Федеральное государственное бюджетное учреждение науки
Институт системного программирования Российской академии наук

На правах рукописи

Мордань Виталий Олегович

МЕТОДЫ ВЕРИФИКАЦИИ ПРОГРАММ НА ОСНОВЕ КОМПОЗИЦИИ ЗАДАЧ ДОСТИЖИМОСТИ

05.13.11 – математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ
на соискание ученой степени
кандидата физико-математических наук

Научный руководитель:
профессор, д. ф.-м. н.
Петренко Александр Константинович

Москва – 2017

Содержание

Введение....5
Глава 1. Обзор существующих решений....13
1.1. Базовый подход решения задачи статической верификации....13
1.1.1. Постановка задачи статической верификации....14
1.1.2. Внутреннее представление программы....15
1.1.3. Абстрактная модель программы....17
1.1.4. Подход уточнения абстракции по контрпримерам (CEGAR)....18
1.2. Подготовка задачи статической верификации....20
1.2.1. Инструментирование исходного кода....21
1.2.2. Передача модели требования независимо от исходного кода....24
1.3. Системы верификации....25
1.3.1. Система Static Driver Verification....26
1.3.2. Система Linux Driver Verification Tools....26
1.3.3. Метод последовательной верификации....29
1.3.4. Метод пакетной верификации....30
1.4. Методы переиспользования информации в статической верификации....30
1.4.1. Адаптивный статический анализ....31
1.4.2. Условная проверка моделей....32
1.4.3. Регрессионная верификация....35
1.4.4. Перепроверка результатов верификации....39
1.4.5. Комбинирование тестирования и верификации....40
1.4.6. Проверка нескольких требований в смежных областях верификации...41
1.5. Выводы....42
Глава 2. Методы многоаспектной верификации....43
2.1. Подготовка задачи достижимости относительно композиции требований...43
2.1.1. Ограничение множества объединяемых моделей требований....43
2.1.2. Алгоритм объединения моделей требований....45
2.2. Метод обнаружения всех однотипных нарушений....47
2.2.1. Формализация эквивалентности трасс ошибок....50
2.2.2. Автоматическая фильтрация трасс ошибок....51
2.2.3. Полуавтоматическая фильтрация трасс ошибок....53
2.3. Алгоритм многоаспектной верификации....56
2.3.1. Представление утверждений....57
2.3.2. Аппроксимация с одним проверяемым утверждением....58
2.3.3. Остановка проверки утверждения....60
2.3.4. Полнота и корректность алгоритма....62
2.4. Расширения алгоритма многоаспектной верификации....63
2.4.1. Типы верификационных фактов....63

2.4.2. Стратегии корректировки уровня абстракции.....64
2.4.3. Внутренние лимиты.....64
2.4.4. Точки смены утверждений.....65
2.4.5. Расширение используемой аппроксимации.....66
2.4.6. Использование идей алгоритма вне подхода CEGAR.....66
2.5. Метод условной многоаспектной верификации.....67
2.5.1. Внешняя условная многоаспектная верификация.....68
2.5.2. Внутренняя условная многоаспектная верификация.....71
2.6. Метод условной многоаспектной верификации с обнаружением всех однотипных нарушений.....72
2.7. Выводы.....74
Глава 3. Методы декомпозиции автоматной спецификации.....76
3.1. Метод автоматных спецификаций.....76
3.1.1. Наблюдательные автоматы.....77
3.1.2. Описание языка автоматных спецификаций.....78
3.1.3. Сопоставление автоматных спецификаций с инструментированием.….82
3.1.4. Автоматные спецификации в адаптивном статическом анализе.....82
3.2. Метод декомпозиции автоматной спецификации.....84
3.2.1. Общий алгоритм декомпозиции автоматной спецификации.....85
3.2.2. Полнота и корректность метода.....87
3.3. Стратегии разбиения в методе декомпозиции автоматной спецификации.…88
3.3.1. Стратегия Совместная проверка.....89
3.3.2. Стратегия Последовательная проверка 89
3.3.3. Стратегия Совместно-последовательная проверка.....90
3.3.4. Стратегия Релевантность.....91
3.4. Выводы.....93
Глава 4. Реализация предложенных методов.....95
4.1. Расширения системы верификации 95
4.1.1. Новая архитектура системы верификации.....95
4.1.2. Формализация проверяемых требований.....97
4.1.3. Объединение моделей требований.....98
4.1.4. Поддержка метода условной многоаспектной верификации.....99
4.1.5. Поддержка методов, основанных на автоматной спецификации.....100
4.1.6. Поддержка методов обнаружения всех однотипных нарушений.....101
4.2. Расширения статического верификатора.....102
4.2.1. Метод условной многоаспектной верификации.....103
4.2.2. Метод декомпозиции автоматной спецификации.....107
4.2.3. Сравнение реализаций.....107
4.3. Последовательная комбинация предложенных методов.....109
4.4. Выводы.....111

Глава 5. Экспериментальная оценка предложенных методов.....112
5.1. Оценка метода последовательной верификации.....115
5.2. Оценка метода пакетной верификации.....115
5.3. Оценка метода обнаружения всех однотипных нарушений.....116
5.4. Оценка метода условной многоаспектной верификации.....118
5.5. Оценка метода условной многоаспектной верификации с обнаружением всех однотипных нарушений.....121
5.6. Оценка метода автоматных спецификаций.....122
5.7. Оценка метода декомпозиции автоматной спецификации.....124
5.8. Сопоставление методов верификации композиции требований.....125
5.9. Зависимость результата и ускорения от числа требований.....128
5.10. Выводы.....131
Заключение.....133
Список сокращений и условных обозначений.....134
Список используемой литературы.....135
Свидетельства о государственной регистрации программы для ЭВМ.....144
Приложение А Описание проверяемых требований.....145
Приложение Б Рекомендации по выбору параметров использования предложенных методов верификации.....181
Приложение В Доказательство теорем и утверждений.....200

Введение

Актуальность темы

В современном мире программное обеспечение играет значительную роль в жизни общества, поэтому проблема его надежности является крайне актуальной. Хорошо известны примеры [1-6], в которых ошибки в программном обеспечении становились причинами катастроф. Поскольку используемое на практике программное обеспечение постоянно развивается и усложняется, то возрастают и потребности в его автоматической проверке.

Одним из примеров ответственного программного обеспечения с требованиями повышенной надежности является ядро операционной системы Linux [7]. Помимо персональных компьютеров операционная система Linux широко распространена на серверах и суперкомпьютерах, а в последнее время и на мобильных устройствах. На данный момент ядро Linux состоит из более 20 миллионов строк кода на языке программирования C [8]. При этом в процесс его разработки вовлечены тысячи разработчиков, каждые 2-3 месяца выходит новая версия, содержащая тысячи изменений. В среднем каждый час для модулей ядра Linux выходит 7-8 изменений, нацеленных как на исправление ошибок, так и на расширение функциональности, каждое из которых потенциально может добавить и новую ошибку. В то же самое время каждая ошибка в ядре Linux является критической и может привести к отказу всей операционной системы. Согласно исследованиям [9], большинство подобных ошибок находится в модулях операционной системы, что делает проверку модулей ядра Linux достаточно актуальной задачей.

На практике для автоматической проверки программного обеспечения, как правило, применяется тестирование [10] или другие методы поиска ошибок (например, статический анализ кода программ [11]). Однако применение данных

подходов не гарантирует отсутствия ошибок, следствием чего является множество примеров, в которых критические ошибки не были выявлены [1-6].

Статическая верификация программного обеспечения является средством проверки исходного кода без его выполнения, при этом рассматриваются все возможные пути выполнения программы. Главное достоинство статической верификации заключается в том, что она нацелена на доказательство корректности программного обеспечения, а не только на поиск часто встречающихся ошибок. Основной недостаток, который затрудняет ее применение на практике, – это необходимость большого количества вычислительных ресурсов (таких, как процессорное время и оперативная память) в особенности для больших программных систем. Помимо этого, задача статической верификации в общем случае не является разрешимой, поэтому простое увеличение вычислительных ресурсов не всегда помогает решить задачу.

Наиболее значимые научные результаты в области статической верификации достигнуты исследователями, развивающими технологии software model checking (наиболее известные проекты BLAST [27, 28], CPAchecker [29, 30], CBMC [54], SLAM [19] и др.). В настоящее время инструменты статической верификации (статические верификаторы), основанные на подходе уточнения абстракции по контрпримерам (от англ. counterexample guided abstraction refinement, или CEGAR [17, 18]), являются масштабируемыми и, в частности, могут использоваться для верификации драйверов операционных систем, что демонстрируется в ежегодно проводимых международных мероприятиях по верификации SV-COMP (competitions on software verification) [12-16].

Каждая проверяемая программа должна удовлетворять большому числу требований, начиная от правил используемого языка программирования и заканчивая специфичными функциональными и нефункциональными требованиями к программе. Примером нарушения специфичного требования

является некорректное использование интерфейсов сердцевины ядра в модулях ядра операционной системы, что может привести к различным негативным последствиям (например, к утечкам памяти или взаимным блокировкам). Для ядра Linux проведенное исследование [21] показало, что количество ошибок на специфичные требования в модулях составляет более половины от всех ошибок, являющихся нарушениями требований. Исследования, проведенные Microsoft, показывают, что число подобных специфичных требований к драйверам операционной системы измеряется сотнями [22]. Если для проверки общих правил, характерных для языков программирования, существует большое количество инструментов, кроме того, соответствующие проверки зачастую встраиваются в компиляторы, то проверка специфичных требований, как правило, ограничивается тестированием. Статическая верификация является одним из наиболее перспективных средств для проверки именно специфичных требований.

Однако статические верификаторы не предназначены для непосредственной проверки требований в программе, которые могут быть заданы неформально (например, в виде текстового описания), а способны решать задачу достижимости – доказывать недостижимость некоторой точки программы из точки входа в программу. Поэтому для проверки требования в программе с помощью статической верификации обычно программа модифицируется путем добавления вспомогательных проверок, нарушение которых помечается специальной меткой и соответствует нарушению требования. Подготовленный таким образом код программы (задача достижимости) подается на вход статическому верификатору, который либо доказывает корректность программы относительно проверяемого требования, либо находит нарушение проверяемого требования и предоставляет одно из его возможных проявлений, либо не справляется с решением задачи.

На практике, как правило, требуется проверять программу относительно многих требований. Для этого обычно используется последовательная

верификация, т. е. для каждого требования создается и решается отдельная задача достижимости. Однако данный метод является неэффективным. Во-первых, необходимые на верификацию ресурсы в среднем возрастают пропорционально количеству проверяемых требований. Во-вторых, статические верификаторы останавливаются после нахождения уже первого нарушения требования, что ведет к увеличению числа запусков, времени и ресурсов верификации для выявления всех нарушений требования. При этом в обоих случаях никак не учитывается тот факт, что многократно проверяется одна и та же программа, в результате чего выполняется множество однотипных действий и полученные промежуточные результаты верификации (например, построенная абстракция программы) забываются, что приводит к нерациональному использованию вычислительных ресурсов.

Альтернативным способом является объединение всех или части требований, создание одной задачи достижимости и верификация композиции требований. В данном случае программа проверяется однократно, а полученные промежуточные результаты верификации не забываются. Однако появляются новые проблемы. Во-первых, если задача не может быть успешно решена хотя бы для одного требования, то она не будет решена для всех требований. Во-вторых, нахождение нарушения для одного требования приведет к остановке верификации и потере результата для остальных требований. Таким образом, верификация композиции требований «в лоб» ведет к ухудшению результата и, в частности, к пропуску нарушений требований относительно последовательной верификации.

Для примера рассмотрим конфигурируемую систему статической верификации Linux Driver Verification Tools (LDV Tools) [23-26]. Система LDV Tools нацелена на проверку требований, описывающих корректное использование программных интерфейсов сердцевины ядра, в модулях ядра Linux с помощью различных статических верификаторов. Процесс верификации всех модулей ядра

Linux последних версий с помощью системы LDV Tools относительно одного требования с использованием статических верификаторов BLAST [27, 28] или CPAchecker [29, 30] на современном компьютере требует более 3 дней процессорного времени, следовательно, проверка сотни требований займет годы процессорного времени, в то время как каждый час производится в среднем 7-8 изменений в ядре Linux [8].

Таким образом, можно утверждать, что задача верификации композиции требований и разработка методов ее эффективного решения является актуальной темой исследований и разработок.

Цель и задачи работы

Цель работы – разработка методов статической верификации программного обеспечения для проверки соответствия программ композиции требований.

Для достижения данной цели были поставлены следующие задачи:

- Провести анализ существующих методов статической верификации для определения того, насколько они подходят для решения поставленной цели.
- Разработать новые методы верификации программного обеспечения, предназначенные для проверки композиции требований с учетом того, что каждое требование в программе может нарушаться более одного раза.
- Реализовать предложенные методы.
- Дать оценку области применимости предложенных методов и составить рекомендации по их использованию.

Научная новизна работы

Научной новизной обладают следующие результаты работы:

- Метод статической верификации программного обеспечения для обнаружения всех однотипных нарушений проверяемого требования.
- Метод статической верификации программного обеспечения для проверки

выполнения композиции требований (условная многоаспектная верификация).

- Метод статической верификации программного обеспечения, расширяющий возможности представления требований в виде их автоматных спецификаций.
- Метод статической верификации программного обеспечения на основе декомпозиции автоматной спецификации требований на группы требований для совместной верификации внутри группы.
- Сформулированы и доказаны утверждения и теоремы, являющиеся обоснованием корректности предложенных методов.

Теоретическая и практическая значимость

В данной работе были предложены методы статической верификации программного обеспечения, нацеленные на проверку выполнения композиции требований с возможностью нахождения нескольких нарушений требований. Для предложенных методов были сформулированы и доказаны утверждения и теоремы, обосновывающие их корректность. Эти результаты могут использоваться в исследовательских проектах и обучении в курсах формальных методов разработки и анализа программ.

Предложенные методы были реализованы в качестве расширения системы верификации Linux Driver Verification Tools [26] и статического верификатора CPAchecker [30]. Проведенные эксперименты демонстрируют повышение производительности верификации в 4-5 раз.

Результаты данной работы в первую очередь полезны для разработчиков статических верификаторов. Методы проверки выполнения композиции требований позволяют существенно повысить производительность всего процесса верификации при проверке многих требований. Методы обнаружения всех однотипных нарушений требований позволяют выявлять больше ошибок в

программном обеспечении с помощью однократного выполнения статической верификации.

Положения, выносимые на защиту

- Методы статической верификации программного обеспечения, основанные на инструментировании исходного кода и предназначенные для обнаружения всех однотипных нарушений (ОВН) и проверки выполнения композиции требований с помощью условной многоаспектной верификации (УМАВ).
- Методы статической верификации программного обеспечения, с использованием формализации требований в виде автоматных спецификаций (АС) и декомпозиции автоматной спецификации на группы требований для совместной верификации (ДАС).
- Теорема о полноте и корректности предложенных методов для требований, удовлетворяющих ограничениям инструментирования исходного кода программы.

Публикации и личный вклад автора

По теме диссертации автором опубликовано 5 работ [31-35] (работы [33-35] опубликованы в изданиях из перечня ВАК, они же индексированы в Web of Science и Scopus). В работе [32] автором предложены методы фильтрации трасс ошибок, являющиеся основой для метода ОВН. В работе [33] представлен метод УМАВ и описана его апробация на практике. Возможность совместного использования методов УМАВ и ОВН обоснована автором в работе [34]. В работе [35] автором была предложена основная идея методов АС и ДАС, а также стратегия разбиения спецификации для метода ДАС, оказавшаяся наиболее эффективной в проведенных экспериментах.

В ходе выполнения работы было получено 2 свидетельства о государственной регистрации программы для ЭВМ [1-2, см. с. 144].

Апробация результатов работы

Основные положения работы докладывались на следующих конференциях и семинарах:

- международный молодежный научный форум «ЛОМОНОСОВ-2014»;
- семинар Института системного программирования РАН (г. Москва, 2014 г.);
- 8-й весенний коллоквиум молодых исследователей в области программной инженерии (SYRCoSE: Spring Young Researchers Colloquium on Software Engineering, г. Санкт-Петербург, 2014 г.);
- 9-й весенний коллоквиум молодых исследователей в области программной инженерии (SYRCoSE: Spring Young Researchers Colloquium on Software Engineering, г. Самара, 2015 г.);
- 10-я международная Ершовская конференция по информатике PSI-2015 (Казань, 2015 г.);
- 5-й международный семинар Linux Driver Verification (г. Москва, 2015 г.);
- 1-й международный семинар, посвященный инструменту CPAchecker (г. Пассау, Германия, 2016 г.).

Структура и объем диссертации

Работа состоит из введения, пяти глав, заключения и списка литературы (77 наименований) и трех приложений. Основной текст диссертации (без приложений и списка литературы) занимает 133 страницы, общий объем – 205 страниц.

Глава 1. Обзор существующих решений

В данном обзоре рассмотрены базовые подходы подготовки и решения задач статической верификации с точки зрения возможности их использования для проверки композиции требований. Кроме того, рассмотрены методы переиспользования информации в статической верификации, нацеленные на повышение производительности верификации.

1.1. Базовый подход решения задачи статической верификации

Статическая верификация программного обеспечения – это автоматический подход к доказательству того, что код программы удовлетворяет своей спецификации. Под спецификацией в данной работе будет пониматься набор требований, каждое из которых сводится к проблеме достижимости (т. е. тем самым может быть проверено с помощью статической верификации). Подобные требования описывают множество ситуаций, которые не должны достигаться во время выполнения программы (например, запрещается повторно освобождать один и тот же ресурс), т. е. нарушение требования является потенциальной ошибкой в коде программы.

Наиболее распространены на практике 2 подхода статической верификации [37] – уточнение абстракции по контрпримерам (от англ. counterexample guided abstraction refinement), или CEGAR [17, 18], и ограничиваемая проверка моделей (от англ. bounded model checking), или BMC [36]. Подход CEGAR основан на итеративном построении модели программы и используется в верификации драйверов операционных систем [25, 37, 44]. Подход BMC анализирует все циклы и рекурсивные вызовы в программе только на заданное число итераций и применяется в верификации устройств [38].

Для определенности в качестве базового подхода статической верификации

будет рассмотрен подход CEGAR, при этом большинство идей верны и для любых подходов статической верификации.

1.1.1. Постановка задачи статической верификации

Пусть для верификации дана программа *P* на языке программирования C[1], в которой каждый оператор помечен уникальной меткой. Часть меток считается **метками ошибок**, которые соответствуют нарушению заданных требований к программе. Для упрощения можно считать, что метки ошибки расставил автор программы (способы автоматического связывания требований с кодом программы будут рассмотрены в п. 1.2). Таким образом, задача статической верификации непосредственно сводится к задаче достижимости – требуется доказать недостижимость меток ошибок из точки входа в программу (например, функции *main*).

```
void __error(void);
void strncpy(char *dst, const char *src, int n) {
L1:      if (src == 0)
ERROR_1:   __error();
L2:      if (dst == 0)
ERROR_2:   __error();
L3:      if (n < 0)
ERROR_3:   __error();
L4:      //...
}
```

Рис. 1.1. Пример простейшей программы.

Например, пусть дана функция *strncpy*[2], выполняющая копирование n символов из строки *src* в строку *dst*, и требуется доказать выполнение следующих требований к параметрам функции – указатели *dst* и *src* не должны быть нулевыми, а число *n* должно быть положительным. Обозначив нарушение требований метками ошибок *ERROR_1*, *ERROR_2* и *ERROR_3* соответственно,

1 Язык программирования C рассматривается для определенности, при необходимости может быть заменен.
2 Стандартная функция языка C: http://linux.die.net/man/3/strncpy.

получим код функции *strncpy*, который представлен на рисунке 1.1.

1.1.2. Внутреннее представление программы

Перед началом верификации необходимо произвести преобразование программы во внутреннее представление, с которым будет работать алгоритм статической верификации. В подходе CEGAR для этого используется **граф потока управления** [18], или **ГПУ**, (от англ. control flow graph) – это ориентированный граф, в вершинах которого находятся уникальные метки программы, а ребра помечены одним или несколькими операторами программы (в зависимости от представления последовательности инструкций). Например, при кодировании малыми блоками [39] (от англ. single block encoding) каждому оператору программы соответствует ровно одно ребро, а при крупноблочном кодировании [39] (от англ. large block encoding) для каждого линейного участка программы операторы склеиваются и в ГПУ им соответствует только одно ребро. При построении ГПУ для оператора ветвления создается две дуги – одна с выполнением соответствующего условия, другая с его отрицанием, а для всех остальных операторов – только одна дуга. Начальная вершина ГПУ соответствует точке входа в программу.

Для рассмотренного выше примера программы ГПУ представлен на рисунке 1.2. Дуги, ведущие в состояние `"__error"` (т. е. к метке ошибки), соответствуют нарушению требования.

Назовем **состоянием программы** метку следующего оператора программы и означивание всех переменных программы [18], т. е. отображение имен всех переменных на соответствующие им конкретные значения. Начальное состояние программы содержит начальную вершину ГПУ и исходные значения переменных (которые, возможно, не определены). Например, для рассмотренного примера одним из возможных начальных состояний является следующее:

```
L1: {src == 0, dst == "str", n == 3},
```

которое, как нетрудно видеть, ведет к нарушению требования к программе.

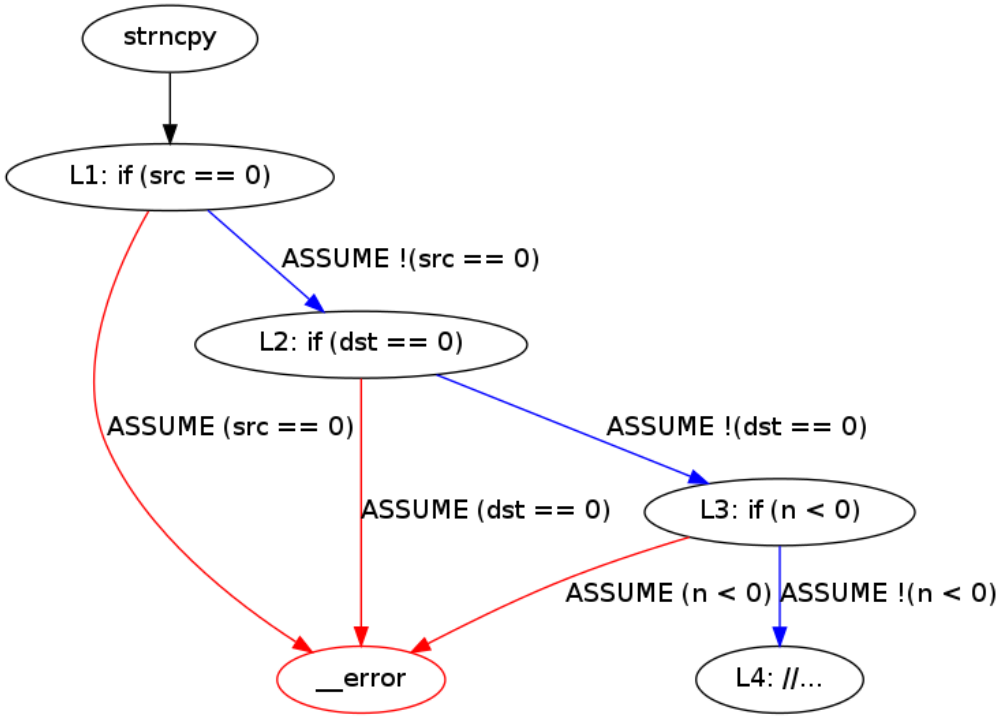

Рис. 1.2. Граф потока управления для простейшей программы.

Введем отношение **непосредственной достижимости** между двумя состояниями программы. Состояние *P* считается непосредственно достижимым из состояния *Q*, если есть дуга в ГПУ из *P* в *Q*, при этом переменные состояний удовлетворяют условию перехода или в случае оператора присваивания меняются согласно данному оператору [18]. Например, состояния

```
L1: {src == 0, dst == "str", n == 3}
```

и

```
ERROR_1: {src == 0, dst == "str", n == 3}
```

являются непосредственно достижимыми.

Теперь можно определить понятие **пути выполнения** программы – это последовательность состояний программы, в которой для каждой пары соседних состояний выполнено отношение непосредственной достижимости [18]. Граф

достижимости представляет собой ориентированный граф, в вершинах которого находятся состояния программы, а дуги соединяют два непосредственно достижимых состояния. Таким образом, граф достижимости содержит все возможные пути выполнения программы, т. е. представляет собой точную модель программы, и задача верификации может быть переформулирована следующим образом – требуется доказать, что в графе достижимости нет состояний с меткой ошибки. Однако уже для одной 32-битной переменной без начального значения нужно перебрать 2^{32} только начальных состояний, а для рассматриваемой простейшей программы граф достижимости состоял бы из $3*2^{32}$ (почти 13 миллиардов) состояний. Таким образом, решение задачи достижимости «в лоб» ведет к проблеме экспоненциального роста числа состояний, что делает невозможным верификацию точной модели программы на практике за разумное время.

1.1.3. Абстрактная модель программы

Для решения описанной выше проблемы экспоненциального роста в подходе CEGAR используется абстрактная модель программы. В отличие от точной модели, абстрактная строится на основе абстрактных состояний, которые состоят из множества состояний программы [18].

Абстрактный граф достижимости, или **АГД**, (от англ. abstract reachability graph) – граф, представляющий собой абстракцию программы, в вершинах которого находятся абстрактные состояния программы, а ребра соответствуют переходам в ГПУ [18]. Ключевым при построении АГД является выбор абстрактного домена, который определяет, на основании чего состояния программы будут объединяться в абстрактные состояния. Примерами абстрактных доменов являются предикатная абстракция [40, 44] (от англ. predicate abstraction), анализ явных значений [41] (от англ. explicit value analysis), анализ рекурсивных

структур данных [38] (от англ. shape analysis) и др. В случае предикатной абстракции абстрактные состояния задаются с помощью предикатов (т. е. логических конструкций над переменными программы), для вычисления переходов между состояниями используются SMT-решатели [42], задача которых состоит в том, чтобы для заданной логической формулы определить, существуют ли значения переменных, при которых она обращается в истину. Анализ явных значений для каждой отслеживаемой переменной хранит ее точное значение. Анализ рекурсивных структур данных нацелен на моделирование рекурсивных структур данных (например, списков, деревьев).

При построении АГД необходимо правильно выбрать уровень абстракции, т. е. определить, насколько точно она будет моделировать точную модель программы. С одной стороны, слишком грубая абстракция может привести к невозможности доказательства выполнения требований к программе, с другой стороны, слишком точная абстракция может привести к экспоненциальному росту числа абстрактных состояний, что сделает невозможным решение задачи. Уровень абстракции задается с помощью **точности абстракции** (от англ. precision), которая определяет, какая информация должна использоваться для построения абстракции, а какая должна пропускаться [43]. Например, в предикатной абстракции [40] точность определяет отслеживаемые предикаты, в анализе явных значений [41] точность определяет отслеживаемые переменные.

1.1.4. Подход уточнения абстракции по контрпримерам (CEGAR)

Основная идея подхода уточнения абстракции по контрпримерам, или CEGAR, заключается в построении абстрактной модели программы на основе заданного абстрактного домена, ее итеративном уточнении и доказательстве недостижимости меток ошибок в абстрактной модели (рисунок 1.3) [17, 18].

В начале подход CEGAR строит в выбранном абстрактном домене АГД на

основе начальной точности (например, пустой). Если указанная метка ошибки не была достигнута, то алгоритм завершается с вердиктом *Safe*, т. е. успешно была доказана корректность программы относительно проверяемого требования. В противном случае CEGAR строит контрпример, который представляет собой путь в ГПУ из точки входа до метки ошибки, и проверяет его на выполнимость с помощью решателя. Если он выполним, то на его основе строится трасса ошибки, представляющая собой последовательность операций в исходной программе от точки входа до метки ошибки, и алгоритм завершается с вердиктом *Unsafe* – найдено нарушение проверяемого требования в программе. В противном случае происходит уточнение абстракции на основе ложного контрпримера, добавляется новая точность, и цикл CEGAR продолжается. Одним из основных методов для получения новой точности является интерполяция Крейга [45].

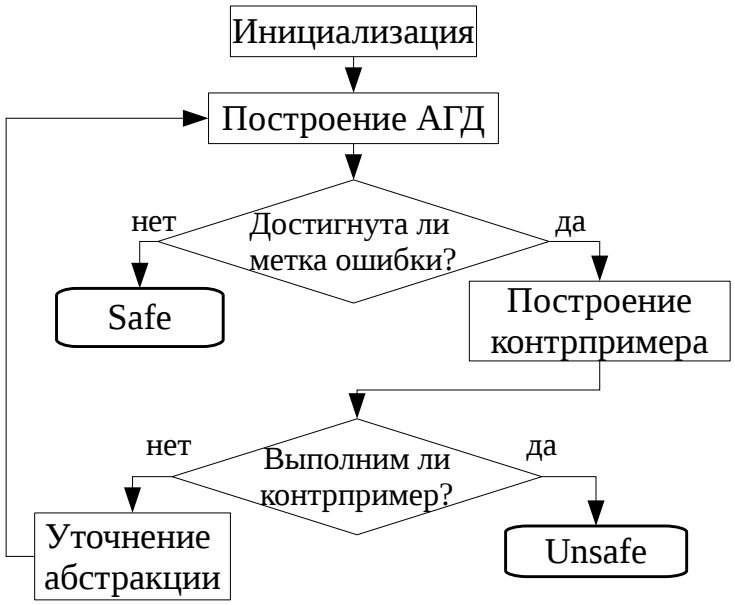

Рис. 1.3. Цикл CEGAR.

Поскольку к данной задаче сводится проблема останова [46], то статическая верификация является неразрешимой задачей в общем случае и подход CEGAR может «зациклиться» и не дать ответ на вопрос, нарушается ли требование в программе. В данном случае происходит экспоненциальный рост числа

абстрактных состояний в АГД и за разумное время решить задачу становится невозможно. Поэтому на практике статические верификаторы, реализующие подход CEGAR, работают с ограниченными ресурсами (такими, как процессорное время и память), и если задача не была решена за отведенное время, то статический верификатор завершает работу с вердиктом *Unknown*.

Таким образом, подход CEGAR либо успешно доказывает выполнение проверяемого требования в программе, либо находит одно из возможных нарушений требования и останавливает верификацию, поскольку считается, что программа уже нарушает проверяемое требование и нет необходимости пытаться доказать ее корректность, либо зацикливается.

Заметим, что подход CEGAR может одновременно доказывать недостижимость нескольких меток ошибок, каждая из которых может соответствовать нарушениям разных требований. Однако при этом нахождение нарушения одного из требований приведет к остановке верификации и потере результата для остальных требований, кроме того, чем больше будет проверок, тем больше будет вероятность экспоненциального роста числа состояний в АГД.

1.2. Подготовка задачи статической верификации

Подход CEGAR решает задачи достижимости, в которых определенные точки помечены метками ошибки, при этом на практике в проверяемых программах, как правило, нет специально расставленных меток ошибок. Поэтому для использования подхода CEGAR на практике необходимо автоматическое преобразование программы в задачу достижимости.

Для начала требование необходимо формализовать на некотором языке описания требований. Формализованное требование назовем **моделью** требования. В общем случае каждому требованию может соответствовать несколько моделей, формализующих требование на разных языках.

Глобально существует 2 подхода к созданию задач достижимости – инструментирование исходного кода [47], т. е. модификация кода программы путем добавления в него модели требования, и передача модели верификатору независимо от исходного кода.

1.2.1. Инструментирование исходного кода

Рассмотрим схему инструментирования исходного кода на примере метода контрактных спецификаций, описанного в диссертации [47]. В данном методе для начала определяются так называемые **точки использования**, которые соответствуют релевантным для проверяемого требования элементам программы (как правило, вызовам функций или макрофункциям). Далее для каждой точки использования пишется модель на языке программирования C. В подобных моделях могут выполняться вспомогательные действия (например, изменение внутреннего состояния формализованного требования) и проверяться различные предусловия, записанные в виде **утверждений** (вызов вспомогательной функции *assert*). Заметим, что вспомогательные действия могут содержать сколь угодно сложные конструкции языка программирования C. Нарушение утверждений при этом помечается специальной меткой ошибки, достижимость которой и будет соответствовать нарушению проверяемого требования.

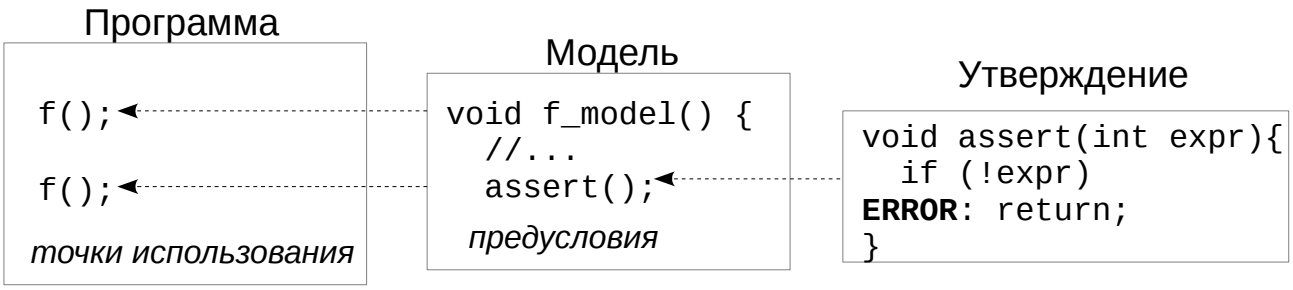

Рис. 1.4. Схема создания задачи достижимости методом инструментирования.

Процесс инструментирования (рис. 1.4) заключается в добавлении моделей в соответствующие им точки использования в коде программы. При этом либо исходный вызов функции полностью заменяется на соответствующий ему вызов

модельной функции (при необходимости моделирующий и его возвращаемое значение), либо вызов модельной функции помещается до исходного (например, исключительно для проверки предусловия). После проведения указанных преобразований получается задача достижимости, которую способен решать подход CEGAR.

Данный метод имеет ряд преимуществ. Во-первых, в описании модели требования используются произвольные конструкции языка программирования C, что обеспечивает широкие возможности по формализации требований. Например, можно точно моделировать возвращаемые значения библиотечных функций, отсекать невозможные пути выполнения программы, работать с указателями. Во-вторых, списки точек использования могут быть параметризованы или сгенерированы по шаблону, что позволяет создавать модельные функции динамически (например, для каждого уникального ресурса – отдельная модельная функция с уникальными проверками) и поддерживать изменения программных интерфейсов (например, добавление нового параметра функции или изменение ее имени). В-третьих, подготовленные таким образом задачи достижимости могут быть решены с помощью любого подхода статической верификации, при этом вся необходимая информация о проверке требования уже находится в исходном коде.

Недостатком данного метода является изменение исходного кода программы и, следовательно, его усложнение за счет добавления модельных функций. При добавлении большого количества модельных функций размер проверяемого кода может быть увеличен на десятки тысяч строк, кроме того, вспомогательные проверки усложняют структуру ГПУ за счет новых операторов ветвления. Поэтому инструментирование предполагает создание задачи достижимости для проверки единственного требования, что позволяет минимизировать усложнение кода. Стоит отметить, что после проведения инструментирования исходного кода данную процедуру в общем случае нельзя повторить для добавления модели другого

требования, поскольку в общем случае точки использования после инструментирования могут исчезнуть (например, вызов исходной функции был полностью заменен на вызов модельной функции). Поэтому если два требования имеют пересечение точек использования, то последовательно применять метод инструментирования для создания одной задачи достижимости заведомо нельзя.

Рассмотрим пример задания модели требования для функции *strncpy* методом инструментирования. Заметим, что данная функция является стандартной библиотечной функцией языка С, требование же формулируется для программ, в которых она может быть использована. Поэтому для верификации описанных выше требований к параметрам этой функции необходимо добавить соответствующие проверки в качестве предусловий перед каждым вызовом функции в программе. На аспектно-ориентированном расширении языка С [47] соответствующая точка использования описывается следующим образом:

```
before: call(void strncpy(char *dst,
                          const char *src,
                          int n)) {
    model_function_check_strncpy(dst, src, n);
}
```

После проведения процедуры инструментирования перед каждым вызовом функции *strncpy* будет добавлен вызов функции *model_function_check_strncpy*:

```
void model_function_check_strncpy(char *dst,
                                  const char *src,
                                  int n) {
    assert(dst != 0);
    assert(src != 0);
    assert(n > 0);
}
```

При этом каждое утверждение (функция *assert(expr)*) содержит метку ошибки, которая достигается при нарушении заданного условия (рис. 1.4).

Аналогичный метод используется и в системе верификации драйверов

операционной системы Windows [20], в которой требования формализованы с помощью языка SLIC [48], являющегося упрощением языка программирования C.

1.2.2. Передача модели требования независимо от исходного кода

Для решения описанной выше проблемы инструментирования, которая заключается в усложнении исходного кода, вполне логично передавать модель требования верификатору независимо от исходного кода. В данном случае верификатору необходимо передать немодифицированный исходный код программы и модель требования в том формате, который поддерживается данным верификатором. При этом добавление вспомогательных проверок и расстановка соответствующих меток ошибок производится непосредственно во время верификации.

Несмотря на то что данный подход решает более простые задачи достижимости и потенциально более эффективен, у него существуют и недостатки. Во-первых, для описания моделей требований внутри верификатора необходим специальный язык, возможности которого, скорее всего, будут существенно уступать языку программирования C. Во-вторых, в данном случае для каждого верификатора необходимо создавать свою модель одного и того же требования, чтобы поддерживать используемый в нем язык описания требований.

Примером подобных методов является язык запросов BLAST [49], который используется в инструменте статической верификации BLAST [27, 28]. В данном языке используются наблюдательные автоматы (от англ. observer automata), которые описывают все множество корректных выполнений программы с помощью синтаксических шаблонов. С их помощью возможно проверять параметры функций и простейшие последовательности их вызовов. Более сложные проверки, например, требующие моделирования внутреннего состояния, не могут быть сведены к наблюдательным автоматам.

