УДК 004.05

Трансформация и верификация программы сортировки прикрепленных к шине устройств

Шелехов В.И. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Описывается трансформация и верификация программы bus_sort_breadthfirst, принадлежащей ядру ОС Linux и реализующей сортировку устройств, прикрепленных к шине компьютера. Программа трансформируется с языка Си в язык cP с раскрытием макросов, реорганизацией структур и устранением указателей. Далее программа преобразуется на язык функционального программирования WhyML. Для полученной программы строится спецификация и проводится дедуктивная верификация.

Ключевые слова: *дедуктивная верификация, трансформации программ, функциональное программирование, предикатное программирование, операционная система Linux, шина компьютера, двусвязный список.*

1. Введение

Методы дедуктивной верификации и связанные с ней методы трансформации программ ранее разрабатывались для библиотечных программ из ядра ОС Linux [3-7]. Разработана универсальная система трансформаций, устраняющих указатели, для операций с массивами и рекурсивными структурами данных. Однако при применении данной технологии к не самому сложному фрагменту самой ОС Linux обнаружилась необходимость модификации не только наборов трансформаций, но и архитектуры всей системы трансформации.

Исходной задачей является дедуктивная верификация программы bus_sort_breadthfirst, принадлежащей модулю bus.c, организующему работу с драйверами и устройствами, прикрепленными к шине компьютера. Модуль bus.c входит в состав ядра операционной системы (ОС) Linux. Программа bus_sort_breadthfirst реализует сортировку списка устройств, прикрепленных к шине.

Используется простой алгоритм сортировки вставками (insertion sort). По исходному двусвязному списку устройств строится новый сортированный список. Очередной элемент исходного списка перемещается в подходящее место нового списка среди ранее упорядоченных элементов. Во избежание коллизий при одновременном асинхронном

доступе разными процессами к списку устройств, на период сортировки блокируется доступ к списку для других процессов.

Сложность программы bus_sort_breadthfirst следует признать невысокой в сравнении с другими фрагментами ядра ОС Linux. Тем не менее, ее дедуктивная верификация нереализуема в рамках существующих инструментов дедуктивной верификации программ на языке Си. Одна из причин этого кроется в используемом стиле написания программ ОС Linux, в частности, при работе со списками.

Цель настоящей работы – на примере программы bus_sort_breadthfirst определить методы трансляции программ ОС Linux на язык функционального программирования WhyML [22] через промежуточный язык cP. Язык cP не содержит указателей, но максимально близок к языку Си по типам данных и языковым конструкциям. Итоговая программа на языке WhyML станет намного проще исходной программы на языке Си, что существенно упрощает дедуктивную верификацию в рамках инструмента автоматического доказательства Why3 [22].

Во втором разделе дается описание особенностей трансляции с языка Си на язык cP. В третьем разделе описывается многоэтапный процесс трансформации исходной программы bus_sort_breadthfirst с получением рекурсивной программы на языке cP. В следующем разделе описывается процесс спецификации программы. Кодирование программы на языке WhyML [22] представлено в пятом разделе. Особенности процесса дедуктивной верификации предикатной программы в системе Why3 [22] описывается в шестом разделе. Далее обзор работ по верификации программ с двусвязными списками. В заключении анализируются новые виды трансформации. В Приложении 1 представлена итоговая программа со спецификациями на языке WhyML. В приложении 2 приводятся формулы корректности, подлежащие доказательству в системе Why3 [22].

2. Трансляция с языка Си на язык cP

Язык cP получается из языка Си устранением в нем указателей. Любое вхождение указателя в программе заменяется объектом, являющимся значением указателя. Операции с указателями заменяются эквивалентными действиями без указателей. Остальные конструкции языка Си: типы данных, наборы операторов и операций – максимально сохраняются в языке cP. Некоторые конструкции взяты из языка WhyML [22], что упростит последующий перевод на WhyML. Есть конструкции, заимствованные из языка предикатного программирования P [1].

Трансляция с устранением указателей реализуется применением набора трансформаций разного вида. *Трансформация* A --> B определяет замену конструкции A, содержащей указатели, на эквивалентную ей конструкцию B без указателей, возможно при соблюдении определенных условий. Дополнительным результатом трансляции является *трансформационная программа*, содержащая модификации программы, в частности, модифицированные описания типов и заголовков функций. Реализация автоматической трансляции нетривиальна. У пользователя должна быть возможность изменить трансформационную программу и запустить трансляцию повторно, в частности, поменять введенные при трансформации имена типов и переменных.

Разработка языка функционального программирования cP и методов трансляции на него с языка Си осуществляется с апробацией на наборе библиотечных программ из ядра ОС Linux [3-7]. Разработана универсальная система трансформаций, устраняющих указатели, для операций с массивами и рекурсивными структурами данных. Однако при применении данной технологии к не самому сложному фрагменту самой ОС Linux обнаружилась необходимость модификации не только наборов трансформаций, но и архитектуры всей системы трансформации при трансляции с языка Си на язык cP.

Применению трансформаций предшествует потоковый анализ программы, определяющий информационные связи между переменными программы. Строится картина памяти в виде набора независимых гнезд объектов после каждого оператора. С этой целью применяется символьное исполнение программы, используемое иногда при анализе видов структур (shape analysis) [8, 11,12, 15, 16, 20, 21].

Мы рассматриваем *объекты* как объекты памяти исполняемой программы. Это простые переменные, массивы, структуры – записи в виде набора полей, списки, деревья и другие рекурсивные структуры. Выделяются *подобъекты*: элементы массивов, поля структур, вершины списка и т.д. *Гнездо объекта* определяет объект, его структуру и набор переменных, для которых объект или его компоненты являются значениями переменных.

3. Трансформация программы сортировки устройств шины

3.1. Программа сортировки устройств шины

Программа bus_sort_breadthfirst реализует сортировку списка устройств, прикрепленных к шине. Ее код находится в модуле bus.c и доступен по адресу:

https://elixir.bootlin.com/linux/v5.9/source/drivers/base/bus.c#L952

Функция bus_sort_breadthfirst оперирует переменными нескольких типов структур, вызывает другие функции и макросы в своем теле. Подробное описание всех используемых типов, функций и макросов было бы очень объемным и сложным. Проведем процедуру извлечения нужного нам фрагмента программы методом построения *вырезки программы*, когда в извлекаемом фрагменте остаются только те функции и макросы, вызовы которых встречаются в теле функции. В описаниях структур будем опускать те поля, которые не используются в выделяемом фрагменте.

Функция bus_sort_breadthfirst вызывает функции device_insertion_sort_klist, bus_get_device_klist, тела которых помещаются в извлекаемый фрагмент. Тела других вызываемых функций мы не намерены рассматривать, и при верификации будем использовать только их спецификацию. Аналогичным образом, в дальнейшем раскрываются не все вызовы макросов, а лишь некоторые. Для автоматического извлечения фрагмента при трансляции необходимо будет как-то указывать, какие объекты включаются во фрагмент. Для этого можно использовать независимую трансформационную программу, а при первом запуске транслятора решение о включении объекта решается в диалоге с пользователем.

Дадим сначала описания базисных структур данных.

3.1.1. Двусвязный список определяется описанием типа**:**

```
struct list_head {
        struct list_head *next, *prev;
};
```

Указатель по полю next ссылается на следующий элемент списка, указатель по полю prev – на предыдущий элемент списка. Для всякого элемента списка x выполняются два равенства: x->next->prev = x и x->prev->next = x. Список кольцевой. Имеется *головной элемент* с указателями на начальный и конечный элементы списка. Пустой список определяется единственным головным элементом s, причем s->next = s и s->prev = s. Указатель **NULL** не используется.

Описание переменной s с инициализацией пустым списком реализуется макросом LIST_HEAD(s), порождающим описание:

```
struct list_head  s = { &s, &s };
```

Отметим, что определяемая структура находится не в куче, а в стеке текущей функции, и поэтому существует только до завершения вызова этой функции.

В приведенном описании list_head нет поля для собственно элемента списка, то есть данных, хранящихся в элементе списка. В соответствии со стилем, принятым в ОС Linux,

элемент списка, то есть структура list_head, является частью другой структуры klist_node по полю n_node:

```
struct klist_node { ... struct list_head  n_node; ... };
```

Указатель на структуру klist_node получается вызовом стандартного макроса container_of(ptr, klist_node, n_node), применяемого к указателю ptr на структуру list_head внутри структуры klist_node.

Такой способ реализации списков не требует введения дополнительных структур и переменных. Достаточно включить в объект поле типа list_head. Таким способом объект может быть включен в несколько разных списков. Например, структура device участвует более чем в пяти разных списках.

Отметим, что головной элемент списка обычно не включается внутрь другого объекта.

3.1.2. Список с асинхронным доступом определяется описанием типа:

```
struct klist {
        spinlock_t          k_lock;
        struct list_head k_list;   ...
};
```

Список с асинхронным доступом является двусвязным списком. Доступ к списку может быть блокирован через поле k_lock. Поле k_list определяет головной элемент списка. Другие элементы списка, доступные через поля next и prev, помещаются внутрь структуры klist_node [1]:

```
struct klist_node { ... struct list_head  n_node; ... };
```

В частности, элемент списка klist_node хранит информацию о числе процессов, имеющих доступ к этой вершине.

3.1.3. Основные структуры

Структуры, используемые в программе bus_sort_breadthfirst, большие и сложные. Здесь описываются лишь те их поля, которые встречаются в программе.

Исходным объектом является шина, определяемая структурой [2]:

```
struct bus_type {...struct subsys_private *p; ...};
```

Структура subsys_private [3] определяет внутреннюю информацию, связанную с шиной:

```
struct subsys_private { ...struct klist klist_devices; ...};
```

1 https://elixir.bootlin.com/linux/v5.9.1/source/include/linux/klist.h
2 https://elixir.bootlin.com/linux/v5.9/source/include/linux/device/bus.h#L82
3 https://elixir.bootlin.com/linux/v5.9/source/drivers/base/base.h#L40

Поле klist_devices определяет внутреннюю подструктуру klist для головного элемента списка устройств, прикрепленных к шине. Устройство, принадлежащее этому списку, определяется структурой [4]:

struct device_private { **… struct** klist_node knode_bus; **… struct** device *device; **…** };

Поле knode_bus определяет внутреннюю подструктуру klist_node для очередного элемента из списка устройств, прикрепленных к шине. Данная структура device_private содержит внутреннюю информацию об устройстве. Ссылка на само устройство находится в поле device.

Наконец, структура device из 48 полей [5]:

struct device { …**struct** device_private *p; …};

Здесь поле p типа device_private, ссылающееся на внутреннюю информацию об устройстве.

Отметим особенности организации списка устройств, затрудняющие понимание программы bus_sort_breadthfirst. Список устройств реализован через вложенные структуры klist_node внутри структуры device_private. При этом ссылка на следующую структуру klist_node внутри структуры для другого устройства реализуется через поле next структуры list_head внутри структуры klist_node.

3.1.4. Программа bus_sort_breadthfirst

Программа bus_sort_breadthfirst [6] вызывает функции bus_get_device_klist и device_insertion_sort_klist, представленные далее перед основной программой.

```
struct klist *bus_get_device_klist(struct bus_type *bus)
{
        return &bus->p->klist_devices;
}
```

Функция bus_get_device_klist реализует доступ к головной структуре списка устройств. Функция device_insertion_sort_klist реализует вставку устройства a в упорядоченный список устройств list таким образом, чтобы модифицированный список list остался упорядоченным. Сравнение реализуется через функцию compare стандартным образом: compare(a, b) <= 0 означает a<=b.

---

4 https://elixir.bootlin.com/linux/v5.9/source/drivers/base/base.h#L88
5 https://elixir.bootlin.com/linux/v5.9/source/include/linux/device.h#L515
6 https://elixir.bootlin.com/linux/v5.9/source/drivers/base/bus.c#L952

```
static void device_insertion_sort_klist(struct device *a, struct list_head *list,
                                        int (*compare)(const struct device *a,
                                                       const struct device *b))
{
        struct klist_node *n;
        struct device_private *dev_prv;
        struct device *b;

        list_for_each_entry(n, list, n_node) {
                dev_prv = to_device_private_bus(n);
                b = dev_prv->device;
                if (compare(a, b) <= 0) {
                        list_move_tail(&a->p->knode_bus.n_node,
                                       &b->p->knode_bus.n_node);
                        return;
                }
        }
        list_move_tail(&a->p->knode_bus.n_node, list);
}
```

В теле функции device_insertion_sort_klist макрос list_for_each_entry раскрывается в заголовок цикла, итерирующего по списку list. Макрос to_device_private_bus реализует доступ структуры device_private по указателю вложенной в него структуры klist_node. Функция list_move_tail(x, y) удаляет начальный элемент из списка x и помещает его перед начальным элементом списка y.

```
void bus_sort_breadthfirst(struct bus_type *bus,
                           int (*compare)( const struct device *a,
                                           const struct device *b))
{
        LIST_HEAD(sorted_devices);
        struct klist_node *n, *tmp;
        struct device_private *dev_prv;
        struct device *dev;
        struct klist *device_klist;

        device_klist = bus_get_device_klist(bus);

        spin_lock(&device_klist->k_lock);
        list_for_each_entry_safe(n, tmp, &device_klist->k_list, n_node) {
                dev_prv = to_device_private_bus(n);
                dev = dev_prv->device;
                device_insertion_sort_klist(dev, &sorted_devices, compare);
        }
        list_splice(&sorted_devices, &device_klist->k_list);
        spin_unlock(&device_klist->k_lock);
}
```

В начале тела заводится пустой список sorted_devices с головным элементом в стеке. Список устройств, прикрепленных к шине bus, фиксируется в переменной device_klist. Макрос spin_lock блокирует доступ к списку устройств с возможным ожиданием окончания предыдущей блокировки. Макрос spin_unlock освобождает список от блокировки. Макрос list_for_each_entry_safe раскрывается в заголовок цикла, итерирующего по списку устройств. Поскольку очередной элемент далее удаляется из списка, в переменной tmp предварительно фиксируется следующий элемент списка. Вызов функции list_splice прикрепляет отсортированный список sorted_devices к опустевшему списку устройств шины с переносом всех элементов, кроме головного.

Требуется трансформировать данную программу в эквивалентную программу на языке WhyML [22] и провести ее дедуктивную верификацию.

3.2. Реорганизация структур

Такие особенности, как возвратная ссылка из структуры device_private на структуру device и использование макроса container_of для доступа к элементам списка серьезно усложняют анализ программы и применение трансформаций. Для устранения этих особенностей применяется аппарат слияния структур.

3.2.1. Слияние структур

В структуре device_private поле device ссылается на структуру device. А в структуре device поле p ссылается на структуру device_private. Такая зависимость между двумя типами характерна для рекурсивных типов. Однако в данном случае, поле device в структуре device_private всего лишь возвратная ссылка на главную структуру device для облегчения доступа к ней. Для переменной dev_prv имеет место равенство dev_prv->device->p = dev_prv. Аналогично, для переменной dev выполняется dev->p->device = dev.

Возвратная ссылка, вообще говоря, не является обязательной. Поле device можно устранить и передавать структуру device дополнительным параметром всякий раз, когда нужен доступ к ней. Такого рода трансформацию можно было бы применить в данном случае. Однако более подходящей является другая трансформация: объединение структур device и device_private в одну структуру device’ с устранением полей p и device. Представим ее в следующем виде:

device\p ++ device_private\device ---> device’

В соответствии с этой трансформацией dev_prv->device заменяется на dev_prv, а dev->p на dev. Объединенная структура device’ имеет следующее определение:

struct device’ { **… struct** klist_node knode_bus; **…** };

Особенность в том, что трансформацию слияния структур должен представить пользователь. Потому что на основе анализа представленной выше вырезки кода программы невозможно определить, что ссылка по полю device является возвратной. Потенциально при такой рекурсивной зависимости возможны структуры разного вида: списки, деревья и даже графы.

Представим программу после слияния структур. Дополнительно раскроем макросы. Макросы для заголовков циклов содержат внутри вызовы других макросов, которые также требует раскрытия.

```
static void device_insertion_sort_klist(struct device’ *a, struct list_head *list,
                                        int (*compare)(const struct device’ *a,
                                                       const struct device’ *b))
{
        struct klist_node *n;
        struct device’ *dev_prv;
        struct device’ *b;

        for (n = container_of(list-> next, klist_node, n_node);
            &n -> n_node != list;
            n = container_of (n -> n_node.next, klist_node, n_node)) {
                dev_prv = container_of(n, device’, knode_bus);
                b = dev_prv;
                if (compare(a, b) <= 0) {
                        list_move_tail(&a->knode_bus.n_node,
                                       &b->knode_bus.n_node);
                        return;
                }
        }
        list_move_tail(&a->knode_bus.n_node, list);
}
```

```
void bus_sort_breadthfirst(struct bus_type *bus,
                           int (*compare)( const struct device' *a,
                                           const struct device' *b))
{
        struct list_head  sorted_devices = { & sorted_devices, & sorted_devices};
        struct klist_node *n, *tmp;
        struct device' *dev_prv;
        struct device' *dev;
        struct klist *device_klist;

        device_klist = bus_get_device_klist(bus);

        spin_lock(&device_klist->k_lock);
       list_head *dev_list = &device_klist->k_list;
       for (n = container_of(dev_list->next, klist_node, n_node),
             tmp = container_of (n -> n_node.next, klist_node, n_node);
         & n -> n_node != dev_list;
         n = tmp, tmp = container_of (tmp-> n_node.next, klist_node, n_node)) {
                dev_prv = container_of(n, device', knode_bus);
                dev = dev_prv;
                device_insertion_sort_klist(dev, &sorted_devices, compare);
        }
        list_splice(&sorted_devices, dev_list);
        spin_unlock(&device_klist->k_lock);
}
```

Код после раскрытия макросов громоздкий. Чтобы его упростить, введена промежуточная переменная dev_list:

```
list_head *dev_list = &device_klist->k_list;
```

Отметим, что раскрываются лишь макросы, входящие в вырезку программы. Не раскрываются макросы spin_lock и spin_unlock.

3.2.2. Реорганизация списков

Рассмотрим оператор dev_prv = container_of(n, device', knode_bus) и связанные с ним описания типов структур:

```
struct klist_node { ... struct list_head  n_node; ... };
struct device' { ... struct klist_node  knode_bus; ... };
```

Макрос container_of по указателю n на вложенную структуру klist_node по полю knode_bus внутри структуры device' вычисляет указатель на структуру device'.

Список устройств, определяемый через вложенную структуру klist_node, можно определить явным образом удалением поля knode_bus из структуры device'. Указатель на модифицированную структуру device' вставляется последним полем в структуру klist_node:

```
struct device'' { ... ... };
struct klist_node' { ... struct list_head n_node; ... ; struct device'' *knode_dev};
```

Это универсальный способ реорганизации списков, применимый при реорганизации нескольких списков для структуры device'. Однако в нашей вырезке кода список только один, и поэтому можно применить слияние структур следующего вида:

klist_node ++ device'\knode_bus ---> klist_node'

Поля структуры device' помещаются в конец структуры klist_node. Тип device' всюду заменяется на klist_node':

```
struct klist_node' { ... struct list_head n_node; ... ; /* поля device'*/};
```

Реализуются трансформации вида:

container_of(n, device', knode_bus) ---> n
&a->knode_bus.n_node ---> &a.n_node

После реорганизации списков на базе структуры klist_node аналогичным образом для модифицированной программы проводится реорганизация списков на базе структур list_head. Применяется слияние структур следующего вида:

list_head ++ klist_node'\n_node ---> list_head'

Поля структуры klist_node' помещаются в конец структуры list_head. Тип klist_node' всюду заменяется на list_head':

```
struct list_head'_{ struct list_head *next, *prev; /* поля klist_node' и device'*/};
```

Реализуются трансформации вида:

container_of(ptr, klist_node, n_node) ---> ptr
&a.n_node ---> a
n -> n_node.next ---> n -> next
&n -> n_node ---> n

Применение двух стадий реорганизации списков дает следующую программу:

```
struct bus_type {...struct subsys_private *p; ...};
struct subsys_private { ...struct klist klist_devices; ...};
struct klist {      spinlock_t k_lock;  struct list_head'  k_list; ... }
struct list_head' { struct list_head *next, *prev; ... /* поля klist_node' и device'*/};

struct klist *bus_get_device_klist(struct bus_type *bus)
{          return &bus->p->klist_devices; }
```

```
static void device_insertion_sort_klist(struct list_head' *a, struct list_head' *list,
                                        int (*compare)(const struct list_head' *a,
                                                       const struct list_head' *b))
{
        struct list_head' *n;
        struct list_head' *dev_prv;
        struct list_head' *b;

        for (n = list-> next;  n != list; n = n -> next) {
                dev_prv = n;
                b = dev_prv;
                if (compare(a, b) <= 0) {
                        list_move_tail(a, b);
                        return;
                }
        }
        list_move_tail(a, list);
}

void bus_sort_breadthfirst(struct bus_type *bus,
                           int (*compare)( const struct list_head' *a,
                                           const struct list_head' *b))
{
        struct list_head'  sorted_devices = { & sorted_devices, & sorted_devices};
        struct list_head' *n, *tmp;
        struct list_head' *dev_prv;
        struct list_head' *dev;
        struct klist *device_klist;

        device_klist = bus_get_device_klist(bus);

        spin_lock(&device_klist->k_lock);
       list_head' *dev_list = &device_klist->k_list;
       for (n = dev_list->next,  tmp = n -> next; n != dev_list; n = tmp, tmp = tmp-> next)
{
                dev_prv = n;
                dev = dev_prv;
                device_insertion_sort_klist(dev, &sorted_devices, compare);
        }
        list_splice(&sorted_devices, dev_list);
        spin_unlock(&device_klist->k_lock);
}
```

Программа после реорганизации структур существенно упростилась. Возникает вопрос, насколько удачным является используемый в ОС Linux аппарат работы со списками. Безусловно, он сокращает число структур и тем самым улучшает работу с памятью. Однако, к сожалению, значительно осложняет программу.

3.3. Обрезание программы

В принципе, можно было бы проводить верификацию программы, полученной в результате проведенных трансформаций. Однако есть особенности, которые невозможно или неинтересно проверять для данной вырезки программы. Структура шины фактически не задействована в алгоритме. Процесс доступа к списку устройств шины через структуры bus_type и subsys_private не может быть предметом верификации, поскольку здесь нет другой спецификации, кроме самого кода. Указанные структуры являются лишними. Исходной можно было бы считать структуру klist. Однако обнаруживается, что проверка правильности вставки примитивов spin_lock и spin_unlock является тривиальной задачей и совсем неинтересной для дедуктивной верификации. Поэтому входным объектом будем считать список устройств, определяемых переменной dev_list.

