ТЕОРЕТИЧЕСКИЙ И ПРИКЛАДНОЙ НАУЧНО-ТЕХНИЧЕСКИЙ ЖУРНАЛ

Издается с сентября 2010 г.

Журнал издается при поддержке Отделения математических наук РАН, Отделения нанотехнологий и информационных технологий РАН, МГУ имени М.В. Ломоносова, МГТУ имени Н.Э. Баумана, ОАО "Концерн "Сириус".

СОДЕРЖАНИЕ

Журнал зарегистрирован в Федеральной службе по надзору в сфере связи, информационных технологий и массовых коммуникаций. Свидетельство о регистрации ПИ № ФС77-38590 от 24 декабря 2009 г.

Журнал распространяется по подписке, которую можно оформить в любом почтовом отделении (индексы: по каталогу агентства "Роспечать" — **22765**, по Объединенному каталогу "Пресса России" — **39795**) или непосредственно в редакции.
Тел.: (499) 269-53-97. Факс: (499) 269-55-10.
Http://novtex.ru/pi.html E-mail: prin@novtex.ru

Журнал включен в систему Российского индекса научного цитирования.
Журнал входит в Перечень научных журналов, в которых по рекомендации ВАК РФ должны быть опубликованы научные результаты диссертаций на соискание ученой степени доктора и кандидата наук.

THEORETICAL AND APPLIED SCIENTIFIC AND TECHNICAL JOURNAL

SOFTWARE ENGINEERING

PROGRAMMNAYA INGENERIA

№ 7
July
2014

Published since September 2010

CONTENTS

Information about the jornal is available online at:
http://novtex.ru/pi.html, e-mail:prin@novtex.ru

УДК 004.4'416

Н. И. Вьюкова, ст. науч. сотр., e-mail: niva@niisi.msk.ru,
В. А. Галатенко, д-р физ.-мат. наук, ст. науч. сотр., e-mail; galat@niisi.msk.ru,
С. В. Самборский, ст. науч. сотр., НИИ системных исследований РАН, г. Москва

К вопросу об оптимальной линеаризации программ

Представлены два метода оптимальной линеаризации программ с точки зрения минимизации числа переходов. Первый из них относится к простому случаю, когда принимаются во внимание только безусловные переходы. Эта задача может быть сформулирована как поиск оптимального покрытия путями вершин графа, являющегося подграфом графа потоков управления (Control Flow Graph — CFG). Рассматриваемый подграф (Fall Through Graph — FTG) включает только дуги, соответствующие безусловным переходам, и соединяемые ими вершины. Метод заключается в построении оптимального покрытия FTG путями и циклами с последующим размыканием циклов. Сложность алгоритма O(N), где N — число вершин FTG.

В заключительном разделе рассматривается NP-полная задача оптимального покрытия путями произвольного орграфа, к которой сводится задача оптимальной линеаризации программ с учетом всех переходов, включая условные. Предлагается решение рекурсивным перебором, основанным на построении оптимального покрытия путями и циклами с последующим размыканием циклов.

Ключевые слова: *оптимизация переходов, линеаризация программы, покрытие путями*

N. I. Vyukova, V. A. Galatenko, S. V. Samborskij

On the Issue of Optimal Program Linearization

The paper discusses two methods for optimal program linearization from the viewpoint of minimizing the number of unconditional jumps. The first one relates to the simple case when only unconditional jumps are considered. This task can be formulated as the task of finding optimal path covering for a subgraph of CFG (Control Flow Graph). The subgraph in question (Fall Through Graph — FTG) includes only fallthrough edges and the vertices connected with them. The method implies finding the optimal "path and cycle" covering of FTG with subsequent breaking of cycles. The complexity of the algorithm is O(N) where N is the number of vertices in FTG.

The final section of the paper discusses the NP-complete task of path covering of an arbitrary digraph which arises during optimal program linearization with respect to all kinds of jumps including conditional ones. We propose a recursive enumeration based on building the optimal "path and cycle" covering of the digraph with subsequent breaking of cycles.

Keywords: *jump optimization, program linearization, path covering*

Введение

В работе [1] рассмотрена задача оптимизации переходов в системе динамической компиляции. Смысл оптимизации заключается в том, чтобы исключить наиболее частые безусловные переходы, расположив соответствующие линейные участки подряд, один за другим.

Математически эта задача формулируется как задача покрытия путями вершин взвешенного ориентированного графа. Рассматривается граф "провалов" FTG (*Fall Through Graph*), полученный из графа потоков управления (CFG — *Control Flow Graph*) удалением всех дуг, не являющихся безусловными переходами, а также всех вершин, не связанных оставшимися дугами. Удаляются также петли, т. е. дуги вида (v, v), где v — вершина графа. Веса дуг отражают частоту соответствующего перехода. Требуется найти покрытие вершин FTG путями, имеющее максимальный вес.

Покрытием вершин графа путями называется такая совокупность простых путей, что каждая вершина принадлежит в точности одному пути. При этом допускаются пути нулевой длины, состоящие из одной вершины.

В работе [1] предложен эвристический, "жадный" алгоритм нахождения наилучшего покрытия путями

вершин FTG, имеющий сложность $O(N^2)$. В настоящей статье представлен алгоритм точного решения данной задачи, имеющий сложность $O(N)$. Метод заключается в построении оптимального покрытия FTG путями и циклами с последующим размыканием циклов. Простое решение оказывается возможным в силу ограниченности рассматриваемого класса графов. Отметим, что сужение класса рассматриваемых графов позволяет решить за линейное время также задачу оптимизации переходов в более общей постановке [2].

В заключительном разделе рассмотрена NP-полная задача оптимального покрытия путями произвольного орграфа, к которой сводится задача оптимальной линеаризации программ с учетом всех переходов, включая условные. Предлагается решение рекурсивным перебором, основанным на построении оптимального покрытия путями и циклами с последующим размыканием циклов. Можно рассчитывать на эффективность подобных методов для орграфов с небольшим числом циклов. Этот подход также применим для поиска субоптимальных покрытий (например, не хуже, чем 90 % оптимума).

1. Оптимальный алгоритм покрытия путями вершин FTG

В данном разделе представлен алгоритм точного решения задачи покрытия вершин FTG путями, имеющий сложность $O(N)$. Рассмотрены особенности структуры графов FTG, алгоритм для построения покрытий и используемые структуры данных. Приведено обоснование оптимальности получаемых покрытий.

1.1. Структура графа FTG

Вершина FTG может иметь не более одной исходящей дуги. Компонента связности такого графа не может содержать более одного цикла [1]. Таким образом, компонента связности FTG может быть либо деревом, ориентированным к корню, либо циклом, "обросшим" такими деревьями (рис. 1).

Следуя терминологии, принятой в англоязычной литературе, граф такого вида можно назвать *ориентированным псевдолесом* (*pseudoforest*), а его компоненты связности — *ориентированными псевдодеревьями* [3]. Графы этого вида относят к категории разреженных, т. е. графов с относительно малым (по отношению к максимально возможному) числом дуг. Линейная сложность приведенного далее алгоритма является следствием того, что число дуг в графах FTG не превышает числа вершин.

1.2. Алгоритм нахождения оптимального покрытия

Заметим, что искомое покрытие однозначно определяется набором содержащихся в нем дуг и в частности тем, какая из входящих дуг для каждой вершины включается в покрытие. Идея предлагаемого алгоритма заключается в том, чтобы вначале построить покрытие, включающее "лучшую" (имеющую максимальный вес) входящую дугу для каждой вершины графа. Такое покрытие может включать простые пути и циклы. На рис. 2 дан пример покрытия, составленного из "лучших" входных дуг, которые изображены стрелками черного цвета. Светлым отмечены дуги, не вошедшие в покрытие, целые числа соответствуют весам дуг.

Следующий шаг заключается в размыкания циклических компонент, если они присутствуют. Для этого в каждом цикле необходимо удалить одну из дуг или заменить ее на другую входящую дугу той же вершины таким образом, чтобы "потеря веса" оказалась минимальной. На рис. 3 дан пример размыкания цикла. Светлыми стрелками показаны дуги графа, не вошедшие в покрытие. Размыкание проводится путем удаления из покрытия одной из дуг цикла и добавления дуги с весом 20, входящей в ту же вершину, что и удаленная дуга.

Заметим, что предварительное разбиение графа на компоненты связности не требуется.

Для того чтобы обеспечить сложность $O(N)$, необходимо позаботиться о правильном выборе структур данных и некоторых алгоритмических решений. Рассмотрим чуть более подробно один из возможных алгоритмов.

Пусть имеется ориентированный взвешенный граф $D = (V, E)$, такой, что степень любой его вершины по выходу не превышает 1. Пусть N — число вершин графа, M — число дуг, $M \leqslant N$. Алгоритм работает со следующими массивами:

`V[N]` — массив структур, описывающих вершины графа;

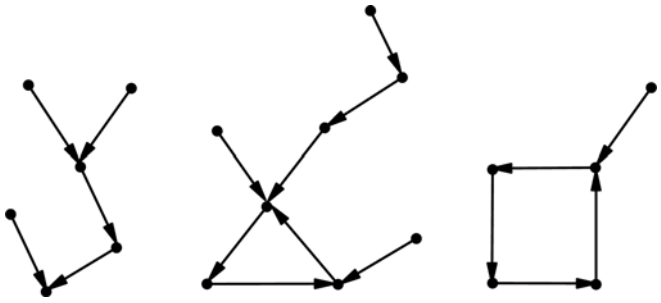


Рис. 1. Примеры компонент связности FTG

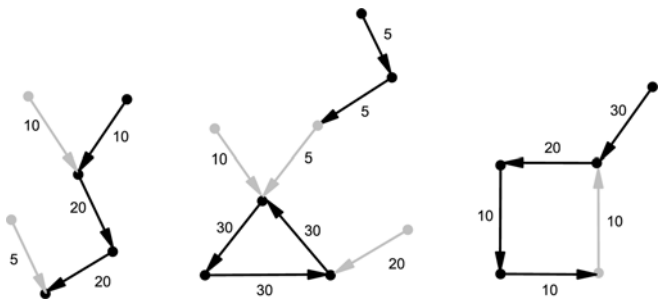


Рис. 2. Покрытие, состоящее из "лучших" входных дуг

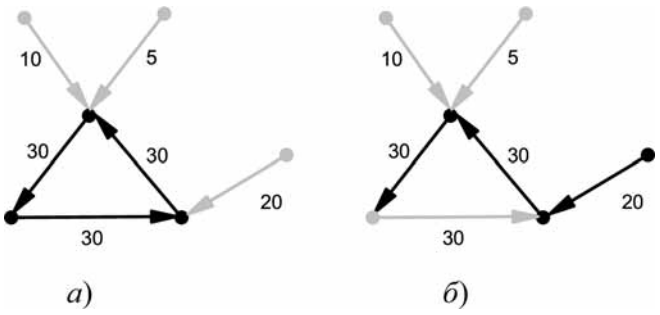


Рис. 3. Пример размыкания цикла:
а — цикл; *б* — результат размыкания цикла

`E[M]` — массив структур, описывающих дуги графа;
`P[N]` — массив структур, описывающих элементы покрытия (простые и циклические пути).

Обозначим через `[1..N]`, `[1..M]` типы данных, включающие целые значения в диапазоне от 1 до `N` или `M` соответственно. Структура, описывающая вершины графа, содержит следующие компоненты:

`out: [1..M]` — выходная дуга вершины (индекс в массиве `E`);

`in:` — множество входных дуг вершины, представленное односвязным списком значений типа `[1..M]`;

`path: [1..N]` — номер элемента покрытия, которому принадлежит вершина (индекс в массиве `P`).

Структура, описывающая дуги, включает следующие компоненты:

`from, to: [1..N]` — номера вершин, соответствующих началу и концу дуги;

`wght:` — целое $\geqslant 0$, вес дуги.

Путь в FTG полностью определяется начальной и конечной вершинами, поскольку из каждой вершины исходит не более одной дуги. Поэтому в структурах, описывающих пути, достаточно иметь следующие компоненты:

`first, last: [1..N]` — номера первой и последней вершин пути;

`t_wght:` — целое $\geqslant 0$, общий вес пути;

`type:` — значение перечислимого типа с элементами `path` (простой путь), `cycle` (цикл), `dead` (путь удален в результате слияния с другим путем).

Исходными данными для алгоритма являются значения `N`, `M` и содержимое массива описателей дуг `E`.

Шаг 1. Первый шаг алгоритма заключается в инициализации массивов `P` и `V`. На этом шаге дуги отсутствуют и строится покрытие для графа (`V`, ∅), т. е. элемент покрытия `P[i]` состоит из одной вершины с номером `i`. Для всех `i` ∈ `[1..N]`:

```
P[i].first = i; P[i].last = i;
P[i].t_wght = 0; P[i].type = path;
V[i].out = 0; V[i].in = null; //пустой список
V[i].path = i;
```

Шаг 2. На втором шаге считывается массив дуг `E`, и в структурах массива `V` заполняются поля `in`, `out`. Для того чтобы обеспечить сложность $O(N)$, процедура добавления дуги `E[i]` (`i` ∈ `[1..M]`) в множество входящих дуг вершины с номером `E[i].to` должно иметь сложность $O(1)$. Например, может применяться вставка в голову списка `V[E[i].to].in`.

Шаг 3. На этом шаге "жадным" образом строится предварительное покрытие вершин графа простыми путями и циклами. Для каждой вершины `V[i]`, если множество ее входящих дуг не пусто, находится входящая дуга с наибольшим весом и добавляется к покрытию `P`. Поиск входящих дуг с максимальным весом для всех вершин имеет суммарную (для всех вершин) сложность $O(N)$, так как общее число дуг не превышает `N`.

Добавление дуги `e` к покрытию может быть реализовано как процедура сложности $O(1)$ следующим образом.

В случае `V[e.from].path` ≠ `V[e.to].path` выполняется объединение двух путей (рис. 4, *а*):

```
p1 = V[e.from].path;
p2 = V[e.to].path;
```

Заметим, что добавляемая дуга должна соединять последнюю точку пути `P[p1]` с первой точкой пути `P[p2]`:

```
assert (P[p1].last == e.from);
assert (P[p2].first == e.to);
```

`//` Добавляем путь `p2` в конец `p1`:

```
last_point = P[p2].last;
P[p1].last = last_point;
```

`//` Изменяем поле `path` у последней точки присоединяемого пути:

```
V[last_point].path = p1;
```

`//` Корректируем вес пути `P[p1]` и удаляем путь `P[p2]`:

```
P[p1].t_wght = P[p1].t_wght + P[p2].t_wght +
               + e.wght;
P[p2].type = dead;
```

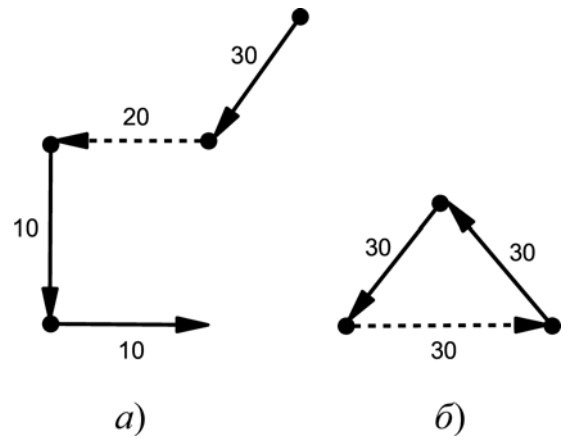


Рис. 4. Добавление дуги к покрытию:
а — объединение двух путей; *б* — зацикливание пути. Штриховыми линиями показаны добавляемые дуги

Заметим, что правильное значение поля path поддерживается только у первой и последней точек каждого пути, иначе не удастся обеспечить сложность $O(1)$ данной процедуры. Этого достаточно, поскольку при добавлении дуг важно содержимое path только в крайних точках путей.

В случае V[e.from].path = V[e.to].path выполняется зацикливание пути (рис. 4, *б*):

```
p = V[e.from].path;
P[p].type = cycle;
```

Добавляемая дуга должна соединять первую и последнюю точки пути P[p]:

```
assert (P[p].last == e.from);
assert (P[p].first == e.to);
P[p].t_wght = P[p].t_wght + e.wght;
P[p].last = P[p].first;
```

Шаг 4. Этот последний шаг заключается в размыкании циклов, если они присутствуют в построенном ранее покрытии. Просматривается массив P, и если P[i].type == cycle, выполняется размыкание цикла p = P[i]. Пусть v — вершина цикла p. Пусть ebest(v) — "лучшая" входящая дуга v (дуга, принадлежащая пути p), а enext(v) — дуга с наибольшим весом среди остальных входящих дуг v, если они есть. Определим функцию d(v) = ebest(v).wght − − enext(v).wght, если v имеет более одной входящей дуги, и d(v) = ebest(v).wght в противном случае. Пусть v0 — вершина с минимальным значением d среди всех вершин пути p. Тогда размыкание p проводится путем замены ebest(v0) на enext(v0) или путем удаления ebest(v0), если это единственная входящая дуга v0. Легко видеть, что в результате выполнения этого шага не могут образоваться новые циклы.

Сложность этого шага $O(N)$, так как общее число просматриваемых дуг не превышает N.

1.3. Обоснование оптимальности получаемых покрытий

Рассмотрим сначала случай произвольного орграфа $G = (V, E)$. Назовем $E' \subset E$ *частичным покрытием путями* (ЧПП), если E' не содержит:

— двух дуг, входящих в одну вершину;

— двух дуг, выходящих из одной вершины;

— ориентированных циклов.

В работе [1] доказано (теорема 3.1), что такое подмножество является объединением простых путей. При этом ЧПП может не покрывать все вершины. Однако любому ЧПП взаимно однозначно соответствует покрытие путями, полученное добавлением путей нулевой длины, покрывающих вершины, не покрытые ЧПП. Вес этого покрытия равен весу ЧПП.

Заметим, что ЧПП можно определить как множество дуг подграфа данного орграфа, не содержащего *запрещенных подграфов*. Запрещенными подграфами в данном случае будут все ориентированные циклы орграфа, все пары дуг с общей конечной вершиной и все пары дуг с общей начальной вершиной.

Введем на E бинарное отношение принадлежности к одному запрещенному подграфу. Это отношение симметрично. Выполнив его рефлексивно-транзитивное замыкание [4] получим отношение эквивалентности и, соответственно, разбиение множества дуг орграфа на классы эквивалентности (далее просто *классы*).

Справедлива следующая лемма: $E' \subset E$ является ЧПП тогда и только тогда, когда пересечение E' с любым из построенных выше классов является ЧПП.

Действительно, доказательство в одну сторону очевидно: если E' является ЧПП, т. е. не содержит запрещенных подграфов, то и все его подмножества являются ЧПП. В другую сторону доказательство также очевидно. Предположим, что пересечения E' со всеми классами являются ЧПП, а E' не является ЧПП. Тогда в нем есть запрещенный подграф, который не может принадлежать какому-то одному классу, так как все пересечения E' с классами — ЧПП, и не может разделяться между разными классами, поскольку, по построению, дуги любого запрещенного подграфа должны принадлежать одному классу. Таким образом, приходим к противоречию.

С учетом изложенного выше ясно, что ЧПП можно строить независимо в каждом классе. Поскольку вес ЧПП равен сумме весов пересечений ЧПП с классами, можно утверждать, что оптимальное ЧПП получается объединением оптимальных ЧПП, построенных в отдельных классах.

Вернемся к FTG и посмотрим, как одна компонента связности распадается на классы. Если эта компонента ациклическая, т. е. представляет собой ориентированное к корню дерево, то ее запрещенные подграфы — это пары дуг с общим концом, и множество дуг распадается на классы, соответствующие конечным вершинам дуг. Следовательно, на такой компоненте связности уже на шаге 3 "жадный" алгоритм построит оптимальное ЧПП. В работе [1] также приводится доказательство того, что на ациклических компонентах "жадный" алгоритм строит оптимальное покрытие (теорема 3.3).

В случае, если компонента связности — ориентированный цикл, обросший деревьями, то в ней выделяется еще один класс: "цикл со щетиной", состоящий из всех дуг, конечная вершина которых принадлежит циклу (включая все дуги цикла). Остальные дуги распадаются на классы по конечным вершинам аналогично ациклическому случаю. Легко видеть, что приведенный алгоритм корректен и оптимален на "циклах со щетиной". Тем самым, алгоритм строит оптимальное покрытие путями вершин FTG.

2. О покрытии путями произвольного орграфа

Как можно видеть, поиск оптимального покрытия FTG оказывается простой задачей. В практике компиляции чаще применяется оптимизация переходов,

учитывающая все виды передач управления, включая условные. Такая оптимизация сводится к NP-полной задаче поиска оптимального покрытия путями произвольного взвешенного орграфа (CFG или производного от него, как показано в работе [5]). Под оптимальным покрытием подразумевается покрытие, имеющее максимальный суммарный вес.

Частным случаем задачи об оптимальном покрытии, соответствующем единичным весам всех дуг, является известная задача покрытия орграфа минимальным числом путей. NP-полнота этой задачи следует из того факта, что ее частным случаем является задача о существовании гамильтонова пути в орграфе (см. [6, п. TG39]).

Полезно рассмотреть задачу об оптимальном покрытии как задачу о пересечении *матроидов* (см. [7, п. 12.4]). В работе [7, п. 12.6], показано, что любой гамильтонов путь в орграфе принадлежит как множество ребер трем матроидам на множестве ребер орграфа. Два из этих матроидов соответствуют подграфам исходного графа со степенями не более 1 по выходу и входу соответственно. Легко видеть, что это *матроиды разбиения* по началам и концам дуг орграфа.

Остается условие, что множество ребер не содержит циклов. К сожалению, нельзя просто взять все ациклические подграфы орграфа — они не образуют матроида. Но матроид образуют ациклические подграфы (леса) неориентированного графа (*графический матроид*). Учитывая, что "неправильно ориентированный" цикл орграфа обязательно содержит вершину, в которую входят сразу две дуги, и еще вершину, из которой исходят две дуги, то при условии, что мы рассматриваем пересечение хотя бы с одним из матроидов разбиения описанных выше, мы имеем право подменить множество ациклических подграфов на множество подграфов, ациклических при утрате ориентации.

Понятно, что не только гамильтоновы пути, но и любые линеаризации, рассматриваемые как множество ребер, также должны принадлежать этим трем матроидам.

Задача о максимальном пересечении трех матроидов — известная NP-полная задача (см. [6, п. MP11]). Однако задача о максимальном пересечении двух матроидов решается за полиномиальное время [7], в том числе, во взвешенном варианте.

Если исключить из рассмотрения один из трех матроидов, то получаем задачи о покрытии орграфа деревьями, ориентированными к корню, деревьями, ориентированными от корня, и задачу покрытия орграфа путями и циклами. Все эти задачи решают за полиномиальное время как задачи о пересечении двух матроидов. Наиболее интересна из них последняя задача — о покрытии путями и циклами.

Поскольку нами был исключен графический матроид, остались два матроида разбиения. Задача об их максимальном пересечении — хорошо изученная задача о максимальном паросочетании в двудольном графе. Более конкретно, для данного орграфа строим

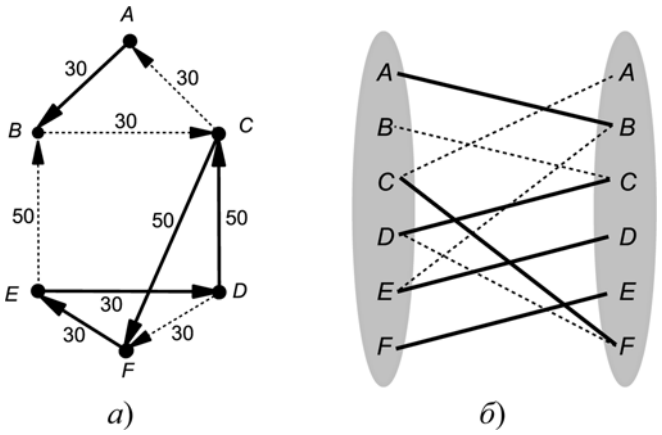


Рис. 5. Покрытие орграфа простыми путями и циклами (*а*), полученное как решение задачи о максимальном паросочетании в двудольном графе (*б*):
сплошными линиями показаны дуги, вошедшие в покрытие; пунктиром — остальные дуги исходного графа

неориентированный двудольный граф. Вершины одной доли этого графа соответствуют вершинам исходного орграфа с ненулевой степенью по выходу, а вершины второй доли — вершинам орграфа с ненулевой степенью по входу. Затем вершины разных долей соединяются при условии наличия дуги в исходном орграфе (рис. 5). Покрытиям путями и циклами орграфа взаимно однозначно соответствуют паросочетания в построенном двудольном графе. Это описано в классической работе [8].

Таким образом, задача о покрытии путями и циклами решается во взвешенном случае за время $O(n^2 \log n + nm)$ [9], где n — число вершин, а m — число дуг графа. Отметим, что данная асимптотика соответствует наихудшему случаю; практически, задача о паросочетании часто решается быстрее. Отсюда в первую очередь следует, что для ациклических орграфов задача покрытия путями полиноминальна. Действительно, если искать покрытие путями и циклами, то будет найдено покрытие одними путями.