Например, автомат, устанавливающий проверки на параметры функции *strncpy*, на языке запросов BLAST выглядит следующим образом:

```
EVENT {
    PATTERN { strncpy($1, $2, $3); }
    ASSERT { $1 != 0; }
    ASSERT { $2 != 0; }
    ASSERT { $3 > 0; }
}
```

Проведенное исследование [49] показало широкие возможности подобной формализации требований. На выбранном наборе драйверов операционной системы Linux с помощью передачи модели требований отдельно от исходного кода с помощью языка запросов BLAST удалось достичь ускорения от 4 до 12 раз по сравнению с проверкой тех же требований с инструментированием исходного кода.

Таким образом, несмотря на то что инструментирование остается общепринятой практикой при формализации требований в статической верификации во многом из-за широких возможностей по заданию самих требований (либо на языке самой программы, либо на близком к языку программы) и поддержки любыми статическими верификаторами, передача формализованных требований независимо от исходного кода является достаточно перспективным направлением.

1.3. Системы верификации

Для поддержания всего процесса верификации используются системы верификации. В общем случае задача системы верификации состоит в том, чтобы подготовить объект верификации (т. е. подмножество исходного кода целевой программы вместе с опциями препроцессирования и параметрами сборки) для заданной программной системы, поставить задачу достижимости для проверки формализованного требования, решить ее и предоставить результат для анализа

человеку. Рассмотрим существующие системы верификации и способы проверки нескольких требований в них.

1.3.1. Система Static Driver Verification

Наиболее известной и успешно используемой на практике системой верификации является Static Driver Verification, или SDV [20], от Microsoft, предназначенная для верификации драйверов операционной системы Windows. SDV использует тот факт, что интерфейс ядра операционной системы Windows стабилен, т. е. требования к драйверам не меняются. Поскольку разработчики драйверов Windows обязаны аннотировать код для задания модели окружения, то существенно упрощается процесс подготовки объектов верификации. Для проверки имеется несколько сотен формализованных требований [22], которые заданы с помощью языка описания требований SLIC, добавление новых ввиду стабильности интерфейса ядра не предусмотрено. Для подготовки задач достижимости используется инструментирование исходного кода. Задачи достижимости решаются с помощью статических верификаторов SLAM [19] и Yogi [51], в которых реализован подход CEGAR и предикатная абстракция. Для анализа результатов верификации имеется пользовательский интерфейс.

Однако код системы SDV является закрытым, а сама система не применима для верификации программ, отличных от драйверов операционной системы Windows.

1.3.2. Система Linux Driver Verification Tools

Система с открытым кодом Linux Driver Verification Tools, или LDV Tools [23-26], созданная и поддерживаемая в ИСП РАН, предназначена для верификации модулей ядра операционной системы Linux относительно требований, описывающих корректное использование программных интерфейсов сердцевины ядра, с помощью различных статических верификаторов. Помимо

этого, LDV Tools позволяет сравнивать различные статические верификаторы на существующих наборах подготовленных задач достижимости. На момент написания данной работы в системе было формализовано 30 требований с помощью аспектно-ориентированного расширения языка C [47]. Система LDV Tools помогла найти более 240 ошибок [52], которые были приняты и исправлены разработчиками ядра операционной системы Linux.

Входными параметрами системы LDV Tools являются ядро операционной системы Linux, список модулей для верификации, список идентификаторов требований и дополнительные параметры (например, ограничения на ресурсы). На выходе пользователю предоставляется результат, в котором для каждого модуля будет выдан вердикт. Для каждого заданного требования и модуля ядра система LDV Tools последовательно выполняет верификацию (рис. 1.5).

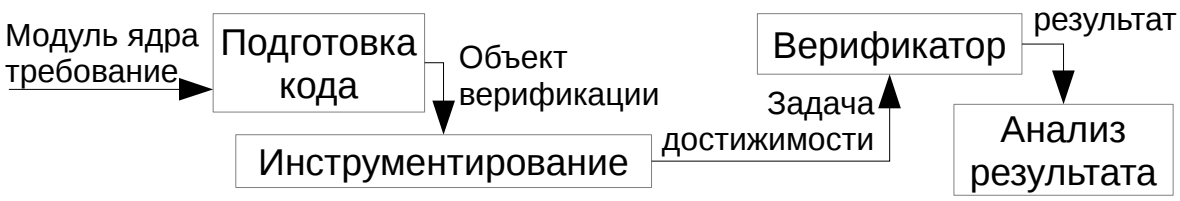

Рис. 1.5. Архитектура системы LDV Tools.

На первом шаге происходит создание так называемого объекта верификации для проверяемого модуля. Для начала по командам сборки ядра выделяются все файлы, которые составляют данный модуль. Поскольку в модулях ядра нет заданной точки входа, а для постановки задачи достижимости она необходима, то далее происходит генерация искусственной точки входа, из которой вызываются обработчики модуля [53]. Помимо этого, при генерации точки входа может учитываться специфика проверяемого требования. На выходе объект верификации представляет из себя подмножество кода ядра Linux с командами их сборки. Данный шаг полностью зависит от специфики ядра Linux и не может быть непосредственно применен для верификации других программных систем.

На втором шаге производится связывание исходного кода с проверяемым требованием с помощью метода инструментирования. В системе LDV Tools каждому требованию соответствует текстовое описание и формальная модель на аспектно-ориентированном расширении языка C [47]. На выходе получается задача достижимости, которая может быть решена с помощью произвольного статического верификатора. Данный шаг уже не зависит от специфики ядра Linux и может быть применен для верификации произвольных программ на языке C.

На третьем шаге заданный статический верификатор решает поставленную задачу достижимости. В системе LDV Tools использовались статические верификаторы, реализующие как подход CEGAR (CPAchecker [29], BLAST [27], UFO [55]), так и подход BMC (CBMC [54]). По умолчанию используется статический верификатор CPAchecker [29]. В общем случае может быть добавлена поддержка для любого верификатора, решающего задачи достижимости. После завершения работы статического верификатора результат его работы в системе LDV Tools предоставляется для анализа человеку.

Для анализа результатов верификации в системе LDV Tools существует web-интерфейс [25], в котором представлена общая статистика (вердикт статического верификатора, затраченные ресурсы и т. д.), при этом для вердиктов *Unsafe* предоставляется трасса ошибки со ссылками на исходный код [56], а для вердиктов *Unknown* предоставляется его причина (например, нарушение ограничения по времени). Данный этап производится человеком и не имеет непосредственного отношения к самому процессу верификации, однако необходим на практике. Главным образом анализ результатов нацелен на вердикты *Unsafe*, т. е. потенциальные ошибки в программе. Для каждой найденной трассы ошибки необходимо установить, соответствует ли она реальной ошибке либо является следствием ложного сообщения об ошибке [25]. В случае реальной ошибки по найденной трассе в ходе анализа обычно готовится ее исправление. В противном

случае необходимо установить причину ложного сообщения об ошибке, которая может находиться на любом шаге процесса верификации: статические верификаторы полностью поддерживают не все конструкции языка программирования C, формализованное требование может не быть адекватным в частных случаях, сам код может быть подготовлен неверно (например, из-за некорректно подготовленной последовательности вызовов функций обработчиков проверяемого модуля [53]). В общем случае из-за подобных причин возможен и пропуск ошибок (т. е. была доказана корректность программы относительно требования, хотя на самом деле оно нарушается в программе), для выявления которого обычно необходимо выяснить, является ли подготовленный код задачи достижимости адекватным проверяемому модулю ядра.

1.3.3. Метод последовательной верификации

В рассмотренных системах верификации существует множество требований, которым должны удовлетворять проверяемые программы. Базовым методом проверки программы относительно нескольких требований является последовательная верификация всех требований, т. е. для проверки каждого требования необходимо подготовить и решить отдельную задачу достижимости. При этом необходимые для верификации ресурсы в среднем возрастают пропорционально числу требований (как за счет подготовки большего числа задач достижимости, так и за счет их решения). Никак не используется тот факт, что проверяется одна и та же программа, тем самым теряются полученные ранее знания о верификации программы и неэффективно расходуются ресурсы. Однако независимая проверка требований позволяет получить наиболее точный результат.

Таким образом, метод последовательной верификации является простым и надежным способом решения поставленной задачи, но при этом неэффективным.

1.3.4. Метод пакетной верификации

Альтернативным способом является метод пакетной верификации, который заключается в объединении всех или части требований, подготовке и решении для них одной задачи достижимости. На практике объединение формализованных требований может производиться вручную и представлять собой новое требование. В частности, подобные решения использовались в подходе BMC [75]. В данном случае по построению решается задача переиспользования полученных знаний во время верификации. Но при этом появляются следующие проблемы. Во-первых, задача может стать потенциально неразрешимой (т. е. при построении абстракции возникает экспоненциальный рост числа состояний), поскольку для проверки нескольких требований необходима более точная абстракция. В частности, это может привести к потере части результата из-за недостатка ресурсов в сравнении с последовательной верификацией. Во-вторых, остается открытой проблема остановки верификатора после нахождения нарушения, что приводит к тому, что остальные требования проверяться не будут и их нарушения могут быть потеряны. Эта проблема существует всегда в статической верификации, но в данном случае проявляется наиболее остро. Поэтому рассматриваемый метод не способен предоставить результат, сопоставимый с последовательной верификацией.

Таким образом, метод последовательной верификации неэффективен, а метод пакетной верификации ведет к существенному ухудшению результата.

1.4. Методы переиспользования информации в статической верификации

Основная причина неэффективности последовательной верификации заключается в отсутствии переиспользования знаний о верификации. Решение данной проблемы «в лоб» в методе пакетной верификации (т. е. переиспользование всех знаний о верификации) приводит к существенным потерям результата. В

данном пункте представлены существующие методы решения данной проблемы.

Стоит отметить, что в данном обзоре не рассматриваются методы параллельной верификации, поскольку подобные методы главным образом нацелены на уменьшение астрономического времени решения задач за счет использования нескольких компьютеров или нескольких ядер процессора, а не на снижение требуемых вычислительных ресурсов.

1.4.1. Адаптивный статический анализ

Адаптивный статический анализ [57] (от англ. configurable program analysis), или **CPA**, нацелен на объединение нескольких абстрактных доменов в виде элементов CPA для использования их преимуществ. Формально каждый CPA состоит из абстрактного домена (abstract domain), текущей точности абстракции (precicion), отношения переходов (transfer relation), определяющего для каждого абстрактного состояния множество его потомков, оператора слияния (merge operator), определяющего, какие абстрактные состояния следует объединить, и оператора останова (stop operator), который определяет, является ли данное абстрактное состояние покрытым одним из уже построенных. Элементы CPA могут быть объединены в так называемый композитный CPA, что позволит объединять эффективность одного абстрактного домена с точностью другого. Адаптивный статический анализ можно рассматривать независимо от алгоритмов статической верификации (например, в подходе CEGAR он будет отвечать за построение АГД), поэтому он позволяет объединять абстрактные домены. Во многих случаях подобное комбинирование позволяет существенно ускорить процесс верификации или повысить точность верификации. Таким образом, адаптивный статический анализ позволяет комбинировать различные виды анализа для переиспользования информации внутри алгоритма верификации.

1.4.2. Условная проверка моделей

Как известно, статическая верификация программного обеспечения в общем случае является неразрешимой задачей, поэтому на практике не все задачи удается успешно решить (например, для них исчерпываются выделенные ресурсы). В данном случае ничего полезного не было получено и ресурсы были потрачены впустую. При этом известно много примеров, в которых неразрешимая задача в одном абстрактном домене (или в общем случае с помощью одного верификатора) относительно легко решается в другом.

Метод, решающий данную проблему, получил название **условной проверки моделей** (от англ. conditional model checking) [58]. Основная идея метода заключается в том, что вместо стандартного вердикта *Unknown* верификатор должен выдавать некоторое условие, которое описывает успешно верифицированные части программы. В качестве такого формата был предложен специальный автомат (англ. assumption automaton), который строится на основе АГД заменой всех полностью проверифицированных ветвей на переход в состояние *True*. Далее любой другой верификатор (или тот же самый, но с другой конфигурацией), поддерживающий указанный формат, принимает на вход данный автомат и продолжает верификацию на тех путях АГД, которые не были полностью проверены (т. е. которые не ведут в состояние *True*). Для этого необходимо по данному на вход автомату восстановить соответствующую часть АГД, пропустив при этом все ветви с переходом в состояние *True*.

Данная идея может быть обобщена и на случай нескольких произвольных верификаторов. Последовательная условная проверка моделей предполагает наличие N верификаторов, каждый из которых может выдавать условие в случае неуспешного завершения верификации и принимать это условие на вход. Процедура продолжается до тех пор, пока задача не будет решена (рисунок 1.6). При этом вердикт *Unknown* выдается только в том случае, если все верификаторы

не справились с поставленной задачей.

Данный метод противопоставляется объединению абстрактных доменов внутри одного запуска верификатора на основе адаптивного статического анализа. Во-первых, условная проверка моделей переиспользует только часть информации и только в случае, в котором это действительно необходимо (т. е. если один алгоритм не справился с задачей). Во-вторых, адаптивный статический анализ требует реализации всех алгоритмов внутри одного инструмента в виде CPA, а кроме того, не может разрешить ситуацию, при которой один из CPA завершает работу некорректно.

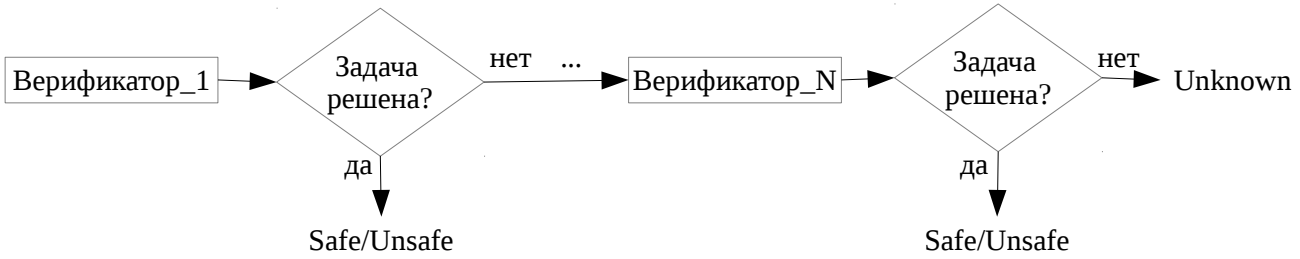

Рис. 1.6. Схема последовательной условной проверки моделей.

Данный метод был реализован в инструменте статической верификации CPAchecker. Для экспериментов рассматривались 2 конфигурации одного инструмента – анализ явных значений [41] и предикатная абстракция [40]. Оба абстрактных домена имеют свои преимущества и недостатки – анализ явных значений требует меньше итераций алгоритма CEGAR для определения точности и достаточно эффективен во многих случаях, однако он тратит больше времени на обход циклов; предикатная абстракция более ресурсоемка, в особенности для верификации относительно больших программ. Сравнивалось использование каждого из данных абстрактных доменов по отдельности, их объединение с помощью адаптивного статического анализа и с помощью условной проверки моделей. Эксперименты проводились с набором из 85 задач достижимости, взятых из категорий «DeviceDrivers64» и «SystemC» мероприятий SV-COMP'12 [12]. На

решение одной задачи давалось 15 минут процессорного времени и 15 GB оперативной памяти. Для эксперимента с условной проверкой моделей сначала запускался анализ явных значений, которому отводилось 100 секунд, в случае неуспешного его завершения происходила передача соответствующего автомата предикатной абстракции, суммарное ограничение времени на оба верификатора составило 15 минут (как и в остальных экспериментах). Результаты экспериментов представлены в таблице 1.1.

Эксперимент	Решено задач	Процессорное время (с)
Анализ явных значений	38 / 85	27 000
Предикатная абстракция	58 / 85	31 000
Адаптивный статический анализ	51 / 85	34 000
Условная проверка моделей	75 / 85	14 000

Табл. 1.1. Результаты экспериментов с адаптивным статическим анализом и условной проверкой моделей.

Проведенные эксперименты показывают преимущество условной проверки моделей как относительно использования различных конфигураций верификатора по отдельности, так и относительно их комбинации с помощью адаптивного статического анализа. Недостатки данного метода начинают проявляться тогда, когда передаваемый автомат становится слишком большим. Так для 7 случаев (8%) автомат состоял из более чем 200 000 состояний и накладные расходы на его сохранение и считывание превосходили все преимущества данного метода. Потенциально данная проблема может быть решена с помощью создания более простого языка описания условий, который позволит передавать части АГД в более компактном виде.

Таким образом, данный метод на основе частичного переиспользования информации о верификации между различными алгоритмами верификации позволяет во многих случаях как существенно ускорить верификацию, так и разрешить ранее неразрешимые задачи.

1.4.3. Регрессионная верификация

Переиспользование знаний о верификации широко распространено в методах регрессионной верификации. Регрессионная верификация нацелена на проверку каждой новой ревизии программы для предотвращения добавления новых ошибок. При этом предлагается использовать тот факт, что проверяемые программы достаточно похожи, поэтому частично можно переиспользовать промежуточные результаты верификации.

Основная идея подобных методов заключается в том [59], что инструменты верификации тратят много времени на вычисление некоторых **верификационных фактов**, необходимых для доказательства корректности проверяемого требования или нахождения его нарушения в программе, и в большинстве случаев эти результаты забываются после завершения верификации, а при верификации новой версии программы данные верификационные факты придется получать заново. При решении задачи регрессионной верификации требуется проверить большое количество версий программы, т. е. примерно одинаковый исходный код, поэтому достаточно полезно сохранять и в дальнейшем переиспользовать подобные промежуточные результаты. Вначале, как правило, необходима полная верификация исходной версии программы, полученные при этом верификационные факты запоминаются. В дальнейшем при верификации измененной программы запомненные ранее верификационные факты будут переиспользованы с целью получения результата быстрее. Основной вопрос, который необходимо решать в методах регрессионной верификации, состоит в определении того, какие верификационные факты наиболее эффективно переиспользовать и каковы могут быть негативные последствия подобного переиспользования. В общем случае возможно переиспользование различных типов верификационных фактов.

В работе [43] был предложен метод переиспользования точности абстракции

в подходе CEGAR, который позволил достичь ускорения верификации в несколько раз, однако в некоторых случаях использование большей точности при построении абстракции приводило к существенному усложнению задачи верификации и тем самым к замедлению верификации. Метод был реализован в статическом верификаторе CPAchecker.

Большинство подходов верификации опирается на возможности решателей [42], поэтому в работе [60] был предложен метод переиспользования ранее полученных результатов запросов к решателям между различными запусками верификаторов. Преимущества метода – поддержка произвольных методов верификации, высокая эффективность при верификации небольшого количества похожих программ (ускорение верификации до 6 раз). Главный недостаток метода – масштабируемость, поскольку с ростом числа различных верифицируемых программ совпадений запросов к решателям становится все меньше. Данный метод был реализован в инструменте Green и использовался поверх верификатора Symbolic PathFinder [61], который взаимодействовал с решателями CVC3 [62], Choco [63] и LattE [64].

В методах проверки моделей (от англ. explicit state model checker) был предложен метод переиспользования пространства состояний [65], который для незначительных изменений в программе демонстрировал ускорение в среднем порядка 30 раз, однако при относительно большом количестве изменений ускорение практически отсутствовало. Данный метод был реализован в инструменте Java PathFinder [66].

Аналогичный результат был получен и для подхода BMC [36], в рамках которого был предложен метод переиспользования аннотаций функций [67] (логических формул из присваиваний и условий для каждой функции), – большое ускорение при верификации похожих версий программы (порядка 11 раз) и замедление верификации из-за накладных расходов для достаточно

различающихся версий программы. Данный метод был реализован в верификаторе eVolCheck [67].

Таким образом, переиспользование каждого типа верификационных фактов имеет преимущества и недостатки, а также свои ограничения. При переиспользовании «полезной» информации верификация ускоряется в разы, однако переиспользование «лишней» информации может только замедлить верификацию.

1.4.3.1. Переиспользование точности абстракции

Рассмотрим более подробно особенности метода переиспользования точности абстракции [43], который наиболее подходит для подхода CEGAR. Как правило, в подходе CEGAR практикуется построение первой абстракции по пустой точности, т. е. абстракция создается наиболее грубой и в результате получается наиболее простой АГД. Если был получен ложный контрпример из-за недостаточной точности абстракции, то производится уточнение абстракции. Данная процедура продолжается в цикле до получения необходимой точности абстракции, с помощью которой и будет построен АГД, позволяющий доказать корректность проверяемого требования или найти его нарушение. Поэтому нахождение требуемой точности абстракции является одним из наиболее ресурсоемких шагов в подходе CEGAR, и если на вход сразу подать требуемую точность абстракции, то верификатор пропустит наиболее ресурсоемкий шаг верификации и сразу сможет построить требуемый АГД. Кроме того, задание на вход алгоритма некорректной точности (например, включающей в себя удаленную при модификации программы переменную) не повлияет в худшую сторону на верификацию, поскольку она будет проигнорирована, а требуемая точность будет заново определена, как и без переиспользования.

Для хранения информации о полученной точности был предложен

формат [43], который может использоваться для любого абстрактного домена. Данный формат подразумевает сохранение пар «место в программе» и «точность абстракции». Поддержка данного формата была реализована в инструменте CPAchecker, который может быть запущен как с опцией для сохранения полученной точности абстракции в данном формате в некоторый файл, так и с опцией для считывания ранее полученной точности из указанного файла перед началом алгоритма CEGAR. Если в ходе верификации очередной версии программы точность была изменена относительно переиспользуемой, то данное изменение также будет запомнено. Данный метод был реализован для двух абстрактных доменов – предикатной абстракции и анализа явных значений. В случае предикатной абстракции в качестве точности используются предикаты, которые получаются автоматически при доказательстве невыполнимости формулы, присутствующей в контрпримере. При анализе явных значений в качестве точности берутся имена переменных программы.

Для экспериментальной оценки данного метода использовался набор из 4 193 задач достижимости, полученных с помощью системы LDV Tools [23-26] при верификации различных версий 62 драйверов операционной системы Linux относительно 6 различных требований, суммарно было рассмотрено 1 119 различных версий драйверов [43]. В качестве абстрактного домена отдельно рассматривался анализ явных значений [41] и предикатная абстракция [40]. Ограничения по ресурсам составили 15 минут процессорного времени и 15 GB оперативной памяти. Предикатная абстракция успешно справилась с решением 4 048 задач за 83 000 секунд процессорного времени без переиспользования точности и с 4 078 задач за 23 000 секунд с переиспользованием. Анализ явных значений в обоих случаях успешно справился с решением 4 193 задач, без переиспользования за 13 000 секунд, а с переиспользованием за 4 900 секунд. При этом среднее ускорение верификации при переиспользовании точности

абстракции составило 4.3 (предикатная абстракция) и 3.8 (анализ явных значений). Файлы с сохраненной точностью занимали не более 4 килобайт.

Для нескольких задач (менее 1%) время верификации возросло при переиспользовании точности, что показывает потенциальные ограничения в применимости данного метода. Детальный анализ подобных ситуация показал, что иногда для измененной программы требуется более грубая абстракция, которой соответствует гораздо более простой АГД, т. е. изменение упростило программу для верификации, а верификатор вынужден был это игнорировать из-за переиспользования. Например, в одном драйвере из абстракции ушла переменная-счетчик цикла, на анализ которого тратилась большая часть времени, поэтому переиспользование точности увеличило время верификации в 2 раза. Тем не менее данные эксперименты показывают, что подобных ситуаций относительно мало.

1.4.4. Перепроверка результатов верификации

При успешном завершении верификации результатом является либо трасса ошибки, либо доказательство корректности (например, в виде АГД). Для унификации трасс ошибок в [15] был предложен единый формат, который должны поддерживать все статические верификаторы, участвующие в мероприятиях SV-COMP. Это позволило подавать трассы ошибок, полученных от одного инструмента, для перепроверки другому для исключения ложных сообщений об ошибке, вызванных проблемами конкретного инструмента. Эксперименты показали, что подобная перепроверка требует не более 4% от времени, необходимого на нахождение оригинальной трассы ошибки. Помимо этого, трасса ошибки может быть переиспользована для того, чтобы быстрее обнаружить ее в измененной версии программы. Однако на практике обычно требуется убедиться, что в новой версии программы больше нет ранее обнаруженной ошибки.

На данный момент ведутся работы в области унификации доказательства

корректности различных верификаторов, что позволит переиспользовать результат верификации и для вердиктов *Safe* [15], однако сейчас полные доказательства занимают слишком много места, поэтому их переиспользование не повышает эффективность верификации.

1.4.5. Комбинирование тестирования и верификации

Поскольку тестирование остается общепринятой практикой проверки программного обеспечения, то некоторые методы нацелены на совместное использование тестирования и верификации. Для этого результаты одного из методов переиспользуются в другом.

Поскольку качество тестирования напрямую зависит от тестового набора, то в работе [68] предлагается повысить качество тестового набора с помощью автоматической генерации тестов с использованием статических верификаторов. Основываясь на подходе CEGAR и предикатной абстракции, предлагается разбить множество всех входов программы на группы, которые могут быть описаны, например, с помощью автомата, после чего генерировать тесты для каждой группы. При этом тесты можно переиспользовать для последующих версий программы, поэтому данный метод потребует гораздо меньше ресурсов, чем регрессионная верификация.

В методе раздельной регрессионной верификации [69] (от англ. partition-based regression verification) предлагается итеративный алгоритм разбиения входов программы, на которых возможно доказательство корректности программы с помощью решателя. Метод не предполагает полноценное доказательство корректности, однако нацелен на перебор большинства входов программы, на которых может гарантироваться корректность. При этом те входы программы, для которых не удалось доказать корректность, используются при подготовке дополнительных тестовых наборов. В экспериментах раздельная регрессионная

верификация помогла найти ошибку в интерпретаторе командной строки Apache CLI, которая была пропущена в процессе проведения базового тестирования.

В работах [70-72] представлены статико-динамические методы, которые нацелены на использование сильных сторон верификации и тестирования. В частности, статическая верификация использует большое количество упрощений, например, поддерживаются не все конструкции языка программирования, поэтому полная корректность программ все равно не гарантируется. Для каждого подобного упрощения предлагается создавать набор тестов, что позволит увеличить число находимых ошибок.

Таким образом, подобные методы демонстрируют, что переиспользование знаний верификации может быть полезно и за пределами верификации.

1.4.6. Проверка нескольких требований в смежных областях верификации

В методах верификации устройств (от англ. hardware verification) уже предлагалась одновременная проверка нескольких требований. Например, в работе [73] был предложен метод интерполяции на основе уточнения (от англ. interpolation with guided refinement), для которого авторы предлагают его расширение для проверки нескольких требований на основе переиспользования пространства состояний, однако эксперименты с данным расширением не приводятся. В работе [74] утверждается, что предложенная модификация алгоритма ic3 применима как для регрессионной верификации незначительно измененных программ, так и для незначительно измененных требований к программе, однако эксперименты проводятся только для модифицированных программ.

В подходе BMC также известны алгоритмы для совместной проверки нескольких формально заданных требований на основе расширения формулы, передаваемой решателю [75]. Применение данного метода к верификации

устройства Intel позволило достичь ускорения в 7 раз. Однако основной целью ставилось доказательство корректности, поэтому не рассматривалась проблема невозможности доказательства некоторого требования, как и возможность нарушения нескольких требований.

Таким образом, использование совместной проверки нескольких требований к программе применялось в смежных областях верификации аналогично методу пакетной верификации, при этом его проблемы не поднимались и не решались.

1.5. Выводы

Проведенный обзор методов подготовки и решения задач статической верификации и методов переиспользования информации в верификации показал, что рассмотренные методы не подходят напрямую для решения поставленной цели работы. Для успешной верификации композиции требований необходимо изменить весь процесс верификации. Во-первых, необходимо подготавливать задачу достижимости относительно нескольких требований, в то время как используемые на практике методы инструментирования не предназначены для этого. Во-вторых, необходимо решать подготовленные таким образом задачи достижимости без потери результата в сравнении с методом последовательной верификации. Для этого необходимо устранить проблемы, которые возникают при верификации композиции требований, т. е. остановку верификации после нахождения первого нарушения требования и экспоненциальный рост числа состояний. При решении проблемы экспоненциального роста числа состояний необходимо учитывать опыт существующих методов переиспользования информации в верификации.

Таким образом, для решения поставленной цели работы необходима разработка новых методов статической верификации программного обеспечения.

Глава 2. Методы многоаспектной верификации

В данной главе описываются методы обнаружения всех однотипных нарушений и условной многоаспектной верификации. С одной стороны, в данных методах преследуется цель – повышение производительности верификации композиции требований при сохранении качества верификации, а с другой – нахождение большего числа нарушений требований в программах.

2.1. Подготовка задачи достижимости относительно композиции требований

Для любого метода верификации композиции требований вначале необходимо подготовить задачу достижимости. Самое простое и примитивное решение, которое используется в пакетной верификации, заключается в ручном объединении моделей и создании тем самым новой модели, на основе которой и будет подготовлена задача достижимости. Однако если множество проверяемых требований часто меняется, то более предпочтительным является автоматическое объединение моделей требований. Рассмотрим способ автоматического объединения моделей требований для подготовки задач достижимости на основе аспектно-ориентированного расширения языка C (см. п. 1.2.1), который нацелен на сохранение корректности (т. е. результат верификации для объединенной модели должен совпадать с результатом верификации каждого из них по отдельности).

2.1.1. Ограничение множества объединяемых моделей требований

Поскольку метод инструментирования обладает широкими возможностями по формализации требований, то необходимо учитывать, что какие-то модели в общем случае могут «мешать» друг другу, поэтому автоматически объединять их нельзя. Например, в модели некоторого требования ограничивается множество возвращаемых значений функции *kmalloc* до *NULL* (что в общем случае неверно, однако допустимо для данного требования и серьезно упрощает его проверку,

позволяя находить для него больше нарушений). При объединении такой модели с другими возможен пропуск их нарушений, для получения которых функция *kmalloc* должна вернуть не *NULL*.

Для решения подобных проблем рассматриваемое множество моделей требований было ограничено следующим образом:

1. Запрещается изменять существующие операторы программы за исключением, возможно, замены вызовов исходных функций (макрофункций) на соответствующие им модельные (сохранение корректности).
2. Все модельные функции, которые заменяют вызовы исходных функций (макрофункций), должны быть адекватны соответствующим исходным функциям, т. е. запрещается ограничивать пути выполнения программы относительно исходной функции (сохранение полноты).
3. Запрещается использовать конструкции языка программирования С, которые не поддерживаются (или частично поддерживаются) статическим верификатором.
4. Все объединяемые модели должны использовать один и тот же процесс подготовки исходного кода.

Первое ограничение исключает модели, которые могут привести к нахождению ложных нарушений других требований (при модификации операторов программы могут появиться новые пути выполнения, которые не соответствуют оригинальной программе). Второе ограничение исключает модели, которые могут привести к пропуску нарушений других требований (из-за ограничения путей выполнения оригинальной программы). Третье ограничение нацелено на исключение ситуаций, в которых неподдерживаемые конструкции одной модели приведут к невозможности верификации других. Последнее ограничение предназначено для возможности создания верификационных объектов (т. е. проверяемого кода) относительно нескольких моделей.

Таким образом, указанные выше ограничения выделяют класс таких

моделей требований, которые добавляют вспомогательные проверки, необходимые для проверки требования, при этом не изменяя поведение самой программы.

2.1.2. Алгоритм объединения моделей требований

Пусть заданные модели требований удовлетворят ограничениям из п. 2.1.1. Алгоритм их объединения включает в себя два шага.

На первом шаге объединяются все модельные функции, при этом производится переименование всех используемых имен добавлением в качестве суффикса уникального идентификатора соответствующего требования. Данная процедура исключит возможные конфликты имен, а также позволит позже идентифицировать, к какому требованию относится то или иное имя. В частности, нарушению требования будет отвечать отдельная метка ошибки.

Например, модельная функция из требования с идентификатором X:

```
int model_var;
void model_function(void) {
  assert(model_var >= 1); // утверждение
  model_var--;
}
```

будет преобразована следующим образом:

```
int modev_var_X;
void model_function_X(void) {
  assert_X(model_var_X >= 1); // утверждение X
  model_var_X--;
}
```

Помимо этого, для каждого требования создается отдельное утверждение:

```
void error_X(void);
void assert_X(int expr) {
  if (!expr)
ERROR_X: error_X();
}
```

На втором шаге объединяются точки использования. Если множества точек

использования для каждого объединяемого требования не пересекаются, то никаких дополнительных операций на данном шаге не требуется. Предположим, что имеются две модели требований с идентификаторами *X* и *Y*, которые для одной и той же точки использования добавляют модельные функции

`return_type model_function_X(p)` и
`return_type model_function_Y(q)`,

где *return_type* – тип возвращаемого значения (возможно, *void*), *p* и *q* – список параметров, возможно, пустой. Для подобных ситуаций необходимо создать вспомогательную функцию, в которой будет производиться вызов каждой из исходных модельных функций с учетом их возвращаемого значения, по следующей схеме:

```
return_type joint_model_function(p, q) {
  return_type result_X = model_function_X(p);
  return_type result_Y = model_function_Y(q);
  /* Проверка возвращаемых значений */
  assume(result_X == result_Y);
  return result_X;
}
```

Проверка возвращаемых значений исключит из рассмотрения верификатора все пути, в которых исходные модельные функции возвращают разные значения (т. е. исключит возможность добавления ранее невозможных путей выполнения программы). Конструкция *assume* с учетом специфики современных верификаторов представляется в виде бесконечного цикла следующим образом:

```
void assume(int expr) {
  if (!expr)
STOP: goto STOP;
}
```

В случае, если модельные функции не возвращают значений (т. е. *return_type* равен *void*), то дополнительной проверки возвращаемых значений не требуется.

Аналогично данную процедуру можно обобщить на произвольное

количество модельных функций для одной точки использования.

Для предложенного метода объединения моделей требований справедливо следующее утверждение (доказательство приведено в Приложении В):

Утверждение 1. Полнота и корректность верификации требований, удовлетворяющих ограничениям из п. 2.1.1, не изменяются при подготовке задач достижимости относительно композиции требований в сравнении с подготовкой задач достижимости относительно каждого требования в отдельности.

Примечание. При этом на практике возможна потеря результата (т. е. невозможность решить часть задач достижимости с теми же ограничениями на ресурсы, что и в базовом методе), поскольку подготовленная задача достижимости на основе объединения моделей требований заведомо более сложная.

2.2. Метод обнаружения всех однотипных нарушений

Первая проблема, которая мешает использованию метода пакетной верификации и тем самым должна быть решена для верификации композиции требований, заключается в том, что базовые подходы верификации (в том числе и CEGAR) останавливаются после нахождения нарушения требования (рис. 1.3), поскольку программа уже нарушает проверяемое требование и нет необходимости пытаться доказать ее корректность. Однако в общем случае нарушение одного требования автоматически не влечет за собой нарушение остальных требований и не ведет к невозможности доказательства их корректности, поэтому при верификации программы относительно нескольких требований существует очевидная необходимость продолжения верификации после нахождения нарушения одного требования. Помимо этого, данное ограничение существует и при проверке одного требования – базовый подход не способен находить более одного нарушения проверяемого требования. При этом существуют примеры программ, в которых одно требование нарушается несколько раз. Для того чтобы с

помощью подхода CEGAR найти более одного нарушения заданного требования, необходимо итеративно исправлять найденные нарушения и повторять верификацию до того момента, пока не будет доказана корректность программы относительно проверяемого требования (т. е. получен вердикт *Safe*). Однако на практике такое решение неэффективно, поскольку требуется неопределенное число раз решать практически одну и ту же задачу достижимости, кроме того, процесс исправления нарушения требования не может быть автоматизирован. Например, известны случаи, в которых исправление нескольких нарушений одного требования в модулях ядра Linux, выявленных с помощью статической верификации, занимало месяцы [32].

Для решения данной проблемы предлагается метод **обнаружения всех нарушений**, или **ОВН** [32], (англ. multiple error analysis – MEA). Метод ОВН нацелен на нахождение всех однотипных[3] нарушений требования и анализ найденных трасс ошибок (рис. 2.1). Очевидно, что метод ОВН требует больше ресурсов, чем базовый алгоритм CEGAR, поскольку он продолжает работу там, где базовый алгоритм завершается.

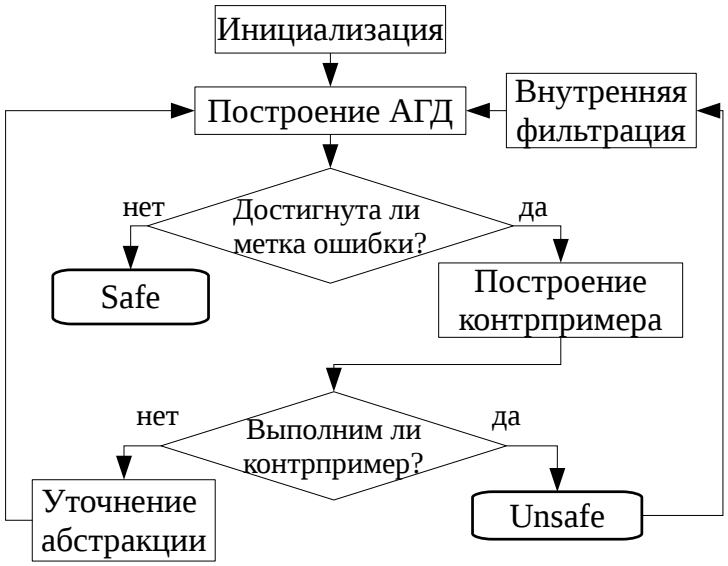

Рис. 2.1. Схема метода ОВН.

3 Для краткости термин «однотипных» далее будет опускаться.

Основная проблема, которая возникает в данном методе, заключается в том, что каждому нарушению требования может соответствовать несколько, а в общем случае и бесконечно много проявлений (т. е. «наведенных» трасс ошибок). Например, имеется утечка памяти в функции, которая может вызываться произвольное количество раз. С одной стороны, данная проблема ведет к тому, что время работы метода существенно увеличивается и в общем случае нельзя найти все нарушения требования. В частности, это приводит к появлению нового типа вердиктов в данном методе – *Unsafe-incomplete*, при котором часть трасс ошибок была найдена, но верификация полностью не была завершена (например, были израсходованы выделенные ресурсы), т. е. часть нарушений требования, возможно, была пропущена. С другой стороны, трассы ошибок, соответствующие одному нарушению требования, не представляют никакой пользы для человека, анализирующего результаты работы метода с целью исправления найденных нарушений. Например, метод установил, что проверяемое требование в программе нарушается несколько тысяч раз, а при анализе результата человеком выясняется, что на самом деле нарушений всего 2-3. В данном случае человеку необходимо будет потратить достаточно много времени для того, чтобы установить подобный факт, что делает непривлекательным использование предложенного метода.

