УДК 51.681.3

ОБ АЛГОРИТМАХ ПРЕДСТАВЛЕНИЯ ТРАНЗИЦИОННЫХ СИСТЕМ С ПОМОЩЬЮ БИНАРНЫХ ДИАГРАМ РЕШЕНИЙ

Багрий Р.О.
Хмельницкий технологический университет "ПОДОЛЬЯ"
(email: ruslan@beta.tup.km.ua)

Аннотация. Под реактивной системой понимается система, состоящая из нескольких компонент-объектов, взаимодействующих между собой и с окружающей их средой. Класс реактивных систем включает операционные системы, мультипрограммные среды, параллельные и распределенные дискретные системы, дискретные системы реального времени, вычислительные сети и т. д. В данной работе описываются алгоритмы представления конечных транзиционных систем с помощью упорядоченных бинарных таблиц решений (так называемых УБДР или OBDD – ordered binary decision diagrams) и некоторые экспериментальные данные, полученные с помощью этих алгоритмов.

1. Введение

Реактивную систему и ее окружение представляют, как правило, в виде параллельной системы, в которой все компоненты системы и ее окружение функционируют параллельно, взаимодействуя друг с другом. Семантика реактивной системы в большинстве случаев описывается протоколами взаимодействия и декларацией определенных свойств, которым должна удовлетворять такая система. Под свойством понимается множество ожидаемых поведений, которые должны быть присущи реактивной системе. При этом сама система представляется в виде графа (как павило, размеченного), а ее свойства записываются в виде формул подходящей темпоральной логики. Подход, использующий методы темпоральной логики и применяемый для спецификации свойств реактивных систем при решении задач моделирования и анализа реальных систем, основывается на предположении о том, что как сама система, так и ее окружение, моделируются как дискретные процессы. Элементами параллельной дискретной системы являются процессы, параллельно функционирующие во времени. Они могут быть независимыми друг от друга и конкурировать за общие ресурсы (асинхронные конкурентные системы), взаимодействовать между собой, выполняя общую задачу (кооперативные распределенные системы) и т.д.

Теория реактивных систем в основном предназначена для решения проблем спецификации и верификации свойств. Под спецификацией понимают разработку математической модели реактивной системы [1],. Транзиционная система (ТС), есть одним из математических формализмов, который используется для изучения статических и динамических свойств процессов. При помощи этого формализма можно моделировать спецификации требований к поведению реальной системы. Типичными требованиями к параллельным системам являются, например, гарантия недопустимости определенных (аварийных) ситуаций в процессе ее функционирования, требование отсутствия тупиковых ситуаций при работе с общими ресурсами, требование взаимного исключения при вхождении в критическую секцию нескольких параллельных процессов, гарантия отсутствия бесконечного ожидания ресурса некоторым процессом, гарантия достижения некоторой цели (за определенное время) и т.д. Требования к поведению делятся на два основных класса (классификация Л.Лэмпорта [2],[3]):

- требования корректности (safety) – гарантия того, что некоторое свойство сохраняется во всех состояниях всех вычислений системы;
- требования жизнеспособности (liveness) – гарантия того, что некоторое событие когда-то в будущем произойдет в системе, или, другими словами, некоторое свойство будет истинно в некотором достижимом состоянии системы.

2. Необходимые понятия и определения

Спецификации требований к поведению системы, как отмечалось, выше, удобно задавать формулами темпоральной логики. Верификацией называется доказательство истинности некоторого требования к поведению на некоторой вычислительной модели реактивной системы за ограниченное время. Верифицировать реактивные системы невозможно путем тестирования, так как количество возможных вариантов ее поведения может быть астрономическим числом или даже бесконечным. Основная проблема, которая возникает при работе с конечными ТС, – это огромное число состояний такой ТС. Приводимые ниже алгоритмы ориентрованы, прежде всего, на решение этой проблемы с помощью представления ТС в виде OBDD.

