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)
```