Очевидным решением данной проблемы является фильтрация найденных трасс ошибок. Однако в общем случае данная задача является неразрешимой, поскольку автоматически точно установить, где именно находится причина нарушения требования в трассе, и тем более получить исправление найденного нарушения, нельзя. С точки зрения человека, нарушение требования характеризуется либо словесным неформальным описанием (например, «пропущено освобождение *data->lock_mutex* на одном из путей в *ep_read()*»[4]), либо исправляющим нарушение требования патчем[5], который также создается

4 Описание ошибки, найденной с помощью системы LDV Tools: http://linuxtesting.ru/results/report?num=L0029.

5 Пример патча: http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=00cc7a5f.

вручную. Кроме того, трасса ошибки может соответствовать ложному сообщению об ошибке (например, из-за неполной поддержки верификатором некоторых элементов языка программирования C [25]), что также определить может только человек. Поэтому для фильтрации найденных трасс ошибок необходимо формализовать понятие их эквивалентности.

2.2.1. Формализация эквивалентности трасс ошибок

Под **трассой ошибки** понимается последовательность операций, соответствующая пути выполнения программы от точки входа до метки ошибки. При этом каждая операция должна соответствовать определенной строке в исходном коде верифицируемой программы. Поддерживаются следующие типы операций:

- CALL <имя функции> – вызов функции;
- RETURN <возвращаемое значение> – возврат из последней вызванной функции;
- ASSIGN <переменная> <значение> – оператор присваивания;
- ASSUME <условие> – выполнение условия в операторе ветвления;
- GOTO <метка> – оператор перехода.

Данный формат является упрощением единого формата трасс ошибок [15], который должны поддерживать большинство статических верификаторов, т. е. любую трассу ошибки от данных верификаторов можно представить в предложенном виде.

Например, для программы, представленной на рисунке 1.1, существует следующая трасса ошибки (в формате <номер строки> <операция> <параметры>):

```
2   CALL    strncpy
2   ASSIGN  dst 0
2   ASSIGN  src "str"
2   ASSIGN  n 3
3   ASSUME  (src != 0)
5   ASSUME  (!(dst != 0))
6   GOTO    ERROR_2
```

Пусть даны две трассы ошибки t_1 и t_2, представленные в указанном выше формате. Данные трассы называются **эквивалентными относительно фильтра *f***, если $f(t_1, t_2) = true$. В идеале функция f должна возвращать ложь для тех трасс ошибок, которые соответствуют различным нарушениям требования, и истину в противном случае. Однако поскольку данная проблема является неразрешимой, то на практике можно лишь предложить аппроксимацию данной функции. Основное ограничение на данную функцию заключается в том, что она не должна возвращать истину для трасс ошибок, соответствующих различным нарушениям требования (т. е. терять новые найденные нарушения требования), при этом функция может вернуть ложь для трасс ошибок, соответствующих одному нарушению требования (т. е. отфильтровать не все трассы ошибок). В общем случае данная функция может быть задана вручную.

2.2.2. Автоматическая фильтрация трасс ошибок

Несмотря на то что фильтрация найденных трасс ошибок автоматически не является разрешимой задачей, можно предложить различные аппроксимации фильтра f для исключения очевидных дубликатов трасс ошибок, что позволит решить часть проблем автоматически и оптимизирует последующие действия по их обработке. Например, всегда можно сравнивать трассы на полную эквивалентность.

Автоматическая фильтрация может быть добавлена на разных шагах алгоритма. Наиболее эффективный способ – сравнивать трассы во время работы алгоритма верификации на основе их внутреннего представления – так называемая внутренняя фильтрация (рис. 2.1). Основная задача внутренней фильтрации заключается в том, чтобы исключить идентичные трассы ошибок, которые были получены разными способами при построении АГД. Внутренняя фильтрация представляет собой реализацию фильтра f внутри верификатора. Все трассы

ошибок, прошедшие фильтрацию, представляются элементами множества *T*. Каждая новая найденная трасса ошибки t_1 проходит фильтрацию и добавляется во множество *T* при выполнении следующего условия:

$$f(t_1, t_2) = false, \ \forall t_2 \in T$$

Если указанное условие не выполняется, то трасса считается эквивалентной одной из уже найденных, поэтому она отбрасывается и верификатор не тратит ресурсы на дополнительную обработку данной трассы (например, означивание переменных, вывод трассы в файл). В качестве функции *f* можно использовать, например, сравнение на полную эквивалентность или сравнение по блокам программы в ГПУ [40], которое исключит трассы, различающиеся только означиванием переменных. Подобные критерии, во-первых, эффективно реализуются внутри верификатора и тем самым требуют минимум накладных расходов, во-вторых, применимы к произвольным программам.

Однако внутренняя фильтрация никак не учитывает специфику формализации требований, которая позволяет задать более узкие критерии сравнения трасс ошибок. Кроме того, необходимо принимать во внимание то, как именно трассы ошибок предоставляются для анализа человеку, чтобы сравнивать только их «полезные» элементы. Например, начальная инициализация некоторых структур данных может не иметь никакого отношения к проверяемому требованию, а потому при сравнении трасс ошибок, полученных при проверке данного требования, вполне можно пренебречь подобной инициализацией. Для учета специфики формализации требований была предложена внешняя фильтрация трасс ошибок, которая производится после завершения работы верификатора системой верификации. Данный тип фильтрации основан на **функции преобразования** *conversion(t)*, которая удаляет из трассы ошибки *t* все элементы, не существенные для нарушенного требования. При этом фильтр *f* задается следующим образом – две трассы считаются эквивалентными, если

результат применения к ним функции преобразования совпадает, т. е.:

$$f(t_1, t_2) = (conversion(t_1) \equiv conversion(t_2))$$

Например, функция преобразования может отбросить операторы присваивания для всех переменных, не относящихся к проверке требования. Данный тип фильтрации может занимать больше времени и не быть применимым к произвольным способам формализации требований.

Таким образом, предложенные методы автоматической фильтрации позволяют существенно сократить число трасс ошибок, которые необходимо анализировать человеку, а во многих случаях и полностью решить задачу фильтрации трасс ошибок.

2.2.3. Полуавтоматическая фильтрация трасс ошибок

Как уже отмечалось, в общем случае фильтр f может быть задан сколько угодно сложно вручную, что позволит решить проблему фильтрации трасс ошибок в частном случае. Однако задавать отдельную функцию для каждого конкретного случая не всегда удобно, поэтому предлагается полуавтоматическая фильтрация, которая систематизирует процесс задания фильтра и позволит за минимальное число действий человека выделить трассы ошибок, отвечающие различным нарушениям требования.

Для того чтобы понять, какие действия требуются от человека, рассмотрим примеры ситуаций, в которых не удается автоматически отфильтровать все трассы ошибок, соответствующие одним и тем же нарушениям требования. Во-первых, возможны ситуации, в которых одно нарушение требования проявляется в различных местах. Например, имеются 2 трассы ошибки:

```
function_1() <причина нарушения требования>
function_2() <проявление нарушения требования 1> и

function_1() <причина нарушения требования>
function_3() <проявление нарушения требования 2>
```

В данном случае нарушение проверяемого требования (например, утечка ресурса) находится в одном месте, для исправления которого необходимо одно изменение в функции *function_1()*, однако проявляется оно в различных местах (например, в данных функциях осуществляются проверки). В то же самое время причина нарушения требования (т. е. место в программе, в котором необходимо выполнять исправление) могла бы находиться и в функциях *function_2()* и *function_3()*. В частности, возможна ситуация бесконечного числа подобных трасс ошибок, при которой одна трасса ошибки целиком содержится в следующей (так называемые повторные трассы ошибок).

Во-вторых, верификатор может находить несколько различных путей к одному нарушению требования, например:

```
function_1() <путь к нарушению требования 1>
function_3() <причина нарушения требования> и

function_2() <путь к нарушению требования 2>
function_3() <причина нарушения требования>
```

В данном случае также непонятно, требуется ли выполнять исправление только в функции *function_3()* (т. е. трассы эквивалентны) или в функциях *function_1()* и *function_2()* (т. е. трассы не эквивалентны).

Заметим, что для решения подобных проблем необходимо установить четкие границы причины нарушения требования и вместо всей трассы рассматривать только выделенную часть, что и было предложено в полуавтоматической фильтрации.

Процедура полуавтоматической фильтрации представляется следующим образом. В начале человек анализирует некоторую трассу ошибки, находит причину нарушения требования и помечает ее границы. После этого человек задает функцию преобразования *conversion(t)*, которая производит обработку трасс ошибок аналогично внешней фильтрации, и **функцию сравнения** *comparison*(t_1, t_2), которая определяет, как именно трасса t_1 будет сравниваться с

трассой t_2 (например, проверка на полное совпадение или включение). При этом используемый фильтр f строится автоматически следующим образом:

$$f(t_1, t_2) = comparison(conversion(t_1), conversion(t_2))$$

Те трассы ошибок, для которых функция f возвращает истину, помечаются эквивалентными анализируемой. Данная процедура повторяется до тех пор, пока все трассы ошибок не окажутся размеченными.

Главное достоинство данного метода состоит в том, что число действий человека по анализу трасс ошибок будет равно числу нарушений требований. Кроме того, если все (или наиболее часто используемые) функции преобразования и сравнения реализованы в специальной библиотеке, то человеку при проведении полуавтоматической фильтрации достаточно только выделять фрагмент трассы ошибки и конструировать фильтр f из уже реализованных элементов. При этом в случае некорректной разметки трасс ошибок (например, был выделен фрагмент трассы неверно) имеется возможность внести исправление и алгоритм переразметит трассы ошибок.

Стоит заметить, что в данном методе не делается различий между ложным сообщением об ошибке и реальной ошибкой в коде программы, оба термина заменяются нарушением проверяемого требования. В данном случае определение того, соответствует ли найденное нарушение требования ложному сообщению об ошибке или нет, производится уже после полуавтоматической фильтрации (возможно, другим человеком). Кроме того, на практике возможны ситуации, в которых в базовом подходе верификации нахождение ложного сообщения об ошибке не позволяет найти настоящую ошибку, что успешно решается в предложенном методе. В случае ложного сообщения об ошибке фильтрация уберет из рассмотрения все аналогичные трассы ошибок с тем же ложным сообщением об ошибке.

Таким образом, с помощью алгоритмов фильтрации метод ОВН способен

находить все нарушения заданного требования в проверяемой программе при получении вердикта *Unsafe* или часть из них при получении вердикта *Unsafe-incomplete*. Однако метод ОВН требует больше ресурсов как на саму верификацию, так и на проведение полуавтоматической фильтрации. Помимо этого, верификация может «зациклиться» на нахождении бесконечного числа повторных трасс ошибок для одного требования и результат для остальных требований не будет получен.

2.3. Алгоритм многоаспектной верификации

Многоаспектная верификация, или **MAB** [31, 33, 34], (англ. multi-aspect verification – MAV) нацелена на конфигурируемую проверку нескольких требований для заданной задачи достижимости за один запуск статического верификатора. Многоаспектная верификация предоставляет «разностороннюю» проверку программы (т. е. разных аспектов) за один запуск верификатора. Основная цель многоаспектной верификации заключается в повышении производительности верификации композиции требований за счет решения проблемы экспоненциального роста числа состояний. Здесь и далее многоаспектная верификация предлагается как расширение подхода CEGAR, но потенциально те же идеи могут быть применены и для других подходов статической верификации.

Предполагается, что задача достижимости была подготовлена на основе объединения требований (см. п. 2.1), модели которых заданы с помощью аспектно-ориентированного расширения языка C [47] и удовлетворяют ограничениям из п. 2.1.1. В отличие от метода пакетной верификации вместе с задачей достижимости необходимо передать алгоритму отображение идентификаторов проверяемых требований на соответствующие их нарушениям имена меток ошибок, что возможно при указании идентификатора требования в имени метки ошибки (метка ошибки *ERROR_X* соответствует нарушению требования *X*). На

выходе от данного алгоритма ожидается вердикт для каждого требования, как и в методе последовательной верификации.

2.3.1. Представление утверждений

Алгоритм MAB (рис. 2.2) принимает на вход множество меток ошибок и для каждой из них создает **утверждение**, которое далее будет проверять недостижимость соответствующей метки ошибки. Каждое такое утверждение состоит из следующих атрибутов: уникальный идентификатор, метка ошибки, вердикт утверждения, потраченные ресурсы и верификационные факты. На шаге *создание утверждений* для каждой метки ошибки выполняются следующие действия:

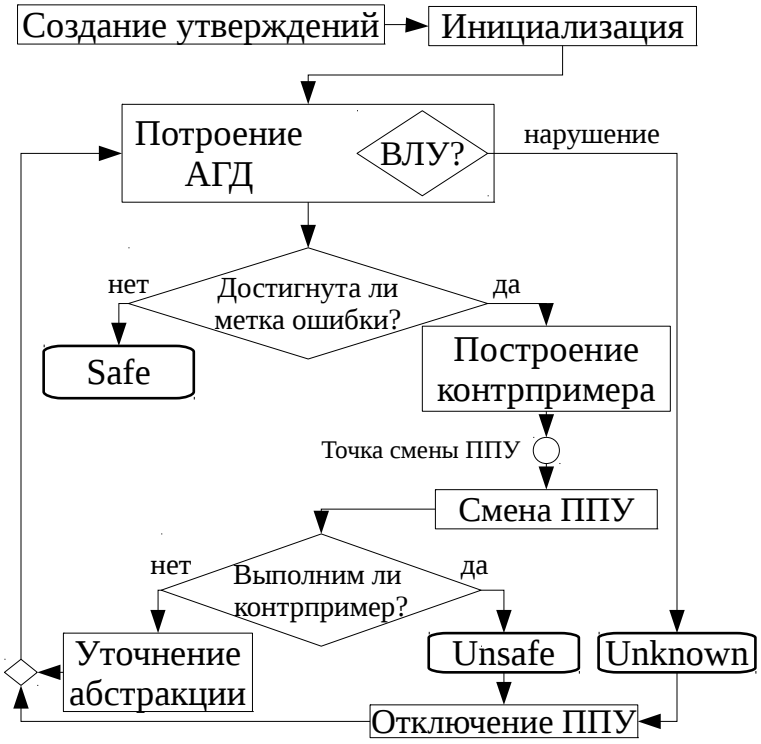

Рис. 2.2. Алгоритм MAB.

1. Идентификатор утверждения устанавливается равным идентификатору соответствующего требования.

2. Запоминается соответствующая утверждению метка ошибки. При этом следует отметить, что в данном случае может быть задана не обязательно метка в

терминах языка С, а некоторая точка в программе, например, вызов функции (для примера из п. 2.1.2 возможно задание метки *ERROR_X* или функции *error_X()*).

3. Вердикт утверждение устанавливается во вспомогательное значение *Checking*, которое показывает, что вердикт для данного утверждения еще не был получен.

4. Потраченные ресурсы и верификационные факты обнуляются. Потраченные ресурсы будут использоваться для ограничения времени на проверку каждого утверждения, а верификационные факты показывают уровень абстракции, который необходим для верификации конкретного утверждения. Потраченные ресурсы измеряются в процессорном времени, в качестве верификационных фактов могут использоваться любые знания о верификации программы (например, точность абстракции).

Вердикт утверждения представляет собой внутренний вердикт для данного утверждения в процессе работы алгоритма MAB. После завершения алгоритм выдает вердикт утверждения для каждого проверяемого утверждения, как и метод последовательной верификации. Вердикт утверждения может принимать стандартные значения вердикта (*Safe*, *Unsafe*, *Unknown*) и вспомогательное значение *Checking*, которое показывает, что вердикт еще не был получен.

Таким образом, по сравнению с методом пакетной верификации MAB способен выдавать вердикт для каждого проверяемого требования.

2.3.2. Аппроксимация с одним проверяемым утверждением

Для того чтобы знать, какое утверждение проверяется в данный момент времени (для смены вердикта утверждения, учета ресурсов и т. д.), вводится понятие **последнего проверяемого утверждения** (**ППУ**). ППУ – это утверждение, для которого алгоритм выполнял проверку последним, т. е. вел построение АГД на его основе. Однако совершенно очевидно, что во время построения АГД несколько

утверждений могут использоваться одновременно (в чем, собственно, и состоит идея MAB). Поэтому предлагается следующая аппроксимация: только одно утверждение проверяется в каждый момент времени, и алгоритм точно знает только то утверждение, которое проверялось последним, т. е. ППУ.

В начале работы алгоритма ППУ не задано. Далее все время верификации можно разделить так называемыми *точками смены ППУ*, которые представляют собой моменты времени, в которых ППУ меняет свое значение. При этом текущее проверяемое утверждение в рамках данной аппроксимации считается равным ППУ вплоть до следующей точки смены ППУ. Поскольку контрпример в подходе CEGAR всегда содержит достигнутую метку ошибки, то по нему всегда возможно определить, какое утверждение он нарушает. При этом то время, которое алгоритм потратил на построение АГД для получения данного контрпримера, в рамках предложенной аппроксимации предлагается считать потраченным на проверку нарушенного в контрпримере утверждения. Это предположение следует из того, что если контрпример является истинным, то так было найдено нарушение данного утверждения, а в противном случае абстракция уточняется в том числе и на основе «нарушенного» утверждения, т. е. можно гарантировать, что данное утверждение точно проверялось, хотя, возможно, и не только оно одно. Таким образом, *точки смены ППУ* обязательно должны присутствовать после шага *построение контрпримера* в алгоритме MAB (рис. 2.2).

После расстановки точек смены ППУ все время верификации разбивается на так называемые **интервалы ППУ**, которые представляют собой временные интервалы между двумя соседними точками смены ППУ. Это позволяет использовать интервалы ППУ для вычисления процессорного времени, потраченного на каждое утверждение, и получения соответствующих утверждению верификационных фактов с точностью до предложенной аппроксимации. Для этого на шаге *смена ППУ* выполняются следующие действия:

1. Время последнего интервала ППУ добавляется ко времени соответствующего ППУ.
2. Верификационные факты, полученные на последнем интервале ППУ, добавляются к верификационным фактам данного утверждения.

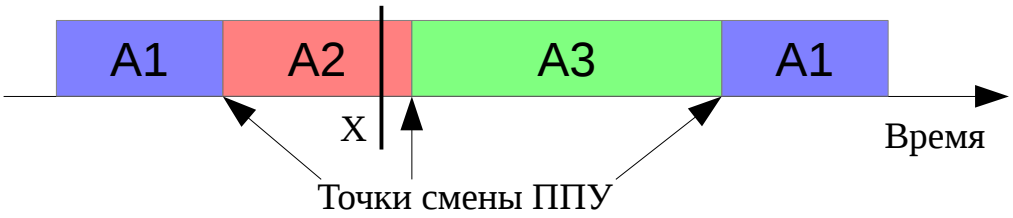

Рис. 2.3. Пример разбиения времени верификации на интервалы ППУ.

Рассмотрим следующий пример, представленный на рис. 2.3. Здесь все время верификации трех утверждений (A1, A2 и A3) было разбито на 4 интервала ППУ. Так, например, считается, что в самом начале верификации проверялось утверждение A1, в момент времени X – A2, а последним проверялось утверждение A1. Таким образом, данная аппроксимация в каждый момент времени позволяет:

1. мгновенно получить наиболее вероятное текущее проверяемое утверждение;
2. отслеживать затраченные на верификацию утверждения ресурсы;
3. определять уровень абстракции отдельно для каждого утверждения.

2.3.3. Остановка проверки утверждения

Поскольку MAB может отслеживать затраченное процессорное время для каждого утверждения отдельно, то появляется возможность ограничить время для каждого утверждения, как и в методе последовательной верификации, с помощью **внутреннего лимита утверждения** (**ВЛУ**). ВЛУ создается отдельно для каждого проверяемого утверждения на шаге *создание утверждений* и обнуляется. Далее на шаге *смена ППУ* соответствующий ППУ ВЛУ обновляется. Во время построения абстракции (шаг *построение АГД*) для соответствующего ППУ ВЛУ асинхронно проверяется, что сумма текущего интервала ППУ и время, ранее затраченное на ППУ, не превосходит заданного в ВЛУ лимита, поскольку любое действие в

алгоритме CEGAR (например, *построение АГД*) может потребовать больше времени, чем ВЛУ. В случае нарушения ВЛУ ППУ будет отключено на шаге *отключение ППУ*.

Таким образом, во время работы алгоритма MAB вердикты утверждений изменяются следующим образом. На шаге *создание утверждений* каждое утверждение получает вердикт *Checking*. В случае нахождения трассы ошибки для какого-либо утверждения, его вердикт изменяется на *Unsafe*. В случае нарушения ВЛУ, вердикт соответствующего ему утверждения изменится на *Unknown*. Если алгоритм завершается на шаге *Safe*, то все утверждения, имеющие на тот момент вердикт *Checking*, получают вердикт *Safe*.

После получения вердикта *Unsafe* или *Unknown* MAB должен остановить проверку относительно соответствующих утверждений, для чего и служит шаг *отключение ППУ*. На данном шаге выполняются следующие действия:

1. Прекращается проверка соответствующей ППУ метки ошибки, т. е. производится частичное изменение конфигурации. После этого алгоритм будет продолжен, но при этом отключенное утверждения будет игнорироваться, что позволит получить результат для остальных утверждений.

2. Удаляются соответствующие ППУ верификационные факты. Поскольку для проверки всех утверждений используется одна абстракция, то данная операция таким образом ее скорректирует, чтобы предотвратить экспоненциальный рост числа состояний в АГД, вызванный требованием, которое больше не проверяется. Данная операция проводится как для вердиктов *Unknown* (т. е. утверждение не может быть проверено), так и для вердиктов *Unsafe*, поскольку в общем случае продолжение верификации после нахождения первого нарушения требования может не завершиться успешно. Таким образом, MAB продолжает верификацию после нахождения нарушения утверждения, но уже без проверки нарушенного утверждения.

3. Обнуляется ППУ. После данного действия алгоритм будет считать, что ни одно из утверждений не проверяется вплоть до следующей *точки смены ППУ*.

Таким образом, предложенный алгоритм многоаспектной верификации способен проверять N утверждений и выдавать N вердиктов для них. Задача переиспользования решается во многом аналогично идеям регрессионной верификации, что позволит не повторять выполнение множества действий, как в методе последовательной верификации. За счет ограничения ресурсов на проверку каждого требования в отдельности и корректировки уровня абстракции должна решиться проблема экспоненциального роста числа состояний. Продолжение верификации после нахождения нарушения требования, но без нарушенного требования позволит получить результат для остальных требований и без необходимости проведения ручной фильтрации (как в методе ОВН).

2.3.4. Полнота и корректность алгоритма

Для предложенного алгоритма MAB справедлива следующая теорема (доказательство приведено в Приложении В):

Теорема 1. При верификации требований, удовлетворяющих ограничениям из п. 2.1.1, полнота и корректность алгоритма MAB не изменяются относительно метода последовательной верификации.

Примечание. При этом на практике возможна потеря результата (т. е. невозможность решить часть задач достижимости с теми же ограничениями на ресурсы, что и в базовом методе). Во-первых, подготовленная задача достижимости на основе объединения моделей требований заведомо более сложная. Во-вторых, предложенная аппроксимация может не всегда работать, при этом учет ресурсов и точности абстракции ведется с точностью до предложенной аппроксимации, следовательно, ограничение по времени для каждого требования также работает не всегда точно, что может привести к выделению на проверку

требования меньшего количества ресурсов, чем в методе последовательной верификации.

2.4. Расширения алгоритма многоаспектной верификации

Алгоритм MAB описан достаточно абстрактно, что, с одной стороны, требует уточнений при его реализации, а с другой стороны, предоставляет достаточно широкие возможности по его расширению. Рассмотрим основные направления подобных расширений.

2.4.1. Типы верификационных фактов

Опыт переиспользования в регрессионной верификации показывает, что различные верификационные факты по разному моделируют уровень абстракции, что непосредственным образом влияет на эффективность их переиспользования (см. п. 1.4.3). В MAB по умолчанию переиспользуются верификационные факты для всех утверждений (например, используется одна абстракция), однако предусмотрен механизм для их очистки, если утверждение не может быть проверифицировано с заданным ограничением на ресурсы или было найдено его нарушение. В алгоритме MAB не оговаривается, какие именно верификационные факты должны удаляться при отключении утверждений, поэтому в общем случае предполагается поддерживать сохранение различных типов верификационных фактов для утверждения. При этом главным требованием при очистке верификационных фактов является сохранение путей выполнения исходной программы. В данном случае при удалении (возможно ошибочном) верификационных фактов, не соответствующих отключаемому утверждению, возможно их дальнейшее получение, что, разумеется, потребует накладных расходов. Например, множество абстрактных состояний наиболее полно описывает абстракцию и MAB использует один и тот же АГД для проверки разных утверждений, однако операции сохранения и в особенности удаления множества

абстрактных состояний неэффективны, поскольку в среднем АГД состоит из миллионов абстрактных состояний. При этом большинство промежуточных верификационных фактов (например, запросов к решателю) в течение всего запуска может храниться в кэше верификатора и переиспользоваться.

2.4.2. Стратегии корректировки уровня абстракции

В идеале на шаге *отключение ППУ* полностью удаляются верификационные факты, относящиеся к отключаемому утверждению, чтобы предотвратить его негативное влияние на проверку остальных утверждений. Однако следует учитывать, что в общем случае переиспользование того или иного верификационного факта может как ускорить верификацию других утверждений (позитивный эффект переиспользования), так и замедлить (негативный эффект переиспользования). Например, при получении вердикта *Unknown* для некоторого утверждения возможна ситуация, при которой полученный АГД или сохраненный в кэше запрос к решателю позволит быстрее доказать корректность других утверждений, и соответствующая корректировка абстракции (которая также требует времени) лишь увеличит общее время верификации. Таким образом, на практике стратегия корректировки уровня абстракции должна определить баланс между позитивным и негативным эффектом переиспользования верификационных фактов и, помимо этого, быть эффективной.

2.4.3. Внутренние лимиты

В общем случае предложенный ранее ВЛУ может быть расширен для задания произвольного внутреннего лимита. Основная идея любого подобного внутреннего лимита состоит в том, чтобы ограничить время выполнения некоторой операции и выполнить предопределенные действия в случае нарушения установленного ограничения. Так, ВЛУ ограничивает общее время проверки утверждения и в случае его нарушения отключает данное утверждение с

вердиктом *Unknown*. Потенциально в MAB можно ограничить время на любую операцию алгоритма (например, шаг *отключение ППУ*), на любой интервал ППУ и др. Так, например, может быть известно, что если первый интервал ППУ занимает времени больше некоторого заданного значения (т. е. сама программа является слишком сложной для верификации), то нет смысла ее продолжать, все утверждения, вероятнее всего, так или иначе превысят ВЛУ; в данном случае вполне логично ограничить время первого интервала заданным значением. Основная задача подобных лимитов состоит в том, чтобы как можно быстрее выявлять утверждения, которые не только не могут быть верифицированы, но и ведут к экспоненциальному росту числа состояний в АГД, что может существенно помешать проверке других утверждений или потребовать значительное количество накладных расходов на выполнение шага *отключение ППУ*.

2.4.4. Точки смены утверждений

В алгоритме MAB предлагается использовать только одну точку смены ППУ, однако в реальности таких точек может быть больше. Очевидно, что на шаге *построение АГД* используется несколько утверждений, поэтому совершенно логично было бы отслеживать смену утверждений именно там. Однако в этом случае задача расстановки точек смены ППУ будет целиком зависеть от абстрактного домена и в общем случае решена быть не может. Например, в анализе явных значений [41] можно для каждого утверждения хранить также и имена переменных, которые были добавлены в точность абстракции в тот момент, когда данное утверждение являлось ППУ, после чего появится возможность «переключаться» на проверку соответствующего утверждения каждый раз, когда данная переменная непосредственно используется при создании нового абстрактного состояния. Однако следует помнить, что большое количество вспомогательных действий (например, *смена ППУ* при построении каждого

нового абстрактного состояния) увеличит накладные расходы, при этом любое решение все равно будет эвристическим, поскольку в реальности несколько утверждений используются одновременно.

2.4.5. Расширение используемой аппроксимации

Предложенная аппроксимация об одном проверяемом утверждении является главной эвристикой в алгоритме MAB. С одной стороны, аппроксимация позволяет точно вычислять время, затраченное на каждое утверждение в отдельности, и предотвращать экспоненциальный рост числа состояний, вызванный определенными утверждениями, путем удаления соответствующих им верификационных фактов. С другой стороны, на практике данная аппроксимация не всегда верна. Для того чтобы расширить MAB до более точной аппроксимации с несколькими одновременно проверяемыми утверждениями требуется следующее:

1. поддержка многих точек смены ППУ (см. п. 2.4.4);
2. способ контроля множества проверяемых утверждений (например, на основе алгоритмов вытеснения кэш памяти);
3. способ разделения ресурсов при использовании множества проверяемых утверждений (например, разделять затраченные ресурсы поровну между всеми проверяемыми утверждениями);
4. способ отслеживания верификационных фактов (например, добавлять полученные верификационные факты ко всем проверяемым утверждениям).

2.4.6. Использование идей алгоритма вне подхода CEGAR

Как уже отмечалось, основные идеи алгоритма MAB могут быть использованы и для любого другого подхода статической верификации. Единственный элемент алгоритма, который полностью зависит от подхода CEGAR, – аппроксимация об одном проверяемом утверждении и связанные с ней

действия. В п. 2.4.5 было показано, что в общем случае возможно обойтись и без данной аппроксимации. При этом большинство идей MAB не потребует изменений:

1. Связь проверяемого утверждения с местом в программе (или в ГПУ), где оно нарушается.
2. Получение отдельно вердикта для каждого проверяемого утверждения.
3. Ограничение ресурсов на проверку каждого утверждения в отдельности и изоляция утверждений, которые ведут к экспоненциальному росту числа состояний при построении абстракции.
4. Продолжение верификации после нахождения нарушения утверждения, но без проверки нарушенного утверждения.
5. Переиспользование верификационных фактов для всех утверждений, которые могут быть верифицированы.
6. Корректировка уровня абстракции для предотвращения экспоненциального роста числа состояний.

В главе 3 будет предложен метод, обобщающий данные идеи MAB за пределы подхода CEGAR.

2.5. Метод условной многоаспектной верификации

В теории алгоритм MAB всегда выдает вердикт *Safe*, *Unsafe* или *Unknown* для каждого проверяемого утверждения, однако на практике это может быть не так, если инструмент, реализующий алгоритм, не был завершен успешно. Например, только одно утверждение не может быть проверифицировано и приводит к ошибке в статическом верификаторе или к исчерпанию выделенной памяти. Кроме того, если не удалось достаточно рано установить, что проверка некоторого утверждения ведет к экспоненциальному росту числа состояний, то дальнейшая верификация может также не принести результата. В данном случае от

алгоритма MAB ожидаются вердикты для остальных утверждений, однако на практике они могут быть потеряны.

Для решения данной проблемы алгоритм MAB был расширен на основе условной проверки моделей [58]. Основная идея состоит в том, что по тем или иным причинам получить результат для всех утверждений за один запуск верификатора (или алгоритма верификации) может не получиться, поэтому в этом случае необходимо выдать условие, описывающее, какие утверждение не могут быть верифицированы точно, и передать его новому алгоритму MAB, в котором данных утверждений не будет. Таким образом, алгоритм MAB будет перезапускаться, пока все утверждения не получат вердикт. Данное расширение было названо методом **условной многоаспектной верификации** (**УMAB**).

При этом перезапуск можно осуществлять как внутри верификатора (например, перехватывая исключительные ситуации), так и извне.

2.5.1. Внешняя условная многоаспектная верификация

При внешней УMAB предлагается сохранять промежуточный результат в файл, обрабатывать его и запускать требуемое число раз алгоритм MAB. Схема внешнего УMAB представлена на рис. 2.4.

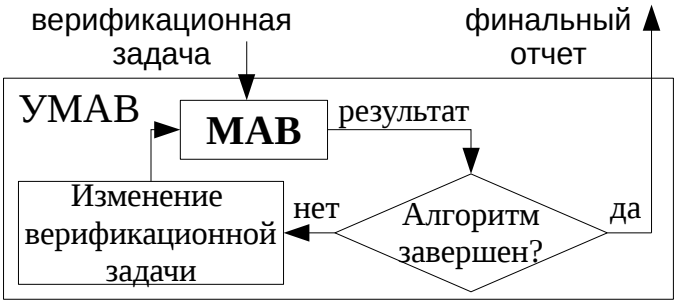

Рис. 2.4. Схема метода УMAB.

Для сохранения промежуточного результата, который будет передаваться между запусками MAB, был предложен общий формат результата УMAB. Для каждого утверждения хранятся следующие параметры:

- идентификатор утверждения;
- вердикт утверждения;
- затраченное процессорное время;
- идентификатор трассы ошибки (для утверждений с вердиктом *Unsafe*);
- причина вердикта *Unknown* (для утверждений с вердиктом *Unknown*).

Помимо этого, предлагается хранить идентификатор ППУ.

Результат в предложенном формате может быть сохранен в указанный файл. При этом на шагах *создание утверждений*, *смена ППУ* и *отключение ППУ*, а также при завершении верификации информация в данном файле обновляется. Таким образом, в случае непредвиденного завершения работы инструмента в указанном файле будет содержаться вся актуальная информация о ходе верификации.

Назовем каждый запуск алгоритма МАВ **итерацией** УМАВ. После каждой итерации метод УМАВ анализирует результат, переданный в общем формате УМАВ, на основе следующего алгоритма завершения УМАВ:

1. Если результата нет, то считается, что произошла глобальная ошибка (например, синтаксическая ошибка в коде программы, неверная конфигурация инструмента), все утверждения получают вердикт *Unknown* и верификация завершается, поскольку не был выполнен уже первый шаг алгоритма *создание утверждений*.
2. Если число утверждений с вердиктом *Checking* равно 0, то верификация завершается – все утверждения уже получили один из вердиктов *Safe*, *Unsafe* или *Unknown*.
3. Если хотя бы одно утверждение имеет вердикт *Unsafe* или *Unknown*, то результат для них запоминается, они полностью удаляются из задачи достижимости и запускается следующая итерация.

4. Если все утверждения имеют вердикт *Checking* и ППУ не задан (т. е. верификация не достигла первой точки смены ППУ), то считается, что имеет место глобальная проблема (например, падение верификатора в начале построения абстракции), все утверждения получают вердикт *Unknown* и верификация завершается.

5. Если все утверждения имеют вердикт *Checking* и ППУ задан, то именно ППУ считается «виновником» некорректного завершения работы, поэтому данное утверждение получает вердикт *Unknown* и удаляется из задачи достижимости, после чего запускается следующая итерация.

После завершения последней итерации УМАВ объединяет промежуточный результат со всех итераций и формирует финальный отчет, в котором для каждого требования представлен вердикт, трассы ошибок для всех вердиктов *Unsafe* и для каждого вердикта *Unknown* указана причина, по которой соответствующее ему утверждение не было верифицировано (например, из-за нарушения ВЛУ или исчерпания всей доступной оперативной памяти для итерации).

Преимуществами внешнего перезапуска являются:

- устойчивость к некорректной работе верификатора;
- возможность корректировать задачу достижимости между запусками, полностью удаляя из кода проверки уже получивших вердикт утверждений;
- полная очистка всех верификационных фактов (включая АГД и кэши верификатора) между запусками;
- возможность задания относительно малого ограничения по времени на итерацию по сравнению с ограничением на верификацию всех требований.

Недостатками внешнего перезапуска являются:

- невозможность переиспользования промежуточных результатов между итерациями (например, для каждой итерации требуется производить разбор

практически одной и той же программы) и верификационных фактов от все еще проверяемых утверждений (например, кэши верификатора);

- дополнительные расходы на УМАВ (переподготовка задач достижимости, проверка завершения УМАВ);
- дополнительные расходы на относительно большое число операций записи в файл внутри алгоритма МАВ;
- неизвестное заранее число запусков верификатора, в худшем случае равное числу проверяемых утверждений.

2.5.2. Внутренняя условная многоаспектная верификация

В теории все наиболее распространенные случаи некорректного завершения работы верификатора можно перехватывать внутри верификатора. Так, например, если верификатор не может построить абстракцию программы для выделенного набора утверждений или размер построенного АГД превышает заданное ограничение, то можно полностью очистить АГД, определить «виновные» утверждения, отключить их проверку и начать алгоритм МАВ заново. На данных идеях основана внутренняя условная многоаспектная верификация.

Внутренняя УМАВ использует ту же схему, что и внешний УМАВ (рис. 2.4), однако все действия осуществляются внутри статического верификатора. При этом под изменением задачи достижимости будет подразумеваться удаление соответствующих им утверждений до начала построения нового АГД. Главным преимуществом внутреннего перезапуска является возможность переиспользовать промежуточные результаты (ГПУ и кэши верификатора) между всеми итерациями. При этом сохраняется возможность удалять верификационные факты, относящиеся к отключаемым утверждениям (точность абстракции, АГД). Помимо этого, внутри верификатора можно гораздо эффективней реализовать проверку завершимости алгоритма и не требуется все промежуточные результаты

записывать во внешний файл. Поэтому внутренний перезапуск по многим параметрам является более оптимизированным, нежели внешний.

Однако у данного метода есть и свои недостатки. Во-первых, внутренний перезапуск никак не сможет обработать различные ошибки в самом инструменте, которые всегда присутствуют на практике. Во-вторых, нет возможности после каждой итерации удалять из задачи достижимости дополнительные проверки, соответствующие отключаемым утверждениям, т. е. упрощать задачу, как во внешнем перезапуске (что, в свою очередь, требует определенных затрат ресурсов). В-третьих, для большого числа итераций кэши верификатора с верификационными фактами могут переполняться, в результате чего эффективность их переиспользования будет сокращаться. Поэтому во многих случаях внешний перезапуск позволит получить более точный результат.

