Моделирование окружения драйверов устройств операционной системы Linux[1]

Захаров И.С., Мутилин В.С., Новиков Е.М., Хорошилов А.В.
{ilja.zakharov, mutilin, novikov, khoroshilov}@ispras.ru

Аннотация. При статической верификации драйверов устройств операционной системы Linux необходимо учитывать особенности взаимодействия драйверов с сердцевиной ядра, так как это взаимодействие оказывает определяющее влияние на работу драйвера. В то же время, верификации драйвера в комбинации с исходным кодом сердцевины ядра не представляется возможной ввиду сложности и объема получающегося кода. В качестве решения этой проблемы в статье предлагается метод моделирования окружения драйверов на основе π-исчисления Р.Милнера и метод трансляции π-модели окружения в программу на языке Си, которая при связывании с исходным кодом драйвера описывает с точки зрения инструментов статической верификации те же сценарии работы драйвера, что и реальное окружение драйвера в операционной системе.

Ключевые слова: операционная система; драйвер; окружение; верификация.

1 Введение

Обеспечение надежности и информационной безопасности программных и программно-аппаратных систем является одной из главных задач современных информационных технологий. Особый вклад в решение этой задачи вносят работы по верификации системного программного обеспечения, в первую очередь компонентов операционных систем (ОС). Верификация драйверов ОС Linux является критически важной задачей, так как:

- Корректность драйверов является необходимым условием обеспечения надежности и безопасности систем, поскольку драйверы работают с тем же уровнем привилегий, что и остальное ядро [1].

[1] Работа поддержана ФЦП "Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы" (контракт N 11.519.11.4006)

- Драйверы ОС Linux, входящие в ядро, это большой, стремительно растущий класс программных систем. На данный момент суммарный объем исходного кода драйверов составляет более 9 миллионов строк. За последний год он увеличился на полмиллиона строк.

1.1. Драйверы операционной системы Linux

В ядре Linux можно выделить «сердцевину» ядра и драйверы (см. Рис. 1). Сердцевина ядра отвечает за управление процессами и памятью, содержит сетевую подсистему и др. Драйверы позволяют ОС и пользовательским приложениям использовать возможности соответствующей аппаратуры. Они составляют большую часть исходного кода ядра – около 70%. Большинство драйверов могут быть скомпилированы как динамически загружаемые модули ядра Linux.

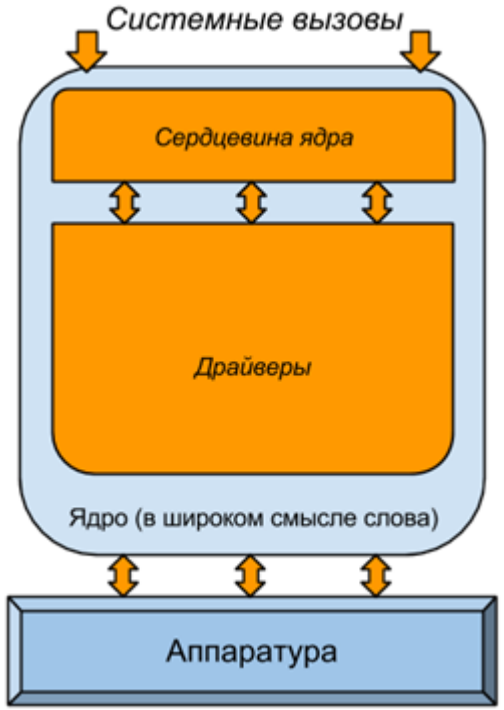

Рис. 1. Драйверы в ядре ОС Linux

Драйверы сильно отличаются от традиционных программ на языке Си: они не имеют функции *main*, а последовательность выполнения кода во многом зависит от взаимодействия драйверов с сердцевиной ядра. Особенности устройства драйвера рассмотрены ниже на примере драйвера USB CDC Phonet

(*cdc-phonet.c*) из сетевой USB подсистемы (*drivers/net/usb*) ядра версии 3.0.1 (см. Рис. 2):

- **Функция инициализации** (далее функция *init*). Драйвер начинает свою работу после загрузки соответствующего модуля в память. Загрузка модуля происходит в процессе старта ОС Linux или после того, как в процессе работы возникла необходимость использовать соответствующее устройство. В процессе загрузки модуля драйвера сердцевина ядра вызывает функцию инициализации драйвера. В примере на Рис. 2 такой функцией является *usbpn_init*. Если инициализация проходит успешно, то возвращается значение «0», в противном случае возвращается соответствующий код ошибки.
- **Функция выхода** (далее функция *exit*). Устройство можно использовать до тех пор, пока соответствующий модуль не будет выгружен из памяти. Перед выгрузкой сердцевина ядра выполняет специальную функцию драйвера *exit*, в которой, например, освобождаются захваченные драйвером ресурсы. На Рис. 2 такой функцией является *usbpn_exit*.

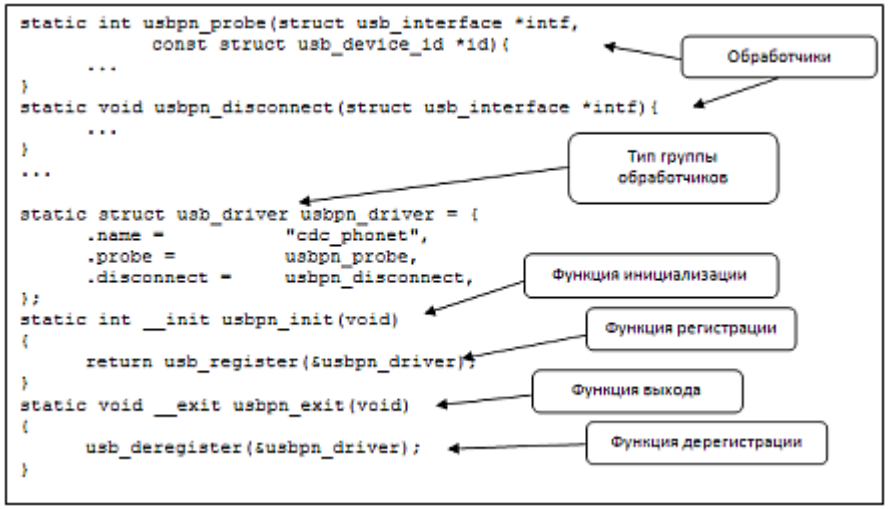

Рис. 2. Выдержки из исходного кода драйвера drivers/net/usb/cdc-phonet.c

- **Функции-обработчики.** Основная функциональность драйвера сосредоточена в функциях-обработчиках событий (далее *обработчики*). Обработчики вызываются сердцевиной ядра, когда требуется обработать события, связанные с драйвером, например, прерывания от устройств или системные вызовы от пользовательских приложений. В примере на Рис. 2 обработчиками являются функции *usbpn_probe* и *usbpn_disconnect*.