Отметим, что верификация функций list_move_tail и list_splice, принадлежащих модулю list.h, была бы интересной, однако не проводится в рамках поставленной задачи. Для этих функций используются лишь их спецификации. С учетом этого, запись отсортированного списка устройств на место исходного списка, не представляется интересной для верификации. Поэтому будем считать, что исходная программа просто возвращает отсортированный список.

Функцию compare полезно вынести из параметров и сделать глобальной неинтерпретированной функцией (без тела).

Представим программу после обрезания частей, где дедуктивная верификация плохо применима. Попутно устраним лишние переменные.

```
struct list_head’{ struct list_head *next, *prev; … /* поля klist_node’ и device’*/};
int compare(const struct list_head’*a,  const struct list_head’ *b);

static void device_insertion_sort_klist(struct list_head’ *a, struct list_head’ *list)
{
        struct list_head’ *n;

        for (n = list-> next;  n != list; n = n -> next) {
                if (compare(a, n) <= 0) {
                        list_move_tail(a, n);
                        return;
                }
        }
        list_move_tail(a, list);
}
```

```
list_head' * bus_sort_breadthfirst(list_head' *dev_list)
{
        struct list_head'  sorted_devices = { & sorted_devices, & sorted_devices};
        struct list_head' *n, *tmp;
       for (n = dev_list->next,  tmp = n -> next; n != dev_list; n = tmp, tmp = tmp-> next)
{
                device_insertion_sort_klist(n, &sorted_devices);
        }
        return  &sorted_devices;
}
```

3.4. Анализ программы

Трансформации, устраняющие указатели в программе с заменой на эквивалентные конструкции без указателей, требуют предварительного проведения потокового анализа программы, использующего символьное исполнение программы [7]. Дополнительно контролируется правильность операций со структурами. Одна из возможных ошибок – это замыкание некоторого элемента списка на один из предыдущих элементов, в результате чего головной элемент становится недоступным из нового цикла, а на последнем потерянном элементе нарушается условие: x->next->prev = x.

Предварительно заменим циклы **for** на циклы типа **while**.

```
list_head' * bus_sort_breadthfirst(list_head' *dev_list)
{
        struct list_head'  sorted_devices = { & sorted_devices, & sorted_devices};
        struct list_head' *n, *tmp;
        n = dev_list->next;  tmp = n -> next;
       while (n != dev_list) {
                device_insertion_sort_klist(n, &sorted_devices);
                n = tmp; tmp = tmp-> next
        }
        return  &sorted_devices;
}
```

Очевидно, что операторы tmp = tmp -> next и tmp = n -> next эквивалентны. Это позволяет упростить программу следующим образом:

```
list_head' * bus_sort_breadthfirst(list_head' *dev_list)
{
        struct list_head'  sorted_devices = { & sorted_devices, & sorted_devices};
        struct list_head' *n, *tmp;
        n = dev_list->next;
       while (n != dev_list) {
                tmp = n -> next;
                device_insertion_sort_klist(n, &sorted_devices);
                n = tmp;
        }
        return  &sorted_devices;
}
```

В процессе потокового анализа для всякой переменной типа list_head'* определяется, является ли ее значение указателем на головной элемент. Переменная dev_list указывает на головной элемент, поскольку для нее используется только две операции: dev_list->next и n != dev_list. То же самое для переменной list. Начальная инициализация переменной sorted_devices определяет ее значением головной элемент. Все остальные переменные не ссылаются на головной элемент.

Объект, соответствующий головному элементу, будем обозначать именем H. В начальный момент символьного исполнения значением параметра dev_list является объект {H, S} – это список, начинающийся головным элементом H, за которым следует объект S. Данный факт будем изображать в виде гнезда: «dev_list: {H, S}». Через S, S1, S2, ... будем обозначать, возможно, пустую, последовательность элементов, завершающуюся выходом на головной элемент H.

Второе гнездо «sorted_devices: {H}» возникает при исполнении описания переменной sorted_devices.

Результатом исполнения оператора n = dev_list->next является следующее гнездо:

dev_list: {H, n: N, S1} | dev_list, n: {H}

Объект N обозначает элемент, следующий за H и являющийся значением переменной n. Объект N непустой и отличен от всех других объектов. Если значением dev_list является пустой список, реализуется вторая альтернатива: dev_list, n: {H}. Здесь значением переменных dev_list и n является головной элемент.

Определим состояние памяти при входе в цикл:

< dev_list: {H, n: N, S1} | dev_list, n: {H} ; sorted_devices: {H, S2} >

При потоковом анализе устанавливается, что переменная sorted_devices модифицируется. Поэтому значение sorted_devices обобщается введением объекта S2. Переменная dev_list также модифицируется. Ее обобщение неочевидно и возможно потребуется в дальнейшем.

После исполнения условия n != dev_list и входе в цикл состояние памяти меняется следующим образом:

< dev_list: {H, n: N, S1} ; sorted_devices: {H, S2} >

После исполнения оператора tmp = n -> next получим состояние памяти:

< dev_list: {H, n: N, tmp: T, S3} | dev_list, tmp: { H, n: N}; sorted_devices: {H, S2} >

При входе в заголовок функции device_insertion_sort_klist состояние памяти модифицируется следующим образом:

< DV ; list, sorted_devices: {H, S2} >,

где DV = dev_list: {H, a, n: N, tmp: T, S3} | dev_list, tmp: {H, a, n: N} .

Представим код функции device_insertion_sort_klist после замены цикла **for** на **while**:

```
static void device_insertion_sort_klist(struct list_head' *a, struct list_head' *list)
{       struct list_head' *n = list-> next;
        while (n != list) {
                if (compare(a, n) <= 0) {
                        list_move_tail(a, n);
                        return;
                };
                n = n -> next
        }
        list_move_tail(a, list);
}
```

Первый оператор n = list-> next модифицирует состояние памяти следующим образом:

< DV ; list, sorted_devices: {H, n: R, S4} | list, n, sorted_devices: {H} >

В начале цикла необходимо следующее обобщение:

< DV ; list, sorted_devices: {H, X, n: R, S4} | list, n, sorted_devices: {H, X } >

Здесь X – некоторая, возможно пустая, последовательность элементов между H и R.

После исполнения условия n != list и входе в цикл состояние памяти меняется следующим образом:

< DV ; list, sorted_devices: {H, X, n: R, S4} >

Вызов функции list_move_tail(a, n) удаляет объект N из списка dev_list и вставляет его в список list перед объектом R. Получим:

<dev_list:{H, tmp:T, S3} | dev_list, tmp:{H} ; list, sorted_devices:{H, X, a, n: N, n: R, S4}>

Чтобы данный шаг символьного исполнения вызова функции, не включенной в вырезку, мог быть сделан автоматически, необходим некоторый способ определения такого вычисления от пользователя.

По другой ветви тела цикла после выполнения n = n -> next получим:

< DV ; list, sorted_devices: {H, X, R, n: R1, S5} | list, n, sorted_devices: {H, X, R } >

Отметим, что замены X, R на X и S5 на S4 унифицируют данное состояние памяти с состоянием в начале цикла. При завершении цикла по условию n = list получим следующее состояние памяти:

< DV ; list, n, sorted_devices: {H, X } >

Вызов функции list_move_tail(a, list) удаляет объект N из списка dev_list и вставляет его в список list перед объектом H. Получим:

<dev_list:{H, tmp:T, S3} | dev_list, tmp:{H} ; list, n, sorted_devices: { a, n: N, H, X}>

Поскольку список list кольцевой, это равносильно вставке элемента в конец цикла:

<dev_list:{H, tmp:T, S3} | dev_list, tmp:{H} ; list, n, sorted_devices: { H, X, a, n: N}>

Далее необходимо унифицировать состояния памяти двух разных выходов из функции device_insertion_sort_klist:

<dev_list:{H, tmp:T, S3} | dev_list, tmp:{H} ; sorted_devices: { H, X, n: N, S6}>

Отметим, что мы удалили локальные имена list, n и a из состояния памяти.

После выполнения оператора n = tmp получим:

<dev_list:{H, n, tmp:T, S3} | dev_list, n, tmp:{H} ; sorted_devices: { H, X, N, S6}>

Отметим, что переменная n, значением которой был объект N, теперь перемещается на объект T. Данное конечное состояние унифицируемо с состоянием в начале цикла.

Таким образом, завершен анализ программы. Операции со структурами корректны при условии, что действия функция list_move_tail корректны. В частности, подтверждена конфигурация памяти: dev_list: {H, n: N, S1} | dev_list, n: {H}, в соответствии с которой переменная n указывает на следующий элемент после головного элемента, и между ними нет промежуточных элементов.

3.5. Замена двусвязных списков стандартными списками

В списке list_head′ элемент списка (собственно данные) представлен набором неиспользуемых полей. Для верификации элемент списка нужно представить явно. Будем использовать неинтерпретированный тип device для элементов списка, а также для аргументов функции compare.

```
struct list_head′ { struct list_head *next, *prev; struct device *elem};
int compare(const struct device  *a,  const struct device *b);
```

Далее применяются трансформации, устраняющие указатели с заменой операций с двусвязными списками эквивалентными действиями со стандартными списками.

Определим сначала трансформации для типов:

```
list_head′  --->  list (device)
```

Тип list определен в языке cP следующим описанием:

```
type list (type T)  = union { nil; cons(T car, list(T) cdr) };
```

Пустой список из единственного головного элемента трансформируется в конструктор nil. Указатель по полю next трансформируется в поле cdr, значением которого является трансформированный список по полю next. Для проведения трансформаций операций по указателю x.prev исходный список представляется в виде конкатенации двух списков: y++x.

Описания типа list_head′ и функции compare трансформируются на язык cP следующим образом:

```
type device;
type devList = list(device);
int compare(device  a,  b);
```

Здесь введено имя devList для упрощения трансформированной программы. Для описания вида list_head′ *x применяется трансформация:

```
list_head′ *x  ---> devList x
```

Если значением x является головной элемент, то для операции сравнения n != x используется трансформация:

```
n != x   ---> n != nil
```

Трансформация конструкции вида x->next имеет особенности. Если значением x является головной элемент, то используется трансформация:

```
x -> next   ---> x
```

Рассмотрим случай, когда головной элемент не является значением переменной x. В общем случае действует трансформация:

```
x -> next   ---> if (x = nil) nil else  x.cdr
```

Пусть C – выражение и значением переменной x не является головной элемент. Присваивание вида x = C трансформируется в эквивалентный оператор присоединения:

```
x = C,   ---> x <- C
```

Вызов функции f(C) для параметра типа list_head′* трансформируется с подстановкой параметра в режиме присоединения:

f(C) ---> f(&C)

Для инициализации головным элементом используется трансформация:

sorted_devices = { &sorted_devices, &sorted_devices } → sorted_devices = nil

Вызов list_move_tail(a, list) применяется к переменной list, значением которой является головным элемент. В результате элемент a.car вставляется перед головным элементом, что эквивалентно вставке a.car в конце списка list. Данную особенность следует учитывать введением специальной трансформации, которая данный вызов заменяет вызовом другой функции, вставляющей a.car в конце списка.

list_move_tail(a, list) ---> list_move_end(a, list)

В результате применения указанных выше трансформаций получаем следующую программу:

```
type device;
type devList = list(device);
int compare(device  a,  b);
device_insertion_sort_klist(devList  a, devList  list)
{       devList n <- list;
        while (n != nil) {
                if (compare(a.car, n.car) <= 0) {
                        list_move_tail(&a, &n);
                        return;
                };
                n <- n.cdr
        }
        list_move_end(&a, &list);
}
devList bus_sort_breadthfirst(devList dev_list)
{
        devList  sorted_devices = nil;
        devList n, tmp;
        n <- dev_list;
       while (n != nil) {
                tmp <- n.cdr;
                device_insertion_sort_klist(&n, &sorted_devices);
                n <- tmp;
        }
        return  sorted_devices;
}
```

3.6. Замена присоединения на присваивание

Чтобы транслировать полученную программу на язык WhyML [22], необходимо заменить операторы присоединения эквивалентными операторами присваивания. Сначала определим функции list_move_tail и list_move_end на языке cP.

```
devList list_move_tail(devList  a, y, n){ devList t = y++a.car++n; a<-a.cdr; return t };
devList list_move_end(devList  a, list){ list = list++a.car; a<-a.cdr; return list };
```

Исходный список list представляется в виде конкатенации двух списков: list = y++n. Список y передается дополнительным параметром функции list_move_tail. Вычисление y должно проводиться в теле device_insertion_sort_klist синхронно с изменениями списка n. Здесь определена более простая версия функций с учетом того, что параметр a, подставляемый присоединением, указывает на первый элемент списка. Удаления первого элемента из списка не происходит. Имя a просто перемещается на следующий элемент.

Заменим подстановку присоединением &sorted_devices на подстановку по значению в вызове функции:

```
sorted_devices = device_insertion_sort_klist(&n, sorted_devices);
```

Модифицированное значение sorted_devices в теле функции необходимо будет передавать как результат функции.

Проведем открытую подстановку тел функций list_move_tail и list_move_end на место их вызовов. Вставим операторы перевычисления списка y. Оператор tmp <- n.cdr и втянем в начало тела функции device_insertion_sort_klist. Оператор n <- tmp втянем в конец тела функции перед операторами **return**. Получим:

```
devList device_insertion_sort_klist(devList  a, devList  list)
{       devList n <- list; devList y = nil;
        devList tmp<- a.cdr;
        while (n != nil) {
                if (compare(a.car, n.car) <= 0) {
                        list = y++a.car++n; a<-a.cdr// list_move_tail(&a, y, n);
                        a <- tmp;
                        return list;
                };
                y = y++n.car; n <- n.cdr
        }
        list = list++a.car; a<-a.cdr // list_move_end(&a, list);
        a <- tmp;
        return list;
}
```

Поскольку обе ветви функции завершаются оператором a<-a.cdr, оператор a <- tmp эквивалентен a <- a . Это дает возможность удалить операторы tmp<- a.cdr и a <- tmp. Ставший последним оператор a<-a.cdr выносится из тела функции после ее вызова. Получим оператор n <- n.cdr. Как следствие, переменная a более не является результатом функции и становится просто аргументом. После проведенных преобразований оказывается возможным заменить все оставшиеся операторы присоединения эквивалентными операторами присваивания, а также заменить подстановку присоединением на подстановку значением.

Итоговая программа:

```
type device;
type devList = list(device);
int compare(device a,  b);
devList device_insertion_sort_klist(devList  a, devList  list)
{       devList n = list; devList y = nil;
        while (n != nil) {
                if (compare(a.car, n.car) <= 0) {
                        return y++a.car++n;
                };
                y = y++n.car; n = n.cdr
        }
        return list++a.car;
}

devList bus_sort_breadthfirst(devList dev_list)
{
        devList  sorted_devices = nil;
        devList n = dev_list;
       while (n != nil) {
                sorted_devices = device_insertion_sort_klist(n, sorted_devices);
                n = n.cdr;
        }
        return  sorted_devices;
}
```

3.7. Трансформация в рекурсивную программу

Итоговую программу можно было бы далее кодировать на языке WhyML [22], писать предусловия и постусловия функций и инварианты циклов; затем проводить доказательство условий корректности в системе Why3 [22]. Опыт показывает [7], что верификация эквивалентной рекурсивной программы существенно проще и быстрее.

Чтобы построить рекурсивную программу, нужно сначала построить рекурсивную программу для каждого цикла. Ниже приведены рекурсивные программы двух циклов.

```
devList dev_insert(devList a, y, n)
{       if (n = nil)  return list++a.car
        else if (compare(a.car, n.car) <= 0) return y++a.car++n
        else   return dev_insert (a, y++n.car, n.cdr)
}
devList  dev_sort (devList  sorted_devices, n){
        if (n = nil) return  sorted_devices
        else return dev_sort(device_insertion_sort_klist(n, sorted_devices), n.cdr)
}
```

Функции device_insertion_sort_klist и bus_sort_breadthfirst преобразуются к виду:

```
devList device_insertion_sort_klist(devList  a, devList  list)
{ return dev_insert(a, nil, list) }

devList bus_sort_breadthfirst(devList dev_list)
{  return dev_sort (nil, dev_list); }
```

Подставим тело функции device_insertion_sort_klist на место вызова. Представим полную рекурсивную программу.

```
type device;
type devList = list(device);
int compare(device  a,  b);
devList dev_insert(devList a, y, n)
{       if (n = nil)  return list++a.car
        else if (compare(a.car, n.car) <= 0) return y++a.car++n
        else   return dev_insert (a, y++n.car, n.cdr)
}


devList  dev_sort (devList  sorted_devices, n){
        if (n = nil) return  sorted_devices
        else return dev_sort(dev_insert(n, nil, sorted_devices), n.cdr)
}
devList bus_sort_breadthfirst(devList dev_list)
{  return dev_sort (nil, dev_list); }
```

4. Спецификация программы

Для каждой из трех функций программы необходимо написать предусловие и постусловие. Свойства функции compare надо определить набором аксиом.

Для спецификации используются теории List, Append, Sorted и Permut модуля list (http://why3.lri.fr/stdlib/list.html) стандартной библиотеки системы Why3 [22]. Операция конкатенации «++» определена в теории Append.

```
type device;
type devList = list(device);
int compare(device  a,  b);
axiom Simm :   ∀ device x, y.  compare(x, y)= - compare(y, x)
axiom TotalLe : ∀ device x, y. compare(x, y) <=0 or compare(y, x) <=0;
axiom TransLe : ∀ device x, y, z.
                compare(x, y)<=0 & compare(y, z)<=0 ⇒ compare(x,z)<=0;
```

Аксиомы функции compare использовались при верификации алгоритма пирамидальной сортировки [4]. Спецификации функций приведены ниже. Стандартная переменная result используется для возвращаемого значения результата функции.

```
devList dev_insert(devList a, list, y, n)
pre a!=nil & sorted( list) & list = y++n
post sorted(result) & permut(result, y ++ a.car ++ n)
{       if (n = nil)  return list++a.car
        else if (compare(a.car, n.car) <= 0) return y++a.car++n
        else  return  dev_insert (a, y++n.car, n.cdr)
}

devList  dev_sort (devList  sorted_devices, n)
pre sorted(sorted_devices)
post sorted(result) & permut(result, sorted_devices++n)
{       if (n = nil) return  sorted_devices
        else return dev_insert(dev_insert(n, nil, sorted_devices), n.cdr)
}

devList bus_sort_breadthfirst(devList dev_list)
post sorted(result) & permut(result, dev_list)
{   return dev_sort (nil, dev_list); }
```

5. Кодирование на WhyML

Программа вместе с ее спецификацией транслируется на язык WhyML [22]. В языке WhyML нет имен полей конструктора типа list. Доступ к полям реализуется только через оператор **match**. Отношение вида n != nil представлено функцией notempty. Однако можно использовать hd a вместо a.car.

В функции dev_insert вместо a.car используется переменная ah. Поскольку элемент списка не допускается в операции конкатенации, вхождения элементов обрамляются конструктором Cons.

```
theory Bus_sort
  use int.Int,  list.List, list.Append, list.Sorted, list.Permut, list.HdTlNoOpt
  type device
  type devList = list device
  function compare (a b: device): int
  axiom Simm :   forall x, y: device. compare x y = - compare y x
  axiom TotalLe : forall x, y: device. compare p x y <=0 \/ compare y x <=0
  axiom TransLe : forall x, y, z: device.
          compare x y <=0 -> compare y z <=0-> compare x z <=0
  predicate le (x,y: device) = compare x y <= 0
  clone export list.Sorted with type t = device, predicate re = le

  let function notempty (l: dev_node) : bool =
     match l with
     | Nil      -> false
     | Cons _ _ -> true
     end

  let rec function  dev_insert (ah: device) (list y x: devList): devList
        requires { sorted list /\ list = y++x }
        ensures{sorted result /\ permut result (y++ (Cons ah x))}
  =
     match x with
     | Nil      ->  list ++ (Cons ah Nil)
     | Cons xh xt ->
       if compare ah xh <= 0 then  y++( Cons ah x)
       else  dev_insert a list (y++  (Cons xh Nil)) xt
     end

  let rec function    dev_sort(x sorted_devices  : devList) : devList
        requires {sorted sorted_devices  }
        ensures{ sorted result /\ permut result  (x++sorted_devices) }
  =
     match x with
     | Nil      ->  sorted_devices
     | Cons xh xt ->
         let sorted_devices1 = dev_insert x sorted_devices  Nil sorted_devices in
             dev_sort xt sorted_devices1
     end

  let function bus_sort_breadthfirst(dev_list : devList  ): devList
        ensures{ sorted result /\ permut result dev_list }
  =
           dev_sort dev_list Nil

end (*Bus_sort *)
```

6. Процесс дедуктивной верификации

Верификация проводилась в системе Why3 версии 1.3.1 с SMT-решателями Z3 4.8.6, CVC3 2.4.1, CVC4 1.7. Система Why3 [22] базируется на языке функционального программирования WhyML. Его частью является язык спецификаций why3. Для дедуктивной верификации применяется классический метод Хоара [14] с генерацией формул корректности на языке why3. Для доказательства формул корректности к системе Why3 может быть подключено более 20 разнообразных инструментов автоматического доказательства теорем. Это автоматические решатели, SMT-решатели и интерактивные инструменты, такие как PVS [17], HOL и Coq [10], в которых пользователь строит доказательство, применяя команды инструмента.

Система Why3 имеет собственные средства интерактивного доказательства. На предыдущих версиях системы Why3 этими средствами в комбинации с SMT-решателями часто не удавалось завершить доказательство. Некоторые ветви доказательства приходилось переводить в систему Coq, где доказательство часто было длительным и сложным [4]. Средства интерактивного доказательства системы Why3 были расширены в последней версии 1.3.1, что позволило полностью проводить доказательство в системе Why3 без перехода в систему Coq.

Итоговая программа bus_sort_breadthfirst на языке WhyML с дополнительными леммами представлена в Приложении 1. Для этой программы система Why3 сгенерировала три формулы корректности, по одной на каждую функцию, общим объемом в 56 строк; см. Приложение 2. Каждая из трех формул – это связка формул корректности. Набор формул корректности получаются из связки применением стандартной стратегии split_vc.