Таким образом, метод УМАВ способен найти вердикты для N утверждений не более чем за N итераций в условиях ограниченных ресурсов и возможного некорректного завершения работы статического верификатора. При этом внешний и внутренний перезапуски алгоритма обладают своими достоинствами и недостатками. Стоит отметить, что поскольку последующие итерации УМАВ предназначены только для уточнения результата (т. е. на первой итерации теряется результат для части требований), а задачи достижимости между итерациями только упрощаются удалением дополнительных проверок (во внешней УМАВ), соответствующих отключенным требованиям, то для метода УМАВ также справедлива теорема 1.

2.6. Метод условной многоаспектной верификации с обнаружением всех однотипных нарушений

Теперь перейдем к задаче доказательства корректности всех проверяемых утверждений или нахождению всех их нарушений. Поскольку метод УМАВ способен находить одно нарушение для каждого из проверяемых утверждений или

доказывать их корректность, а метод ОВН нацелен на нахождение всех нарушений одного утверждения, то для решения данной задачи необходимо объединить данные методы.

Для этого в алгоритме МАВ требуется продолжать верификацию после нахождения нарушения утверждения, не отключая данное утверждение (рис. 2.5). Кроме того, аналогично ОВН в данном месте требуется внутренняя фильтрация трасс ошибок. Прошедшие внутреннюю фильтрацию трассы ошибок сохраняются в атрибутах соответствующего утверждения (для использования во внутренней фильтрации других трасс) и сразу после этого записываются в файл (поскольку верификация может завершиться некорректно), при этом вердикт утверждения меняется на *Unsafe*. При нарушении ВЛУ утверждение отключается, а его вердикт меняется на *Unsafe-incomplete*, если ранее он был *Unsafe*, и на *Unknown* в противном случае (рис. 2.5).

В алгоритме завершения УМАВ требуется изменить шаг 3 следующим образом. Если имеются вердикты *Unsafe* и *Checking*, то все вердикты *Unsafe* заменяются на *Unsafe-incomplete* (алгоритм не был завершен корректно, не все трассы ошибок могли быть найдены). Если хотя бы одно утверждение имеет вердикт *Unsafe*, *Unsafe-incomplete* или *Unknown*, производится процедура внешней фильтрации для найденных трасс ошибок, отфильтрованные трассы запоминаются, соответствующие утверждения полностью удаляются из задачи достижимости и запускается следующая итерация. Финальный отчет будет содержать вердикт для каждого требования, а также трассы ошибок для утверждений, имеющих вердикты *Unsafe* и *Unsafe-incomplete*.

После завершения метода УМАВ при необходимости производится полуавтоматическая фильтрация трасс ошибок аналогично методу ОВН (см. п. 2.2.3).

Таким образом, метод условной многоаспектной верификации с

обнаружением всех однотипных нарушений (УМАВ с ОВН) способен гарантировать, что были найдены для вердиктов *Unsafe* все нарушения данного утверждения, а для вердиктов *Unsafe-incomplete* – некоторые нарушения утверждения, но, возможно, не все. Данный метод объединяет преимущества методов УМАВ и ОВН, при этом ему также присущи и недостатки метода ОВН.

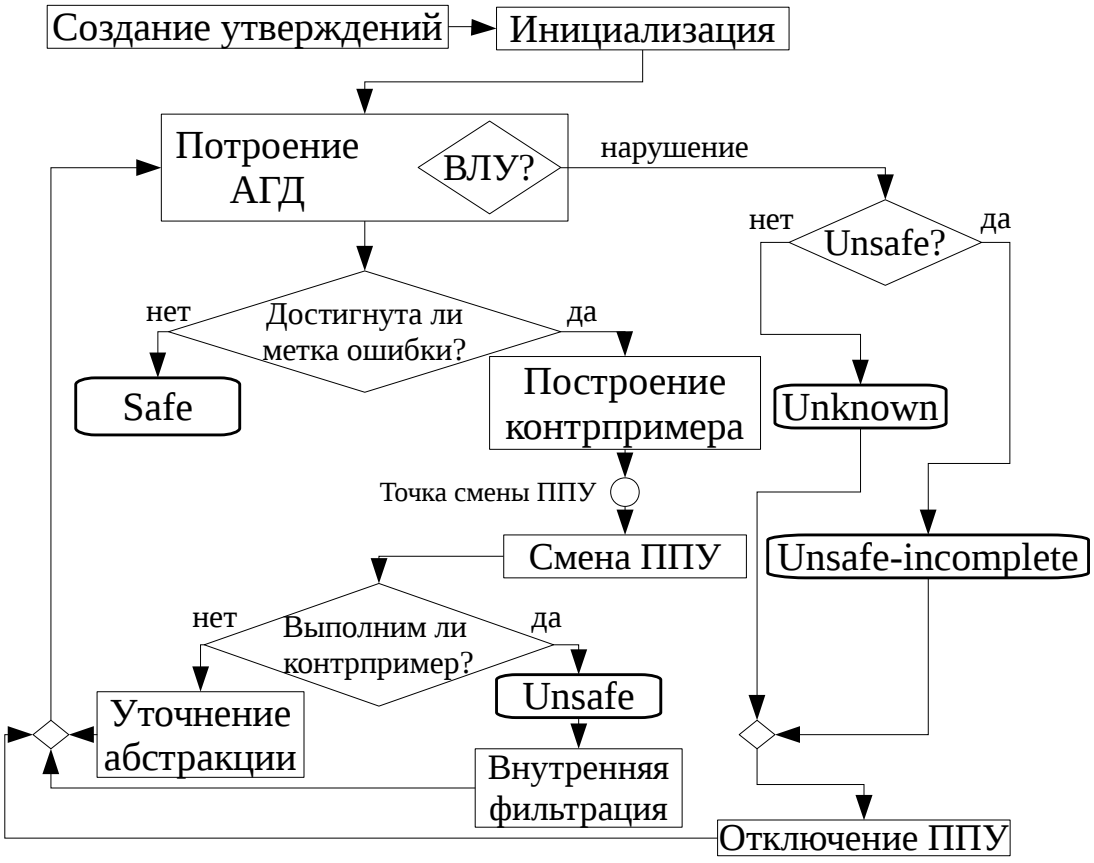

Рис. 2.5. Схема метода УМАВ с ОВН.

2.7. Выводы

В данной главе были предложены методы обнаружения всех однотипных нарушений и условной многоаспектной верификации, которые отвечают поставленным в главе 1 требованиям. Метод обнаружения всех однотипных нарушений предназначен для нахождения всех нарушений проверяемого требования, однако для этого ему необходимо больше вычислительных ресурсов и проведение полуавтоматической фильтрации найденных трасс ошибок. Метод

условной многоаспектной верификации расширяет подход CEGAR для проверки композиции требований, модели которых удовлетворяют сформулированным в п. 2.1.1 ограничениям, с целью повышения производительности верификации. Для метода условной многоаспектной верификации было доказано, что он не изменяет полноту и корректность метода последовательной верификации. Предложенные методы можно использовать совместно для нахождения всех нарушений композиции проверяемых требований.

В следующей главе метод условной многоаспектной верификации будет расширен за пределы подхода CEGAR на основе передачи верификатору моделей требований независимо от исходного кода программы.

Глава 3. Методы декомпозиции автоматной спецификации

Предложенный в предыдущей главе метод условной многоаспектной верификации имеет две проблемы, которые могут существенно ограничить его область применимости на практике.

Во-первых, метод УМАВ решает потенциально более сложные задачи достижимости за счет добавления в них большего числа дополнительных проверок во время инструментирования исходного кода (см. п. 2.1), и чем больше проверяемых требований, тем больше будут усложняться подготавливаемые задачи. Кроме того, при инструментировании проверки требований находятся в исходном коде и нет возможности их удалить во время верификации, например, если было найдено нарушение требования и больше его не нужно проверять. Данная проблема не была столь актуальной при проверке одного требования, однако с ростом числа требований может привести к большим потерям результата.

Во-вторых, метод УМАВ основан на эвристике, которая имеет смысл только в рамках подхода CEGAR и в общем случае, как и любая другая эвристика, может не работать (т. е. приводить к потере результата).

В данной главе предлагается новый метод верификации программного обеспечения, нацеленный на решение данных проблем.

3.1. Метод автоматных спецификаций

Можно заметить, что первая проблема – усложнение задач достижимости – не связана с методом УМАВ, а относится к инструментированию исходного кода, поэтому для ее решения необходим новый метод формализации проверяемых требований. Логичным решением данной проблемы является передача моделей требований верификатору независимо от исходного кода, однако на данный момент подобные методы используются достаточно редко (см. п. 1.2), во многом из-за того, что по возможностям формализации требований они серьезно уступают

методам инструментирования, в которых модель может быть представлена на языке самой программы. Поэтому предлагается новый метод формализации требований, который позволит представлять требование в виде конечного автомата и передавать его верификатору независимо от исходного кода программы, при этом обладающий теми же возможностями по формализации требований, что и метод инструментирования.

3.1.1. Наблюдательные автоматы

Все специфические требования к программе могут быть сведены к формулам темпоральной логики [76], которые задают множество корректных выполнений программы. Один из возможных способов задания подобных формул темпоральной логики опирается на наблюдательные автоматы [49] (от англ. observe automaton). Наблюдательный автомат – это недетерминированный конечный автомат, который состоит из множества состояний $Q=\{q_1, \ldots, q_n\}$, входного алфавита A, содержащего ребра из ГПУ (т. е. операторы программы), и функции перехода между состояниями $f(q_i, a) = q_j$, где q_i и q_j являются состояниями автомата, a – оператор программы. При этом существует специальное ошибочное состояние, переход в которое и будет обозначать нарушение проверяемого требования. Отличительной характеристикой данного типа автоматов является то, что они не могут модифицировать исходную программу (например, выполнять присваивание в переменные программы), поэтому их использование не влияет на полноту и корректность верификации [49]. Таким образом, требование к программе может быть формализовано с помощью наблюдательного автомата, что ранее неоднократно применялось на практике [49, 50, 77].

При построении абстракции программы наблюдательный автомат принимает на вход последовательность операторов программы от алгоритма верификации,

при этом если автомат переходит в ошибочное состояние, то считается, что найдено нарушение проверяемого требования. Если в процессе построения абстракции хотя бы один из переходов автомата использовался, то соответствующее автомату требование считается **релевантным** для проверяемой программы, поскольку потенциально оно может нарушиться. В противном случае требование является нерелевантным и даже потенциально не может быть нарушено (например, программа не содержит присутствующих в автомате вызовов функций). Однако стоит заметить, что задача определения релевантности проверяемого автомата для произвольной программы, как и задача статической верификации, в общем случае неразрешима.

Очевидно, что не все требования могут быть представлены с помощью наблюдательных автоматов. Например, если требуется проверить, что количество вызовов функций выделения памяти совпадает с количеством вызовов функций освобождения памяти (что в инструментировании легко моделируется с помощью внутренней переменной-счетчика), наблюдательный автомат не справится с задачей. Поэтому для начала необходимо расширить синтаксис и семантику наблюдательных автоматов и тем самым приблизить данный метод по возможностям формализации требований к инструментированию.

3.1.2. Описание языка автоматных спецификаций

Основная задача, которая преследовалась при расширении наблюдательных автоматов, – добавить поддержку внутренних переменных и расширить функцию перехода между состояниями для поддержки произвольных конструкции языка программирования C (в том объеме, в котором они поддерживаются статическим верификатором). Для этого был предложен метод **автоматных спецификаций**, или **АС** [35], (англ. specification automaton), нацеленный на верификацию требований, передаваемых верификатору с помощью конечных автоматов отдельно

от исходного кода. При этом требование формализуется в виде одного или нескольких расширенных наблюдательных автоматов.

Грамматика предложенного языка описания автоматных спецификаций, представляющего собой расширение языка описания наблюдательных автоматов, представлена на рисунке 3.1.

```
automata_specification ::= observer_automaton+
observer_automaton ::= header init_state states
header ::= OBSERVER AUTOMATON automaton_name
automaton_name ::= identifier
init_state ::= INITIAL STATE state_name;
state_name ::= identifier
states ::= state+
state ::= STATE deterministic state_name:
    transitions END AUTOMATON
deterministic ::= USEFIRST|USEALL
transitions ::= transition+
transition ::= transition_condition ->
    if_section|if_else_section;
transition_condition ::= (MATCH statement)|TRUE
statement ::= ((CALL|RETURN)? {identifier})|ENTRY|EXIT
if_section ::= assume? encode? goto
if_else_section ::= SPLIT assume? encode? goto
    NEGATION encode? goto
assume ::= ASSUME {C_expression}
encode ::= ENCODE {C_operator}
goto ::= (GOTO state_name) | ERROR("identifier")
```

Рис. 3.1. Грамматика языка автоматных спецификаций.

Согласно предложенной грамматике для каждого автомата задается уникальное имя, соответствующее идентификатору требования, и множество состояний, одно из которых является начальным (INITIAL STATE). Помимо этого, есть предопределенное состояния – ERROR, обозначающее нарушение требования. Для каждого состояния можно задать множество переходов (transitions) и определить, является ли оно детерминированным (USEFIRST), т. е. все условия перехода (transition_condition) уникальны, или нет (USEALL). Условие

перехода может совпадать с произвольным оператором программы (именем функции или переменной), при этом имеется возможность параметризовать аргументы функции (CALL) или ее возвращаемое значение (RETURN). Помимо этого, имеется возможность привязывать переход к моменту входа в состояние (ENTRY) или к моменту выхода из него (EXIT). Конструкция ENCODE отвечает за добавление произвольных операторов на языке C (C_operator) в ГПУ во время верификации. Таким образом, можно добавлять и модифицировать в программе внутренние переменные, необходимые для проверки требования, непосредственно при построении АГД. Кроме того, можно наложить дополнительное ограничение на параметры функции, ее возвращаемое значение и введенные с помощью конструкции ENCODE внутренние переменные с помощью конструкции ASSUME, которая содержит произвольное выражение на языке C (C_expression). Конструкция GOTO осуществляет переход в другое состояние, которое может совпадать с текущим, при выполнении условия ASSUME.

Оператор языка C, добавляемый с помощью конструкции ENCODE, может содержать объявление новых (глобальных) переменных, например:

```
ENCODE {int var;};
```

и изменение значений добавленных переменных, например:

```
ENCODE {var++;}.
```

Кроме добавления и изменения значений добавленных переменных запрещается производить какие-либо действия в данной конструкции.

Произвольное выражение языка C, добавляемое с помощью конструкции ASSUME, может содержать условие как с добавленными переменными, так и с параметрами функции и ее возвращаемым значением, кроме того, оно должно содержать только те операции, которые поддерживаются верификатором. При этом доступ к параметрам функции и ее возвращаемому значению осуществляется с помощью специального оператора «$N», где N – порядковый номер параметра или

возвращаемого значения, полученный при задании условия перехода. При этом оператор «$?» используется для задания функции с произвольным числом параметров. Например, следующая конструкция параметризует возвращаемое значение функции с произвольным числом параметров и накладывает ограничение на возвращаемое значение:

```
MATCH RETURN {$1=malloc($?)} -> ASSUME {$1!=0} ...
```

Следующая конструкция накладывает ограничение на количество параметров функции и выполняет переход в том случае, если второй параметр больше первого:

```
MATCH CALL {find_bit($1,$2)} -> ASSUME {$2>$1} ...
```

На рисунке 3.2 представлен пример задания автоматной спецификации для проверки того, что вся выделенная память с помощью функции *malloc* освобождается с помощью функции *free* («memory leak») и перед освобождением корректно выделяется память («double free»). При этом память считается выделенной только в том случае, если функция *malloc* возвращает ненулевой указатель. Для формализации требования вводится переменная-счетчик *allocated*.

```
OBSERVER AUTOMATON memory_alloc
INITIAL STATE Init;
STATE USEALL Init :
  MATCH ENTRY -> ENCODE {int allocated = 0;} GOTO Init;
  MATCH RETURN {$1 = malloc($?)} ->
    ASSUME {$1 != 0} ENCODE {allocated++;} GOTO Init;
  MATCH RETURN {$1 = malloc($?)} ->
    ASSUME {$1 == 0} GOTO Init;
  MATCH CALL {free($1)} -> ASSUME {allocated >  0}
    ENCODE {allocated--;} GOTO Init;
  MATCH CALL {free($1)} -> ASSUME {allocated <= 0}
    ERROR("double free");
  MATCH EXIT -> ASSUME {allocated != 0}
    ERROR("memory leak");
END AUTOMATON
```

Рис. 3.2. Пример задания автоматной спецификации.

3.1.3. Сопоставление автоматных спецификаций с инструментированием

Для предложенного метода АС справедливы следующие утверждения (доказательства приведены в Приложении В):

Утверждение 2. Для требований, удовлетворяющих ограничениям из п. 2.1.1, метод АС полностью совпадает по возможностям их формализации с методом инструментирования.

Утверждение 3. При верификации требований, удовлетворяющих ограничениям из п. 2.1.1, полнота и корректность метода АС не изменяются относительно метода последовательной верификации верификации на основе инструментирования.

Примечание. Несмотря на то что подготавливаемые методом АС задачи достижимости заведомо проще, в общем случае на практике процесс их решения может различаться. Например, в некоторых частных случаях абстракцию строить проще для более сложной задачи, следовательно, такая задача будет решена быстрее методом инструментирования.

Таким образом, предложенный метод АС полностью совпадает по возможностям формализации требований с инструментированием для требований, удовлетворяющих ограничениям из п. 2.1.1, при этом подготавливаемые им задачи достижимости получаются заведомо проще.

3.1.4. Автоматные спецификации в адаптивном статическом анализе

Однако предложенный метод АС не выполняет непосредственную проверку формализованных с помощью автоматов требований, а описывает способ их формализации. Поэтому для использования метода АС на практике необходимо интегрировать его в существующие подходы статической верификации.

Заметим, что предложенные автоматы в методе АС удовлетворяют условиям адаптивного статического анализа. Абстрактным доменом является множество

всех выполнений программы, точность абстракции определяет, используются ли при построении абстракции добавленные в автомате конструкции, отношение переходов совпадает с функцией переходов в автомате, оператор слияния никогда не объединяет состояния, а оператор останова работает аналогично остальным CPA. Таким образом, возможно объединить автоматную спецификацию в композитном CPA в качестве одного из CPA. При этом в построении АГД будут использованы встраиваемые конструкции языка C, присутствующие в конструкциях ASSUME и ENCODE автомата. При переходе в состояние ERROR, автомат отправляет остальным CPA сигнал о том, что найден контрпример.

Помимо автоматных спецификаций в композитном CPA могут присутствовать произвольные типы анализа, в том числе и реализующие подход CEGAR. Кроме того, возможно использование нескольких CPA, каждый из которых представляет одно требование (метод пакетной верификации с автоматной спецификацией). При этом не требуется дополнительное объединение автоматов, поскольку, даже если в них используются одинаковые имена функций в переходах между состояниями, подобные конфликты разрешаются автоматически на основе адаптивного статического анализа во время верификации.

Помимо этого, возможно использование идей метода ОВН и в методе АС. Для этого необходимо не отключать соответствующий CPA после нахождения нарушения требования, а по завершении верификации проводить внешнюю и полуавтоматическую фильтрацию трасс ошибок на основе перехватываемых на переходах автомата имен (например, вызовов функций).

Таким образом, метод АС представляет собой более универсальный способ подготовки задач достижимости, в котором исходный код верифицируемой программы не изменяется, а требование передается верификатору отдельно в виде автомата, при этом возможности по формализации требований не уступают инструментированию. Главное преимущество предложенного метода состоит в

том, что не производится усложнение исходного кода программы, что особенно актуально при проверке многих требований.

3.2. Метод декомпозиции автоматной спецификации

Поставленную в данной работе цель можно переформулировать следующим образом – дана спецификация, состоящая из N требований, и необходимо для нее найти такое разбиение на M групп требований, при котором все требования, приводящие к экспоненциальному росту числа состояний, проверяются отдельно. В данной формулировке в методах пакетной верификации и алгоритме МАВ все требования помещаются в одну группу (т. е. $M=1$), а в методе последовательной верификации в каждую группу помещается только одно требование (т. е. $M=N$). Однако, как уже говорилось ранее, не все требования могут быть верифицированы совместно, в частности, поэтому требуется расширение алгоритма МАВ до УМАВ, в котором итеративно удаляются требования, приводящие к экспоненциальному росту числа состояний.

Декомпозиция автоматной спецификации, или **ДАС** [35], (англ. on-the-fly specification decomposition) нацелена на автоматическое разбиение спецификации на группы требований для совместной верификации более эффективным образом. В идеале ДАС разделяет пары требований, которые при совместной верификации ведут к экспоненциальному росту числа состояний в АГД и тем самым к существенному снижению эффективности верификации всех требований, и объединяет в одну группу требования, совместная верификация которых более эффективна. ДАС предлагается на основе идей адаптивного статического анализа, в котором каждое требование формализовано с помощью одного или нескольких автоматов в одном CPA. Все идеи метода УМАВ, которые не относятся к подходу CEGAR (см. п. 2.4.6), также используются и в методе ДАС.

3.2.1. Общий алгоритм декомпозиции автоматной спецификации

Алгоритм ДАС принимает на вход всю спецификацию, состоящую из набора требований, а на выходе выдает вердикт для каждого проверяемого требования (при этом вердиктам *Unsafe* соответствует трасса ошибки, а вердиктам *Unknown* – указание причины, по которой требование не может быть верифицировано). Алгоритм основан на построении так называемых **разбиений** (англ. partition) или групп требований с помощью **стратегий разбиения** (англ. partitioning strategy). Разбиение представляет собой подмножество всех проверяемых требований и заданные ограничения на ресурсы (например, ограничение на использование процессорного времени при верификации разбиения). Стратегия разбиения по множеству предыдущих разбиений строит новое множество разбиений, т. е. выполняет декомпозицию. Алгоритм ДАС состоит из следующих шагов:

1. С помощью стратегии разбиения строится новое множество разбиений на основе старого и списка проверяемых требований. Изначально список проверяемых требований включает в себя все требования.

2. Для каждого разбиения запускается алгоритм верификации, основанный на адаптивном статическом анализе, которому будут переданы CPA, соответствующие требованиям из разбиения. Ограничения по ресурсам на данный алгоритм верификации указаны в разбиении.

3. Если для разбиения удалось доказать корректность всех проверяемых требований или найти нарушение требования, то данные требования получают финальный вердикт и удаляются из списка проверяемых требований. Если же верификация разбиения исчерпала выделенные ресурсы и разбиение состояло всего из одного требования, то по аналогии с последовательной верификацией ему присваивается вердикт *Unknown* и оно удаляется из списка проверяемых. В противном случае (т. е. если верификация разбиения исчерпала выделенные ресурсы и число требований в разбиении больше одного) требования разбиения

остаются в списке проверяемых.

4. Если после завершения верификации всех построенных разбиений список проверяемых требований не пуст, то необходимо перейти на шаг 1.

Таким образом, итерационный метод ДАС является более общим вариантом метода УМАВ. Соответственно верификация каждого нового множества разбиений может представлять собой новый запуск статического верификатора или же весь алгоритм может быть реализован в одном запуске верификатора. Данные методы имеют свои преимущества и недостатки, схожие с рассмотренными в п. 2.5.

Основной алгоритм верификации, используемый на шаге 2, практически полностью совпадает с методом АС, рассмотренным в п. 3.1. Данный алгоритм работает со множеством CPA, одна часть из которых отвечает за верификацию (например, построение предикатной абстракции), а другая часть реализует автоматы, отвечающие проверяемым требованиям. Главным отличием является то, что при нахождении нарушения требования соответствующий ему CPA удаляется из композитного CPA, а найденная трасса ошибки выводится в файл на случай некорректного завершения всего алгоритма, после чего верификация продолжается.

Алгоритм продолжается до тех пор, пока не будет получено разбиение, в котором удастся доказать корректность всех требований или найти их нарушения, или же разбиение с одним требованием исчерпает выделенные ресурсы, что эквивалентно получению вердикта *Unknown* в методе АС при проверке одного требования. Кроме того, возможно наложить общее ограничение на все итерации, при исчерпании которого все еще проверяемые требования получают вердикт *Unknown.*

Ядром всего алгоритма ДАС является стратегия разбиения, которая строит множества разбиений и от которой напрямую зависит эффективность всего

алгоритма. Помимо этого, стратегия разбиения должна учитывать, что на решение всей задачи выделены определенные ресурсы, которые необходимо некоторым образом распределить между всеми разбиениями. В общем случае стратегия разбиения является произвольной, единственное требование к ней заключается в том, чтобы максимальное число требований среди всех построенных разбиений уменьшалось при построении нового множества разбиений, чтобы алгоритм ДАС сходился. В данном месте могут быть реализованы различные эвристики (одна из которых и была предложена в методе УМАВ).

3.2.2. Полнота и корректность метода

Для предложенного метода ДАС справедлива следующая теорема (доказательство приведено в Приложении В):

Теорема 2. При верификации требований, удовлетворяющих ограничениям из п. 2.1.1, полнота и корректность метода ДАС не изменяются относительно метода АС.

Примечание 1. Поскольку метод ДАС в общем случае на практике требует больше накладных расходов на поддержание многих СРА, то возможна потеря результата относительно менее ресурсоемкого метода АС.

Примечание 2. Данная теорема верна для любой стратегии разбиения, в частности, можно использовать стратегию, выполняющую случайное разбиение. Однако результат сильно зависит от грамотного распределения ресурсов между всеми разбиениями, что в общем случае может приводить к потерям результата. Кроме того, при использовании эвристических стратегий разбиения возможна потеря результата, если эвристика не работает.

Следствие. При верификации требований, удовлетворяющих ограничениям из п. 2.1.1, полнота и корректность метода ДАС не изменяется относительно метода последовательной верификации на основе инструментирования.

Таким образом, метод ДАС позволяет получить такой же результат, что и при верификации требований на основе инструментирования (в частности, с помощью метода УМАВ).

3.3. Стратегии разбиения в методе декомпозиции автоматной спецификации

Стратегия разбиения является основой алгоритма ДАС. Можно предложить различные эвристики для более точного определения того, какое именно из проверяемых требований приводит к экспоненциальному росту числа состояний. При этом следует помнить, что наиболее эффективные эвристики, как правило, могут значительно сузить область применимости алгоритма. Метод ДАС представляет универсальный интерфейс для реализации и сравнения подобных эвристик.

Как уже отмечалось, метод УМАВ является частным случаем метода ДАС, поэтому он потенциально может быть реализован в качестве стратегии разбиения. Данная стратегия разбиения должна отслеживать последнее проверяемое утверждение по проверяемому автомату, суммировать время проверки отдельно для каждого автомата и по исчерпании ограничения на проверку одного автомата создавать новое разбиение без данного автомата. При этом на практике подобная реализация может не быть эффективной из-за дополнительных накладных расходов на проверку многих автоматов по сравнению с методом УМАВ, кроме того, область применимости метода ДАС будет сокращена.

В качестве базовых стратегий разбиения будут предложены те, которые не опираются на эвристики и не сужают область применимости метода. В общем случае задача стратегии разбиения формулируется следующим образом. Пусть спецификация состоит из N требований, реализованных автоматами, на верификацию каждого требования выделено t секунд процессорного времени (т. е. на верификацию всех требований выделено $N*t$ секунд процессорного времени).

Требуется разбить спецификацию на группы требований и распределить выделенные ресурсы на верификацию каждой группы.

3.3.1. Стратегия Совместная проверка

Самая простая из возможных стратегий разбиения заключается в том, чтобы поместить всю спецификацию в одно разбиение и выделить ей все ресурсы. Таким образом, стратегия построит одно множество разбиений, состоящее из одного разбиения, которому будет выделено *N*t* секунд. Если в разбиении будут исчерпаны все ресурсы, то все требования за исключением тех, для которых было найдено нарушение, получат вердикт *Unknown*.

Преимущества:

- стратегия способна продолжать верификацию после нахождения нарушения требования (в противовес методу пакетной верификации);
- полностью переиспользует верификационные факты (АГД, точность абстракции, кэши верификатора) между верификацией всех требований.

Недостатки:

- если хотя бы одно требование по тем или иным причинам не может быть верифицировано (ведет к экспоненциальному росту числа состояний), то не будут верифицированы все требования (негативный эффект переиспользования).

На практике данная стратегия нужна только для моделирования пакетной верификации с возможностью продолжать верификацию без нарушенного требования и не предназначена для реального использования в рамках метода ДАС.

3.3.2. Стратегия Последовательная проверка

Другая элементарная стратегия заключается в том, чтобы поместить каждое проверяемое требование в одно разбиение и выделить всем разбиениям равное

количество ресурсов, т. е. t секунд. Таким образом, каждое разбиение будет представлять собой метод AC с одним CPA, формализующим требование.

Преимущества:

- полная изоляция верификации различных требований (т. е. негативный эффект переиспользования невозможен);
- переиспользуются результаты предварительной подготовки (т. е. ГПУ) и кэши верификатора между всеми разбиениями (в отличие от N запусков метода AC).

Недостатки:

- низкая эффектность из-за того, что большинство верификационных фактов не будет переиспользоваться между разбиениями.

Данная стратегия является модифицированным методом AC, в котором переиспользуются ГПУ и кэши верификатора.

3.3.3. Стратегия Совместно-последовательная проверка

Данная стратегия нацелена на объединение преимуществ двух предыдущих стратегий и состоит из 2 шагов. Вначале создается одно разбиение для всей спецификации с ограничением процессорного времени в $t_1 \sim t$ секунд (аналогично стратегии *Совместная проверка*). В этом разбиении будут полностью переиспользованы верификационные факты между всеми требованиями, что позволит задействовать позитивный эффект переиспользования и решить быстрее часть задач, для которых не наблюдается экспоненциального роста числа состояний при совместной верификации. Ограничение по времени t_1 устанавливает границу для эффекта переиспользования – если она нарушается, то считается, что преобладает негативный эффект переиспользования и совместная проверка требований ведет к экспоненциальному росту числа состояний. В этом случае создается новое множество разбиений для всех требований (за

исключением, может быть, нарушенных), в котором каждое требование помещается в отдельное разбиение с ограничением процессорного времени в *t* секунд (аналогично стратегии *Последовательная проверка*). Таким образом, если быстро не удается проверить все требования вместе, то производится их полная изоляция, т. е. предполагается, что любое может приводить к экспоненциальному росту числа состояний.

Преимущества:

- использование элементов «динамического» переиспользования – в зависимости от результата переиспользуются либо все верификационные факты, либо только кэши верификатора и ГПУ.

Недостатки:

- нет четкого определения требования, ответственного за экспоненциальный рост числа состояний, что ведет к снижению эффективности на обоих шагах (первый тратится впустую, а на втором большинство верификационных фактов не переиспользуется).

3.3.4. Стратегия Релевантность

Данная стратегия основана на следующем утверждении: верификация любого числа нерелевантных требований для поставленной задачи достижимости по сложности совпадает с верификацией любого одного нерелевантного требования.

Докажем данное утверждение. Нерелевантность требования означает, что при верификации программы ни один из переходов автомата не сработает и соответствующий автомату CPA при верификации будет проигнорирован. А поскольку верификация любого числа нерелевантных требований и верификация одного нерелевантного требования методом ДАС различается только количеством CPA, отвечающих требованиям, которые будут игнорироваться, то в обоих случаях

будет получен один и тот же АГД (и пустая точность абстракции, поскольку уточнение абстракции не производится для нерелевантных требований). Таким образом, в обоих случаях решается одна и та же задача и добавление дополнительных «игнорируемых» условий ее не усложнит.

На практике, конечно, могут возникать накладные расходы на поддержание многих CPA, однако это будет проблема реализации, а не метода.

На основании данного утверждения можно заключить, что между нерелевантными требованиями не может возникнуть негативного эффекта переиспользования, в то время как достигает максимума позитивный эффект (полностью переиспользуется АГД). Поэтому нерелевантные требования нужно проверять вместе.

Помимо этого, на практике число нерелевантных требований может быть достаточно большим. Например, многие требования задаются на специфические интерфейсы, которые в большинстве программ не используются.

Поэтому основная идея данной стратегии и состоит в том, чтобы определить все нерелевантные требования и относительно быстро доказать их корректность. Однако задача определения релевантности требования не является разрешимой и для достаточно больших программ сама по себе будет занимать много времени.

Предлагаемая стратегия состоит из трех шагов:

1. Определение релевантности. Вся спецификация помещается в одно разбиение с ограничением по времени в $t_1 < t$ секунд (аналогично стратегии *Совместная проверка*). Данное ограничение не должно быть большим, поскольку основная задача данного шага вспомогательная – определение множества релевантных требований. Если за это время верификация была успешно завершена, то алгоритм ДАС завершается. В противном случае получается множество некоторых (в общем случае не всех) релевантных требований.
2. Поэтапное доказательство корректности для нерелевантных требований. Все

нерелевантные требования, полученные на первом шаге, помещаются в одно разбиение с ограничением по времени в $t_2 > t$ секунд. Данное ограничение уже должно быть относительно большим, чтобы позволить доказать нерелевантность всех проверяемых требований или же найти среди них релевантные, которые не были найдены с малым ограничением по времени на первом шаге. Если верификация завершается неуспешно и были найдены новые релевантные требования, то их список обновляется и шаг перезапускается. Если же все проверяемые требования так и остались нерелевантными и ограничение по времени было исчерпано, то считается, что задача слишком сложная (возможно, неразрешимая), и все требования получают вердикт *Unknown*.

3. Проверка всех релевантных требований. Для каждого релевантного требования, определенного на предыдущих шагах, создается отдельное разбиение и выделяются равные ресурсы (т. е. t секунд). Данный шаг аналогичен стратегии *Последовательная проверка*.

Преимущества:

- стратегия оптимизирует баланс переиспользования (для нерелевантных – полное переиспользование, для релевантных – минимальное).

Недостатки:

- при большом количестве релевантных требований эффективность будет снижаться.

Данная стратегия предлагается в качестве базовой для метода ДАС.

3.4. Выводы

В данной главе был предложен метод формализации требований в виде автоматной спецификации и алгоритм декомпозиции автоматной спецификации, которые рассчитаны на передачу требований верификатору отдельно от исходного кода, что упрощает задачи достижимости и тем самым снижает требуемые на их

верификацию ресурсы. Метод декомпозиции автоматной спецификации обобщает метод условной многоаспектной верификации и предоставляют широкие возможности для создания и сравнения новых алгоритмов верификации композиции требований. Помимо этого, для данных методов было доказано, что они не изменяют полноту и корректность метода последовательной верификации (на основе инструментирования исходного кода).

В следующей главе будет описана реализация всех предложенных методов.

Глава 4. Реализация предложенных методов

Реализация предложенных методов производилась на основе системы верификации LDV Tools [23-26], в которой на тот момент имелось 30 требований на корректное использование программных интерфейсов ядра операционной системы Linux, а их проверка осуществлялась с помощью метода последовательной верификации в статическом верификаторе CPAchecker [29, 30], в котором реализованы подходы адаптивного статического анализа и CEGAR.

4.1. Расширения системы верификации

В базовом варианте система LDV Tools готовит задачу достижимости на основе модуля ядра и требования, формализованного с помощью инструментирования, подает ее статическому верификатору и предоставляет результат верификации человеку для анализа (рис. 1.5). Расширения системы верификации нацелены на то, чтобы появилась возможность подготавливать задачу достижимости на основе композиции требований, формализованных как с помощью методов инструментирования, так и с помощью автоматных спецификаций, и поддерживать методы обнаружения всех однотипных нарушений, условной многоаспектной верификации, автоматной спецификации и декомпозиции автоматной спецификации. Предложенные расширения не опираются на специфику ядра Linux (т. е. не затрагивают компонент подготовки исходного кода) и могут быть применены для верификации произвольных программ.

4.1.1. Новая архитектура системы верификации

На основе поставленных требований была предложена новая архитектура системы верификации, которая представлена на рисунке 4.1. Предполагается, что проверяются только те требования, которые удовлетворяют ограничениям из

п. 2.1.1.

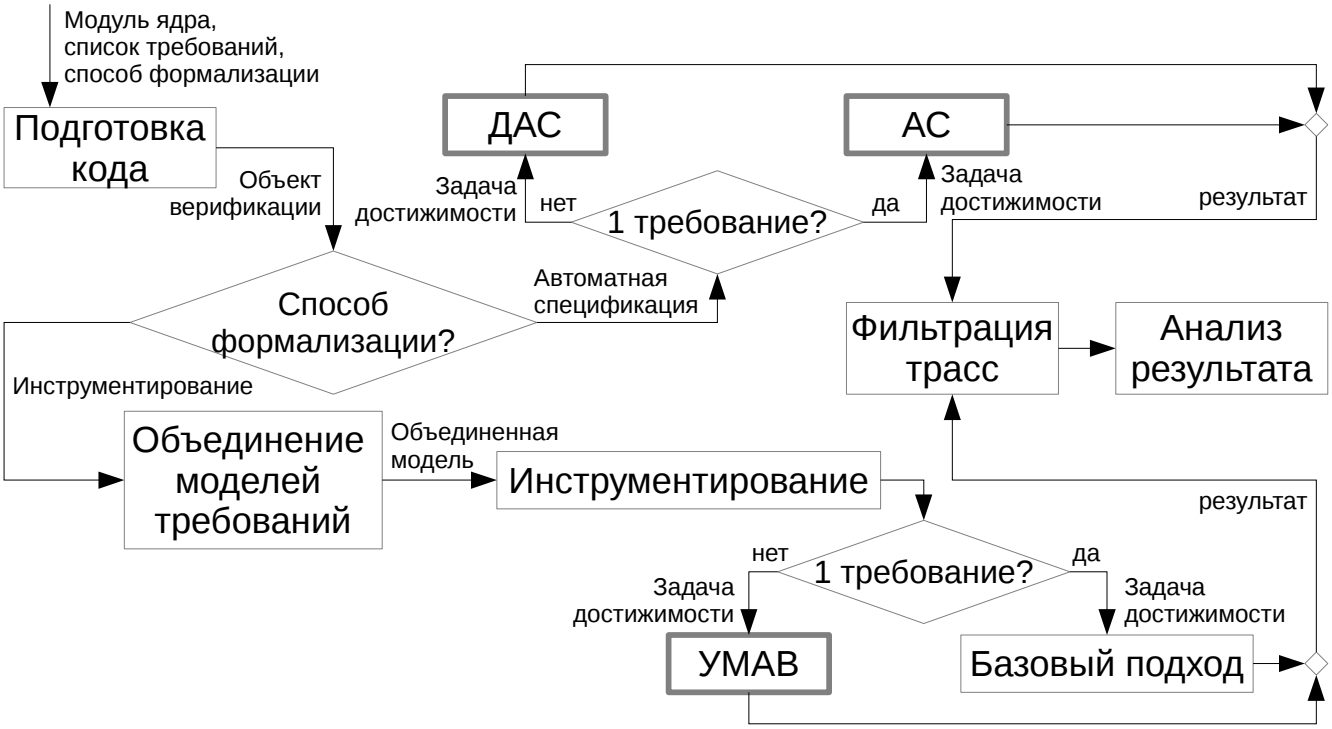

Рис. 4.1. Новая архитектура системы LDV Tools.