Кроме того, если орграф "почти ациклический", т. е. содержит только один цикл, то оптимальное покрытие путями также будет найдено за полиномиальное время. В этом случае можно по очереди исключать по одной дуге из цикла, решать задачу для полученного ациклического орграфа, а потом выбрать лучшее покрытие. Здесь можно отметить, что имеет смысл сначала найти покрытие путями и циклами для исходного орграфа, и только если его единственный цикл вошел в покрытие, начинать перебор с исключением ребер. При этом не обязательно каждый раз решать задачу о паросочетании с самого начала: можно стартовать с паросочетания для исходного орграфа, исключая одно ребро из двудольного графа. Этот способ работает, так как задача нахождения оптимального паросочетания решается в некотором смысле локальным поиском.

Наконец, развивая последние соображения можно организовать поиск оптимальной линеаризации для

произвольного орграфа методом ветвей и границ. Поясним, что имеется в виду, описав один (не эффективный, но простой) вариант алгоритма перебора.

Сначала, как было описано выше, строится оптимальное покрытие путями и циклами. Если в покрытии нет циклов, то оптимальное покрытие уже получено.

Если в покрытии есть циклы, то нельзя, как было сделано для FTG, просто разомкнуть эти циклы, поскольку для произвольного графа добавление дуг может замкнуть новые циклы, а размыкание циклов без добавления дуг скорее всего не даст оптимального покрытия путями. По этой причине выбираем один цикл, и для каждой дуги из этого цикла рекурсивно запускаем алгоритм для орграфа, из которого удалена выбранная дуга. Лучшее из полученных покрытий будет оптимальным покрытием путями. Завершение алгоритма гарантируется тем обстоятельством, что глубина рекурсии ограничена числом дуг в исходном орграфе.

Можно организовать ветвление и другими способами, но важно то, что на каждом шаге найденное покрытие путями и циклами автоматически дает хорошую оценку веса сверху, а любое корректное, но не обязательно оптимальное размыкание циклических компонент является покрытием путями, и его вес может служить оценкой снизу. Если оценка сверху меньше или равна наибольшей полученной ранее оценке снизу (веса лучшего построенного покрытия), ветвь обрывается. Можно также искать субоптимальные решения, требуя чтобы оценка сверху была не просто больше достигнутого веса, а например, больше на 10 %.

Например, из орграфа на рис. 5 достаточно удалить одну дугу *ED*, разомкнув тем самым цикл *CFED*, чтобы получить оптимальное покрытие из двух путей, а именно — [*DCFEB*] весом 180 и [*A*] веса 0 (путь нулевой длины).

Можно рассчитывать, что подобный алгоритм будет эффективнее простого метода ветвей и границ, предложенного в работе [5], особенно в случае не очень большого числа циклов в орграфе. Тем не менее, алгоритм остается экспоненциальным по времени в худшем случае, поэтому крайне полезна предварительная декомпозиция орграфа в соответствии с изложенным в разд. 1.3. По-видимому, этот подход можно также разными способами сочетать с методом декомпозиции по *гамакам*, предложенным в работе [10].

Авторы выражают благодарность Волконскому Владимиру Юрьевичу и Рыбакову Алексею Анатольевичу за полезные замечания по содержанию статьи и предоставленный текст диссертации [1].

Список литературы

1. **Рыбаков А. А.** Методы и алгоритмы оптимизации переходов в компиляторе базового уровня системы двоичной трансляции для архитектуры "ЭЛЬБРУС": Автореф. дис. ... канд. физ.-мат. наук. М., 2013.

2. **Ramanath M. V. S., Solomon M.** Jump Minimization in Linear Time // ACM Transactions on Programming Languages and Systems. 1984. Vol. 6, N 4. P. 527—545.

3. **Academic** Dictionaries and Encyclopedias. URL: http://en.academic.ru/dic.nsf/enwiki/7492134.

4. **Общая** алгебра. Т. 1. / Под ред. Л. А. Скорнякова. М.: Наука, 1990. 592 с.

5. **Медведев О. В.** Линеаризация графа потока управления с учетом результатов профилирования // Системное программирование. 2006. Т. 2, № 1. С. 25—47.

6. **Гэри М., Джонсон Д.** Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982. 416 с.

7. **Пападимитриу Х., Стайглиц К.** Комбинаторная оптимизация. Алгоритмы и сложность. М.: Мир, 1985. 510 с.

8. **Boesch F. T., Gimpel J. F.** Covering the points of a digraph with point-disjoint paths and its application to code optimization // Journal of the ACM 24. 1977. N 2. P. 192—198.

9. **Fredman M. L., Tarjan R. E.** Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms // Journal of the ACM. 1987. Vol. 34, Is. 3. P. 596—615.

10. **Medvedev O.** Optimal Basic Block Reordering via Hammock Decomposition. URL: http://syrcose.ispras.ru/2007/files/2007_01_paper.pdf.

ИНФОРМАЦИЯ

3—4 октября 2014 г. в Москве пройдет 4-я Международная конференция в области управления ИТ-проектами

"Software Project Management Conference"

Конференция состоится при поддержке Гильдии менеджеров программных проектов SPM Guild.

В конференции примут участие руководители ИТ-компаний, директора и руководители ИТ-проектов из различных городов и стран.

Тематика докладов разбита на четыре группы (принцип 4П): персонал, процессы, проекты, продукты.

Рабочие языки конференции русский и английский.

Подробности на сайте конференции: **http://spmconf.ru/ru/index**

УДК 004.414.23

А. О. Соколов, аспирант, инженер-конструктор, e-mail: wedmeed@mail.ru,
Саратовский государственный технический университет им. Гагарина Ю. А.,
ОАО "КБ Электроприбор", г. Саратов

Модель программного обеспечения систем реального времени

Представлена модель программного обеспечения, построенная с использованием понятий из трех смежных областей науки, касающихся систем реального времени — теории вычислений, теории планирования, теории операционных систем. Модель позволяет описать специфические особенности различных типов встроенных систем через базовые понятия. В качестве примера приведено описание цифрового регулятора, входящего в состав системы автоматического управления.

Ключевые слова: *программа, операционная система, модель, планирование, реальное время, процессор, ресурс*

A. O. Sokolov

Model of the Real-Time Systems Software

The paper presents a generic model of real-time software. It includes concepts from the theory of computation, scheduling theory and operating systems theory. The model allows describing specific features of different systems. The common terminology and structure is saved. The paper is divided into three parts. The first part describes a base model suiting all real-time systems. The second part defined a set of features inherent in most real-time systems. At the last part of the paper an example of the digital controller software model is present. The model can be used in situations when the specific features of the real-time system cannot be simplified or omitted. Due to this the modernization of scheduling models is provided possible (treating scheduler as a task, interrupt handlers considered during feasibility test, etc.).

Keywords: *software, operating system, model, scheduling, real-time, computing, processor, resource, algorithm, embedded*

Введение

Разработка программного обеспечения (ПО) для систем реального времени (РВ) является непростой задачей. Большое число ограничений в виде крайних сроков, ограничений предшествования, а также большое разнообразие аппаратных платформ с различными архитектурами заставляют искать нетривиальные решения для каждой конкретной разработки.

Одним из наиболее распространенных инструментов являются операционные системы (ОС) реального времени. Для описания процессов, происходящих в ОС РВ создано большое число моделей. Для примера можно обратиться к работам [1—6]. Однако большинство этих моделей предназначены для решения специфических задач. Например модели, описанные в работе [2], требуют перестроения принципов функционирования при описании некоторых особенностей, таких как добавление серверов и подсистем планирования, слоев управления, упрощений по учету планировщика и т. д. В большинстве работ применяемая терминология разобщена или не связана с терминологией других направлений науки о вычислениях.

Перечисленные трудности связаны с большим технологическим скачком, произошедшим в последнее время: за последние 30 лет появилось множество объектов, работающих в реальном времени и имеющих разную специфику; заметно усложнились модели ПО — появились различные апериодические задачи [2, 5], нестандартные ограничения и параметры задач [3, 4]. Сильное влияние на терминологию оказали неточности перевода. К тому же, рассматриваемая задача исследуется с разных точек зрения в разных областях знаний.

В данной работе приведено решение задачи построения модели вычислительной системы РВ, исполь-

зующей унифицированную терминологию и обладающей достаточной гибкостью. Основные определения соответствуют терминам из работ [1, 2, 7—9]. Для всех отличающихся от стандартных понятий (*ресурс*, *событие*, *задача*, *контекст*) приведены уточняющие формулировки. Графические обозначения выполнены максимально соответствующими работе [10]. Работа разбита на три части, каждая из которых является не модификацией, а уточнением предыдущей. Сначала описывается модель, применимая для всех вычислительных систем РВ. Затем в нее добавляется специфика, общая для большинства проектов РВ (приоритеты и запросы). В заключение показывается способ использования на конкретном специфическом примере.

В качестве основы для модели использованы работы [2—5]. Основным ее отличием от исходных является более глубокая связь с идеологией и терминологией смежных областей науки, таких как теория алгоритмов [7, 11] и построение операционных систем [9, 12]. Понятия основной части не имеют конкретного алгоритма работы и являются абстракциями, показывающими поведение системы. Это позволяет сохранить общность основных принципов, заложенных в модель, и независимость от используемой аппаратной платформы при разработке ПО. Уточнение абстракций в виде добавления ограничений РВ позволяет описать многие особенности, для которых ранее приходилось принимать упрощения: различие аппаратных и программных приоритетов, циклическое выполнение задач, наличие разных типов задач и т. д.

Общее описание модели программного обеспечения

При построении модели будем рассматривать типовую одноядерную вычислительную систему. Это позволит сделать описание системы максимально приближенным к моделям, рассматриваемым в работах [7, 11]. В более общем случае любая многопроцессорная или многоядерная система может быть сведена к набору однопроцессорных систем с общими ресурсами, что будет приведено далее.

Основой модели является вычислительная система, имеющая в своем составе *процессор* и множество *запоминающих устройств*, которые содержат *данные*. При работе процессор непрерывно совершает *действия* над данными в соответствии с *инструкциями*, которые получает в *закодированном* виде из самих данных. Последовательность инструкций (*код*) и используемые при этом данные составляют *программу*. Типичным примером основной части модели служит классическая машина Тьюринга, что позволяет легко использовать задействованные здесь определения. Далее, при расширении модели, приведенное сходство теряется (увеличивается число лент и машин, машины перестают быть привязанными к одной конкретной ленте).

В вычислительной системе могут существовать *периферийные устройства*. Они имеют доступ к определенным данным и могут совершать действия над ними под влиянием внешних к системе факторов. Примером периферийных устройств могут быть сопроцессоры, устройства прямого доступа в память (*DMA — direct memory access*), модули последовательной передачи данных, другие процессоры или ядра. Процессор не имеет сведений об их числе и принципах работы.

Любое изменение данных будем называть *событием*. Некоторые события могут влиять на поведение процессора, в том числе и изменять выполняемую им программу. События могут быть сгенерированы как процессором при выполнении инструкции (программно), так и периферийным устройством (аппаратно).

Процессор является устройством, работающим в однородном дискретном времени $t \in N$, N — множество натуральных чисел. Таким образом, с точки зрения выполнения программ, все периферийные устройства и события происходят в соответствующие дискретные моменты. Это происходит потому, что интерфейсом между внешней средой и программой является именно процессор — влияние любого события на программу скажется только при выполнении очередной инструкции. Периодом между двумя моментами времени будет считаться время выполнения одной инструкции.

Все данные в вычислительной системе считаются структурированными по ресурсам. *Ресурсом* в данной работе будем называть запоминающее устройство или любую его часть, данные в котором обладают единым набором свойств (например, используются в ограниченном наборе программ, доступны периферийному устройству и т. д.). Ресурсы не должны пересекаться и зависеть от данных в других ресурсах. Если такая зависимость фактически существует, то она должна рассматриваться как наличие некоторого периферийного устройства, изменяющего эти ресурсы. Все ресурсы, к которым имеют доступ периферийные устройства, будем называть *внешними*.

Для случая многоядерных систем или при наличии сопроцессоров, взаимодействие между модулями осуществляется только через внешние ресурсы (память, регистры периферии, регистры прерываний и т. д.). Такая абстракция легко укладывается в существующие системы. Для ограничения числа внешних ресурсов, в целях увеличения надежности, на практике возможно ограничивать область доступа периферийных устройств путем перестроения карты памяти.

Все программы в вычислительной системе считаются структурированными по задачам. *Задачей* в данной работе будем называть целостную и автономную программу, обладающую собственным контекстом. *Контекст задачи* — набор специальных данных, описывающих состояние задачи. *Контекст процессора* — набор специальных ресурсов (не внешних), влияющих на поведение процессора (данные в этих ресурсах задействуются при выполнении большей части инструкций). При загрузке в контекст процессора контекста некоторой задачи, процессор переключается на

ее выполнение. Таким образом, задача жестко привязана к процессору (любые другие действия в системе параллельно с выполнением задачи рассматриваются как работа периферийных устройств). Примером данных в контексте процессора может служить совокупность указателя счетчика команд (*PC — program counter*), указателя стека (*SC — stack counter*), содержимого стека, регистров общего назначения (*CPR — common-purpose register*), регистров специального назначения (*SPR — special-purpose register*) и т. д. [13—15]. Состав контекста в конкретном случае определяется особенностями процессора и самих задач.

Целостность задачи заключается в том, что она может корректно выполняться независимо от наличия каких-либо других задач. Автономность задачи означает, что любое взаимодействие с другой задачей должно быть реализовано так, чтобы учитывать:

— возможность отсутствия задачи, с которой предполагается взаимодействие;

— возможность некорректной работы задачи, с которой предполагается взаимодействие;

— возможность недоступности задачи, с которой предполагается взаимодействие;

— вычислительное время и количество ресурсов, необходимые для взаимодействия (с другой задачей недетерминированы);

— все ресурсы, являющиеся внешними или используемые другими задачами (могут быть изменены в любой момент неизвестным для задачи способом).

Учет таких особенностей позволяет выполнять задачи в произвольном порядке, в параллельном режиме и в псевдопараллельном режиме.

Визуальное представление вычислительной системы приведено на рис. 1.

Множество всех ресурсов обозначим $\{\psi_1, ..., \psi_k, ..., \psi_m\}$, где m — общее число ресурсов; k — порядковый номер произвольно взятого ресурса; $1 \leqslant k \leqslant m$. Для определения ресурсов, используемых процессором в момент времени t, может применяться следующая функция:

$$\psi(t) = \Psi_t, \tag{1}$$

где Ψ_t — некоторый произвольный набор ресурсов, зависящий от выполняемых процессором в данный момент инструкций, $\Psi_t \subseteq \{\psi_1, \psi_2, ..., \psi_m\}$. Изменение ресурса ψ_k в момент времени t определяется функцией

$$\Delta\psi_k(t) = \Delta\psi_{k,\,t}, \tag{2}$$

где $\Delta\psi_{k,\,t}$ — произошедшее изменение, такое, что $\psi_k \notin \psi(t) \Leftrightarrow \Delta\psi_k(t) = \varnothing$.

Множество всех задач в системе описывается набором $\{\tau_0, ..., \tau_i, ..., \tau_n\}$, где $n + 1$ — общее число задач; i — порядковый номер произвольно взятой задачи; $0 \leqslant i \leqslant n$. В один момент времени процессор может выполнять только одну задачу (только ее контекст загружен в контекст процессора, процессор выполняет инструкции из кода, принадлежащего этой задаче). Выполняемую задачу будем называть в данной работе активной. Для определения выполняемой в момент времени t задачи может использоваться следующая функция:

$$\tau(t) = \tau_i. \tag{3}$$

События могут происходить в произвольном порядке и влиять на произвольное число задач. Таким образом, для описания состояния задачи недостаточно только двух основных режимов (активный и неактивный). С точки зрения выполнения программ каждая задача τ_i может находиться в одном из следующих состояний:

- *wait* — состояние ожидания, задача неактивна до наступления определенного события;

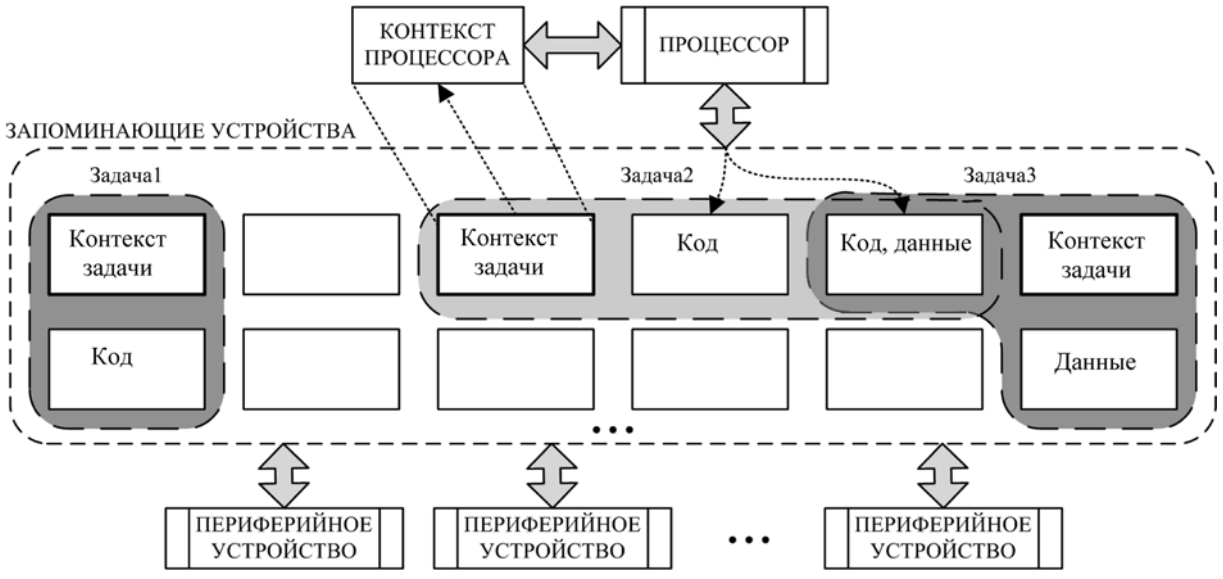


Рис. 1. Пример представления вычислительной системы в терминах модели (изображено выполнение процессором задачи 2)

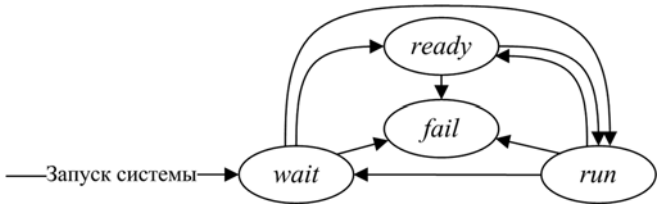


Рис. 2. Граф состояний задачи в вычислительной системе

- *run* — состояние выполнения, задача активна (только одна задача может единовременно быть в этом состоянии);
- *ready* — состояние готовности, задача неактивна до освобождения процессора;
- *fail* — ошибочное состояние, задача не может быть активна вообще, контекст задачи не может быть загружен в контекст процессора.

Переход задачи между состояниями описывается графом (рис. 2). Действия, выполняемые для смены состояния задачи, включающие в себя перенастройку процессора и смену контекста, называют диспетчеризацией. Выполняться диспетчеризация может как аппаратно, так и программно. В первом случае процессор должен иметь механизм (например, прерывания), который при наступлении определенного события переносит в контекст процессора контекст связанной с этим событием задачи. Программная диспетчеризация предполагает, что некоторая служебная задача (задача-диспетчер, *context switcher*) самостоятельно выполняет операцию смены контекста, что и является событием изменения состояний.

В любой момент времени состояние задачи τ_i можно однозначно определить соответствующей функцией:

$$\hat{\tau}_i(t) = state, \qquad (4)$$

где *state* — состояние задачи; $state \in \{wait, run, ready, fail\}$.

Формулы (2) и (4) могут быть использованы для полного описания состояния системы в рамках данной модели в любой момент времени. Более детальное раскрытие модели зависит от особенностей вычислительной системы и заключается в определении структуры и коэффициентов перечисленных функций. При этом основные принципы функционирования не меняются.

Отсутствие состояния задачи *block* или подобных объясняется необходимостью интеграции с моделями, описанными в работах [2—6]. В соответствии, например, с работой [12], *block* подразумевает необходимость явного вызова задачи процедурой *fork* или *copy*. Для систем РВ необходимо предусмотреть создание задачи по внешнему событию. Поэтому в качестве *block* используется *wait*. Для систем общего назначения эти состояния являются идентичными (так как нет внешних инициирующих событий). Для систем РВ состояние *wait* отличается наличием некоторого механизма (например, системы прерываний), готового в любой момент инициировать задачу. Этот механизм активен даже при условии, что задача ни разу не запускалась.

Особенности модели в системах реального времени

Дальнейшее расширение модели уже связано с углублением в специфику систем РВ. Конкретизация является необходимой для связи модели с работами по планированию [2—6], но она не исключает применение модели в таком виде и для других областей использования ПО.

Фоновая задача. С точки зрения выполнения программ может произойти ситуация, когда ни одна задача не может находиться в состоянии *run*.

В некоторых современных микропроцессорных устройствах реализован режим простоя процессора [15—17]. Этот режим позволяет приостанавливать выполнение процессором инструкций до наступления определенного события, уменьшая его энергопотребление. Данный режим является специфическим и во многих устройствах реализован по-разному или может отсутствовать [18].

Другим решением, подходящим к модели, является организация специальной задачи, выполняемой в фоновом режиме. К тому же, эта задача может быть совмещена с задачей, реализующей начальную инициализацию вычислительной системы.

Выполнение процессором программ начинается со служебной задачи τ_0. Служебная задача настраивает вычислительную систему, т. е. инициализирует наборы ресурсов и задач, запускает механизмы отслеживания событий процессором. После проведения настройки служебная программа входит в пустой бесконечный цикл. Задача τ_0 не может находиться в состоянии *wait* или *fail*, что обеспечивает безостановочную работу процессора. Поведение остальных задач зависит от происходящих далее событий.

Если в вычислительной системе предусмотрен режим простоя процессора, то он может быть включен в бесконечный цикл задачи τ_0.

Запросы задач. Другой особенностью систем РВ является частое циклическое выполнение одних и тех же функций. При этом каждое выполнение не зависит напрямую (через контекстные данные) от времени или результатов предыдущего выполнения. Вся информация передается через глобальные переменные и рассматривается как поступившее извне начальное условие.

В предлагаемом расширении модели будем считать, что все задачи могут выполняться процессором неограниченное число раз (т. е. задачи не завершаются, а переходят в состояние *wait*). *Запросом* $\tau_{i,j}$ с порядковым номером j, $1 \leqslant j \leqslant \infty$, назовем каждое j-е выполнение задачи τ_i, инициированное отдельным событием. *Событием инициации* далее будем называть событие, при котором эта задача, имеющая состояние *wait*, изменяет его на *run* или *ready*.

Для завершения выполнения запроса в коде задачи должна присутствовать подпрограмма, генерирующая событие завершения. *Событием завершения* будем называть событие, переводящее задачу из состояния *run* в состояние *wait* (т. е. завершающее запрос задачи, но не задачу). Событие завершения может быть инициировано только самой задачей.

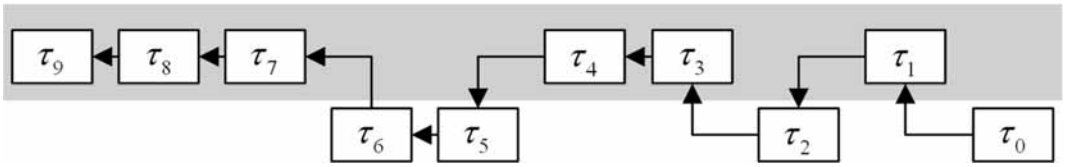


Рис. 3. Иерархия задач по приоритетам для случая $P_0 < P_1 < P_2 < P_3 < P_4 < P_5 < P_6 < P_7 < P_8 < P_9$: на сером фоне задачи, приоритет которых обрабатывается аппаратно; стрелки указывают направление увеличения приоритета

Если процессор приостанавливает выполнение запущенного запроса, то задача переходит в состояние *ready*. Такая ситуация, называемая вытеснением (*preemption*), может произойти, например, когда процессор начинает выполнять другую задачу. Если процессор откладывает выполнение вновь запускающегося запроса некоторой задачи, то она также переходит в состояние *ready* (сразу из состояния *wait*). Время нахождения задачи в состоянии *ready* будем называть временем *простоя*.

Контекст задачи после завершения запроса может как сохраняться до следующей инициации, так и сбрасываться (принимать состояние по умолчанию) событием завершения. Правила обработки контекста задачи определяются алгоритмом планирования задач и особенностями самой задачи.

Событие ошибки для задачи генерируется при обнаружении некорректного поведения задачи. Независимо от состояния задачи, она переходит в *fail*. После этого дальнейшие события задачей игнорируются, а запросы не инициируются.

Приоритеты задач. Для разрешения ситуаций, когда несколько задач могут быть выполнены процессором, применяют различные способы планирования. Обзор основных способов рассмотрен в работах [4, 10]. Для осуществления большинства способов планирования требуется наличие специального параметра задачи — приоритета.