Алгоритм обхода состояний системы и проверки истинности темпоральной формулы в каждом состоянии получил название Model Checking [4],[5]. Он является полностью автоматическим методом для проверки правильности свойств. Исходными данными для метода Model Checking есть модель системы (в виде ТС) и формулировка необходимых свойств. Алгоритм, по завершению своей работы, должен определить являются ли требования выполнимыми для данной модели. Во время работы алгоритму

необходимо обойти все состояния системы. При сегодняшнем уровне технических средств более менее эффективно в реальное время можно верифицировать ТС с не более, чем 10^6 состояний. Чтобы проверить выполняемость темпоральной формулы, представляющей некоторое совйство системы, нужно уметь эффективно представлять ТС с большим числом состояний.

В данное время разработано несколько подходов к решению задачи Model Checking. Одним из них есть символьное представление множества состояний и переходов между ними при помощи упорядоченных бинарных диаграмм решений (OBBD) [5],[6], благодаря возможности задания отношения переходов данной ТС с помощью булевской функции. Ниже рассматриваются размеченные конечные ТС.

Напомним, что булевой функцией называется m-арная функция $f:\{0,1\}^m \rightarrow \{0,1\}$. Множество переменных $X = \{x_1,\ldots,x_m\}$, где каждая переменная x_i может принимать одно из двух значений 0 или 1, называется алфавитом булевых переменных. Отсюда вытекает, что область значений булевой функции от m аргументов конечная и состоит ровно из 2^m элементов. OBDD позволяют манипулировать с булевыми функциями более просто и эффективно с точки зрения сложности вычислений.

OBDD, представляющая булевскую функцию, является корневым ациклическим помеченным графом (см. рис. 1,2). Главная идея использования OBDD – кодирование каждого состояния и символов входного алфавита вместе с отношением переходов транзиционной системы с помощью булевой функции. Отношение переходов представляется при помощи характеристической функции $\delta(x,a,b)$. Эта функция принимает значение 1 тогда, когда входной сигнал *x* вызывает переход из состояния *a* в состояние *b.*

3. Алгоритм построения OBDD

3.1. Общий алгоритм построения OBDD

Опишем алгоритм построения упорядоченного бинарного дерева решений (OBBD) для произвольной конечной транзиционной системы (которая в этом случае является конечным автоматом) [7].

1. Кодируем каждое состояние и символы входного алфавита бинарной последовательностью.
2. Составляем таблицу значений функции *δ*.
3. На основе таблицы строим бинарное дерево, используя рекурсивный обход от корня к вершинам.

При построении дерева используются следующие правила редукции.

- Объединение одинаковых констант и переориентирование всех входящих дуг на соответствующую константу.
- Объединение одинаковых вершин и переориентирование на нее всех входящих дуг.
- Удаление тех вершин, исходящие дуги которых ведут в одну и ту же вершину.

Перед построением OBDD, таблицу значений необходимо преобразовать в булевую функцию. Один из способов – построение СДНФ и дальнейшее ее сокращение. Рекурсивный обход от корня к вершинам применяется для возможности одновременного построения дерева с применением правил редукции. Хранение информации о конечном автомате в виде булевой функции и использование рекурсии позволяет экономить память при построении дерева решений.

Рассмотрим работу алгоритма на примере.

Предположим, что результатом кодирования некого конечного автомата есть следующая булевая функция $f(x_1,x_2,x_3) = (x_1 + x_2) * x_3$. Таблица истинности и полное дерево решений на рис.1.

x_1	X_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

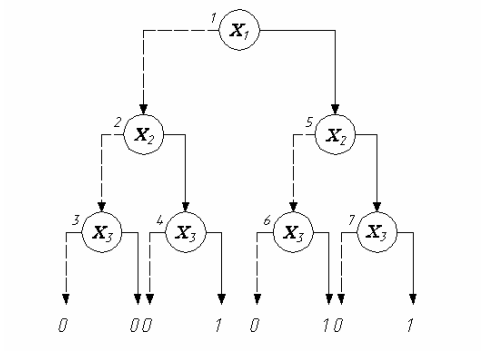


Рис.1.Таблица истинности булевой функции и ее дерево решений”

Цифрами 1, 2,…, 7 показан путь рекурсии по вершинам. Из каждой вершины исходит две дуги. Дуга изображенная пунктирной линией соответствует значению 0, а сплошной – значению 1. Применяя правила редукции, описанные выше, упрощаем дерево решений (рис. 2).