На вход системе верификации теперь вместе с модулем ядра подается список требований, а также способ формализации требований (инструментирование или автоматная спецификация). Компонент подготовки объектов верификации (т. е. целевого кода) остается без изменений. Согласно ограничению из п. 2.1.1 для всех требований должен быть один и тот же способ подготовки целевого кода, иначе верификация их композиции невозможна. Для метода инструментирования производится объединение моделей требований, после чего выполняется процедура инструментирования исходного кода на основе объединенной модели требований.

Для задач достижимости, полученных методом инструментирования, используется метод УМАВ или базовый подход верификации (например, CEGAR), если число проверяемых требований равно 1. Метод УМАВ поддерживает как внутренний перезапуск, так и внешний, производимый в этом случае системой

верификации. Для автоматной спецификации применяются методы ДАС или АС (если проверяется одно требование). Сами алгоритмы верификации (базовый подход, УМАВ, АС, ДАС) реализуются статическим верификатором, в общем случае произвольным (при этом от верификатора требуется поддержка соответствующих алгоритмов). Для метода ОВН полученные от верификатора трассы ошибок проходят дополнительную внешнюю фильтрацию перед представлением результата для анализа человеку в web-интерфейсе системы LDV Tools, в котором далее возможна полуавтоматическая фильтрация трасс ошибок.

Таким образом, предложенные расширения архитектуры системы верификации позволяют использовать как базовый подход верификации с инструментированием исходного кода, так и все предложенные в данной работе методы.

4.1.2. Формализация проверяемых требований

Система LDV Tools на момент выполнения данной работы содержала 30 требований (приведены в Приложении А) на корректное использование сердцевины ядра операционной системы Linux. Однако предложенные методы поддерживают только ограниченные возможности по их формализации, поэтому все требования были проанализированы с целью проверки соответствия ограничениям из п. 2.1.1. После этого требования были переписаны таким образом, что все найденные нарушения данных ограничений были устранены. Как оказалось на практике, введенные в п. 2.1.1 ограничения позволяют формализовывать требования на корректное использование интерфейсов сердцевины ядра Linux.

Помимо этого, для поддержки методов, основанных на автоматных спецификациях, требования системы LDV Tools были формализованы в виде автоматов. Для каждого требования две модели (заданные на аспектно-

ориентированном расширении языка C [47] и в виде автомата) были тщательно проанализированы для подтверждения того факта, что они являются эквивалентными, кроме того, были написаны и включены в состав системы LDV Tools соответствующие тесты (несколько сотен).

Таким образом, для указанных 30 требований возможно применение предложенных в данной работе методов. Для поддержки произвольного нового требования необходимо:

- удовлетворение требованиям из п. 2.1.1 (для всех предложенных методов);
- модель на аспектно-ориентированном расширении языка C [47] (методы ОВН и УМАВ);
- автоматная спецификация (методы АС и ДАС).

4.1.3. Объединение моделей требований

Процедура объединения производится только для инструментирования и нацелена на подготовку задачи достижимости для метода УМАВ. Все объединяемые модели требований должны удовлетворять условиям из п. 2.1.1. В соответствии с описанным в п. 2.1.2 методом на основе нескольких моделей требований, формализованных на аспектно-ориентированном расширении языка C [47], создается объединенная модель, в которой нарушение каждого требования помечается отдельной меткой ошибки. Таким образом, нарушению каждого требования соответствует уникальная метка, при этом в самом названии метки присутствует идентификатор требования (*X*).

Далее необходимо передать верификатору информацию о том, что соответствующие метки являются метками ошибки. В некоторых статических верификаторах (например, для BLAST [27, 28] или CBMC[53]) для этого используются опции верификатора. Для статического верификатора CPAchecker подобная информация передается в виде простейшего наблюдательного

автомата [49]. Для базового подхода верификации (CEGAR) в системе LDV Tools подобный автомат включает в себя одно состояние, из которого имеется один переход в ошибочное состояние при достижении заданной метки ошибки (рис. 4.2). Для метода УМАВ данный автомат был модифицирован следующим образом – для каждой метки ошибки создавался отдельный переход в ошибочное состояние (рис. 4.3).

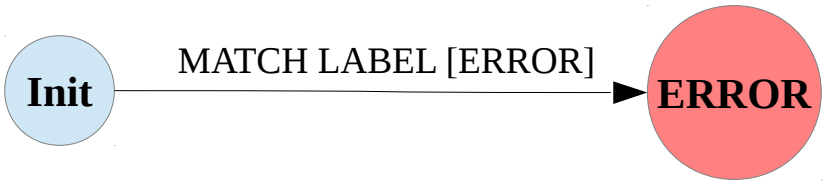

Рис. 4.2. Автомат, переходящий в ошибочное состояние при достижении метки *ERROR*.

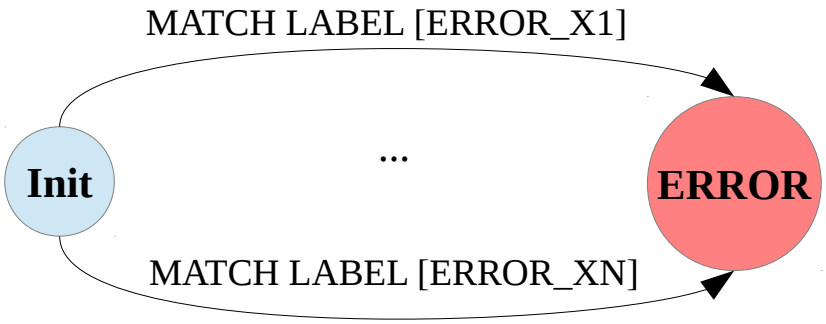

Рис. 4.3. Автомат, переходящий в ошибочное состояние при достижении одной из меток *ERROR_X1*, …, *ERROR_XN*.

Данный способ является достаточно эффективным, поскольку требует минимум накладных расходов на проверку наблюдательного автомата (аналогично базовому подходу верификации). Кроме того, добавляя или удаляя переходы в этом автомате, можно в процессе верификации частично модифицировать задачу достижимости, меняя множество проверяемых требований (например, на шаге *отключение ППУ* в методе УМАВ).

4.1.4. Поддержка метода условной многоаспектной верификации

После объединения моделей требований подготовленная задача достижимости (т. е. инструментированный код и автомат с метками ошибок) передается методу УМАВ. В случае внутреннего перезапуска под методом УМАВ

понимается алгоритм статического верификатора, который после завершения предоставит в систему LDV Tools результат верификации, т. е. вердикт для каждого из проверяемых требований и, возможно, трассы ошибок или причины невозможности верификации требования.

В случае внешнего перезапуска предполагается, что статический верификатор реализует только базовый алгоритм MAB. В этом случае после завершения каждой итерации система LDV Tools получает на вход результат в общем формате УМАВ и применяет к нему алгоритм завершения УМАВ для того, чтобы определить, нужна ли следующая итерация. Если верификация не завершена, то полученный в ходе итерации результат запоминается, задача достижимости изменяется путем удаления из нее всех модельных функций, соответствующих требованиям, которые уже получили вердикт, создается новый автомат с метками ошибок и запускается следующая итерация. В противном случае система LDV Tools собирает информацию, полученную от всех итераций, и предоставляет ее в качестве результата верификации.

4.1.5. Поддержка методов, основанных на автоматной спецификации

Если автоматная спецификация представлена только одним требованием, то она передается вместе с подготовленным кодом методу AC – базовому методу верификации для автоматной спецификации. В данном случае верификатор обязан поддерживать метод AC.

Если же автоматная спецификация задана несколькими требованиями, то используется метод ДАС. В общем случае каждому требованию может соответствовать несколько автоматов, поэтому предлагается все подобные автоматы записывать в один файл с именем, соответствующим идентификатору требования, и передавать верификатору набор файлов с автоматами. Никакие дополнительные действия по объединению автоматов не требуются. На выходе от

верификатора ожидается вердикт для каждого проверяемого требования и трассы ошибок для нарушенных требований.

4.1.6. Поддержка методов обнаружения всех однотипных нарушений

Метод ОВН может использоваться совместно с любым методом последовательной верификации и методом УМАВ. В данном случае от статического верификатора ожидается множество трасс ошибок и проведение их внутренней фильтрации. После завершения верификации все трассы ошибок проходят через внешнюю фильтрацию (рис. 4.1), а затем представляются для анализа человеку.

Для внешней фильтрации в системе LDV Tools был реализован фильтр «модельные функции». В данном случае под модельными функциями понимаются либо все добавленные в результате инструментирования функции на нарушенное в трассе ошибки требование, либо присутствующие на переходах в автомате, соответствующем нарушенному требованию. Функция преобразования для фильтра «модельные функции» работает по следующей схеме. Вначале строится дерево вызовов функций – упорядоченное дерево, которое соответствует вызовам функций в трассе ошибки. Для получения дерева вызова функций в трассе ошибки оставляются только операции вызова функций (т. е. CALL) и возврата из функции (т. е. RETURN). Далее в построенном дереве функций удаляются все поддеревья, в которых не присутствуют вызовы модельных функций. Данный фильтр основан на способе представления трассы ошибки в системе LDV Tools [25], в котором именно в таком виде трасса ошибки и представляется для анализа человеку (т. е. нет необходимости оставлять в трассе элементы, которые не будут показаны человеку).

Для поддержки полуавтоматической фильтрации был расширен существующий web-интерфейс системы LDV Tools, предназначенный для ручного

анализа результатов верификации. Во-первых, добавлена возможность помечать часть трассы ошибки, которая, по мнению человека, содержит причину нарушения требования. Во-вторых, реализованы 2 функции преобразования (без преобразования и на основе модельных функций) и 2 функции сравнения (на полную эквивалентность и на включение), из которых человек может выбирать произвольную комбинацию. В частности, имеется возможность выделить фрагмент трассы, выделить из него только вызовы модельных функций и искать включение данного фрагмента в других трассах (для отделения причины нарушения требования от его проявлений). Помимо этого, имеется возможность добавления собственных функций преобразования и сравнения.

4.2. Расширения статического верификатора

В качестве базового статического верификатора для реализации предложенных методов был выбран инструмент CPAchecker [29, 30]. Статический верификатор CPAchecker, созданный и поддерживаемый в университете Пассау (Германия), является одним из лидеров в категории верификации драйверов устройств. Например, на международных мероприятиях SV-COMP 2016 (competition on software verification) [16] данный инструмент завоевал 3 золотых (в том числе и в категории верификации драйверов устройств), 4 серебряных (в том числе и в суммарной категории) и 2 бронзовых медали[6]. Данный инструмент реализует подходы CEGAR и CPA, поддерживает наблюдательные автоматы, является достаточно легко расширяемым, а кроме того, имеет возможность продолжать верификацию после нахождения первого нарушения требования с возможностью осуществления внутренней фильтрации найденных трасс ошибок с помощью 2 простейших фильтров – сравнение на полную эквивалентность трасс ошибок и сравнение по блокам ГПУ программы.

В инструменте CPAchecker был реализован метод УМАВ с возможностью

6 http://sv-comp.sosy-lab.org/2016/results/results-verified.

внутреннего перезапуска как расширение подхода CEGAR, метод АС как расширение наблюдательных автоматов и метод ДАС как расширение подхода CPA.

4.2.1. Метод условной многоаспектной верификации

Метод УМАВ был реализован согласно описанию, предложенному в п. 2.3 и п. 2.6. Помимо этого, частично были реализованы расширения, о которых шла речь в п. 2.4.

4.2.1.1. Внутреннее представление утверждений

CPAchecker принимает от LDV Tools конфигурацию, которая представляет из себя автомат с метками (рис. 4.3). При этом в названии метки содержится уникальный идентификатор соответствующего ей требования. На шаге *создание утверждений* для каждой метки создается отдельное утверждение, для которого запоминается данная метка ошибки. На шаге *отключение ППУ* CPAchecker перестает проверять соответствующую метку, удаляя соответствующий ей переход в автомате (рис. 4.3).

4.2.1.2. Верификационные факты

В качестве верификационных фактов используются абстрактные состояния, точность абстракции и кэши верификатора (содержащие, например, запросы к решателю). Для каждого утверждения сохраняется только точность абстракции (на шаге *смена ППУ*) из-за малого размера и большого влияния на алгоритм [43]. На шаге *отключение ППУ* производится корректировка точности абстракции с целью удаления точности, соответствующей отключаемому утверждению. Абстрактные состояния и кэши верификатора не отслеживаются для каждого утверждения и, соответственно, не удаляются, поскольку данные операции неэффективны. Однако может возникнуть ситуация, в которой произошел экспоненциальный рост числа состояний при построении АГД (например, 99% всех абстрактных состояний

соответствует утверждению, нарушившему ВЛУ) или кэши верификатора содержат только информацию об удаленном утверждении, а потому будут бесполезны. В таких случаях гораздо проще и быстрее полностью перестроить АГД. Для этого предлагается искусственно запускать новую итерацию УМАВ (подробнее в следующем пункте). Таким образом, в течение всей итерации УМАВ абстрактные состояния и кэши верификатора переиспользуются между всеми проверяемыми утверждениями.

Для реализации операций сохранения и корректировки точность абстракции была обобщена до регулируемой точности. Регулируемая точность добавляет операции сложения, вычитания и очистки для точности абстракции, описанной в п. 1.1, на основе операций с множествами. На шаге *смена ППУ* полученная в ходе уточнения абстракции точность складывается с регулируемой точностью ППУ. Данная регулируемая точность будет переиспользоваться между всеми утверждениями вплоть до шага *отключение ППУ* для данного утверждения (т. е. пока соответствующее ей утверждение не получит вердикт *Unsafe* или *Unknown*) или до окончания итерации УМАВ.

4.2.1.3. Внутренние лимиты

Одним из наиболее эффективных решений проблемы экспоненциального роста числа состояний при построении АГД является запуск новой итерации УМАВ – экспоненциальный рост числа состояний предотвратить не удалось, однако известно, какое требование его вызвало. Для этого предлагается использовать внутренний лимит пустых интервалов (ВЛПИ). Пустым интервалом назовем промежуток времени от операции *отключение ППУ* до следующей точки смены ППУ. Заметим, что пустой интервал возможен только в случае, в котором одно из утверждений получило вердикт *Unknown* или *Unsafe* (или *Unsafe-incomplete* в случае использования метода УМАВ с ОВН). ВЛПИ ограничивает

время пустого интервала и при его нарушении завершает текущую итерацию УМАВ, при этом для следующей итерации уже имеется хотя бы на одно утверждение меньше. В данном случае считается, что слишком большой пустой интервал соответствует экспоненциальному росту числа состояний в АГД, поэтому лучше перезапустить алгоритм для остальных утверждений. Данный лимит определяет баланс переиспользования абстрактных состояний и кэшей верификатора.

Помимо этого, предлагается ограничить время произвольного интервала ППУ с помощью внутреннего лимита интервала (ВЛИ). В случае нарушения ВЛИ ППУ получит вердикт *Unknown* и будет отключено. Слишком большой интервал ППУ означает, что некоторое утверждение требует значительно более точной абстракции и, скорее всего, так или иначе нарушит ВЛУ, а при этом также приведет к экспоненциальному росту числа состояний, что помешает проверке других утверждений. Поэтому требуется как можно скорее найти и изолировать подобные утверждения.

Также предлагается ограничивать время первого интервала, считая, что слишком большой первый интервал соответствует слишком сложной задаче достижимости, с помощью внутреннего лимита начального интервала (ВЛНИ). Как известно, построение абстракции для пустой точности (т. е. первый интервал в методе УМАВ) гораздо проще, чем более точное построение абстракции, поэтому если алгоритм не может справиться с самым простым шагом, то с более сложными шагами он также не сможет справиться.

Стоит заметить, что предложенная реализация внутренних лимитов может быть полезна и за рамками верификации. Так, например, при построении АГД можно выявлять сложные участки ГПУ с помощью подобных внутренних лимитов, а затем пропускать их, корректируя уровень абстракции соответствующим образом, чтобы предотвратить негативное влияние

отключаемых элементов ГПУ. В данном случае не удастся доказать корректность, однако подобные идеи могут быть использованы для быстрого поиска ошибок.

4.2.1.4. Стратегии корректировки уровня абстракции

Поскольку в качестве верификационных фактов для каждого утверждения хранится только регулируемая точность, то на шаге *отключение ППУ* требуется очистить только регулируемую точность. Предлагаются различные стратегии корректировки уровня абстракции в зависимости от того, где производится операция – либо в каждом абстрактном состоянии (весь АГД), либо только в тех абстрактных состояниях, которые еще не были обработаны (так называемый лист ожидания, или ЛО, (от англ. waitlist)), и как – либо вычитанием из точности абстрактного состояния точности ППУ, либо полной очисткой точности абстрактного состояния. В общем случае возможно 5 различных стратегий:

1. *Ничего* (ничего не очищать);
2. *ЛО/Вычитание* (вычесть из точности каждого абстрактного состояния из листа ожидания точность ППУ);
3. *АГД/Вычитание* (вычесть из точности каждого абстрактного состояния из АГД точность ППУ);
4. *ЛО/Очистка* (полностью очистить точность в листе ожидания);
5. *Все* (полностью очистить точность во всем АГД).

Данные стратегии влияют на баланс между позитивным и негативным эффектом переиспользования точности, с одной стороны, и между временем и качеством самой операции, с другой стороны. В общем случае существуют примеры, в которых определенная стратегия будет работать лучше других.

4.2.1.5. Внутренняя условная многоаспектная верификация

Помимо базового алгоритма MAB в инструменте CPAchecker была реализована поддержка условной многоаспектной верификации с внутренним

перезапуском. Главным регулятором итераций УМАВ выступает ВЛПИ, в случае нарушения которого текущая итерация УМАВ завершается и начинается новая. В методе внутренней УМАВ ВЛПИ перехватывается на уровне верификатора, происходит остановка верификации, АГД полностью очищается, после чего начинается новая итерация. Множество проверяемых утверждений (т. е. автомат с метками) при перезапуске итерации никак не модифицируется, поскольку на момент нарушения ВЛПИ как минимум одно утверждение должно уже быть отключено. Кэши верификатора при этом предлагается переиспользовать между итерациями, поскольку по ним проще начинать «восстанавливать» новый АГД, однако уже без отключенных утверждений. Помимо этого, также переиспользуется и ГПУ программы между итерациями.

Конечно, внутренний перезапуск не сможет перехватывать ошибки в самом инструменте и нехватку оперативной памяти, вызванную только частью требований, однако на практике подобных ситуаций относительно мало.

4.2.2. Метод декомпозиции автоматной спецификации

Методы АС и ДАС были реализованы как развитие уже имеющейся в инструменте CPAchecker технологии наблюдательных автоматов (примеры которых приведены на рис. 4.2 и 4.3). Язык задания наблюдательных автоматов был расширен на основе предложенной в п. 3.1 грамматики. Для каждого переданного автомата создается свой CPA, используемый вместе с остальными в композитном CPA. Алгоритм ДАС был реализован как общий интерфейс поверх подхода CPA и способен работать с произвольными алгоритмами, реализованными в инструменте CPAchecker. Предложенные стратегии разбиения являются реализациями данного интерфейса.

4.2.3. Сравнение реализаций

Итак, было реализовано два метода для одновременной проверки нескольких

требований – УМАВ и ДАС. Для определения того, какой из методов лучше, рассмотрим их основные отличия:

1. Сопоставление инструментирования и автоматов. В методе УМАВ основной недостаток инструментирования (т. е. усложнение кода) проявляются особенно ярко при проверке композиции требований – десятки тысяч строк кода могут быть добавлены дополнительно к проверяемому при верификации любого относительно большого количества требований. С данной точки зрения, метод ДАС является более эффективным.

2. В то время как метод УМАВ реализует одну эвристику, которая имеет смысл только в рамках подхода CEGAR, метод ДАС нацелен на поддержание инфраструктуры для создания подобных эвристик. С данной точки зрения, метод УМАВ более оптимизирован в рамках подхода CEGAR, а метод ДАС может быть использован и вне подхода CEGAR.

3. Предложенные методы по-разному подходят к решению проблемы экспоненциального роста числа состояний. В методе УМАВ изначально все требования проверяются в одной группе, поэтому экспоненциальный рост числа состояний, вызванный проверкой одного требования, обязательно затруднит верификацию для остальных. Именно поэтому в методе УМАВ были реализованы дополнительные эвристики для более раннего обнаружения таких ситуаций (см. п. 4.2.1.3). Метод ДАС решает несколько другую задачу – разбиение спецификации на группы требований для изоляции требований, которые ведут к экспоненциальному росту числа состояний. В данном случае метод ДАС более устойчив к добавлению требований, для которых относительно часто будет наблюдаться экспоненциальный рост числа состояний, т. к. они будут проверяться независимо от остальных. Поэтому метод ДАС не требует обязательного использования эвристик.

4. При этом метод ДАС является более ресурсоемким. Во-первых, метод

требует больше накладных расходов на поддержание всей инфраструктуры (множество автоматов в виде CPA, применение стратегий разбиения поверх алгоритма верификации и т. д.), в то время как метод УМАВ по накладным расходам сопоставим с базовым методом верификации. Во-вторых, в методе ДАС не выполняются различные оптимизации, например, корректировка уровня абстракции, поскольку данные операции могут не быть определены за рамками подхода CEGAR.

5. Нет поддержки ОВН в методе ДАС[7] – дополнительное ограничение на область применимости метода.

Таким образом, оба метода имеют свои достоинства и недостатки, а также свою область применения.

4.3. Последовательная комбинация предложенных методов

Как было отмечено в предыдущем пункте, оба метода УМАВ и ДАС обладают своими преимуществами и недостатками и в общем случае существуют задачи, для которых больше подойдет один из методов. Однако на практике заранее может быть неясно, какой метод наиболее подходящий, а пользователю нужно эффективное решение задачи с минимальными потерями результата.

Для решения данной проблемы была предложена последовательная комбинация данных методов на основе уже созданной инфраструктуры системы верификации (рис. 4.1). Основная цель последовательной комбинации – максимально использовать преимущества обоих методов и минимизировать проявление их недостатков.

Последовательная комбинация состоит из трех шагов:

1. Все требования проверяются в первой итерации внешнего УМАВ. Поскольку метод УМАВ является более оптимизированным, то так удастся решить

7 Поддержка не была добавлена, поскольку основная задача метода заключалась именно в верификации нескольких требований; для поиска всех нарушений требований вполне достаточно метода ОВН.

большинство задач более эффективно. Если все требования получили вердикт (*Safe* или *Unsafe*), то алгоритм завершается.

2. Все требования, получившие вердикт *Unknown* на предыдущем шаге из-за эвристического ВЛИ, проверяются методом ДАС с помощью стратегии *Последовательная проверка*. В данном случае предполагается, что подобные требования приводят к экспоненциальному росту числа состояний при проверке вместе с остальными, поэтому их эффективнее проверять по отдельности, переиспользуя при этом ГПУ и кэши верификатора. Если же получилось только одно такое требование, то в данном случае достаточно использовать менее ресурсоемкий метод АС.

3. Все требования, которые не получили вердикт на первом шаге или которые получили вердикт *Unknown* из-за эвристического ВЛНИ, проверяются методом ДАС с помощью базовой стратегии разбиения (по умолчанию с помощью стратегии *Релевантность*). В этом случае считается, что для данных требований метод УМАВ не смог ничего установить, поэтому, скорее всего, проверяемая задача достаточно сложная и на ее решение вполне разумно использовать более дорогой метод ДАС, выполняющий автоматическую декомпозицию требований.

Если требуется использование метода ОВН, то вся стратегия заменяется на метод УМАВ с ОВН.

Из теорем 1 и 2 следует полнота и корректность последовательной комбинации методов относительно метода последовательной верификации для требований, удовлетворяющих ограничениям из п. 2.1.1.

Данная стратегия обладает рядом преимуществ. Во-первых, для второго и третьего шагов из задачи достижимости будут удалены модельные функции, добавленные при инструментировании для метода УМАВ, что позволит использовать более эффективные автоматные спецификации. Во-вторых, ресурсоемкий метод ДАС, позволяющий разбивать спецификацию автоматически,

будет применяться только для потенциально наиболее сложных задач, в то время как более оптимизированный метод УМАВ сможет быстрее решать простые задачи. В-третьих, результат работы первого шага всегда будет переиспользован – либо он полностью решает задачу, либо предоставляет информацию последующим шагам о выполнении декомпозиции спецификации, что повышает эффективность метода. В-четвертых, все шаги представляют собой отдельные запуски верификатора, которым подаются различные задачи, что делает стратегию устойчивой к внутренним ошибкам в верификаторе.

Стоит отметить, что данная стратегия не претендует на оптимальность, однако она будет полезна для многих пользователей в качестве стратегии по умолчанию, поскольку не требует никаких дополнительных знаний для достижения наиболее эффективных результатов в большинстве случаев.

4.4. Выводы

Таким образом, все предложенные в данной работе методы (обнаружение всех однотипных нарушений, условная многоаспектная верификация, автоматные спецификации и декомпозиция автоматной спецификации) были реализованы как расширения системы верификации LDV Tools и статического верификатора CPAchecker. Все реализованные методы будут сравнены и оценены как с базовым подходом, так и между собой в следующей главе.

Глава 5. Экспериментальная оценка предложенных методов

Для полноценного сопоставления реализованных методов была предложена следующая схема проведения экспериментов (рис. 5.1).

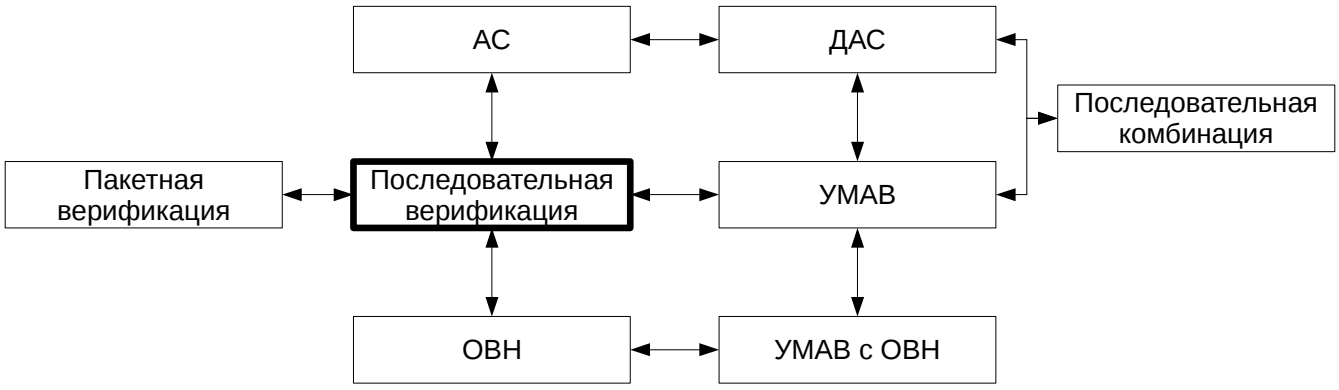

Рис. 5.1. Схема проведения экспериментов.

В качестве базового метода используется последовательная верификация (на основе подхода CEGAR и инструментирования исходного кода). Согласно схеме (рис. 5.1) были произведены эксперименты, нацеленные на:

1. оценку результатов базового метода;
2. сравнение базового метода с методом ОВН;
3. сравнение базового метода с методом пакетной верификации;
4. сравнение базового метода с методом УМАВ;
5. сравнение методов УМАВ и ОВН с методом УМАВ с ОВН;
6. сравнение базового метода с методом АС;
7. сравнение метода АС с методом ДАС;
8. сопоставление методов УМАВ, ДАС и их последовательной комбинации между собой.

Поскольку цели методов различаются (методы ОВН и УМАВ с ОВН нацелены на нахождение большего числа нарушений требований, а остальные – на повышение производительности статической верификации нескольких требований), то для их сопоставления с базовым методом использовались

различные критерии. При сравнении методов обнаружения всех однотипных нарушений главными критериями были дополнительные нарушения требований (в том числе и реальные ошибки) в сравнении с базовым методом и необходимые для их выявления ресурсы. При сравнении методов проверки композиции требований главными критериями были ускорение и потери[8] относительно базового метода. Кроме того, в данном случае интерес представляют новые вердикты относительно базового метода (т. е. не решенные базовым методом задачи, которые оказались успешно решены предложенным). В обоих случаях оценивались как необходимые для работы методов ресурсы (в процессорном времени), так и интегральное влияние методов на весь процесс верификации (т. е. процессорное время всего процесса верификации и время, которое человек должен потратить на анализ результатов).

Все эксперименты проводились с помощью системы верификации LDV Tools[9][26], которая подготавливала задачи достижимости на основе модулей ядра Linux, и статического верификатора CPAchecker [30], который решал задачи достижимости. Проверялись все модули ядра операционной системы Linux версии 4.0-rc1[10] в конфигурации *allmodconfig* (т. е. 4 779 модулей), для которых система LDV Tools указанной версии успешно готовит 4 041 задачу достижимости, относительно 30 требований (приведены в Приложении А), т. е. от системы верификации ожидаются 121 230 вердиктов. В совокупности проверяемые модули содержат около 9 млн. строк исходного кода. Для 738 модулей (15% от числа всех модулей) система LDV Tools не смогла подготовить задачи достижимости из-за неуспешного завершения других компонентов (например, из-за того, что не удается подготовить модель окружения для модуля). Для методов условной

8 Т. е. задачи достижимости, которые не могут быть успешно решены предложенным методом с теми же ограничениями на ресурсы, что и в базовом методе. Стоит отметить, что согласно утверждениям 1-3 и теоремам 1 и 2 в предложенных методах нет пропущенных нарушений требований и новых ложных нарушений.

9 Git репозиторий: https://forge.ispras.ru/git/klever.git, ветка *composition_of_reachability_tasks*, коммит 63fa9f39.

10 https://www.kernel.org/pub/linux/kernel/v4.x/testing/linux-4.0-rc1.tar.xz.

многоаспектной верификации использовалась версия 23 131 инструмента CPAchecker[11] ветки *cmav*, для методов декомпозиции автоматной спецификации – версия 20 125 инструмента CPAchecker ветки *muauto*. Предложенные версии инструмента являются сопоставимыми, т. е. берут начало из одной ревизии основной ветки инструмента CPAchecker и отличаются только реализованными методами. Инструмент CPAchecker использовался с конфигурацией *ldv*, которая включает предикатную абстракцию [40] и анализ явных значений [41]. Все эксперименты проводились на компьютерах (узлах кластера) со следующими характеристиками: процессор Intel Xeon E312xx (Sandy Bridge) 2.6 GHz (8 ядер), 64 GB оперативной памяти, операционная система Ubuntu 14.04 (64-bit) с ядром Linux 3.13, Java версии 1.7.0_101 (вычислительный кластер ИСП РАН).

Базовое ограничение на используемые ресурсы было выбрано в соответствии с международными мероприятиями по верификации SV-COMP[12]: 15 минут процессорного времени и 15 GB оперативной памяти на проверку одного требования в одной задаче достижимости. Дополнительно использовалось ограничение по памяти на размер кучи для виртуальной машины Java в 13 GB (13/15 от базового ограничения на оперативную память), что необходимо для корректной работы верификатора CPAchecker. При этом на проверку N требований в одной задаче достижимости ограничение на процессорное время увеличивалось пропорционально числу требований, поскольку сравнение методов предполагалось в условиях максимально возможного одинакового ограничения на ресурсы. Однако ограничение на оперативную память не изменялось, т. к. подготавливаемые задачи должны решаться на тех же компьютерах.

В Приложении Б приведены рекомендации по выбору параметров использования предложенных методов на основе проведения дополнительных экспериментов.

11 Svn репозиторий: https://svn.sosy-lab.org/software/cpachecker.
12 https://sv-comp.sosy-lab.org/2016/rules.php.

5.1. Оценка метода последовательной верификации

На подготовку 121 230 задач достижимости система LDV Tools затратила 2 853 000 секунд процессорного времени. Решение данных задач статическим верификатором CPAchecker потребовало 3 889 000 секунд процессорного времени (т. е. суммарно процесс верификации занял 6 742 000 секунд процессорного времени), в результате чего было получено 118 703 вердикта *Safe* и 667 вердиктов *Unsafe* (т. е. нарушений требований), с решением 1 860 задач верификатор не справился (в основном из-за исчерпания выделенных ресурсов).

Анализ найденных нарушений требований выявил 121 реальную ошибку (т. е. 18% от всех вердиктов *Unsafe*) в модулях ядра Linux и потребовал порядка 3 дней (одним человеком). Среди 546 ложных сообщений об ошибках 356 проблем (65%) связано с некорректной подготовкой модели окружения [53] на этапе подготовки исходного кода модуля для верификации (см. рис. 4.1), 118 проблем (22%) было вызвано неполной поддержкой верификатором CPAchecker некоторых конструкций языка программирования C (например, частичная поддержка указателей) и 72 проблемы (13%) заключались в различных упрощениях моделей требований (например, в модельных функциях зачастую не моделируются побочные эффекты, поскольку на практике это серьезно усложняет задачи достижимости). С практической точки зрения методы верификации композиции требований не должны терять 121 реальную ошибку.

5.2. Оценка метода пакетной верификации

Данный эксперимент был нацелен на сравнение необходимых ресурсов и результата методов последовательной и пакетной верификации. Вместо 121 230 задач метод пакетной верификации подготавливал 4 041 задачу (на основе предложенного в п. 2.1 метода объединения моделей требований), на решение каждой из которых выделялось 27 000 секунд процессорного времени (по 900 секунд на 30 требований).

Результаты эксперимента представлены в таблице 5.1. Метод пакетной верификации потерял порядка 17% результата, при этом время решения задач достижимости (т. е. верификатора CPAchecker) практически не сократилось в сравнении с методом последовательной верификации. Суммарное время процесса верификации (т. е. системы LDV Tools) сократилось почти в два раза, но только за счет того, что необходимо подготавливать в 30 раз меньше задач достижимости.

Метод	*Safe*	*Unsafe*	Потери / новые (%)	Процессорное время / ускорение	
				CPAchecker	LDV Tools
Последовательная верификация	118 703	667	–0 +0	3 889 000 1	6 742 000 1
Пакетная верификация	98 580	527	–16.8 +0.09	3 527 000 1.1	3 780 000 1.78

Табл. 5.1. Сравнение методов последовательной и пакетной верификации.

Таким образом, пакетная верификация практически не сокращает требуемые для верификатора ресурсы, при этом приводит к значительным потерям результата.

5.3. Оценка метода обнаружения всех однотипных нарушений

Данный эксперимент был нацелен на определение того, сколько новых реальных ошибок может быть найдено с помощью метода ОВН и какие для этого необходимы ресурсы в сравнении с последовательной верификацией. Данные характеристики должны показать, перспективно ли с помощью статической верификации стремиться найти все нарушения требования.

Без фильтрации трасс ошибок метод ОВН вместо 667 в методе последовательной верификации нашел 110 874 трассы ошибок, поэтому без фильтрации трасс данный метод не имеет перспектив использования на практике. В качестве внутренней фильтрации использовался фильтр по блокам ГПУ программы [39] в инструменте CPAchecker, который позволил сократить число трасс ошибок до 15 700 (14% от первоначального числа). Внешняя фильтрация на

основе фильтра «модельные функции» (см. п. 4.1.6) позволила уменьшить число трасс ошибок до 4 380 (4% от первоначального числа) и дополнительно потребовала 4 700 секунд процессорного времени. Таким образом, большая часть найденных трасс ошибок в методе ОВН может быть отфильтрована автоматически.

Для полученных после автоматической фильтрации трасс ошибок была проведена полуавтоматическая фильтрация, которая заняла около 5 дней[13] (одним человеком) и позволила выявить 1 042 различных нарушения требований (в том числе и 375 новых), т. е. метод нашел в 1.56 раза больше нарушений требований. Среди новых нарушений требований было выявлено 70 реальных ошибок. При анализе 121 известной реальной ошибки для 41 случая (34%) новых нарушений не обнаружено, для 21 случая (17%) в дальнейшем обнаружено новое ложное нарушение требования, для 59 случаев (49%) в дальнейшем обнаружена новая реальная ошибка. Таким образом, метод ОВН всего позволил выявить 191 реальную ошибку (включая 70 новых), что в 1.58 раза больше базового метода.

Из 667 вердиктов *Unsafe* 269 (40%) перешли в *Unsafe-incomplete*, т. е. в 60% случаев удалось доказать, что найдены все нарушения проверяемого требования. При этом время решения задач достижимости, в которых нарушается проверяемое требование, возросло с 45 000 секунд процессорного времени до 300 000 секунд (т. е. примерно в 7 раз). Поскольку для остальных вердиктов (*Safe* и *Unknown*) метод ОВН ничем не отличается от метода последовательной верификации, то увеличение времени решения всех задач достижимости относительно мало (с 3 889 000 секунд до 4 149 000, т. е. в 1.1 раза).

Данный эксперимент показал, что метод ОВН способен находить в среднем в 1.5 раза больше реальных ошибок, чем последовательная верификация, помимо этого, в более чем половине случаев доказывает, что найдены все нарушения

13 При этом параллельно определялось, соответствует ли нарушение требования реальной ошибке.