Каждая задача τ_i характеризуется приоритетом P_i (здесь и далее индекс параметра задачи будет обозначать номер характеризуемой задачи). Во всех ситуациях, когда происходит вытеснение или отложенное выполнение, из задач, готовых к выполнению (находящихся в состоянии *ready*), выбирается на выполнение (переводится в состояние *run*) та задача, чей приоритет выше:

$$\forall(i{:}P_i = \max(\{P_k{:}1 \leqslant k \leqslant n,\ \hat{\tau}_k(t) \neq wait,\ \hat{\tau}_k(t) \neq fail\})),\ \hat{\tau}_i(t) = run. \quad (5)$$

Если несколько готовых к выполнению задач имеют одинаковый приоритет, то выполняемая задача определяется произвольным образом. Такое решение позволяет расширить применимость модели, но является неэффективным. Например, при наличии нескольких задач с одинаковым приоритетом, время до окончания выполнения каждой является суммой времени выполнения всех задач (по наихудшему случаю). В той же ситуации, но при разных приоритетах, этот же параметр определяется как сумма времени выполнения только рассматриваемой задачи и более высокоприоритетных по отношению к ней. При рассмотрении конкретного способа планирования, для улучшения результатов могут быть приняты различные упрощения [1—6, 10], например, отсутствие задач с одинаковыми приоритетами или более однозначные алгоритмы выбора задачи для выполнения.

Для работы механизма приоритетов необходимо чтобы событие инициации задачи могло вытеснить все более низкоприоритетные задачи. Если это поддерживается на аппаратном уровне, например, механизмом прерываний, то никаких дополнительных мер не требуется. Если диспетчеризация выполняется программным способом, то приоритет задачи (далее будем называть его программным) не может быть выше приоритета задачи диспетчера (диспетчер является отдельной задачей и также имеет приоритет, так как в предлагаемой модели все программы структурированы по задачам, для исключения диспетчера из набора задач необходимо принять упрощения, такие как в работах [2, 3, 5, 6]). Для задач с программным приоритетом событием инициации может быть только событие завершения задачи диспетчера. Подобное осложнение модели необходимо для охвата некоторых механизмов диспетчеризации, используемых в ОС.

Если задача должна реагировать на некоторое другое событие, то это событие сначала должно инициировать диспетчер (аппаратным способом), который затем инициирует нужную программу. Пример иерархии задач по приоритетам изображен на рис. 3. Если в такой системе должен быть реализован только один диспетчер, то он должен иметь приоритет выше или равный P_7.

Задача τ_0 имеет самый низкий приоритет, иначе, так как τ_0 никогда не сгенерирует событие завершения, задачи с приоритетом ниже P_0 никогда не смогут быть запущены на выполнение.

Параметры задач и запросов в модели

Для описания выполнения задач процессором для каждого запроса $\tau_{i,j}$ может использоваться следующий набор параметров (здесь и далее индексы параметра запроса будут обозначать номер задачи, которой принадлежит запрос, и номер самого запроса):

- $a_{i,j}$ — момент инициации запроса, возникновение события инициации;
- $s_{i,j}$ — момент запуска запроса, первый переход задачи в состояние *run* после возникновения события инициации;
- $c_{i,j}$ — время выполнения запроса (без учета времени вытеснения);
- $f_{i,j}$ — момент завершения запроса, генерация события завершения и переход задачи в состояние *wait*;

- $r_{i,j}$ — время отклика запроса (время выполнения с учетом времени вытеснения);
- $d_{i,j}$ — крайний срок завершения запроса.

$$\hat{\tau}_i(a_{i,j}) \neq wait,\ \hat{\tau}_i(a_{i,j}) \neq fail,\ \hat{\tau}_i(a_{i,j} - 1) = wait. \quad (6)$$

$$\hat{\tau}_i(s_{i,j}) = run,\ \forall(t{:}f_{i,j-1} \leqslant t < s_{i,j}),\ \hat{\tau}_i(t) \neq run. \quad (7)$$

$$c_{i,j} = f_{i,j} - s_{i,j} - \sum_{\forall\{a,b\}} b - a, \quad (8)$$

где пары $\{a, b\}$ определяют такие моменты времени между $s_{i,j}$ и $f_{i,j}$, в которые задача не выполнялась, $\{a, b\}$: $s_{i,j} < a,\ b < f_{i,j},\ \forall(t{:}a < t < b,\ \hat{\tau}_i(t) \neq run)$.

$$\hat{\tau}_i(f_{i,j}) = wait,\ \hat{\tau}_i(f_{i,j} - 1) = run. \quad (9)$$

$$r_{i,j} = f_{i,j} - a_{i,j}. \quad (10)$$

$$\forall(\tau_i{:}\ \hat{\tau}_i(d_{i,j}) = run),\ \hat{\tau}_i(d_{i,j} + 1) = fail. \quad (11)$$

$$a_{i,j} \leqslant s_{i,j} < f_{i,j} \leqslant d_{i,j}. \quad (12)$$

Условия (6), (9) и (12) предполагают, что для переключения на следующий запрос задача хотя бы на время выполнения одной инструкции переходит в состояние *wait* (для выполнения диспетчерирования), каждый запрос содержит минимум одну инструкцию для выполнения (а именно инструкцию завершения запроса).

Формулы (7)—(9) описывают общие ограничения для $s_{i,j}$, $f_{i,j}$ и $c_{i,j}$. В конкретной ситуации эти ограничения могут быть увеличены, но не должны противоречить приведенным.

Формула (10) определяет время отклика запроса задачи. Это время не должно быть бóльшим, чем $d_{i,j} - a_{i,j}$. В противном случае будет нарушен крайний срок и может произойти ситуация, описанная в формуле (11).

Перечисленные выше параметры являются коэффициентами функции (4). В силу наличия случайных факторов при работе системы, значение этих коэффициентов и структура (4) не могут быть определены заранее и уточняются в процессе работы. Однако для анализа системы до выполнения могут быть использованы параметры задач, которые являются ограничениями для параметров запросов:

- T_i — период инициации запросов задачи;
- Φ_i — фазовый сдвиг — время инициации первого запроса задачи;
- C_i — максимальное время выполнения запроса задачи;
- D_i — относительный крайний срок задачи.

По характеру моментов инициации запросов, задачи могут делиться на периодические, спорадические и регулярные [2, 4]. Для всех периодических задач период инициации T_i и фазовый сдвиг Φ_i определяют четкие моменты инициации запросов:

$$\forall j,\ \begin{cases} T_i = a_{i,j+1} - a_{i,j} \\ \Phi_i = a_{i,1}. \end{cases} \quad (13)$$

Для всех спорадических задач T_i определяет минимальное время между инициациями запросов, а Φ_i — минимальное время до инициации первого запроса:

$$\forall j,\ \begin{cases} T_i \leqslant a_{i,j+1} - a_{i,j} \\ \Phi_i \leqslant a_{i,1}. \end{cases} \quad (14)$$

Для регулярных задач T_i и Φ_i определяют границы времени, между которыми может быть инициирован запрос:

$$\forall j,\ T_i \cdot (j-1) + \Phi_i \leqslant a_{i,j} \leqslant T_i \cdot j + \Phi_i. \quad (15)$$

Формулы (13)—(15) позволяют описать характер выполнения любой задачи жесткого (*hard*) реального времени. Для задач мягкого (*soft*) и строгого (*firm*) времени ограничения будут отличаться.

В качестве общего показателя времени выполнения используется C_i (*WCET — worst case execution time* — время выполнения в худшем случае):

$$C_i = \max\left(\bigcup_{j=1}^{\infty} c_{i,j}\right). \quad (16)$$

Крайний срок выполнения запроса (*deadline* — дедлайн) характеризует необходимость обеспечить реакцию на событие, инициировавшее запрос, за ограниченное время. Нарушение крайнего срока переводит задачу в состояние *fail* (наступление времени крайнего срока запроса до его завершения является событием ошибки). Крайний срок для каждого отдельного запроса определяется по относительному крайнему сроку задачи:

$$\forall j,\ d_{i,j} = D_i + a_{i,j}. \quad (17)$$

$$D_i \geqslant C_i. \quad (18)$$

На рис. 4 изображены графики выполнения задач при различных условиях. Примеры, представленные на графиках, подобраны так, чтобы наглядно показать используемые в модели временные параметры запросов и задач.

Время выполнения задачи (состояние *run*) отмечено на временной оси серым прямоугольником со сплошной границей. Момент инициации запроса задачи обозначен пустой окружностью на временной оси. Время простоя (состояние *ready*) показано сплошной линией, соединяющей момент инициации и отрезки времени, в которые задача выполняется. Время, оставшееся до крайнего срока (задача в состоянии *wait*) обозначено пунктиром, а сам момент крайнего срока — стрелкой, указывающей вниз. Время, в которое задача могла потенциально работать, обозначено прямоугольником с штриховой границей. Во все необозначенное на временной оси время задача находится в состоянии *wait*.

Параметризация (1) и (2) является нетривиальной задачей, так как сильно зависит от используемого планирования ресурсов. Наиболее часто в системах РВ встречается статическое распределение ресурсов. В этом случае для каждой задачи τ_i определяется набор ресурсов $\Psi_i \subseteq \{\psi_1, \psi_2, ..., \psi_m\}$. В один момент времени задача может задействовать любое число ресур-

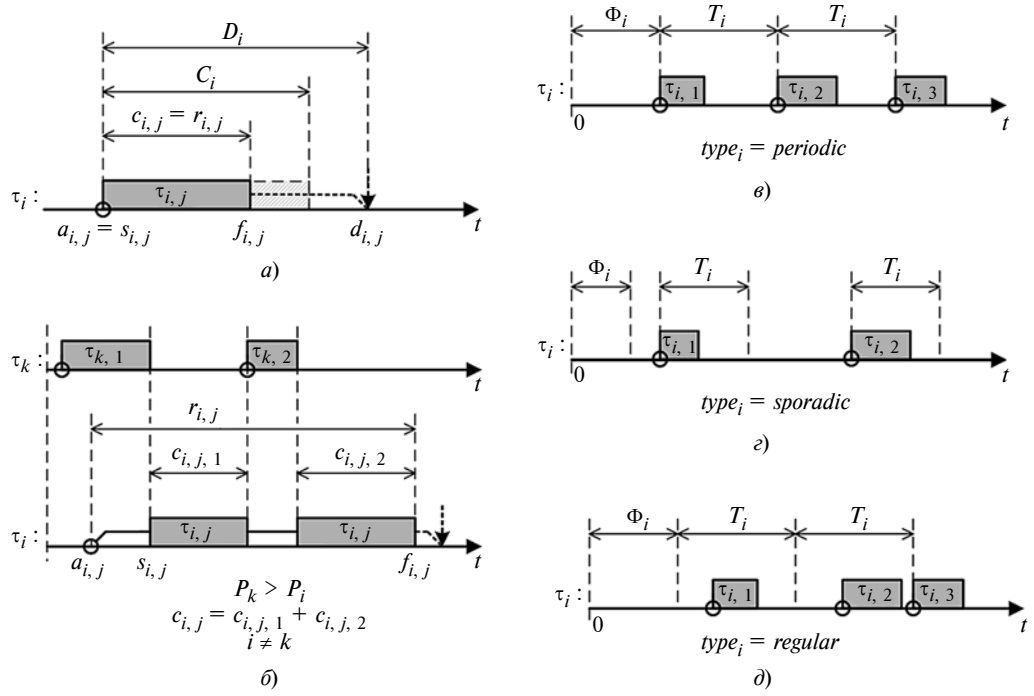


Рис. 4. Примеры выполнения задач в вычислительной системе:
а — выполнение запроса задачи без прерываний; *б* — отложенное выполнение и вытеснение запроса задачи; *в* — инициации и выполнение запросов периодической задачи; *г* — инициации и выполнение запросов спорадической задачи; *д* — инициации и выполнение запросов регулярной задачи

сов из Ψ_i. Условием работоспособности системы является достаточность ресурсов для всех задач:

$$\bigcup_{i=0}^{n} \Psi_i \subseteq \{\psi_1, \psi_2, ..., \psi_m\}.$$

Для случаев динамического распределения ресурсов или при возможности контроля характера их изменения набор используемых параметров должен быть расширен.

Пример применения модели

Одним из примеров систем РВ являются цифровые регуляторы (ЦР). Несмотря на многообразие специфических свойств [19], они могут быть практически полностью описаны представленной моделью.

Неоднородность задач. Все задачи жесткого РВ требуют реакции на все поступающие события за определенное время:

$$D_i \leqslant T_i.$$

Задачи строгого РВ могут допускать пропуск некоторых событий инициации, что может быть выражено как $D_i < \infty$.

Большинство задач, составляющих ПО ЦР, могут быть описаны ограниченным числом типов:

pf — периодические задачи строгого РВ, к которым могут быть отнесены различные технологические задачи или задачи, выполняющие только вычислительные операции;

ph — периодические задачи жесткого РВ, которые обусловлены необходимостью воздействия на внешние устройства в строго определенные моменты времени;

sh — спорадические задачи жесткого РВ, являющиеся обработчиками прерываний;

fn — фоновая задача τ_0, обладающая следующими параметрами:

$$D_0 = \infty,\ C_0 = c_{0,1} = \infty,\ a_{0,1} = 0,\ P_0 = \min\left(\bigcup_{num=0}^{n} P_{num}\right).$$

Автономность ПО. Так как ЦР является ВС с заранее известным набором внешних устройств и объектов управления, то считается что функции, выполняемые ЦР, в общем случае не изменяются после ввода его в эксплуатацию [20—22]. Соответственно, не изменяется и ПО, реализующее эти функции, и ресурсы, которыми ПО пользуется. Это условие легко выразить, если установить общее число задач *n* и ресурсов *m* постоянными:

$$\begin{cases} m = \text{const} \\ n = \text{const}. \end{cases}$$

Помимо самих задач, в автономном ПО ЦР остаются постоянными и их настройки:

$$\forall j,\ D_i = \text{const},\ C_i = \text{const},\ T_i = \text{const},$$
$$\Phi_i = \text{const},\ P_i = \text{const}. \qquad (19)$$

Приоритет P_i может быть как постоянным, так и динамическим, в зависимости от принятой парадигмы планирования. Условие (19) соответствует способу планирования с постоянными приоритетами, который является наиболее подходящим для систем РВ [4].

Статическое распределение ресурсов. Очевидно, что при выполнении τ_i использование ресурса, не входящего в Ψ_i, является нарушением функционирования. По этому признаку возможно детектирование неисправности задачи:

$$\psi(t) \not\subset \Psi_i,\ \hat{\tau}_i(t) = run.$$

Взаимозависимость задач. В ПО ЦР существует ряд программ, задачи которых зависимы друг от друга и должны соблюдать либо строгую последовательность запуска, либо определенные интервалы времени между запусками [23]. В качестве примера возьмем задачи, реализующие функции управления: τ_{sam}, τ_{com}, τ_{act} (соответственно, ввод, расчет, вывод — *sampling*, *computation*, *actuation*). Эти задачи должны выполняться строго последовательно. Ограничения предшествования можно выразить в следующем виде:

$$\forall j,\ a_{sam,\,j} < d_{sam,\,j} \leqslant a_{com,\,j} < d_{com,\,j} \leqslant a_{act,\,j} < d_{act,\,j}.$$

Стойкость задач. Повышенные требования к надежности и стойкости ПО ЦР требуют не только правильного проектирования системы, но и контроля во время выполнения. Результаты теста выполнимости [2, 5] являются актуальными только тогда, когда функционирование каждой задачи проходит согласно заданным ограничениям:

$$f_{i,\,j} \geqslant d_{i,\,j},\ c_{i,\,j} > C_i.$$

Заключение

Представленная модель отличается от использованных ранее более высоким формализмом. Ее достоинствами являются гибкость, использование общепринятой терминологии, наличие формулировок всех нововведенных терминов и связь с терминологией построения операционных систем.

Последний фактор является особенно важным — он упрощает переносимость теории планирования в практику создания операционных систем. Фактически, функция (4) является формальным представлением таблиц задач (процессов и потоков) в ОС, функция (3) является изображением работы программы-планировщика, а функция (1) аппаратно реализована во многих современных процессорах в виде устройства защиты памяти (*MPU — memory protection unit*).

Описание большинства алгоритмов планирования РВ использует параметры, представленные в формулах (5)—(18). Фактически, его полное формальное описание и будет являться функцией (3). Все описанные параметры задач могут быть использованы и при проектировании ОС. Приведенный в данной работе пример является частью работы по разработке ОС для ЦР. Особенности ЦР, выраженные в формальном виде, позволяют аналитически доказать актуальность многих нетривиальных решений, целесообразность которых раньше вызывала сомнения (например, использование нескольких программ-диспетчеров для каждого типа задач, учет диспетчера в тестах выполнимости, использование специфических архитектур ОС в системах РВ и т. д.). В частности, представленная модель будет использована в дальнейших работах по методам применения экзоядерной архитектуры для реализации ЦР.

В более общих случаях для ОС другого типа приведенные параметры могут оказаться неприменимыми или неоптимальными. Однако изменение их состава не нарушит идеологию модели.

Список литературы

1. **Bruno J. L., Coffman E. G., Graham R. L.** et al. Computer and job-shop scheduling theory. New York: Wiley, 1976.
2. **Buttazzo G. C.** Hard real-time computing systems: predictable scheduling algorithms and applications. New York: Springer, 2011.
3. **Кавалеров М. В., Матушкин Н. Н.** Применение алгоритма получения условия допустимости стандартного ограничения реального времени для примеров линейных интервальных ограничений // Вестник ПНИПУ. Электротехника, информационные технологии, системы управления. 2012. № 6. С. 104—114.
4. **Кавалеров М. В.** Планирование задач в системах автоматизации и управления при нестандартных ограничениях реального времени. дис. ... канд. техн. наук. Пермь, 2007.
5. **Tindell K. W., Burns A., Wellings A. J.** An extendible approach for analyzing fixed priority hard real-time tasks // Real-Time Systems. 1994. Vol. 6, № 2. P. 133—151.
6. **Isovic D., Fohler G.** Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints // Proc. of The 21st IEEE Real-Time Systems Symposium, 2000. IEEE, 2000. P. 207—216.
7. **Вирт Н.** Систематическое программирование. Введение. М.: Мир, 1977. 183 с.
8. **ГОСТ 19781—90** Обеспечение систем обработки информации программное. Термины и определения. 1992. 14 с.
9. **Таненбаум Э., Вудхалл А.** Операционные системы: разработка и реализация. СПб.: Питер, 2007. 704 с.
10. **Кавалеров М. В.** Современное состояние исследований и практических внедрений, связанных с проблемами планирования задач реального времени в системах управления, контроля и измерения // Труды Третьей российской конференции с международным участием "Технические и программные средства систем управления, контроля и измерения". М.: ИПУ РАН, 2012. С. 1360—1372.
11. **Дейкстра Э.** Взаимодействие последовательных процессов. Языки программирования. М.: Мир. 1972. Т. 17.
12. **Таненбаум Э.** Современные операционные системы. 3-е изд. СПб.: Питер, 2010. 1120 с.
13. **Нестеров П. В., Шаньгин В. Ф., Горбунов В. Л.** и др. Микропроцессоры: кн. 1. Архитектура и проектирование микро-ЭВМ. Организация вычислительных процессов: учебник для вузов / Под редакцией Л. Н. Преснухина. М.: Высшая школа, 1986. 495 с.
14. **Корнеев В., Киселев А.** Современные микропроцессоры. 3-е издание. СПб.: БХВ-Петербург, 2003. 448 с.
15. **Cortex-M3** Technical Reference Manual. ARM Limited, 2010. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337i/DDI0337I_cortexm3_r2p1_trm.pdf (дата обращения: 25.04.14).
16. **Бородин В. Б., Калинин А. В.** Системы на микроконтроллерах и БИС программируемой логики. М.: ЭКОМ, 2002. 400 с.
17. **32-разрядный** контроллер для авиационного применения 1986ВЕ1Т, К1986ВЕ1Т, К1986ВЕ1QI, К1986ВЕ1Н4. Спецификация ТСКЯ.431296.008СП. ЗАО "ПКК Миландр". URL: http://milandr.ru/uploads/Products/product_241/spec_1986BE1.pdf (дата обращения 20.03.2014).
18. **Микросхема** интегральная 1887ВЕ3Т. Техническое описание КФДЛ.431295.029ТО. ОАО "НИИЭТ". URL: http://www.niiet.ru/chips/microcontrollers?id=160 (дата обращения 20.03.2014).
19. **Соколов А. О., Терентьев О. А.** Показатели качества программного обеспечения цифровых регуляторов // Вестник ВГТУ. 2013. Т. 9, № 4. С. 98—102.
20. **Садомцев Ю. В.** Основы анализа дискретных систем автоматического управления: Учебное пособие. Саратов: Изд-во СГТУ, 1998. 94 с.
21. **Camposano R., Wilberg J.** Embedded system design // Design Automation for Embedded Systems. 1996. Vol. 1, № 1—2. P. 5—50.
22. **Gajski D. D., Abdi S., Gerstlauer A.** et al. Embedded System Design: Modeling, Synthesis and Verification. New York: Springer, 2009.
23. **Marti P. Fuertes J. M., Fohler G.** et al. Jitter compensation for real-time control systems // Proc. of 22nd IEEE Real-Time Systems Symposium, 2001. (RTSS 2001). IEEE, 2001. P. 39—48.

УДК 519.688

С. Ю. Болотова, препод., e-mail: bolotova.svetlana@gmail.com,
С. Д. Махортов, д-р. физ.-мат. наук, доц., зав. каф., Воронежский государственный университет

Параллельные алгоритмы релевантного LP-вывода

Релевантный LP-вывод представляет эффективное средство для разработки, верификации и оптимизации продукционно-логических систем. В данной работе изложены усовершенствованные базовые положения теории LP-структур. Приведены описания параллельных алгоритмов решения продукционно-логических уравнений в слоях, а также алгоритмов исследования начальных прообразов на предмет истинности. Результаты могут быть применены для оптимизации логического вывода и верификации соответствующих баз знаний.

Ключевые слова: *LP-структура, логические уравнения, продукционные системы, обратный вывод, релевантность, параллелизм*

S. Yu. Bolotova, S. D. Makhortov

Parallel Algorithms of the Relevant LP Inference

The relevant LP inference is an effective tool for development, verification and optimization of the production logical systems. This paper presents the advanced basic tenets of the LP structures theory. It also describes parallel algorithms for finding the solution of the production logic equations in layers, for processing the initial preimages. The results can be used to the logic inference optimization and the knowledge bases verification.

Keywords: *LP structure, logical equations, production systems, backward inference, the relevance, parallelism*

Введение

При решении задач представления знаний и управления знаниями достаточно эффективными оказываются алгебраические методы [1, 2].

В работах [3, 4] предложен математический аппарат, позволяющий рассматривать формализацию знаний и моделирование продукционно-логического вывода с точки зрения теории решеток и отношений. Согласно данному подходу, информация о предметнои области хранится в виде иерархической структуры — решетки [5]. Продукционные связи (т. е. совокупность правил) представляются дополнительным бинарным отношением с некоторыми логическими свойствами (рефлексивность, транзитивность и т. д.). Решетка с заданным на ней логическим отношением была названа LP-структурой (*Lattice Production Structure*).

Стратегия релевантного LP-вывода направлена на минимизацию числа медленно выполняемых запросов (к базе данных или интерактивному пользователю). Запросы по возможности соответствуют фактам, которые действительно необходимы при выводе. Отрицательный ответ на запрос исключает последующие запросы об элементах связанного подмножества фактов. Кроме того, при LP-выводе предпочтение отдается тестированию множеств фактов минимальной мощности.

Отмеченные выше теоретические положения получили реализацию в виде готовых алгоритмов [6]. Эксперименты показывают [7], что применение теории LP-структур позволяет существенно повысить эффективность обратного продукционно-логического вывода. Использование в LP-выводе параллельных вычислений позволило добиться еще более весомых результатов [8].

Представленные в настоящей статье результаты, в отличие от описанных в работах [3, 4], основаны на более слабых условиях на LP-структуру. Основное изменение связано с использованием вместо решеточного отношения частичного порядка $\supseteq$ отношения вида $\supseteq_R$, содержащего лишь необходимое для исследования отношения R подмножество $\supseteq$ в основных определениях, формулировках теорем и доказательствах. Кроме того, оптимизирована схема процесса решения продукционно-логического уравнения, в результате чего из нее исключен избыточный этап. Наконец, в настоящей работе приведена и использована

усовершенствованная схема "конвейера" параллельных вычислений, применение которой позволяет повысить степень параллелизма.

Статья состоит из следующих основных разделов. В разд. 1 введены необходимые базовые понятия и обозначения, сформулированы основные математические результаты. В целях сокращения объема текста доказательства не приведены. В разд. 2 показано, как изложенные теоретические концепции могут быть применены для формализации экспертных продукционных систем. В этом же разделе дана формальная постановка задачи релевантного вывода. В разд. 3 описана реализация предложенных идей в виде алгоритмов с применением параллельных вычислений. В заключении подведены итоги исследования и указаны некоторые перспективы рассмотренного метода.

1. Основная терминология и теоретические результаты

Обозначения и определения, необходимые для дальнейшего изложения, были введены в работах [3, 4]. Здесь приведены их уточнения и развитие. Необходимые сведения о математических решетках даны в работе [5].

Под *LP-структурой* подразумевается алгебраическая система, представляющая решетку, на которой задано дополнительное бинарное отношение, обладающее некоторыми продукционно-логическими свойствами. Решетка в контексте данного определения рассматривается в широком смысле, и ее тип может уточняться в конкретных моделях.