- **Группы обработчиков.** Большинство обработчиков объединены в группы. Каждая группа имеет определенный тип в зависимости от назначения ее обработчиков. Порядок вызова обработчиков и параметры их вызова сердцевиной ядра определяются ролями обработчиков в группе. В большинстве случаев, тип группы обработчиков можно определить по типу структуры, в полях которой хранятся функциональные указатели на обработчики (например, на Рис. 2 *usb_driver*). Поля структуры определяют роли, которые выполняют хранящиеся в этих полях указатели на функции-обработчики в конкретном драйвере. Для работы с *USB* устройством в примере на Рис. 2 представлена группа из двух обработчиков *usbpn_probe* и *usbpn_disconnect* с типом *usb_driver*. Данная группа задана при помощи структуры *struct usb_driver* с полями *probe* и *disconnect*.
- **Использование библиотечных функций сердцевины ядра.** Так как сердцевина ядра может вызывать только зарегистрированные обработчики, то наиболее важными библиотечными функциями с точки зрения моделирования окружения являются библиотечные функции регистрации и дерегистрации групп обработчиков драйвера. В начале некоторые обработчики регистрируются в процессе выполнения функции инициализации модуля, остальные регистрируются при вызове обработчиков, зарегистрированных ранее. Дерегистрация групп обработчиков часто осуществляется во время выполнения функции выхода драйвера. В примере на Рис. 2 группа обработчиков *usb_driver* регистрируется при помощи функции *usb_register* и дерегистрируется с помощью функции *usb_deregister*, описанных в заголовочном файле ядра *include/linux/usb.h*. Данные функции вызываются в теле функций инициализации и выхода драйвера. Не все обработчики можно объединить в группы, например обработчики прерываний или функции выполнения отложенных задач в очередях, таймерах, тасклетах и т.д. Подобные обработчики также требуют соответствующей регистрации и дерегистрации и вызываются сердцевиной ядра.

Рассмотренный пример демонстрирует, что работа драйвера существенным образом зависит от внешних событий в системе, которые приводят к вызовам сердцевиной ядра функций инициализации, выхода и функций-обработчиков. Драйвер, в свою очередь, использует библиотечные функции ядра, такие как функции регистрации и дерегистрации.

1.2. Статическая верификация драйверов

Осуществлять поддержку корректности драйверов ОС Linux вручную слишком трудоемкая задача. Одним из направлений ее решения является применение инструментов статической верификации. Работа драйвера

неотделимо связана с работой сердцевины ядра, но анализировать модуль драйвера вместе с ней слишком сложная задача для инструментов верификации на сегодняшний день. Причиной этому являются высокая сложность и большой объем исходного кода сердцевины ядра. Для решения этой проблемы драйвер нужно заключить в некоторое «окружение» с тем, чтобы программный код был замкнутым. Такое окружение будем называть *синтезируемым окружением*. К синтезируемому окружению предъявляются две группы требований.

Первая группа требований определяет ограничения на конструкцию (структуру) синтезируемого окружения, обусловленные возможностями современных инструментов статической верификации. Так, большинство из них работает с последовательными программами на языке программирования Си, имеющими точку входа (аналог функции *main*), с которой начинается выполнение. Кроме того, в значительной части имеются ограничения по точности анализа функциональных указателей, рекурсивных структур данных и т.д.

Наибольший интерес представляет вторая группа требований к окружению, связанная с тем, что оно должно воспроизводить те же сценарии взаимодействия с драйвером, что и реальное окружение драйвера в операционной системе. Чтобы не упустить ошибки, окружение должно быть полным, т.е. включать все последовательности событий, которые могут происходить в реальной системе с учетом всевозможных событий со стороны пользователей и аппаратуры. При этом окружение должно быть корректным.

Основными видами требований корректности к синтезируемому окружению являются:

1. Требования к порядку вызова функций *init* и *exit*.
2. Требования к вызову обработчиков, принадлежащих одной группе, включающие:
    а) Ограничения на параметры вызова обработчиков;
    б) Ограничения на порядок вызова обработчиков.
3. Требования к порядку и к параметрам вызова обработчиков из разных групп обработчиков.
4. Требования к моделированию контекста вызова обработчиков (учет возможности вызова прерываний, учет захваченных сердцевиной ядра блокировок и др.).

Нарушение требований корректности может приводить к ложным сообщениям об ошибках, т.е. ситуациям, когда инструмент статической верификации сигнализирует об ошибке, но ошибка в реальном окружении не воспроизводима.

Примером требований к порядку вызовов в рамках одной группы обработчиков типа *usb_driver* является ограничение на порядок вызовов функций с ролями *probe* и *disconnect*. Функция с ролью *disconnect* может быть вызвана, только если до этого был сделан успешный вызов функции с ролью

probe. Также при вызове *disconnect* должен передаваться тот же указатель на структуру *usb_interface*, которая была успешно инициализирована функцией *probe*. Если в группу типа *usb_driver* входят функции с ролями *suspend, resume*, то их вызов возможен только после успешного вызова *probe* и до вызова *disconnect*. Другой пример: перед вызовом некоторых функций требуется, чтобы был захвачен мьютекс.

В качестве примера требований на порядок вызова обработчиков из разных групп, возьмем группы обработчиков типа *usb_driver* и *file_operations*. Вызов функций из группы типа *file_operations* возможен, только после успешной регистрации данной группы в обработчике с ролью *probe* из группы типа *usb_driver*.

Другой пример требований: после успешного вызова *probe* окружение не должно вызывать функцию выхода *exit* до вызова *disconnect*.

1.3. Моделирование окружения драйверов

Окружение моделирует работу реальной сердцевины ядра, которая получает события как из пространства пользователя в виде системных вызовов, так и от аппаратуры. В зависимости от приходящих событий, окружение выбирает один из доступных обработчиков драйвера и вызывает его, а кроме того, изменяет свое состояние. В процессе выполнения обработчика, драйвер вызывает библиотечные функции окружения, например, вызывает функции регистрации новых групп обработчиков, которые становятся доступными для вызова из окружения.

Окружение и драйвер можно рассматривать как процессы в π-исчислении Р.Милнера [7]. Процесс выполняет шаги, каждый из которых – это прием или посылка сигнала и переход к следующему процессу. Вызов обработчиков драйвера можно рассматривать как посылку сигнала от окружения к драйверу, а возвраты значений как посылку сигнала от драйвера к окружению. Получая сигнал, драйвер осуществляет его обработку, при этом он может обращаться к библиотечным функциям сердцевины ядра, что также можно рассматривать как посылку и прием сигналов.

Переход к следующему процессу моделирует изменение состояния, в частности за счет этого моделируется порядок вызова обработчиков. Например, в реальном окружении до вызова обработчика с ролью *probe*, оно находится в состоянии, в котором нельзя вызывать обработчик с ролью *disconnect*. Это моделируется процессом, принимающим только сигнал *probe*. Как только окружение получит от драйвера сигнал с успешным кодом возврата из *probe*, обработчик с ролью *disconnect* можно будет вызывать. Это моделируется переходом по приему сигнала успешного возврата к следующему процессу, в котором доступен сигнал *disconnect*. Тем самым обеспечивается порядок вызова обработчиков с ролями *probe* и *disconnect*.

Предлагаемый в данной статье метод построения синтезируемого окружения состоит из двух шагов (см. Рис. 3). На первом шаге строится π-модель реального окружения. Эта модель учитывает требования полноты и корректности, накладываемые на окружение. Для формального описания модели используется π-исчисление Р.Милнера [7]. С помощью данного исчисления удается довольно компактно и наглядно описывать последовательности вызовов обработчиков драйвера (сценарии воздействия на драйвер). Кроме того, π-исчисление позволяет учитывать многопоточность окружения и асинхронность взаимодействий драйвера с окружением.

На втором шаге из π-модели окружения генерируется код на языке Си. Для исходного кода драйвера генерируется обертка, которая преобразует сигналы π-модели в вызовы обработчиков драйвера так, чтобы драйвер соответствовал своей π-модели. В результате на выходе имеется программа, которая соответствует π-модели, и при этом является однопоточной и отвечает ограничениям инструментов верификации.