требования в программе. Однако для этого требуется примерно в 1.1 раза больше ресурсов на решение задач достижимости и значительное время человека для проведения полуавтоматической фильтрации трасс ошибок. Таким образом, метод ОВН требует больше ресурсов и позволяет находить больше ошибок.

5.4. Оценка метода условной многоаспектной верификации

Основная цель данного эксперимента состояла в том, чтобы оценить ускорение и потери метода УМАВ относительно базового метода. В качестве конфигурации метода УМАВ был взят внутренний перезапуск, стратегия корректировки уровня абстракции *Все* и следующие значения внутренних лимитов: ВЛУ=900 секунд (т. е. базовое ограничение на проверку одного требования), ВЛПИ=20 секунд, ВЛИ=100 секунд, ВЛНИ=100 секунд (подробно выбор конфигурации описан в Приложении Б). Результаты эксперимента представлены в таблице 5.2.

Метод	*Safe*	*Unsafe*	Потери / новые (%)	Процессорное время / ускорение	
				CPAchecker	LDV Tools
Последовательная верификация	118 703	667	–0 +0	3 889 000 1	6 742 000 1
Метод УМАВ	117 162	634	–1.36 +0.06	1 289 000 3.02	1 514 000 4.45

Табл. 5.2. Сравнение метода УМАВ с последовательной верификацией.

Метод УМАВ позволил сократить время решения задач достижимости более чем в 3 раза по сравнению с последовательной верификацией, а с учетом генерации задач достижимости – примерно в 4.5 раза. Среднее время решения одной задачи составило 318 секунд, максимальное время решения одной задачи – 12 200 секунд. Среднее ускорение[14] решения задач примерно равно 8, что демонстрирует эффективность применения метода УМАВ для решения отдельных

14 Т. е. среднее арифметическое отношений времени решения каждой задачи базовым методом ко времени решения той же задачи предложенным методом.

задач достижимости. Более детальное сопоставление времени решения задач методами УМАВ и последовательной верификации представлено на рисунке 5.2.

Однако помимо производительности важной характеристикой предложенного метода является результат при его сопоставлении с базовым методом верификации. Число потерь УМАВ (т. е. вердикты *Safe* или *Unsafe* в последовательной верификации перешли в *Unknown* в УМАВ) составило 1 648 вердиктов (или 1.36%). Стоит отметить, что в УМАВ нет пропущенных ошибок (т.е. переходов из *Unsafe* в *Safe*) и дополнительных ложных сообщений об ошибках (т.е. переходов из *Safe* в *Unsafe*) в сравнении с последовательной верификацией, т. е. справедливость теоремы 1 подтверждается на практике.

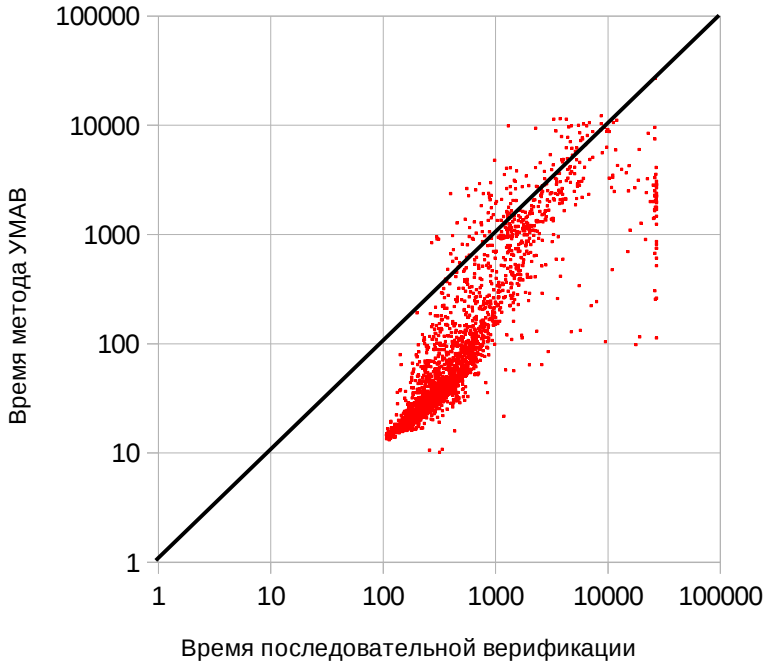

Рис. 5.2. Сравнение времени решения задач достижимости методами УМАВ и последовательной верификации.

Первой причиной потерь в УМАВ являются более сложные задачи достижимости, созданные на основе инструментирования 30 моделей требований, а не одного, как в последовательной верификации. В данном случае для получения

результата, аналогичного последовательной верификации, необходимо увеличивать базовое ограничение на проверку требования в методе УМАВ (т. е. ВЛУ), устанавливая его больше, чем в последовательной верификации (например, с 900 секунд до 1 000). Более концептуально данная проблема решается в методах АС и ДАС, в которых задачи не усложняются при проверке композиции требований.

Вторая причина потерь в УМАВ заключается в эвристических внутренних лимитах (т. е. ВЛИ и ВЛНИ). Так, например, при увеличении ВЛИ до 255 секунд можно найти 7 потерянных вердиктов *Unsafe*, однако при этом общее время верификации увеличится почти в 1.7 раза (более подробно проблема описана в Приложении Б).

Третья причина потерь в УМАВ заключается в использовании эвристики об одном единовременно проверяемом утверждении. Так, например, более половины всех пропущенных вердиктов получили вердикт *Unknown* из-за того, что текущее проверяемое утверждение ошибочно считалось равным последнему проверяемому. В итоге перед отключением «настоящего» *Unknown*, ошибочно отключались другие утверждения. Одним из возможных методов решения данной проблемы является метод ДАС, который не основан на эвристиках.

В то же время метод УМАВ получил 74 новых вердикта (т. е. вердикты *Unknown* в последовательной верификации перешли в вердикты *Safe* или *Unsafe* в методе УМАВ), основная причина появления которых заключается в переиспользовании точности абстракции, АГД и кэшей верификатора при проверке различных требований. Среди данных новых вердиктов имелось 2 реальные ошибки, которые из-за нехватки ресурсов не могли быть выявлены с помощью базового метода. Поэтому, несмотря на то что новые вердикты – редкое явление (0.06% от общего числа вердиктов), они могут быть достаточно полезны на практике.

Данный эксперимент демонстрирует, что метод УМАВ сокращает требуемые ресурсы как на решение задач достижимости более чем в 3 раза, так и на весь процесс верификации более чем в 4 раза, при этом потери результата метода менее 2%. Таким образом, метод УМАВ существенно повышает производительность верификации композиции требований с незначительными потерями результата в рамках подхода CEGAR.

5.5. Оценка метода условной многоаспектной верификации с обнаружением всех однотипных нарушений

Данный эксперимент был нацелен на определение того, можно ли объединять методы УМАВ и ОВН на практике для нахождения всех нарушений всех проверяемых требований. Для этого использовался метод УМАВ с базовой конфигурацией (см. п. 5.4) и метод ОВН с фильтром по блокам ГПУ программы [39] для внутренней фильтрации и фильтром «модельные функции» для внешней фильтрации (см. п. 5.3).

Без фильтрации трасс ошибок метод нашел 160 456 трасс ошибок, внутренняя фильтрация сократила их число до 14 595 трасс, а внешняя – до 2 576, т. е. автоматически было отфильтровано более 98% трасс ошибок. Время проведения внешней фильтрации заняло 5 500 секунд процессорного времени. Для проведения полуавтоматической фильтрации была применена полученная в методе ОВН ручная разметка трасс ошибок, которая показала, что в 95% случаев метод УМАВ с ОВН демонстрирует точно такой же результат, поэтому процесс полуавтоматической фильтрации для данного метода занял всего несколько часов (менее 1 дня)[15]. По сравнению с методом ОВН данный метод потерял 47 нарушений требований, при этом было обнаружено 7 новых нарушений требований, т. е. всего было выявлено 1 002 различных нарушения требований (т. е. в 1.58 раза больше метода УМАВ), среди которых было 368 новых. Из

15 Без имеющейся ручной разметки трасс данный процесс занял бы примерно столько же времени, что и в п. 5.3 (т. е. около 5 дней), поскольку 95% результата данных методов идентичны.

найденных нарушений требований метод УМАВ с ОВН потерял 11 реальных ошибок относительно метода ОВН и нашел 2 новых, т. е. всего метод УМАВ с ОВН нашел 182 реальных ошибки (т. е. в 1.6 раза больше метода УМАВ).

Из 634 вердиктов *Unsafe* 325 (51%) перешли в *Unsafe-incomplete*, т. е. в 49% случаев было доказано, что найдены все нарушения требований. При этом время решения задач достижимости, в которых нарушается хотя бы одно требование, возросло с 463 000 секунд процессорного времени до 639 000 секунд (в 1.4 раза). Большая часть времени в обоих случаях уходит на обработку вердиктов *Unknown* для других требований. Время решения всех задач достижимости увеличилось незначительно (с 1 289 000 секунд до 1 471 000, т. е. в 1.1 раза).

Данный эксперимент показал, что на практике возможно совместное использование методов УМАВ и ОВН. В сравнении с методом ОВН данный метод для меньшего числа задач способен найти все нарушения требования из-за того, что решаются заведомо более сложные задачи. С одной стороны, данный метод требует значительно меньше ресурсов (почти в 3 раза), чем метод ОВН, на решение задач из-за того, что метод УМАВ эффективнее метода последовательной верификации. С другой стороны, метод УМАВ с ОВН увеличивает время верификации в сравнении с методом УМАВ примерно в 1.1 раза. Таким образом, метод УМАВ с ОВН объединяет преимущества методов ОВН и УМАВ, однако ему присущи и недостатки метода ОВН (в частности, необходимость полуавтоматической фильтрации).

5.6. Оценка метода автоматных спецификаций

Данный эксперимент был нацелен на сопоставление эффективности верификации одних и тех же требований, формализованных на основе инструментирования и автоматных спецификаций. Результаты эксперимента представлены в таблице 5.3, детальное сопоставление времени решения задач при

последовательной верификации и в методе АС представлено на рисунке 5.3.

Метод	*Safe*	*Unsafe*	Потери / новые (%)	Процессорное время / ускорение	
				CPAchecker	LDV Tools
Инструментирование	118 703	667	–0 +0	3 889 000 1	6 742 000 1
Метод АС	118 929	680	–0.15 +0.35	3 434 000 1.13	6 107 000 1.1

Таблица 5.3. Результаты сравнения верификации требований, формализованных с помощью инструментирования и метода автоматных спецификаций.

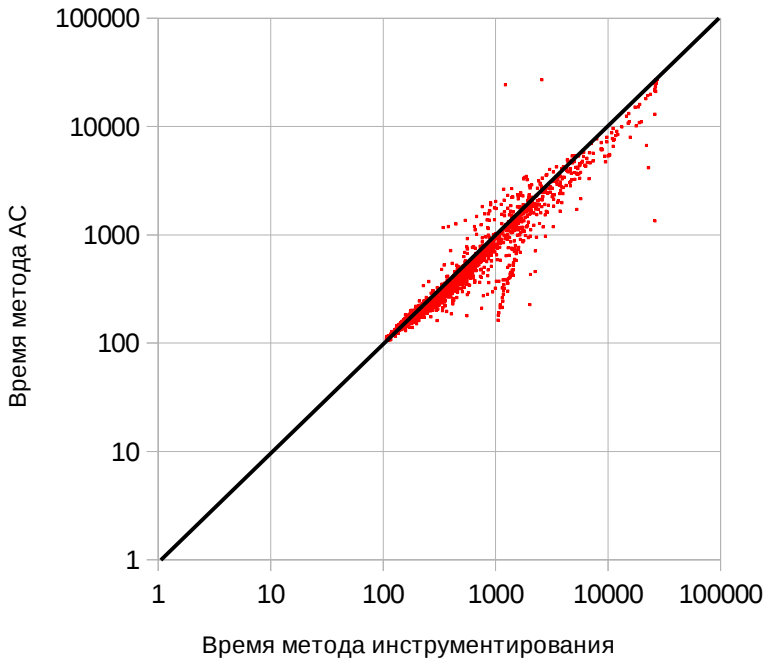

Рис. 5.3. Сравнение времени решения задач достижимости, формализованных с помощью инструментирования и метода автоматных спецификаций.

Метод АС успешно решил на 421 задачу больше (0.35%), при этом на решение задач потребовалось в 1.1 раза меньше процессорного времени, поскольку решаются более простые задачи. Среди новых успешно решенных задач оказалось 34 вердикта *Unsafe*, 2 из которых соответствовали новым реальным ошибкам в модулях ядра Linux (т. е. данные ошибки не могли быть найдены с помощью последовательной верификации с теми же ограничениями на ресурсы).

Кроме того, метод АС требует меньше ресурсов и на подготовку задач достижимости, поскольку в нем пропускается шаг инструментирования. Однако 182 вердикта (или 0.15%) оказалось потеряно в сравнении с инструментированием, для данных задач абстракция строилась быстрее для метода инструментирования. При этом среди потерянных вердиктов *Unsafe* нет тех, которые соответствуют реальным ошибкам.

Таким образом, эксперимент подтвердил предположение о том, что передача моделей требований независимо от исходного кода является более перспективным направлением, нежели инструментирование исходного кода – относительно инструментирования метод АС ускорил процесс верификации примерно в 1.1 раза и при этом смог успешно решить на 0.35% задач больше.

5.7. Оценка метода декомпозиции автоматной спецификации

Данный эксперимент был нацелен на оценку ускорения и потерь метода ДАС относительно метода АС. В качестве базовой конфигурации метода ДАС была выбрана стратегия разбиения *Релевантность*, первый шаг которой был ограничен 200 секундами, а второй – 1 200 секундами (выбор конфигурации описан в Приложении Б). Результаты эксперимента представлены в таблице 5.4.

Метод	*Safe*	*Unsafe*	Потери / новые (%)	Процессорное время / ускорение	
				CPAchecker	LDV Tools
Метод АС	118 929	680	–0 +0	3 434 000 1	6 107 000 1
Метод ДАС	118 386	673	–0.49 +0.04	1 373 000 2.5	1 550 000 3.94

Таблица 5.4. Результаты сравнения методов АС и ДАС.

Метод ДАС демонстрирует существенное ускорение как решения задач достижимости, так и всего процесса верификации. Детальное сопоставление времени решения задач с методом АС представлено на рисунке 5.4.

Число потерь метода менее 0.5%. Основная причина данных потерь

заключается в том, что метод является более ресурсоемким, чем верификация требований по отдельности методом АС (для каждого разбиения необходимо выполнять достаточно много вспомогательных операций: отключение и подключение соответствующих требованиям CPA с автоматами, полное перестроение абстракции и т. д.). Среди потерь имелись 2 реальные ошибки, в то же время среди новых вердиктов была найдена 1 новая реальная ошибка.

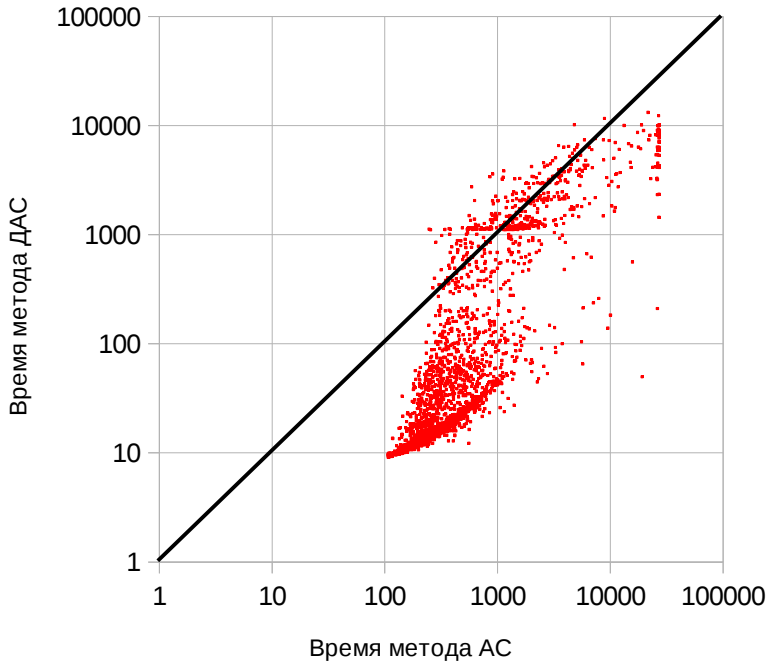

Рис. 5.4. Сравнение времени решения задач достижимости методами АС и ДАС.

Таким образом, метод ДАС демонстрирует сопоставимые с методом УМАВ результаты (общее ускорение верификации порядка 4 раз, потери менее 0.5%), при этом является более общим и применимым для большего круга задач.

5.8. Сопоставление методов верификации композиции требований

Данный эксперимент был нацелен на сравнение двух предложенных методов проверки композиции требований – УМАВ и ДАС – и их последовательной комбинации. В качестве результатов для методов УМАВ и ДАС использовались

эксперименты из п. 5.4 и п. 5.7 соответственно. В последовательной комбинации первый шаг (метод УМАВ) был ограничен 900 секундами процессорного времени. Для решения проблемы усложнения задач в методе УМАВ было предложено разбиение 30 требований на 2 группы требований (более подробно разбиение описано в Приложении Б). Результаты экспериментов представлены в таблице 5.5 (сравнение производится с последовательной верификацией).

Метод	*Safe*	*Unsafe* / реальные ошибки	Потери / новые (%)	Среднее ускорение	Процессорное время / общее ускорение	
					CPAchecker	LDV Tools
Последовательная верификация	118 703	667 –0 121 +0	–0 +0	1	3 889 000 1	6 742 000 1
Метод УМАВ	117 162	634 –9 114 +2	–1.36 +0.06	7.85	1 289 000 3.02	1 514 000 4.45
Метод УМАВ (с разбиением)	117 628	660 –2 121 +2	–0.93 +0.04	6.08	1 041 000 3.74	1 382 000 4.88
Метод ДАС	118 386	673 –2 120 +1	–0.45 +0.2	13.42	1 373 000 2.83	1 550 000 4.35
Последовательная комбинация	118 679	695 –0 125 +4	–0.26 +0.26	7.59	1 367 000 2.84	1 592 000 4.23

Табл. 5.5. Результаты сопоставления различных методов верификации композиции требований (в графе *Unsafe* указаны изменения числа реальных ошибок относительно последовательной верификации).

При сравнении результата верификации наилучший показатель демонстрирует последовательная комбинация, которая теряет 311 вердиктов (или 0.26%) и при этом получает 315 новых вердиктов (или 0.26%). Помимо этого, последовательная комбинация методов позволяет найти наибольшее количество нарушений требований (больше базового метода), чему способствует комбинирование сильных сторон методов УМАВ (эвристики и оптимизации для раннего предотвращения экспоненциального роста числа состояний) и ДАС (более эффективная автоматная спецификация и автоматическое разбиение

спецификации). Метод ДАС теряет незначительно больше вердиктов – 0.45% (из-за накладных расходов), а метод УМАВ теряет наибольший процент вердиктов – 1.36%. Разбиение спецификации на две группы требований помогло незначительно сократить потери метода УМАВ.

Наивысшее ускорение решения задач достижимости демонстрирует метод УМАВ с разбиением спецификации на 2 группы. Однако стоит отметить, что данный метод требует дополнительных знаний о решаемых задачах для осуществления данного разбиения, кроме того, данное разбиение имеет смысл только для решаемых задач (подробнее об идеях подобного разбиения см. Приложение Б). Подобный метод может успешно использоваться совместно с регрессионной верификацией (т. е. для примерно одного набора задач с известными характеристиками). Без подобных разбиений наибольшее ускорение демонстрирует метод УМАВ. Метод ДАС и последовательная комбинация методов требуют несколько больше ресурсов для решения задач.

Общее время процесса верификации оказалось наименьшим для метода УМАВ с разбиением спецификации на 2 группы требований (почти в 5 раз быстрее базового метода) даже несмотря на то, что подготавливалось в 2 раза больше задач достижимости. Время подготовки задач достижимости минимальное в методе ДАС (в 16 раз быстрее базового метода).

Среднее ускорение наивысшее для метода ДАС (за счет использования более эффективных автоматных спецификаций), а метод УМАВ с разбиением спецификации уступает остальным (поскольку большинство задач не требует разбиения на 2 группы). Сопоставление числа наиболее быстрых задач для предложенных методов представлено на рисунке 5.5[16].

Среди потерянных вердиктов у метода УМАВ имеются 9 реальных ошибок (т. е. 7.5%), у метода ДАС – 2 реальных ошибки (1.7%), у последовательной

16 В данном графике отсутствует последовательная комбинация, поскольку ее график для 90% задач совпадает с графиком метода УМАВ.

комбинации методов – 0. При этом метод УМАВ с разбиением спецификации существенно улучшает данный показатель до 2 потерянных ошибок. Метод УМАВ за счет новых успешно решенных задач в обоих случаях находит еще 2 реальных ошибки, метод ДАС – 1 реальную ошибку, а последовательная комбинация методов – 4 реальных ошибки.

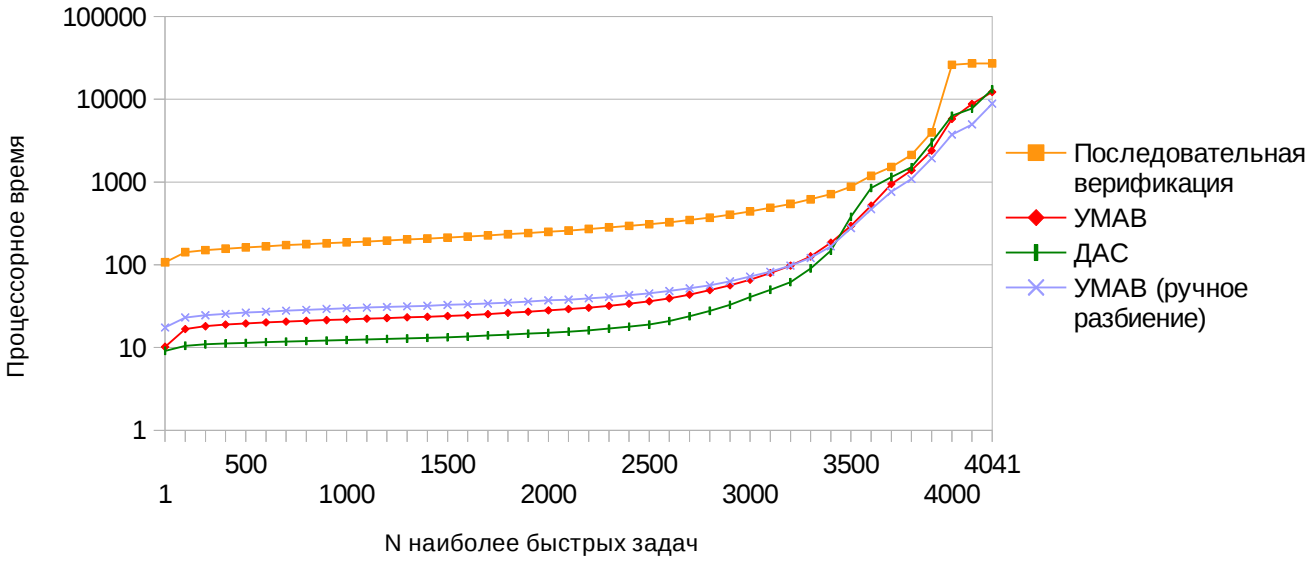

Рис. 5.5. Сопоставление числа наиболее быстрых задач для предложенных методов.

Таким образом, предложенные методы верификации композиции требований демонстрируют сопоставимые результаты. Для получения точного результата наиболее подходит последовательная комбинации методов УМАВ и ДАС. Наиболее сократить требуемые ресурсы удается методу УМАВ, особенно при проведении правильной декомпозиции всей спецификации на группы требований. Метод ДАС демонстрирует наивысшее среднее ускорение (т. е. лучше подходит для решения отдельных задач достижимости) и быстрее других справляется с подготовкой задач достижимости.

5.9. Зависимость результата и ускорения от числа требований

Важной характеристикой методов верификации композиции требований

также является зависимость результата и ускорения верификации от числа проверяемых требований. С одной стороны, в идеале ускорение верификации должно увеличиваться пропорционально числу проверяемых требований за счет большего переиспользования в сравнении с последовательной верификацией. С другой стороны, инструментирование и эвристики в методе УМАВ и накладные расходы в ДАС могут привести к сокращению эффективности методов с ростом числа требований. Для оценки подобных зависимостей все требования были отсортированы случайным образом и последовательно брались для экспериментов, в которых для каждого метода использовалась конфигурация из предыдущего эксперимента.

Для искусственного увеличения проверяемых требований для экспериментов каждое требование было разбито по возможным типам нарушений на несколько требований, в каждом из которых добавлялись только вспомогательные проверки (т. е. конструкции *assert* в инструментировании и переходы в состояние *ERROR* в автоматах), вызывавшие данное нарушение требования. Например, требования корректного использования ресурсов могут иметь два нарушения – двойное освобождение ресурса и утечку ресурса, поэтому подобные требования можно разбить на два, каждое из которых будет проверять только один тип нарушений. Подобное разбиение требований позволило увеличить их число до 88 (подробно все типы нарушений перечислены в Приложении А). Стоит заметить, что на практике проверять каждый тип нарушения требования избыточно, поскольку типы нарушений одного требования, как правило, взаимосвязаны.

Полученная зависимость ускорения методов верификации композиции требований относительно последовательной верификации от числа требований представлена на рисунке 5.6, зависимость процента потерь от числа требований – на рисунке 5.7.

Метод УМАВ демонстрирует наивысшее ускорение, которое возрастает при верификации относительно большого числа требований, во многом благодаря используемым эвристикам и оптимизациям. При этом число потерь метода УМАВ, хотя и превосходит все остальные методы, но является относительно небольшим (менее 2%) и стабильным при увеличении числа требований. Поэтому используемая конфигурация метода УМАВ является наиболее масштабируемой.

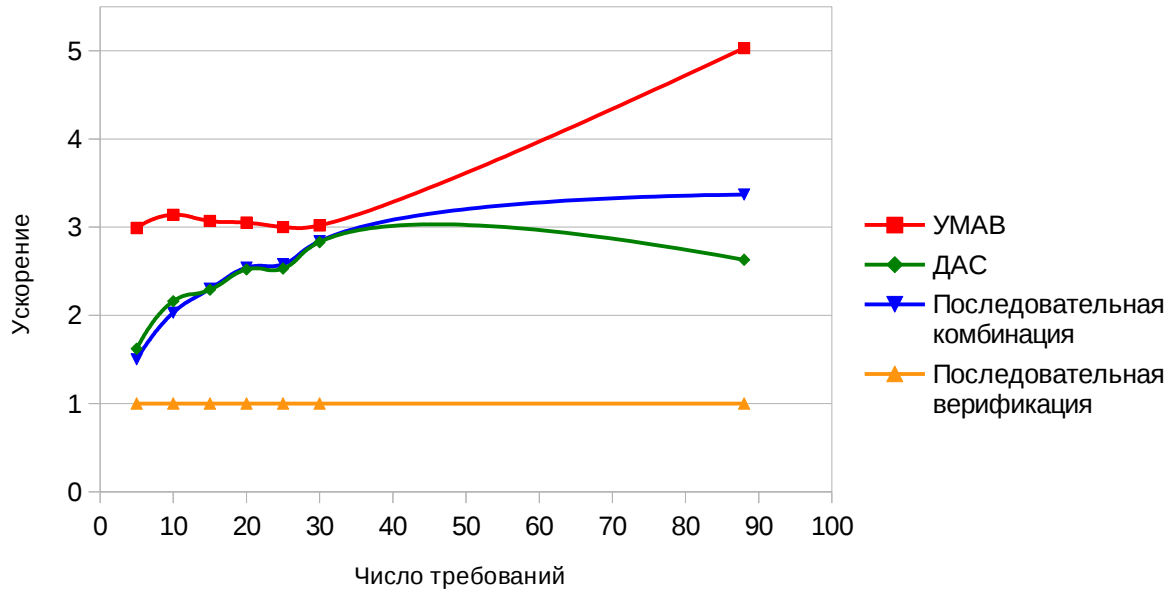

Рис. 5.6. Зависимость ускорения методов верификации композиции требований относительно базового метода последовательной верификации от числа требований.

Ускорение метода ДАС возрастает с ростом числа требований, однако для относительно большого числа требований эффективность метода снижается из-за накладных расходов, и его ускорение падает. Число потерь метода существенно меньше, чем в методе УМАВ, для относительно небольшого числа требований (до 30), однако при значительном увеличении числа требований оно возрастает и превосходит потери метода УМАВ.

Последовательная комбинация методов УМАВ и ДАС демонстрирует минимальные потери для любого числа требований. Для относительно

небольшого числа требований (до 15) данный метод является более медленным, однако для большего числа требований он демонстрирует существенно большее ускорение, чем метод ДАС.

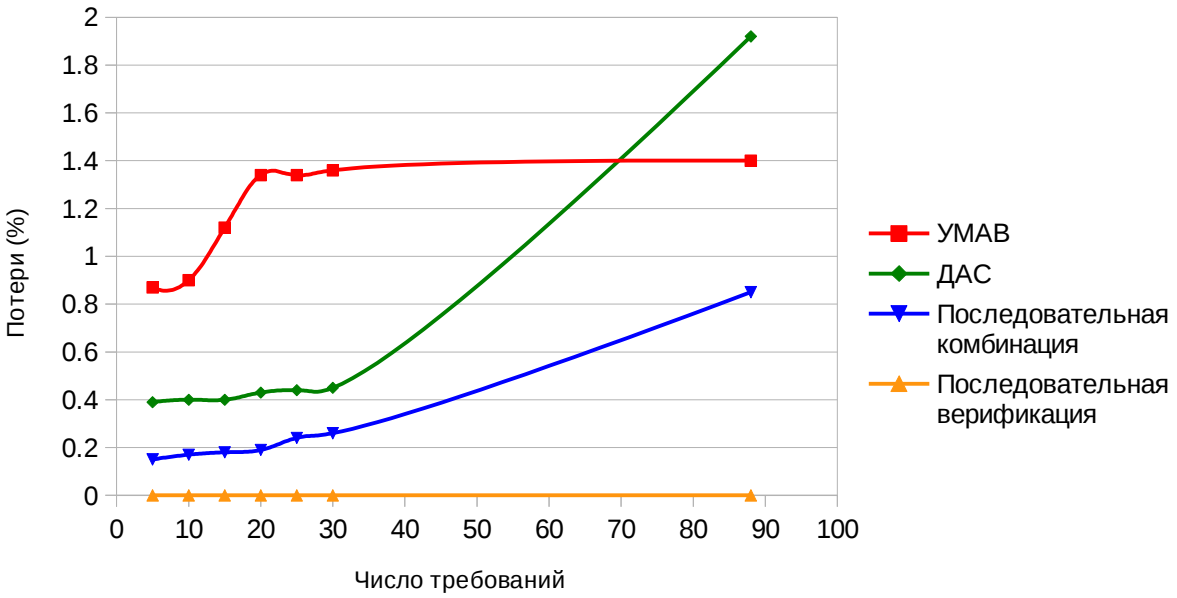

Рис. 5.7. Зависимость процента потерь методов верификации композиции требований относительно базового метода последовательной верификации от числа требований.

Таким образом, проведенный эксперимент показывает, что предложенные методы являются достаточно масштабируемыми.

5.10. Выводы

Проведенные оценки характеристик предложенных методов показывают, что все методы приводят к сокращению времени верификации. Выбор того или иного метода или их комбинации зависит от конкретных условий процесса верификации.

Метод УМАВ позволяет существенно повысить производительность верификации композиции требований с незначительным ухудшением результата. В рамках подхода CEGAR данный метод является наиболее производительным и масштабируемым. При проверке относительно большого числа требований разбиение проверяемой спецификации на группы требований позволяет улучшить

характеристики метода для решаемых задач, что особенно полезно в регрессионной верификации.

Метод ОВН способен находить больше реальных ошибок в программах, чем последовательная верификация, однако для этого требуется больше ресурсов (как на саму верификацию, так и на ручной анализ результата). Метод УМАВ с ОВН способен решать ту же задачу для композиции требований существенно быстрее метода ОВН с незначительным ухудшением результата.

Метод АС предлагает новый способ формализации требований, который в перспективе может стать полноценной заменой инструментированию. Данный метод может использоваться в последовательной верификации или в методе ОВН.

Метод ДАС является расширением метода УМАВ за рамки подхода CEGAR на основе метода АС. Данный метод является перспективным, поскольку может использоваться в любом подходе статической верификации и предоставляет удобный интерфейс для создания новых алгоритмов верификации композиции требований.

Для пользователей системы верификации LDV Tools рекомендуется использовать последовательную комбинацию методов УМАВ и ДАС, которая объединяет их преимущества с учетом решаемых задач в рамках подхода CEGAR и нацелена на минимизацию потерь результата при достаточно эффективном решении задач.

Таким образом, предложенные методы верификации композиции требований (УМАВ и ДАС) повышают производительность верификации в 4-5 раз при незначительных потерях результата (примерно 1%) относительно базового метода верификации, а метод ОВН способен выявлять в 1.5 раза больше реальных ошибок при увеличении вычислительных ресурсов примерно на 10%, при этом метод требует проведения ручного анализа результатов.

Заключение

Основные результаты диссертационной работы, выносимые на защиту, состоят в следующем:

- Разработаны методы статической верификации программного обеспечения, основанные на инструментировании исходного кода и предназначенные для обнаружения всех однотипных нарушений и проверки выполнения композиции требований с помощью условной многоаспектной верификации.
- Разработаны методы статической верификации программного обеспечения, с использованием формализации требований в виде автоматных спецификаций и декомпозиции автоматной спецификации на группы требований для совместной верификации.
- Доказана полнота и корректность предложенных методов для требований, удовлетворяющих ограничениям инструментирования исходного кода программы.

Возможные направления дальнейших исследований включают в себя:

- Расширение метода автоматных спецификаций для поддержки большего класса моделей требований (например, за счет полноценной поддержки указателей).
- Разработка новых стратегий разбиения в методе декомпозиции автоматной спецификации, способных повысить эффективность метода.

Предложенные методы были реализованы в рамках проектов отдела Технологий программирования Института системного программирования РАН при непосредственном участии автора данной работы, проведенные эксперименты показали их широкий потенциал в повышении производительности статической верификации программного обеспечения композиции требований. Автор выражает свою признательность всем участникам данного проекта.

Список сокращений и условных обозначений

CEGAR – counterexample guided abstraction refinement.

SV-COMP – competitions on software verification.

LDV Tools – Linux driver verification tools.

BMC – bounded model checking.

ГПУ – граф потока управления.

АГД – абстрактный граф достижимости.

CPA – configurable program analysis.

ОВН – обнаружение всех (однотипных) нарушений.

МАВ – многоаспектная верификация.

ППУ – последнее проверяемое утверждение.

ВЛУ – внутренний лимит утверждения.

УМАВ – условная многоаспектная верификация.

АС – автоматные спецификации.

ДАС – декомпозиция автоматной спецификации.

ВЛПИ – внутренний лимит пустого интервала.

ВЛИ – внутренний лимит интервала.

ВЛНИ – внутренний лимит начального интервала.

ЛО – лист ожидания.

Список используемой литературы

[1] Dershowitz N. Software horror stories. [Электронный ресурс] // Режим доступа: https://www.cs.tau.ac.il/~nachumd/horror.html, свободный.

[2] The Economic Impacts of Inadequate Infrastructure for Software Testing. NIST Report. 2002. [Электронный ресурс] // Режим доступа: https://www.nist.gov/sites/default/files/documents/director/planning/report02-3.pdf, свободный.

[3] Levenson N., Turner. C. S. An Investigation of the Therac-25 Accidents. [Электронный ресурс] // Режим доступа: https://www.cs.umd.edu/class/spring2003/cmsc838p/Misc/therac.pdf, свободный.

[4] Sagdeev R. Z., Zakharov A. V. Brief history of the Phobos mission // Nature, vol. 341, pp. 581-585, 1989.

[5] Lewis G. N., Fetter S., Gronlund L. Why were Scud casualties so low? // Nature, vol. 361, pp. 293-296, 1993.

[6] Mars Climate Orbiter Mishap Investigation Board Phase I Report. 1999. [Электронный ресурс] // Режим доступа: ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf, свободный.

[7] Beyer D., Petrenko A. Linux Driver Verification // Proceedings of ISoLA. LNCS, vol. 7610, pp. 1-6, 2012.

[8] Corbet J., Kroah-Hartman G., McPherson A. Linux kernel development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. [Электронный ресурс] // Режим доступа: https://www.linux.com/publications/linux-kernel-development-how-fast-it-going-who-doing-it-what-they-are-doing-and-who-5, свободный, 2016.

[9] Chou A., Yang J., Chelf B., Hallem S., Engler D. An Empirical Study of

Operating System Errors // Proceedings of 18th ACM Symposium on Operating Systems Principles (SOSP), pp. 73-88, 2001.

[10] Mathur A. P. Foundations of Software Testing. [Электронный ресурс] // Режим доступа: https://www.cs.purdue.edu/homes/apm/FoundationsBookSecondEdition/Slides/ConsolidatedSlides.pdf, свободный, 2006.

[11] Emanuelsson P., Nilsson U. A Comparative Study of Industrial Static Analysis Tools // Proceedings of the 3rd International Workshop on Systems Software Verification. ENTCS, vol. 217, pp. 5-21, 2008.

[12] Beyer D. Competition on Software Verification // Proceedings of TACAS. LNCS, vol. 7214, pp. 504-524, 2012.

[13] Beyer D. Second Competition on Software Verification (Summary of SV-COMP 2013) // Proceedings of TACAS. LNCS, vol. 7795, pp. 594-609, 2013.

[14] Beyer D. Status Report on Software Verification (Competition Summary SV-COMP 2014) // Proceedings of TACAS. LNCS, vol. 8413, pp. 373-388, 2014.

[15] Beyer D. Software Verification and Verifiable Witnesses // Proceedings of TACAS. LNCS, vol. 9035, pp. 401-416, 2015.

[16] Beyer D. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016) // Proceedings of TACAS. LNCS, vol. 9636, pp. 887-904, 2016.