Отношение называется *продукционно-логическим*, если оно обладает рефлексивностью, т. е. содержит все пары вида (a, a), транзитивностью и другими свойствами, которые определяются конкретной моделью. Одно из таких свойств — *дистрибутивность*. Неформально дистрибутивность отношения означает возможность логического вывода по частям и объединения его результатов на основе решеточных операций $\wedge$ и $\vee$. Заданное на абстрактной решетке отношение R называется:

$\wedge$-дистрибутивным, если из $(a, b_1), (a, b_2) \in R$ следует $(a, b_1 \wedge b_2) \in R$;

$\vee$-дистрибутивным, если из $(a_1, b), (a_2, b) \in R$ следует $(a_1 \vee a_2, b) \in R$.

Отношение называется дистрибутивным при наличии обоих указанных свойств.

В настоящей работе в качестве основы LP-структур рассматриваются решетки с семантикой подмножеств — $\lambda(F)$ (множество конечных подмножеств F) или булеан 2^F (множество всех подмножеств F). Поэтому вместо символов $\leqslant$, $\geqslant$, $\wedge$ и $\vee$ используются знаки теоретико-множественных операций $\subseteq$, $\supseteq$, $\cap$ и $\cup$, а элементы решетки обозначаются, как правило, большими буквами. Под дистрибутивностью отношения подразумевается лишь второе из двух указанных выше свойств. В такой нотации $\cup$-дистрибутивность трактуется в следующем смысле: из $(A, B_1), (A, B_2) \in R$ следует $(A, B_1 \cup B_2) \in R$.

Пусть на решетке $\mathbb{F}$ задано некоторое бинарное отношение R. Совокупность всех атомов решетки, содержащихся в элементах пар отношения R, будем называть множеством атомов, которыми оперирует отношение R. Построенное на объединениях этих атомов подмножество исходной решетки обозначим $\mathbb{F}_R$. Очевидно, $\mathbb{F}_R$ — подрешетка в $\mathbb{F}$.

Для заданного R введем бинарное отношение $\supseteq_R$ — такое подмножество решеточного частичного порядка $\supseteq$, что элементы всех пар $\supseteq_R$ принадлежат $\mathbb{F}_R$. Обозначим также $\supset_R$ подмножество $\supseteq_R$, не содержащее рефлексивных пар (вида (A, A)).

Определение 1. Бинарное отношение R на решетке называется продукционно-логическим (или просто *логическим*), если оно содержит $\supseteq_R$, дистрибутивно и транзитивно.

В отличие от работ [3, 4], использующих в аналогичной ситуации отношение $\supseteq$, определение 1 формулирует *более слабое условие* на определяемое понятие и, следовательно, открывает возможности для получения новых результатов.

Рассматриваемый тип отношений относится к так называемым *монотонным* отношениям. Можно показать, что при определении логического отношения условие дистрибутивности можно заменить условием монотонности:

$$\text{если } (A, B) \in R, \text{ то } (A, A \cup B) \in R \ (\forall A, B \in \mathbb{F}).$$

Для классов продукционно-логических отношений актуальны следующие основные вопросы: о замыкании, эквивалентных преобразованиях, канонической форме, логической редукции. Логическим замыканием произвольного бинарного отношения R называется наименьшее продукционно-логическое отношение, содержащее R. Два отношения R_1, R_2 называют *эквивалентными*, если их логические замыкания совпадают. Для таких отношений используется обозначение $R_1 \sim R_2$. *Эквивалентным преобразованием* данного отношения называют некоторую замену подмножества его пар, приводящую к эквивалентному отношению.

Ниже приведены теоремы о существовании логического замыкания для произвольного бинарного отношения и о принципе локальности эквивалентных преобразований логических отношений. Эти результаты открывают возможности автоматических преобразований баз знаний.

Теорема 1. Для произвольного бинарного отношения R на решетке логическое замыкание существует и совпадает со множеством $\xrightarrow{R}$ всех упорядоченных пар, логически связанных отношением R.

Теорема 2. Пусть R_1, R_2, R_3, R_4 — отношения на общей решетке. Если при этом $R_1 \sim R_2$ и $R_3 \sim R_4$, то $R_1 \cup R_3 \sim R_2 \cup R_4$.

Отношение R на атомно-порожденной решетке $\mathbb{F}$ называют каноническим, если оно задано множеством пар вида (A, a), где $A \in \mathbb{F}$, a — атом в $\mathbb{F}$.

Утверждение 1. Для любого отношения на атомно-порожденной решетке существует эквивалентное ему каноническое отношение.

Рассмотрим вопрос о минимизации бинарных отношений с сохранением их свойств. *Логической редукцией* отношения R на решетке называется любое минимальное отношение, эквивалентное R. Справедлива теорема о существовании логической редукции и способе ее построения, аналогичная доказанной в работе [3].

Еще раз подчеркнем, что изложенные выше результаты содержат существенное усиление по сравнению с соответствующими теоремами, приведенными в работах [3, 4]. Новые теоремы имеют похожие формулировки, однако основаны на использовании в определении логических отношений операции $\supseteq_R$ вместо отношения частичного порядка $\supseteq$, как правило, бесконечного в бесконечной решетке.

В работе [3] был введен новый класс продукционно-логических уравнений и обоснован метод их решения, что в применении к продукционным системам соответствует полному обратному выводу. Указанные результаты переносятся и на LP-структуры, рассматриваемые в настоящей статье. На уравнениях остановимся немного подробнее, поскольку они играют основную роль в качестве теоретической основы дальнейшего изложения.

Пусть дано некоторое бинарное отношение R на решетке $\mathbb{F}$ и справедливо $A \xrightarrow{R} B$. Тогда B называют образом A, а A — прообразом B при отношении $\xrightarrow{R}$. Любой элемент $B_1 \subset B$ будет образом A и каждый $A_1 \supset A$ является прообразом B.

Для данного элемента $B \in \mathbb{F}$ минимальным прообразом при отношении $\xrightarrow{R}$ называют такой элемент $A \in \mathbb{F}$, что $A \xrightarrow{R} B$ и A является минимальным, т. е. не содержит никакого другого $A_1 \in \mathbb{F}$, для которого $A_1 \xrightarrow{R} B$.

Определение 2. Атом x решетки $\mathbb{F}$ называют начальным при отношении R, если в R нет ни одной такой пары (A, B), что x содержится в B и не содержится в A. Элемент X решетки $\mathbb{F}$ называют начальным, если все его атомы являются начальными при отношении R. Подмножество $\mathbb{F}_0(R)$ (будем обозначать $\mathbb{F}_0$, если это не вызовет неоднозначностей) решетки $\mathbb{F}$, состоящее из всех начальных элементов $\mathbb{F}$, называют начальным множеством решетки $\mathbb{F}$ (при отношении R).

Очевидно, начальное множество $\mathbb{F}_0$ образует подрешетку в $\mathbb{F}$.

Обозначим R^L логическое замыкание отношения R и рассмотрим уравнение

$$R^L(X) = B, \tag{1}$$

где $B \in \mathbb{F}$ — заданный элемент решетки; $X \in \mathbb{F}$ — неизвестный.

Определение 3. Частным решением X уравнения (1) называется любой минимальный прообраз элемента B, содержащийся в $\mathbb{F}_0$. Приближенным (частным) решением X уравнения (1) называют любой прообраз элемента B, содержащийся в $\mathbb{F}_0$. Общим решением уравнения (1) называют совокупность всех его частных решений $\{X_s\}$, $s \in S$.

По определению точное решение будет и приближенным. Приближенное решение всегда содержит хотя бы одно точное решение [3].

Уравнения вида (1) называют продукционно-логическими (или просто логическими) уравнениями на решетках.

Далее будем предполагать, что R — конечное каноническое отношение на атомно-порожденной решетке $\mathbb{F}$, не содержащее пар отношения $\supseteq_R$, а правая часть B уравнения (1) представляет собой конечное объединение атомов.

Аналогично [3] введем понятие структурного расслоения исходного отношения R на виртуальные слои (частичные отношения) $\{R_t | \ t \in T\}$, позволяющее облегчить построение и исследование ряда алгоритмов, связанных с решением соответствующих уравнений. Кроме того, концепция слоев лежит в основе распараллеливания процесса нахождения решений.

С указанной целью вначале разобьем R на непересекающиеся подмножества, каждое из которых образовано парами вида (A, x_p) с одним и тем же атомом x_p в качестве правой части. Такое разбиение имеет смысл благодаря тому, что R является каноническим. Обозначим эти подмножества R^p соответственно их элементу x_p, $p \in P$.

Определение 4. Слоем R_t в отношении R называют подмножество R, образованное упорядоченными парами, взятыми по одной из каждого непустого R^p, $p \in P$. Два слоя, отличающиеся хотя бы одной парой, считают различными.

Справедлива следующая теорема.

Теорема 4. Для нахождения общего решения уравнения (1) достаточно найти частное решение X_t в каждом слое R_t, если оно существует. Далее из полученного множества решений необходимо исключить элементы, содержащие другие элементы этого же множества.

Теорема 4 позволяет свести вопрос о решении уравнения (1) к задаче нахождения частного решения уравнения

$$R_t^L(X) = B, \tag{2}$$

где B — неначальный элемент решетки $\mathbb{F}$; R_t — произвольный слой в R.

Аналогично [3] можно показать, что для решения уравнения (2) достаточно решить уравнение с каждым атомом элемента B в качестве правой части. Принимая во внимание это обстоятельство, рассмотрим задачу нахождения частного решения следующего уравнения:

$$R_t^L(X) = b, \tag{3}$$

где b — неначальный атом решетки $\mathbb{F}$; R_t — произвольный слой в R.

В работе [3] подробно рассмотрены методы решения этой задачи. Показано, что в отдельном слое решение уравнения сводится к задаче перечисления множества входных вершин, соответствующего слою ориентированного графа $G_{R_i, b}$. Его вершины соответствуют атомам решетки, дуги — парам слоя отношения R_i. Аналогичные результаты справедливы и в рамках усовершенствованного класса LP-структур, используемого в настоящей статье.

Заметим также, что выше было внесено изменение в пошаговую схему решения уравнения, описанную в работе [3]. Разбиение исходного отношения на слои проводится сразу, до упрощения правой части уравнения. В результате из процесса решения исключается избыточный этап, связанный с идентификацией приближенных решений.

2. Об алгоритмах релевантного вывода

Аппарат продукционно-логических уравнений предоставляет новые возможности для оптимизации обратного вывода в продукционной системе. При описании применения рассматриваемой методики к исследованию баз знаний и моделированию обратного вывода будем использовать связанную с экспертными продукционными системами терминологию, введенную в работе [9].

Предлагаемый подход к исследованию баз знаний и усовершенствованию обратного вывода основан на представлении наборов фактов и правил LP-структурой. Каждый элементарный факт изображается атомом решетки, а именно булеана, предпосылка и заключение правила — соответствующими элементами решетки, а сами правила представляются парами бинарного отношения в LP-структуре. Обратный продукционно-логический вывод в данной модели может быть осуществлен путем нахождения решений уравнения вида (1) с гипотезой в правой части. При этом искомый начальный прообраз гипотезы должен содержаться в рабочей памяти.

Стратегия релевантности может быть направлена на снижение числа обращений к внешним источникам информации. Переходя от экспертной системы к терминологии LP-структур, сформулируем соответствующую релевантному выводу следующую задачу нахождения истинного начального прообраза.

Даны конечная атомно-порожденная решетка $\mathbb{F}$ (атомы изображают элементарные факты) и на ней бинарное отношение R (представляет совокупность продукционных правил). Не ограничивая общности можно считать, что R является каноническим отношением на атомно-порожденной решетке, а правая часть уравнения (1) представляет собой конечное объединение атомов. Тогда, в силу теорем из разд. 1, вместо (1) достаточно рассмотреть уравнения с атомами в правой части:

$$R^L(X) = b. \qquad (4)$$

Пусть выбран атом $b \in \mathbb{F}$, которому соответствует общее решение уравнения (4) — множество $\{X\}$ всех его минимальных начальных прообразов. На множестве атомов решетки введем частично определенную булеву функцию *True* (функцию истинности), которую можно доопределять путем обращения к внешнему источнику информации. В моделируемой продукционной системе интерпретация данной функции такова:

- $True(x) = 1$, если соответствующий факт x содержится в рабочей памяти;
- $True(x) = 0$, если достоверно известно, что x не может содержаться в рабочей памяти;
- $True(x) = null$, если проверка x еще не проводилась.

Введем обозначения $T = \cup x_k$ $(True(x_k) = 1)$; $F = \cup x_j$ $(True(x_j) = 0)$. Необходимо найти такой элемент $X^0 \in \{X\}$, что $X^0 \subseteq T$ (если он существует). В процессе решения задачи требуется также обойтись как можно меньшим числом доопределений функции *True*.

Последнее требование лежит в основе метода релевантного вывода. Метод состоит из двух стадий: 1) решение уравнения — вычисление множества начальных прообразов и 2) нахождение среди них истинного прообраза. Выполнение указанных стадий может быть организовано в виде конвейера — независимо и параллельно. Работа на каждой из стадий также может быть распределена между параллельными вычислителями.

Упрощенные описания нерекурсивного алгоритма обычного обратного вывода и релевантного вывода приведены в работе [6]. Там же описан алгоритм кластерно-релевантного LP-вывода, который предполагает последовательное построение кластеров начальных прообразов (подмножеств ограниченной мощности) с их динамическим релевантным исследованием. В работе [8] показаны возможности распараллеливания этих алгоритмов. Следующий раздел настоящей статьи посвящен параллельным алгоритмам релевантного вывода, основанным на усовершенствованной схеме "конвейера" параллельных вычислений, использование которой позволяет повысить степень параллелизма.

3. Параллельная реализация

При больших объемах баз знаний и их достаточно "глубокой" структуре необходимо использовать имеющиеся в наличии возможности повышения эффективности логического вывода. В связи с этим метод релевантного LP-вывода модифицирован с использованием параллельных вычислений. Параллельный релевантный LP-вывод предполагает одновременное построение набора начальных прообразов с их дальнейшим исследованием в разных потоках. Здесь и далее под потоками подразумевают *threads* — потоки выполнения [10].

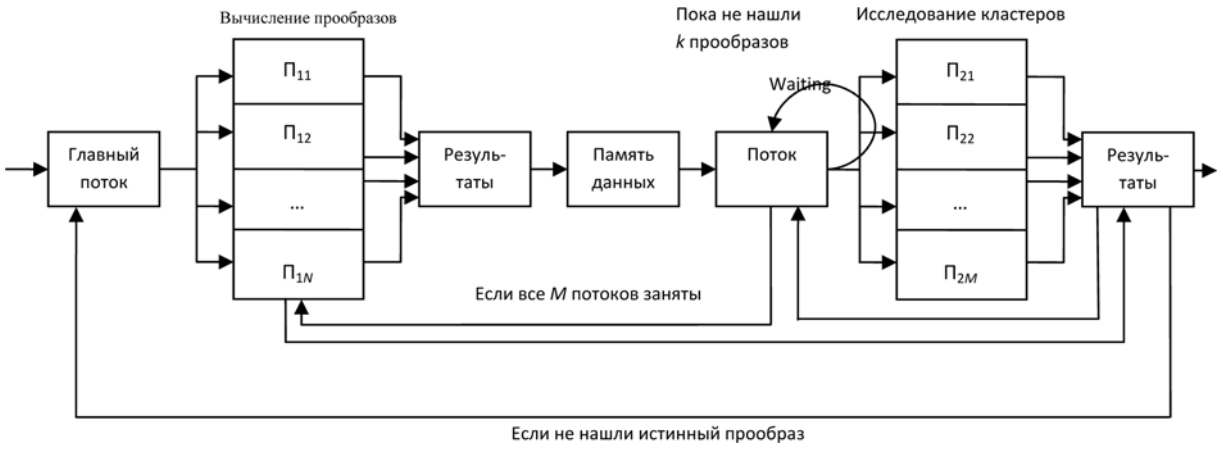


Рис. 1

Как уже было отмечено в разд. 1, для решения уравнения исходное каноническое отношение R представляется в виде слоев. Работа с различными слоями может быть организована независимо и параллельно. Ниже приведены основные положения такой организации.

Первичный поток приложения создает новые потоки (их число ограничено параметром *MaxThreads*), передавая им пакеты данных. Созданный поток вычисляет решение продукционно-логического уравнения в отдельном слое. Как только накапливается достаточное число прообразов (кластер или все прообразы), модуль LP-вывода начинает (релевантно) их исследовать на предмет истинности, обращаясь при необходимости за фактами к внешнему источнику. Если при обработке очередного кластера прообразов истинный прообраз выявить не удается, то процесс вычисления прообразов продолжается. При этом запоминается информация об установленных на предыдущем шаге ложных фактах. На ее основе перед очередным процессом вычисления прообразов сужается множество актуальных правил. Это обстоятельство также существенно ускоряет работу. По окончании работы вторичные потоки ликвидируются.

В случае когда рабочих потоков оказывается слишком много, в соответствии с законом Амдала эффективность параллельных вычислений снижается [10]. Интенсивное создание и завершение потоков с малым временем работы, а также множественные переключения их контекстов увеличивают объем расходуемых ресурсов. Поэтому при реализации параллельного LP-вывода используется заранее создаваемый пул потоков [10], максимальный размер которого ограничен специальным параметром. Его значение должно быть бóльшим, чем число процессоров, чтобы добиться приемлемой степени одновременности.

Главный поток выбирает поток из пула и передает ему необходимые данные для обработки. Когда число активных потоков достигает максимума, запрос помещается в очередь. В качестве механизма синхронизации потоков могут быть использованы критические секции, которые инициируются в процессе активизации.

В целях повышения производительности процессы вычисления прообразов и их дальнейшего исследования выполняются асинхронно. Пока решение не найдено, полученный на очередном шаге кластер прообразов помещается в динамическую структуру данных Q, организованную в виде очереди. Главный поток извлекает кластер прообразов из очереди Q и при наличии свободного вторичного потока выполнения инициирует параллельное исследование элементов кластера на релевантность.

Найденные на очередном этапе наиболее релевантные элементы помещаются в очередь с приоритетом P, реализованную на основе двоичной кучи [10]. Максимальный размер такой очереди задается специальным параметром *maxRelevanceQueueSize*. Подобная организация данных позволяет отсортировать исследованные элементы в соответствии с их релевантностью, при необходимости менять порядок их следования при добавлении новых элементов и за константное время получать доступ к элементу с максимальным показателем. Процесс определения релевантных объектов построенных кластеров продолжается до тех пор, пока решение не будет найдено.

Общая структура алгоритма параллельного релевантного вывода может быть представлена в виде схемы, изображенной на рис. 1.

На рис. 2 приведено более подробное описание алгоритма.

В функции *getPreImagesCluster*(b) инициируется процесс вычисления фиксированного набора минимальных начальных прообразов атома b. Как уже было отмечено, этот процесс осуществляется в отдельных потоках для обеспечения наибольшей эффективности.

Функция *getRelevantIndex*($\{Y\}$, T, *relevance* = 0) в отдельном потоке для каждого кластера находит релевантный элемент из нерассмотренных на данный момент. Она возвращает его показатель релевантности в переменной *relevance*. Далее элемент с показателем

```
// Параллельный релевантный вывод
X^0 = null
while X^0 = null do
  asynchronous call {X} = getPreImagesCluster(b)
  if {X} ≠ ∅ then
    addClusterToQueue(Q, {X}) // Добавление кластер в очередь Q
  end
  while not clustersQueueIsEmpty(Q) do in parallel // Если очередь не пуста
    {Y} = extractClusterFromQueue(Q) // Извлечение кластера из очереди
    foreach (Y_j ∈ {Y}) do
      if Y_j ⋂ F ≠ ∅ then {Y} = {Y} \ Y_j
    end
    while X^0 = null do
      asynchronous if (relevanceQueueSize(P) < maxRelevanceQueueSize)
      then in parallel // Если в P есть свободное место
        // Индекс k релевантного объекта и его показатель relevance
        foreach (Y_j ∈ {Y}) do
          if Y_j ⋂ F ≠ ∅ then {Y} = {Y} \ Y_j
        end
        k = getRelevantIndex ({Y}, T, relevance = 0)
        InsertToQueue(P, k, relevance) // Объект в очередь с приоритетом
      else
        Waiting() // Ожидание освобождения места в очереди
      end
      asynchronous while (relevanceQueueSize(P)! = 0) do
        k = ExtractMaxFromQueue(P) // Элемент с максимальной релевантностью
        Ask (y_k) // Определение истинности k-го факта
        foreach Y_j ∈ {Y} do
          if Y_j ⊆ T then
            X^0 = Y_j
            break
          end
          if Y_j ⋂ F ≠ ∅ then
            foreach (y_m ∈ {Y_j}) do
              // Элемент в очереди P — понижение релевантности
              if elemIsMemberOfQueue(y_m, P) then change Relevance(y_m, P)
            end
          end
        end
      end
    end
  end
end
```

Рис. 2

релевантности помещается в очередь с приоритетом. Если записываемый в очередь элемент уже находится в ней, то показатели релевантности складываются, и очередь перестраивается.

Параллельно с описанными действиями функция $Ask(y_k)$ запрашивает внешний источник об истинности наиболее релевантного факта, извлеченного из очереди с приоритетом. При получении отрицательного ответа на запрос прообразы, содержащие элемент y_k, исключаются из рассмотрения, а релевантности уже исследованных элементов модифицируются.

Далее приведен алгоритм соответствующей функции *getPreImagesCluster*(b). Обозначим множество слоев $\{R_i\}$ через R'. Как уже отмечалось, решение в каждом слое вычисляется отдельным потоком. Число активных в данный момент потоков хранится в переменной *countUsedThreads*. В случае, если в пуле есть свободный поток, он запускается и в нем вызы-

вается функция *FindEquationSolution*(R_i), которая находит решение уравнения в очередном слое R_i. Алгоритм ее работы приведен на рис. 3.

Функция *getRelevantIndex*({Y}, T, *relevance*) находит индекс k любого из наиболее релевантных и ранее непроверенных на истинность атомов, содержащихся в элементах текущего кластера прообразов {Y}, и его показатель релевантности *relevance*. При имеющемся (обычно большом) числе прообразов процесс выявления релевантных объектов весьма ресурсозатратен, поэтому он также распараллеливается. Для очередного кластера прообразов в отдельном потоке (число потоков аналогично ограничено параметром *MaxThreadsCount*) запускается процесс их релевантного исследования на предмет истинности. Число активных в данный момент потоков хранится в переменной *usedThreadsNumber*. Здесь возможны различные способы подсчета релевантности, в частности, описанные в работе [3]. Для синхронизации потоков используется механизм критических секций.

На рис. 4 представлен один из возможных параллельных вариантов функции *getRelevantIndex*.

```
// Нахождение решения уравнения на каждом слое в отдельном потоке
foreach R_i ∈ R' do in parallel
  if countUsedThreads < MaxThreads then
    BeginThread() // Начать выполнение потока
    countUsedThreads++
    FindEquationSolution(R_i)
    ExitThread() // Завершить поток
    countUsedThreads--
  else
    Waiting() // Ожидание освобождения потока
  end
end
```

Рис. 3

```
// Параллельный подсчет релевантности
int getRelevantIndex ({Y}, T, relevance)
if (usedThreadsNumber < MaxThreadsCount ) or
  (countUsedThreads < MaxThreads ) then
   BeginThread() // Начать выполнение потока
   if (usedThreadsNumber < MaxThreadsCount ) then
     usedThreadsNumber++
     usedSelfThreads = true
   else
     countUsedThreads++
     usedSelfThreads = false
   end
   k = MostRelevantFind(relevance) // Индекс релевантного атома
   ExitThread() // Завершить поток
   if usedSelfThreads then
     usedThreadsNumber--
   else
     countUsedThreads--
   end
  return k
else
  Waiting() // Ожидание освобождения потока
end
```

Рис. 4

Используемая в этом алгоритме функция *MostRelevantFind*(*relevance*) позволяет найти индекс k наиболее релевантного объекта и запомнить его показатель релевантности в переменной *relevance*.

Эксперименты показывают, что параллельный LP-вывод дает достаточно эффективные результаты для обработки баз знаний больших размеров.

Использование метода релевантного вывода приводит к уменьшению числа вызовов функции $Ask(y_k)$ — обращений за информацией к внешнему источнику, хотя при этом в основной памяти может проводиться значительно больше вычислений по сравнению с обычным обратным выводом. Формальное обоснование данного утверждения представляет собой задачу, существенно зависящую от исходных данных. Однако эффективность LP-вывода обоснована результатами экспериментов и их статистической обработкой: число внешних запросов в среднем снижается на 15...20 %.

Заключение

В настоящей работе сформулирована и исследована обобщенная модель LP-структуры, расширяющая область применения этой алгебраической теории. Разработана усовершенствованная схема "конвейера" параллельных вычислений при LP-выводе. На ее основе приведены параллельные алгоритмы релевантного и кластерно-релевантного LP-вывода, а также параллельные алгоритмы вычисления истинных прообразов в LP-структурах, повышающие быстродействие описываемого метода.

Поскольку многие модели в информатике имеют продукционный характер [11], дальнейшие исследования на рассматриваемом направлении могут быть связаны с выбором LP-структур более сложных типов и соответствующих им предметных областей, перенос концепций настоящей работы на эти модели. Общие методы исследования при этом останутся прежними.