При доказательстве формул корректности для функции dev_insert вставлено дополнительное предусловие:

```
requires { forall d: device. mem d y -> le d ah }
```

Здесь утверждается, что любой элемент списка y не больше вставляемого элемента ah. Это условие необходимо для доказательства сортированности списка после включения ah.

В процессе доказательства формул корректности введены дополнительные три леммы:

```
lemma sorted_mem1:
  forall c: device, l: list device.
    (forall d: device. mem d l -> le d c) /\ sorted l <-> sorted (l++( Cons c Nil))
lemma consAppend: forall ah: device, y x2: list device.
                          ((y ++ Cons ah Nil) ++ x2) = (y ++ Cons ah x2)
lemma Append_nil_l:    forall l: list 'a. Nil ++ l = l
```

Автоматическое доказательство с помощью SMT-решателей использует набор лемм из импортируемых библиотечных теорий. В теории Sorted имеется лемма sorted_mem, где элемент присоединяется к сортированной последовательности слева. Потребовалась симметричная лемма sorted_mem1, где элемент присоединяется справа. Аналогично, лемма Append_nil_l симметрична лемме Append_nil из теории Append. В подобных случаях часто использовалась команда assert(Nil ++ l = l). Однако когда ситуация встречается более одного раза, лучше ввести лемму.

Доказательство формул корректности в системе Why3 было существенно проще и быстрее, чем доказательство для программ memweight [2] и kstrtoull [5]. Обнаружились трудности при декомпозиции доказательства, в частности, при наличии конструкций let. Средства интерактивного доказательства в системе Why3 пока еще существенно слабее, чем в PVS [17] и Coq [10].

7. Другие работы по верификации программ с двусвязными списками

Не так много работ по дедуктивной верификации программ, оперирующих двусвязными списками. Во всех работах верификация опосредована через модель программы. Спецификация определяется через модель. Верифицируемая программа модифицируется под модель.

В работе [13] представлен гибридный функциональный/императивный язык DASL с дедуктивной верификацией операций с двусвязными списками, бинарными деревьями, графами и другими сложными структурами данных. Реализована трансляция с DASL на языки Java, ADA, CakeML, VHDL. Данная технология обеспечивает получение эффективных программ на этих языках для критических приложений, где требуется высокий уровень надежности. Система верификации на базе языка DASL интегрирована с системой автоматического доказательства ACL2, поддерживающей обширные библиотеки для разнообразных структур данных.

В работе [18] определена разрешимая логика для операций со сложными циклическими структурами данных, такими как двухсвязные списки. Представлен эффективный набор правил доказательства в этой логике. Доступ по указателю заменяется вызовами специальных функций. Спецификации программ на языке псевдокода представлены формулами в этой логике. Эффективность автоматической верификации подтверждается на наборе небольших программ, в том числе функций list_add, list_add_tail и list_del

(https://elixir.bootlin.com/linux/v5.9/source/include/linux/list.h) из стандартной библиотеки ядра ОС Linux.

В работе [19] представлена дедуктивная верификация реализации двусвязных списков в ядре ОС FreeRTOS для архитектуры ARM V7. Верификация проводилась в инструменте автоматического доказательства HOL4 на базе абстрактной модели, включающей: модель памяти (кучи), содержащей списки и элементы списков, формализации структур данных, модели операций над списками (создания, включения элемента, упорядоченного включения, удаления), инварианта памяти и инварианта списка. Корректность инварианта списка и операций над списками доказана для модели машинного кода проведением моделирования относительно абстрактной модели. Общая трудоемкость составляет 6 чел/мес.

В работе [9] описывается верификация модуля ограниченных двусвязных списков, входящего в библиотеку GNAT языка ADA. В целях верификации модуль переписан на язык Си с привязкой к абстрактной модели списков, где каждый элемент списка является элементом одного большого массива, а указатель на список представляется индексом элемента в этом массиве. Спецификация модуля для ADA 2012 была написана ранее с использованием логики разделения (separation logic). Доказательство корректности модуля относительно спецификации проводилось в инструменте автоматического доказательства VeriFast, поддерживающего логику разделения. Завершена верификация 27 из 39 функций, входящих в модуль ограниченных двусвязных списков.

Наш подход к верификации программ с двусвязными списками принципиально отличается от применяемых подходов к верификации через модель программы. Цикличность в данных это не фундаментальное свойство программ, а всего лишь способ кодирования данных. Аналогично, двусвязный список это один из способов кодирования классического списка. Нужно не моделировать структуру двусвязного списка, а реализовать способ его раскодирования в обычный список.

8. Заключение

До сих пор процесс трансформации и дедуктивной верификации программ из библиотеки ядра ОС Linux рассматривался по следующей схеме. Происходит автоматическая трансляция с языка Си на язык WhyML [22] с получением в качестве дополнительного результата трансформационной программы, содержащей модификации программы, в частности, модифицированные описания типов и заголовков функций. Пользователь может изменить трансформационную программу и запустить трансляцию повторно, в частности, поменять введенные при трансформации имена типов и переменных. Далее строятся

спецификации программы на языке WhyML, и проводится дедуктивная верификация в системе Why3 [22].

Процесс трансформации программы bus_sort_breadthfirst не укладывается в указанную схему. Требуется другая архитектура инструмента трансформации с языка Си на язык cP.

Построение вырезки программы, описанное в разд. 3.1, предполагает диалог с пользователем и может быть реализовано независимым инструментом. Если вырезку надо будет проводить повторно, полезно сохранять набор выбранных объектов (типов, функций, макросов и др.), возможно, в трансформационной программе.

Появились новые виды трансформаций: раскрытие макросов, слияние структур, устранение возвратных ссылок в структурах, реорганизация списков с вынесением заголовков списков из структур, замена присоединений на присваивание. Новая специфика – несколько этапов, когда трансформации очередного этапа строятся по программе, полученной на предыдущем этапе. Здесь необходимо будет визуализировать промежуточное состояние трансформируемой программы. Устранение возвратных ссылок при слиянии структур нельзя сделать автоматически, поскольку для вырезки программы невозможно установить, что ссылка является возвратной. Неочевидна и последовательность трансформаций. Все это требует серьезной доработки с апробацией на других программах.

Система трансформаций для операций с двусвязными списками отличается от ранее разработанной системы для рекурсивных структур, использующих пустой указатель NULL. Предварительно в потоковом анализе необходимо распознавать списки, начинающиеся головным элементом. Особый случай – вызов list_move_tail(a, list), преобразование которого возможно следует задавать через трансформационную программу.

Список литературы

1. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования P. Версия 0.12. Новосибирск, 2013. 28с. http://persons.iis.nsk.su/files/persons/pages/plang12.pdf (дата обращения 15.12.2021)
2. Тумуров Э.Г., Шелехов В.И. Трансформация, спецификация и верификация программы вычисления числа элементов множества, представленного в виде битовой шкалы. — Системная информатика, № 16. — Новосибирск, 2020. — С. 103-136. URL: https://www.system-informatics.ru/files/article/tumurovshelehov.pdf (дата обращения 15.12.2021)
3. Шелехов В.И. Верификация предикатной программы бинарного поиска объекта произвольного типа // Системная информатика, № 15. Новосибирск, 2019. С. 45-64. URL: http://persons.iis.nsk.su/files/persons/pages/fsearch2.pdf (дата обращения 14.12.2020)

4. Шелехов В.И. Верификация предикатной программы пирамидальной сортировки с применением обратных трансформаций // Системная информатика, № 16. Новосибирск, 2020. С. 75-102. URL: https://persons.iis.nsk.su/files/persons/pages/sort9.pdf (дата обращения 14.12.2020)
5. Шелехов В.И. Верификация программы преобразования строки в целое число // Системная информатика, № 17. — Новосибирск, 2020. — С. 43-90. https://persons.iis.nsk.su/files/persons/pages/kstr2.pdf (дата обращения 15.12.2021)
6. Шелехов В.И. Дедуктивная верификация программы конкатенации строк с применением обратной трансформации // Знания-Онтологии-Теории (ЗОНТ-19). Новосибирск, ИМ СО РАН, 2019. С. 369-378. URL: http://persons.iis.nsk.su/files/persons/pages/logcflc1.pdf (дата обращения 14.12.2020)
7. Шелехов В.И. Методы трансформации и дедуктивной верификации программы инвертирования списков // Программная инженерия. 2021. 22с .https://persons.iis.nsk.su/files/persons/pages/listspi.pdf (дата обращения 15.12.2021)
8. Boockmann J.H., Lüttgen G., Mühlberg J.T. Generating Inductive Shape Predicates for Runtime Checking and Formal Verification // Leveraging Applications of Formal Methods, Verification and Validation. Verification. 2018. LNCS 11245. P. 64-74.
9. Cauderlier R., Sighireanu M. A Verified Implementation of the Bounded List Container // TACAS 2018, LNCS 10805, P. 172-189.
10. The Coq Proof Assistant. https://coq.inria.fr/ (дата обращения 15.12.2021)
11. Dudka K., Holík L., Peringer P., Trtík M., Vojnar T. From Low-Level Pointers to High-Level Containers // Verification, Model Checking, and Abstract Interpretation. 2016. LNCS 9583. P. 431-452.
12. Haller I., Slowinska A., Bos H. Scalable data structure detection and classification for C/C++ binaries // Empirical Software Engineering. 2016, v. 21, Issue 3. P. 778–810.
13. Hardin D., Slind K. Using ACL2 in the Design of Efficient, Verifiable Data Structures for High-Assurance Systems // EPTCS 280, 2018, P. 61-76.
14. Hoare C. A. R. An axiomatic basis for computer programming // Communications of the ACM. Vol. 12 (10). 1969. P.576–585.
15. Holík L., Lengál O., Rogalewicz A., Šimáček J., Vojnar T. Fully Automated Shape Analysis Based on Forest Automata // Computer Aided Verification. CAV 2013. LNCS 8044. P. 740-755.
16. Jung C., Clark N. DDT: Design and evaluation of a dynamic program analysis for optimizing data structure usage // 42nd Int. Symposium on Microarchitecture (MICRO 42), NY, 2009. P. 56-66.
17. PVS Specification and Verification System. *SRI International*. http://pvs.csl.sri.com/. (дата обращения 14.12.2020)
18. RakamarićJesse Z., Bingham J., Hu A.J. An Inference-Rule-Based Decision Procedure for Verification of Heap-Manipulating Programs with Mutable Data and Cyclic Data Structures // Verification, Model Checking, and Abstract Interpretation VMCAI 2007. LNCS 4349, P 106-121.

19. Sanan D., Liu Y., Zhao Y., Xing Z., Hinchey M. Verifying FreeRTOS' Cyclic Doubly Linked List Implementation: From Abstract Specification to Machine Code // 20th International Conference on Engineering of Complex Computer Systems (ICECCS 2015). 2015, P. 120-129.
20. White D., Rupprecht T., L¨uttgen G.. DSI: An Evidence-based Approach to Identify Dynamic Data Structures in C Programs // Intl. Symposium on Software Testing and Analysis (ISSTA 2016). ACM, 2016. P. 259-269.
21. White D.H., Lüttgen G. Identifying Dynamic Data Structures by Learning Evolving Patterns in Memory // Tools and Algorithms for the Construction and Analysis of Systems. 2013. LNCS 7795, P. 354-369.
22. Why 3. Where Programs Meet Provers. URL: http://why3.lri.fr (дата обращения 15.12.2021)

Приложение 1

Программа Bus_sort на языке WhyML

```
theory Bus_sort
  use int.Int,  list.List, list.Append, list.Sorted, list.Permut, list.HdTlNoOpt
  use  list.Length, list.Mem
  type device
  type devList = list device
  function compare (a b: device): int
  axiom Simm :   forall x, y: device. compare x y = - compare y x
  axiom TotalLe : forall x, y: device. compare x y <=0 \/ compare y x <=0
  axiom TransLe : forall x, y, z: device.
    compare x y <=0 -> compare y z <=0-> compare x z <=0
  predicate le (x y: device) = compare x y <= 0
  clone export list.Sorted with type t = device, predicate le = le

  lemma sorted_mem1:
    forall c: device, l: list device.
     (forall d: device. mem d l -> le d c) /\ sorted l <-> sorted (l++( Cons c Nil))
  lemma consAppend: forall ah: device, y x2: list device.
      ((y ++ Cons ah Nil) ++ x2) = (y ++ Cons ah x2)
  lemma Append_nil_l:    forall l: list 'a. Nil ++ l = l
  let rec ghost function  dev_insert (ah: device) (list y x: devList): devList
      requires { sorted list /\ list = y++x }
      requires { forall d: device. mem d y -> le d ah }
      ensures {sorted result /\ permut result (y++( Cons ah x))}
      variant { length x }
    =
      match x with
       | Nil        ->   list ++ (Cons ah Nil)
       | Cons xh xt ->
         if compare ah xh <= 0 then   y++( Cons ah x)
         else  dev_insert ah list (y++( Cons xh Nil)) xt
      end

  let rec ghost function   dev_sort(x sorted_devices  : devList) : devList
      requires {sorted sorted_devices  }
      ensures { sorted result /\ permut result  (x++sorted_devices) }
    =
      match x with
       | Nil      ->  sorted_devices
       | Cons xh xt ->
         let sorted_devices1 = dev_insert xh sorted_devices  Nil sorted_devices in
         dev_sort xt sorted_devices1
      end
```

```
  let ghost function bus_sort_breadthfirst(dev_list : devList  ): devList
      ensures { sorted result /\ permut result dev_list }
      =
          dev_sort dev_list Nil
end (*Bus_sort *)
```

Приложение 2

Формулы корректности программы Bus_sort

```
goal dev_insert'vc :
  forall ah:device, list1:list device, y:list device, x:list device.
   (sorted list1 /\ list1 = (y ++ x)) /\
   (forall d:device. mem d y -> le d ah) ->
   match x with
   | Nil -> true
   | Cons xh xt ->
      not compare ah xh <= 0 ->
      (let o = y ++ Cons xh (Nil: list device) in
       (0 <= length x /\ length xt < length x) /\
       (sorted list1 /\ list1 = (o ++ xt)) &&
       (forall d:device. mem d o -> le d ah))
   end /\
   (forall result:list device.
     match x with
     | Nil -> result = (list1 ++ Cons ah (Nil: list device))
     | Cons xh xt ->
        if compare ah xh <= 0 then result = (y ++ Cons ah x)
        else sorted result /\
             permut result ((y ++ Cons xh (Nil: list device)) ++ Cons ah xt)
     end -> sorted result /\ permut result (y ++ Cons ah x))

goal dev_sort'vc :
  forall x:list device, sorted_devices:list device.
   sorted sorted_devices ->
   match x with
   | Nil -> true
   | Cons xh xt ->
      let o = (Nil: list device) in
      ((sorted sorted_devices /\ sorted_devices = (o ++ sorted_devices)) &&
       (forall d:device. mem d o -> le d xh)) /\
      (let sorted_devices1 =
         dev_insert xh sorted_devices o sorted_devices
       in
       sorted sorted_devices1 /\
       permut sorted_devices1 (o ++ Cons xh sorted_devices) ->
```

```
      sorted sorted_devices1)
  end /\
  (forall result:list device.
    match x with
    | Nil -> result = sorted_devices
    | Cons xh xt ->
      let o = (Nil: list device) in
      let sorted_devices1 =
        dev_insert xh sorted_devices o sorted_devices
      in
      (sorted sorted_devices1 /\
       permut sorted_devices1 (o ++ Cons xh sorted_devices)) /\
      sorted result /\ permut result (xt ++ sorted_devices1)
    end -> sorted result /\ permut result (x ++ sorted_devices))

goal bus_sort_breadthfirst'vc :
  forall dev_list:list device.
   let o = (Nil: list device) in
   sorted o /\
   (let result = dev_sort dev_list o in
    sorted result /\ permut result (dev_list ++ o) ->
    sorted result /\ permut result dev_list)
```

УДК 004.65

Последовательность как абстракция структурированного построения баз данных

Марчук А.Г. (Институт систем информатики СО РАН, Новосибирский государственный университет)

В статье представлена модель баз данных, основанная на последовательности объектов. Рассмотрены вопросы сериализации/десериализации объектов, индексных построений, структуры и методов универсальной последовательности и универсального индекса, реализация полного базиса редактирования через использование первичного ключа, временной отметки и «пустого» значения.

Ключевые слова: *база данных, последовательность, индексное построение.*

1. Введение

Классическим моделям, связанным с базами данных уже более 50 лет. В информационных технологиях закрепились реляционные базы данных, построенные на реляционной алгебре [1]. Однако, в последние два десятка лет активно развиваются и альтернативные, точнее дополняющие подходы, получившие обобщающее название NoSQL, которое чаще интерпретируют не как отрицание SQL, а как Not Only SQL [2]. Сейчас уже твердо вошли в практику, получили свою классификацию такие подходы, как документные базы данных, хранилища ключ-значение, семейства столбцов, графовые СУБД.

Предлагается модель структурного объекта, способного быть основой для разных построений баз данных. В принципе, это обобщение реляционных таблиц на случаи, когда не обязателен фиксированный набор колонок и скалярный тип этих колонок. Существенной частью модели является введение темпоральности, эквивалентности элементов, индексных построений. Это позволяет решать более широкий круг задач, чем в рамках табличного подхода. Кроме того, модель ориентируется на хранение/обработку больших данных, в частности на распределение базы данных по узлам, экономное решение задач избыточного хранения и восстановления данных, фиксации промежуточных состояний и откатов к предыдущим состояниям. Модель реализована в рамках системы PolarDB как библиотека классов в среде .NET и C#.

2. Что такое последовательность

Последовательность – это упорядоченный, конечный (ноль или более), растущий набор элементов:

```
e0, e1, e2, …, eN
ei – объекты
```

Частный случай последовательности, это Stream – растущая последовательность байтов обычно фигурирующая в процессах ввода/вывода. В общем случае элементы последовательности изменяемые, последовательность может иметь ноль элементов. В любой момент времени последовательность конечная. Последовательность может прирастать «вправо» через добавление элементов. Не предполагается, что можно добавлять элементы «слева» и исключать элементы из последовательности.

Элементы последовательности это структурные значения. Структурные значения – в данном случае это значения, выполненные в базисе построений той или иной системы программирования, включают в себя значения примитивных типов, таких как булевские, байты, символы, числа и составные значения. Мы ориентируемся на рекуррентное и иерархическое построение значений. В качестве примеров можно привести базис современной объектно-ориентированной системы программирования, напр. .NET, Java. Более специализированными построениями напр. являются структурный базис JSON (значения, записи, массивы) и DOM – внутреннее представление объектов XML. Также важным использованным классом является структуризация системы Polar [3].

Нас будут интересовать «большие» последовательности. Большие последовательности (или в общем случае, большие данные) в нашем случае, это такие информационные образования, которые невозможно или нерационально располагать в оперативной памяти. А это, в частности означает, что должна существовать форма представления последовательностей для оперативной работы с ними. Таких форм предлагается две: в виде потоков объектов (генерация потока, фильтрация, функциональное преобразование, группирование, редукция, использование) в функциональном слое программирования, напр. LINQ в C#, Scala или Flink в Java и в виде отображения последовательности объектов на устройства внешней памяти. В примерах потоковая обработка будет иллюстрирована средствами C# и LINQ [4]. А отображение на внешнюю память реализуется через сериализацию/десериализацию.

3. Сериализация и десериализация последовательностей

Сериализация кодирует структурное значение последовательностью алфавита кодирования, десериализация делает обратное преобразование последовательности символов кодирования в объектное представление. Алфавит кодирования может быть битами, байтами, символами.

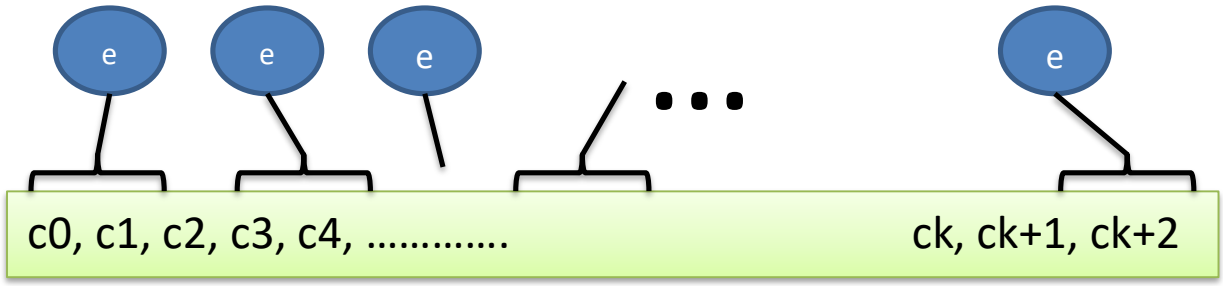


Рис. 1. Отображение последовательности элементов на цепочку символов и обратно

На рисунке 1 иллюстрируется отображение последовательности объектов (овалы сверху) на цепочку символов какого-то алфавита. Элементы основной последовательности преобразуются в цепочки символов этого алфавита, цепочки укладываются в результирующую последовательность подряд, возможно, через разделители. Число символов в цепочке, соответствующих одному элементу основной последовательности – произвольное. Обратный процесс является десериализацией.

Хорошая сериализация допускает десериализацию с получением эквивалентной последовательности. Эквивалентность последовательностей и значений определяется рекуррентно.

В общем случае, десериализация требует синтаксического анализа. Также заметим, что последовательность может рассматриваться как единый объект – массив объектов, обычно сериализация последовательности выполняется по тем же правилам, что и сериализация элементов.