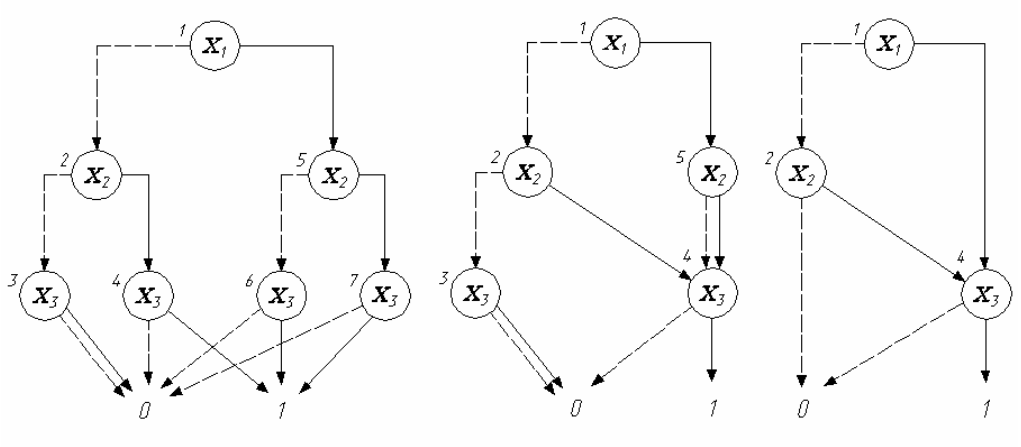


Рис. 2. Редукция дерева решений

После всех преобразований дерево решений будет иметь 5 вершин. При построении данного дерева было рассчитано 7 вершин, 4 из них удалены по правилам редукции.

3.2. Характеристика алгоритма

Данный алгоритм имеет один существенный недостаток - необходимость посещать каждую вершину дерева. Пределом эффективности работы данного алгоритма есть булевские функции с не более, чем 20 переменных. Для современных приложений этого количества недостаточно. Поэтому возникает задача усовершенствования данного алгоритма и возможности построения OBBD для булевских функций с большим числом булевых переменных.

Увеличение скорости работы алгоритма можно достичь, не рассчитывая те вершины дерева, которые будут удалены правилами редукции. Это возможно только когда заведомо (до построения) известно, что вершина будет удалена. В рассмотренном примере вершина 3 (x_3=0) будет удалена, так как две ее исходящие дуги ведут в одну и ту же константу 0. Действительно, при значениях x_1=0, x_2=0 и при любых значениях x_3 значение функции будет равно 0. Вершина 5 (x_2=1) также будет удалена, так как при x_1=1, значение функции будет зависеть только от значения переменной x_3. Итак, необходимо найти алгоритм, который позволит на стадии анализа булевой формулы выявить те интерпретации, при которых не все значения переменных влияют на результат.

Для решения поставленной подзадачи можно воспользоваться методом семантического табло. Этот метод используется для поиска интерпретаций, при которых данная формула выполнима. Он применяется к формулам логики высказываний, логики предикатов, модальной и темпоральной логик. Для построения всех интерпретаций удобно использовать помеченные деревья – отдельный случай помеченных графов [8]. При использовании метода семантического табло для логики высказываний все формулы делятся на два класса:

- α-формулы для конъюнкций,
- β-формулы для дизъюнкций;

Правила преобразований для α- и β-формул представлены в виде таблиц:

Таблица 1. Правила для α-формул.