Список литературы

1. **Sowa J. F.** Knowledge Representation: Logical, Philosophical and Computational Foundations. Pacific Grove, CA: Brooks Cole Publishing Co., 1999. 608 p.

2. **Бениаминов Е. М.** Алгебраические методы в теории баз данных и представлении знаний. М.: Научный мир, 2003. 184 с.

3. **Махортов С. Д.** Логические уравнения на решетках // Вестник ВГУ. Сер. Физика, математика. 2004. № 2. С. 170—178.

4. **Болотова С. Ю.** Алгебраическая модель релевантного обратного вывода на основе решения уравнений // Математическое моделирование. 2012. № 12, Т. 24. С. 3—8.

5. **Биркгоф Г.** Теория решеток: пер. с англ. М.: Наука, 1984. 568 с.

6. **Болотова С. Ю., Махортов С. Д.** Алгоритмы релевантного обратного вывода, основанные на решении продукционно-логических уравнений // Искусственный интеллект и принятие решений. 2011. № 2. С. 40—50.

7. **Махортов С. Д.** Интегрированная среда логического программирования LPExpert // Информационные технологии. 2009. № 12. С. 65—66.

8. **Болотова С. Ю.** Реализация многопоточности в релевантном LP-выводе // Программная инженерия. 2014. № 1. С. 12—18.

9. **Sowyer B., Foster D.** Programming Expert Systems in Pascal. John Wiley & Sons, Inc., 1986. 186 p.

10. **Рихтер Дж.** Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows: пер. с англ. 4-е изд. СПб.: Питер; М.: Русская Редакция, 2001. 732 с.

11. **Махортов С. Д.** Основанный на решетках подход к исследованию и оптимизации множества правил условной системы переписывания термов // Интеллектуальные системы. 2009. Т. 13, вып. 1—4. С. 51—68.

ИНФОРМАЦИЯ

13—16 октября 2014 г. в Объединенном институте ядерных исследований (г. Дубна) при поддержке Российского фонда фундаментальных исследований, Института проблем информатики РАН, Московской секции ACM SIGMOD будет проводиться очередная XVI Всероссийская научная конференция с международным участием RCDL-2014 — «Электронные библиотеки: перспективные методы и технологии, электронные коллекции».

В рамках конференции RCDL-2014 планируется проведение традиционного семинара молодых ученых "Диссертационные исследования по тематике информационных технологий, связанных с электронными библиотеками", на котором авторам работ, отобранных на основе предварительного рецензирования, будет дана возможность представить текущие результаты своих исследований, а также обсудить их сильные и слабые стороны с более опытными коллегами.

Подробную информацию о конференции можно найти на сайте ***http://rcdl2014.jinr.ru***

УДК 004.75 + 004.41

А. С. Шундеев, канд. физ.-мат. наук, вед. науч. сотр., НИИ механики МГУ имени М. В. Ломоносова,
e-mail: alex.shundeev@gmail.com

Виртуальный компьютерный класс

Работа посвящена описанию архитектуры распределенной программной системы, предназначенной для проведения практических занятий по изучению языков программирования и смежным темам. С технической точки зрения задача сводится к удаленному запуску процесса операционной системы на одном из доступных сетевых хостов и обеспечению интерактивного взаимодействия с запущенным процессом через его стандартные потоки ввода/вывода. В качестве процессов запускаются командные интерпретаторы (например, bash). Конечный пользователь взаимодействует с удаленным процессом через веб-интерфейс, в котором эмулируется терминал операционной системы.

Ключевые слова: *распределенные системы, Erlang/OTP, WebSocket, клиентский JavaScript, удаленный терминал*

A. S. Shundeev

Virtual Computer Classroom

This paper describes a distributed software architecture of a system for practical lessons in the study of programming languages and related topics. From a technical point of view, the problem is reduced to running processes remotely operating system on one of the available network hosts, and provide interactivity with a running process via its standard input/output streams. As the process runs a shell (for example, bash). The end user interacts with the remote process through the web interface, which emulates a terminal operating system.

Keywords: *distributed systems, Erlang/OTP, WebSocket, client JavaScript, remote terminal*

Введение

Учебные курсы по изучению языков программирования и по смежным дисциплинам подразумевают проведение практических занятий в компьютерном классе. Работа в компьютерном классе существенно отличается от выполнения заданий в домашних условиях. Во-первых, это непосредственное взаимодействие с преподавателем. Всегда есть возможность уточнить условие задания, получить справочную информацию по языку программирования или рабочему окружению, а также написать трудный участок программы под непосредственным руководством преподавателя. Во-вторых, такая работа подразумевает использование предустановленного программного обеспечения и настроек. Нередко программа, которая была написана и выполнялась на домашнем компьютере учащегося, в компьютерном классе даже не компилируется. Поясним данную ситуацию на примере.

В случае языка программирования `C` и использования компилятора `gcc` программа, текст которой содержится в файле `prog.c`, может быть скомпилирована в исполняемый файл `prog.exe` с помощью следующей команды:

```
gcc prog.c —o prog.exe
```

В компьютерном классе компилятор вызывается с целым набором дополнительных опций:

```
-Wall -Werror -Wformat-security
-Wignored-qualifiers
-Winit-self -Wswitch-default
-Wfloat-equal -Wshadow
-Wpointer-arith -Wtype-limits
-Wempty-body -Wlogical-op
-Wstrict-prototypes
-Wold-style-declaration
-Wold-style-definition
-Wmissing-parameter-type
-Wmissing-field-initializers
-Wnested-externs
-Wno-pointer-sign -std = gnu99
```

Данные опции предписывают компилятору строго придерживаться стандарта языка программирования,

а также выводить всевозможные предупреждения. Кроме того, предупреждения трактуются как ошибки, а при наличии ошибок исполняемый код не будет сгенерирован.

Существуют и ограничения, связанные с использованием компьютерного класса в учебном процессе. Данные ограничения в большей степени носят административный и организационный характер. Во-первых, использование компьютерного класса жестко связано с учебным расписанием. Каждой учебной группе выделяют фиксированное время для проведения практических занятий. Во-вторых, существуют объективные трудности с установкой дополнительного программного обеспечения. Подобные установки могут привести к возникновению нештатных ситуаций в функционировании компьютерного класса и пагубно повлиять на учебный процесс в целом.

Рис. 1. Компонентная архитектура программной системы

Описанные ограничения не могут существенным образом повлиять на основные обязательные учебные курсы. Однако они являются препятствием для проведения практических занятий по специальным курсам и для внедрения экспериментальных курсов.

Настоящая статья посвящена описанию архитектуры и принципов реализации программной системы под названием *виртуальный компьютерный класс*. Данная система предназначена для проведения практических занятий, а также выполнения домашних заданий учащимися. Использование подобной системы позволяет обойти описанные выше ограничения традиционного компьютерного класса.

Основная идея виртуального компьютерного класса состоит в следующем. Пользователь-учащийся работает с веб-приложением. Веб-приложение эмулирует терминал операционной системы типа Unix. При активации терминала на удаленном сервере запускается командный интерпретатор, например `bash`. Команды, введенные в веб-терминале, перенаправляются на стандартный поток ввода (`stdin`) запущенного интерпретатора. Данные, появившиеся в потоках стандартного вывода (`stdout`) и стандартного протокола (`stderr`), отображаются в веб-терминале.

С технической точки зрения задача сводится к обеспечению интерактивного взаимодействия с удаленным процессом операционной системы через его стандартные потоки ввода/вывода. Для этого используют методы и подходы, описанные в предыдущих работах автора [1—3].

Компонентная архитектура виртуального компьютерного класса схематично представлена на рис. 1. Дадим краткую характеристику каждому из выделенных компонентов. Веб-браузер является клиентской программой, с которой работает пользователь-учащийся. Средствами веб-браузера осуществляется эмуляция терминала. Используемый веб-сервер должен поддерживать технологию Comet. Данная технология реализует активное двунаправленное взаимодействие между веб-браузером и веб-сервером. Компонент супервизор отвечает за запуск и перехват ввода/вывода командных интерпретаторов. Одновременно может быть несколько компонентов типа супервизор. Для масштабирования системы целесообразно запускать данные компоненты на доступных вычислительных хостах. Для взаимодействия между супервизорами и веб-сервером реализуется внутренняя коммуникационная среда.

Дальнейшее содержание статьи посвящено описанию соответствующего компонента или технологии. При этом используется следующий подход. Разделяется концептуальное описание архитектуры виртуального компьютерного класса и реализация этой архитектуры средствами программной платформы Erlang/OTP [4].

Comet-сервер

Прежде чем перейти непосредственно к технологии Comet, проанализируем базовую архитектуру веб-приложений. Технологическим фундаментом данной архитектуры являются протокол HTTP и спецификация URI (*Uniform Resource Identifier*). Теоретическим обоснованием является архитектурный стиль REST (*REpresentational State Transfer*), предложенный Р. Филдингом [5].

В соответствии с архитектурным стилем REST веб-приложение представляет собой набор *ресурсов* — поименованных данных. Взаимодействие с программной системой сводится к манипулированию подобными ресурсами через *унифицированный интерфейс*. Для наименования ресурсов используется спецификация URI, а для реализации унифицированного интерфейса — протокол HTTP.

Унифицированный интерфейс содержит ограниченный набор типов операции. К числу основных операций относятся следующие: создать/изменить/удалить ресурс, получить состояние ресурса, а также получить список всех ресурсов заданного типа. Данные операции соответствуют методам POST, PUT, DELETE и GET протокола HTTP.

Отметим, утверждение, что метод GET возвращает состояние ресурса, не совсем верно. Метод GET возвращает *представление* (*representation*) состояния ресурса. У одного и того же ресурса может одновременно поддерживаться несколько форматов представлений, например, HTML, XML, XHTML, JSON.

Веб-приложения, ориентированные на конечных пользователей, используют незначительное подмножество средств архитектурного стиля REST (представление HTML, XHTML) и протокола HTTP (методы GET и POST) [6]. Это связано с тем, что у таких при-

```
01. var conn = new WebSocket("ws://localhost:8081/term");
02. conn.onmessage = onMessage;
03.   ...
04. function onMessage(evt) {
05.     // Treat evt.data.
06. }
07.   ...
08. conn.send(...);
```

Рис. 2. Листинг 1. Работа с веб-сокетом на стороне клиента

ложений в качестве основного клиента используется веб-браузер, который не поддерживает других средств. В дальнейшем представления в форматах HTML, XHTML будем называть веб-страницами.

Для манипулирования содержанием веб-страницы на стороне веб-браузера поддерживается специальный программный интерфейс DOM (*Document Object Model* — объектная модель документа). Интерфейс DOM представляет содержимое веб-страницы в виде дерева его составных элементов (DOM-дерево). В качестве языка манипулирования выступает язык программирования JavaScript. Существует возможность добавлять/удалять узлы в DOM-дереве, изменять атрибуты узлов. При этом будет соответственно изменяться отображение веб-страницы в окне веб-браузера.

Повторное выполнение запроса GET (получение текущего состояния ресурса с сервера) или выполнение запроса POST (создание/изменение состояния ресурса на стороне сервера) приводит к разрушению старого DOM-дерева и построению нового. При этом, естественно, информация, которая содержалась в старом DOM-дереве, будет потеряна.

Для того чтобы осуществлять взаимодействие с веб-сервером и не разрушать структуру DOM-дерева страницы используется технология AJAX [7]. Суть ее состоит в следующем. При обработке некоторого события веб-страницы осуществляется HTTP-запрос на веб-сервер (как правило, для этого используется метод POST и формат сообщений JSON). Далее, в том же обработчике осуществляется ожидание ответа от сервера. После ответ разбирается и в соответствии с его содержимым вносятся изменения в существующее DOM-дерево.

Описанные методы взаимодействия подразумевают активность веб-браузера (инициатора запросов) и пассивность веб-сервера, который только реагирует на приходящие запросы. Таким образом, можно говорить об одностороннем взаимодействии.

В случае виртуального компьютерного класса данные на сервере могут формироваться постепенно и по мере их формирования должны отправляться клиенту. Таким образом, возникает потребность в организации активного двустороннего взаимодействия между веб-браузером и веб-сервером. Технологии, реализующие модель подобного взаимодействия, получили название Comet.

Технология Comet может быть смоделирована при помощи средств AJAX. При помощи таймера через некоторые промежутки времени выполняется AJAX-запрос на сервер. Подобный запрос проверяет "готовность" данных на сервере и в случае готовности "забирает" их.

Протокол WebSocket [8] напрямую реализует технологию Comet. Рассмотрим принципы работы с веб-сокетами на стороне веб-браузера (см. листинг 1, рис. 2).

В строке 01 при помощи конструктора `WebSocket` создается объект `conn`. Данный объект представляет собой концевую точку двустороннего взаимодействия на стороне клиента. Конструктор `WebSocket` в качестве аргумента принимает URI концевой точки взаимодействия на стороне сервера.

Объект `conn` содержит поля `onopen`, `onmessage`, `onerror`, `onclose`. Данные поля должны содержать функции-обработчики, которые вызываются соответственно при открытии веб-сокета, при получении сообщения с сервера, при возникновении ошибки и при закрытии веб-сокета. В строках 04—06 приведен фрагмент определения функции-обработчика `onMessage`, вызываемой при получении сообщения с сервера. В строке 02 осуществляется регистрация этой функции. Функция `onMessage` в качестве аргумента получает объект `evt`. Поле `data` этого объекта содержит собственно данные с сервера.

Для того чтобы передать данные на сервер необходимо вызвать метод `send` (строка 08) у объекта `conn`. Данные, отправляемые на сервер, передаются в качестве входного аргумента метода `send`.

При реализации виртуального компьютерного класса использовался написанный на языке Erlang веб-сервер Cowboy [9]. Данный веб-сервер поддерживает протокол WebSocket, в связи с чем этот протокол и был выбран для реализации Comet-сервера виртуального компьютерного класса.

Пользовательский интерфейс

Клиентское приложение виртуального компьютерного класса устроено следующим образом. В веб-браузере загружается страница, на которой создается специализированный элемент управления — терминал, эмулирующий функциональные возможности консоли операционной системы Unix. После этого создается веб-сокет для двунаправленного взаимодействия с сер-

```
01. function createTerm(term_div) {
02.     var term;
03.     var conn;
04.
05.     function termHandler(cmd, t) {
06.         if(conn)
07.             conn.send(cmd);
08.         else
09.             t.error("Not connected!");
10.     }
11.
12.     term = $(term_div).terminal(
13.         termHandler, {height: 400, width: 600});
14.
15.     function onMessage(evt) {
16.         term.echo(evt.data);
17.     }
18.
19.     conn = new WebSocket("ws://localhost:8081/term");
20.     conn.onmessage = onMessage;
21. }
```

Рис. 3. Листинг 2. Создание веб-сокета и терминала

вером. Команды, введенные в терминале, через веб-сокет передаются на сервере. Пришедшие данные с сервера отображаются в терминале. Для эмуляции терминала используется jQuery-библиотека terminal [10].

На самом деле, на одной странице одновременно может присутствовать несколько терминалов, независимо друг от друга взаимодействующих с сервером. Для создания отдельных взаимодействующих друг с другом пар (терминал, веб-сокет) используется JavaScript-процедура `createTerm` (см. листинг 2, рис. 3).

Данная процедура на вход получает параметр `term_div` — jQuery-селектор, указывающий на структурный элемент страницы типа `div`, в котором должен быть отрисован терминал.

В строках 02 и 03 объявлены переменные `term`, `conn`. Данные переменные будут содержать объекты, соответствующие созданному терминалу и открытому веб-сокету.

Терминал создается в строках 12—13. Соответствующий конструктор `terminal` на вход получает два аргумента. Первый аргумент — функция-обработчик `termHandler`, которая вызывается каждый раз, когда пользователь ввел в терминале команду. Второй аргумент задает параметры для отображения терминала. В рассматриваемом примере задаются высота и ширина терминала.

Функция-обработчик `termHandler` определяется в строках 05—10. Данная функция имеет два входных параметра: `cmd` — введенные данные и `t` — ссылка на терминал, для которого был вызван обработчик. В строке 06 осуществляется проверка — был ли открыт веб-сокет. Если он был открыт, то данные `cmd` отправляются на сервер (строка 07). В противном случае в терминале выводится сообщение об ошибке (строка 09).

Веб-сокет создается в строке 19. В строке 20 происходит регистрация функции-обработчика `onMessage`, которая вызывается каждый раз, когда с сервера приходит новая порция данных. Функция `onMessage` (строки 15—17) выводит полученные с сервера данные в терминал.

В дополнение к листингу 2 (рис. 3) должен быть реализован следующий механизм. Для отображения данных в терминале используется функция `echo`. Эта функция после вывода данных осуществляет переход на новую строку. Данные с севера могут поступать неравномерно. Вначале может прийти порция, содержащая начальную часть строки, затем порция, содержащая окончание этой строки и начало следующей строки. В результате при отображении в терминале может нарушиться желаемая разбивка текста на строки, отправляемого с сервера. Во избежание этой ситуации необходимо использовать буфер, в котором будут накапливаться порции данных, приходящие с сервера. Только после появления в буфере целых строк эти строки должны выводиться функцией `echo` в терминал.

Коммуникационная среда

Коммуникационная среда виртуального компьютерного класса (рис. 4) разделена на две части — публичную зону и доверенную зону. Веб-сервер является шлюзом между этими двумя частями. В публичной зоне конечные пользователи взаимодействуют через веб-браузеры с веб-сервером. Для этого используют протоколы HTTP (для начальной загрузки веб-страниц) и WebSocket (для организации активного двустороннего взаимодействия).

В доверенной зоне на соответствующих сетевых хостах развернуты компоненты типа супервизор, отвечающие за запуск и перенаправление ввода/вывода командных интерпретаторов операционной системы. В целях обеспечения безопасности конечные пользователи напрямую не могут взаимодействовать с супервизорами и взаимодействуют только через веб-сервер.

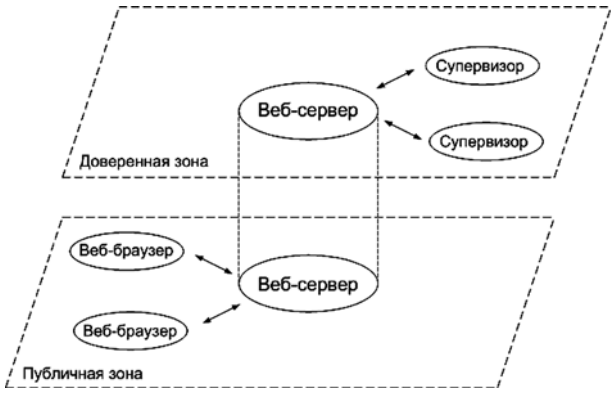


Рис. 4. Структура коммуникационной среды

Супервизоры и веб-сервер взаимодействуют друг с другом при помощи обмена сообщениями вида "тип сообщения, тело сообщения". Подробнее типы сообщений будут рассмотрены позднее в разделе, посвященном супервизору. Рассмотрим реализацию подобного обмена сообщениями средствами языка программирования Erlang [4].

Программы на языке Erlang разбиваются на модули, которые компилируются в байт-код. Байт-код выполняется виртуальной машиной. Вызов функции в Erlang имеет следующий вид:

```
module:func(A1, ... An),
```

где `module` — имя модуля, в котором объявлена функция; `func` — уникальное в рамках модуля имя функции; `A1, ... An` — входные параметры.

Функция может быть вызвана в виде создания Erlang-процесса

```
Pid = spawn(module, func, [A1, ... An]),
```

где `Pid` — переменная, в которую сохраняется уникальный идентификатор созданного процесса.

Подобные процессы выполняются параллельно и независимо друг от друга. Они являются легковесными по сравнению с процессами и потоками операционной системы. Легковесность говорит о том, что требуется значительно меньше времени на их создание/уничтожение. Эти процессы потребляют значительно меньшее количество памяти.

Erlang-процессы взаимодействуют друг с другом при помощи обмена сообщениями. Для этого используются выражения вида

```
Pid ! Msg
```

Данное выражение предписывает послать процессу с идентификатором `Pid` сообщение `Msg`. Процесс-получатель может перехватить и обработать присланное сообщение при помощи конструкции вида

```
receive
    Pattern1 -> Action1;
    Pattern2 -> Action2;
      ...
end
```

Внутри блока `receive/end` перечислены пары "образец, действие". Подобные пары просматриваются сверху вниз. Обнаружив образец, которому соответствует полученное сообщение `Msg`, будет выполнено соответствующее действие.

Необходимо отметить, что посылать друг другу сообщения могут процессы, запущенные на разных виртуальных машинах Erlang. Более того, эти виртуальные машины могут выполняться на разных сетевых хостах.

Перенаправление ввода/вывода

Перенаправление ввода/вывода удаленно запущенного интерпретатора команд в виртуальном компьютерном классе базируется на использовании механизма межпроцессного взаимодействия под названием *безымянный канал* [11]. Безымянный канал создается при помощи системного вызова `pipe`. Данный вызов возвращает пару файловых дескрипторов `fd0` и `fd1`. Дескриптор `fd0` используется для чтения данных с помощью системного вызова `read`, а `fd1` — для записи данных с помощью системного вызова `write`. Тем самым организуется однонаправленный поток данных. Создавший безымянный канал процесс может передать файловый дескриптор `fd0` (`fd1`) другому процессу. При этом процесс, создавший безымянный канал, становится поставщиком (потребителем) данных. Процесс, которому передан дескриптор, становится наоборот потребителем (поставщиком) данных.

Так как у запущенного интерпретатора команд перенаправляются стандартные потоки ввода/вывода, то необходимо дополнительно использовать системный вызов `dup2`. Данный системный вызов для существующего активного файлового дескриптора позволяет задать псевдоним.

На рис. 5 приведен пример программы (листинг 3), в котором описывается схема порождения дочернего процесса, а также перенаправление стандартных потоков ввода/вывода при помощи системных вызовов `pipe` и `dup2`.

В строках 06 и 07 описано создание двух безымянных каналов. Первый канал будет использован для передачи данных от родительского процесса к дочернему процессу, а второй канал — наоборот, от дочернего процесса к родительскому. Файловые дескрипторы первого канала сохраняются в массиве `fdPair1`, а второго канала — в массиве `fdPair2`.

В строке 09 описано порождение дочернего процесса при помощи системного вызова `fork`. Далее, родительский процесс будет выполнять фрагмент кода, расположенный в строках 11—17. Дочерний процесс будет выполнять фрагмент кода, расположенный в строках 21—24.

Рассмотрим поведение дочернего процесса. В строке 21 осуществляется закрытие стандартного потока ввода `close(0)`. После этого читающему файловому дескриптору первого безымянного канала назначается псевдоним `0`. В строках 22 и 23 осуществляется закрытие стандартных потоков вывода и протокола. Запи-

```
01. int main(void) {
02.     int fdPair1[2];
03.     int fdPair2[2];
04.     pid_t pid;
05.
06.     pipe(fdPair1);
07.     pipe(fdPair2);
08.
09.     pid = fork();
10.     if(pid > 0) { // Parent process code.
11.         char    buf[1024];
12.         ssize_t size;
13.
14.         write(fdPair1[1], "ls\n", 3);
15.         size = read(fdPair2[0], buf, 1024);
16.         buf[size] = '\0';
17.         puts(buf);
18.     }
19.     else
20.         if(pid == 0) { // Child proces code.
21.             close(0); dup2(fdPair1[0], 0);
22.             close(1); dup2(fdPair2[1], 1);
23.             close(2); dup2(fdPair2[1], 2);
24.             execl("/bin/bash", "bash", (char*)0);
25.         }
26.
27.     return 0;
28. }
```

Рис. 5. Листинг 3. Межпроцессное взаимодействие на основе безымянного канала

сывающему файловому дескриптору второго безымянного канала назначаются псевдонимы `1` и `2`. Далее, контекст исполнения дочернего процесса заменяется исполняемым кодом командного интерпретатора `/bin/bash`.

В родительском процессе (строка 14) в первый безымянный канал записывается строка `"ls\n"` (`ls` — команда операционной системы, выводящая текущее содержимое директории). В строке 15 осуществляется чтение из второго безымянного канала. Считанные данные (результат выполнения команды `ls` интерпретатором команд `bash`, запущенным в качестве дочернего процесса) сохраняются в буфер `buf`. В строке 17 задается вывод содержимого буфера `buf` на экран.

Любая выбранная программная платформа для создания виртуального компьютерного класса должна предоставлять возможность для реализации сценария листинга 3 (рис. 5). Одним из критериев выбора программной платформы может служить эффективность ("легкость") реализации этого сценария.

Для порождения внешних процессов операционной системы и перенаправления их стандартных потоков ввода/вывода в языке программирования Erlang реализован механизм портов (см. листинг 4, рис. 6).

```
01. Port = erlang:open_port(
02.     {spawn_executable, "/bin/bash"}, [
03.     stream, stderr_to_stdout, binary, exit_status,
04.     {args, []}, {cd, "."}
05.     ]),
06. ...
07. erlang:port_command(Port, <<...>>),
08. ...
09. receive
10.     {Port, {data, Data}} -> ...;
11.     _ -> ...
12. end.
```

Рис. 6. Листинг 4. Механизм портов в Erlang

Строки 01—05 содержат пример создания порта. В строке 02 задается имя исполняемого файла, который будет использоваться при создании процесса операционной системы. Строка 03 содержит параметры, предписывающие использовать для взаимодействия с созданным процессом его стандартные потоки ввода/вывода. Строка 04 задает аргументы командной строки (аргументы отсутствуют), с которыми должен создаваться процесс операционной системы, а также задает рабочую директорию этого процесса (текущая директория). Уникальный идентификатор созданного порта сохраняется в переменной `Port`. Для отправки через порт порции данных созданному процессу операционной системы используется функция `port_command` (строка 07). Данные `Data` из стандартного потока вывода процесса операционной системы доставляются в виде сообщения вида `{Port, {data, Data}}`. Erlang-процесс, создавший порт, может перехватывать подобные сообщения при помощи конструкции `receive/end` (строки 09—12).