Поскольку трансформация многопоточной π-модели в однопоточную Си программу не является тривиальной, возникает необходимость доказать корректность построения синтезируемого окружения. Дается определение эквивалентности между моделью в π-исчислении и программой на языке Си, в котором учитываются трассы (последовательности событий), содержащие только общие действия для процессов драйвера и окружения с ограничением на переключения при выполнении обработчиков драйвера. Далее приводится теорема о том, что для каждого драйвера в предположении, что код драйвера вместе с обертками эквивалентен его π-модели, π-модель эквивалентна синтезированной Си программе.

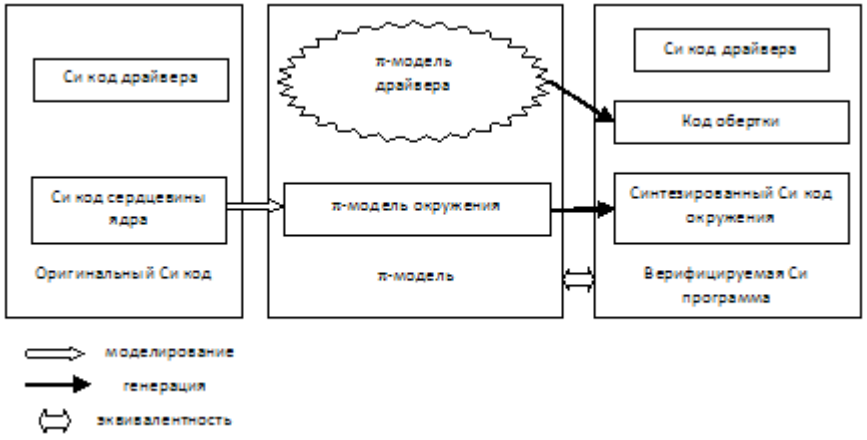

Рис. 3. Моделирование окружения драйверов

2. Основные определения

Введем понятия, используемые в π-исчислении. Синтаксис π-исчисления позволяет представлять *процессы* – абстракции независимых потоков управления. *Исчисление* позволяет задавать параллельную композицию процессов, синхронное взаимодействие между процессами с помощью сигналов, создание процессов и недетерминизм. Сигнал задается *меткой* (*каналом*) – абстракция связи между двумя процессами, с помощью которой осуществляется взаимодействие.

Сигналы могут иметь параметры. В определении «The Polyadic π-Calculus: a Tutorial, Robin Milner, October 1991» [7] допускается передача любых меток в качестве параметра сигнала.

Множество меток приема сигналов обозначим как A. Множество меток посылки сигнала обозначим как $\bar{A}$, оно состоит из меток $\bar{\alpha}$, где $\alpha \in A$. Обозначим $\Lambda = A \cup \bar{A}$.

Множество действий включает в себя действия посылки/получения для меток, а также специальное пустое действие τ, $Act = \Lambda \cup \tau$.

Непустые действия могут иметь параметры, которые записываются в круглых скобках. Жирным шрифтом будем обозначать вектор параметров, например $\boldsymbol{x}$.

Для действий посылки сигнала $\overline{\alpha(e(\boldsymbol{x}))}$ вычисляется значение выражения $e(\boldsymbol{x})$, на которое заменяется переменная $\boldsymbol{y}$ в действии приема сигнала $\alpha(\boldsymbol{y})$.

Кроме того, в определении для удобства записи добавлен оператор *if-then-else*.

Определение 1.

Процессы верхнего уровня:

$$P := P_1 \mid P_2 \mid !P \mid (\nu\alpha)P \mid N$$

где

- $P_1 \mid P_2$ – параллельная композиция, одновременно выполняются процессы P_1, P_2;
- $(\nu\alpha)$P – создание новой метки $\alpha \in A$;
- $!P$ – создание копии процесса;
- N – процесс нижнего уровня.

Процессы нижнего уровня:

$$N(\boldsymbol{x}) := 0 \mid K_1 + \cdots + K_n \mid \; if\; b(\boldsymbol{x})\; then\; N_1(\boldsymbol{x})\; else\; N_2(\boldsymbol{x})$$

- 0 – пустой процесс;
- $K_1 + \cdots + K_n$ – выбор, где процесс может продолжаться одним из вариантов:
  - $N_i = \alpha_i(y_i).K_i(e_i(\boldsymbol{x}, y_i))$, $\alpha_i \in \Lambda$ – прием сигнала;

- $N_i = \overline{\alpha_i(e(\boldsymbol{x}))}.K_i(e_i(\boldsymbol{x}))$ – посылка сигнала;
- $N_i = K_i(e_i(\boldsymbol{x}))$ – τ действие;

- $if\ b(\boldsymbol{x})\ then\ N_1(\boldsymbol{x})\ else\ N_2(\boldsymbol{x})$ – если условие $b(\boldsymbol{x})$ выполнено, то процесс продолжается как $N_1(\boldsymbol{x})$, иначе как $N_2(\boldsymbol{x})$.

Для сокращенного задания идентификаторов процессов нижнего уровня будем использовать запись:

$$N_j(\boldsymbol{x}) := \cdots.\langle N_i\rangle \text{ выражение1 ...}$$

которая означает:

$$N_j(\boldsymbol{x}) := \cdots.N_i(\boldsymbol{x}) \text{ ...}$$

$$N_i(\boldsymbol{x}) := \text{ выражение1}$$

Как можно видеть, в определении на верхнем уровне предусмотрены конструкции порождения новых процессов, создание новых меток. Процессы верхнего уровня могут быть запущены параллельно.

Для каждого процесса верхнего уровня определяются процессы нижнего уровня. Так как параллельная композиция определена для процессов верхнего уровня, то процессы нижнего уровня можно рассматривать как состояния соответствующего процесса верхнего уровня. Осуществляя переходы от одного процесса нижнего уровня к следующему процессу, процесс верхнего уровня меняет свое состояние.

Конструкция выбора $K_1 + \cdots + K_n$ позволяет задавать множество доступных для посылки сигналов и множество сигналов, которые процесс готов принять. При выборе одного из действий процесс переходит в следующее состояние, тем самым меняются множества доступных сигналов. Это позволяет моделировать различные ограничения на порядок вызова обработчиков. Моделируется это с помощью процессов нижнего уровня следующим образом. При получении сигнала вызова, для которого нужно наложить ограничение на порядок вызова, мы переходим к следующему процессу, в котором доступны уже другие сигналы в соответствии с допустимыми порядками вызова. Например, как только мы получили сигнал *probe* в некотором процессе N_1, он переходит к процессу N_2, в котором множество доступных действий включает посылку сигнала *disconnect*.

3. Формальная модель драйвера и его окружения в π-исчислении

Рассмотрим моделирование окружения с помощью π-исчисления. При моделировании драйвер рассматривается как набор процессов, запущенных параллельно:

$$Drv_\pi := P_{init} \mid P_{exit} \mid ! P_{fcall}$$

Процессы P_{init} и P_{exit} представляют функции инициализации *init* и *exit* соответственно. Эти процессы принимают на вход сигналы *init(ret)*, *exit(ret)* и отвечают на них $\overline{ret(x)}$, $\overline{ret()}$ соответственно. Процесс P_{fcall} представляет обработчики драйвера, которые окружение может вызывать после регистрации. Эти процессы принимают сигнал вызова функции обработчика $\overline{f(ret_i, f_i, ctx_i, params_i)}$, в параметрах сигнала которого передаются: ret_i – метка сигнала для ответа, f_i – вызываемая функция, ctx_i – контекст вызова, $params_i$ – параметры вызова. Причем, так как обработчики могут выполняться параллельно, для каждого вызова порождается отдельная копия процесса. В ответ P_{fcall} посылает сигнал с возвращаемым значением $ret_i(result_i)$.