[17] Clarke E., Grumberg O., Jha S., Lu Y., Veith H. Counterexample-guided abstraction refinement // Proceedings of CAV, LNCS, vol. 1855, pp. 154-169, 2000.

[18] Мандрыкин М. У., Мутилин В. С., Хорошилов А. В. Введение в метод CEGAR – уточнение абстракции по контрпримерам // Труды Института системного программирования РАН, т. 24, стр. 219-292, 2013.

[19] Ball T., Rajamani S. K. The Slam project. Debugging system software via static analysis // Proceedings of Symposium on Principles of Programming Languages

(POPL), pp. 1–3, 2002.

[20] Ball T., Bounimova E., Levin V., Kumar R., Lichtenberg J. The Static Driver Verifier Research Platform // Proceedings of Computer Aided Verification. LNCS, vol. 6174, pp. 119-122, 2010.

[21] Мутилин В. С., Новиков Е. М., Хорошилов А. В. Анализ типовых ошибок в драйверах операционной системы Linux // Труды Института системного программирования РАН, т. 22, с. 349-374, 2012.

[22] Ball T., Levin V., Rajamani S. K. A decade of software model checking with SLAM. Communications of the ACM, vol. 54, issue 7, pp. 68-76, 2011.

[23] Мутилин В. С., Новиков Е. М., Страх А. В., Хорошилов А. В., Швед П. Е. Архитектура Linux Driver Verification // Труды Института системного программирования РАН, т. 20, с. 163-187, 2011.

[24] Khoroshilov A., Mutilin V., Novikov E., Shved P., Strakh A. Towards an open framework for C verification tools benchmarking. Perspectives of Systems Informatics. LNCS, vol. 7162, pp. 179-192, 2012.

[25] Захаров И. С., Мандрыкин М. У., Мутилин В. С., Новиков Е. М., Петренко А. К., Хорошилов А. В. Конфигурируемая система статической верификации модулей ядра операционных систем // Программирование, т. 1, с. 49-64, 2015.

[26] Открытая система верификации модулей ядра Linux LDV Tools [Электронный ресурс] // Режим доступа: http://linuxtesting.org/ldv, свободный.

[27] Shved P., Mandrykin M., Mutilin V. Predicate analysis with BLAST 2.7 // Proceedings of TACAS. LNCS, vol. 7214, pp. 525-527, 2012.

[28] Beyer D., Henzinger T., Jhala R., Majumdar R. The software model checker BLAST // International Journal on Software Tools for Technology Transfer, vol. 9, issue 5, pp. 505-525, 2007.

[29] Beyer D., Keremoglu M. CPAchecker: A tool for configurable software verification // Proceedings of Computer Aided Verification. LNCS, vol. 6806, pp. 184–190, 2011.

[30] Открытый инструмент статической верификации CPAchecker [Электронный ресурс] // Режим доступа: https://cpachecker.sosy-lab.org, свободный.

[31] Мордань В. О. Многоаспектная верификация модулей ядра операционной системы Linux // Материалы XXI Международной молодежной научной конференции студентов, аспирантов и молодых ученых «Ломоносов», с. 122-123, 2014.

[32] Mordan V., Novikov E. Minimizing the number of static verifier traces to reduce time for finding bugs in Linux kernel modules // Proceedings of 8th Spring/Summer Young Researchers Colloquium on Software Engineering, vol. 1, 2014.

[33] Mordan V., Mutilin V. Checking several requirements at once with CEGAR // Perspectives of Systems Informatics. LNCS, vol. 9609, pp. 218-232, 2016.

[34] Мордань В. О., Мутилин В. С. Проверка нескольких требований за один запуск инструмента статической верификации с помощью CEGAR // Программирование, т. 4. с. 225-238, 2016.

[35] Apel S., Beyer D., Mordan V., Mutilin V., Stahlbauer A. On-The-Fly Decomposition of Specifications in Software Model Checking // Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 349-361, 2016.

[36] Biere A., Cimatti A, Clarke E., Strichman O., Zhu Y. Bounded model checking // Advances in Computers, vol. 58, pp. 117-148, 2003.

[37] Ball T., Bounimova E., Cook B., Levin V., Lichtenberg J., McGarvey C., Ondrusek B., Rajamani S. K., Ustuner A. Thorough static analysis of device drivers // Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on

Computer Systems. ACM, vol. 40, issue 4, pp. 73-85, 2006.

[38] Мандрыкин М. У., Мутилин В. С., Новиков Е. М., Хорошилов А. В. Обзор инструментов статической верификации Си программ в применении к драйверам устройств операционной системы Linux // Труды Института системного программирования РАН, т. 22, стр. 293-326, 2012.

[39] Beyer D., Cimatti A., Griggio A., Keremoglu M. E., Sebastiani R. Software Model Checking via Large-Block Encoding // Proceedings of Formal Methods in Computer-Aided Design (FMCAD), pp. 25–32, 2009.

[40] Beyer D., Keremoglu M., Wendler P. Predicate abstraction with adjustable-block encoding // Proceedings of Formal Methods in Computer-Aided Design, pp. 189-198, 2010.

[41] Beyer D., Löwe S. Explicit-state software model checking based on CEGAR and interpolation // Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013). LNCS, vol. 7793, pp. 146-162, 2013.

[42] Barrett C., Deters M., de Moura L., Oliveras A., Stump A. 6 years of SMT-COMP // Journal of Automated Reasoning, vol. 50, issue 3, pp. 243-277, 2013.

[43] Beyer D., Löwe S., Novikov E., Stahlbauer A., Wendler P. Precision Reuse for Efficient Regression Verification // Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE 2013), pp. 389-399, 2013.

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

[45] Craig W. Linear reasoning // Symbolic Logic, vol. 22, pp. 250-268, 1957.

[46] Turing A. M. On Computable numbers, with an application to the Entscheidungsproblem // Proceedings of the London Mathematical Society.

pp. 230-265, 1936.

[47] Новиков Е. М. Развитие метода контрактных спецификаций для верификации модулей ядра операционной системы Linux. Диссертация на соискание ученой степени к.ф.-м.н., Москва, 2013.

[48] Ball T., Rajamani S. K. SLIC: A specification language for interface checking (of C) // Technical Report MSR-TR-2001-21, Microsoft Research, 2002.

[49] Beyer D., Chlipala A., Henzinger T. A., Jhala R., Majumdar R. The BLAST query language for software verification // Proceedings of SAS. LNCS, vol. 3148, pp. 2-18, 2004.

[50] Šerý O. Enhanced property specification and verification in BLAST // Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE 2009). LNCS, vol. 5503, pp. 456-469, 2009.

[51] Nori A. V., Rajamani S. K., Tetali S., Thakur A. V. The Yogi Project: Software Property Checking via Static Analysis and Testing // Proceedings of TACAS. LNCS, vol. 5505, pp. 178-181, 2009.

[52] Список ошибок, выявленных в модулях ядра Linux с помощью системы LDV Tools [Электронный ресурс] // Режим доступа: http://linuxtesting.org/results/ldv, свободный.

[53] Захаров И. С., Мутилин В. С., Новиков Е. М., Хорошилов А. В. Метод генерации модели окружения драйверов устройств операционной системы Linux // Труды Института системного программирования РАН, т. 25, 2013.

[54] Clarke E., Kroening D., Lerda F. A tool for checking ANSI-C programs // Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 2988, pp. 168-176, 2004.

[55] Albarghouthi A., Li Y., Gurfinkel A., Chechik M. UFO: a framework for abstraction and interpolation-based software verification // Proceedings of the 24th International Conference on Computer Aided Verification, pp. 672-678,

2012.

[56] Новиков Е. М. Упрощение анализа трасс ошибок инструментов статического анализа кода // Сборник научных трудов научно-практической конференции «Актуальные проблемы программной инженерии», стр. 215-221, 2011.

[57] Beyer D., Henzinger T. A., Theoduloz G. Configurable software verification: Concretizing the convergence of model checking and program analysis // Proceedings of CAV. LNCS, vol. 4590, pp. 504-518, 2007.

[58] Beyer D., Henzinger T. A., Keremoglu M. E., Wendler P. Conditional model checking: a technique to pass information between verifiers // Proceedings ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE 2012), article 57, 2012.

[59] Beyer D., Wendler P. Reuse of verification results // Proceedings of the 20th International Workshop on Model Checking Software (SPIN 2013). LNCS, vol. 7976, pp. 1-17, 2013.

[60] Visser W., Geldenhuys J., Dwyer M. B. Green: reducing, reusing, and recycling constraints in program analysis // Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE 2012), article 58, 2012.

[61] Păsăreanu C. S., Visser W., Bushnell D., Geldenhuys J., Mehlitz P., Rungta N. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis // Automated Software Engineering, vol. 20, issue 3, pp. 391–425, 2013.

[62] Barrett C., Tinelli C. CVC3 // Proceedings of the 19th International Conference on Computer Aided Verification (CAV’07). LNCS, vol. 4590, pp. 298-302, 2007.

[63] Открытый решатель Choco [Электронный ресурс] // Режим доступа: http://www.choco-solver.org, свободный.

[64] De Loera J. A., Hemmecke R., Tauzer J., Yoshida R. Effective Lattice Point

Counting in Rational Convex Polytopes // Symbolic Computation in Algebra and Geometry, vol. 38, issue 4, pp. 1273–1302, 2004.

[65] Guowei Y., Dwyer M. B., Rothermel G. Regression model checking // Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2009), pp. 115-124, 2009.

[66] Visser W., Mehlitz P. Model Checking Programs with Java PathFinder // Proceedings of 12th International SPIN Workshop. LNCS, vol. 3639, pp. 27-27, 2005.

[67] Sery O., Fedyukovich G., Sharygina N. Incremental upgrade checking by means of interpolation-based function summaries // Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD 2012), pp. 114-121, 2012.

[68] Beyer D., Holzer A., Tautschnig M., Veith H. Information reuse for multi-goal reachability analyses // Proceedings of the 22nd European Symposium on Programming Languages and Systems (ESOP 2013). LNCS, vol. 7792, pp. 472-491, 2013.

[69] Böhme M., Oliveira B. C. d. S., Roychoudhury A. Partition-based regression verification. In Proceedings of the 35th International Conference on Software Engineering (ICSE 2013), pp. 302-311, 2013.

[70] Chockler H., Denaro G., Meijia L., Fedyukovich G., Hyvrinen A. E. J., Mariani L., Muhammad A., Oriol M., Rajan A., Sery O., Sharygina N., Tautschnig M. Pincette validating changes and upgrades in networked software // Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013), pp. 461-464, 2013.

[71] Christakis M., Müller P., Wüstholz V. Collaborative verification and testing with explicit assumptions // Proceedings of the 18th International Symposium on Formal Methods (FM 2012). LNCS, vol. 7436, pp. 132-146, 2012.

[72] Godefroid P., Nori A. V., Rajamani S. K., Tetali S. D. Compositional may-must

program analysis: unleashing the power of alternation // Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010), pp. 43-56, 2010.

[73] Bradley A. R. Sat-based model checking without unrolling // Proceedings of VMCAI. LNCS, vol. 6538, pp. 70-87, 2011.

[74] Chockler H., Ivrii A., Matsliah A., Moran S., Nevo Z. Incremental formal verification of hardware // Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD 2011), pp. 135-143, 2011.

[75] Khasidashvili Z., Nadel A., Palti A., Hanna Z. Simultaneous SAT-based model checking of safety properties // Proceedings of HAV. LNCS, vol. 3875, pp. 56-75, 2005.

[76] Muller D. E. Infinite sequences and finite machines // Proceedings of SWCT, pp. 3-16, 1963.

[77] Dams D., Namjoshi K. S. Orion: High-precision methods for static error analysis of C and C++ programs // Proceedings of FMCO. LNCS, vol. 4111, pp. 138-160, 2005.

Свидетельства о государственной регистрации программы для ЭВМ

[1] Мордань В.О. «Программный компонент для выявления нескольких ошибок в программном обеспечении». Свидетельство о государственной регистрации программы для ЭВМ № 2016616600 от 15.06.2016.

[2] Мордань В.О. «Программный компонент для проверки нескольких правил корректности за один запуск инструмента статической верификации». Свидетельство о государственной регистрации программы для ЭВМ № 2016616661 от 16.06.2016.

Приложение А
Описание проверяемых требований

В данном приложении описаны 30 требований системы LDV Tools, которые были формализованы с помощью автоматных спецификаций. Большинство из данных требований были предложены и формализованы с помощью аспектно-ориентированного расширения языка C в диссертации [47].

Для каждого требования приведен его уникальный идентификатор в системе LDV Tools, краткое описание, список типов нарушений и соответствующая автоматная спецификация. Всего различных типов нарушений 88 (использовались в эксперименте из п. 5.9). Каждый тип нарушения в автоматах соответствует переходу в состояние ERROR с соответствующим идентификатором, поэтому для проверки только одного типа нарушения необходимо удалить все переходы в состояние ERROR с другими идентификаторами.

A.1. Требование linux:module

Описание: проверка корректного использования блокировки от выгрузки модуля ядра.

Список типов нарушений:

- модулям запрещается разрешать выгрузку модулей, выгрузку которых они не блокировали (*"linux:module::less initial decrement"*);
- на момент завершения работы модули должны разрешить выгрузку всех модулей, для которых она была запрещена (*"linux:module::more initial at exit"*).

Автомат:

```
OBSERVER AUTOMATON linux_module
INITIAL STATE Init;

STATE USEALL Init :
  MATCH ENTRY -> ENCODE {int module_state = 0;} GOTO Init;
  MATCH CALL {__module_get($1)} -> ASSUME {((struct module *)$1) != 0}
```

```
    ENCODE {module_state=1;} GOTO Inc;
  MATCH CALL {__module_get($1)} -> ASSUME {((struct module *)$1) == 0} GOTO Init;
  MATCH RETURN {$1=try_module_get($2)} -> ASSUME {((int)$1)!=0;
    ((struct module *)$2)!=0} ENCODE {module_state=1;} GOTO Inc;
  MATCH RETURN {$1=try_module_get($2)} -> ASSUME {((int)$1)==0;
    ((struct module *)$2)!=0} GOTO Init;
  MATCH RETURN {$1=try_module_get($2)} -> ASSUME {((int)$1)!=0;
    ((struct module *)$2)==0} GOTO Init;
  MATCH RETURN {$1=try_module_get($2)} -> ASSUME {((int)$1)==0;
    ((struct module *)$2)==0} GOTO Stop;
  MATCH CALL {module_put($1)} -> ASSUME {((struct module *)$1) == 0} GOTO Init;
  MATCH CALL {module_put($1)} -> ASSUME {((struct module *)$1) != 0}
    ERROR("linux:module::less initial decrement");
  MATCH CALL {module_put_and_exit($?)} ->
    ERROR("linux:module::less initial decrement");
  MATCH RETURN {$1 = module_refcount($?)} -> SPLIT {((int)$1)==0} GOTO Init
    NEGATION GOTO Stop;

STATE USEALL Inc :
  MATCH CALL {__module_get($1)} -> ASSUME {((struct module *)$1) != 0}
    ENCODE {module_state=module_state+1;} GOTO Inc;
  MATCH CALL {__module_get($1)} -> ASSUME {((struct module *)$1) == 0} GOTO Inc;
  MATCH RETURN {$1=try_module_get($2)} -> ASSUME {((int)$1)!=0;
    ((struct module *)$2)!=0} ENCODE {module_state=module_state+1;} GOTO Inc;
  MATCH RETURN {$1=try_module_get($2)} -> ASSUME {((int)$1)==0;
    ((struct module *)$2)!=0} GOTO Inc;
  MATCH RETURN {$1=try_module_get($2)} -> ASSUME {((int)$1)!=0;
    ((struct module *)$2)==0} GOTO Inc;
  MATCH RETURN {$1=try_module_get($2)} -> ASSUME {((int)$1)==0;
    ((struct module *)$2)==0} GOTO Stop;
  MATCH CALL {module_put($1)} -> ASSUME {((struct module *)$1) != 0;
    module_state >  1} ENCODE {module_state=module_state-1;} GOTO Inc;
  MATCH CALL {module_put($1)} -> ASSUME {((struct module *)$1) != 0;
    module_state <= 1} ENCODE {module_state=module_state-1;} GOTO Init;
  MATCH CALL {module_put($1)} -> ASSUME {((struct module *)$1) == 0} GOTO Inc;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:module::more initial at exit");
  MATCH CALL {module_put_and_exit($?)} -> GOTO Stop;
  MATCH RETURN {$1 = module_refcount($?)} -> SPLIT {((int)$1)==module_state}
    GOTO Inc NEGATION GOTO Stop;

STATE USEFIRST Stop :
  TRUE -> GOTO Stop;

END AUTOMATON
```

Примечание: функция *ldv_check_final_state()* вызывается при построении модели окружения в системе LDV Tools после завершения работы с модулем.

A.2. Требование linux:alloc:irq

Описание: проверка корректного выделения памяти в контексте прерывания.

Список типов нарушений:

- в контексте прерывания модулям запрещается выделять ресурсы

блокирующим образом, т. е. без использования флагов атомарного выделения, например *GFP_ATOMIC* (*"linux:alloc:irq::wrong flags"*);

- в контексте прерывания модулям запрещается использовать заведомо блокирующее выделение памяти (*"linux:alloc:irq::nonatomic"*).

Автомат:

```
OBSERVER AUTOMATON linux_alloc_irq
INITIAL STATE Process_context;

STATE USEALL Process_context :
  MATCH CALL {ldv_switch_to_interrupt_context($?)} -> GOTO Interrupt_context;

STATE USEALL Interrupt_context :
  MATCH CALL {ldv_switch_to_process_context($?)} -> GOTO Process_context;
  MATCH CALL {ldv_check_alloc_flags($1)} -> ASSUME {((int)$1)==32} GOTO Locked;
  MATCH CALL {ldv_check_alloc_flags($1)} -> ASSUME {((int)$1)!=32}
    ERROR("linux:alloc:irq::wrong flags");
  MATCH CALL {ldv_check_alloc_nonatomic($?)} ->
    ERROR("linux:alloc:irq::nonatomic");

END AUTOMATON
```

Примечание: функция *ldv_switch_to_interrupt_context()* используется при построении модели окружения в системе LDV Tools для перехода в атомарный контекст; функция *ldv_switch_to_process_context()* используется при построении модели окружения в системе LDV Tools для выхода из атомарного контекста; функция *ldv_check_alloc_flags(gfp_t flags)* добавляется системой LDV Tools перед каждым выделением памяти в модуле с флагом *flags*; функция *ldv_check_alloc_nonatomic())* добавляется системой LDV Tools перед каждым заведомо блокирующим выделением памяти в модуле (например, с помощью функции *void *vmalloc(unsigned long size)*).

A.3. Требование linux:gendisk

Описание: проверка корректного использования счетчика ссылок общих жестких дисков (*generic hard disk*).

Список типов нарушений:

- модулям запрещается использовать счетчик ссылок общих жестких дисков

до его создания (*"linux:gendisk::use before allocation"*);

- модулям запрещается удалять счетчик ссылок общих жестких дисков до его создания (*"linux:gendisk::free before allocation"*);
- модулям запрещается повторно создавать счетчик ссылок общих жестких дисков (*"linux:gendisk::double allocation"*);
- модулям запрещается уменьшать счетчик ссылок общих жестких дисков на большее значение, чем он был увеличен (*"linux:gendisk::delete before add"*);
- на момент завершения работы в модулях счетчик ссылок общих жестких дисков должен быть равен нулю (*"linux:gendisk::more initial at exit"*).

Автомат:

```
OBSERVER AUTOMATON linux_gendisk
INITIAL STATE Init;

STATE USEALL Init :
  MATCH RETURN {$1 = alloc_disk($?)} -> ASSUME {((struct gendisk *)$1) != 0}
    GOTO Allocated;
  MATCH RETURN {$1 = alloc_disk($?)} -> ASSUME {((struct gendisk *)$1) == 0}
    GOTO Init;
  MATCH CALL {add_disk($?)} -> ERROR("linux:gendisk::use before allocation");
  MATCH CALL {del_gendisk($?)} -> ERROR("linux:gendisk::delete before add");
  MATCH CALL {put_disk($1)} -> ASSUME {((struct gendisk *)$1) != 0}
    ERROR("linux:gendisk::free before allocation");
  MATCH CALL {put_disk($1)} -> ASSUME {((struct gendisk *)$1) == 0} GOTO Init;

STATE USEALL Allocated :
  MATCH RETURN {$1 = alloc_disk($?)} -> ERROR("linux:gendisk::double allocation");
  MATCH CALL {add_disk($?)} -> GOTO Added;
  MATCH CALL {del_gendisk($?)} -> ERROR("linux:gendisk::delete before add");
  MATCH CALL {put_disk($1)} -> ASSUME {((struct gendisk *)$1) != 0} GOTO Init;
  MATCH CALL {put_disk($1)} -> ASSUME {((struct gendisk *)$1) == 0}
    GOTO Allocated;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:gendisk::more initial at exit");

STATE USEALL Added :
  MATCH RETURN {$1 = alloc_disk($?)} -> ERROR("linux:gendisk::double allocation");
  MATCH CALL {add_disk($?)} -> ERROR("linux:gendisk::use before allocation");
  MATCH CALL {del_gendisk($?)} -> GOTO Allocated;
  MATCH CALL {put_disk($1)} -> ASSUME {((struct gendisk *)$1) != 0} GOTO Init;
  MATCH CALL {put_disk($1)} -> ASSUME {((struct gendisk *)$1) == 0} GOTO Added;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:gendisk::more initial at exit");

END AUTOMATON
```

A.4. Требование linux:blk:queue

Описание: проверка корректного использования очередей устройств блочного ввода-вывода (*block devices queue*).

Список типов нарушений:

- модулям запрещается освобождать незарезервированные ими очереди устройств блочного ввода-вывода (*"linux:blk:queue::no init"*);
- модулям запрещается повторно инициализировать очереди устройств блочного ввода-вывода с одинаковыми спин-блокировками (*"linux:blk:queue::double init"*);
- на момент завершения работы модули должны освободить все зарезервированные ими очереди устройств блочного ввода-вывода (*"linux:blk:queue::more initial at exit"*).

Автомат:

```
OBSERVER AUTOMATON linux_blk_queue
INITIAL STATE Zero;

STATE USEALL Zero :
  MATCH RETURN {$1=request_queue($?)} ->
    ASSUME {((struct request_queue *)$1) != 0} GOTO Got;
  MATCH RETURN {$1=request_queue($?)} ->
    ASSUME {((struct request_queue *)$1) == 0} GOTO Zero;
  MATCH CALL {blk_cleanup_queue($?)} ->
    ERROR("linux:blk:queue::no init");

STATE USEFIRST Got :
  MATCH RETURN {$1=request_queue($?)} ->
    ERROR("linux:blk:queue::double init");
  MATCH CALL {blk_cleanup_queue($?)} -> GOTO Zero;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:blk:queue::more initial at exit");

END AUTOMATON
```

A.5. Требование linux:mutex

Описание: проверка корректного использования мютексов в одном потоке.

Список типов нарушений:

- модулям запрещается повторно захватывать одни и те же мьютексы (*"linux:mutex::one thread:double lock"*);

- модулям запрещается пытаться захватить уже захваченные мьютексы с помощью *mutex_trylock* (*"linux:mutex::one thread:double lock try"*);
- модулям запрещается освобождать незахваченные ими мьютексы (*"linux:mutex::one thread:double unlock"*);
- на момент завершения работы модули должны освободить все захваченные ими мьютексы (*"linux:mutex::one thread:locked at exit"*).

Автомат:

```
// for arg_sign in mutex_arg_signs
OBSERVER AUTOMATON linux_mutex_as{{ arg_sign.id }}

INITIAL STATE Unlocked;

STATE USEALL Unlocked :
  MATCH CALL {mutex_lock{{ arg_sign.id }}($?)} -> GOTO Locked;
  MATCH RETURN {$1 = mutex_lock_interruptible_or_killable{{ arg_sign.id }}($?)} ->
    ASSUME {((int)$1) == 0} GOTO Locked;
  MATCH RETURN {$1 = mutex_lock_interruptible_or_killable{{ arg_sign.id }}($?)} ->
    ASSUME {((int)$1) <  0} GOTO Unlocked;
  MATCH RETURN {$1 = mutex_lock_interruptible_or_killable{{ arg_sign.id }}($?)} ->
    ASSUME {((int)$1) >  0} GOTO Stop;
  MATCH RETURN {$1 = mutex_trylock{{ arg_sign.id }}($?)} -> SPLIT {((int)$1)!=0}
    GOTO Locked NEGATION GOTO Unlocked;
  MATCH RETURN {$1 = atomic_dec_and_mutex_lock{{ arg_sign.id }}($?)} -> SPLIT
    {((int)$1)!=0} GOTO Locked NEGATION GOTO Unlocked;
  MATCH CALL {mutex_unlock{{ arg_sign.id }}($?)} ->
    ERROR("linux:mutex::one thread:double unlock");

STATE USEALL Locked :
  MATCH CALL {mutex_lock{{ arg_sign.id }}($?)} ->
    ERROR("linux:mutex::one thread:double lock");
  MATCH RETURN {$1 = mutex_lock_interruptible_or_killable{{ arg_sign.id }}($?)} ->
    ERROR("linux:mutex::one thread:double lock");
  MATCH RETURN {$1 = mutex_is_locked{{ arg_sign.id }}($?)} ->
    SPLIT {((int)$1) == 1} GOTO Locked NEGATION GOTO Stop;
  MATCH RETURN {$1 = mutex_trylock{{ arg_sign.id }}($?)}  ->
    ERROR("linux:mutex::one thread:double lock try");
  MATCH RETURN {$1 = atomic_dec_and_mutex_lock{{ arg_sign.id }}($?)} ->
    ERROR("linux:mutex::one thread:double lock");
  MATCH CALL {mutex_unlock{{ arg_sign.id }}($?)} ->  GOTO Unlocked;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:mutex::one thread:locked at exit");

STATE USEFIRST Stop :
  TRUE -> GOTO Stop;

END AUTOMATON
// endfor
```

Примечание: данные автоматы генерируются заменой шаблона

{{ arg_sign.id }} на используемые фактические параметры для конкретного модуля системой LDV Tools.

A.6. Требование linux:spinlock

Описание: проверка корректного использования спин-блокировок в одном потоке (*spin_lock*).

Список типов нарушений:

- модулям запрещается повторно захватывать одни и те же спин-блокировки (*"linux:spinlock::one thread:double lock"*);
- модулям запрещается пытаться захватить уже захваченные спин-блокировки с помощью *spin_trylock* (*"linux:spinlock::one thread:double lock try"*);
- модулям запрещается освобождать незахваченные ими спин-блокировки (*"linux:spinlock::one thread:double unlock"*);
- на момент завершения работы модули должны освободить все захваченные ими спин-блокировки (*"linux:spinlock::one thread:locked at exit"*).

Автомат:

```
// for arg_sign in spinlock_arg_signs
OBSERVER AUTOMATON linux_spinlock{{ arg_sign.id }}

INITIAL STATE Unlocked;

STATE USEALL Unlocked :
  MATCH CALL {spin_lock{{ arg_sign.id }}($?)} -> GOTO Locked;
  MATCH RETURN {$1 = spin_trylock{{ arg_sign.id }}($?)} -> SPLIT {((int)$1)!=0}
    GOTO Locked NEGATION GOTO Unlocked;
  MATCH RETURN {$1 = atomic_dec_and_lock{{ arg_sign.id }}($?)} ->
    SPLIT {((int)$1)!=0} GOTO Locked NEGATION GOTO Unlocked;
  MATCH CALL {spin_unlock{{ arg_sign.id }}($?)} ->
    ERROR("linux:spinlock::one thread:double unlock");

STATE USEALL Locked :
  MATCH CALL {spin_lock{{ arg_sign.id }}($?)} ->
    ERROR("linux:spinlock::one thread:double lock");
  MATCH RETURN {$1 = spin_is_locked{{ arg_sign.id }}($?)} ->
    SPLIT {((int)$1) == 1} GOTO Locked NEGATION GOTO Stop;
  MATCH RETURN {$1 = spin_can_lock{{ arg_sign.id }}($?)} -> SPLIT {((int)$1) == 0}
    GOTO Locked NEGATION GOTO Stop;
  MATCH RETURN {$1 = spin_trylock{{ arg_sign.id }}($?)} ->
    ERROR("linux:spinlock::one thread:double lock try");
  MATCH CALL {spin_unlock_wait{{ arg_sign.id }}($?)} ->
    ERROR("linux:spinlock::one thread:double lock try");
```

```
  MATCH RETURN {$1 = atomic_dec_and_lock{{ arg_sign.id }}($?)} ->
    ERROR("linux:spinlock::one thread:double lock try");
  MATCH CALL {spin_unlock{{ arg_sign.id }}($?)} ->  GOTO Unlocked;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:spinlock::one thread:locked at exit");

STATE USEFIRST Stop :
  TRUE -> GOTO Stop;

END AUTOMATON
// endfor
```

A.7. Требование linux:alloc:spin lock

Описание: проверка корректного выделения памяти при захваченной спин-блокировке (*spin_lock*).

Список типов нарушений:

- при захваченной спин-блокировке модулям запрещается выделять ресурсы блокирующим образом, т. е. без использования флагов атомарного выделения, например *GFP_ATOMIC* (*"linux:alloc:spin lock::wrong flags"*);
- при захваченной спин-блокировке модулям запрещается использовать заведомо блокирующее выделение памяти (*"linux:alloc:spin lock::nonatomic"*).

Автомат:

```
// for arg_sign in spinlock_arg_signs
OBSERVER AUTOMATON linux_alloc_spinlock{{ arg_sign.id }}

INITIAL STATE Unlocked;

STATE USEALL Unlocked :
  MATCH CALL {spin_lock{{ arg_sign.id }}($?)} -> GOTO Locked;
  MATCH RETURN {$1 = spin_trylock{{ arg_sign.id }}($?)} -> SPLIT {((int)$1)!=0}
    GOTO Locked NEGATION GOTO Unlocked;
  MATCH RETURN {$1 = atomic_dec_and_lock{{ arg_sign.id }}($?)} ->
    SPLIT {((int)$1)!=0} GOTO Locked NEGATION GOTO Unlocked;
  MATCH CALL {spin_unlock{{ arg_sign.id }}($?)} -> GOTO Stop;

STATE USEALL Locked :
  MATCH CALL {spin_lock{{ arg_sign.id }}($?)} -> GOTO Stop;
  MATCH RETURN {$1 = spin_is_locked{{ arg_sign.id }}($?)} ->
    SPLIT {((int)$1) == 1} GOTO Locked NEGATION GOTO Stop;
  MATCH RETURN {$1 = spin_can_lock{{ arg_sign.id }}($?)} -> SPLIT {((int)$1) == 0}
    GOTO Locked NEGATION GOTO Stop;
  MATCH RETURN {$1 = spin_trylock{{ arg_sign.id }}($?)} -> GOTO Stop;
  MATCH CALL {spin_unlock_wait{{ arg_sign.id }}($?)} -> GOTO Stop;
  MATCH RETURN {$1 = atomic_dec_and_lock{{ arg_sign.id }}($?)} -> GOTO Stop;
```

```
  MATCH CALL {spin_unlock{{ arg_sign.id }}($?)} ->  GOTO Unlocked;

  MATCH CALL {ldv_check_alloc_flags($1)} -> ASSUME {((int)$1)==32} GOTO Locked;
  MATCH CALL {ldv_check_alloc_flags($1)} -> ASSUME {((int)$1)==0} GOTO Locked;
  MATCH CALL {ldv_check_alloc_flags($1)} -> ASSUME {((int)$1)!=0; ((int)$1)!=32}
    ERROR("linux:alloc:spin lock::wrong flags");
  MATCH CALL {ldv_check_alloc_nonatomic($?)} ->
    ERROR("linux:alloc:spin lock::nonatomic");

STATE USEFIRST Stop :
  TRUE -> GOTO Stop;

END AUTOMATON
// endfor
```

A.8. Требование linux:usb:urb

Описание: проверка корректного использования ресурсов вида *URB* (*USB request buffer*).

Список типов нарушений:

- модулям запрещается освобождать невыделенные ими URB (*"linux:usb:urb::less initial decrement"*);
- на момент завершения работы модули должны освободить все выделенные ими URB (*"linux:usb:urb::more initial at exit"*).

Автомат:

```
OBSERVER AUTOMATON linux_usb_urb
INITIAL STATE Init;

STATE USEALL Init :
  MATCH ENTRY -> ENCODE {int urb_state = 0;} GOTO Init;
  MATCH RETURN {$1=usb_alloc_urb($?)} -> ASSUME {((void *)$1) != ((void *)0)}
    ENCODE {urb_state=urb_state+1;} GOTO Inc;
  MATCH RETURN {$1=usb_alloc_urb($?)} -> ASSUME {((void *)$1) == ((void *)0)}
    GOTO Init;
  MATCH CALL {usb_free_urb($1)} -> ASSUME {((void *)$1) == ((void *)0)} GOTO Init;
  MATCH CALL {usb_free_urb($1)} -> ASSUME {((void *)$1) != ((void *)0)}
    ERROR("linux:usb:urb::less initial decrement");

STATE USEALL Inc :
  MATCH RETURN {$1=usb_alloc_urb($?)} -> ASSUME {((void *)$1) != ((void *)0)}
    ENCODE {urb_state=urb_state+1;} GOTO Inc;
  MATCH RETURN {$1=usb_alloc_urb($?)} -> ASSUME {((void *)$1) == ((void *)0)}
    GOTO Inc;
  MATCH CALL {usb_free_urb($1)} -> ASSUME {((void *)$1) != ((void *)0);
    urb_state > 1} ENCODE {urb_state=urb_state-1;} GOTO Inc;
  MATCH CALL {usb_free_urb($1)} -> ASSUME {((void *)$1) != ((void *)0);
    urb_state <= 1} ENCODE {urb_state=urb_state-1;} GOTO Init;
  MATCH CALL {usb_free_urb($1)} -> ASSUME {((void *)$1) == ((void *)0)} GOTO Inc;
  MATCH CALL {ldv_check_final_state($?)} ->
```

```
    ERROR("linux:usb:urb::more initial at exit");

END AUTOMATON
```

A.9. Требование linux:usb:coherent

Описание: проверка корректного использования ресурсов вида *DMA-буфер USB-устройства*.

Список типов нарушений:

- модулям запрещается освобождать невыделенные ими DMA-буферы USB-устройств (*"linux:usb:coherent::less initial decrement"*);
- на момент завершения работы модули должны освободить все выделенные ими DMA-буферы USB-устройств (*"linux:usb:coherent::more initial at exit"*).

Автомат:

```
OBSERVER AUTOMATON linux_usb_coherent
INITIAL STATE Init;

STATE USEALL Init :
  MATCH ENTRY -> ENCODE {int coherent_state = 0;} GOTO Init;
  MATCH RETURN {$1=usb_alloc_coherent($?)} -> ASSUME {((void *)$1) != ((void *)0)}
    ENCODE {coherent_state=coherent_state+1;} GOTO Inc;
  MATCH RETURN {$1=usb_alloc_coherent($?)} -> ASSUME {((void *)$1) == ((void *)0)}
    GOTO Init;
  MATCH CALL {usb_free_coherent($1)} -> ASSUME {((void *)$1) == ((void *)0)}
    GOTO Init;
  MATCH CALL {usb_free_coherent($1)} -> ASSUME {((void *)$1) != ((void *)0)}
    ERROR("linux:usb:coherent::less initial decrement");

STATE USEALL Inc :
  MATCH RETURN {$1=usb_alloc_coherent($?)} -> ASSUME {((void *)$1) != ((void *)0)}
    ENCODE {coherent_state=coherent_state+1;} GOTO Inc;
  MATCH RETURN {$1=usb_alloc_coherent($?)} -> ASSUME {((void *)$1) == ((void *)0)}
    GOTO Inc;
  MATCH CALL {usb_free_coherent($1)} -> ASSUME {((void *)$1) != ((void *)0);
    coherent_state > 1;} ENCODE {coherent_state=coherent_state-1;} GOTO Inc;
  MATCH CALL {usb_free_coherent($1)} -> ASSUME {((void *)$1) != ((void *)0);
    coherent_state <= 1;} ENCODE {coherent_state=coherent_state-1;} GOTO Init;
  MATCH CALL {usb_free_coherent($1)} -> ASSUME {((void *)$1) == ((void *)0)}
    GOTO Inc;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:usb:coherent::more initial at exit");

END AUTOMATON
```

A.10. Требование linux:alloc:usb lock

Описание: проверка корректного выделения памяти при захваченной

блокировке *USB-устройств* (*USB device locking*).

Список типов нарушений:

- при захваченной блокировке USB-устройств модулям запрещается выделять ресурсы блокирующим образом, т. е. без использования флагов атомарного выделения, например GFP_ATOMIC (*"linux:alloc:usb lock::wrong flags"*);
- при захваченной блокировке USB-устройств модулям запрещается использовать заведомо блокирующее выделение памяти (*"linux:alloc:usb lock::nonatomic"*).

Автомат:

```
OBSERVER AUTOMATON linux_alloc_usblock
INITIAL STATE Unlocked;

STATE USEALL Unlocked :
  MATCH CALL {usb_lock_device($?)} -> GOTO Locked;
  MATCH RETURN {$1=usb_trylock_device($?)} -> SPLIT {((int)$1)!=0} GOTO Locked
NEGATION GOTO Unlocked;
  MATCH RETURN {$1=usb_lock_device_for_reset($?)} -> SPLIT {((int)$1)==0} GOTO
Locked NEGATION GOTO Unlocked;

STATE USEALL Locked :
  MATCH CALL {usb_unlock_device($?)} -> GOTO Unlocked;
  MATCH CALL {ldv_check_alloc_flags($1)} -> ASSUME {((int)$1)!=16; ((int)$1)!=32}
ERROR("linux:alloc:usb lock::wrong flags");
  MATCH CALL {ldv_check_alloc_flags($1)} -> ASSUME {((int)$1)==32} GOTO Locked;
  MATCH CALL {ldv_check_alloc_flags($1)} -> ASSUME {((int)$1)==16} GOTO Locked;
  MATCH CALL {ldv_check_alloc_nonatomic($?)} ->  ERROR("linux:alloc:usb
lock::nonatomic");

END AUTOMATON
```

A.11. Требование linux:blk:request

Описание: проверка корректного использования запросов устройств блочного ввода-вывода (*block devices request*).