Супервизор

Рассмотрим схему взаимодействия веб-сервера и супервизора, изображенную на рис. 7. На этой схеме окружностями обозначены Erlang-процессы, а прямоугольником — порт. Тип процесса вписан внутрь окружности. Этапы взаимодействия между процессами пронумерованы.

На стороне веб-сервера обработка каждого веб-сокета осуществляется отдельным Erlang-процессом. На схеме они обозначены как `ws`.

На стороне супервизора выделяются два типа Erlang-процессов — `sup` и `term`. Процесс `term` отвечает за создание порта (запуск и перенаправление ввода/вывода интерпретатора команд `bash`). Один процесс `term` отвечает ровно за один порт. Процесс `sup` отвечает за управление процессами типа `term` (создание и перезапуск, в случае возникновения нештатных ситуаций).

При создании веб-сокета соответствующий процесс типа `ws` отправляет сообщение процессу `sup` (действие *1*). Данное сообщение предписывает запустить новый интерпретатор команд. Процесс `sup` создает процесс `term` (действие *2*), который в свою очередь создает порт (действие *3*). Уникальный идентификатор созданного процесса `term` отсылается процессу `ws`. Идентификатор процесса `ws` сохраняется в списке рассылки рассматриваемого процесса `term`.

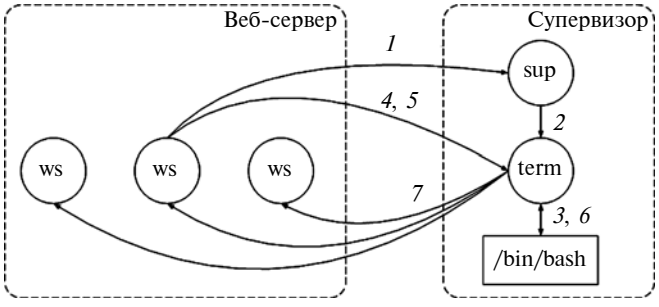


Рис. 7. Схема взаимодействия веб-сервера и супервизора

В дальнейшем любой процесс типа `ws` может добавить/удалить себя из списка рассылки процесса `term` (действие *4*). Команды на выполнение процесс `ws` отсылает (действие *5*) процессу `term`, запустившему соответствующий интерпретатор команд. Процесс `term` самостоятельно отправляет эти команды через порт на входной поток интерпретатора, а также перехватывает готовые порции данных (результаты выполнения команд) со стандартного потока вывода интерпретатора (действие *6*). Далее, процесс `term` отсылает полученные через порт данные всем процессам типа `ws`, присутствующим в его списке рассылки (действие *7*).

Тестирование функциональных возможностей

Для тестирования функциональных возможностей виртуального компьютерного класса использовались три серии тестов. Первая серия состояла из использования простейших команд операционной системы для обхода файловой системы (`cd`, `pwd`), вывода на экран содержимого директорий (`ls`) и текстовых файлов (`cat`, `head`, `tail`). Также осуществлялась компиляция и сборка программ (`gcc`, `make`).

Вторая серия тестов базировалась на запуске и работе с интерактивными программами, не требующими захвата терминального устройства операционной системы. В частности, была проверена возможность работы с отладчиком `gdb`, с интерпретаторами языков программирования Erlang (`erl`), Ocaml (`ocaml`), JavaScript (`node -i`).

Третья серия тестов базировалась на использовании учебной программы — неблокирующего TCP-сервера. Данный сервер может принимать входящие соединения, считывать порции данных из активных сокетов и записывать в них ответ (отправлять ответ клиенту). Ответ строится следующим образом. Если были прочитаны данные `<порция данных>`, то ответом будет последовательность `<префикс><порция данных>`, где в качестве префикса может использоваться строка `Echo`. Префикс может быть изменен. Считанная со стандартного ввода TCP-сервера последовательность символов становится новым префиксом.

Учебный TCP-клиент через аргументы командной строки считывает текст сообщения `Msg` и число `N`. Клиент устанавливает соединение с сервером. В рамках этого соединения клиент `N` раз отправляет сообщение `Msg` на сервер, дожидается ответа на него и печатает этот ответ в свой стандартный поток вывода.

В рамках проведения тестов использовалась веб-страница, на которой эмулировались два терминала. Отдельно (в отдельных веб-браузерах) загружались две подобные страницы. На первой странице запускался TCP-сервер и TCP-клиент, а на второй два

TCP-клиента. В процессе тестирования у TCP-сервера периодически изменялся префикс. Узкое место тестируемой системы оказалось на стороне веб-браузера. Перерисовка содержимого эмулируемого терминала происходила с заметными задержками.

Заключение

В настоящей статье была описана архитектура программной системы (виртуальный компьютерный класс), предназначенной для проведения практических занятий по учебным курсам, связанным с изучением программирования. В частности, была описана архитектура подсистемы виртуального компьютерного класса, позволяющая удаленно запускать командные интерпретаторы (например `bash`) операционной системы и обеспечивающая интерактивное взаимодействие с ними через эмуляцию терминала в веб-интерфейсе. Сознательно не были рассмотрены механизмы виртуального компьютерного класса, связанные с управлением пользовательскими данными, в том числе с редактированием файлов и визуализацией результатов вычислений. Освещение данных вопросов может составить содержание отдельной статьи.

Настоящая статья является естественным продолжением работ автора [1—3]. В этих работах описывалось решение задачи построения системы распределенных вычислений на базе платформы Erlang/OTP [4]. С технической точки зрения было описано решение следующей задачи. На одном из доступных сетевых хостов запускался процесс операционной системы, перехватывался его стандартный вывод, который интерпретировался как результат выполнения вычислительного задания. Описанная в настоящей статье задача также сводится к запуску процесса операционной системы. Отличие состоит в том, что реализуется интерактивное взаимодействие с запущенным процессом через его стандартные потоки ввода/вывода.

Проведенное тестирование виртуального компьютерного класса, в том числе с запуском неблокирующего TCP-сервера и его клиента, показало, что виртуальный компьютерный класс может успешно применяться для проведения практических занятий по дисциплинам, связанным с изучением языков программирования, алгоритмов и структур данных, сетевого программирования.

Список литературы

1. **Шундеев А. С.** Система распределенных вычислений на базе платформы Erlang/OTP // Программная инженерия. 2014. № 5. С. 3—5.

2. **Шундеев А. С.** Система распределенной обработки данных на базе платформы Erlang/OTP // Материалы III Научно-практической конференции "Актуальные проблемы системной и программной инженерии". Сб. науч. трудов. М.: Изд-во МЭСИ, 2013. С. 188—191.

3. **Шундеев А. С., Пучков Ф. М., Кривчиков М. А.** Организация распределенных вычислений на базе платформы ERLANG/OTP // Ломоносовские чтения. Тезисы докладов науч. конф. Секция механики. 15—23 апреля 2013 г., Москва, МГУ имени М. В. Ломоносова. М.: Изд-во Московского университета, 2013. С. 150.

4. **Laurent S.** Introducing Erlang. O'Reilly Media, 2013. 204 p.

5. **Fielding R. T.** Architectural Styles and the Design of Network-based Software Architectures. Dissertation doctor of philosophy in Information and Computer Science. University of California, 2000.

6. **Васенин В. А., Шундеев А. С.** Развитие веб-технологий и промежуточного программного обеспечения // Информационные технологии 2011. № 12. С. 2—9.

7. **Ferguson R., Heilmann C.** Beginning JavaScript with DOM Scripting and Ajax: Apress, 2013. 388 p.

8. **Pterneas V.** Getting Started with HTML5 WebSocket Programming. Packt Publishing, 2013. 110 p.

9. **Cowboy** User Guide. URL: http://ninenines.eu/docs/en/cowboy/HEAD/guide

10. **JQuery** Terminal Emulator. URL: http://terminal.jcubic.pl/

11. **Kerrisk M.** The Linux programming interface: a Linux and UNIX system programming handbook. No Starch Press, 2010. 1552 p.

ИНФОРМАЦИЯ

23—27 сентября 2014 г. на базе Севастопольского национального технического университета (СевНТУ) состоится Международный научно-технический семинар

"Современные проблемы прикладной математики, информатики, связи, автоматизации и управления"

Организаторы семинара:

Севастопольский национальный технический университет (г. Севастополь, Россия);
Институт проблем информатики РАН (г. Москва, Россия).

Принимаются доклады, посвященные перспективным комплексным направлениям исследований в области прикладной математики, информатики, связи, автоматизации и управления.

Контакты:
Тел.: +38-0692-43-51-30
E-mail: iggurevich@gmail.com
tk.sevntu@gmail.com

УДК 004.921

А. М. Гиацинтов, мл. науч. сотр., e-mail: algts@inbox.ru,
К. А. Мамросенко, канд. техн. наук, зав. отд. СИТ, e-mail: kirillam@ya.ru,
Центр визуализации и спутниковых информационных технологий,
НИИ системных исследований РАН, г. Москва

Воспроизведение потоковых видеоматериалов в подсистеме визуализации тренажерно-обучающей системы

Предложена архитектура подсистемы воспроизведения потоковых видеоматериалов в виртуальной трехмерной сцене в тренажерно-обучающих системах. Рассмотрены принципы работы технологии потокового мультимедиа, определены ее положительные и отрицательные аспекты применительно к воспроизведению видеоматериалов. Предложен новый метод одновременного воспроизведения видеоматериалов высокой четкости в подсистеме визуализации тренажерно-обучающей системы. Приведен алгоритм работы декодера потоковых видеоматериалов.

Ключевые слова: *система визуализации, тренажерно-обучающая система, рендеринг, разнородные видеоматериалы, декодер видео*

A. M. Giatsintov, K. A. Mamrosenko

Playback of Streaming Video in Visualization Subsystem of Training Simulation System

Article describes subsystem's architecture for playback of streaming video in virtual 3d scene of training simulation systems. Operation principles of media streaming technology, its pros and cons related to streaming video playback are reviewed. A new method for playback of several high definition videos in visualization subsystem of training simulation system is presented. Streaming video decoder algorithm is also presented.

Keywords: *visualization system, training simulation system, rendering, heterogeneous video, video decoder*

Введение

Под тренажерно-обучающей системой (ТОС) оператора сложной технической системы (СТС) будем понимать техническое средство для подготовки операторов в едином информационном окружении, отвечающее требованиям методик подготовки, создающее условия для получения знаний, навыков и умений, реализующее модель таких систем и обеспечивающее контроль над действиями обучаемого, а также для исследований. Кроме формирования индивидуальных профессиональных навыков и умений, ТОС можно применять для отработки групповых операций. Возможно использование ТОС при подготовке специалистов в широком спектре проблемно-ориентированных областей [1].

Подсистема визуализации обеспечивает отображение результатов моделирования внешней среды и объекта управления с помощью устройств отображения информации. Отображение разнородных видеоматериалов в виртуальной трехмерной сцене является одним из требований к ТОС.

Особый интерес представляют следующие виды аудиовизуальной учебной информации:

- динамические графики развития реальных процессов;
- диаграммы, гистограммы для анализа массивов данных;

- иллюстративные материалы изучаемых объектов;
- мнемосхемы;
- трехмерные модели объектов, их частей;
- функциональные схемы взаимодействия отдельных подсистем, а также обобщенные схемы работы изучаемой системы в целом;
- результаты работы моделирующих комплексов в форме видео-образов с сохранением управляемости приложения;
- видеоматериалы реальных объектов.

Виртуальное трехмерное окружение является контейнером, в котором происходит объединение разнородной информации. Одним из применений видеоматериалов в ТОС является отображение графического виртуального образа инструктора, получаемого в реальном масштабе времени с видеокамеры. Кроме того, одновременное воспроизведение видеоматериалов может использоваться для отображения результатов моделирования нескольких параллельно выполняемых процессов.

Воспроизведение видеоматериалов внутри виртуальной трехмерной сцены является сложной задачей, так как необходимо учитывать факторы, не актуальные при воспроизведении в медиаплеерах. Подсистема визуализации должна обеспечивать отображение трехмерной сцены с приемлемой частотой кадров (не менее 25 кадров в секунду) и должна быть способна при этом реагировать на внешние воздействия, в том числе на изменения параметров трехмерной сцены или загрузку дополнительных объектов. Для отображения нескольких видеоматериалов высокой четкости в виртуальной трехмерной сцене была разработана и внедрена архитектура декодера кадров видеоматериалов. Также были созданы методы эффективной передачи данных в видеокарту для обеспечения высокой производительности подсистемы визуализации при отображении видеоматериалов высокой четкости.

Метод отображения нескольких видеоматериалов высокой четкости в подсистеме визуализации ТОС

Разработанная архитектура графической подсистемы позволяет декодировать и отображать одновременно несколько видеоматериалов высокой четкости в трехмерной сцене. Состав архитектуры включает: декодер видеоматериалов, в котором происходит декодирование видео- и аудиопакетов; подсистему воспроизведения декодированного звука; управляющую структуру, необходимую для запуска видеоматериалов, паузы воспроизведения, выставления громкости и т. д.; интерфейс взаимодействия с графическим модулем, необходимый для обновления видеокадров; интерфейс управления графической подсистемой [2].

Использование библиотеки libavcodec в декодере позволяет декодировать практически любые разнородные видео- и аудиоматериалы в таких популярных форматах, как AVI, Mpeg 2, DIVX, H264, MP3, OGG, AC3 и т. п.

В большинстве видеофайлов число аудиопакетов значительно меньше, чем число видеопакетов — приблизительно 75 % видеопакетов и 25 % аудиопакетов. В зависимости от типа контейнера и используемого кодера расположение пакетов внутри контейнера может быть разным. Зачастую пакеты расположены в виде последовательности очередей пакетов разного типа, например, около 20 видеопакетов, расположенных подряд, затем несколько аудиопакетов, после которых снова подряд расположены около 20 видеопакетов.

Процесс декодирования является ресурсоемким. Например, декодирование одного видеопакета для видеоматериала с разрешением 1280 × 720 (720 p) на процессоре Intel Core 2 Quad 2,66 ГГц занимает от 5...10 мс, для видеоматериала с разрешением 1920 × 1080 (1080 p) — 10...20 мс. Декодирование пакета звуковых данных занимает значительно меньше времени (0,5...2 мс), но если одновременно декодируется звук для нескольких воспроизводимых видеоматериалов, это также может серьезно снизить производительность. Скорость получения пакетов из видеофайла ограничивается производительностью системы хранения данных (СХД), что в свою очередь влияет на производительность подсистемы визуализации. Задержки образуются, если в момент считывания пакетов из видеофайла СХД обрабатывала другие команды, либо файл являлся фрагментированным, что заставило бы СХД искать фрагменты данного файла. Для преодоления трудностей с производительностью была использована многопоточность.

Для оптимизации процесса декодирования видеоматериалов используют отдельные потоки для сбора пакетов, декодирования видео- и аудиопакетов. Процесс воспроизведения звука также происходит в дополнительном потоке.

Обновление видеотекстуры в подсистеме визуализации происходит в основном потоке программы (потоке рендеринга). Каждый кадр подсистемы визуализации происходит проверка — следует ли обновлять видеотекстуру в трехмерной сцене или нет. Если обновление требуется, то видеокадр берется из очереди декодированных кадров[1] и помещается в память видеокарты в качестве текстуры, которая может быть наложена на поверхность любого трехмерного объекта.

Также для повышения производительности используют два так называемых пиксельных буфера. Пиксельные буферы применяют для хранения пиксельных данных (текстуры) в видеопамяти, они позволяют значительно снизить временные затраты на обновление видеотекстуры в подсистеме визуализации.

Метод, который позволил получить оптимальную производительность при воспроизведении несколь-

[1] Декодированный видеокадр — изображение без сжатия, которое может быть использовано подсистемой визуализации для обновления текстур в генерируемом виртуальном трехмерном окружении.

ких видеоматериалов высокой четкости, основан на применении приоритетов к обновлению воспроизводимых видеофайлов.

Для каждого видеоматериала применяется весовой коэффициент, отражающий общий приоритет видеоматериала и весовой коэффициент выполняемого действия. Общий приоритет видеоматериала основывается на разрешении видеоматериала — чем ниже разрешение, тем выше приоритет, так как видеоматериал меньшего размера обновляется быстрее. Существуют три общих приоритета видеоматериала — высокий (8), средний (4), низкий (0). Для видеоматериала с разрешением до 720 × 576 выставляется высокий приоритет, с разрешением до 1280 × 720 выставляется средний приоритет, для видеоматериала с разрешением до 1920 × 1080 выставляется низкий приоритет. Весовой коэффициент выполняемой операции меняется каждый раз при обновлении видеоматериала в зависимости от действия, которое необходимо будет выполнить при следующей итерации цикла (ИЦ) подсистемы визуализации. Выполняемые операции имеют следующие весовые коэффициенты (обновление видеотекстуры имеет наибольший коэффициент): обновление видеотекстуры (3), обновление видеотекстуры и пиксельного буфера (2), обновление пиксельного буфера (1), исключение видеоматериала (0). Операция "исключение видеоматериала" выполняется в случае, когда на следующей ИЦ подсистемы визуализации нет необходимости обновлять видеотекстуру, а также все пиксельные буферы заполнены. Видеоматериал исключается из списка обновляемых, однако весовой коэффициент выполняемого действия меняется на высший для того, чтобы состояние видеоматериала было проверено на следующей ИЦ подсистемы визуализации.

Список обновляемых видеоматериалов определяется на каждой ИЦ подсистемы визуализации. На основе общего приоритета видеоматериала и весового коэффициента выполняемого действия определяется позиция видеоматериала в списке обновления.

Алгоритм обновления видеотекстуры в подсистеме визуализации был также изменен для повышения производительности. Теперь первым осуществляется обновление видеотекстуры, а не обновление пиксельного буфера. Так как операция обновления текстуры из пиксельного буфера является асинхронной, то она выполняется очень быстро, в то время как операция обновления информации в пиксельном буфере может вызвать синхронизацию процессора и видеокарты, что приведет к задержкам. Также видеокарте после обновления пиксельного буфера может понадобиться некоторое время для обработки полученных данных. Поэтому при обновлении текстуры сразу после обновления пиксельного буфера может появиться задержка. Изменение алгоритма обновления видеотекстуры дает больше времени на обработку данных пиксельного буфера. После обновления пиксельного буфера одновременно с обработкой данных в пиксельном буфере видеокартой происходит расчет весового коэффициента действия для следующей ИЦ подсистемы визуализации и переход к обновлению следующей видеотекстуры.

При традиционном подходе существует возможность воспроизведения одного видеоматериала с разрешением 1920 × 1080 (FullHD) или 1280 × 720 (HD), либо нескольких видеоматериалов с разрешением не выше 720 × 576 (576p, SD) при сохранении частоты генерации кадров сцены не ниже 60 кадров в секунду. При одновременном воспроизведении нескольких видеоматериалов высокой четкости наблюдается значительное увеличение времени генерации кадра подсистемой визуализации — число кадров в секунду снижается до 10...15. Опыт проводился на оборудовании Intel Core 2 Quad 2.66 GHz, Nvidia Geforce 250, Microsoft Windows XP.

Применение описанного выше алгоритма позволяет одновременно воспроизвести три видеоматериала HD качества (1280 × 720) или два видеоматериала в формате FullHD (1920 × 1080) в виртуальной трехмерной сцене при сохранении частоты генерации кадров сцены не ниже 60 кадров в секунду.

При наличии звука в видеоматериале применяется метод синхронизации видеоизображения по звуку, т. е. скорость воспроизведения звука считается постоянной величиной (определяется на основании информации о частоте дискретизации и числе каналов, указанной в видеофайле), а время отображения видеокадра рассчитывается на основании времени, записанного в пакетах, и времени, прошедшего с начала воспроизведения звука. В разработанном алгоритме скорость воспроизведения звука считается величиной постоянной, а время вывода видеотекстуры на экран рассчитывается с учетом производительности (кадров в секунду) подсистемы визуализации. Если звук не присутствует в видеоматериале, то время вывода видеотекстуры на экран рассчитывается только на основе количества кадров в секунду подсистемы визуализации.

Можно выделить два основных применения возможности отображения нескольких видеоматериалов в ТОС: отображение виртуального графического образа инструктора и разнородных видеоматериалов, отображение результатов моделирования единовременно выполняющихся процессов. Для первого случая задача синхронизации нескольких видеоматериалов не актуальна. Для второго случая отображение видеоматериалов должно начаться одновременно в целях обеспечения синхронизации. На данный момент система позволяет осуществить одновременный запуск нескольких видеоматериалов с погрешностью приблизительно 5 мс, что является приемлемым результатом.

Архитектура подсистемы отображения потоковых видеоматериалов в подсистеме визуализации ТОС

Воспроизведение потоковых видеоматериалов отличается от воспроизведения видеоматериалов, хранящихся в виде файлов. Основной задачей при воспроизведении потоковых видеоматериалов является достижение наименьшей задержки между получением информации с внешнего устройства и отображением уже преобразованной информации. При воспроизведении видеоматериалов из файлов для обеспечения плавности воспроизведения применяется технология кэширования данных в оперативной памяти. Однако использование кэширования в значительной степени ограничено либо вовсе неприменимо при воспроизведении потоковых видеоматериалов в ТОС. При кэшировании данных образуется дополнительная задержка между получением данных с внешнего устройства и их отображением. Например, при воспроизведении потоковых видеоматериалов в формате HDV задержка воспроизведения может составлять более секунды даже без использования кэширования. При кэшировании большого количества данных такая задержка будет увеличиваться, что неприемлемо для ТОС. Например, по требованиям ICAO к тренажерным системам, задержка отображения информации с момента подачи управляющего воздействия не должна превышать 100 мс [3]. Следовательно, использование технологии кэширования при воспроизведении потоковых видеоматериалов в ТОС следует минимизировать.

Поток данных с внешнего устройства (с видеокамеры) представляет собой поступающие с равной периодичностью массивы данных одинакового размера. В зависимости от формата передаваемых данных число массивов данных в секунду и их размер могут различаться. Авторами было определено, что, например, для видеоформата DV число передаваемых массивов данных равно частоте кадров в секунду, соответственно, при 25 кадрах в секунду, массивы данных с камеры поступают каждые 1/25 кадров = 40 мс. Для формата Mpeg 2, применяемом во многих камерах, поддерживающих видео высокой четкости, число передаваемых массивов данных может быть более 80, так как технология Mpeg 2 позволяет передавать не целый кадр, а только его изменения по сравнению с предыдущим. Также, для получения целого видеокадра может потребоваться несколько массивов данных (пакетов).

Зачастую поток данных, приходящий с видеокамеры, совмещает в себе аудио- и видеоинформацию. Такой поток называется мультиплексированным. Как формат DV, так и формат Mpeg 2 передают свои данные мультиплексированными. Для их воспроизведения сначала необходимо демультиплексировать поступающую информацию, т. е. провести разделение видео- и аудиоданных на отдельные пакеты.

Основной трудностью получения пакетов с видеокамеры является отсутствие единого стандартного интерфейса для работы с внешними устройствами. Так, на операционных системах семейства Windows для работы с внешними мультимедийными устройствами применяется интерфейс DirectShow [4], на операционных системах семейства Linux — Video For Linux 2 (v4l2), на операционных системах семейства Macintosh (MacOS) — QuickTime. Каждый из этих интерфейсов имеет собственную структуру и в значительной степени отличается от других интерфейсов.

Разработанная архитектура подсистемы воспроизведения потоковых видеоматериалов позволяет осуществлять захват данных с внешних устройств на операционных системах Windows, Linux, MacOS X. Архитектуру можно условно разделить на две части: подсистема захвата данных и подсистема декодирования данных.

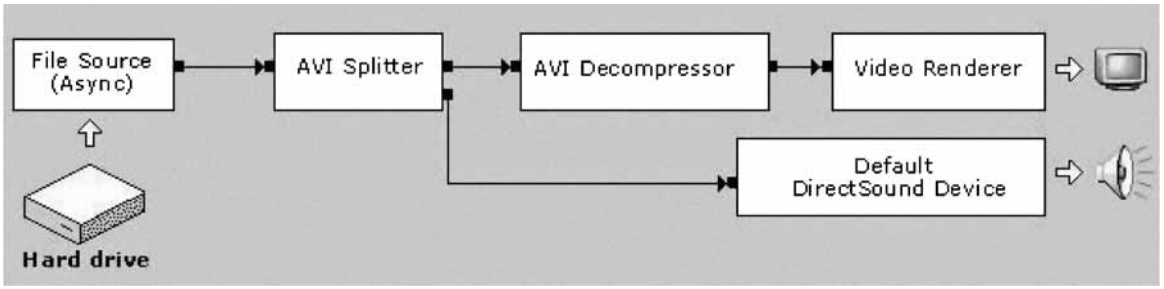
Подсистема захвата данных использует интерфейсы платформы для захвата данных с таких устройств, как видеокамеры. Подсистема декодирования основана на библиотеке FFmpeg и позволяет осуществлять декодирование большинства современных форматов.