Во время выполнения инициализации, выхода, а также при выполнении других обработчиков драйвер может:

- обращаться к библиотечным функциям сердцевины ядра $\overline{g_1(ret_1, params_1)}, \ldots, \overline{g_l(ret_l, params_l)}$ и получать от них ответы $ret_1(result_1), \ldots, ret_l(result_l)$,
- обращаться к глобальным переменным, с помощью сигналов $\overline{set_{v_i}(x)}, get_{v_i}(x)$ к процессам P_{v_i}.

Таким образом, π-модель окружения предоставляет обработку:

- вызовов библиотечных функций $g_1(ret_1, params_1), \ldots, g_l(ret_l, params_l)$;
- обращений к глобальным переменным, с помощью сигналов $\overline{set_{v_i}(x)}$, $get_{v_i}(x)$.

А также осуществляет вызовы:

- функций инициализации и выхода драйвера $\overline{init(ret)}$, $\overline{exit(ret)}$;
- функций-обработчиков для зарегистрированных групп $\overline{f(ret_i, f_i, ctx_i, params_i)}$.

В самый начальный момент времени активируется основной процесс окружения P_{module}, отвечающий за инициализацию и выход. Рассмотрим пример главного процесса окружения P_{module}, который вызывает функции инициализации и выгрузки драйвера:

$P_{module} \coloneqq L0$

$L0 \coloneqq (vret)L1$

Процесс *L1* посылает драйверу сигнал инициализации *init*. Драйвер осуществляет инициализацию, посылая сигналы сердцевине ядра. Например, для выделения необходимых ресурсов и регистрации группы функций обработчиков драйвера. Если она прошла успешно, то посылается сигнал ret с кодом ответа 0, в противном случае, посылается соответствующий код ошибки:

$L1 \coloneqq \overline{init(ret)}.\langle L3\rangle ret(r).\langle L4\rangle if\ r == 0\ then\ L2\ else\ 0,$

где у процесса *L4* имеется параметр *r*, через который передается возвращаемое значение функции *init*.

Окружение взаимодействует с процессом драйвера до тех пор, пока не будет послан сигнал *exit*. Обрабатывая данный сигнал, драйвер посылает сигналы окружению, например, для освобождения занятых ресурсов и дерегистрации групп обработчиков.

$L2 \coloneqq \overline{mstop}.\langle L5\rangle\overline{exit(ret)}.\langle L6\rangle ret.0$

Процесс P_{module} обращается к процессу $P_{trymoduleget}$, который моделирует захват модуля драйвера, так чтобы его нельзя было выгрузить, т.е. окружение не вызвало функцию выхода *exit*. Процесс, захватывающий модуль драйвера, посылает сигнал $\overline{tmg}$, и получает ответ $tmg^{ret}(true)$ в случае успешного захвата модуля. Сигнал *mstop* нужен для того, чтобы процесс не допускал новые захваты.

$P_{trymoduleget} \coloneqq M(0)$

$M(0) \coloneqq tmg.\langle M1\rangle\overline{tmg^{ret}(true)}.M(1) + mstop.MD$

Для $i \geq 1$:

$M(i) \coloneqq tmg.\langle M2\rangle\overline{tmg^{ret}(true)}.M(i+1) + mput.\langle M3\rangle\overline{mput^{ret}}.M(i-1)$

$MD \coloneqq mstop.MD + tmg.\langle MD1\rangle\overline{tmg^{ret}(false)}.MD$

На Рис. 4 показаны изображения процессов P_{module} и $P_{trymoduleget}$ в виде графов. В вершинах изображены состояния процессов нижнего уровня, на дугах показаны события посылки и приема сигналов. Процесс $P_{trymoduleget}$ имеет потенциально неограниченное число состояний соответствующих параметру *i*.

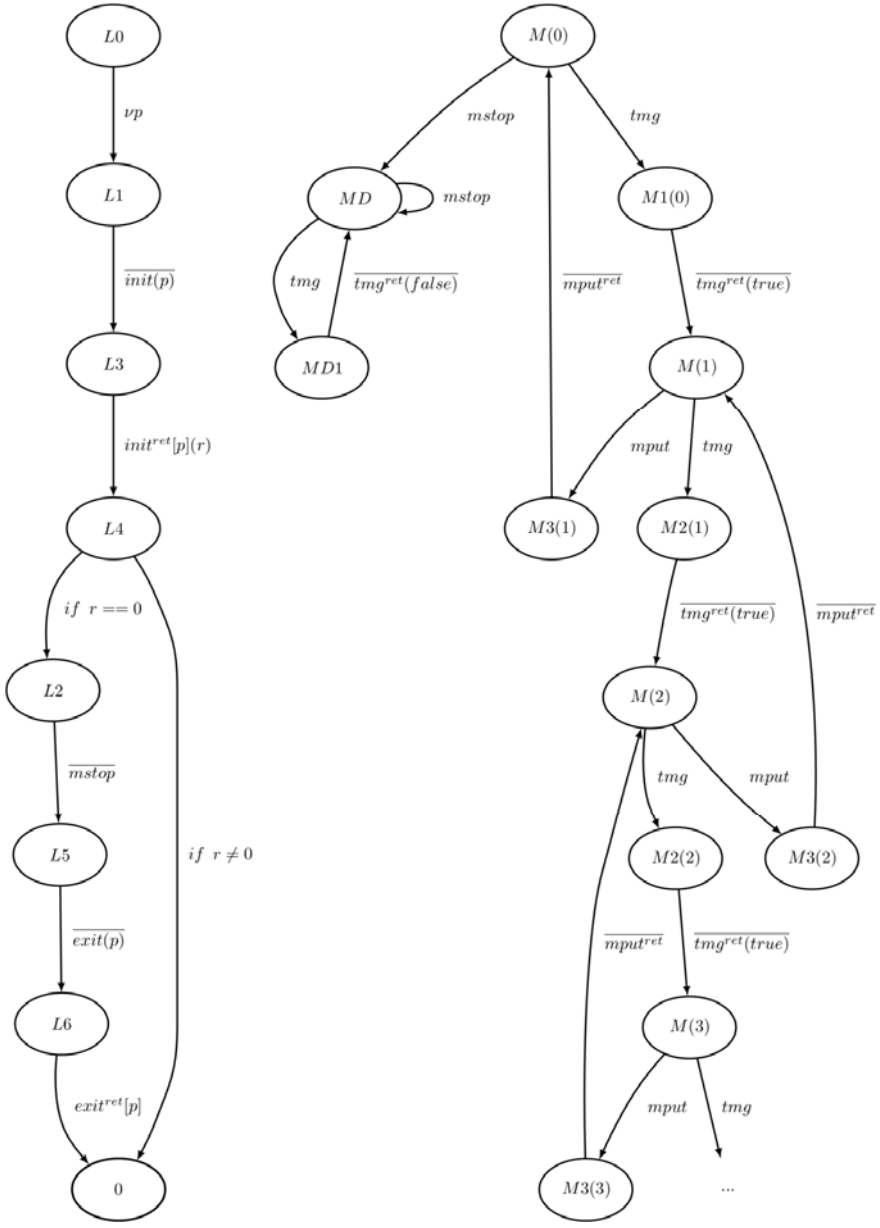

Рис. 4. Пример процесса P_{module} (слева) и процесса $P_{trymoduleget}$ (справа)

Регистрация групп обработчиков осуществляется с помощью библиотечных функций сердцевины ядра. В библиотечные функции передается группа обработчиков, как правило, в виде структуры языка Си из указателей на функции обработчики.

В примере драйвера на Рис. 2 вызывается функция-регистрации *usb_register*, которой передается указатель на переменную *usbpn_driver* типа *usb_driver*, содержащую обработчики *probe* и *disconnect*.