A	α_1	α_2
$\neg\neg A$	A	
$A_1 \wedge A_2$	A_1	A_2
$\neg(A_1 \vee A_2)$	$\neg A_1$	$\neg A_2$
$\neg(A_1 \rightarrow A_2)$	A_1	$\neg A_2$
$A_1 \leftrightarrow A_2$	$A_1 \rightarrow A_2$	$A_2 \rightarrow A_1$

Таблица 2. Правила для β-формул.

B	β_1	β_2
$B_1 \vee B_2$	B_1	B_2
$\neg(B_1 \wedge B_2)$	$\neg B_1$	$\neg B_2$
$B_1 \rightarrow B_2$	$\neg B_1$	B_2
$\neg(B_1 \leftrightarrow B_2)$	$\neg(B_1 \rightarrow B_2)$	$\neg(B_2 \rightarrow B_1)$

3.3. Алгоритм построения семантического табло

Алгоритм построения семантического табло имеет следующий вид. Пусть дана формула А.

1. Корень дерева помечаем формулой А.
2. Если формула является α-формулой, то строим новую вершину-сына формулы, которую рассматривали, и размечаем ее по правилам преобразований для этой группы формул.
3. Если формула является β-формулой, то строим две новых вершины-сыновей рассматриваемой вершины и размечаем их по правилам преобразований для этой группы формул.
4. Если вершина помечена множеством атомарных формул, то если оно включает контрарные пары (т.е. пары вида (p, ¬ p)), то помечаем данную вершину символом ×, в противном случае, символом ○.
5. Если все листья дерева рассмотрены и помечены символами × или ○, то закончить роботу алгоритма, иначе перейти к шагу 2.

Имея множество таких интерпретаций и, проанализировав их, можно до построения дерева решений определить какие из вершин не влияют на результат. Это множество интерпретаций определяет условия, при которых функция будет выполнима.

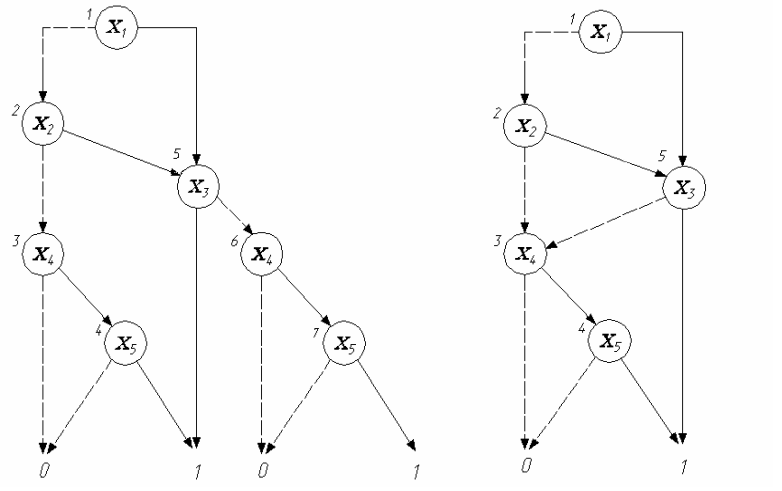
Так, для предыдущего примера, метод семантического табло формирует следующее множество интерпретаций: x_1, x_3 и x_2, x_3. Если все эти условия нарушены, то есть текущие значения переменных противоречат всем интерпретациям, это означает, что в дальнейшем построении дерева нет необходимости и можно исходящую дугу перевести на константу 0. Примером этого может служить вершина 3. И наоборот, если какое-то условие выполнилось при текущих значениях переменных, это означает, что в дальнейшем построении дерева нет необходимости и можно исходящую дугу перевести на константу 1. Также, при рекурсивном обходе дерева, перед переходом на следующую вершину, анализируя условия выполнимости, можно определить, будет ли влиять значение переменной на результат, и если нет, то рекурсия может пропустить данную вершину. Примером такого анализа есть вершина 5 из рассматриваемого примера. И действительно, при x_1=1, в первом условии выполнимости следующей значимой переменной будет x_3, а во втором хоть и есть зависимость от x_2, но это условие перекрывается первым ввиду принятого порядка сортировки переменных (x_1> x_2> x_3). Данный алгоритм ускоряет построение дерева решений, однако не настолько эффективно как это требуется в настоящее время для приложений.