Преимуществом такого разделения функций является неизменность подсистемы декодирования для всех платформ, изменяется лишь подсистема захвата данных. Кроме того, использование библиотеки FFmpeg позволяет добиться независимости от декодеров, установленных на машине пользователя.

Рассмотрим реализацию архитектуры на платформе Windows XP и API DirectShow. Основным объектом в DirectShow является фильтр. Существуют три вида фильтров: фильтры захвата (получения данных), фильтры преобразования и фильтры визуализации (рендеринга) [5].

Для работы с фильтрами DirectShow создается граф фильтров, на который добавляются все необходимые фильтры захвата, преобразования и визуализации (см. пример на рисунке).

Вместе с DirectShow поставляется множество стандартных фильтров для получения данных с внешних источников, преобразования этих данных и их отображения. Например, существуют отдельные фильтры захвата данных как для видеокамер, которые выводят видеоинформацию в формате DV, так и для видеокамер, выводящих информацию в формате Mpeg 2. Однако среди множества представленных стандартных фильтров отсутствует фильтр, который бы позволял получить неизмененные данные с видеокамеры для их дальнейшего преобразования внутри приложения. Было определено, что многие приложения, такие как Media Player Classic, FFmpeg, использующие DirectShow, не способны получать данные с видеокамер формата HDV. Программы, способные получать данные с камер в формате HDV, такие как VLC Player, используют собственные DirectShow-фильтры. В силу этого было принято решение разработать собственный фильтр получения данных с видеокамеры.



Пример использования графа фильтров для воспроизведения видеофайла

Задачей разработанного фильтра является получение и хранение определенного числа пакетов, поступивших с видеокамеры. Фильтр подсоединяется к выходу фильтра захвата видеокамеры и принимает пакеты данных через определенные интервалы времени (зависит от выходного формата видеоинформации).

Разработанный фильтр состоит из трех основных частей: вход фильтра, который принимает данные с видеокамеры; интерфейс фильтра, позволяющий настраивать работу фильтра; выход фильтра, с которого происходит получение пакетов, полученных с камеры, приложением. Фильтр способен хранить до 20 поступивших с камеры пакетов. При превышении этого порога наиболее старый пакет удаляется из очереди.

Получение данных с видеокамеры и определение параметров входного потока состоят из нескольких этапов. Первым этапом является инициализация COM-модели и поиск внешних устройств захвата. Такими устройствами могут являться видеокамеры, платы видеозахвата или устройства захвата звука (например, звуковые карты). Если хотя бы одно устройство было найдено, то происходит переход ко второму этапу. В первую очередь осуществляется поиск видеокамер и плат видеозахвата.

Второй этап состоит в определении типа выходного устройства, определении наличия и типа выходов фильтра захвата данного устройства, а также определении формата выходного сигнала с данного фильтра. Не для каждого типа выходного сигнала возможно определение параметров выходного сигнала, таких как используемая цветовая модель, высота и ширина изображения и т. д. Например, параметры сигнала, поступающего с видеокамеры в формате DV, могут быть определены сразу же после определения выходов фильтра захвата. Однако для формата Mpeg 2 это невозможно, так как информация о параметрах выходного сигнала хранится не в каждом пакете, а только в пакетах, предшествующих началу группы изображений (*Group of Pictures*), в так называемых заголовках начала последовательности (*sequence header*). Для получения информации о параметрах выходного сигнала необходимо декодировать несколько пакетов. При обнаружении выходного сигнала в формате Mpeg 2 потоку данных присваиваются параметры по умолчанию, а конкретные параметры определяются на следующих этапах.

Третьим этапом является создание экземпляра разработанного фильтра для получения данных с видеокамеры и присоединение его входа к выходу фильтра захвата видеокамеры. После выполнения третьего этапа возможно получение пакетов данных с видеокамеры.

Четвертым этапом является определение параметров входного сигнала и выставление параметров подсистемы декодирования FFmpeg. Для определения параметров входного сигнала происходит получение нескольких пакетов с видеокамеры. Авторами было установлено, что в зависимости от формата входного сигнала требуется различное число пакетов. Например, для формата DV достаточно всего одного пакета для определения всех необходимых параметров декодирования подсистемы FFmpeg. Для формата Mpeg 2 требуется около 130...140 пакетов для корректного определения параметров входного сигнала. Во время определения параметров обрабатываемые пакеты автоматически демультиплексируются и сохраняются во внутренних буферах FFmpeg. После определения параметров входного сигнала все использованные пакеты удаляются из очереди хранения и внутренних буферов FFmpeg чтобы исключить повреждение отображаемого при воспроизведении изображения вследствие несовпадения изображений, хранимых в старых и новых пакетах.

По завершению четвертого этапа видеоматериалы, полученные с видеокамеры, могут быть воспроизведены в виртуальной трехмерной сцене.

Декодер потоковых видеоматериалов

Рассмотрим подробнее структуру декодера поступающих с видеокамеры данных и его алгоритм работы.

Декодер включает в себя три независимых процесса:

- получение пакетов с видеокамеры, их демультиплексирование и сортировку видео- и аудиопакетов по соответствующим очередям;
- декодирование видеопакетов;
- декодирование аудиопакетов.

Каждый из этих процессов работает в собственном потоке. Использование многопоточности позволяет

значительно повысить производительность подсистемы декодирования при использовании многоядерного процессора.

После команды запуска воспроизведения видеоматериалов, поступающих с видеокамеры, создается три независимых потока и запускается фильтр получения пакетов с видеокамеры. В потоке получения пакетов с видеокамеры после запуска происходит кэширование определенного числа пакетов. Количество кэшированных пакетов зависит от формата входного сигнала. Например, для формата DV кэшируется до четырех пакетов, для формата Mpeg 2 — до десяти. После окончания кэширования заданного числа пакетов запускается рабочий цикл потока, в котором последовательно происходит получение пакетов с видеокамеры, их демультиплексирование и сортировка полученных видео- и аудиопакетов[2] по соответствующим очередям. В этом же рабочем цикле происходит проверка на необходимость кэширования декодированных аудио- и видеоданных. Если воспроизведение видеоматериалов было только что запущено, то при каждой итерации рабочего цикла потока происходит проверка на количество декодированных видеокадров и аудиоданных. Если число декодированных видеокадров меньше установленного порога, то отправляется запрос на декодирование потоку декодирования видеопакетов. Аналогично проверяется количество декодированных аудиоданных.

Потоки декодирования работают в непрерывном режиме до тех пор, пока не превысят заданный порог. Например, поток декодирования видеопакетов работает непрерывно до тех пор, пока не будет декодировано два видеокадра. После этого выполнение потока останавливается до тех пор, пока от управляющего приложения не придет сигнал о необходимости декодирования следующего видеокадра.

Существует возможность ускорения процесса декодирования поступающих потоковых данных благодаря использованию вычислений на видеокарте (GPGPU). Данная возможность способна значительно повысить скорость декодирования, однако у нее есть недостатки. Во-первых, происходит снижение качества изображения [6]. Во-вторых, декодирование данных с использованием GPGPU даст дополнительную нагрузку на видеокарту, которая уже используется для визуализации виртуальной трехмерной сцены. Если применяемая трехмерная сцена является "тяжелой" для визуализации, то видеокарта может не успеть своевременно декодировать поступающие данные, что может привести к рассинхронизации видео- и аудиоданных, задержкам и рывкам изображения. Несмотря на перечисленные ограничения, при необходимости декодирования нескольких потоковых видеоматериалов с разрешением 3840 × 2160 и выше, использование мощностей видеокарты является оправданным.

Первоначально планировалось использовать стандартные фильтры, поставляемые вместе с ОС Windows, для демультиплексирования сигнала, поступающего с видеокамеры. Однако у данного подхода были выявлены существенные недостатки.

Во-первых, для каждого выходного формата требуется свой фильтр демультиплексирования. Тестирование всех возможных выходных форматов является трудоемкой задачей и требует значительных финансовых затрат на приобретение оборудования для тестирования. Вследствие этого существует риск, что разработчик не сможет определить и предоставить вместе со своим приложением все необходимые фильтры, что повлечет за собой невозможность воспроизведения потоковых видеоматериалов на компьютере пользователя при определенной конфигурации оборудования.

Во-вторых, для настройки работы фильтров демультиплексирования могут потребоваться дополнительные фильтры. Например, для фильтра-демультиплексора формата Mpeg 2 требуется дополнительный PSI-фильтр, который по входящему на фильтр-демультиплексор сигналу определяет наличие видео- и аудиосигналов и создает соответствующие выходы у фильтра-демультиплексора. Данный PSI-фильтр не поставляется вместе с ОС Windows и должен быть включен в комплект поставки приложения.

В-третьих, при использовании фильтров демультиплексирования могут возникнуть существенные трудности с определением параметров входящего сигнала. Например, для успешного декодирования входного сигнала в формате Mpeg 2 необходимо инициализировать подсистему декодирования FFmpeg с корректными параметрами. Для того чтобы получить данные из входного сигнала, необходимо в каждом поступающем с демультиплексора видеопакете проверять наличие так называемого заголовка начала последовательности (*sequence header*). В данном заголовке хранятся данные о высоте и ширине изображения, о формате видеокадра (независимо сжатые кадры (I-кадры), кадры, сжатые с использованием предсказания движения в одном направлении (P-кадры), и кадры, сжатые с использованием предсказания движения в двух направлениях (B-кадры)), скорость передачи данных и т. д. Кроме того, специфика формата Mpeg 2 состоит в том, что в отличие от формата Mpeg 1, где все данные о входном потоке хранились только в заголовках начала последовательности, в формате Mpeg 2 присутствует механизм расширений, который позволяет хранить в заголовке начала последовательности только часть необходимой информации. Другая часть хранится в так называемом расширении заголовка начала последовательности (*sequence extension*). Для получения корректных данных о входном сигнале необходимо вручную, побитно (в форматах Mpeg 1 и Mpeg 2 многие данные хранятся в нестандартных типах данных

[2] Здесь видеопакет — пакет, содержащий видеоинформацию, представленный в формате, используемом подсистемой декодирования FFmpeg; аудиопакет — пакет, содержащий звуковую информацию, представленный в формате, используемом подсистемой декодирования FFmpeg.

для экономии места, например, переменная, определяющая высоту изображения, занимает 12 бит) декодировать входящие видеопакеты, объединить данные, полученные из основного заголовка и его расширения, и только затем применить полученные данные для инициализации подсистемы FFmpeg [7].

Закономерным является вопрос о необходимости использования подсистемы FFmpeg вместе с DirectShow, когда интерфейс DirectShow предоставляет все необходимые возможности для декодирования потоковых данных без использования FFmpeg. Во-первых, подсистема FFmpeg уже была успешно внедрена в подсистему визуализации и используется для воспроизведения видеоматериалов из файлов в виртуальной трехмерной сцене. Во-вторых, многие фильтры DirectShow, такие как FFDShow, используют для декодирования видеоматериалов подсистему FFmpeg. Соответственно, использование таких фильтров, как FFDShow дублировало бы уже внедренную и испытанную подсистему декодирования. В-третьих, использование только интерфейса DirectShow для декодирования потоковых видеоданных не избавило бы от задачи подбора и тестирования фильтров, описанной выше.

Заключение

Разработан метод одновременного воспроизведения видеоматериалов в синтезированном трехмерном окружении. В отличие от традиционных подходов, позволяющих отображать только один видеоматериал в формате FullHD (с разрешением 1920 × 1080), созданный метод дает возможность отобразить два и более видеоматериала в формате FullHD при сохранении частоты генерации кадров подсистемы визуализации не ниже 60 кадров в секунду. Данный метод может быть использован, например, для одновременного отображения виртуального графического образа инструктора и разнородных видеоматериалов в ТОС.

Разработана архитектура подсистемы воспроизведения потоковых видеоматериалов, получаемых с таких источников, как видеокамеры и платы захвата. Основными преимуществами предложенной архитектуры являются: возможность получать данные с внешних устройств с минимальной задержкой — до двух видеокадров против типового значения 8...10 видеокадров; работа на платформах Windows, Linux, MacOS X; большая часть функций остается неизменной на всех платформах, меняются только зависящие от платформы функции захвата данных; независимость от кодеков, установленных на машине пользователя.

Определены ограничения, связанные с декодированием потоковых материалов в различных форматах, таких как Mpeg 2 и DV. Например, для формата DV достаточно всего одного пакета для определения всех необходимых параметров декодирования подсистемы FFmpeg. Для формата Mpeg 2 требуется около 130...140 пакетов для корректного определения параметров входного сигнала. Кроме того, для формата Mpeg 2 приходится кэшировать больше пакетов, что связано с возможностью данного формата передавать изменения в ранее полученном изображении, а не целое изображение, в виде нескольких отдельных пакетов.

Использование описанных выше методов и алгоритмов позволяет воспроизводить потоковые видеоматериалы практически любого формата в подсистеме визуализации ТОС. Тестирование видеоформатов, не рассмотренных в данной статье, может быть проведено для поиска оптимального количества кэшируемых пакетов, приходящих с камеры, в целях уменьшения задержки перед отображением декодированной информации.

Список литературы

1. **Mamrosenko K. A.** Training simulation systems for open distance learning // Proceedings Conference Open Distance Learning Towards Building Sustainable Global Learning Communities. Hanoi, Vietnam: The Gioi, 2010. P. 509—515.

2. **Гиацинтов А. М.** Отображение разнородных видеоматериалов на гранях трехмерных объектов в подсистеме визуализации тренажерных обучающих систем // Программные продукты и системы. 2012. № 3. С. 80—86.

3. **International** Civil Aviation Organization. Doc 9625-AN/938 Manual of Criteria for the Qualification of Flight Simulation Training Devices. Montreal, Canada: International Civil Aviation Organization, 2009.

4. **DirectShow** [Электронный ресурс]. URL: http://msdn.microsoft.com/en-us/library/windows/desktop/dd375454(v = vs.85).aspx (дата обращения: 13.01.2012).

5. **Вовк В.** Что такое DirectShow? Что такое фильтр? — DirectShow по-русски [Электронный ресурс]. URL: http://directshow.wonderu.com/статьи/первые-шаги-с-directshow/что-такое-directshow-что-такое-фильтр (дата обращения: 13.12.2011).

6. **Сжатие** видео и декодирование | чем и на чем лучше | Intel Quick Sync, Nvidia CUDA, AMD APP — THG.RU [Электронный ресурс]. URL: http://www.thg.ru/video/video_transcoding_amd_nvidia_intel/print.html (дата обращения: 27.03.2014).

7. **Using** the MPEG-2 Demultiplexer [Электронный ресурс]. URL: http://msdn.microsoft.com/en-us/library/windows/desktop/dd407285(v = vs.85).aspx (дата обращения: 13.01.2012).

УДК 519.688, 669.14.242

Н. Н. Светушков, канд. техн. наук, ст. науч. сотр., Московский авиационный институт (национальный исследовательский университет),
e-mail: svt.n.n@mail.ru

Модель объединенного кластера в трехмерной графике

Описана разработка новых принципов создания сложных графических объектов, включая двумерные и трехмерные. Введено понятие элементарного кластера, который может рассматриваться как обобщение классических геометрических фигур — окружности, эллипса и прямоугольника, сферы и параллелепипеда. Описана специализированная программная среда и продемонстрированы созданные с ее помощью геометрические объекты, основанные на модели объединенного кластера. Приведен анализ возможностей программного продукта и сделан вывод, что создание объектов на основе кластерной модели позволяет не только уменьшить объем хранимой информации, но и в значительной степени упросить интерфейс программных средств, используемых для трехмерного моделирования. Разработанный программный комплекс может быть использован как для решения задач моделирования различных технологических процессов, так и для прототипирования, в том числе и в авиационной промышленности. Он позволяет простыми средствами создавать сложные геометрические объекты неискушенному в трехмерном моделировании пользователю.

Ключевые слова: *кластерная модель, трехмерное моделирование, программное обеспечение, визуализация, интерфейсы*

N. N. Svetushkov

Joint Cluster Model in Three-Dimensional Graphics

Article is devoted to the development of new principles for creating complex graphic objects, including two- and three-dimensional. Introduced the concept of an elementary cluster which can be regarded as a generalization of the classical geometric shapes — circle, ellipse and rectangle, sphere, parallelepiped. Software environment created demonstrates geometric objects, developed on the model of the joint cluster. There is an analysis of the opportunities of created software, and it is concluded that the principles of object creation based on the cluster model afford not only reduce the amount of stored information, but also to a large extent to exhort interface software used for three-dimensional modeling. Developed software package can be used to solve the problems of modeling various processes and prototyping, including the aviation industry, allowing simple means to create complex geometric objects in three-dimensional modeling of an inexperienced user.

Keywords: *cluster model, three-dimensional modeling software, visualization, interfaces*

Введение

Методы визуализации данных и возможности представления информации в виде, наиболее удобном для восприятия, тесно связаны с задачами трехмерного моделирования. Программы и технологии трехмерного моделирования широко применяют и во многих областях практической деятельности, включая производство, строительство, проектирование интерьеров, а также в масс-медиа культуре. В качестве преимуществ трехмерного моделирования перед другими способами визуализации можно отметить следующие: трехмерное моделирование дает достаточно точную модель, максимально приближенную к реальности, при этом современные программы трехмерного моделирования помогают достичь высокой детализации таких моделей, значительно увеличивая наглядность проекта. Трехмерная визуализация дает возможность тщательно проработать и просмотреть все детали. Такой способ визуализации представляется более естественным для целевой аудитории. В последнее время появилась возможность "распечатать" практически любую трехмерную модель на 3D-принтере, что еще более подчеркивает актуальность и реалистичность создаваемых трехмерных образов.

Существует довольно большое число программ для трехмерного моделирования, включая 3DS Max Design, AutoCad, Maya, T-flex, Компас-3D и др. Компания Google выпустили 3D-редактор с упрощенным интерфейсом пользователя — Google SketchUp. Эти программы позволяют создавать и компоновать различные объекты, задавать траектории перемещений и в конечном итоге создавать видео с участием трехмерных моделей. Однако такая работа требует у специалиста серьезных навыков. Распространенные программы имеют довольно сложный интерфейс и при начальном изучении представляют значительные трудности (для ознакомления с принципами их работы часто используют специальные обучающие программные средства).

С широким распространением различного рода 3D-принтеров как в промышленности, так и в среде обычных пользователей возникает задача разработки простого и понятного широкому кругу пользователей программного средства, которое может быть использовано для трехмерного прототипирования, в том числе, например, и в авиационной промышленности.

Цели и подходы

Целью работы, описанной в статье, было исследование возможностей новых принципов проектирования сложных трехмерных изделий в целях упрощения интерфейса программного обеспечения и создания интуитивно понятных средств разработки, которые доступны пользователю, не имеющему опыта в трехмерном моделировании.

В известных программных средствах для создания трехмерных объектов используется самый очевидный подход — задание пользователем положения точек поверхности модели. Такой принцип неявно подразумевает, что внутренняя часть объекта ничем не заполнена, т. е. о ней отсутствует какая-либо информация. Для технического моделирования реальных изделий или при визуализации, например, температурных полей в процессе термообработки, необходимо иметь возможность задавать теплофизические или другие характеристики объекта во внутренней части ограниченной поверхностью области. Таким образом, возникает задача описания не только внешней поверхности или границы изделия, но и его внутренней части, которая, к тому же, может иметь сложную структуру.

Если провести аналогию с геометрией, то для задания, например, круга нет необходимости отдельно определять координату каждой из точек окружности — это можно сделать по известному правилу. Область круга определяется положением его центра и алгоритмом расчета точек, являющихся граничными (в построении используется задаваемый параметр — радиус круга). Аналогичные принципы лежат и в основе построения областей, ограниченных квадратом или прямоугольником. Таким образом, если ввести некоторое обобщение понятия классических геометрических фигур — кластерный элемент, то отпадает необходимость в явном задании всех граничных точек или поверхности модели. В основе разрабатываемого подхода лежит принцип, при котором пользователь задает не координаты поверхности, а определяет принадлежность выбранной точки (кластерного элемента) формируемому объекту. Структура или описание кластерного элемента должны давать возможность алгоритмического построения окружающей его области и, соответственно, обеспечивать программное построение точек поверхности (или контура). Такой подход позволяет уменьшить число действий пользователя по созданию двумерного или трехмерного объекта, а также в значительной степени "облегчить" интерфейс программной среды для трехмерного моделирования.

Модель объединенного кластера — МОК-объекты

Предлагаемый новый подход для создания геометрических объектов сложной формы — модель объединенного кластера (МОК) — основан на возможности устанавливать соответствие между "простейшей" геометрической областью и точкой [1, 2]. Таким образом, модель объединенного кластера (рис. 1) позволяет строить объекты, которые обладают своей собственной внутренней структурой, однако имеют внешний вид единого целостного геометрического объекта.

Вид внутренней области $\Omega = \Omega(P_1, P_2, ..., P_n)$ и его граница $\Gamma = \Gamma(P_1, P_2, ..., P_n)$ зависят от свойств и взаимного расположения входящих в модель кластерных элементов P_1, P_2, P_3, P_4, P_5, P_6. Далее будет показано, что даже если задать достаточно простые свойства кластерного элемента (рис. 2, см. вторую сторону обложки), определяющие вокруг себя некоторую двумерную пространственную область, а также правила их объединения в единую модель, то появляется возможность простыми средствами создавать довольно сложные геометрические фигуры.

С математической точки зрения кластерная модель обеспечивает конечномерную параметризацию континуальной пространственной области. Каждый клас-

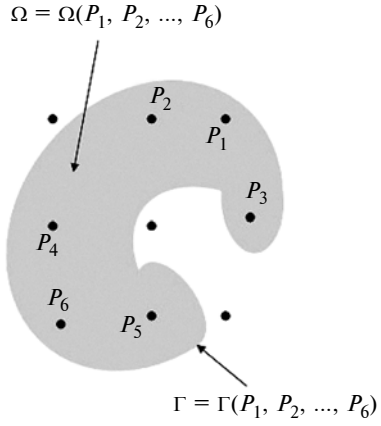


Рис. 1. Двумерный объект, являющийся объединением шести кластерных

терный элемент, имеющий определенные свойства — длину базовых направлений, типы соединяющих линий, направления касательных и др., позволяет сформировать отдельную небольшую часть от всей геометрической области. Объединение этих частей в единую геометрическую фигуру осуществляется на основании анализа взаимного расположения элементарных кластеров (топологии кластера) и может быть сделано программными средствами. Целью такой параметризации может быть в том числе минимизация хранимой информации об объекте и возможность его простой модификации (например, при деформации или разрушении). Таким образом, параметризация произвольной области с помощью конечного набора элементов фактически является алгоритмом сжатия данных, позволяющим компактно хранить данные о сложном объекте, передавать по каналам связи, а также обрабатывать и видоизменять.

Для обеспечения по возможности более гибкого описания двумерной области, принадлежащей или охватываемой кластерным элементом, может быть использована, например, структура, представленная на рис. 2, см. вторую сторону обложки.

Описание кластерного элемента включает четыре базовых вектора, определяющих направления и длины ($\mathbf{L}_{14}$, $\mathbf{L}_{34}$, $\mathbf{L}_{12}$, $\mathbf{L}_{23}$), между которыми образуются четыре подобласти, пронумерованные от I до IV, каждая из которых может иметь собственный тип геометрии: часть эллипса (круга), треугольник, два треугольника. Данные типы задаются пользователем, а вид этих областей характеризуется направлениями касательных векторов $\mathbf{D}^L_i$, $\mathbf{D}^R_i$ ($i = 0, ..., 4$), верхние индексы указывают левое и правое направления (Left, Right). Как видно на рис. 3, см. вторую сторону обложки, такая структура позволяет единообразно описать как круг, так и квадрат.

Фигуры, представленные на рис. 3 (см. вторую сторону обложки), сформированы в разработанной программной среде (язык Visual C++), они получены при выборе пользователем всего лишь одной точки, идентифицированной с кластерным элементом. Кластерный элемент может иметь форму не только круга или прямоугольника, но и более сложную форму, состоящую из отдельных частей классических геометрических фигур. Эти фигуры могут быть легко видоизменены в интерактивном режиме путем задания различных базовых длин и типов соединяющих линий, а также соответствующих направлений касательных. Отметим немаловажное преимущество кластерного подхода, при котором при создании классических геометрических фигур на плоскости — окружности, эллипса, квадрата, прямоугольника и др. достаточно лишь элементарного кластера.

Моделирование с помощью кластерного подхода естественным образом предоставляет возможность пользователю задавать в области кластерного элемента не только цвет, но и различные физические параметры, такие как плотность, теплоемкость, теплопроводность и др., что является необходимым в случае моделирования различного рода технических изделий и технологических процессов. Кластерный способ описания геометрического объекта является и более понятным для неподготовленного пользователя — ему лишь необходимо отмечать с помощью "мыши" на экране точки, ассоциированные с требуемыми кластерными элементами.