Группа обработчиков работает с некоторыми сущностями ядра, и в зависимости от этого набора сущностей создается соответствующий набор экземпляров процессов группы обработчиков. Например, для группы обработчиков *file_operations*, предназначенной для работы с файлами, такой сущностью является файл, который может быть в состояниях открыт и закрыт. В открытом состоянии хранится текущее смещение внутри файла, используемое операциями чтения и записи.

Для того чтобы моделировать порождение сущностей, с которыми осуществляются операции вводится специальный процесс регистрации. Для каждой сущности он создает процесс группы, который осуществляет вызовы обработчиков из группы с данной сущностью. Если известно для скольких сущностей предназначена группа обработчиков, то создается соответствующее количество экземпляров процесса группы. Неограниченное число экземпляров может моделироваться как недетерминированное порождение новых сущностей. Если имеется единственная сущность, то создается единственный процесс группы.

Рассмотрим пример моделирования регистрации для группы обработчиков типа *usb_driver*. В этом примере для группы будем создавать единственный экземпляр P_{usb_driver} процесса для каждой регистрируемой группы, поэтому специального процесса регистрации не требуется.

$$P_{usb\_driver} \coloneqq S0$$

$$S0 \coloneqq\ !usb\_register(usb\_deregister, probe, disconnect).S1$$

Для краткости далее параметры *probe* и *disconnect* опускаются.

$$S1 \coloneqq\ usb\_deregister.0\ +\ \overline{tmg}.\langle S4\rangle tmg^{ret}(r).\langle S5\rangle if\ r\ then\ S2\ else\ S1$$

$$S2 \coloneqq\ \overline{f(ret, probe)}.\langle S6\rangle ret(r).$$

$$\langle S7\rangle\ if\ r == 0\ then\ S3\ else\ S2\ +\ usb\_deregister.\langle S8\rangle\overline{mput}.0$$

$$S3 \coloneqq\ \overline{f(ret, disconnect)}.\langle S9\rangle ret.S2$$

Для моделирования зависимостей, как между группами, так и зависимостей с функциями инициализации и выхода используются сигналы между процессами. В примере на Рис.2 модуль драйвера нельзя выгружать после того как процесс перешел в состояние $S2$, в котором он готов вызывать

обработчики *probe* и *disconnect*. С точки зрения окружения, это означает, что нельзя вызывать функцию выхода *exit* в основном процессе окружения P_{module}. Для хранения состояния счетчика захватов модуля используется процесс $P_{trymoduleget}$.

Поэтому перед тем, как вызывать какие либо обработчики, процесс группы типа *usb_driver* посылает сигнал $\overline{tmg}$. В случае успеха, он переходит в состояние $S2$. В случае, если начата выгрузка модуля, то процесс остается в состоянии $S1$ и ожидает дерегистрацию, тем самым не осуществляя вызовы обработчиков группы usb_driver.

Другой пример взаимодействия процессов можно продемонстрировать на примере группы *file_operations*. Как только файл переходит в открытое состояние после успешного вызова *open*, необходимо обеспечить невозможность выгрузки модуля. Для этого также используется процесс $P_{trymoduleget}$.

Ограничения на порядок вызова моделируются с помощью процессов S1, …, S9. Например, для установления порядка между *probe* и *disconnect*, как только мы получили сигнал *probe* в процессе $S2$, он переходит к процессу $S3$, в котором множество доступных действий включает посылку сигнала *disconnect*.

Требования к порядку вызова обработчиков нескольких групп учитываются за счет передачи сообщений между процессами групп.

4. Трансляция процессов в Си-программу

Трансляция процессов описана в приложении А диссертации [8]. В разделе А.1 описан общий метод трансляции в многопоточную программу, а в разделе А.2 описывается его модификация для трансляции в последовательную Си программу.

Для упрощения трансляции в диссертации [8] используется модифицированное определение π-процессов. Вводится разделение на статические и динамические метки. Статические – это метки, которые не передаются в качестве параметров сигнала. Данные метки были введены, так как для них можно упростить генерируемый код.

Вместо того чтобы передавать непосредственно сами метки в качестве параметра, вводятся и передаются специальные параметры меток со значениями из потенциально бесконечного множества П. Метки, которые имеют специальные параметры, называются динамическими, обозначим $\Phi = F \cup \bar{F}$, где F – метки приема и $\bar{F}$ – метки посылки. Специальные параметры метки пишутся в квадратных скобках после метки.

Динамически связываемые метки приема сигнала $f[p_1]$ и посылки сигнала $\bar{f}[p_2]$ осуществляют взаимодействие, только если значения параметров меток совпадают. Т.е. процесс получатель принимает сигнал $f[p_1]$, только если

значение передаваемого параметра метки p_1 совпадает со значением получателя p_2. Процесс отправитель считает сигнал $\bar{f}[p_2]$ отправленным, только если нашелся получатель с совпадающим значением параметра метки p_1.

Отметим, что введение динамических меток необходимо нам лишь для удобства записи трансляции в Си программу. Любую модель, записанную в классических π-процессах, можно свести к π-процессам с динамическими метками. Метки, не передаваемые как параметры, записываются как статические. Каждой метке, передаваемой через параметры, ставим в соответствие натуральное число $p \in \Pi$. Вводим динамическую метку *f*. Заменяем все места, в которых использовались метки для приема и посылки на динамическую метку *f[p]* с соответствующим параметром p. Там, где метка передавалась в качестве параметра, передаем значение p.

Например, если имеются процессы:

$P := \nu x\ \overline{a(x)}.x()$

$Q :=\ a(x).\overline{x()}$

то их можно переписать как:

$P := \nu p\ \overline{a(p)}.f[p]()$

$Q := a(p).\overline{f[p]()}$

4.1. Отношение редукции

При первом прочтении можно пропустить определение отношения редукции, которая требуется для задания формальной семантики π-процессов с динамическими метками и формулировки теоремы.

Вспомогательное отношение структурной эквивалентности ≡ определяется аналогично [7].

Определим отношение редукции → над процессами с динамическими метками аналогично [7], при этом пометим пары в отношении метками α из $A \cup F \cup \{\tau\}$, т.е. метками приема сигналов и пустой меткой τ.

Отношение редукции $\xrightarrow{\alpha}$ над процессами – это наименьшее отношение, удовлетворяющее следующим правилам:

- Если есть определение процессов $K(\boldsymbol{x})$ и $N(\boldsymbol{x})$, в каждом из которых есть слагаемое, которое может взаимодействовать с другим, то это взаимодействие осуществляется с меткой приема и значениями

переданных параметров.

$$COMM_1(\alpha \in A):\ \frac{\begin{matrix}K(\boldsymbol{x}) := \left(\cdots + \alpha(y).K_i\left(e_i(\boldsymbol{x},y)\right)\right)\\ N(\boldsymbol{x}) := \left(\cdots + \overline{\alpha(e(\boldsymbol{x}))}.N_j\left(e_j(\boldsymbol{x})\right)\right)\end{matrix}}{K(\boldsymbol{u}) \mid N(\boldsymbol{v}) \xrightarrow{\alpha(e(\boldsymbol{v}))} K_i\left(e_i(\boldsymbol{u},\ e(\boldsymbol{v}))\right) \mid N_j(e_j(\boldsymbol{v}))}$$

- Для динамических меток сравниваются также значения параметров ($\boldsymbol{x}_i$ – i-ый элемент вектора $\boldsymbol{x}$).