3.4. Улучшение алгоритма

При построении дерева решений часто возникает ситуация, когда формируются одинаковые по поведению ветви. Многократно используя правила редукции над этими ветвями, их можно объединить. Количество таких ветвей и их вложенность зависит от формулы и может быть настолько значительным, что построение дерева займет неопределенное количество времени. Данную проблему принято решать, разбивая формулу на несколько простых составляющих, далее строить для каждой из них деревья решений, и затем объединять их в одно общее дерево, используя метод APPLY[7].

Недостатком данного метода является сложность определения простых составляющих формулы. Отсюда возникает следующая подзадача – разработать альтернативный алгоритм построения таких деревьев.

Сначала необходимо выяснить, при каких обстоятельствах формируются одинаковые ветви. Рассмотрим новый пример: дана булевская формула – $f(x_1, x_2, x_3, x_4, x_5) = (x_1 + x_2) * x_3 + x_4 * x_5$ и интерпретации, при которых она выполнима – $x_1, x_3;\ x_2, x_3;\ x_4, x_5;$ Шаг, на котором видно две идентичные ветви изображен на рис. 3а), а редуцированное дерево – на рис. 3б).



а) *б)*

Рис. 3. OBDD для функции $f(x_1, x_2, x_3, x_4, x_5) = (x_1 + x_2) * x_3 + x_4 * x_5$

Ветвь-дубликат – это реализация интерпретации x_4, x_5. Двойное построение происходит из-за того, что при нарушении и первого и второго условия выполнимости результат функции продолжает зависеть от последнего условия. Для того чтобы дважды не просчитывать данную интерпретацию, для каждого условия выполнимости необходимо запоминать вершину начала, чтобы при повторном обращении сразу переводить дуги на нее.

4. Эксперименты

Ниже приводятся некоторые экспериментальные данные скоростных характеристик вышеизложенных алгоритмов, реализованных в виде программной системы. При помощи этой системы было протестировано несколько классов булевых функций. Все расчеты проводились на компьютере с процессором Pentium 4 3.0Гц, с 512Мб оперативной памяти. Так как скорость построения зависит в большей степени от количества просчитанных вершин, то они также приведены в таблице.

Таблица 3. Временные характеристики построение УБДР.

№	Булевая функция	кол. вершин/кол. просчитанных вершин/макс. кол. вершин	Время, мс
1	¬ (¬ (x1 ∨ x2) ∧ (x3 ∨ x4 ∧ x5)) ∨ x6	6/6/64	31
2	¬x1∧¬x2∧¬x3∧¬x4∧x5∨¬x1∧x2∧¬x3∧¬x4∧x5∨ ¬x1∧¬x2∧x3∧x4∧x5∨¬x1∧x2∧x3∧x4∧x5∨ x1∧¬x2∧x3∧¬x4∧¬x5∨x1∧x2∧x3∧¬x4∧¬x5∨ x1∧¬x2∧¬x3∧x4∧¬x5∨x1∧x2∧¬x3∧x4∧¬x5	9/23/32	47
3	¬x1∧¬x3∧¬x4∧x5∨¬x1∧x3∧x4∧x5∨ x1∧x3∧¬x4∧¬x5∨x1∧¬x3∧x4∧¬x5	9/11/16	47
4	¬(¬(x1∨x2)∧(x3∨x4∧x5))∨x6∧¬(¬(x7∨x8)∧ (x9∨x10∧x11))∨x12∧¬(¬(x13∨x14)∧(x15∨x16∧x17))∨ x18∧¬(¬(x19∨x20)∧(x21∨x22∧x23))∨x24∨x25∨x26∧ x27∧(¬x28∨x29)∧x30∧(x31>x32)∧(x33∨¬x34)∨ (x35>x36)∨x37∨¬x38∧(x39>x40)∧(x41∨x42)∧x43∧ (x44>x45)∧x46∧(x47∨x48)∧x49∨x50∨¬(¬(x51>x52) ∧(x53∨x54∧x55))∧x56∨¬x57∧x58∨(x59>x60)∨x61∨ ¬x62∨x63∧x64∨(x65>x66)∧x67∧¬x68∨x69∧x70∨x71∨ x72∨¬x73∧x74∨(x75>¬x76)∧¬x77∧x78∨x79∧¬x80∨ x81∨¬x82∨x83∧x84∨(¬x85>x86)∧x87∧x88∨¬x89∧x90∨ ¬x91∨x92∨¬x93∧x94∨(x95>¬x96)∧x97∧¬x98∨x99∧x100	$100/173/2^{100}$	1422