3.1 Текстовая сериализация

Текстовая сериализация отображает объект или последовательность объектов. Алфавитом служат символы. Наиболее часто используемая кодировка символов – юникод. Обычно, речь идет о кодировании структурных значений, иногда довольно специфичного вида. Для кодирования структурных данных используется инфиксная рекуррентная форма вида: открывающая скобка – значения, перечисленные через разделитель, закрывающая скобка. К этому виду текстового изображения структурных значений относятся языки разметка, в

первую очередь – XML, JSON и некоторые другие. Есть специализированные построения подобных текстовых структур, напр. LISP.

```
<db>
  …
  <person>
    <firstName>Иван</firstName>
    <lastName>Иванов</lastName>
    <address>
      <streetAddress>
        Московское ш., 101, кв.101
      </streetAddress>
      <city>Ленинград</city>
      <postalCode> 101101
      </postalCode>
    </address>
    <phoneNumbers>
      <n> 812 123-1234 </n>
      <n> 916 123-4567 </n>
    </phoneNumbers>
  </person>
  …
</db>
```

```
[ …
  {   "firstName": "Иван",
      "lastName": "Иванов",
      "address":
      {
          "streetAddress": "Московское
ш., 101, кв.101",
          "city": "Ленинград",
          "postalCode": 101101
       },
      "phoneNumbers": [ "812 123-
1234", "916 123-4567" ]
  }  …
]
```

Таблица 1. Образцы текстовых сериализаций в формате XML и JSON

Текстовая развертка (сериализация) структурных значений используется для случаев, когда нужно не только сохранить значение объекта или объектов, но и иметь его доступным для изучения и редактирования. Для целей среды сохранения данных в базах данных, такой способ неудобен по двум причинам: неэкономное использование байтового пространства и, главное, потребность в синтаксическом анализе при вводе (десериализации).

В ячейках таблицы 1 приведены примеры текстов XML и JSON, показывающих как выглядят записи условной базы данных. Видно, что JSON выигрывает и в компактности и в наглядности.

3.2 Байтовая сериализация

Для баз данных, наиболее естественной и эффективной, является байтовая сериализация. Обычно используется префиксная (польская) запись структуры: указание вида структуры или структуризатора, количество аргументов (если надо), подряд стоящие изображения аргументов. Такой подход годится как для общего безтипового случая, так и для типизированной записи значений. Основа бинарного байтового отображения структур зафиксирована в наборах действий с байтовыми потоками (Stream) в любой развитой системе программирования через класс BinaryWriter и класса чтения BinaryReader. Напр.

```
BinaryWriter bw = new BinaryWriter(file);
bw.Write(false);
```

```
bw.Write((byte)121);
bw.Write(999);
bw.Write(“Sample string”);
```

Чтение с того же места той же последовательности типов значений позволяет получить значения, эквивалентные тем, которые записывались, Введение в эту схему рекуррентного применения записи в случае массива объектов требует лишь дополнительной записи количества элементов:

```
bw.Write(nelements); bw.Write(el1); bw.Write(el2); …
```

Проблема в том, что если мы не знаем типа читаемого элемента, мы можем прочитать «не то». Соответственно, для безтиповой последовательности, запись каждого значения должна предваряться записью варианта его типа.

```
Object x =…
x is bool      t b
x is byte      t b
x is int       t bbbb
x is string    t bssssss
x is Object[]  t bbbbbbbb (e0) (e1) …
```

t – номер варианта (0-bool, 1-byte, 2-int, 3-string, 4-array

b – байт значения

s – байт значения строки

e – развертка элемента массива (скобки для подчеркивания рекуррентного раскрытия элементов.

Сериализация для типизованных значений строится по этой же схеме. Только отсутствует колонка t – номеров вариантов. Это потому, что имеющаяся внешняя информация о типах определяет номер варианта. Соответственно, при раскрытии элементов, их тип вычисляется и снова не требует фиксации номера.

3.3 Битовая сериализация

При такой сериализации осуществлется отображение объектов на цепочки битов. В принципе, идейная часть сериализации аналогична байтовой. Существенная разница в том, что координаты позиции измеряются в числе битов от начала последовательности и, кроме того, надо активно использовать теорию кодирования для достижения оптимальных решений. Рассмотрим пример (в синтаксисе языка Поляр):

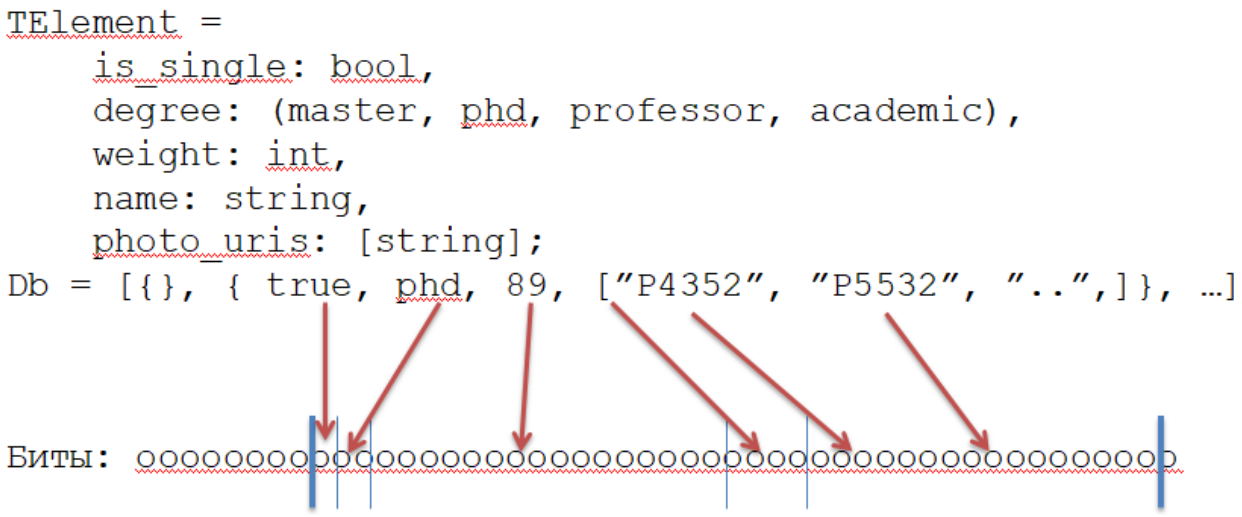


Рис. 2. Пример отображения полей записи на битовую последовательность

На рисунке 2 отдельные биты обозначены кружочками и условно демонстрируются следующие эффекты битовой сериализации: а) булевское значение можно отображать в 1 бит; 2) перечислимое значение второго поля отображаются в минимальное количество бит, зависящее от числа вариантов; 3) целочисленное значение динамически кодируется так, чтобы минимизировать применяемое число бит; 4) то же самое касается указания количества элементов в подпоследовательности.

3.4 Позиции элементов в последовательности

Как уже отмечалось, (большая) последовательность существует либо в виде динамического потока элементов, либо в виде сериализации. А фактически, после первичной загрузки данных – в виде сериализации. Позицией элемента в сериализации назовем число байтов (битов, в случае битовой сериализации) от начала последовательности до начала элемента. Пара сериализация-десериализация должна строиться так, чтобы элементы корректно прочитывались если правильно указана позиция элементов и тип, в случае типизированной последовательности. Для текстовой сериализации можно было бы вести подсчет символов от начала, но практичнее – количество байтов, поскольку позицию можно использовать для «прямого» доступа к элементу последовательности (через метод seek или свойство .Position, определенных для класса Stream). Для битовой развертки последовательности, использование позиции несколько усложняется, хотя принципиально такое же.

В дальнейшем, мы будем говорить об элементе последовательности как о паре: (элемент, позиция). Более того, элемент последовательности теперь определен через позицию. По

позиции из сериализации прочитывается элемент и снова получается пара. Введем несколько обозначений. Сериализации будем обозначать **S**, элементы последовательности **e**, позицию конкретного элемента **e** как e_{pos}**.** Просто позицию обозначим **p**, через позицию вычисляется сам элемент **e = S(p)**.

Позиции можно использовать в различных информационных построениях. Например, если сформировать массив позиций **arr**, то **i**-ый элемент последовательности будет **S(arr[i])**.

Стоит отметить, что позиция (в нашем случае) реализуется длинным целым. То есть, значением с фиксированной длиной в сериализации. Соответственно, для реализации прямого доступа к элементам последовательности (по индексу) можно пользоваться не только массивом, но и последовательностью позиций, обозначим последовательности позиций через **P**. Тогда по индексу **ind** можно найти позицию в **P** как

```
offset = число_байтов_в_начале + 8 * ind
(8 – число байтов в целом двойной точности)
p = P(offset)
e = S(p)
```

4. Индексное построение

Рассмотрим задачу поиска элемента в последовательности. Точнее, элементов, удовлетворяющих какому-то критерию. Пусть **X** – множество элементов последовательности, поиск по критерию:

```
{ x ∈ X | C(x) }, параметрический вариант { x ∈ X | Q(x, y) }
```

В общем случае, задача решается перебором элементов с проверкой критерия на каждом. В частных случаях, можно выстроить дополнительную информационную структуру, которая сделает поиск более экономным. Эту структуру будем называть индексным построением (database index).

4.1 Ключевой индекс

Пусть на элементах **X** определен функционал **F(x)**, а критерием будет **Q(x, y) : F(x) = y**. Тогда достаточно для каждого элемента из **X**, создать пару **(F(x),** x_{pos}**)**. Множество всех таких пар отсортируем по первому значению. Тогда поиск можно выполнять как бинарный поиск во множестве пар, по найденным парам вычисляем элементы-результаты поиска. На рисунке 3 приведена иллюстрация формирования и использования ключевого индекса для поиска по заданному значению.

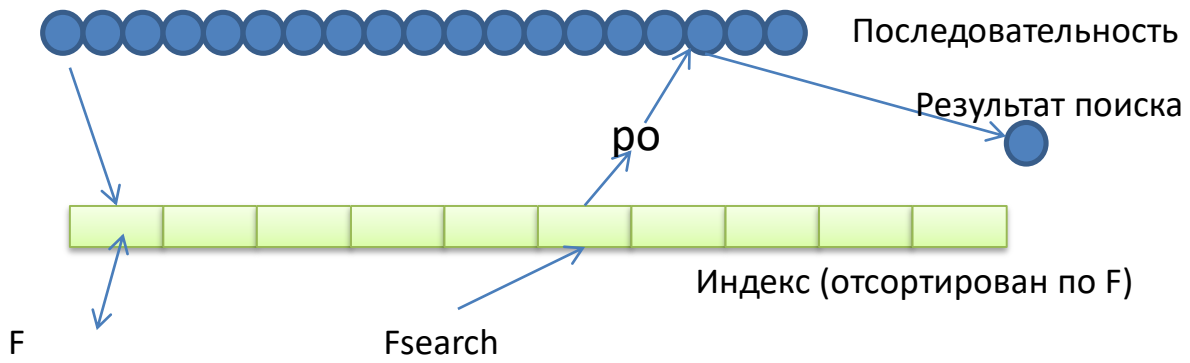


Рис. 3. Ключевой индекс и его использование для поиска по значению

Из последовательности выделяются позиции элементов, формируется последовательность пар, далее, эта последовательность сортируется по первому значению пары, индекс построен. Использование индекса выполняется через бинарный поиск всех пар, в которых первое значение совпадает с образцом F_{search}. Далее, из выявленных элементов берутся позиции и по ним, получаются результаты поиска. Функционал `F(x)` называется в этом случае ключевой функцией.

Такая схема не получается в случае, если тип значений ключевой функции не дает фиксированного размера в сериализации. Например, часто встречающаяся ключевая функция – строковый идентификатор именно такой. Схема изменяется на более простую. В индекс помещаются только позиции элементов. При этом сортировка ведется на элементах, вычисленных по позиции. Соответственно изменяется процедура поиска, в которой бинарный поиск выполняется также на вычисленных через позицию элементах. Такая косвенность заметно «утяжеляет» и построение индекса и поиск по индексу.

4.2 Использование компаратора

Главное в сортировочном индексе, это упорядочение элементов в соответствии с отношением порядка, заданного на элементах. Для объектов, достаточно общей случай отношения порядка задается компаратором. Компаратор, это функция, определенная на множестве значений элементов последовательности такая, что

```
int Compare(object a, object b)
```

Значения компаратора: -1, 0, 1, соответствующие разным вариантам отношения, кроме того, должны удовлетворяться метрические аксиомы, напр. транзитивность.

Использование компаратора выполняется следующим образом. Для подготовки индекса по сериализованной последовательности строится массив позиций, массив сортируется с

использованием компаратора. При использовании формируется элемент-образец `sample`, далее бинарным поиском находятся все элементы последовательности, для которых

```
Compare(sample, x) = 0
```

эти элементы составляют решение.

Имеется некоторое неудобство проведения поиска через компаратор: это необходимость критерий поиска превращать в целостный элемент. Как правило, это не вызывает проблем и эффективность поиска по-прежнему остается высокой.

4.3 Ускорители для индексов

Что из себя представляют индексы? Логически, индексы, это последовательности позиций, или пар (позиция, значение). Поскольку последовательности предполагаются большими, то и индексы будут большими. А значит, их следует разворачивать также в сериализации таких значений. Позиции (длинные целые) – значения одинакового размера (8 байтов), если добавляется числовое значение, то такая запись также фиксированного размера. Это позволяет вычислять позицию элемента во вспомогательной индексной сериализации по его номеру и полноценно использовать элементы для поисков по значению или по компаратору.

Иногда желательно усложнить индексное построение для того, чтобы ускорить поиск элементов. На уровне объектной абстракции, предлагается два варианта построения ускорительных дополнительных построений.

Прореженный массив

Предположим, есть индексный массив (сериализация) уже упорядоченный в соответствии с критерием данного индекса, т.е. либо в соответствии с компаратором, либо в соответствии с частным случаем компаратора – ключевой функцией. Тогда сделаем выборку значений из массива с равномерным шагом, как это изображено на рисунке 4.

Рис. 4. Разреженная выборка значений из отсортированного массива позиций

Выбираются значения, участвующие в упорядочивании по заданному критерию. Если в массиве находятся только позиции, как в случае компараторного индекса, то по смещениям прочитываются элементы основной последовательности. При наличии построенного

дополнительного массива, решение основной задачи – поиску по образцу, производится сначала в дополнительном массиве, который отсортирован оп построению, а потом в поддиапазоне основного индекса.

Существует интересный вариант формирования прореженного массива. Пусть число элементов индексного построения являются степенью двойки и пусть отсортированные элементы индексного построения p_i переупорядочиваются в массив q_j таким образом, что новый индекс **j** является битовой инверсией исходного индекса **i**. То есть, если **i** состоит из битов **b0, b1, b2, … bk**, то инверсный индекс будет **bk, bk-1, …, b0**. В этом построении, элемент q_0 будет соответствовать точке **N/2**, т.е. середине массива. Следующие два элемента будут соответствовать серединам первой половины и второй половины и т.д. То есть, эти начальные, также как и следующие точки, будут повторять логику двоичного поиска. Если **K**, степень двойки, элементов инверсного массива построить отдельно, это будет прореженный массив с фиксированным интервалом между элементами.

Создание шкалы значений

Другой способ реализуется для числовых ключей, значения которых более или менее распределяются в диапазоне **[min, max]**. Этот диапазон разбивается на интервалы равномерным шагом по значению и фиксируются номера элементов массива индексных значений, соответствующие началам интервалов.

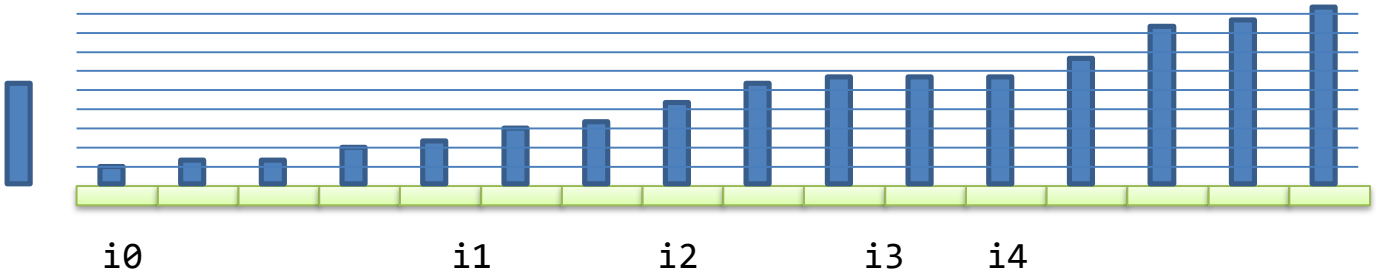


Рис. 5. Шкалирование упорядоченных числовых значений

Теперь, когда появляется ключевое значение (левый столбик), поисковая процедура определяет какому интервалу это значение принадлежит и границы интервала становятся интервалом поиска группы значений, имеющих тот же ключ.

4.4 Эффективность использования индексного построения

Рассмотрим производительность нашего построения последовательности с заданным критерием и построенным индексом. Пусть есть случайный набор ключевых значений или поисковых образов. Нас будет интересовать средняя скорость выполнения выборки по критерию.

Сначала обратим внимание собственно на сериализацию последовательности. Предположим, мы таинственным образом знаем позиции элементов, которые надо найти. Остается только прочитать (десериализовать) значение. Эту ситуацию можно промоделировать. Для этого запишем в файл с помощью BinaryWriter массив длинных целых или чего-то аналогичного. А потом будем делать выборку этих значений по случайному номеру по позициям, которые легко вычисляются.

Массив заполняется довольно быстро: 100 млн. значений загружаются около 2.5 сек. Прочтение всех записанных величин через бинарное чтение «подряд» выполняется приблизительно за то же самое время. А прочтение элементов по случайно заданному номеру, выполняется 7 миллисекунд на 1000 чтений. И эта скорость сохранится если мы отключим заполнение файла поскольку он уже сформирован. Однако, если теперь мы перезагрузим компьютер и снова запустим тест скорости случайных выборок, то результат может обескуражить: будет около 100 выборок в секунду. Это из-за того, что современные операционные системы кешируют в оперативной памяти не только страницы программы, но и страницы файлов. А это означает, что если кеш не работает (не успел накопиться), то доступ к диску будет выполняться в соответствии с характеристиками устройства (7200 rsp для «стандартного»), т.е. порядка 100 выборок в секунду. Использование SSD несколько ускорит этот процесс, но качественно не изменит.

Таким образом, без кеша скорость доступ порядка 100 выборок в сек., с кешем – порядка 100 тыс. выборок в сек., для массива – это порядка 100 млн. выборок в сек.

Для больших данных ситуация с замедлением усугубляется. Если данные не помещаются в оперативную память, а поток запросов равномерен по пространству номеров, то не существует разумной политики страничного кеширования, которая обеспечит высокую (100 тыс.) скорость работы.

В любом случае, в алгоритмах выборки значений по критерию важнейшим фактором является количество чтений из больших массивов, отнесенное на одну выборку. Обычная схема алгоритма следующая: бинарный поиск в индексном массиве и выдача найденных значений. Если в индексном массиве записано ключевое значение элемента, то общее количество выборок **`lg2(N) + 1`**. Если в индексном массиве позиции, тогда для проверки значения выполняется одна выборка из индексного массива, это позиция, и одна выборка из основной последовательности. Общее количество выборок **`2xlg2(N)`**, где **`N`** – число элементов в последовательности.

Рассмотренные ускорители создают дополнительные к базовому отсортированному массиву структуры, но в отличие от основного индекса, ускоритель может иметь разный

размер. Ускорители могут быть реализованы в виде сериализации последовательности или в виде массива в оперативной памяти. Поиск производится сначала в ускорителе. Для прореженного массива поиск выполняется за **`lg2(K)`** шагов, где **`K`** – его размер. Для шкалы поиск в шкале выполняется за одну операцию. Таким образом, выигрыш от ускорителя, реализованного в виде сериализованной последовательности, будет только в варианте шкалы.

Если ускоритель реализуется в виде массива в оперативной памяти, выигрыш будет при любом ускорителе. Эффект он него будет зависеть от выделенного ресурса ОЗУ. Поскольку работа с объектами в оперативной памяти производится существенно быстрее работы с сериализованными объектами, время поиска можно оценить как **`lg2(N/K)`** или **`lg2(N) - lg2(K)`** выборок. Например, для большой последовательности напр. в 1 млрд. элементов число поисковых выборок из индексного массива может быть более 30. Наличие в ОЗУ массива в 1 млн. ускорительных значений может уменьшить это количество до порядка 10. При этом, использование ресурса ОЗУ может оказаться разумным.

Схема с ускорителем, размещаемым в ОЗУ, имеет недостатком не только использование дефицитного ресурса ОЗУ, но и то, что требуется время для загрузки вспомогательного массива. Загрузка существенно ускорится, если прореженный массив будет формироваться из инверсного массива (см. ранее). Точки прореженного массива будут находится рядом и подряд. Для чтения массива из одной (серединной) точки, нужно прочитать первое значение. Прочитав два следующих, прореженный массив будет из трех точек, следующие 4 значения снова его увеличивают и т.д. При этом, расположение точек «подряд» существенно ускоряет их чтение из файла.

5. Объектная модель последовательности

Создадим универсальную последовательность объектов сначала на тех построениях, которые были сделаны ранее по тексту статьи. В данном разделе мы будем придерживаться моделей и синтаксисом языка программирования C#, хотя все подходит и для других средств.

5.1 Структуризация данных

Используемые в построениях структуры данных традиционны для многих языков программирования и реализованы в большинстве систем программирования. Объектом являются булевское, байт, символ, различные числа, строки символов. Также объектом

является массив объектов. Последнее порождает рекуррентное построение, позволяющее моделировать широкий спектр иерархических структуризаций.

Типом данных назовем структурное построение, задающее кодирование/декодирование структурного значения в некотором базисе. В нашем случае, тип будет определять бинарную байтовую сериализацию/десериализацию структурных значений.

Здесь мы будем придерживаться типовой системы языка программирования Поляр [3, 5]. Типизованное значение представляет собой

```
- none (тип, с пустым множеством значений)
- bool
- byte
- char
- int
- long
- double
- запись
- массив
- объединение
```

Последние три – композиционные типы, **`запись`** – это набор полей заданных типов, **`массив`** – это набор элементов заданного типа, **`объединение`** – значение одного из заданных вариантов типов.

При сериализации тип none переходит в 0 байтов, типы `bool` и `byte` – 1 байт, `char` – 2 байта, `int` – 4 байта, `long` – 8 байтов, `double` – 8 байтов. Запись переходит в подряд стоящие сериализации полей, массив переходит в длинное целое, определяющее длину последовательности `N`, а потом `N` значений определенного типа. Объединение представляется номером текущего варианта (1 байт) из заданных вариантов типа, далее идет значение этого варианта.