$$COMM_2(\alpha \in F):\ \frac{\begin{matrix}K(\boldsymbol{x}) := \left(\cdots + \alpha[\boldsymbol{x}_i](y).K_i\left(e_i(\boldsymbol{x},y)\right)\right)\\ N(\boldsymbol{x}) := \left(\cdots + \overline{\alpha[\boldsymbol{x}_i](e(\boldsymbol{x}))}.N_j\left(e_j(\boldsymbol{x})\right)\right)\\ \boldsymbol{u}_i = \boldsymbol{v}_j\end{matrix}}{K(\boldsymbol{u}) \mid N(\boldsymbol{v}) \xrightarrow{\alpha[\boldsymbol{u}_i](e(\boldsymbol{v}))} K_i\left(e_i(\boldsymbol{u},\ e(\boldsymbol{v}))\right) \mid N_j(e_j(\boldsymbol{v}))}$$

- Переход по τ действию может осуществляться всегда.

$$TAU:\ \frac{K(\boldsymbol{x}) := \left(\cdots + \tau.K_i\left(e_i(\boldsymbol{x})\right)\right)}{K(\boldsymbol{u}) \xrightarrow{\tau} K_i\left(e_i(\boldsymbol{u})\right)}$$

- if-then-else оператор.

$$IFTHEN:\ \frac{\begin{matrix}K(\boldsymbol{x}) := \boldsymbol{if}\ b(\boldsymbol{x})\ \boldsymbol{then}\ K_1(\boldsymbol{x})\ \boldsymbol{else}\ K_2(\boldsymbol{x})\\ b(\boldsymbol{u}) = TRUE\end{matrix}}{K(\boldsymbol{u}) \xrightarrow{\tau} K_1(\boldsymbol{u})}$$

$$IFELSE:\ \frac{\begin{matrix}K(\boldsymbol{x}) := \boldsymbol{if}\ b(\boldsymbol{x})\ \boldsymbol{then}\ K_1(\boldsymbol{x})\ \boldsymbol{else}\ K_2(\boldsymbol{x})\\ b(\boldsymbol{u}) = FALSE\end{matrix}}{K(\boldsymbol{u}) \xrightarrow{\tau} K_2(\boldsymbol{u})}$$

- В параллельной композиции редукция компонентов может осуществляться независимо.

$$PAR:\ \frac{K(\boldsymbol{u_1}) \xrightarrow{\alpha} K'(\boldsymbol{u_2})}{K(\boldsymbol{u_1}) \mid N(\boldsymbol{v}) \xrightarrow{\alpha} K'(\boldsymbol{u_2}) \mid N(\boldsymbol{v})}$$

- Создание нового значения.

$$RES:\ \frac{K(\boldsymbol{u_1}) \xrightarrow{\alpha} K'(\boldsymbol{u_2})}{(\nu x)K(\boldsymbol{u_1}) \xrightarrow{\alpha} (\nu x)K'(\boldsymbol{u_2})}$$

- Редукция для структурно эквивалентных процессов.

$$STRUCT:\ \frac{K \equiv N;\ N(\boldsymbol{u_1}) \xrightarrow{\alpha} N'(\boldsymbol{u_2});\ N' \equiv K'}{K(\boldsymbol{u_1}) \xrightarrow{\alpha} K'(\boldsymbol{u_2})}$$

Понятие редукции позволяет нам определить множество возможных трасс выполнения π процесса *P*, обозначаемое как $traces_\pi(P)$. Для редукции

$P \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} P_2 \cdots \xrightarrow{\alpha_n} P_n$, определим трассу как подпоследовательность $\alpha_{i_1}, \ldots, \alpha_{i_k}$ последовательности $\alpha_1, \ldots, \alpha_n$, включающую все непустые действия $\alpha_{i_j} \neq \tau$.
Тогда $traces_\pi(P)$ – это множество трасс для всех возможных редукций процесса *P*. Определим $traces_\pi(P, D)$, где $D \subseteq F \cup A$, как множество трасс, в которых уделены метки не из множества *D* (т.е. включаем только $\alpha_{i_j} \in D$).

4.2. Общий метод трансляции в многопоточную программу

На входе общего метода трансляции А.1 имеется модель окружения в виде π-процессов и исходный код драйвера. В модели окружения есть два набора процессов верхнего уровня: первые – статические, существующие в единственном экземпляре и не использующие сигналы создания копии процесса, вторые – динамически порождаемые, создающиеся при получении сигнала копирования (например $!\,create_1(x_1).C_1(x_1)$).
У процесса верхнего уровня есть состояние, которое при трансляции моделируется структурой, например для процесса P_{module} будет сгенерирована структура *L_state*:

```
struct L_state {
  struct list list;
  enum L_states state;
  int p;
  int L4_param;
}
```

Состояния хранятся в списке *L_state_list* для того, чтобы иметь возможность порождать неограниченное количество копий процесса. Поле *state* – отражает текущий процесс нижнего уровня или дополнительное состояние *STOP*, означающее процесс 0. Например, для процесса P_{module} это *enum L_states {STOP, L1, L2, L3, L4, L5}*. Поле p – параметр динамических меток, генерируемый при создании процесса с помощью vp. Также структура включает в себя параметры процессов нижнего уровня. Например, *L4_param* – параметр процесса L4, которому передается возвращаемое значение функции init.
Порождение экземпляра процесса происходит либо в функции main, если процесс статический, либо при приеме сигнала создания копии процесса, если процесс динамический.

Функция, отвечающая за создание процесса, выделяет память под его состояние, инициализирует его функцией *L_init* и добавляет в список состояний *L_state_list*, например для процесса P_{module}:

```
void create_L(int p) {
  struct L_state *s = malloc(sizeof(struct L_state));
  L_init(p, s);
  list_add(&L_state_list, &s->list)
  pthread_create(L_thread, s);
}
```

Функция *L_init* инициализации устанавливает начальное состояние L1.

```
L_init(int p, struct L_state *s) {
 s->state = L1;
 s->p = p;
}
```

Далее в функции *create_L* создается поток процесса с помощью функции *pthread_create* библиотеки *pthreads*, выполняющий функцию *L_thread*. Функция *L_thread* в бесконечном цикле выполняет шаг процесса *L_step,* пока процесс не придет в состояние *STOP*. Так как потоки процессов выполняются параллельно, то необходимо захватить блокировку, чтобы шаг процесса выполнился атомарно.

```
void L_thread(struct L_state *s) {
 while(true) {
  lock();
  int should_stop = L_step(s);
  if(should_stop) {
   list_del(L_state_list,s);
   free(s);
   unlock();
```

```
      break;
    };
    unlock();
  }
}
```

На каждом шаге процесса происходит выбор одного из действий, выполняемых процессом: посылка сигнала, условный переход или -действие.

Посылка сигнала осуществляется с помощью вызова функции обработки сигнала.

Например, рассмотрим выдержки из функции *L_step* – шаг процесса P_{module}.

```
int L_step(struct S_state *s) {
  switch(nondet_int()) {
    case 0:
      if(s->state == STOP) return 1;
    break;
…
    case 3:
      if(s->state == L2) {
        int res = mstop();
        if(res == 0) {
          //Переход в состояние L5
          s->state = L5;
        }
      }
      break;
…
```



```
    }
    return 0;
  }
}
```

В случае перехода в состояние STOP, при выполнении условия s->state==STOP, функция возвращает 1, что означает завершение процесса.

Для действий посылки сигналов сначала проверяется текущее состояние, в котором осуществляется посылка. Например, для посылки сигнала mstop:

$$L2 := \overline{mstop}.\langle L5\rangle\ldots$$

проверяется нахождение в состоянии L2 (s->state == L2). Если условие выполнено, то осуществляется вызов функции обработки приема сигнала *mstop*. Сигнал считается принятым, только если функция обработки вернула 0, иначе сигнал не принят ни одним из процессов и необходимо перевыполнить посылку. Таким образом, процесс отправитель ожидает пока процесс получатель перейдет в состояние, в котором он готов принять соответствующий сигнал.