Для конечного пользователя процесс моделирования в двумерном случае выглядит следующим образом: задается пустая сеточная область с определенным числом узлов по каждой оси — в декартовой или в цилиндрической системе координат. Пользователь может выбрать любую точку (на рис. 4, см. вторую сторону обложки, показана сетка из $5 \times 5 = 25$ точек в декартовой системе координат) и отметить ее манипулятором "мышь". После создания первоначальной модели (черновой), для каждого кластерного элемента можно поменять его свойства — цвет, направления и длину базовых векторов, а также тип соединяющей линии и направления касательных. В этом случае один и тот же объект (с топологической точки зрения) может представлять или описывать различные геометрические фигуры (см. рис. 4, см. вторую сторону обложки).

Разработанное программное обеспечение позволяет в интерактивном режиме не только менять свойства кластерного элемента, но и положение базовых точек, что также приводит к изменению геометрии финального объекта и значительно расширяет описания модельных объектов.

Возможности гибкого изменения формы геометрического объекта (в том числе разрушения) являются незаменимыми для моделирования различных задач в теории упругости и механики сплошных сред. Кластерная модель естественным образом предоставляет эти возможности. Кроме того, такая модель может быть полезна при решении задач технического дизайна, когда для придания новой формы требуется внести небольшие изменения в готовое техническое изделие.

Трехмерные МОК-объекты

Очевидное обобщение структуры кластерного элемента, представленного на рис. 2 (см. вторую сторону обложки), на трехмерный случай позволяет использовать те же принципы при создании трехмерных объектов. Трехмерное проектирование полностью включают в себя все возможности двумерных моделей, которые в этом случае выступают в качестве сечений по соответствующим плоскостям для исходной трехмерной модели. При этом общие принципы формирования моделей остаются теми же — пользователю достаточно несколько раз нажать клавишу "мыши", чтобы создать геометрически сложный трехмерный объект. На рис. 5 (см. третью сторону обложки), кроме собственно пятиэлементного трехмерного МОК-объекта показаны его три сечения. Процесс создания МОК-объекта в разработанном программном обеспечении допускает две возможности включения новых элементов: либо путем выбора очередной кластерной точки на каком-либо из двумерных сечений, либо на общей трехмерной сеточной области.

Объект, показанный на рис. 5 (см. третью сторону обложки), имеет достаточно экзотический вид и был создан всего лишь пятью нажатиями клавишей "мыши". Создание объекта такого же вида на основе других программных продуктов для трехмерного моделирования является нетривиальной задачей.

На рис. 6 (см. третью сторону обложки) показан трехмерный МОК-объект, который демонстрирует, что объекты такого типа внешне сильно отличаются друг от друга в зависимости от расположения входящих в него кластерных элементов. Создание такого объекта неподготовленным пользователем также потребует всего лишь нескольких нажатий клавиш "мыши".

Заметим, что довольно простая сеточная область, состоящая из девяти точек (куб с ребром из трех точек), дает возможность пользователю простым перебором различных кластерных элементов просто поэкспериментировать с получаемыми геометрическими фигурами. Показанный на рис. 7 (см. третью сторону обложки) МОК-объект получен путем включения в него пяти точек из среднего слоя и двух точек из верхнего или нижнего слоев (двумерные сечения МОК-объекта представлены на главном окне приложения).

Созданные объекты построены на одной и той же сеточной области, что позволяет говорить о большом внешнем многообразии МОК-объектов, которые могут быть получены простым перебором точек, содержащихся в этой кубической сетке.

Программная среда трехмерного моделирования, реализующая МОК-принципы формирования трехмерных объектов, имеет достаточно понятный интерфейс, включающий различные опции вида и простые средства модификации созданного объекта — выбор типа соединяющих поверхностей, определение касательных плоскостей и перемещение кластерных точек в различных направлениях, а также собственно базовых точек. На первом этапе происходит формирование сетки из кластерных (базовых) точек — прямоугольная сетка или осесимметричная, которые затем используются для создания трехмерного МОК-объекта.

Дополнительным преимуществом кластерной модели является возможность ее модификации простыми средствами — например, путем перемещения базовых точек по плоскостям или изменение самих поверхностей, на которых они расположены. Отметим, что возможность перемещения базовых точек кластерных элементов обеспечивает очень гибкие средства изменения внешнего вида МОК-объекта. На рис. 8 (см. четвертую сторону обложки) показана модель, полученная после перемещения базовых точек в сеточной области.

На первый взгляд создается впечатление, что разработанная программная среда является просто игрушкой, не имеющая особого практического значения. Однако, если увеличить число узлов в базовой сетке, то появляется возможность создавать более сложные технические объекты практической значимости, причем, кроме непосредственно объектов, появляется возможность строить их сечения на основе принципов кластерного моделирования — двумерное сечение является двумерным МОК-объектом.

На рис. 9 (см. четвертую сторону обложки) приведены трехмерные модели технических изделий, которые могут быть созданы в программной среде неподготовленным пользователем путем простых нажатий клавиш "мыши" (используется осесимметричная сетка).

Отметим, что с точки зрения пользователя, возможности разработанной программной среды могут быть улучшены введением дополнительных средств по формированию внешней геометрии изделия — задание внешних обводов путем введения ограничивающих линий на двумерных сечениях или поверхностей для трехмерной проекции. Однако даже отсутствие этих возможностей позволяет использовать программную среду для простейшего прототипирования, например, в авиационной и космической отраслях.

Доступность и простота интерфейса программы позволяют экспериментировать при создании различных моделей. На рис. 10 (см. четвертую сторону обложки) представлены МОК-модели самолета-невидимки (*а*) и многоразового спускаемого аппарата (*б*).

Несмотря на то что разработанное программное обеспечение не является коммерческим программным продуктом, очевидны возможности его применения в техническом дизайне — простые средства формирования объектов и очевидный перебор вариантов позволяют придумывать новые формы или идеи для технических решений.

Заключение

Разработанная программная среда трехмерного моделирования, имеющая достаточно простой и интуитивно понятный интерфейс и позволяющая за несколько минут создавать сложные трехмерные объекты, может рассматриваться как первая попытка разработки систем моделирования на новых принципах. Продемонстрированные кластерные модели говорят о принципиальной возможности создания действительно простых систем трехмерного моделирования, а также востребованных коммерческих программных продуктов, базирующихся на принципах объединенных кластерных моделей.

В настоящее время проводятся исследования других типов структур кластерных элементов для обеспечения бóльших возможностей и бóльшей гибкости при описании трехмерных объектов.

Дополнительную информацию о возможных приложениях кластерных моделей при проведении математических расчетов можно найти на сайте www.clustergeometry.ru.

Список литературы

1. **Светушков Н. Н.** Кластерная модель геометрического описания сложных объектов // САПР и графика. 2010. № 3. С. 86—88.

2. **Светушков Н. Н.** Трехмерные кластерные модели для описания технологических процессов // САПР и графика. 2013. № 6. С. 89—91.

УДК 519.683

К. Е. Селезнёв, канд. техн. наук, доц., Воронежский государственный университет,
e-mail: skostik@relex.ru

Пространственный поиск данных на основе хэширования

Рассмотрен новый способ индексации и поиска многомерных данных. Особенностью предлагаемого способа является возможность поиска в произвольном пространстве, где задана функция расстояния.

Ключевые слова: *многомерный поиск, пространственный поиск, ГИС*

K. E. Seleznyov

Spatial Indexing and Searching Based on Hash Functions

The paper describes new approach for spatial data indexing in searching. This approach allows to process data by using distance function.

Keywords: *spatial index, spatial search*

Введение

Многие современные информационные системы обрабатывают данные, для которых задана функция расстояния или функция похожести, полностью удовлетворяющая требованиям, предъявляемым к функции расстояния. Обработка данных такого рода требует организации эффективного хранения, индексации и поиска, что включает в себя выполнение операций вставки, удаления и поиска. Предложен новый способ индексации и поиска многомерной информации, основанный на хэшировании, который можно применить для обработки данных произвольной размерности.

Постановка задачи и обзор методов

Индексация и поиск пространственных данных подразумевает выполнение операций трех видов: вставки, удаления и поиска. У операций поиска при этом существуют два варианта запуска [1]. Первый вариант предполагает, что на вход дается некоторый информационный объект и пороговое значение расстояния, по которым система должна найти все информационные объекты, удаленные от заданного объекта на расстояние, не большее указанного порогового значения. Второй вариант поиска предполагает, что на вход дается информационный объект и некоторое число N, а система должна найти N информационных объектов, ближайших к заданному.

Вычисление расстояния между объектами осуществляется на основе функции метрики, имеющей строгое математическое определение. Так, функция двух аргументов $R(x, y)$ является метрикой, если выполняются следующие условия:

1) функция $R(x, y)$ определена для любых двух информационных объектов x и y;

2) $R(x, y) = 0$ в том и только в том случае, если $x = y$;

3) $R(x, y) > 0$, если $x <> y$;

4) $R(x, y) = R(y, x)$ для любых x и y;

5) выполняется неравенство треугольника: $R(x, y) <= <= R(x, z) + R(z, y)$.

На настоящее время существует большое число различных способов решения поставленной задачи индексации и поиска многомерной информации [1, 2]. Среди них необходимо отметить следующие.

- **Способ Grid** применяется для работы с точками на поверхности, которая предварительно разбивается на множество ячеек. Ячейки покрывают всю поверхность, не пересекаются друг с другом и имеют простую математическую форму (квадраты, прямоугольники, треугольники, шестиугольники и т. п.), которая выбирается исходя из специфики конкретной задачи. Предполагается, что по координатам любой точки поверхности можно легко вычислить индекс соответствующей ей ячейки.

Хранение исходного множества точек осуществляется с помощью ассоциативного массива, ключом которого являются индексы ячеек поверхности, а значениями — множества точек, относящихся к соответс-

твующим ячейкам. Поскольку в каждой ячейке располагается небольшое число точек, то для хранения этого множества можно использовать самые простые структуры данных (например, линейные массивы), а при поиске допустимо выполнять линейное сканирование.

Алгоритм поиска определяет индекс необходимой ячейки и просматривает точки внутри найденной ячейки, а затем — соседних с ней ячеек. Указанное предположение является главным недостатком Grid, поскольку для вычисления индекса ячейки необходимо анализировать внутреннюю структуру объекта и получать его координаты на плоскости.

- **Способ Z-order** основан на проецировании пространственных объектов на числовую ось. Функция проецирования подбирается таким образом, чтобы, во-первых, близко расположенные объекты соответствовали близким значениям на числовой оси, и во-вторых, по двум значениям на числовой оси можно было оценить минимальное и максимальное расстояния между соответствующими этим значениям объектами. Указанные свойства функции проецирования позволяют выполнять индексацию и поиск с помощью различных методов работы с числовыми данными (например, B-деревьями).

Принцип хранения и алгоритм поиска данного способа похожи на те, что лежат в основе Grid, используется ассоциативный массив, ключом которого являются значения функции проецирования, а значениями — множества информационных объектов, для которых функция проецирования дает одно и то же значение. Алгоритм поиска для искомого информационного объекта сначала вычисляет значение функции проецирования, а затем выполняет линейный поиск в множестве точек, соответствующем полученному значению функции проецирования. Недостатком Z-Order является тот факт, что этот способ применим только в том случае, когда можно построить функцию проецирования информационных объектов на числовую ось.

- **Квадрантные деревья** ориентированы на работу с точками на плоскости. Индекс представляет собой сильно ветвящееся дерево, узлы и листья которого ставятся в соответствие частям двумерного пространства, а корень дерева — всему пространству целиком. Построение дерева начинается с корня. Если множество точек состоит из более чем одного элемента, то у корня дерева создается четыре дочерних узла, все пространство разбивается на четыре квадранта, каждому созданному дочернему узлу ставится в соответствие один квадрант и процедура запускается рекурсивно. Таким образом, каждый лист квадрантного дерева содержит не более одного элемента из исходного множества точек. Соответственно, если множество точек пусто или содержит всего одну точку, то квадрантное дерево состоит из одного узла.

Алгоритм поиска точки всегда начинает работу с корня квадрантного дерева, на каждой итерации переходит к дочернему узлу и останавливается в одном из листьев дерева. Далее, анализируются точка в найденном листе, соседних листьях и т. д. Недостатком квадрантных деревьев является то, что они применимы только для обработки точек на плоскости или могут быть обобщены для случая многомерных векторов.

- **Способ R-tree (R-деревья)** ориентирован на работу с множеством векторов в многомерном пространстве. Каждый узел R-дерева соответствует некоторому прямоугольному параллелепипеду и множеству точек, лежащему внутри этого параллелепипеда. У узла может быть несколько подузлов, при этом соответствующие им параллелепипеды могут пересекаться, но обязательно лежат внутри охватывающего параллелепипеда, соответствующего данному родительскому узлу. Корень дерева соответствует прямоугольному параллелепипеду, охватывающему все обрабатываемое множество векторов.

Существуют различные модификации R-деревьев, среди которых следует упомянуть R+-деревья, R*-деревья, Гильбертовы R-деревья и X-деревья. В каждой модификации используются собственные алгоритмы поиска, вставки и удаления элементов, хотя общие идеи внутреннего устройства дерева остаются примерно одинаковые. Недостатком всех перечисленных способов является тот факт, что R-деревья применимы только для случая многомерных векторов.

- **M-деревья** являются модификацией R-деревьев. Отличием M-деревьев является то, что их узлы соответствуют не прямоугольным параллелепипедам, а шарам в пространстве объектов. Это позволяет применять M-деревья для информационных объектов, которые не представимы в виде векторов чисел, но для которых задана метрика.

- **KD-деревья** во многом похожи на квадрантные деревья и R-деревья. В случае KD-деревьев происходит работа с векторами чисел, корень дерева соответствует множеству всех векторов, узел дерева — подмножеству, которое целиком входит в подмножество, соответствующее родительскому узлу. Отличие KD-деревьев от R-деревьев заключается в способе разбиения множества точек на подмножества. Так, KD-деревья всегда являются бинарными. Для каждого их узла выбирается некоторая гиперплоскость, разбивающая множество векторов на две части, одна из которых соответствует одному дочернему узлу, а другая часть — другому дочернему узлу. Недостатком KD-деревьев является то, что они применимы только для случая числовых векторов.

Все перечисленные способы, кроме M-деревьев, позволяют работать с многомерными данными, представимыми в виде векторов чисел, но не применимы для случая, когда известна только функция вычисления расстояния.

Устройство индекса

Для построения индекса предварительно задается N фиксированных информационных объектов A_1, ..., A_N, далее называемых опорными объектами. Для каждого из них указывается набор положительных чисел $R_{i,\,1}$, ..., $R_{i,\,k}$, где i — номер опорного объекта, а k — количество чисел, которое может быть фиксированным, а может отличаться для различных опорных объектов. В данной статье для упрощения рассуждений считаем, что k одинаково для всех опорных объектов. Числа $R_{i,\,1}$, ..., $R_{i,\,k}$ задают радиусы шаров с центром в A_i и, таким образом, разбивают пространство на $k + 1$ областей, далее называемых кольцами опорного объекта A_i. В первом кольце лежат все объекты, находящиеся от A_i на расстоянии, не большем $R_{i,\,1}$. Во втором кольце лежат все объекты, находящиеся от A_i на расстоянии, большем $R_{i,\,1}$, но не большем $R_{i,\,2}$, и т. д. В последнем кольце лежат все объекты, находящиеся от A_i на расстоянии, большем $R_{i,\,k}$.

Набор опорных объектов и радиусов колец для каждого объекта позволяет для любого информационного объекта X определить натуральные числа I_1, ..., I_N, где I_K — это номер кольца объекта A_K, в котором расположен объект X. Числа I_1, ..., I_N представляют собой вектор натуральных чисел и задают код области, в которой расположен объект X. Таким образом получается, что все пространство разбивается на конечное множество областей, далее называемых сегментами. Каждый сегмент однозначно идентифицируется своим кодом, а любой информационный объект всегда лежит в одном и только одном сегменте.

Коды сегментов обладают важными свойствами. Пусть объект X расположен в сегменте с кодом I_1, ..., I_N, а объект Y — в сегменте с кодом J_1, ..., J_N. Во-первых, если коды сегментов совпадают (т. е. вектор I_1, ..., I_N равен вектору J_1, ..., J_N), то X и Y могут быть сколь угодно близки друг к другу. Во-вторых, если соответствующие друг другу компоненты векторов I_1, ..., I_N и J_1, ..., J_N различаются не более чем на 1, то X и Y также могут быть сколь угодно близки друг к другу, поскольку содержащие их области имеют общую границу. Например, если объект X лежит в первом кольце опорного объекта, а Y — во втором, то это не исключает ситуации, когда X и Y лежат сколь угодно близко к границе первого и второго колец, а следовательно, и сколь угодно близко друг к другу.

Третье свойство кодов сегментов обобщает идею второго. Так, если для некоторого i, обозначающего номер опорного информационного объекта, значение I_i отличается от J_i более чем на единицу, то можно вычислить нижнюю оценку расстояния между X и Y. Пусть число I_i меньше J_i. Тогда, расстояние от X до Y не меньше, чем $R_{i,\,p} - R_{i,\,q}$, где $p = J_i - 1$, а $q = I_i$. Фактически, число $D_i = R_{i,\,p} - R_{i,\,q}$ представляет собой суммарную ширину колец, расположенных между кольцами I_i и J_i. Например, если объект X лежит в первом кольце, а Y — в пятом, то расстояние от X до Y как минимум включает в себя ширину второго, третьего и четвертого колец.

По двум заданным информационным объектам X и Y можно вычислить вектор чисел D_i, максимальный элемент $D = \max(D_i)$ будет показывать минимальную оценку расстояния между X и Y. В процессе вычисления вектора D_i сами информационные объекты используются только для определения кодов сегментов (векторов I_i и J_i) и, следовательно, число D показывает минимальное расстояние между сегментами, которое может быть вычислено еще на этапе настройки индекса (задания опорных объектов и радиусов колец).

Хранение множества информационных объектов похоже на использование хэш-таблиц [3] и заключается в отдельном хранении множеств информационных объектов каждого сегмента, при этом могут использоваться любые из известных способов хранения и индексации многомерных данных [1, 2]. Однако, если ведется работа с информационными объектами, для которых известна только функция расстояния, то для эффективной организации хранения данных каждого сегмента могут быть использованы только M-деревья. Альтернативой им являются простейшие линейные структуры (массив, список и т. д.), поиск в которых осуществляется методом полного сканирования. Такой подход реализуется очень просто, но эффективно применим только в том случае, когда число объектов в каждом сегменте сравнительно невелико.

При работе с множеством сегментов используется только операция точного поиска сегмента по его коду. Поэтому для организации хранения множества сегментов могут использоваться любые из известных способов индексации точных данных (хэш-таблицы, B-деревья и т. д.), рассмотренные в работе [3]. Наиболее эффективным способом является применение хэш-таблиц, поскольку код сегмента позволяет легко перейти к числовому значению хэш по формуле $H = I_1 + I_2k + I_3k^2 + ... + I_Nk^{(N-1)}$. Альтернативой хэш-функциям является многомерный массив, у которого в ячейках хранятся сегменты, число измерений равно числу опорных информационных объектов, а диапазон индексов каждого измерения — числу колец у каждого информационного объекта.

Алгоритмы работы с индексом

При добавлении нового информационного объекта сначала происходит определение кода сегмента, к которому относится данный объект, а затем — поиск сегмента по коду и добавление информационного объекта в подмножество объектов, относящихся к данному сегменту. Аналогично, при удалении информационного объекта сначала происходит определение кода сегмента, к которому относится данный объект, а затем — поиск сегмента по коду и удаление информационного объекта из подмножества объектов, относящихся к данному сегменту.

При поиске M объектов, наиболее близких к заданному объекту X, сначала происходит определение кода сегмента $I_1...I_N$, в котором располагается объект X. Затем осуществляется поиск объектов в этом сегменте, а также в сегментах с кодами $J_1...J_N$, при этом сегменты с кодами $J_1...J_N$ рассматриваются в порядке увеличения минимального расстояния от объектов сегмента $J_1...J_N$ до объектов сегмента $I_1...I_N$. Определение минимального расстояния происходит согласно сформулированному выше свойству кодов сегментов. По мере рассмотрения областей $J_1...J_N$ все находящиеся в них объекты добавляются в список найденных объектов Y_1, ..., Y_F.

Процедура поиска останавливается в том случае, когда список Y_1, ..., Y_F содержит более M объектов (т. е. $F > M$) или рассмотрены все сегменты. На завершающем этапе список объектов Y_1, ..., Y_F сортируется в порядке возрастания расстояния от X до элементов этого списка, и, если длина списка оказывается больше M, то происходит удаление элементов в хвосте списка.

Если выполняется поиск объектов, находящихся от X на расстоянии, не бóльшем заданного порогового значения T, то используется тот же самый итеративный алгоритм поиска, но изменяется критерий остановки: поиск останавливается в случае, когда минимальное расстояние от объекта из сегмента $I_1...I_N$ до объекта из очередного просматриваемого сегмента $J_1...J_N$ становится бóльшим, чем T. На завершающем этапе из списка Y_1, ..., Y_F удаляются все объекты, для которых выполняется неравенство $R(X, Y) > T$.

Подбор опорных векторов

Скорость работы представленных выше алгоритмов поиска напрямую зависит от числа коллизий — ситуаций, когда в одном сегменте располагается большое число информационных объектов. В свою очередь, число коллизий напрямую зависит от выбора информационных объектов и радиусов колец, что может проводиться либо вручную, либо автоматически на основе некоторой тестовой выборки данных, при этом сначала определяют сами опорные векторы $A_1...A_n$, а затем — радиусы колец $R_{i,j}$.

Наиболее простым и очевидным решением является следующее. Для определения $A_1...A_n$ сначала выполняется кластеризация множества тестовых объектов (может использоваться любой из известных методов), после которой центры кластеров выбирают в качестве опорных объектов, а радиусы колец выбирают так, чтобы "разрезать" кластеры на примерно равные части и тем самым уменьшить число коллизий. Каждое кольцо $R_{i,j}$ может быть кольцом малого диаметра и разрезать кластер объектов, расположенных рядом с опорным A_i, а может быть кольцом большого диаметра и разрезать другие кластеры.

Тем не менее, в качестве опорных объектов могут использоваться не только центры кластеров, но и любые другие элементы тестового множества. Может оказаться эффективным делать опорными объекты, расположенные на границе кластера, или объекты, расположенные вне какого-либо кластера. Наиболее общий критерий эффективности выбора выглядит следующим образом: информационный объект удобно выбрать в качестве опорного, если возможно задание радиусов колец таким образом, чтобы разделить множество остальных объектов на примерно равные части. Если же некоторые опорные объекты уже заданы, то добавление нового должно минимизировать число коллизий, т. е. для него должно быть возможным задать кольца таким образом, чтобы разделить уже существующие сегменты на равные части и тем самым сократить число коллизий.

Подбор опорных векторов может решаться путем перебора $A_1...A_N$ для заданного числа опорных векторов N и числа колец K. В процессе перебора сначала рассматривают более эффективные опорные объекты, а затем — менее эффективные. Для каждого сочетания $A_1...A_N$ определяют оптимальные радиусы $R_{i,j}$ и оценивают число коллизий. По окончании перебора выбирают вариант, дающий наименьшее число коллизий.

Изложенный принцип подбора опорных векторов и радиусов колец эффективен, если тестовая выборка похожа на реальные данные. В процессе работы может оказаться, что статистические характеристики реальных данных изменились и необходимо откорректировать набор опорных векторов и/или радиусов колец. Для этого выбирают сегменты, в которых возникло наибольшее число коллизий, а затем у уже существующих опорных объектов вводят дополнительные кольца таким образом, чтобы разрезать найденные сегменты на примерно равные части и, тем самым, сократить число коллизий.

Заключение

Предложен новый метод индексации и поиска пространственных данных. Его отличительной чертой является возможность работы с любыми информационными объектами, для которых задана функция расстояния (метрика). Это свойство позволяет надеяться на эффективное применение метода не только в геоинформационных системах, но и в ряде других задач. На данный момент времени создано экспериментальное программное обеспечение, моделирующее работу указанного метода. Оно позволяет задать опорные информационные объекты и радиусы их колец, а затем проследить работу алгоритма поиска. Кроме того, экспериментальное программное обеспечение по заданному тестовому набору данных вычисляет число коллизий в каждой области пространства, что позволяет оценить эффективность задания опорных информационных объектов и их радиусов.

Список литературы

1. **Шекхар Ш.**, **Чаула С.** Основы пространственных баз данных. М.: КУДИЦ-Образ, 2004. 336 с.
2. **Гулаков В. К., Трубаков А. О.** Многомерные структуры данных. Брянск: Изд-во БГТУ, 2010. 387 с.
3. **Кнут Д. Э.** Искусство программирования. Т. 3: Сортировка и поиск: Пер. с англ. Изд. 2. М.: Вильямс, 2004. 832 с.

ИНФОРМАЦИЯ

ООО "Издательство "Новые технологии". 107076, Москва, Стромынский пер., 4

Дизайнер *Т. Н. Погорелова*. Технический редактор *Е. М. Патрушева*. Корректор *Т. В. Пчелкина*

Сдано в набор 13.05.2014 г. Подписано в печать 19.06.2014 г. Формат 60×88 1/8. Заказ PI714
Цена свободная.

Оригинал-макет ООО "Адвансед солюшнз". Отпечатано в ООО "Адвансед солюшнз".
119071, г. Москва, Ленинский пр-т, д. 19, стр. 1.