Ранее мы допускали отсутствие заданного типа, которое при сериализации интерпретировалось как некоторый код-номер, а далее сериализация значения. Такое отсутствие типа можно трактовать как «автоматический» или «динамический» тип. Можно определить и этот («безтиповый») тип в нотации Поляра:

```
Dyna = isnull^none,
    Bool, Byte, Char, Int, Long, Double^none,
    Arr^Dyna;
```

Конкретные типовые определения будем предполагать экземплярами класса PType. Заметим, что общая объектная конструкция не сильно зависит от применяемой системы типов, поскольку конструкция объекта зафиксирована, а отображение значения на байты важно лишь обратимостью операции сериализация – десериализация.

5.2 Хранилище потоков

Еще одной составной частью объектной модели последовательности является хранилище потоков. Его задача – предоставлять байтовые потоки (Stream) для использования в модели. Потоки байтов требуются во всех построениях, где фигурирует сериализация.

Хранилище потоков может быть простым типа директории файловой системы с множеством файлов-потоков (FileStream). Но может быть и довольно сложным, в котором оптимизируется скорость доступа к потокам и кеширование, создается система контрольных сумм и резервного копирования.

Например, в библиотеке PolarDB [5] хранилище байтовых потоков формируется на основе специально разработанного классы PagedStream, расширяющего класс Stream, реализующего «свою» страничную память и псевдофайловую систему, похожую на первые варианты ОС Unix. Все потоки укладываются в страницы, на который разделен опорный файл, страницы имеют свою систему кеширования, возможно использование резервного копирования.

5.3 Универсальная последовательность

Сформируем универсальную последовательность объектов по схеме, изложенной ранее. Типизованная или безтиповая последовательность объектов сериализуется в бинарный байтовый поток (стрим). В стриме сохраняются: количество элементов в последовательности и все элементы в виде сериализаций. В силу обратимости сериализации через десериализацию, полученный стрим можно снова развернуть в поток объектов. Также для универсальной последовательности определен метод добавления единичного объекта. Этот метод влияет не только на «концовку» стрима, но и на начало, где корректируется текущая длина. Общая схема класса следующая:

```
public class UniversalSequence
{
    public UniversalSequence(PType tp_elem, Storage store) { … }
    public void Load(IEnumerable<object> flow) { … }
    public void AppendElement(object element) { … }
    public IEnumerable<object> Elements() { … }
}
```

Такая простая конструкция (имеется некоторое упрощение объектной модели) позволяет организовать довольно сложные построения с индексами и операциями редактирования последовательности. Это будет показано дальше.

5.4 Универсальный индекс

Охватить возможное множество индексных построений вряд ил возможно. В данной модели предлагается один гибкий вариант. В принципе, что нужно? Нужно выстроить массив позиций в соответствии с заданной упорядоченностью. В дальнейшем, этот массив используется для бинарного поиска по этой функции. Соответственно, класс индекса должен содержать ссылку на хранилище, чтобы сериализовать свои построения, ссылку на последовательность, назовем ее опорной, для которой этот индекс будет создаваться, функцию компаратора, задавать упорядочивание. Кроме того, индекс должен быть построен, но также должен и изменяться с изменением (ростом) опорной последовательности. Еще одни нюанс: индекс не обязан быть задан на всех элементах опорной последовательности, это можно регулировать функцией применимости. Удовлетворив требованиям, мы получим что-то вроде:

```
public class UniversalIndex
{
    public UniversalIndex(PType tp_elem, Storage store,
      UniversalSequence tab, Func<object, bool> applicable,
      Comparer<object> comp, Func<object, int> keyFun)
    { … }
    internal void OnBuild() { … }
    internal void OnAppendElement(object element, long pos) { … }
    public IEnumerable<object> Get(object sample) { … }
}
```

Некоторая дополнительная особенность класса в том, что кроме компаратора, мы объявляем ключевую функцию keyFun. Логика здесь такая, что упорядочивание осуществляется сначала по ключевой функции, а при одинаковости значений функции по компаратору. Часто ключевая функция играет роль хеш-функции. Допустимо, чтобы ключевая функция может быть не определена (**=null**) и наоборот, компаратор также может быть не определен.

Индекс работает в связке с опорной последовательностью. Опорная последовательность заполняется, потом строится (добавляем метод Build()), при построении строятся также индексы через запуски индексных методов OnBuild(). При построении индекса происходит формирование массива позиций и, возможно, ключевых значений, массив сортируется по указанным в индексе критериям и обслуживает запросы Get. В случае выполнения в последовательности метода AppendElement(e), запускается обработчик OnAppendElement в каждом из индексов.

5.5 Растущая модель

Сформированная в предыдущем разделе модель индекса, при прямолинейной реализации, не позволяет эффективно выполнять добавление элемента, поскольку при каждом добавлении нужно снова строить, а значит – сортировать индексный массив. Также пока нет механизма редактирования, т.е. не только добавления, но и уничтожения и изменения элементов в последовательности. Требуется более сложная модель индекса!

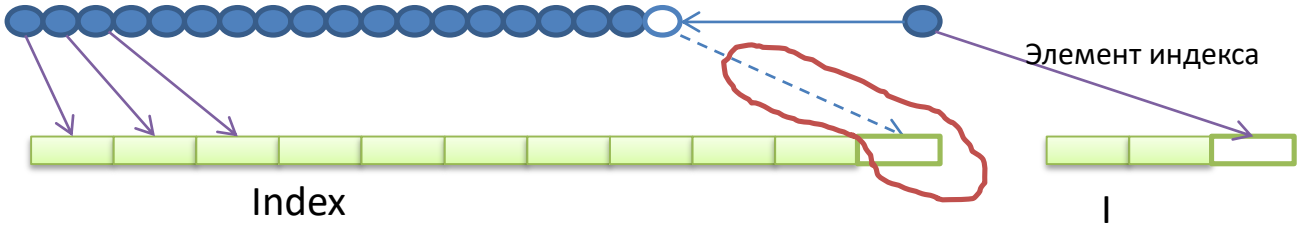


Рис. 6. Разбиение индексного массива на два

На рисунке 6 имеется иллюстрация усложненного устройства индекса на примере выполнения операции добавления элемента. Элемент (синий кружок справа) добавляется к последовательности через штатный метод AppendElement(e). Почему старая схема строения индексного построения не подходит? Потому что добавление в индексный массив (пунктирная стрелка) должно приводить к пересортировке всего индексного массива, что слишком затратно. Предлагается внести изменения в индексное построение тем, что индексное построение сделать из двух массивов: левого, большого, который при отработке индексного метода OnAppendElement не меняется и правого, маленького, в который идет добавление позиции и, возможно ключевого значения, с пересортировкой, что для малых количеств добавленных элементов не представляет проблемы.

Теперь рассмотрим задачи добавления объекта и редактирования объекта. В этом случае для начала, нужно определить способ указания какой именно элемент подвергается изменению.

Один из индексов можно определить как первичный ключ (primary key). Это выделение дает возможность произвести факторизацию по первичному ключу. То есть, элементы, совпадающие по первичному ключу объявляются эквивалентными. Среди эквивалентных, по некоторому критерию, назначается оригинал. Используемый критерий: оригиналом является элемент у которого было более позднее попадание в последовательность! Реализация критерия выполняется или через позиции эквивалентных элементов в последовательности или через добавление временной отметки.

Свойства первичного ключа: первичный ключ дает уникальное значение (по построению);

первичный ключ, как правило, реализуется через ключевое значение (число или строка).

Теперь для выборки можно добавить (и эффективно реализовать) метод единичной выборки по ключу

```
object element = GetByKey(object key);
```

Вернемся к задаче редактирования базы данных. Изменение элемента выполняется добавлением другого значения элемента с тем же первичным ключом. Уничтожение элемента выполняется через изменение на «пустое» значение.

6. Заключение

В статье представлена модель последовательности объектов, разработанная для использования в универсальных и специализированных построениях баз данных и систем управления базами данных.

Модель опирается на: сериализацию объектов; выделение позиции элемента как способа прямой работы с элементами; построение специальной конструкции индексов для реализации эффективной схемы отслеживания добавления элементов; назначение первичного ключа среди определенных индексов; введение на этой основе эквивалентности элементов и средств выявления оригинала, напр. через временную отметку; определение пустого элемента с заданным первичным ключом; реализацию полного набора операций редактирования через добавление элементов.

Модель обобщает классические реляционные таблицы, key-value хранилища, а также ряд NoSQL подходов таких как MondoDB, Cassandra, BigTable и др.

Модель была реализована в библиотеке PolarDB [5]. Эта реализация была использована в полупромышленных проектах фактографических систем ИСИ [6].

Список литературы

1. Дейт К. Дж. Введение в системы баз данных = Introduction to Database Systems. — 8-е изд. — М.: Вильямс, 2005. — 1328 с. — ISBN 5-8459-0788-8 (рус.) 0-321-19784-4 (англ.).
2. Мартин Фаулер, Прамодкумар Дж. Садаладж. NoSQL: новая методология разработки нереляционных баз данных = NoSQL Distilled. — М.: «Вильямс», 2013. — 192 с. — ISBN 978-5-8459-1829-1.
3. Марчук А.Г., Лельчук Т.И. Язык программирования Поляр: описание, использование, реализация, Новосибирск, 1986, 96 с.

4. Синтаксис LINQ = https://docs.microsoft.com/ru-ru/dotnet/csharp/linq/
5. Марчук А.Г Архитектура и основные особенности библиотеки PolarDB работы со структурированными данными // Системная информатика, № 13, 2018. Стр. 25-34
6. А.Г.Марчук, С.В.Лештаев Электронный архив газет: Web-публикация, ассоциация информации с базой данных, создание полнотекстового поиска // Аналитика и управление данными в областях с интенсивным использованием данных, XVIII Международная конференция DAMDID/RCDL'2016, Ершово, Московская обл., Россия, 11-14 октября 2016 года, Труды конференции. Торус пресс, Москва, 2016. Сс. 155-160.
7. А.Г.Марчук, П.А.Марчук Платформа реализации электронных архивов данных и документов // Электронные библиотеки: перспективные методы и технологии, электронные коллекции: Труды XIV Всероссийской научной конференции RCDL'2012. Переславль-Залесский, Россия, 15-18 октября 2012 г. – г. Переславль-Залесский: изд-во «Университет города Переславля», 2012, С. 332-338.

УДК 004.05

Сравнение технологий автоматного программирования и Event-B

Шелехов В.И. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Показано, что спецификация на языке Event-B представима автоматной программой в виде недетерминировнной композиции простых условных операторов, что соответствует узкому подклассу автоматных программ. Спецификация в Event-B является монолитной. Для построения спецификации нет других средств композиции, кроме уточнения, реализующего расширение ранее построенной спецификации.

Сравнение технологий автоматного программирования и Event-B проводится на примере двух задач. Предыдущие решения задачи управления движением на мосту в системе Event-B являются сложными и громоздкими. Предложено более простое решение с верификацией программы в инструменте Rodin. Эффективность методов верификации в Event-B подтверждена нахождением трех нетривиальных ошибок в нашем решении.

Ключевые слова: *автоматное программирование, Event-B, уточнение (refinement), требования, дедуктивная верификация, трансформации программ, функциональное программирование, предикатное программирование.*

1. Введение

Event-B [10] – это метод формальной спецификации и верификации систем в программной и системной инженерии, успешно используемый при разработке производственных систем управления, особенно в железнодорожном транспорте. Сила метода Event-B в том, что этот метод гарантирует обнаружение многих серьезных ошибок с помощью доказательства теорем, что нереализуемо другими известными методами. Несмотря на большое число разнообразных приложений, существует ряд факторов, сдерживающих широкое внедрение подхода Event-B, что отмечается в обзоре [11].

Автоматное программирование [1, 2, 5, 6, 9] ориентировано на класс реактивных систем, реализующих взаимодействие с внешним окружением системы и реагирующих на определенный набор событий (сообщений). Автоматная программа определяет конечный автомат в виде гиперграфовой композиции сегментов кода. Технология автоматного

программирования предлагает комплекс методов для разработки, оптимизации и верификации автоматных программ.

В настоящей работе проводится сравнение технологий автоматного программирования и Event-B. Определена модель спецификации на языке Event-B в языке автоматного программирования. Спецификация Event-B отображается в недетерминированную композицию простых условных операторов. Таким образом, язык Event-B оказывается узким подъязыком автоматного программирования, в котором уточнение (refinement) – единственное средство декомпозиции спецификаций. Тогда как автоматное программирование кроме недетерминированной композиции допускает параллельную композицию процессов, гиперграфовую композицию по управляющим состояниям, подпроцессы, объектно-ориентированную и аспектно-ориентированную композиции.

Сопоставление технологий проводится также на двух примерах, которые в руководствах по Event-B приводятся для обучения технологии Event-B. Для этих примеров построены автоматные программы, которые затем были закодированы в виде спецификации Event-B и верифицированы в инструменте Rodin [22].

Во втором разделе настоящей статьи дается описание языка и технологии автоматного программирования. В третьем разделе описывается подход Event-B и его моделирование в автоматном программировании. В четвертом разделе технологии Event-B и автоматного программирования иллюстрируются на примере управления движением на перекрестке. В следующем разделе для примера управления движением автомобилей на мосту иллюстрируются уточнения, применяемые в Event-B. Детально описывается разработка автоматной программы, исправляющей недостатки реализации в Event-B. Обзор работ в шестом разделе. В заключении основные выводы по сравнению технологий. В Приложении 1 представлена спецификация на языке Event-B автоматной программы управления движением на мосту.

2. Автоматное программирование

2.1. Классы программ

Методы программной инженерии, доказавшие свою эффективность, не всегда успешно применимы для всех программ. Причина здесь в различиях архитектур программ. Это ставит задачу классификации программ, то есть построения системы классов программ и разработку адекватной технологии программирования для каждого класса программ. Теория

программ каждого класса должна определять методы спецификации, верификации, моделирования и эффективной реализации программ.

Генеральная классификация [4] определяет: невзаимодействующие программы (или программы-функции), реактивные системы (или программы-процессы), языковые процессоры и операционные среды. Имеются другие классы.

2.2. Программы-функции (невзаимодействующие программы)

Программа принадлежит классу программ-функций, если она не взаимодействует с внешним окружением. Точнее, если возможно перестроить программу таким образом, чтобы все операторы ввода данных находились в начале программы, а весь вывод собран в конце программы, то такая программа относится к классу невзаимодействующих программ. Программа обязана всегда завершаться, поскольку бесконечно работающая и невзаимодействующая программа бесполезна. Следовательно, программа определяет функцию, вычисляющую по набору входных данных (аргументов) некоторый набор результатов.

Гиперфункция с несколькими ветвями [1, 7, 8] – наиболее общая форма программы-функции. Гиперфункция с одной ветвью определяет обычную функцию. Каждая *ветвь* гиперфункции определяет независимый выход из программы и набор результатов по этой ветви. Наборы результатов по разным ветвям могут различаться. Набор результатов ветви может быть пустым. Гиперфункцию можно закодировать в виде программы-функции с одной ветвью.

В программе на языке Си только один результат. Дополнительные результаты поставляются через аргументы-указатели. В некоторых языках несколько результатов оформляются в виде типа «кортеж» (**tupple**). В языках WhyML [24] и Java дополнительные выходы из программы реализуются с помощью исключений. Результатами выхода по исключению являются параметры исключения. В языке предикатного программирования Р [1] выход по ветви гиперфункции реализуется оператором перехода следующего вида:

#<имя ветви гиперфункции>

Спецификация программы-функции представляется двумя предикатами: *предусловием*, ограничивающим значения аргументов программы, и *постусловием*, определяющим связь между значениями аргументов и результатов. Для гиперфункции предусловие и постусловие определяются по каждой ветви. Корректность программы-функции относительно спецификации реализуется доказательством формул корректности, генерируемых на базе логики Хоара [16].

2.3. Программы-процессы

Программа-процесс является реактивной системой, реагирующей на определенный набор событий (сообщений) во внешнем окружении программы. Программа-процесс является либо автоматной программой, либо она определяется в виде композиции нескольких автоматных программ, исполняемых параллельно и взаимодействующих между собой через прием / посылку сообщений и разделяемые переменные.

Автоматная программа состоит из одного или нескольких сегментов. *Сегмент* имеет один вход, помеченный меткой – *управляющим состоянием*. Сегмент имеет один или несколько выходов. Автоматная программа определяет конечный автомат в виде гиперграфа с набором управляющих состояний в качестве вершин и набором сегментов в качестве ориентированных гипердуг. Гиперграфовая структура автоматной программы является продолжением аппарата гиперфункций. *Состояние* автоматной программы определяется значениями набора переменных, модифицируемых в программе, за исключением локальных переменных.

Программа-процесс состоит из следующих частей:

- требований к реактивной системе;
- набора автоматных программ;
- секций, определяющих состояние автоматных программ и программы-процесса в целом.

Требование – утверждение, определяющее потребность и связанные с ней измеримые условия и ограничения. Требования к реактивной системе включают требования окружения и функциональные требования, определяющие поведение системы. Существенными являются также нефункциональные требования надежности, безопасности, защищенности, отсутствия дедлоков и другие. Разработка требований должна проводиться в соответствии со стандартом ISO/IEC/ IEEE 29148 [23].

2.4. Язык автоматного программирования

Язык автоматного программирования может быть построен расширением некоторого *базисного* языка для класса программ-функций. Это может быть любой язык императивного или функционального программирования. Таким способом определено автоматное расширение [2, 9] языка предикатного программирования P [1, разд.10].

Определим конструкции языка автоматного программирования.

Секция состояния автоматной программы представляется конструкцией:

section <Имя секции> **extends** <Имя секции>
{ <Описания типов, констант, переменных и инвариантов> }

Секция помещается перед автоматной программой. Имя секции может отсутствовать. Секция может быть построена расширением другой секции при наличии **extends** <Имя секции>.

Автоматная программа определяется следующей конструкцией:

process <Имя программы>(< Описания аргументов>) { <Сегменты кода> }

Аргументы могут отсутствовать. Произвольный <Сегмент кода> представляется конструкцией:

<Имя управляющего состояния>**: inv** <Формула>;<Оператор>

Исполнение <Оператора> завершается оператором перехода вида #M, реализующим переход на начало сегмента с управляющим состоянием M. <Формула> определяет инвариант, который должен быть истинным в начале сегмента для данного управляющего состояния.

Предполагается, что сегмент, код которого не содержит вызовов других процессов, является *атомарным*: исполнение сегмента должно быть единым, неделимым актом; мы полагаем, что до завершения исполнения сегмента все другие параллельно исполняемые процессы останавливаются.

Структура управления автоматной программы аналогична используемой в языке Фортран, применяемом в основном для задач вычислительной математики, но не для реактивных систем.

Сегмент кода следующего вида:

M: if (<условие$_1$>&…& <условие$_n$>) { <действие$_1$>; …; <действие$_m$> **#L }**

может быть записан в виде правила [6]:

M: <условие$_1$>,…, <условие$_n$> → <действие$_1$>, …, <действие$_m$> **#L**

Сегмент **M: if** (C) A **else** B **#L** может быть представлен парой правил:

M: C → A **#L**

M: B **#L**

Правила исполняются в порядке их следования. Второе правило **M:** B **#L** может сработать только при ложном условии C.

Для операторов A и B оператор A || B определяет параллельное исполнение операторов A и B. Оператор A | B определяет недетерминированный выбор для исполнения одного из операторов, A или B.

В языке автоматного программирования определяются также операторы приема и посылки сообщений, действия со временем: установка таймера, оператор задержки по времени и другие.

2.5. Технология автоматного программирования

Управляющие состояния и соответствующие им сегменты представляют различные стадии функционирования реактивной системы и, таким образом, определяют естественное структурирование автоматной программы. Некоторые управляющие состояния заложены уже в требованиях. Специализация для каждого управляющего состояния уменьшает число связей между объектами, что принципиально понижает сложность реализации реактивной системы. Отметим, что сложность экспоненциально зависит от числа переменных в состоянии и числа связей между ними.

Достаточно часто автоматная программа строится простым переписыванием функциональных требований. Их удобнее формулировать в виде логических правил [6].

Для получения более эффективной программы применяются эквивалентные трансформации, существенно меняющие структуру автоматной программы [5]. Гиперграфовая композиция является предельно гибкой. Показано, что любую автоматную программу можно представить композицией двух сегментов. Гиперграфовая композиция трех независимых процессов позволила упростить программу управления лифтом [2].

В дополнение к описанным методам иногда полезно использовать методы объектно- и аспектно-ориентированного программирования.

Технология автоматного программирования также представлена сводом правил [2].

Циклы в автоматной программе имеют другую природу по сравнению с циклами императивной программы. *Не рекомендуется использовать циклы типа* ***while*** *и* ***for*** *для конструирования автоматной программы*.

Автоматное программирование универсально. Любая программа-функция может быть запрограммирована в виде автоматной программы, которая, однако, будет значительно сложнее аналогичной функциональной или императивной программы, построенной обычными средствами. Поэтому *не следует использовать автоматное программирование для программ-функций*. Сегменты автоматных программ могут содержать достаточно крупные фрагменты кода, относящиеся к программам-функциям. *Рекомендуется выносить такие фрагменты из автоматных программ и программировать независимыми функциями*.

2.6. Спецификация и верификация автоматных программ

Методы верификации, успешно применяемые для программ-функций, в частности логика Хоара-Флойда [16, 14], непригодны для верификации автоматных программ. Эти методы могут применяться лишь для фрагментов автоматных программ, соответствующих программам-функциям.

Спецификация программы-процесса определяется общими инвариантами и инвариантами управляющих состояний. *Общий инвариант* формулируется для автоматной программы в секции состояния непосредственно перед автоматной программой. Общий инвариант должен быть истинным в начале каждого сегмента. Общий инвариант может быть представлен темпоральной формулой.

Допустим, состояние автоматной программы определяется набором переменных v. Для текущего сегмента построим функцию F(v) таким образом, чтобы формула $v' = F(v)$, где v' – значение набора переменных в конце сегмента, была точной спецификацией текущего сегмента. Текущий сегмент сохраняет общий инвариант Inv(v), если из истинности Inv(v) в начале сегмента следует его истинность в конце сегмента. Сохранность общего инварианта для текущего сегмента гарантируется проведением следующего доказательства:

$$Inv(v)\ \&\ Sinv(v) \vdash Inv(F(v))$$

где Sinv(v) – инвариант управляющего состояния.

Истинность инварианта управляющего состояния M доказывается для каждого оператора перехода #M.

Другие свойства программ-процессов, которые подлежат верификации, – это отсутствие взаимной блокировки процессов (дедлоков) и завершение конечных процессов.