В функциях обработки приема сигналов для всех процессов происходит попытка обработать входной сигнал. Для каждого типа процессов перебираются все имеющиеся экземпляры состояний процессов из списка и вызывается соответствующая функция обработки приема. Например, рассмотрим выдержки из функции приема mstop.

```
int mstop() {
  switch(VERIFIER_nondet_int()) {
  …
    case M:
      int k = list_size(M_state_list);
      if(k == 0) {return -1;}
      int i = nondet_int(k-1);
      struct M_state *s = list_get(M_state_list, i);
      int r = M_mstop(s);
      if(r == 0) return 0;
      else return -1;
```

```
  …
}
```

В функции *mstop* недетерминированно выбирается процесс получатель, для каждого из которых недетерминировано выбирается состояние одного из экземпляров в списке *M_state_list*. Далее вызывается функция обработки сигнала *M_mstop* для процесса $P_{trymoduleget}$.

Функции обработки сигнала передается состояние экземпляра процесса и параметры сигнала. Например, для процесса $P_{trymoduleget}$ генерируется следующая функция *M_stop*:

```
int M_mstop(struct M_state *s) {
  int res = -1;
  if(s->state == M && s->param == 0) {
     s->state = MD;
     res = 0;
  }
  if(s->state == MD) {
     s->state = MD;
     res = 0;
  }
  //установить res в 0 если сигнал обработан
  return res;
}
```

При получении сигнала *mstop* в состоянии *M(0)*, соответствующая проверка *s->state == M && s->param == 0*, осуществляется переход в состояние *MD*, в котором последующие сигналы захватов *tmg* возвращают $tmg^{ret}(false)$. В состоянии *MD* сигнал обрабатывается, но состояние не меняется.

Для связывания исходного кода драйвера с π-процессами, генерируется код для процессов-оберток. Рассмотрим процессы P_{fcall}, для процессов P_{init} и P_{exit} генерируется аналогичный код.

$P_{fcall} := !f(p_i, fptr_i, ctx_i, params_i).CALLED(p_i, fptr_i, ctx_i, params_i)$

$CALLED(p_i, fptr_i, ctx_i, params_i) :=$

$\tau.EXECUTING(p_i, fptr_i, ctx_i, params_i, result)$

Процесс *EXECUTING* имеет специальное значение, его выполнение определяется кодом драйвера, вызывается функция драйвера $fptr_i$.

```
int FCALL_step(struct M_state *s) {
  case 1:
    //lock should be held here
    if(s->state == CALLED) {
      s->state = EXECUTING;
      fptr = s->fptr; //fptr – вызываемая фукнция драйвера,
      ctx = s->ctx; //ctx – контекст вызова
      params = s->params; //params – параметры функции
    } else { break; }
    unlock();
    type y = *fptr(ctx, params);  //EXECUTING: вызов
функции fptr
    lock();
    s->state = RET;
    s->RET_result = y;
  break;
  case 2:
    if(s->state == RET) {
      //Вызов функции действия с заданным параметром
      int res = f^ret(s->p,s->RET_result);
      if(res == 0) {
```

```
        //Переход в состояние STOP
        s->state = STOP;
      }}
      break;
    }
    return 0;
  }
```

После того, как функция драйвера выполнилась, процесс переходит в состояние посылки возвращаемого значения *RET*, значение сохраняется в *RET_result*. В состоянии RET происходит посылка сигнала возврата, в случае успеха, процесс завершается.

В процессе выполнения функции драйвер может вызывать библиотечные функции сердцевины ядра, для которых генерируется модель, осуществляющая посылку сигналов соответствующим процессам. Например, для функции usb_register, генерируется следующий код посылки сигнала start(usbpn_driver, usbpn_driver->probe, usbpn_driver->disconnect) процесса P_{usb_driver}.

4.3. Модификация метода для трансляции в многопоточную программу

В последовательном случае при создании процесса, создается лишь экземпляр состояния и увеличивается счетчик активных процессов, создания потока с помощью pthread_create не происходит.

```
void create_L(int p, f_ptr init, f_ptr exit) {
  …
  //pthread_create(L_thread, s);
  thread_cnt++;
}
```

В функции *main* осуществляется недетерминированный выбор одного из типов процессов, добавляется бесконечный цикл *while(true)*.

```
void main() {
```



```
  int p = globalId++;
  create_L(p);//Создать основной поток
  create_M(0);//Создать поток trymoduleget
  while(true) {
    switch(nondet_int()) {
    case L_label:
      lock();
      int stop_loop = L_step_any();
      unlock();
      if(stop_loop) { goto break_loop;}
      break;
    }
    case …
  }
  break_loop:
}
```

Функции вида P_i*_step_any()* выбирают один из экземпляров процесса P_i и запускают для него *L_step*. В коде ожидания возврата из библиотечных функций наряду ожиданием возврата добавляется возможность совершить действие одному из других процессов, добавляется недетерминированный выбор экземпляров процессов аналогично бесконечному циклу в функции *main*.

4.4. Упрощения генерируемого кода для верификации

Для того чтобы успешно верифицировать генерируемый код с помощью современных инструментов высокоточной верификации, необходимо чтобы он не содержал неподдерживаемых конструкций. В первую очередь, современные верификаторы имеют ограничения при работе потенциально

неограниченными структурами данных в куче, такими как списки, деревья, и т.д. Поэтому необходимо минимизировать использование списков при генерации. На это нацелено первое упрощение, заключающееся в замене списков для хранения экземпляров состояний статических процессов на глобальные переменные. Например, для процесса P_{module}:

struct L_state L_state_0;

Для динамически порождаемых процессов ограничивается количество создаваемых экземпляров. Если количество экземпляров превышает максимальное, то новые экземпляры не порождаются.

Функции, работающие с экземплярами состояния процесса, размножаются по количеству копий, каждая копия работает со своей глобальной переменной, например вместо *L_step* генерируется *L_step_0*. В функциях P_i*_step_any* недетерминировано вызываются сгенерированные функции для каждого экземпляра.

Второе упрощение связано с тем, что инструменты верификации имеют ограниченную поддержку указателей на функции, поэтому в сгенерированном коде вызовы функций по указателю заменяются на явные вызовы. Это осуществляется генерацией для каждой функции драйвера своего процесса обертки с ее явным вызовом. Конечно, в местах вызова функций по указателю должно быть заранее известно, какая функция вызывается. В примере выше связывание указателей на функции с их значениями происходит при регистрации группы *usb_driver* и посылке сигнала *start(usbpn_driver, usbpn_driver->probe, usbpn_driver->disconnect)*. В этом месте создается экземпляр процесса P_{usb_driver}, который вместо посылки сигналов вызова функций по указателю *probe, disconnect* осуществляет посылку конкретным экземплярам *usbpn_probe, usbpn_disconnect* процесса P_{fcall}.

4.5. Теорема об эквивалентности

Далее мы покажем, что получающаяся в результате трансляции последовательная Си программа порождает те, и только те трассы взаимодействий, которые были в π-процессах. Так как исходный код драйвера в последовательном случае выполняется в одном потоке, то мы будем требовать, чтобы окружение в виде π-процессов одновременно осуществляло вызов только одной функции драйвера, т.е. после посылки сигнала f и до приема сигнала возврата f^{ret}, недопустимо посылать еще один сигнал f.