В первом примере приведена простая булевская функция с 6-ю переменными. Результат работы программы идеален – рассчитывались только те вершины, которые необходимы. Во втором примере функция задана в ДНФ. Данный вид представления функций является самым неоптимальным для построения УБДР – рассчитано на ~30% больше вершин, чем необходимо. Намного эффективнее задавать формулы в виде сокращенных ДНФ – количество лишних вершин составляет ~12% (в данном примере 2 вершины). В последнем примере приведена бесповторная булевская функция из 100 переменных. Общее время построения заняло 1.5 секунды, количество рассчитанных вершин намного меньше по сравнению с максимальным количеством (2^{100}).

В Интернете доступны несколько программных реализаций алгоритмов построение OBDD. Одна из них VisBdd – Visualization of the Bdd-ITE algorithm. Это программа строит деревья решений для первых трех случаев примерно за тоже время, что и исследуемая программа, а последний пример не смогла рассчитать из-за нехватки памяти.

5.Выводы

В данной работе представлены три алгоритма построения OBDD. Сравнивая результаты применения этих алгоритмов с результатами программы VisBdd, использующей стандартный алгоритм ITE, можно сделать вывод о том, что при небольших количествах переменных (до 20) – алгоритм и его программная реализация не имеют сильных разбросов по времени (эффективности). Когда количество переменных булевской функции приближается к реальным задачам (например при моделировании 32-ух разрядного регистра), на первое место выходят требования к объему памяти и количеству рассчитанных

вершин. Предложенный алгоритм дает возможность решать задачи представления булевской функции с большим числом состояний (100 переменных и больше).

Список литературы

1. Clarke E.M., Emerson E.A., Systla A.P.: Automatic verification of finite-state concurrent systems using temporal-logic specifications. ACM Transactions on Programming Languages and Systems, 8(2): p.244-263. 1986.
2. .Lamport L.: Specifying concurrent program modules. ACM Trans. Prog. Lang. Syst. 5, p.190-222, 1983.
3. Lamport L.: What good is temporal logic. Proc.IFIP Congress,North-Holland, p.657-668, 1983.
4. Olivero A., Yovine S.: Kronos: a tool for verifying real-time systems. User's Guide and Reference Manual. VERIMAG, Grenoble, France, 1992.
5. Clarke E.M., Emerson E.A., Systla A.P.: Automatic verification of finite-state concurrent systems using temporal-logic specifications. ACM Transactions on Programming Languages and Systems, 8(2): p.244-263. 1986.
6. Clarke E.M., Emerson E.A.: Design and synthesis of synchronisation skeletons using branching-time temporal logic. In Workshop on Logic of Program, Lecture Notes in Comp.Sc. 131. Springer-Verlag, 1981.
7. Clarke E.M.,Bernd-Holger Schlingloff: Model Checking. Handbook of automated reasoning. p. 1468-147,2001.
8. Knut D.E., Bendix P.B. Simple word problem in universal algebras. In J. Leech editor, Computation Problem in Abstract Algebra, Pergamon Press, Oxford, 1970.