Инварианты управляющих состояний часто являются слабыми или отсутствуют, то есть оказываются тождественно истинными. Инварианты управляющих состояний и общие инварианты принципиально отличаются от инвариантов циклов императивной программы.

Доказательство истинности всех инвариантов программы-процесса и отсутствия дедлоков позволяет избежать многих ошибок, но не гарантирует полной корректности программы в отличие от верификации программ-функций, где правильность спецификации и доказательство формул корректности гарантирует корректность программы.

3. Event-B и модель Event-B в автоматном программировании

Event-B [10] – это метод формальной спецификации и верификации систем в программной и системной инженерии с использованием нотации теории множеств и логики первого

порядка. Каждая спецификация на Event-B состоит из компонентов двух видов: контекстов и машин. *Контекст* определяет множества и константы – статическую часть спецификации. *Машина* содержит динамическую часть спецификации: переменные, инварианты, события. Значения переменных формируют текущее *состояние* спецификации. *Инварианты* определены на множестве переменных.

Текущее состояние спецификации может быть изменено событием. *Событие* состоит из названия, параметров, охранных условий и действий. *Охранные условия* ограничивают множество возможных состояний, в которых данное событие может произойти. *Действия* модифицируют значения переменных, изменяя текущее состояние спецификации. Инварианты должны сохраняться после модификаций переменных любыми действиями данной машины. Корректность любого изменения состояния необходимо доказывать.

Произвольное событие далее будем представлять в виде оператора:

if (<Охранные условия>) { <Действие> }

В типичном случае действие реализует присваивание нескольким переменным. Например, { c:=C; d:=D }. Выражения C и D вычисляются и их значения одновременно присваиваются переменным c и d. Эффект более сложного действия можно определить before-after предикатом, связывающим значения исходных и модифицированных переменных.

Все события атомарны и могут произойти, если выполняются их охранные условия. Если одновременно выполняются охранные условия нескольких событий, то только одно из них может произойти в данный момент, причем какое именно событие произойдет, выбирается недетерминированным образом.

Допустим, машина M состоит из событий A, B, E и F. Тогда машину M будем представлять следующим процессом с единственным управляющим состоянием cycle:

process M { cycle: [nA:] A | [nB:] B | [nE:] E | [nF:] F #cycle }

Здесь nA, nB, nE и nF – имена событий A, B, E и F. Имена событий, альтернатив недетерминированной композиции, представлены здесь в квадратных скобках в отличие от имен управляющих состояний.

Спецификация на Event-B состоит из последовательности машин, в которой очередная машина является *уточнением* (refinement) предыдущей машины. Очередная машина содержит новые переменные и события, расширяющие предыдущую машину. Причем поведение новой машины в проекции на наследуемые объекты и события полностью идентично поведению предыдущей машины.

В Event-B имеется возможность доказательства отсутствия состояний взаимной блокировки (дедлока), а также доказательства завершения конечных процессов.

Event-B определяет среду для разработки и верификации спецификаций на платформе Rodin [22]. Доказательство генерируемых для спецификации формул корректности поддерживается автоматическими и интерактивными средствами доказательства с применением SMT-решателей. Степень и возможности автоматизации доказательства существенно выше, чем в системах доказательства PVS [21], Why3 [24] и Coq [12]. В частности, автоматизировано применение эквивалентных замен (rewrite). В дереве текущего процесса доказательства явно обозначены позиции, по которым пользователь может продолжить доказательство, выбрав одну из прилагаемых в позиции альтернатив.

Построим новый язык автоматного программирования расширением языка спецификаций Event-B. Некоторые конструкции языка Event-B заменяются другими, более привычными. Будем использовать примитивные типы BOOL, INT, NAT и оператор присваивания <Переменная> := <Выражение>. Предикат $x \in BOOL$ будем записывать в виде x: BOOL, что соответствует описанию типа переменной в стиле языка Паскаль. Не используется событие INITIALISATION. Начальная инициализация переменных реализуется в секции состояния. Например, x: BOOL := false. Вместо конструкции partition со сложной семантикой используется описание типа перечисления **enum** в стиле языка Си.

4. Управление движением на перекрестке

Данная задача представлена в качестве первого примера в руководстве по платформе Rodin [22] для демонстрации базисных конструкций языка Event-B и начальных действий пользователя в системе Rodin.

Содержательное описание. На пешеходном переходе через автомобильную магистраль имеется два светофора. Светофор для пешеходов: красный или зеленый. Светофор для автомобилей: красный, желтый или зеленый. Запрещено движение на красный светофор для пешеходов и автомобилей. Контроллер управляет переключениями светофоров. Должно соблюдаться требование безопасности: зеленый светофор для пешеходов сочетается только с красным светофором для автомобилей.

Анализ требований. Система управления движением, представленная содержательным описанием, не может использоваться на практике. По меньшей мере, необходимо обеспечить зеленый свет в течение определенного фиксированного времени, чтобы пешеходы успели перейти автомобильную магистраль. Поэтому данная задача – всего лишь учебный пример.

Объекты внешнего окружения: светофор для пешеходов и светофор для автомобилей. Сформулируем требования окружения:

RE1: Светофор для пешеходов может быть зеленым или красным.

RE2: Светофор для автомобилей может быть зеленым, желтым или красным.

Зафиксируем требование безопасности:

RS: Если светофор для пешеходов – зеленый, то светофор для автомобилей может быть только красным.

В руководстве [22] сначала рассматривается упрощенная задача. Красный свет представляется константой false, а зеленый – константой true. Желтого света нет. Светофор для пешеходов представлен логической переменной peds_go, а светофор для автомобилей – переменной cars_go.

Машина для упрощенной задачи в руководстве [22] эквивалентным образом определяется в виде следующей автоматной программы.

```
section {
    peds_go: BOOL := false
    cars_go: BOOL := false
    inv  peds_go = false ∨ cars_go = false
}
process PedsCars0 {
  Cycle:    [set_peds_go:] cars_go = false → peds_go := true  |
            [set_cars_go:] peds_go = false → cars_go := true  |
            [set_cars_stop:] cars_go := false  |
            [set_peds_stop:] peds_go := false
          #Cycle
} PedsCars0
```

Легко доказать, что инвариант сохраняется после исполнения каждой альтернативы недетерминированной композиции.

Представим нашу версию для упрощенной задачи. Введем два управляющих состояния:

- Cars – машины движутся, пешеходы стоят и ждут;
- Peds – пешеходы переходят магистраль, машины стоят перед красным светофором.

Автоматная программа представлена ниже.

```
state {
    peds_go: BOOL := false
    cars_go: BOOL := true
    inv  peds_go = false ∨ cars_go = false
}
process PedsCars1 {
   Cars: inv  peds_go = false & cars_go = true
         #Cars | peds_go := true; cars_go := false #Peds
   Peds: inv  peds_go = true & cars_go = false
         #Peds | peds_go := false; cars_go := true #Cars
} PedsCars1
```

Отметим, что сегмент Cars: #Cars | X эквивалентен сегменту Cars: X. Поэтому далее альтернативу вида #Cars будем опускать.

Чтобы закодировать процесс PedsCars1, например, в таком языке, как Java, реализуется кодирование управляющих связей информационными связями. Этот прием ранее был представлен Switch-технологией [3]. Вместо управляющих состояний Cars и Peds вводятся константы Cars и Peds как значения нового типа перечисления State. Программа процесса PedsCars1 переписывается следующим образом:

```
process PedsCars2 {
    type State = enum { Cars, Peds };
    state: State;
    state := Cars;
    loop:
       switch (state) {
          case Cars:  peds_go := true; cars_go := false; state :=Peds; break
          case Peds: peds_go := false; cars_go := true;  state :=Cars
       }
       #loop
} PedsCars2
```

Отметим, что программа PedsCars1 проще, короче и эффективней по сравнению с программой PedsCars2.

Данный способ кодирования управляющих состояний с помощью новой переменной state далее будем использовать при кодировании автоматной программ в виде спецификации Event-B. Например, кодирование PedsCars1 дает следующую автоматную программу, которая далее непосредственно переписывается в спецификацию на языке Event-B.

```
process PedsCars3 {
   Cycle: [Cars_go:] state=Cars → peds_go := true, cars_go := false, state := Peds |
          [Peds_go:] state=Peds → peds_go := false, cars_go := true, state := Cars
          #Cycle
} PedsCars3
```

Спецификация исходной задачи строится в руководстве [22] посредством пары уточнений спецификации упрощенной задачи, представленной выше. Эти уточнения приведены, видимо, в учебных целях. Однако их полезность в данном случае, сомнительна. Они усложняют решение задачи.

Наша автоматная программа для полной задачи отличается от итоговой спецификации в руководстве [22]. Действия контроллера на желтом свете светофора реализуются неодинаково в зависимости от предыдущего состояния светофора. Чтобы учесть такую особенность, введем два разных желтых света: yellowRed и yellowGreen. Автоматная программа приведена ниже.

```
section {
    type COLOURS = enum {red, yellowRed, yellowGreen, green}
    peds_col: COLOURS := red
    cars_col : COLOURS := green
    inv  peds_col = red ∨ peds_col = green & cars_col = red
}
process LightControl {
   Cars: inv  peds_col = red & cars_col = green
         cars_col := yellowGreen #YellowGreen
   YellowGreen: inv  peds_col = red & cars_col = yellowGreen
         cars_col := red; peds_col := green #Peds
   Peds: inv  peds_col = green & cars_col = red
         peds_col := red;  cars_col := yellowRed #YellowRed
   YellowRed:  inv  peds_col = red & cars_col = yellowRed
         cars_col := green #Cars
} LightControl
```

Данная программа была закодирована в виде спецификации на Event-B. Установлено, что без инвариантов управляющих состояний не проходит автоматическое доказательство всех формул корректности спецификации. Автоматическое доказательство реализуется лишь при введении пары инвариантов:

```
@invPeds   (state = Peds)   ⇔   ( peds_go = green ∧ cars_go = red)
@invCars   (state = Cars)   ⇔   ( peds_go = red ∧ cars_go = green)
```

5. Управление движением автомобилей по мосту

Данная задача представлена в качестве первого примера в известной книге Abrial J.-R. по системе Event-B [10]. На базе этого примера определяются основные концепции технологии Event-B.

5.1. Требования

Содержательное описание. Имеется мост, соединяющий материк и остров. Мост узкий и позволяет двигаться автомобилям по нему только в одну сторону. Имеются два светофора, установленных при въезде на мост с материка и с острова. У каждого светофора два цвета: красный и зеленый. Автомобилям запрещено движение на красный светофор при въезде на мост. Имеются четыре сенсора. Каждый сенсор находится на некотором участке автомобильной трассы и способен фиксировать ситуацию, когда автомобиль находится на этом участке трассы. Первый сенсор находится перед въездом на мост с материка. Второй сенсор на мосту перед съездом на остров. Третий сенсор перед въездом на мост с острова. Четвертый сенсор на мосту перед съездом на материк.

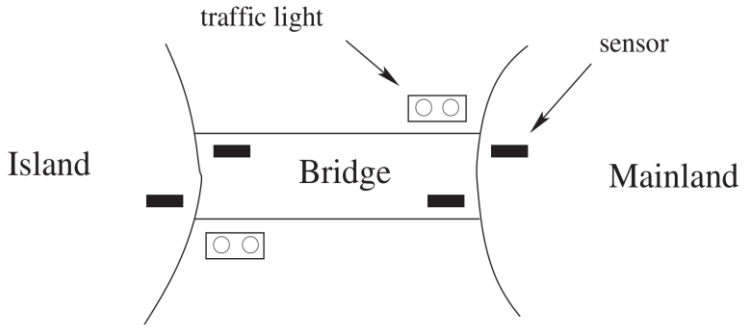


Рис. 1. Схема моста со светофорами и сенсорами.

Число автомобилей, которые могут находиться на острове, ограничено. Необходимо построить контроллер, который переключает светофоры, используя показания сенсоров, и таким образом обеспечивает безопасное движение автомобилей по мосту.

Детализация и анализ требований. Дадим имена светофорам: mtl – светофор на материке при въезде на мост, itl – светофор на острове при въезде на мост. Сформулируем требования окружения:

RE1: Имеются два светофора с именами mtl и itl.

RE2: Каждый светофор может быть зеленым или красным.

Дадим имена сенсорам: mlOut – при въезде на мост с материка, ilIn – при съезде с моста на остров, ilOut – при въезде на мост с острова, mlIn – при съезде с моста на материк.

RE3: Имеются четыре сенсора с именами: mlOut, ilIn, ilOut и mlIn.

RE4: Каждый сенсор может быть включен (значение on) или выключен (значение off).

RE5: Сенсор *отключается*, если он изменяет свое значение с on на off.

Каждый сенсор сопряжен с некоторым участком автомобильной трассы. Сенсор включен, если на данном участке находится автомобиль. Отключение сенсора означает, что автомобиль съехал с данного участка, и участок стал свободным от автомобилей. Сформулируем функциональные требования.

RF1: Отключение сенсора mlOut – очередной автомобиль въехал на мост с материка.

RF2: Отключение сенсора ilIn – очередной автомобиль съехал с моста на остров.

RF3: Отключение сенсора ilOut – очередной автомобиль въехал на мост с острова.

RF4: Отключение сенсора mlIn – очередной автомобиль съехал с моста на материк.

RF5: Число автомобилей на острове ограничено.

RF6: Движение автомобилей по мосту возможно либо с материка на остров, либо с острова на материк. Одновременное движение с материка на остров и с острова на материк невозможно.

Определим требования безопасности:

RS1: При красном светофоре mtl запрещено движение с материка на мост.

RS2: При красном светофоре itl запрещено движение с острова на мост.

Представленная система управления движением по мосту в целом, и архитектура оборудования (Рис.1) в частности, имеют серьезные недостатки и не могут использоваться на практике. Ничто не может помешать автомобилю встать на сенсор, а затем поехать в обратном направлении. Это приведет к неправильному подсчету автомобилей на мосту, что может спровоцировать аварию. Автомобиль может также заехать на мост, а потом съехать с него задним ходом. Наконец, автомобиль может просто поехать на красный свет, если конечно красный свет не сопровождается установкой чугунного шлагбаума.

Поэтому данную задачу построения контроллера управления движением по мосту мы можем рассматривать только как учебную.

5.2. Разработка от простейшей модели

Построение итоговой спецификации определено в книге [10] в виде последовательности уточнений начиная с простейшей модели. Здесь мы покажем начальную модель и первые две ее уточнения.

В начальной модели мост и остров рассматриваются как единое целое. В модели фиксируются автомобили, уходящие с материка, и автомобили, приходящие на материк. Пусть n – число автомобилей на мосту и на острове. Поскольку число автомобилей на острове ограничено, введем константу d, определяющую максимальное число автомобилей

на острове. Начальная машина содержит два события: уход машины с материка и приход машины на материк. Машину представим в виде следующей автоматной программы.

```
section St0 {
    const d: NAT
    n: NAT
    inv  n <= d
}
process carBridge0er {
   cycle: [ML_out:] n:=n+1 |
          [ML_in:] n:=n-1
          #cycle
}
```

Легко обнаружить, что в модели есть ошибки. Например, нет гарантии, что после действия n:=n+1 не будет превышен лимит автомобилей d. Сила метода Event-B [10] в том, что этот метод позволяет обнаруживать все ошибки такого рода в процессе доказательства формул корректности. В данном случае будет представлена формула корректности: n+1<=d, которая здесь недоказуема. Чтобы исправить данную ошибку, необходимо добавить охранное условие: n<d. Вторая ошибка. Для действия n:=n-1 будет сгенерирована формула корректности n-1 ∈ NAT, которая оказывается недоказуемой. Необходимо вставить охранное условие: n>0. Третья ошибка. В начальный момент работы машины, после исполнения события-инициализации, не выполняется формула корректности: n <= d. Причина в том, что n не инициализирована. Необходимо вставить инициализацию: n:=0. Четвертая ошибка. При анализе отсутствия дедлоков генерируется формула корректности: n <= d ⇒ n<d ∨ n>0, которая недоказуема при d=0. В случае d=0 действительно возникает дедлок, поэтому необходимо условие d>0 в виде аксиомы. Исправленная машина представлена ниже.

```
section St0 {
    const d : NAT   максимальное число автомобилей на острове
    axiom d > 0
    n: NAT := 0
    inv  n <= d
}
process carBridge0 {
   cycle: [ML_out:] n < d → n := n + 1 |
          [ML_in:]   n > 0 → n := n - 1
          #cycle
}
```

Далее определим уточнение представленной выше начальной модели. В уточненной модели различаются автомобили, находящиеся на мосту и на острове. Уточнение заключается в замене старой переменной n тремя новыми переменными: a – число

автомобилей, движущихся по мосту с материка на остров, b – число автомобилей на острове, c– число автомобилей, движущихся по мосту с острова на материк. Вводится *связующий инвариант* a+b+c = n, определяющийся отношение между старыми и новыми переменными. Появились два новых события: автомобиль переходит с моста на остров (IL_in) и автомобиль въезжает на мост с острова (IL_out).

```
section St1 extends St0{
    a: NAT := 0
    b: NAT := 0
    c: NAT := 0
    inv a+b+c = n
    inv a=0 ∨ c = 0
}
process carBridge1 refines carBridge0 {
   cycle: [ML_out:]  a+b<d,  c=0 → a:=a+1 |
          [ML_in:]   c>0  →  c:=c-1 |
          [IL_out:]  b>0, a=0 → b:=b-1, c:=c+1 |
          [IL_in:]   a>0  → a:=a-1, b:=b+1
          #cycle
}
```

Мы видим, что события ML_out и ML_in в уточненной модели отличаются от этих же событий в начальной модели. Для этих событий в Event-B генерируются формулы корректности, доказательство которых гарантирует идентичность поведения старых и новых событий. Корректность уточнения для добавленных событий IL_out и IL_in будет обеспечена, если эти события не меняют значений старых переменных, то есть значения переменной n.

Следующая модель строится как уточнение предыдущей модели. Причем это уточнение другого вида. В предыдущей модели уточнялись структуры данных. Здесь же происходит добавление новых структур данных и новых событий.

Вводятся светофоры: mtl – на материке при въезде на мост, itl – на острове при въезде на мост. Вводится инвариант mtl=red ∨ itl=red, обеспечивающий выполнение требования RF6. Два других инварианта введены для упрощения доказательства корректности уточнения: охранное условие некоторого события в уточненной модели должны быть не слабее аналогичного условия в предыдущей модели, а действия в строй и новой моделях должны быть идентичны.

```
section St2 extendes St1{
        type Colour = enum {red, green};
        mtll: Colour := red
        itl: Colour := red
        inv mtl=red ∨ itl=red
        inv mtl=green => a+b<d & c=0
        inv itl=green => b>0 & a=0
    }
    process carBridge2 refines carBridge1{
       cycle: [ML_out:] mtl=green → a:= a+1 |
              [ML_in:]   c>0  → c:=c-1 |
              [IL_out:]  itl=green  →  b:=b-1, c:= c+1 |
              [IL_in:]    a>0 →   a:=a-1, b:=b+1 |
              [Mtl_green:]  mtl=red, c=0, b<d  →  mtl:=green, itl:=red |
              [Itl_green:]   itl=red, a=0, b>0 →  itl:=green, mtl:=red
              #cycle
    }
```

Новые события Mtl_green и Itl_green определяют вроде бы естественные правила переключения светофоров: если мост стал пустым, то оба светофора меняют свет так, чтобы стало возможным движение в противоположную сторону. Далее, поскольку мост все еще пустой, может сработать другое правило, изменяющее направление движению. Таким образом, данные правила приведут к быстрому миганию светофоров. В книге [10] реализовано следующее решение. Если изменено направление движения, то далее необходимо дождаться появления автомобиля на мосту с противоположной стороны. Данное решение неудовлетворительно. Например. Утром все поехали на остров, где запланирован пикник. Первая партия машин проехала. А остальные окажутся заблокированными до вечера, когда с острова появится первый автомобиль.

Правильное решение следующее. Изменение направления движение возможно, если с противоположной стороны имеется автомобиль, стоящий на сенсоре в ожидании зеленого светофора. Реализация данного решения плохо стыкуется с проведенными уточнениями. Разработку контроллера нужно проводить другим способом.

В последнем уточнении добавляются сенсоры. В книге [10] последняя машина оказалась громоздкой и неадекватной, что серьезно дискредитирует сам метод Event-B.

5.3. Разработка по технологии автоматного программирования

Представим нашу реализацию данной задачи.

Константа d, переменные b, mtl, и itl определяются также как и в описанной выше реализации. Переменная a определяет число автомобилей на мосту в любом из направлений движения. Переменная c исключена.

Новые переменные. Переменная ml_out = true, если сенсор mlOut включен, то есть на сенсоре mlOut со стороны материка в данный момент находится машина. Переменная il_out = true, если сенсор ilOut включен, т.е. на сенсоре ilOut со стороны острова находится машина. Переменная empty = true в состоянии ожидания первого автомобиля на пустой мост.

Секция состояния для программы-процесса управления движением по мосту представлена ниже.

```
section St0 {
    const d : NAT
    axiom d >0
    a: NAT := 0
    b: NAT := 0
    type Colour = enum {red, green}
    mtl: Colour := red
    itl:   Colour := red
    ml_out: BOOL := false
    il_out: BOOL := false
    empty: BOOL := false
    inv  a + b <= d
    inv mtl = red ∨ itl = red
}
```

Полная программа-процесс представляется параллельной композицией пяти процессов, определяемых пятью автоматными программами. Для каждого сенсора имеется независимая автоматная программа, управляющая функционированием данного сенсора. Этими программами подсчитывается число автомобилей на мосту и на острове, а также значения логических переменных ml_out и il_out. Управление светофорами реализуется пятой автоматной программой LightControl.

Данная композиция из пяти процессов представляется вполне адекватной. Понятно, что подсчет числа автомобилей на мосту и на острове следует проводить на основе изменений значений сенсоров. И это проще реализовать в программах управления сенсорами. Управление светофорами было бы гораздо труднее реализовать в рамках четырех процессов для сенсоров.

Рассмотрим программу управления светофорами LightControl. Очевидными являются следующие три управляющих состояния:

- zero – мост пуст, оба светофора красные, и по значениям переменных ml_out и il_out реализуется соответствующая переустановка светофоров;
- right – реализуется движение слева направо, то есть с острова на материк;
- left – реализуется движение справа налево, то есть с материка на остров.