Определим понятие трассы выполнения для последовательной Си программы *CP*. Трасса Си программы определяется событиями вызова функции приема сигнала в сгенерированном коде. Данные функции генерируются в пункте 2.3 приложения А метода трансляции. Функции приема сигналов для каждой метки $a[p](y) \in A \cup F$ имеют вид:

int a(int p, type y)

Функции возвращают значение *0*, если сигнал успешно принят, и *-1* иначе. Событие $a[p](y)$ добавляется в трассу, если функция $a(p, y)$ завершается с кодом *0*. Множество всех трасс обозначим как $traces_C(CP)$. Определим $traces_C(CP, D)$, где $D \subseteq F \cup A$, как множество трасс, в которых удалены метки не из множества *D*.

Определим понятие трассовой эквивалентности для π-процесса P и сгенерированной Си программы *CP*.

Определение. π-процесс P трассово-эквивалентен сгенерированной Си программе CP на множестве событий D, обозначим как $P \approx_D CP$, если $traces_\pi(P, D) = traces_C(CP, D)$.

Теорема.

Пусть D – множество событий приема сигналов между драйвером и окружением

$$D = \{init, init^{ret}\} \cup \{exit, exit^{ret}\} \cup \{f, f^{ret}\} \cup \{g_1, g_1^{ret} \dots, g_l, g_l^{ret}\} \cup \{set_{v_i}\} \cup \{get_{v_i}\}.$$

Пусть Sys_π – модель в π исчислении, состоящая из процессов окружения и процессов драйвера, т.е. $Sys_\pi = Env_\pi \mid Drv_\pi$.

Пусть Sys_C – Си код, состоящий из кода, сгенерированного для окружения Env_C (по методу А.2 [8]), и кода драйвера Drv_C.

Пусть $Drv_\pi \approx_D Drv_C$, т.е. код драйвера трассово-эквивалентен π модели.

Тогда $Sys_\pi \approx_D Sys_C$.

Таким образом, для любой трассы π процессов существует эквивалентная трасса в Си программе, и наоборот для любой трассы Си программы существует эквивалентная трасса π-процессов.

Доказательство теоремы приведено в разделе А.3 приложения диссертации [8].

Заключение

Предложенный в работе подход к описанию окружения драйвера позволяет с одной стороны, иметь компактные описания в виде π-процессов, которые позволяют абстрагироваться от многих деталей окружения, допуская разнообразные сценарии взаимодействия. С другой стороны, выразительная мощность π-процессов позволяет описывать необходимые ограничения на взаимодействия окружения с драйвером.

Метод трансляции π-процессов обеспечивает необходимые требования инструментов высокоточного статического анализа, так как в результате

трансляции получается последовательная Си программа, что позволяет осуществлять ее верификацию данными инструментами.

Описанные в работе подходы к моделированию окружения драйверов используются в проекте верификации драйверов ОС Linux [9].

Литература

[1]. N. Palix, G. Thomas, S. Saha, C. Calves, J. Lawall, and Gilles Muller. Faults in Linux: Ten years later. Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '11), USA, 2011.

[2]. D. Beyer, T. Henzinger, R. Jhala, R. Majumdar. The Software Model Checker Blast: Applications to Software Engineering. International Journal on Software Tools for Technology Transfer (STTT), vol. 5, p. 505-525, 2007.

[3]. Shved P., Mandrykin M., Mutilin V. Predicate Analysis with Blast 2.7 //Proceedings of TACAS. 2012. Vol. 7214. P. 525–527.

[4]. D. Beyer, M.E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg, 2011.

[5]. T. Ball, E. Bounimova, R. Kumar, V. Levin. SLAM2: Static Driver Verification with Under 4% False Alarms. FMCAD, 2010.

[6]. J. Corbet, G. Kroah-Hartman, A. McPherson. Linux kernel development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. http://go.linuxfoundation.org/who-writes-linux-2012, 2012.

[7]. Milner R. The Polyadic π-Calculus: a Tutorial. LFCS, Department of Computer Science, University of Edinburgh, 1991. P. 49.

[8]. Мутилин В.С. Верификация драйверов операционной системы Linux при помощи предикатных абстракций. Диссертация на соискание ученой степени к.ф.-м.н., 2012.

[9]. Мандрыкин М. У., Мутилин В. С., Новиков Е. М. и др. Использование драйверов устройств операционной системы Linux для сравнения инструментов статической верификации //Программирование. 2012. Т. 5. С. 54–71.

Environment Modeling of Linux Operating System Device Drivers

Zakharov I.S., Mutilin V.S., Novikov E.M., Khoroshilov A.V.
{ilja.zakharov, mutilin, novikov, khoroshilov}@ispras.ru
ISP RAS, Moscow, Russia

Abstract. To establish verification of Linux operating system device drivers it is necessary to take into account particularity of communication between drivers and a kernel core as far as it plays the main role in drivers behavior. At the same time, verification of a driver together with kernel core source code is not feasible due to complexity and size of the resulting code. That is why we suggest to replace the real environment with its model. Such model of the environment should be correct and complete. Correctness means that it should contain only interaction scenarios which can happen in the real environment. Completeness means that all feasible in the real environment scenarios should present in the corresponding model. Moreover, the environment model should be able to be checked by verification tools. Hence, it should be translated in equivalent representation supported by these tools.

The paper presents a new method for modeling driver environment based on R. Milner calculus and a method of model translation into a program in the C programming language. Being linked with driver source code this program describes the same scenarios of driver behavior as the real driver environment in the operating system.

Keywords: operating system; driver; environment; verification.

References

[1]. Beyer D. Second Competition on Software Verification. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7795, pp. 594-609, 2013. doi: 10.1007/978-3-642-36742-7_43

[2]. Beyer D., Henzinger T., Jhala R., Majumdar R. The Software Model Checker Blast: Applications to Software Engineering. International Journal on Software Tools for Technology Transfer (STTT), vol. 5, pp. 505-525, 2007. doi: 10.1007/s10009-007-0044-z

[3]. Shved P., Mandrykin M., Mutilin V. Predicate Analysis with Blast 2.7. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 525–527, 2012. doi: 10.1007/978-3-642-28756-5_39

[4]. Beyer D., Keremoglu M.E. CPAchecker: A Tool for Configurable Software Verification. In Proc. Computer Aided Verification (CAV), LNCS, vol. 6806, pp. 184–190, 2011. 10.1007/978-3-642-22110-1_16

[5]. Milner R. The Polyadic π-Calculus: a Tutorial. In Proc. Logic and Algebra of Specification, NATO ASI Series, vol. 94, pp 203-246, 1993. doi: 10.1007/978-3-642-58041-3_6

[6]. Mutilin V.S. Verifikatsiya drajverov operatsionnoj sistemy Linux pri pomoshhi predikatnykh abstraktsij [Linux drivers verification with help of predicate abstractions]. Dissertatsiya na soiskanie uchenoj stepeni k.f.-m.n. [PhD thesis], 2012 (in Russian).

[7]. Mandrykin M.U., Mutilin V.S., Novikov E.M., Khoroshilov A.V., Shved P.E. Using linux device drivers for static verification tools benchmarking. Programming and Computer Software, vol. 35, issue 5, pp. 245-256, 2012. doi: 10.1134/S0361768812050039

[8]. Mutilin V.S., Novikov E.M., Strakh A.V., Khoroshilov A.V., Shved P.E. Arkhitektura Linux Driver Verification [Linux Driver Verification Architecture]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 20, pp. 163-187, 2011 (in Russian).

[9]. Khoroshilov A., Mutilin V., Novikov E., Shved P., Strakh A. Towards an Open Framework for C Verification Tools Benchmarking. In Proc. Perspectives of Systems Informatics (PSI), LNCS, vol 7162, pp. 82-91, 2012. doi: 10.1007/978-3-642-29709-0_17