Ashar P., Devadas S., Ghosh A.** Boolean satisfiability and equivalence checking using general binary decision diagrams // The International Conference on Computer Design. — IEEE, New York , 1991. — P. 259–264.
8. **Berman C. L.** Ordered binary decision diagrams and circuit structure // The International Conference on Computer Design. — IEEE, New York, 1989. — P. 392–395.
9. **Blum M. W., Chandra A. K.** Equivalence of free Boolean graphs can be decided probabilistically in polynomial time // Inf. Process. Lett. — 1980. — Vol. 10, N 2. — P. 80–82.
10. **Bollig B., Wegener I.** Improving the variable ordering of OBDDs is NP-complete // IEEE Trans. Computers. — 1996. — Vol. 45. — P. 993–1002.
11. **Brace K. S., Bryant R. E., Rudell R. L.** Efficient implementation of a BDD package. In Proceedings of the 27th ACM / IEEE Design Automation Conference. — ACM, New York, 1990.— P. 40–45.
12. **Brayton R. K., Hachtel G. D., McMullen C. T., Sangiovanni-Vincentelli A. L.** Logic Minimization Algorithms for VLSI Synthesis. — Kluwer Academic Publishers, Boston, 1984.
13. **Bryant R. E.** Graph-based algorithms for Boolean function manipulation // IEEE Trans. Comput. — 1986. — Vol. C-35, N 6. — P. 677–691.

14. **Bryant R. E.** On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication // IEEE Trans. Comput. — 1991. — Vol. 40, N 2. — P. 205–213.
15. **Bryant R. E.** Simbolic Boolean manipulation with ordered binary-decision diagrams // ACM Computing Surveys. — 1992. — Vol. 24, N 3. — P. 293–318.
16. **Burch J. R.** Using BDDs to verify multipliers // Proc. of the 28th ACM / IEEE Design Automation Conference. — ACM, New York, 1991. — P. 408–412.
17. **Cho K., Bryant R. E.** Test pattern generation for sequential MOS circuits by symbolic fault simulation // Proc. of the 26th ACM / IEEE Design Automation Conference. — ACM, New York, 1989. — P. 418–423.
18. **Deguchi Y., Ishiura N., Yajima S.** Probabilistic CTSS: Analysis of timing error probability in asynchronous logic circuits // Proc. of the 28th ACM / IEEE Design Automation Conference. — ACM, New York, 1991. — P. 650–655.
19. **Finkbeiner B.** Language containment checking with nondeterministic BDDs // Lect. Notes Comput. Sci. — 2001. — Vol. 2031. — P. 24–38.
20. **Fortune S., Hopcroft J., Schmidt E. M.** The complexity of equivalence and containment for free single variable program schemes // Lect. Notes Comput. Sci. — 1978. — Vol. 62. — P. 227–240.
21. **Fujita M., Fujisawa H., Kawato N.** Evaluations and improvements of a Boolean comparison program based on binary decision diagrams // The Internat. Conf.on Computer-Aided Design. — IEEE, New York, 1988. — P. 2–5.
22. **Jeong S.-W., Plessier B., Hachtel G. D., Somenzi F.** Variable ordering and selection for FSM traversal // The Internat. Conf. on Computer-Aided Design. — IEEE, New York, 1991. — P. 476–479.
23. **Karplus K.** Using if-then-else DAGs for multi-level logic minimization // Advance Research in VLSI. — MIT Press, Cambridge, Mass., 1989. — P. 101–118.
24. **Katayama T., Ochi H., Tsuda T.** An algorithm for generating genetic BDDs // IEICE Trans. Fundamentals. — 2000. — Vol. E88-A, N. 12. — P. 2505–2512.
25. **Kimura S., Clark E. M.** A parallel algorithm for constructing binary decision diagrams // The Intern. Conf. on Computer Design. — IEEE, New York, 1990. — P. 220–223.
26. **Lee C. Y.** Representation of switching circuits by binary-decision programs // Bell System Tech. J. — 1959. — Vol. 38, N 4. — P. 985–999.
27. **Madre J. C., Billon J. P.** Proving circuit correctness using formal comparison between expected and extracted behaviour // Proc. of the 25th ACM / IEEE Design Automation Conference. — ACM, New York, 1988. — P. 205–210.
28. **Malik S., Wang A., Brayton K. K., Sangiovann-Vincentelli A.** Logic verification using binary decision diagrams in a logic synthesis environment // The Internat. Conf. on Computer-Aided Design. — IEEE, New York, 1966. — P. 6–9.
29. **McMillan K. L.** Symbolic model checking: An approach to the state explosion problem: PhD thesis, School of Computer Science, Carnegie-Mellon Univ, 1992.
30. **Meinel C.** Modified branching programs and their computational power // Lect. Notes Comput. Sci. — 1990. — Vol. 370.

31. **Meinel C., Stangier C.** Data structure for Boolean function. BDDs — foundations and aplications // Lect. Notes Comput. Sci. — 2001. — Vol. 370. — P.61–78.
32. **Minato S., Ishiura N., Yajima S.** Shared binary decision diagram with attributed edges for efficient Boolean function manipulation // Proc. of the 27th ACM / IEEE Design Automation Conf. — ACM, New York, 1990. — P. 52–57.
33. **Ochi H., Ishiura N., Yajima S.** Breadth-first manipulation of SBDD of function for vector processing // Proc. of the 28th ACM / IEEE Design Automation Conf. — ACM, New York, 1991. — P. 413-416.
34. **Reeves D. S., Irwin M. J.** Fast methods for switch-level verification of MOS circuits // IEEE Trans. CAD / IC CAD. — 1987. — Vol. 6, N 5. — P. 766–779.
35. **Srinivasan A., Kam T., Malik S., Brayton R. K.** Algorithms for discrete function manipulation // The Internat. Conf. on Computer-Aided Design. — IEEE, New York, 1990. — P. 92–95.
36. **Tani S., Hamaguchi K., Yajima S.** The complexity of the optimal variable ordering problems of shared binary decision diagrams // Lect. Notes Comput. Sci. — 1993. — Vol. 762. — P. 389–398.
37. **Wegener I.** On the complexity of branching programs and decision trees for clique functions // J. ACM. — 1988. — Vol. 35, N 2. — P. 461–471.