В состояниях right и left проверяется ситуация, когда мост становится пустым. В этом случае реализуется переход в состояние zero. В состоянии left также проверяется ограничение a+b<=d. В ситуации a+b=d светофор mtl устанавливается красным. Для того, чтобы перейти в состояние zero, необходимо сначала освободить мост. Освобождение моста контролируется в дополнительном управляющем состоянии leftscan.

Очевидно, что после перехода с zero на right или left немедленно произойдет возврат на zero, потому что мост пуст. Мы получим быстрое мерцание светофора с зеленого на красный. Поэтому переход с zero реализуется в новые состояния right0 и left0, где ожидается появление на мосту первой машины. При этом в состоянии left0 при появлении первой машины становится возможной ситуация a+b>d. В связи с этим, при переходе с zero на left0 необходимо дополнительно проверять условие b < d.

Возникает ситуация блокировки (дедлока) в состояниях right0 и left0, если первый автомобиль появился на мосту и успел выехать с моста раньше, чем сработает сегмент кода в состояниях right0 или left0. Подобное, однако, может реализоваться, если приостановить работу процесса LightControl. Во избежание блокировки введен признак empty, При истинном значении empty автомобиль не может покинуть мост.

Далее представлена автоматная программа LightControl.

```
process LightControl {
    zero: inv a=0 & mtl = red & itl = red;
          if (il_out) {itl := green; empty := true #right0}
          else if (ml_out & b< d) {mtl := green; empty := true #left0}
          else #zero
    right0: inv mtl = red & itl = green;
            if (a=0) #right0 else { empty :=false #right}
    right:  inv mtl = red & itl = green;
            a=0 → itl := red #zero
    left0:  inv mtl = green & itl = red;
            if (a=0) #left0 else { empty := false #left }
    left:   inv mtl = green & itl = red;
            if (a+b=d) {mtl := red; #leftscan}
            else if (a=0) {mtl := red #zero }
            else #left
    leftscan: inv mtl = red & itl = red;
              a=0 → #zero
}
```

5.4. Программы управления сенсорами

Простейшая программа управления сенсором состоит из двух сегментов включения и выключения сенсора: off: #on и on: #off. В нашем случае четыре программы управления сенсорами нагружены разными дополнительными действиями и реализуют переключения при определенных условиях.

На пустом мосту, при a=0, невозможна ситуация «автомобиль покидает мост», то есть переход с on на off. Автомобили могут заехать на мост только при зеленом светофоре. При переключении сенсоров вычисляются значения переменных ml_out и il_out.

Программы управления сенсорами представлены ниже.

```
process ML_out {
    inv a+b<=d
    off: ml_out:=true #on
    on: mtl=green,  a+b<d → a:=a+1, ml_out:=false #off
}
process IL_in {
    inv a+b<=d
    off: a>0, not empty, itl=red →#on
    on: a:=a-1; b:=b+1 #off
}
process ML_in {
    off: a>0, not empty →#on
    on: a:=a-1 #off
}
process IL_out {
    off: b>0 → il_out:=true #on
    on: itl=green → b:=b-1, a=a+1, il_out:=false
}
```

Полная программа управления движением автомобилей на мосту запускается следующей головной программой:

```
process carBridge {
    LightControl || ML_out || IL_in || IL_out || ML_in
}
```

5.5. Управление сенсорами через монитор

Когда один автомобиль заезжает на мост, а другой автомобиль съезжает с моста, две разные программы, исполняемые параллельно, будут пытаться, возможно, одновременно, изменить значение переменной a. Подобное становится критичным, когда программы управления сенсорами реализуются разными процессорами.

Чтобы исключить возможность одновременного доступа к переменной a разными процессами, используется монитор для пересчета переменных a и b, управляемый семафорами. Определим секцию для семафоров.

```
section Stcar {
    semal: BOOL:=false
    semabl: BOOL:=false
    semar: BOOL:=false
    semabr: BOOL:=false
    inv semal = TRUE ⇒ a+b < d
}
```

Монитор состоит из двух программ, работающих независимо. Первая программа при движении с материка на остров, вторая программа при движении с острова на материк.

```
process Left_ab {
    loop:  if (semal) { a:=a+1; semal := false};
           if (semabl & a>0) { a:=a-1; b:=b+1; semabl := false} #loop
}
process Right_ab {
    loop:  if (semar & a>0) { a:=a-1; semar := false};
           if (semabr) { a:=a+1; b:=b-1; semabr := false} #loop
}
```

Представим новые версии программ управления сенсорами. Чтобы изменить значения a и b каждая программа устанавливает соответствующий семафор в true и переходит в управляющее состояние wait, где ожидает снятия этого семафора, чтобы продолжить работу.

```
process ML_out {
    inv a+b<=d
    off: ml_out:=true #on
    on: mtl=green, a+b<d → semal:=true, #wait
    wait: not semal → ml_out:=false  #off
}
process IL_in {
    inv a+b<=d
    off: a>0, not empty, not semabl, itl=red → semabl:=true #wait
    wait: not semabl→ #off
}
process ML_in {
    inv itl=green
    off: a>0, not empty → semar := true #wait
    wait: not semar → #off
}
process IL_out {
    inv a+b<=d
    off: b>0 → il_out:=true #on
    on: itl=green →  semabr:=true #wait
    wait: not semabr → il_out:=false #off
}
```

Головная программа реализует запуск семи параллельных процессов:

```
process carBridge {
    LightControl ||
    { ML_out || IL_in  || Left_ab } ||
    { IL_out || ML_in || Right_ab }
}
```

5.6. Опыт верификации в системе Rodin

Программа управления движением автомобилей по мосту, описанная выше, была закодирована в виде спецификации на языке Event-B [10] и верифицирована в инструменте

Rodin [22]. Управляющие состояния программы LightControl закодированы как значения переменной state типа перечисления State:

type State = **enum** { zero, right0, right, left0, left, leftscan }

Для кодирования управляющих состояний программ управления сенсорами используются значения следующего типа:

type Sensor = **enum** { off, on, wait }

Сегмент кода автоматной программы может быть закодирован несколькими событиями, от одного до трех. Семь автоматных программ, исполняемых параллельно в полной программе управления движением на мосту, кодируются в одной машине на языке Event-B. Итоговая смесь всех событий в рамках одной машины фактически реализует полный интерливинг атомарных сегментов семи автоматных программ.

Чтобы облегчить доказательство формул корректности, инварианты управляющих состояний добавлены к двум основным инвариантам программы.

79 сгенерированных формул корректности инвариантов были доказаны автоматически в системе Rodin. При этом были уточнены некоторые охранные условия. Потребовалось введение дополнительного инварианта: semal = TRUE ⇒ a+b < d .

К сожалению, не были сгенерированы автоматически формулы корректности для проверки на возможность дедлоков. Компонента Rodin проверки на дедлоки оказалась по непонятной причине заблокирована.

Дальнейшая верификация проводилось применением аниматора – компоненты Rodin, реализующей пошаговое исполнение спецификации Event-B с визуализацией текущего состояния, возможностью выбора любого из возможных вариантов исполнения и возможностью вернуться назад в процессе исполнения. С помощью аниматора удалось эффективно перебрать ключевые варианты исполнения и обнаружить следующие ошибки в исходной программе.

Сначала был обнаружен дедлок в состояниях right0 и left0, когда первый автомобиль на мосту выезжает с него раньше срабатывания сегмента в состояниях right0 или left0. Был введен признак empty для пустого моста. **Вторая ошибка**: дедлок в состоянии left0 при b = d. Для исправления ошибки введено дополнительное охранное условие b < d в состоянии zero. **Третья ошибка** была обнаружена на одном из вариантов анимации. После въезда на мост первого автомобиля с острова этот автомобиль неожиданно возвращался на остров. Здесь оказалось возможным срабатывание сегмента off процесса IL_in. Для исправления ошибки введено дополнительное охранное условие itl = red.

Таким образом, система Rodin показала свою эффективность при верификации автоматных программ.

6. Обзор работ

Построение спецификаций на Event-B реализуется сверху вниз, что для сложных задач приводит к объемным монолитным моделям. В работе [15] предложен механизм, позволяющий проводить разработку моделей снизу, сначала создавая модели более простых компонент, а затем объединяя их в общую модель. Механизм построения снизу интегрирован с техникой уточнений, реализуемой сверху.

Подход иллюстрируется на задаче управления движением автомобилей по мосту. Построена универсальная модель контроллера сенсора. Восемь переменных в состоянии этой модели. Далее четыре экземпляра этих переменных поступают в главную машину управления движением на мосту. Метод уточнений позволяет лишь частично снизить сложность и громоздкость полной модели. Полная модель состоит из пятнадцати машин. Одна из машин длиною в 379 строк. Наша единственная машина содержит 205 строк. В нашей модели всего семь переменных. Связь между процессом LightControl и другими процессами для сенсоров у нас минимизирована двумя переменными ml_out и il_out.

Решение проводить вычисление числа машин на мосту и на острове в центральной части модели, реализованное в руководстве по Event-B [10] и в работе [15], является неудачным, приводящим к сложной реализации. Метод уточнений ситуацию не спасает. Очевидным это становится только в автоматном программировании.

Автоматные методы программирования используются во многих известных языках, таких как UML, SDL, TLA+ [17], Event-B [10] и графических языках. Использование управляющих состояний при разработке программ реактивных систем характерно лишь для автоматного программирования [2, 3], которое доказало свою эффективность на множестве разнообразных приложений.

Существует четыре способа кодирования автоматных программ:

- стиль языка Фортран, используемый в нашем подходе;
- Switch-технология [3];
- рекурсивные вызовы вместо оператора перехода;
- предметно-ориентированный язык (DSL).

Использование меток в TLA+ [17] внешне похоже на управляющие состояния в автоматном программировании. Декларировано, что метки лишь фиксируют участки атомарности в спецификации TLA+.

В Switch-технологии [3] управляющие состояния становятся значениями некоторой переменной, а сегменты кода – альтернативами оператора switch. Показано (Разд. 4), что при таком преобразовании программа становится длиннее, сложнее и менее эффективной. Однако для языков, не содержащих оператора перехода, это лучший способ кодирования автоматных программ.

В третьем способе каждый сегмент кода автоматной программы кодируется независимой рекурсивной функцией. Вместо оператора перехода на другой сегмент вставляется рекурсивный вызов соответствующей рекурсивной функции. Аргументами рекурсивных функций являются переменные состояния автоматной программы. Программа длиннее и сложнее, чем в первых двух способах. Данный способ применяется в алгебрах процессов CCS, CSP и других, а также в языке Erlang [18].

В четвертом способе автомат программы кодируется в специальном предметно-ориентированном языке (DSL). Это, например, языки AbC [20] и Microsoft P [13, 19]. Набор элементарных конструкций в таких языках ограничен. Исполнение программ реализуется через интерпретатор. Имеется также возможность трансляции программы на язык Си.

7. Заключение

В разделе 3 настоящей работы определена модель технологии Event-B в автоматном программировании. С помощью этой модели на примерах задач в разделах 4 и 5 дана демонстрация технологии Event-B [10]. Проведено сравнение с технологией автоматного программирования. Подведем итоги.

Спецификация на языке Event-B представляет собой недетерминированную композицию операторов простого вида: **if** (Охранные условия>) { <Действие> }. Язык Event-B оказывается узким подъязыком автоматного программирования. Для конструирования спецификации Event-B используются лишь недетерминированная композиция и уточнение (refinement). Тогда как автоматное программирование кроме недетерминированной композиции допускает параллельную композицию процессов, гиперграфовую композицию по управляющим состояниям, подпроцессы, объектно-ориентированную и аспектно-ориентированную композиции.

Решения задачи управления движением автомобилей по мосту в руководстве по Event-B [10] и в работе [15], являются сложными и громоздкими, дискредитирующими метод

уточнений. Отметим, что уточнения, представленные в разделе 5.2, не упрощают разработку программы в автоматном программировании. Полезность метода уточнений в автоматном программировании предстоит исследовать в дальнейшем.

Наша программа управления движением по мосту была успешно верифицирована в инструменте Rodin [22] с применением средств автоматического и интерактивного доказательства, по уровню и возможностям выше, чем в системах доказательства PVS [21], Why3 [24] и Coq [12]. С помощью системы Rodin обнаружены три нетривиальные ошибки в нашей программе. Таким образом, систему Rodin следует рекомендовать для верификации критических автоматных программ.

Система правил генерации формул корректности, используемая для спецификаций Event-B, вполне может быть адаптирована для автоматных программ. Следует также предусмотреть интеграцию системы верификации автоматных программ с традиционной системой верификации на базе логики Хоара [16] для применения к фрагментам автоматной программы, относящимся к классу программ-функций.

Список литературы

1. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Р. — Новосибирск, 2010. — 42с. — (Препр. / ИСИ СО РАН; N 153). URL: http://persons.iis.nsk.su/files/persons/pages/plang14.pdf (дата обращения: 10.12.2021)
2. Тумуров Э.Г., Шелехов В.И. Технология автоматного программирования на примере программы управления лифтом // «Программная инженерия», Том 8, № 3, 2017. – С.99-111. http://persons.iis.nsk.su/files/persons/pages/lift1.pdf (дата обращения: 10.12.2021)
3. Шалыто А. А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб: Наука, 1998. http://is.ifmo.ru/books/switch/1
4. Шелехов В.И. Классификация программ, ориентированная на технологию программирования // «Программная инженерия», Том 7, № 12, 2016. — С. 531–538. https://persons.iis.nsk.su/files/persons/pages/prog.pdf (дата обращения: 10.12.2021)
5. Шелехов В.И. Оптимизация автоматных программ методом трансформации требований // «Программная инженерия», №11, 2015. – С. 3-13. http://persons.iis.nsk.su/files/persons/pages/req_k.pdf (дата обращения: 10.12.2021)
6. Шелехов В.И. Разработка автоматных программ на базе определения требований // Системная Информатика. – Новосибирск: ИСИ СО РАН, 2015. – №1. – С. 1-29. URL: http://persons.iis.nsk.su/files/persons/pages/req_tech.pdf (дата обращения: 10.12.2021)

7. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. — Новосибирск, 2012. — 30с. — (Препр. / ИСИ СО РАН. № 164).
8. Шелехов В.И. Разработка программы построения дерева суффиксов в технологии предикатного программирования. — Новосибирск, 2004. — 52с. — (Препр. / ИСИ СО РАН; N 115).
9. Шелехов В.И. Язык и технология автоматного программирования // Программная инженерия. – 2014. – №4. – С. 3–15. URL: http://persons.iis.nsk.su/files/persons/pages/automatProg.pdf (дата обращения: 10.12.2021)
10. Abrial J.-R. Modeling in Event-B: System and Software Engineering. Cambridge University Press, 2010. 586p.
11. Butler M.J., Körner P., Krings S., Lecomte T., Leuschel M., Mejia L.-F., Voisin L. The First Twenty-Five Years of Industrial Use of the B-Method / Formal Methods for Industrial Critical Systems (FMICS). 2020. pp. 189-209.
12. The Coq Proof Assistant. https://coq.inria.fr/ (дата обращения: 10.12.2021)
13. Desai A., Qadeer S., Seshia S. A.: Programming safe robotics systems: Challenges and ad-vances. In: Leveraging Applications of Formal Methods, Verification and Validation. Ver-ification, 2018, LNCS 11245, pp. 103–119.
14. Floyd R. W. Assigning meanings to programs // Proceedings Symposium in Applied Mathematics, Mathematical Aspects of Computer Science. AMS, 1967. pp. 19–32
15. Hoang T. S., Dghaym D., Snook C., Butler M. A Composition Mechanism for Refinement-Based Methods // 22nd International Conference on Engineering of Complex Computer Systems (ICECCS), 2017, pp. 100-109
16. Hoare C. A. R. An axiomatic basis for computer programming // Communications of the ACM. 1969. Vol. 12 (10). P. 576–585.
17. Lamport L. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, 2021. P.382.
18. Larson J. Erlang for concurrent programming. ACM Queue, 2008, No. 5, pp. 18-23.
19. Liu P., Wahl T., Lal A.: Verifying Asynchronous Event-Driven Programs Using Partial Abstract Transformers. In: Computer Aided Verification, LNCS 11562, 2019, pp. 386-404.
20. Nicola R. D., Duong, T., Inverso O.: Verifying AbC Specifications via Emulation. In: Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles, ISoLA 2020, Part II, LNCS 12477, pp. 261-279.
21. PVS Specification and Verification System. SRI International. http://pvs.csl.sri.com/ (дата обращения: 05.03.2020)
22. Rodin User's Handbook. Version 2.8 / Jastram M. (editor). 2014. 184p.
23. Systems and software engineering — Life cycle processes — Requirements engineering. ISO/IEC/ IEEE 29148, 2011, 95p.

24. Why 3. Where Programs Meet Provers. URL: http://why3.lri.fr (дата обращения: 10.12.2021)

Приложение 1

Управление движением на мосту. Спецификация Event-B

Переменные ml_out и il_out, которые были определены как логические, здесь представлены значениями констант **on** и **off**.

```
context EnvControl
sets Sensor Colour States
constants d red green on off wait
          zero right0 right left0 left leftscan
axioms
   @d_def d>0
   @col_ty partition(Colour, {red}, {green} )
   @sens_ty partition(Sensor, {on}, {off}, {wait} )
   @states_ty partition(States, {zero}, {right0}, {right}, {left0}, {left}, {leftscan}
)
end

machine CarBridge
sees EnvControl
variables a b mtl itl ml_out il_out state empty
          semal semabl semar semabr
invariants
     @inva a≥0 @invb b≥0
     @inv_mtl mtl∈Colour @inv_itl itl∈Colour
     @inv_ml_out ml_out∈Sensor @inv_il_out il_out∈Sensor
     @inv_1 empty ∈ BOOL
     @inv_2 semal∈ BOOL @inv_3 semabl ∈ BOOL @inv_4 semar∈ BOOL @inv_5 semabr ∈ BOOL
     @inv_st state ∈ States
     @thm_aplus  semal = TRUE ⇒ a+b < d
     @thm_zero   state = zero  ⇒  mtl = red ∧ itl = red ∧ empty = FALSE
     @thm_right0 state = right0  ⇒ mtl = red ∧ itl = green  ∧ empty = TRUE
     @thm_right  state = right    ⇒ mtl = red ∧ itl = green ∧ empty = FALSE
     @thm_left0  state = left0    ⇒ mtl = green ∧ itl = red  ∧ empty = TRUE
     @thm_left   state = left      ⇒ mtl = green ∧ itl = red ∧ empty = FALSE
     @thm_leftscan state = leftscan  ⇒  mtl = red ∧ itl = red ∧ empty = FALSE
     @inv_abd a+b ≤ d
     @inv_main mtl = red ∨ itl = red
events
  event INITIALISATION
  then
     @act1 a:=0    @act2 b:=0  @act3 mtl:=red @act4 itl:=red
     @act5 ml_out:=off  @act6 il_out:=off
     @act7 semal := FALSE @act71 semar := FALSE
     @act8 semabl := FALSE @act81 semabr := FALSE
     @act11 empty := FALSE
     @act12 state := zero
  end
  event Zero1
     where @grd1 state = zero
           @grd2 a=0 //∧ mtl = red ∧ itl = red
```

```
            @grd3 il_out=on
      then
            @act1 itl ≔ green
            @act2 empty ≔ TRUE
            @act3 state ≔ right0
      end
   event Zero2
      where @grd1 state = zero
            @grd2 a=0 //∧ mtl = red ∧ itl = red
            @grd3 b < d
            @grd4 ml_out=on
      then
            @act1 mtl ≔ green
            @act2 empty ≔ TRUE
            @act3 state ≔ left0
      end
   event Right0
      where @grd1 state = right0
            //@grd2 mtl = red ∧ itl = green
            @grd3 a ≠ 0
      then
            @act1 empty ≔ FALSE
            @act2 state ≔ right
      end
   event Right
      where @grd1 state = right
            //@grd2 mtl = red ∧ itl = green
            @grd3 a = 0
      then
            @act1 itl ≔ red
            @act2 empty ≔ FALSE
            @act3 state ≔ zero
      end
   event Left0
      where @grd1 state = left0
            //@grd2 mtl = green ∧ itl = red
            @grd3 a ≠ 0
      then
            @act1 empty ≔ FALSE
            @act2 state ≔ left
      end
   event Left
      where @grd1 state = left
            //@grd2 mtl = green ∧ itl = red
            @grd3 a+b = d
      then
            @act1 mtl ≔ red
            @act2 state ≔ leftscan
      end
   event Left1
      where @grd1 state = left
            //@grd2 mtl = green ∧ itl = red
            @grd3 a = 0
```

```
    then
          @act1 mtl ≔ red
          @act2 empty ≔ FALSE
          @act3 state ≔ zero
    end
  event Leftscan
    where @grd1 state = leftscan
          //@grd2 mtl = red ∧ itl = red
          @grd3 a = 0
    then
          @act1 empty ≔ FALSE
          @act2 state ≔ zero
    end
  event Left_ap      //ab Left Right
    where @grd1 semal = TRUE
    then
          @act1 a≔a+1
          @act2 semal ≔ FALSE
    end
  event Left_ambp
    where @grd1 semabl = TRUE
          @grd2 a > 0
    then
          @act3 semabl ≔ FALSE
          @act1 a≔a−1
          @act2 b≔b+1
    end
  event Right_am
    where @grd1 semar = TRUE
          @grd2 a > 0
    then
          @act2 semar ≔ FALSE
          @act1 a≔a−1
    end
  event Right_bmap
    where @grd1 semabr = TRUE
          @grd2 b > 0
    then
          @act3 semabr ≔ FALSE
          @act1 a≔a+1
          @act2 b≔b−1
    end
  event ML_out_off          //Sensor control
    where @grd1 a+b≤d
          @grd2 ml_out = off
    then
          @act1 ml_out ≔  on
    end
  event ML_out_on
    where @grd2 ml_out = on
          @grd3 mtl=green
          @grd4 a+b<d
    then
```

```
            @act1 semal ≔ TRUE
            @act2 ml_out ≔ wait
      end
   event ML_out_ap
      where @grd1 a+b≤d
            @grd2 ml_out = wait
            //@grd3 mtl=green
            @grd4 semal = FALSE
      then
            @act1 ml_out ≔ off
      end
   event IL_in_off
      where @grd1 a+b≤d
            @grd2 semabl = FALSE
            @grd3 empty = FALSE
            @grd4 itl = red
            @grd5 a > 0
      then
            @act1 semabl ≔ TRUE
      end
   event ML_in_off
      where @grd1 itl=green
            @grd2 semar = FALSE
            @grd3 empty = FALSE
            @grd4 a > 0
      then
            @act1 semar ≔ TRUE
      end
   event IL_out_off
      where @grd1 a+b≤d
            @grd2 b>0
            @grd3 il_out = off
      then
            @act1 il_out ≔ on
      end
   event IL_out_on
      where @grd1 a+b≤d
            @grd3 il_out = on
            @grd4 itl = green
      then
            @act1 il_out ≔  wait
            @act2 semabr ≔ TRUE
      end
   event IL_out_ap
      where @grd1 a+b≤d
            @grd2 il_out = wait
            //@grd3 itl = green
            @grd4 semabr = FALSE
      then
            @act1 il_out ≔ off
      end
end
```

УДК 519.714.71

Конъюнктивная декомпозиция булевых функций: эксперименты с различными представлениями

Емельянов П.Г. (Институт систем информатики СО РАН, Новосибирский государственный университет)

В статье представлены результаты экспериментов по конъюнктивной декомпозиции различных представлений булевых функций (ZDD, BDD, OKFDD) методами, которые получаются путем специализации общего алгоритма декомпозиции. Тестовыми наборами являются случайные булевы функции с различными параметрами, а также набор широко известных бенчмарков, используемых для тестирования алгоритмов оптимизации логических схем. В сравнении участвуют последовательная и многопоточная реализация алгоритма.

Ключевые слова: *булевы функции, задача оптимизации представления, конъюнктивная декомпозиция с непересекающимися носителями, представления булевых функций.*

1. Введение

Композиционное построение новых объектов - мощное средство современной математики. Примером этого может служить фундаментальная операция декартового произведения. Обращение композиции – декомпозиция – сложных объектов является важнейшим методологическим приемом математики, позволяющий снизить сложность их анализа и преобразований. Декомпозиция некоторого обобщения декартового произведения, известного в компьютерных науках как произведение семейств множеств (Family Product), является одним из важных вариантов декомпозиции, имеющий многочисленные приложения как в фундаментальных областях математики (декомпозиция некоторых алгебраических структур, раскраска гиперграфов и т.д.), так и технологических областях (оптимизация комбинационной части логических схем, декомпозиция таблиц баз данных и таблиц принятия решений, декомпозиция онтологий и т.д.).

Цель данной статьи – представить результаты исследований, относящиеся к изучению свойств конъюнктивной/дизъюнктивной декомпозиции различных представлений булевых функций. Особенностью подхода является то, что алгоритм декомпозиции для конкретного представления получается путем специализации обобщенного алгоритма.

2. Теоретические основания

Произведение семейств множеств определяется следующим, образом. Если A и B – множества подмножеств некоторых непересекающихся конечных базовых множеств, то $A \otimes B = \{ a \cup b \mid a \in A \wedge b \in B \}$. Довольно просто установить связь между произведение семейств множеств и произведением мультилинейных полиномов над полем GF(2) с непересекающимися множествами переменных, если предположить, что мономы представляют элементы A и B.

Таким образом, задача декомпозиции произведение семейств множеств сводится к задаче факторизации указанных полиномов. Впервые полиномиальная сложность задачи факторизации мультилинейный полиномов над конечными полями была указана Шпилькой и Волковичем в 2010 году [10] для полиномов, представленных в виде арифметических схем. Алгоритм (по существу, несколько алгоритмов) полиномиальной временной сложности для полиномов, представленных в виде списка мономов (в точки зрения булевых функций – это представление в АНФ), был получен Емельяновым и Пономаревым в 2014 году [4, 5]. Это дает эффективное средство декомпозиции сразу для нескольких представлений булевых функций, так как эти представления напрямую выражаются с помощью полиномов. С учетом эффективного взятия отрицания и, соответственно, возможности проведения дизъюнктивной декомпозиции, такими представлениями являются полные/частичные таблицы истинности, позитивные ДНФ/КНФ, СДНФ/СКНФ.

Однако теоретическая эффективность не означает практическую. А так как эта задача возникает в важных прикладных областях, среди которых укажем лишь оптимизацию логических схем и анализ данных и извлечение знаний, то ее дальнейшее изучение в этом направлении имеет особую актуальность. Понимание причин, по которым та или иная задача имеет такую сложность, важно для принятия решений при разработке программного обеспечения. Кроме того, на практике представление объектов декомпозиции, оказывает существенное влияние на эффективность декомпозиции/факторизации.

Предыдущие исследования показали, что алгоритм [6], основанный на неявном вычисление формальной производной некоторого полинома, построенного из исходного, может быть адаптирован для произвольных представлений булевых функций и, при выполнении некоторых дополнительных условий, их временная сложность может быть даже полиномиальной. Таким образом, разработан обобщенный алгоритм конъюнктивной

декомпозиции, при соответствующей специализации небольших частей применим различным представлениям (при некоторых предположениях это также гарантирует полиномиальную сложность полученных алгоритмов). Это чрезвычайно актуальное свойство в рамках технологических цепочек, когда представление функций предписано и конвертация их в более "удобное" для декомпозиции представление снижает общую эффективность процесса.

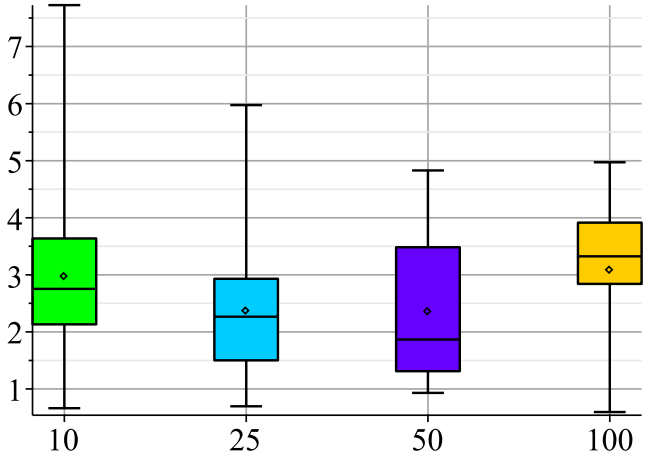
3. Эксперименты с различными представлениями

В рамках проекта по исследованию задачи декомпозиции велись исследования версий алгоритма декомпозиции для важных в рамках синтеза логических схем представлений таких, как частичные таблицы истинности, BDD [2], OKFDD [3], AIG [8]. На данный момент, для применяемых в микроэлектронной индустрии представлений не известны эффективные алгоритмы конъюнктивной/дизъюнктивной декомпозиции с непересекающимися множествами входных переменных (известны довольно быстрые эвристики, которые не гарантируют отсутствие у компонент общих переменных [1]).

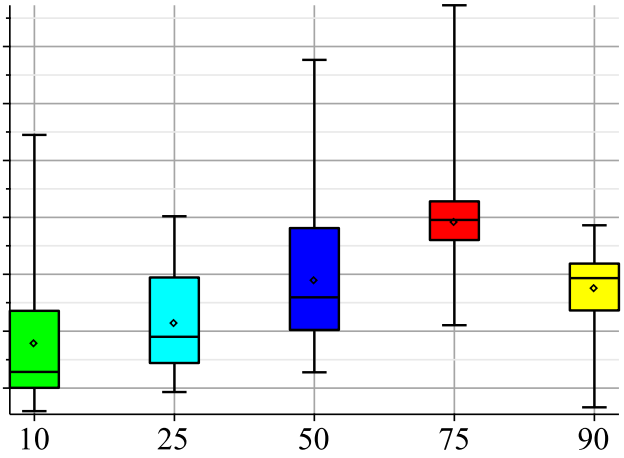
Вначале была выполнена параллельная реализация алгоритма конъюнктивной декомпозиции булевых функций в АНФ форме, где полиномы представлялись в виде ZDD [7]. Для реализации была использована библиотека CUDD 3.0 [11]. Были сгенерированы два набора случайных полиномов, являющихся произведением двух компонент с непересекающимися носителями, со следующими характеристиками. Все содержат 100 переменных, которые разбиваются по компонентам 10 на 90, 25 на 75, 50 на 50, 75 на 25, 90 на 10. Один набор – это полиномы, имеющие104 мономов (по компонентам 10 на 1000, 25 на 400, 50 на 200, 100 на 100), и второй – 105 мономов (100 на 1000, 200 на 400, 316 на 316).

Результаты тестирования первого набора (в вариантах оптимального и случайного упорядочивания переменных и однопоточного и 8-поточного исполнения) представлены на бокс-диаграммах (см. рис. 4). В представленных диаграммах ускорение означает отношение времени исполнения в 8-поточном режиме ко времени исполнения в однопоточном режиме. В общем и целом, результаты тестирования подтверждают гипотезы о зависимости сложностных характеристиках алгоритма от характеристик компонент декомпозиции. Отметим, что для реализации с использованием библиотеки CUDD потребовалась новая по сравнению с реализацией на Maple схема мемоизации результатов вычислений, которая ускорила вычисления на 30-40%, «эффективность кэша» (отношение числа успехов к общему числу обращений) варьировалась от 0.01 до 0.58.

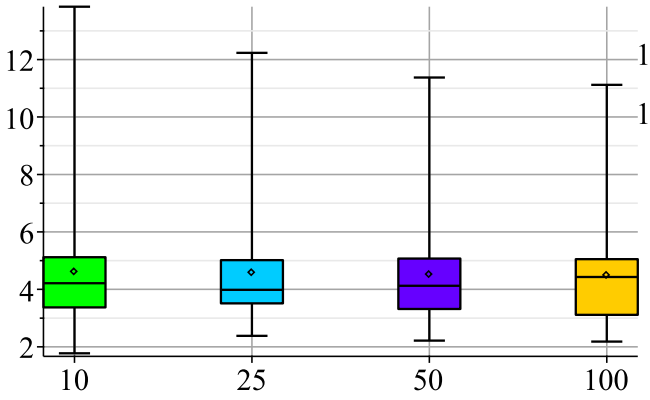
Speedup for the optimal ordering grouped by monom number



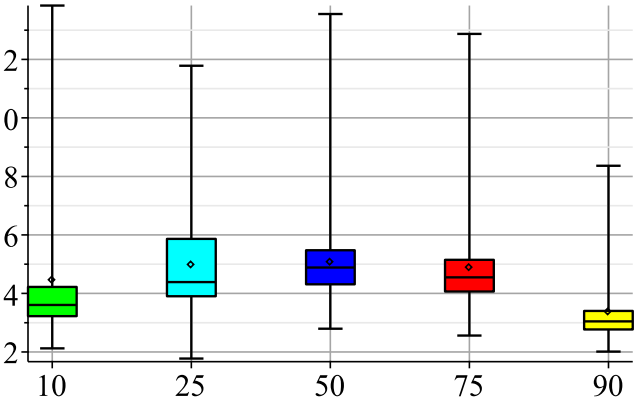
Speedup for the optimal ordering grouped by variable number



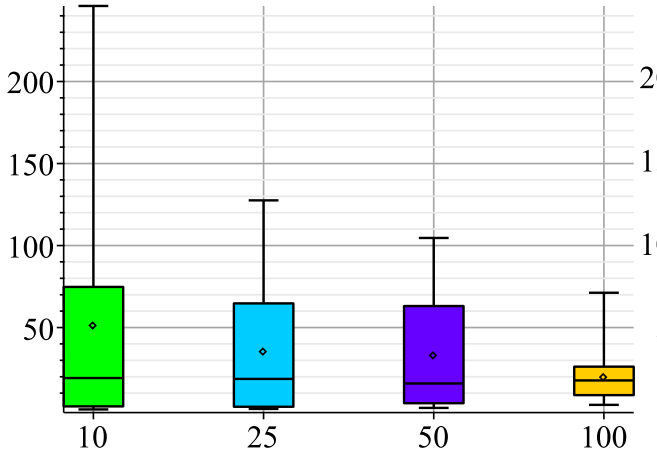
Speedup for a random permutation grouped by monom number



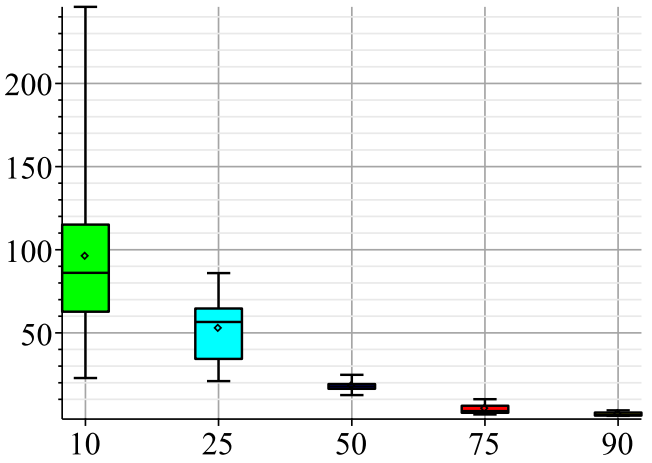
Speedup for a random permutation grouped by variable number



Seq-time for the optimal ordering grouped by monom number



Seq-time for optimal ordering grouped by variable number



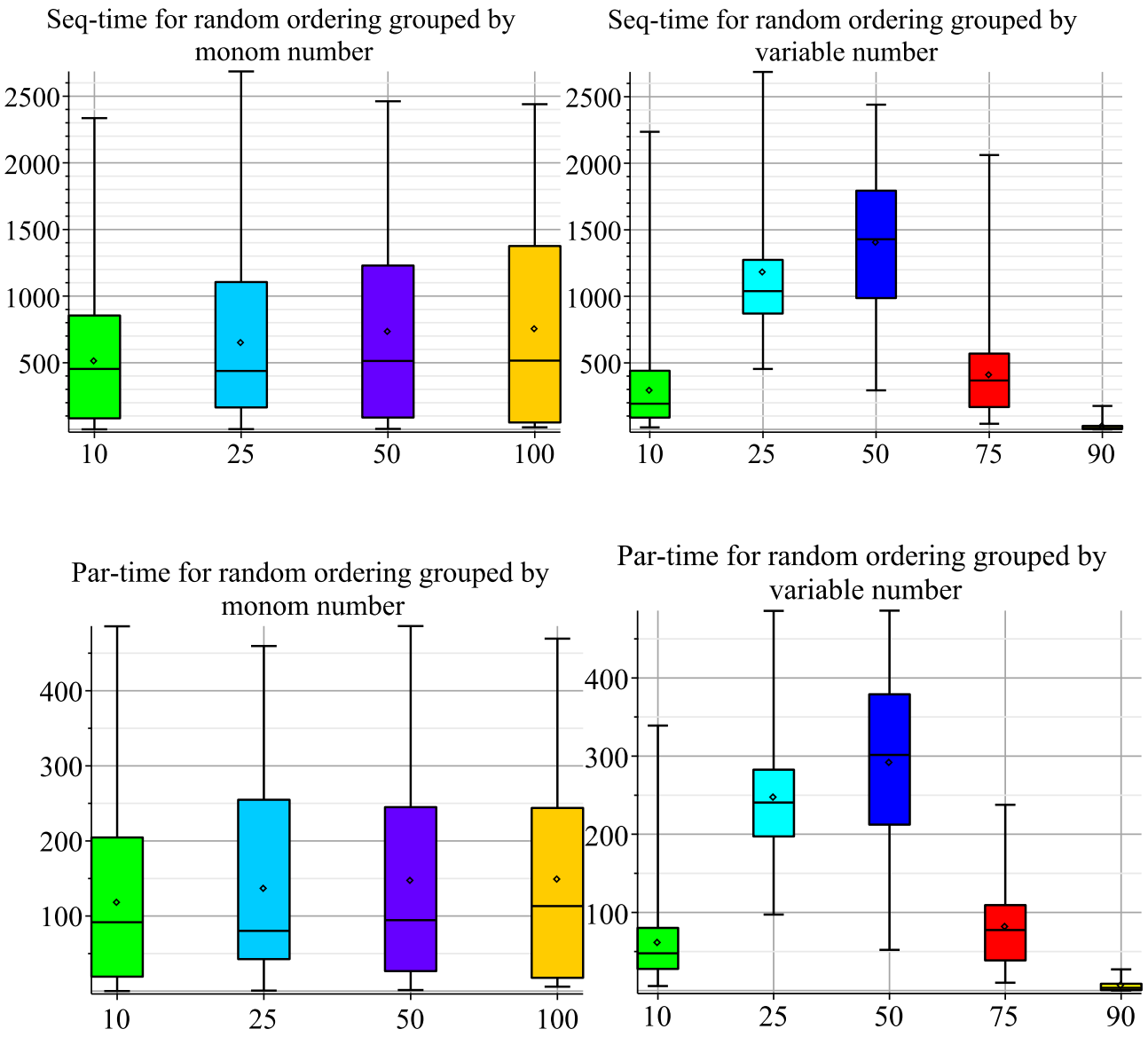


Рисунок 4. Результаты тестирования первого набора полиномов при оптимальном и случайном упорядочивании переменных однопоточного и 8-поточного исполнения

Таблица 1. Характеристики декомпозиции для выходов схем

Бенчмарк	Схема	Тип	№ функции/ имя	Кол-во аргумент.	Кол-во трив. комп.	Кол-во перемен в 1 комп.	Кол-во перемен. во 2 комп.	Время декомпоз. сек.
EPFL	router	or	1/po1	60	0	57	3	0.006
EPFL	mem_ctrl__part0	and	63/po067	113	74	33	6	0.048
ISCAS	s15850__part0	and	34/n560	144	4	115	25	0.034
ISCAS	s15850__part0	and	498/n450	146	1	141	4	0.383
ISCAS	s15850__part0	or	97/n2045	58	8	41	9	0.001
IWLS2005	wb_dma	or	314/n1369	39	0	29	10	0.001
IWLS93	i10__part0	or	135/po191	53	0	49	4	22.453
IWLS93	rot__part0	and	10/po11	45	1	42	2	52.187
LEKO	g625	and	0/po000	500	0	300	200	12632.000
LGSynth91	i2_part0	or	0/po0	201	3	132	69	8935.200
LGSynth91	i10_part0	and	79/po119	48	3	40	5	24.341
LGSynth91	pair_part0	or	4/po004	51	0	28	23	39.557
Other	sudoku_check__part2	and	1/n3454	729,	0	9	720	207620.000
QUIP	oc_minirisc	and	165/n1104	63	4	53	6	1576.636

Далее было решено настроить данный алгоритм для формата BDD, который является де-факто с середины 80-х годов является индустриальным стандартом в области синтеза логических схем. Задача декомпозиции является существенной компонентой этапа анализа булевых функций, составляющих схему, для целей ее оптимизации по времени срабатывания, размеров и энергопотребления. Для сравнения работы алгоритма на случайных функциях был реализован конвертор из текстового представления АНФ в BDD. Для тестирования реальных логических схем был выбран широко известный набор описаний схем в формате BLIF, представленный в работе [9]. Этот набор составлен на основании нескольких широко признанных индустриальных бенчмарков. Из 1314 файлов обработано 1118, для остальных анализ не удалось завершить в виду нехватки памяти. Для 1112 функций обнаружены конъюнктивные разложения. В таблице 1 приведены характеристики декомпозиции для выходов схем, имеющих «много» аргументов, и нетривиальные компоненты разложения (две компоненты имеют носитель, состоящий из не менее 2 переменных). Отметим, что количество входов больше 40 делает невозможным декомпозицию на основе полного перебора.

3. Заключение

В статье представлены результаты экспериментов по конъюнктивной и дизъюнктивной декомпозиции различных представлений булевых функций (ZDD, BDD, OKFDD) методами, которые получаются путем специализации общего алгоритма декомпозиции. Результаты тестирования подтверждают гипотезы о сложностных характеристиках алгоритма. Проведена оценка эффективности мемоизации.

Список литературы

1. Bengtsson T., Martinelli A., Dubrova E., A fast heuristic algorithm for disjoint decomposition of Boolean functions // 11th IEEE/ACM International Workshop on Logic & Synthesis (IWLS’02). New Orleans, USA. 2002, PP. 51–55.
2. Bryant R.E. Graph-Based Algorithms for Boolean Function Manipulation // IEEE Transactions on Computers. 1986. Vol. C-35, № 8. P. 677-691.
3. Drechsler R. and Becker B. Ordered Kronecker functional decision diagrams – a data structure for representation and manipulation of Boolean functions // IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 1998. Vol. 17, № 10. P. 965–973.
4. Emelyanov P. and Ponomaryov D. On the tractability of disjoint AND-decomposition of boolean formulas // PSI 2014: Ershov Informatics Conference / Lecture Notes in Computer Science. Berlin, New York: Springer. 2015. Vol. 8974. P. 92-101. DOI: 10.1007/978-3-662-46823-4_8

5. Emelyanov P. and Ponomaryov D. Algorithmic issues of AND-decomposition of boolean formulas // Programming and Computer Software. 2015. Vol. 41, № 3. P. 162–169. DOI: 10.1134/S0361768815030032. Translated: Programmirovanie, № 3, P. 62-72, 2015.
6. Emelyanov P. and Ponomaryov D. The Complexity of AND-decomposition of Boolean Functions // Discrete Applied Mathematics. 2020. Vol. 280. P. 113–132. DOI: 10.1016/j.dam.2019.07.005.
7. Minato S.-i. Zero–suppressed BDDs for set manipulation in combinatorial problems // 3d International Design Automation Conference (DAC’93). New York: ACM. 1993. P. 272–277.
8. Mishchenko A., Chatterjee S., and Brayton R. DAG-aware AIG rewriting a fresh look at combinational logic synthesis // 43rd annual Design Automation Conference (DAC '06). New York: ACM. 2006. P. 532–535. DOI: https://doi.org/10.1145/1146909.1147048.
9. P. Fišer, J. Schmidt, A comprehensive set of logic synthesis and optimization examples // 12th International Workshop on Boolean Problems (IWSBP’2016). Freiberg, Germany. 2016. P. 151–158.
10. Shpilka A. and Volkovich I. On the relation between polynomial identity testing and finding variable disjoint factors // 37th International Colloquium on Automata, Languages and Programming. Part 1 (ICALP’2010) / Lecture Notes in Computer Science. Berlin, New York: Springer, 2010. Vol. 6198. P. 408-419.
11. Somenzi F. CUDD: CU Decision Diagram Package. University of Colorado at Boulder. 2016. [Электронный ресурс]. URL: https://github.com/ivmai/cudd (дата обращения: 10.03.2022).