Список типов нарушений:

- модулям запрещается освобождать несозданные ими запросы устройств блочного ввода-вывода (*"linux:blk:request::double put"*);
- модулям запрещается повторно создавать запросы устройств блочного ввода-вывода с одинаковыми очередями устройств блочного ввода-вывода

request_queue (*"linux:blk:request::double get"*);

- на момент завершения работы модули должны освободить все созданные ими запросы устройств блочного ввода-вывода (*"linux:blk:request::get at exit"*).

Автомат:

```
OBSERVER AUTOMATON linux_blk_request
INITIAL STATE Zero;

STATE USEALL Zero :
  MATCH RETURN {$1=blk_get_request($2)} -> ASSUME {((int)$2) == 16U;
    ((struct request *)$1) != 0} GOTO Got;
  MATCH RETURN {$1=blk_get_request($2)} -> ASSUME {((int)$2) == 208U;
    ((struct request *)$1) != 0} GOTO Got;
  MATCH RETURN {$1=blk_get_request($2)} -> ASSUME {((int)$2) != 16U;
    ((int)$2) != 208U; ((struct request *)$1) == 0} GOTO Stop;
  MATCH RETURN {$1=blk_get_request($2)} -> ASSUME {((struct request *)$1) != 0}
    GOTO Got;
  MATCH RETURN {$1=blk_get_request($2)} -> ASSUME {((struct request *)$1) == 0}
    GOTO Zero;
  MATCH RETURN {$1=blk_make_request($?)} -> ASSUME {((struct request *)$1) == 0}
    GOTO Stop;
  MATCH RETURN {$1=blk_make_request($?)} -> .
    ASSUME {((unsigned long)$1) > (unsigned long)-MAX_ERRNO} GOTO Zero;
  MATCH RETURN {$1=blk_make_request($?)} ->
    ASSUME {((unsigned long)$1) <= (unsigned long)-MAX_ERRNO} GOTO Got;
  MATCH CALL {put_blk_rq($?)} -> ERROR("linux:blk:request::double put");

STATE USEFIRST Got :
  MATCH RETURN {$1=blk_get_request($?)} -> ERROR("linux:blk:request::double get");
  MATCH RETURN {$1=blk_make_request($?)} ->
    ERROR("linux:blk:request::double get");
  MATCH CALL {put_blk_rq($?)} -> GOTO Zero;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:blk:request::get at exit");

STATE USEFIRST Stop :
  TRUE -> GOTO Stop;

END AUTOMATON
```

A.12. Требование linux:usb:gadget

Описание: проверка корректного использования ресурсов вида *USB-устройство*.

Список типов нарушений:

- модулям запрещается регистрировать классы устройств, если USB-устройство уже зарегистрировано (*"linux:usb:gadget::class registration with*

usb gadget");

- модулям запрещается дерегистрировать классы устройств, если USB-устройство уже зарегистрировано (*"linux:usb:gadget::class deregistration with usb gadget"*);
- модулям запрещается регистрировать номера символьных устройств, если USB-устройство уже зарегистрировано (*"linux:usb:gadget::chrdev registration with usb gadget"*);
- модулям запрещается дерегистрировать номера символьных устройств, если USB-устройство уже зарегистрировано (*"linux:usb:gadget::chrdev deregistration with usb gadget"*);
- модулям запрещается дерегистрировать незарегистрированные ими USB-устройства (*"linux:usb:gadget::double usb gadget deregistration"*);
- модулям запрещается повторно регистрировать USB-устройства для одинаковых драйверов *usb_gadget_driver* (*"linux:usb:gadget::double usb gadget registration"*);
- на момент завершения работы модули должны дерегистрировать все зарегистрированные ими USB-устройства (*"linux:usb:gadget::usb gadget registered at exit"*).

Автомат:

```
OBSERVER AUTOMATON linux_usb_gadget
INITIAL STATE G0_C0_H0;

STATE USEALL G0_C0_H0 :
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((void *)$1) == 0} GOTO Stop;
  MATCH RETURN {$1=create_class($?)} ->
    ASSUME {((unsigned long)$1) > (unsigned long)-MAX_ERRNO} GOTO G0_C0_H0;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0} GOTO G0_C1_H0;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) == 0} GOTO G0_C1_H0;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) <  0} GOTO G0_C0_H0;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) >  0} GOTO Stop;
  MATCH CALL {unregister_class($?)} -> GOTO Stop;
  MATCH CALL {destroy_class($1)} -> ASSUME {((void *)$1) == 0} GOTO G0_C0_H0;
  MATCH CALL {destroy_class($1)} ->
    ASSUME {((unsigned long)$1) > (unsigned long)-MAX_ERRNO} GOTO G0_C0_H0;
```

```
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0} GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) < 0} GOTO G0_C0_H0;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) == 0; ((int)$2) != 0}
    GOTO G0_C0_H1;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) == 0; ((int)$2) == 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >  0; ((int)$2) != 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >  0; ((int)$2) == 0}
    GOTO G0_C0_H1;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) == 0}
    GOTO G0_C0_H1;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) <  0}
    GOTO G0_C0_H0;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) >  0}
    GOTO Stop;
  MATCH CALL {unregister_chrdev_region($?)} -> GOTO Stop;
  MATCH RETURN {$1=register_usb_gadget($?)} -> SPLIT {((int)$1)==0} GOTO G1_C0_H0
    NEGATION GOTO G0_C0_H0;
  MATCH CALL {unregister_usb_gadget($?)} ->
    ERROR("linux:usb:gadget::double usb gadget deregistration");

STATE USEALL G0_C1_H0 :
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((void *)$1) == 0} GOTO Stop;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G0_C1_H0;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0} GOTO Stop;
 MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) == 0} GOTO Stop;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) <  0} GOTO G0_C1_H0;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) >  0} GOTO Stop;
  MATCH CALL {unregister_class($?)} -> GOTO G0_C0_H0;
  MATCH CALL {destroy_class($1)} -> ASSUME {((void *)$1) == 0} GOTO G0_C1_H0;
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G0_C1_H0;
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0} GOTO G0_C0_H0;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) < 0} GOTO G0_C1_H0;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) == 0; ((int)$2) != 0}
    GOTO G0_C1_H1;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) == 0; ((int)$2) == 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >  0; ((int)$2) != 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >  0; ((int)$2) == 0}
    GOTO G0_C1_H1;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) == 0}
    GOTO G0_C1_H1;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) <  0}
    GOTO G0_C1_H0;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) >  0}
    GOTO Stop;
  MATCH CALL {unregister_chrdev_region($?)} -> GOTO Stop;
  MATCH RETURN {$1=register_usb_gadget($?)} -> SPLIT {((int)$1)==0} GOTO G1_C1_H0
    NEGATION GOTO G0_C1_H0;
  MATCH CALL {unregister_usb_gadget($?)} ->
    ERROR("linux:usb:gadget::double usb gadget deregistration");
```

```
STATE USEALL G0_C0_H1 :
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((void *)$1) == 0} GOTO Stop;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G0_C0_H1;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0} GOTO G0_C1_H1;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) == 0} GOTO G0_C1_H1;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) <  0} GOTO G0_C0_H1;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) >  0} GOTO Stop;
  MATCH CALL {unregister_class($?)} -> GOTO Stop;
  MATCH CALL {destroy_class($1)} -> ASSUME {((void *)$1) == 0} GOTO G0_C0_H1;
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G0_C0_H1;
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0} GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) < 0} GOTO G0_C0_H1;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >= 0} GOTO Stop;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) == 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) <  0}
    GOTO G0_C0_H1;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) >  0}
    GOTO Stop;
  MATCH CALL {unregister_chrdev_region($?)} -> GOTO G0_C0_H0;
  MATCH RETURN {$1=register_usb_gadget($?)} -> SPLIT {((int)$1)==0} GOTO G1_C0_H1
    NEGATION GOTO G0_C0_H1;
  MATCH CALL {unregister_usb_gadget($?)} ->
    ERROR("linux:usb:gadget::double usb gadget deregistration");

STATE USEALL G0_C1_H1 :
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((void *)$1) == 0} GOTO Stop;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G0_C1_H1;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0} GOTO Stop;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) == 0} GOTO Stop;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) <  0} GOTO G0_C1_H1;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) >  0} GOTO Stop;
  MATCH CALL {unregister_class($?)} -> GOTO G0_C0_H1;
  MATCH CALL {destroy_class($1)} -> ASSUME {((void *)$1) == 0} GOTO G0_C1_H1;
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G0_C1_H1;
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0} GOTO G0_C0_H1;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) < 0} GOTO G0_C0_H1;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >= 0} GOTO Stop;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) == 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) <  0}
    GOTO G0_C1_H1;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) >  0}
    GOTO Stop;
  MATCH CALL {unregister_chrdev_region($?)} -> GOTO G0_C1_H0;
  MATCH RETURN {$1=register_usb_gadget($?)} -> SPLIT {((int)$1)==0} GOTO G1_C1_H1
    NEGATION GOTO G0_C1_H1;
  MATCH CALL {unregister_usb_gadget($?)} ->
    ERROR("linux:usb:gadget::double usb gadget deregistration");

STATE USEALL G1_C0_H0 :
```

```
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((void *)$1) == 0} GOTO Stop;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G1_C0_H0;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0}
    ERROR ("linux:usb:gadget::class registration with usb gadget");
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) == 0}
    ERROR("linux:usb:gadget::class registration with usb gadget");
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) <  0} GOTO G1_C0_H0;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) >  0} GOTO Stop;
  MATCH CALL {unregister_class($?)} ->
    ERROR("linux:usb:gadget::class deregistration with usb gadget");
  MATCH CALL {destroy_class($1)} -> ASSUME {((void *)$1) == 0} GOTO G1_C0_H0;
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G1_C0_H0;
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0}
    ERROR("linux:usb:gadget::class deregistration with usb gadget");
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) < 0} GOTO G1_C0_H0;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) == 0; ((int)$2) != 0}
    ERROR("linux:usb:gadget::chrdev registration with usb gadget");
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) == 0; ((int)$2) == 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >  0; ((int)$2) != 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >  0; ((int)$2) == 0}
    ERROR("linux:usb:gadget::chrdev registration with usb gadget");
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1)==0}
    ERROR("linux:usb:gadget::chrdev registration with usb gadget");
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) < 0}
    GOTO G1_C0_H0;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) > 0}
    GOTO Stop;
  MATCH CALL {unregister_chrdev_region($?)} ->
    ERROR("linux:usb:gadget::chrdev deregistration with usb gadget");
  MATCH RETURN {$1=register_usb_gadget($?)} -> ASSUME {((int)$1)==0}
    ERROR("linux:usb:gadget::double usb gadget registration");
  MATCH RETURN {$1=register_usb_gadget($?)} -> ASSUME {((int)$1)!=0}
    GOTO G1_C0_H0;
  MATCH CALL {unregister_usb_gadget($?)} -> GOTO G0_C0_H0;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:usb:gadget::usb gadget registered at exit");

STATE USEALL G1_C1_H0 :
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((void *)$1) == 0} GOTO Stop;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G1_C1_H0;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0}
    ERROR ("linux:usb:gadget::class registration with usb gadget");
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1)==0}
    ERROR("linux:usb:gadget::class registration with usb gadget");
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) < 0} GOTO G1_C1_H0;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) > 0} GOTO Stop;
  MATCH CALL {unregister_class($?)} ->
    ERROR("linux:usb:gadget::class deregistration with usb gadget");
  MATCH CALL {destroy_class($1)} -> ASSUME {((void *)$1) == 0} GOTO G1_C1_H0;
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G1_C1_H0;
```

```
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0}
    ERROR("linux:usb:gadget::class deregistration with usb gadget");
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) < 0} GOTO G1_C1_H0;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) == 0; ((int)$2) != 0}
    ERROR("linux:usb:gadget::chrdev registration with usb gadget");
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) == 0; ((int)$2) == 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >  0; ((int)$2) != 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >  0; ((int)$2) == 0}
    ERROR("linux:usb:gadget::chrdev registration with usb gadget");
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1)==0}
    ERROR("linux:usb:gadget::chrdev registration with usb gadget");
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) < 0}
    GOTO G1_C1_H0;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) > 0}
    GOTO Stop;
  MATCH CALL {unregister_chrdev_region($?)} ->
    ERROR("linux:usb:gadget::chrdev deregistration with usb gadget");
  MATCH RETURN {$1=register_usb_gadget($?)} -> ASSUME {((int)$1)==0}
    ERROR("linux:usb:gadget::double usb gadget registration");
  MATCH RETURN {$1=register_usb_gadget($?)} -> ASSUME {((int)$1)!=0}
    GOTO G1_C1_H0;
  MATCH CALL {unregister_usb_gadget($?)} -> GOTO G0_C1_H0;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:usb:gadget::usb gadget registered at exit");

STATE USEALL G1_C0_H1 :
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((void *)$1) == 0} GOTO Stop;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G1_C0_H1;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0}
    ERROR ("linux:usb:gadget::class registration with usb gadget");
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1)==0}
    ERROR("linux:usb:gadget::class registration with usb gadget");
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) < 0} GOTO G1_C0_H1;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) > 0} GOTO Stop;
  MATCH CALL {unregister_class($?)} ->
    ERROR("linux:usb:gadget::class deregistration with usb gadget");
  MATCH CALL {destroy_class($1)} -> ASSUME {((void *)$1) == 0} GOTO G1_C0_H1;
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G1_C0_H1;
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0}
    ERROR("linux:usb:gadget::class deregistration with usb gadget");
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) < 0} GOTO G1_C0_H1;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) == 0; ((int)$2) != 0}
    ERROR("linux:usb:gadget::chrdev registration with usb gadget");
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) == 0; ((int)$2) == 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >  0; ((int)$2) != 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >  0; ((int)$2) == 0}
    ERROR("linux:usb:gadget::chrdev registration with usb gadget");
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1)==0}
    ERROR("linux:usb:gadget::chrdev registration with usb gadget");
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) < 0}
```

```
    GOTO G1_C0_H1;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) > 0}
    GOTO Stop;
  MATCH CALL {unregister_chrdev_region($?)} ->
    ERROR("linux:usb:gadget::chrdev deregistration with usb gadget");
  MATCH RETURN {$1=register_usb_gadget($?)} -> ASSUME {((int)$1)==0}
    ERROR("linux:usb:gadget::double usb gadget registration");
  MATCH RETURN {$1=register_usb_gadget($?)} -> ASSUME {((int)$1)!=0}
    GOTO G1_C0_H1;
  MATCH CALL {unregister_usb_gadget($?)} -> GOTO G0_C0_H1;
  MATCH CALL {ldv_check_final_state($?)} ->
   ERROR("linux:usb:gadget::usb gadget registered at exit");

STATE USEALL G1_C1_H1 :
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((void *)$1) == 0} GOTO Stop;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G1_C1_H1;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0}
    ERROR ("linux:usb:gadget::class registration with usb gadget");
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1)==0}
    ERROR("linux:usb:gadget::class registration with usb gadget");
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) < 0} GOTO G1_C1_H1;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) > 0} GOTO Stop;
  MATCH CALL {unregister_class($?)} ->
    ERROR("linux:usb:gadget::class deregistration with usb gadget");
  MATCH CALL {destroy_class($1)} -> ASSUME {((void *)$1) == 0} GOTO G1_C1_H1;
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G1_C1_H1;
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0}
    ERROR("linux:usb:gadget::class deregistration with usb gadget");
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) < 0} GOTO G1_C1_H1;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) == 0; ((int)$2) != 0}
    ERROR("linux:usb:gadget::chrdev registration with usb gadget");
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) == 0; ((int)$2) == 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >  0; ((int)$2) != 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >  0; ((int)$2) == 0}
    ERROR("linux:usb:gadget::chrdev registration with usb gadget");
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1)==0}
    ERROR("linux:usb:gadget::chrdev registration with usb gadget");
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) < 0}
    GOTO G1_C1_H1;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) > 0}
    GOTO Stop;
  MATCH CALL {unregister_chrdev_region($?)} ->
    ERROR("linux:usb:gadget::chrdev deregistration with usb gadget");
  MATCH RETURN {$1=register_usb_gadget($?)} -> ASSUME {((int)$1)==0}
    ERROR("linux:usb:gadget::double usb gadget registration");
  MATCH RETURN {$1=register_usb_gadget($?)} -> ASSUME {((int)$1)!=0}
    GOTO G1_C1_H1;
  MATCH CALL {unregister_usb_gadget($?)} -> GOTO G0_C1_H1;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:usb:gadget::usb gadget registered at exit");

STATE USEFIRST Stop :
  TRUE -> GOTO Stop;
```

```
END AUTOMATON
```

A.13. Требование linux:class

Описание: проверка корректного использования ресурсов вида класс устройств (*class structure*).

Список типов нарушений:

- модулям запрещается дерегистрировать незарегистрированные ими классы устройств (*"linux:class::double deregistration"*);
- модулям запрещается повторно регистрировать классы устройств с одинаковыми именами, т. е. параметром *const char *name* функции *class_create* (*"linux:class::double registration"*);
- на момент завершения работы модули должны дерегистрировать все зарегистрированные ими классы устройств (*"linux:class::registered at exit"*).

Автомат:

```
OBSERVER AUTOMATON linux_class
INITIAL STATE G0_C0_H0;

STATE USEALL G0_C0_H0 :
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((void *)$1) == 0} GOTO Stop;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G0_C0_H0;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0} GOTO G0_C1_H0;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) == 0} GOTO G0_C1_H0;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) <  0} GOTO G0_C0_H0;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) >  0} GOTO Stop;
  MATCH CALL {unregister_class($?)} ->
    ERROR("linux:class::double deregistration");
  MATCH CALL {destroy_class($1)} -> ASSUME {((void *)$1) == 0} GOTO G0_C0_H0;
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G0_C0_H0;
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0}
    ERROR("linux:class::double deregistration");

STATE USEALL G0_C1_H0 :
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((void *)$1) == 0} GOTO Stop;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G0_C1_H0;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0}
    ERROR("linux:class::double registration");
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) == 0}
```

```
    ERROR("linux:class::double registration");
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) <  0} GOTO G0_C1_H0;
  MATCH RETURN {$1=register_class($?)} -> ASSUME {((int)$1) >  0} GOTO Stop;
  MATCH CALL {unregister_class($?)} -> GOTO G0_C0_H0;
  MATCH CALL {destroy_class($1)} -> ASSUME {((void *)$1) == 0} GOTO G0_C1_H0;
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G0_C1_H0;
  MATCH CALL {destroy_class($1)} -> ASSUME {((unsigned long)$1) <=
    (unsigned long)-MAX_ERRNO; ((unsigned long)$1) > 0} GOTO G0_C0_H0;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:class::registered at exit");

STATE USEFIRST Stop :
  TRUE -> GOTO Stop;

END AUTOMATON
```

A.14. Требование linux:chrdev

Описание: проверка корректного использования ресурсов вида номер символьного устройства (*char device number*).

Список типов нарушений:

- модулям запрещается дерегистрировать незарегистрированные ими номера символьных устройств (*"linux:chrdev::double deregistration"*);
- модулям запрещается повторно регистрировать номера символьных устройств с одинаковыми именами, т. е. параметром *const char *name* функций *register_chrdev* и *register_chrdev_region* (*"linux:chrdev::double registration"*);
- на момент завершения работы модули должны дерегистрировать все зарегистрированные ими номера символьных устройств (*"linux:chrdev::registered at exit"*).

Автомат:

```
OBSERVER AUTOMATON linux_class
INITIAL STATE G0_C0_H0;

STATE USEALL G0_C0_H0 :
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((void *)$1) == 0} GOTO Stop;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) >
    (unsigned long)-MAX_ERRNO} GOTO G0_C0_H0;
  MATCH RETURN {$1=create_class($?)} -> ASSUME {((unsigned long)$1) <=
OBSERVER AUTOMATON linux_chrdev
INITIAL STATE G0_C0_H0;
```

```
STATE USEALL G0_C0_H0 :
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) < 0} GOTO G0_C0_H0;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) == 0; ((int)$2) != 0}
    GOTO G0_C0_H1;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) == 0; ((int)$2) == 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >  0; ((int)$2) != 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >  0; ((int)$2) == 0}
    GOTO G0_C0_H1;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) == 0}
    GOTO G0_C0_H1;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) <  0}
    GOTO G0_C0_H0;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) >  0}
    GOTO Stop;
  MATCH CALL {unregister_chrdev_region($?)} ->
    ERROR("linux:chrdev::double deregistration");

STATE USEALL G0_C0_H1 :
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) < 0} GOTO G0_C0_H1;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) == 0; ((int)$2) != 0}
    ERROR("linux:chrdev::double registration");
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) == 0; ((int)$2) == 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >  0; ((int)$2) != 0}
    GOTO Stop;
  MATCH RETURN {$1=register_chrdev($2)} -> ASSUME {((int)$1) >  0; ((int)$2) == 0}
    ERROR("linux:chrdev::double registration");
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1)==0}
    ERROR("linux:chrdev::double registration");
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) < 0}
    GOTO G0_C0_H1;
  MATCH RETURN {$1=register_chrdev_region($?)} -> ASSUME {((int)$1) > 0}
    GOTO Stop;
  MATCH CALL {unregister_chrdev_region($?)} -> GOTO G0_C0_H0;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:chrdev::registered at exit");

STATE USEFIRST Stop :
  TRUE -> GOTO Stop;

END AUTOMATON
```

A.15. Требование linux:rwlock

Описание: проверка корректного использования *r/w-семафоров* в одном потоке.

Список типов нарушений:

- модулям запрещается захватывать на чтение захваченные на запись r/w-семафоры (*"linux:rwlock::read lock on write lock"*);
- модулям запрещается освобождать на чтение незахваченные ими на чтение

r/w-семафоры (*"linux:rwlock::more read unlocks"*);

- на момент завершения работы модули должны освободить все захваченные ими на чтение r/w-семафоры (*"linux:rwlock::read lock at exit"*);
- модулям запрещается повторно захватывать на запись одни и те же r/w-семафоры (*"linux:rwlock::double write lock"*);
- модулям запрещается освобождать на запись незахваченные ими на запись r/w-семафоры (*"linux:rwlock::double write unlock"*);
- на момент завершения работы модули должны освободить все захваченные ими на запись r/w-семафоры (*"linux:rwlock::write lock at exit"*).

Автомат:

```
OBSERVER AUTOMATON linux_rwlock
INITIAL STATE R0_W0;

STATE USEALL R0_W0 :
  MATCH ENTRY -> ENCODE {int read_lock_state = 0;} GOTO R0_W0;
  MATCH CALL {read_lock($?)} -> ENCODE {read_lock_state=1;} GOTO R1_W0;
  MATCH CALL {read_unlock($?)} -> ERROR("linux:rwlock::more read unlocks");
  MATCH CALL {write_lock($?)} -> GOTO R0_W1;
  MATCH CALL {write_unlock($?)} -> ERROR("linux:rwlock::double write unlock");
  MATCH RETURN {$1=read_trylock($?)} -> ASSUME {((int)$1)==1}
    ENCODE {read_lock_state=1;} GOTO R1_W0;
  MATCH RETURN {$1=write_trylock($?)} -> ASSUME {((int)$1)==1} GOTO R0_W1;

STATE USEALL R0_W1 :
  MATCH CALL {read_lock($?)} ->
    ERROR("linux:rwlock::read lock on write lock");
  MATCH CALL {read_unlock($?)} -> ERROR("linux:rwlock::more read unlocks");
  MATCH CALL {write_lock($?)} -> ERROR("linux:rwlock::double write lock");
  MATCH CALL {write_unlock($?)} -> GOTO R0_W0;
  MATCH RETURN {$1=read_trylock($?)} -> ASSUME {((int)$1)==1}
    ERROR("linux:rwlock::read lock on write lock");
  MATCH RETURN {$1=write_trylock($?)} -> ASSUME {((int)$1)==1}
    ERROR("linux:rwlock::double write lock");
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:rwlock::write lock at exit");

STATE USEALL R1_W0 :
  MATCH CALL {read_lock($?)} -> ENCODE {read_lock_state=read_lock_state+1;}
    GOTO R1_W0;
  MATCH CALL {read_unlock($?)} -> ASSUME {read_lock_state >  1;}
    ENCODE {read_lock_state=read_lock_state-1;} GOTO R1_W0;
  MATCH CALL {read_unlock($?)} -> ASSUME {read_lock_state <= 1;}
    ENCODE {read_lock_state=0;} GOTO R0_W0;
  MATCH CALL {write_lock($?)} -> GOTO R1_W1;
  MATCH CALL {write_unlock($?)} -> ERROR("linux:rwlock::double write unlock");
  MATCH RETURN {$1=read_trylock($?)} -> ASSUME {((int)$1)==1}
    ENCODE {read_lock_state=read_lock_state+1;} GOTO R1_W0;
```

```
  MATCH RETURN {$1=write_trylock($?)} -> ASSUME {((int)$1)==1}
    GOTO R1_W1;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:rwlock::read lock at exit");

STATE USEALL R1_W1 :
  MATCH CALL {read_lock($?)} -> ERROR("linux:rwlock::read lock on write lock");
  MATCH CALL {read_unlock($?)} -> ASSUME {read_lock_state >  1;}
    ENCODE {read_lock_state=read_lock_state-1;} GOTO R1_W1;
  MATCH CALL {read_unlock($?)} -> ASSUME {read_lock_state <= 1;}
    ENCODE {read_lock_state=0;} GOTO R0_W1;
  MATCH CALL {write_lock($?)} -> ERROR("linux:rwlock::double write lock");
  MATCH CALL {write_unlock($?)} -> GOTO R1_W0;
  MATCH RETURN {$1=read_trylock($?)} -> ASSUME {((int)$1)==1}
    ERROR("linux:rwlock::read lock on write lock");
  MATCH RETURN {$1=write_trylock($?)} -> ASSUME {((int)$1)==1}
    ERROR("linux:rwlock::double write lock");
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:rwlock::read lock at exit");
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:rwlock::write lock at exit");

END AUTOMATON
```

A.16. Требование linux:bitops

Описание: проверка корректного использования параметров функций поиска ненулевых битов (*find_next_bit* и *find_first_bit*).

Список типов нарушений:

- Модули не должны вызывать функции поиска ненулевых битов, передавая им в качестве параметров смещение, превышающее размер массива (*"linux:bitops::offset out of range"*).

Автомат:

```
OBSERVER AUTOMATON linux_bitops
INITIAL STATE Init;

STATE USEALL Init :
  MATCH RETURN {$3=find_next_bit($1,$2)} -> ASSUME {((unsigned long)$2) >
    ((unsigned long)$1)} ERROR("linux:bitops::offset out of range");
  MATCH RETURN {$3=find_next_bit($1,$2)} -> ASSUME {((unsigned long)$2) <=
    ((unsigned long)$1);((unsigned long)$3) <= ((unsigned long)$1)} GOTO Init;
  MATCH RETURN {$3=find_next_bit($1,$2)} -> ASSUME {((unsigned long)$2) <=
    ((unsigned long)$1);((unsigned long)$3) >  ((unsigned long)$1)} GOTO Stop;
  MATCH RETURN {$2=find_first_bit($1)} -> ASSUME {((unsigned long)$2) <=
    ((unsigned long)$1)} GOTO Init;
  MATCH RETURN {$2=find_first_bit($1)} -> ASSUME {((unsigned long)$2) >
    ((unsigned long)$1)} GOTO Stop;

STATE USEFIRST Stop :
  TRUE -> GOTO Stop;
```

```
END AUTOMATON
```

A.17. Требование linux:usb:dev

Описание: проверка корректного использования счетчика ссылок *USB-устройств*.

Список типов нарушений:

- модулям запрещается уменьшать счетчики ссылок USB-устройств на большее значение, чем они были увеличены (*"linux:usb:dev::less initial decrement"*);
- модулям запрещается уменьшать счетчики ссылок других USB-устройств (*"linux:usb:dev::unincremented counter decrement"*);
- на момент завершения работы в модулях счетчики ссылок USB-устройств должны быть равны 0 (*"linux:usb:dev::more initial at exit"*);
- в случае некорректного завершения работы probe-функции в модулях счетчики ссылок USB-устройств должны быть равны 0 (*"linux:usb:dev::probe failed"*).

Автомат:

```
OBSERVER AUTOMATON linux_usb_dev
INITIAL STATE Init;

STATE USEALL Init :
  MATCH ENTRY -> ENCODE {int usb_dev_state = 0;} GOTO Init;
  MATCH RETURN {$2=usb_get_dev($1)} -> ASSUME {((struct usb_device *)$1) != 0;
    ((struct usb_device *)$2) != 0} ENCODE {usb_dev_state=usb_dev_state+1;}
    GOTO Inc;
  MATCH RETURN {$2=usb_get_dev($1)} -> ASSUME {((struct usb_device *)$1) == 0;
    ((struct usb_device *)$2) == 0} GOTO Init;
  MATCH RETURN {$2=usb_get_dev($1)} -> ASSUME {((struct usb_device *)$1) == 0;
    ((struct usb_device *)$2) != 0} GOTO Stop;
  MATCH RETURN {$2=usb_get_dev($1)} -> ASSUME {((struct usb_device *)$1) != 0;
    ((struct usb_device *)$2) == 0} GOTO Stop;
  MATCH CALL {usb_put_dev($1)} -> ASSUME {((struct usb_device *)$1) <  0}
    ERROR("linux:usb:dev::unincremented counter decrement");
  MATCH CALL {usb_put_dev($1)} -> ASSUME {((struct usb_device *)$1) >  0}
    ERROR("linux:usb:dev::less initial decrement");
  MATCH CALL {usb_put_dev($1)} -> ASSUME {((struct usb_device *)$1) == 0}
    GOTO Init;

STATE USEALL Inc :
  MATCH RETURN {$2=usb_get_dev($1)} -> ASSUME {((struct usb_device *)$1) != 0;
```

```
    ((struct usb_device *)$2) != 0} ENCODE {usb_dev_state=usb_dev_state+1;}
    GOTO Inc;
  MATCH RETURN {$2=usb_get_dev($1)} -> ASSUME {((struct usb_device *)$1) == 0;
    ((struct usb_device *)$2) == 0} GOTO Inc;
  MATCH RETURN {$2=usb_get_dev($1)} -> ASSUME {((struct usb_device *)$1) == 0;
    ((struct usb_device *)$2) != 0} GOTO Stop;
  MATCH RETURN {$2=usb_get_dev($1)} -> ASSUME {((struct usb_device *)$1) != 0;
    ((struct usb_device *)$2) == 0} GOTO Stop;
  MATCH CALL {usb_put_dev($1)} -> ASSUME {((struct usb_device *)$1) != 0;
    usb_dev_state >  1} ENCODE {usb_dev_state=usb_dev_state-1;} GOTO Inc;
  MATCH CALL {usb_put_dev($1)} -> ASSUME {((struct usb_device *)$1) != 0;
    usb_dev_state == 1} ENCODE {usb_dev_state=usb_dev_state-1;} GOTO Init;
  MATCH CALL {usb_put_dev($1)} -> ASSUME {((struct usb_device *)$1) == 0}
    GOTO Inc;
  MATCH CALL {ldv_check_return_value_probe($1)} -> ASSUME {((int)$1) == 0}
    GOTO Inc;
  MATCH CALL {ldv_check_return_value_probe($1)} -> ASSUME {((int)$1) != 0}
    ERROR("linux:usb:dev::probe failed");
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:usb:dev::more initial at exit");

STATE USEFIRST Stop :
  TRUE -> GOTO Stop;

END AUTOMATON
```

Примечание: функция *ldv_check_return_value_probe(int retval)* используется при построении модели окружения в системе LDV Tools для проверки возвращаемого значения (*retval*) probe-функций.

A.18. Требование linux:usb:register

Описание: проверка корректного использования функции регистрации *USB-драйвера*.

Список типов нарушений:

- некорректное завершение регистрации USB-драйвера должно приводить к некорректному завершению вызывавшей его *probe*-функции (*"linux:usb:register::wrong return value"*).

Автомат:

```
OBSERVER AUTOMATON linux_usb_register
INITIAL STATE Zero;

STATE USEALL Zero :
  MATCH RETURN {$1=usb_register_driver($?)} -> ASSUME {((int)$1) < 0}
    GOTO Probe_error;
  MATCH RETURN {$1=usb_register_driver($?)} -> ASSUME {((int)$1) >= 0} GOTO Zero;
```

```
STATE USEALL Probe_error :
  MATCH CALL {ldv_check_return_value_probe($1)} -> ASSUME {((int)$1)!=0}
    GOTO Zero;
  MATCH CALL {ldv_check_return_value_probe($1)} -> ASSUME {((int)$1)==0}
    ERROR("linux:usb:register::wrong return value");

END AUTOMATON
```

A.19. Требование linux:netdev

Описание: проверка корректного использования функции регистрации сетевого драйвера.

Список типов нарушений:

- некорректное завершение регистрации сетевого драйвера должно приводить к некорректному завершению вызывавшей его *probe*-функции (*"linux:netdev::wrong return value"*).

Автомат:

```
OBSERVER AUTOMATON linux_netdev
INITIAL STATE Zero;

STATE USEALL Zero :
  MATCH RETURN {$1=register_netdev($?)} -> ASSUME {((int)$1) < 0}
    GOTO Probe_error;
  MATCH RETURN {$1=register_netdev($?)} -> ASSUME {((int)$1) >= 0} GOTO Zero;

STATE USEALL Probe_error :
  MATCH CALL {ldv_check_return_value_probe($1)} -> ASSUME {((int)$1)!=0}
    GOTO Zero;
  MATCH CALL {ldv_check_return_value_probe($1)} -> ASSUME {((int)$1)==0}
    ERROR("linux:netdev::wrong return value");

END AUTOMATON
```

A.20. Требование linux:rculock

Описание: проверка корректного использования *RCU* блокировок в одном потоке.

Список типов нарушений:

- модулям запрещается использовать RCU блокировки во время критических секций RCU (*"linux:rculock::locked at read section"*);
- модулям запрещается освобождать незахваченные ими RCU блокировки (*"linux:rculock::more unlocks"*);

- на момент завершения работы модули должны освободить все захваченные ими RCU блокировки (*"linux:rculock::locked at exit"*).

Автомат:

```
OBSERVER AUTOMATON linux_rculock
INITIAL STATE Init;

STATE USEFIRST Init :
  MATCH ENTRY -> ENCODE {int rcu_state = 0;} GOTO Init;
  MATCH CALL {rcu_read_lock($?)} -> ENCODE {rcu_state=rcu_state+1;} GOTO Inc;
  MATCH CALL {rcu_read_unlock($?)} -> ERROR("linux:rculock::more unlocks");

STATE USEALL Inc :
  MATCH CALL {rcu_read_lock($?)} -> ENCODE {rcu_state=rcu_state+1;} GOTO Inc;
  MATCH CALL {rcu_read_unlock($?)} -> ASSUME {rcu_state != 1;}
    ENCODE {rcu_state=rcu_state-1;} GOTO Inc;
  MATCH CALL {rcu_read_unlock($?)} -> ASSUME {rcu_state == 1;}
    ENCODE {rcu_state=rcu_state-1;} GOTO Init;
  MATCH CALL {ldv_check_for_read_section($?)} ->
    ERROR("linux:rculock::locked at read section");
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:rculock::locked at exit");

END AUTOMATON
```

Примечание: функция *ldv_check_for_read_section()* используется при построении модели окружения в системе LDV Tools для моделирования начала критических секций RCU.

A.21. Требование linux:rculockbh

Описание: проверка корректного использования *RCU-bh* блокировок в одном потоке.

Список типов нарушений:

- модулям запрещается использовать RCU-bh блокировки во время критических секций RCU (*"linux:rculockbh::locked at read section"*);
- модулям запрещается освобождать незахваченные ими RCU-bh блокировки (*"linux:rculockbh::more unlocks"*);
- на момент завершения работы модули должны освободить все захваченные ими RCU-bh блокировки (*"linux:rculockbh::locked at exit"*).

Автомат:

```
OBSERVER AUTOMATON linux_rculockbh
INITIAL STATE Init;

STATE USEFIRST Init :
  MATCH ENTRY -> ENCODE {int rcu_bh_state = 0;} GOTO Init;
  MATCH CALL {rcu_read_lock_bh($?)} -> ENCODE {rcu_bh_state=rcu_bh_state+1;}
    GOTO Inc;
  MATCH CALL {rcu_read_unlock_bh($?)} ->
    ERROR("linux:rculockbh::more unlocks");

STATE USEALL Inc :
  MATCH CALL {rcu_read_lock_bh($?)} -> ENCODE {rcu_bh_state=rcu_bh_state+1;}
    GOTO Inc;
  MATCH CALL {rcu_read_unlock_bh($?)} -> ASSUME {rcu_bh_state != 1;}
    ENCODE {rcu_bh_state=rcu_bh_state-1;} GOTO Inc;
  MATCH CALL {rcu_read_unlock_bh($?)} -> ASSUME {rcu_bh_state == 1;}
    ENCODE {rcu_bh_state=rcu_bh_state-1;} GOTO Init;
  MATCH CALL {ldv_check_for_read_section($?)} ->
    ERROR("linux:rculockbh::locked at read section");
  MATCH CALL {ldv_check_final_state($?)} -> ERROR("linux:rculockbh::locked at
exit");
END AUTOMATON
```

A.22. Требование linux:rculocksched

Описание: проверка корректного использования *RCU-sched* блокировок в одном потоке.

Список типов нарушений:

- модулям запрещается использовать RCU-sched блокировки во время критических секций RCU (*"linux:rculocksched::locked at read section"*);
- модулям запрещается освобождать незахваченные ими RCU-sched блокировки (*"linux:rculocksched::more unlocks"*);
- на момент завершения работы модули должны освободить все захваченные ими RCU-sched блокировки (*"linux:rculocksched::locked at exit"*).

Автомат:

```
OBSERVER AUTOMATON linux_rculocksched
INITIAL STATE Init;

STATE USEFIRST Init :
  MATCH ENTRY -> ENCODE {int rcu_sched_state = 0;} GOTO Init;
  MATCH CALL {rcu_read_lock_sched($?)} ->
    ENCODE {rcu_sched_state=rcu_sched_state+1;} GOTO Inc;
  MATCH CALL {rcu_read_unlock_sched($?)} ->
    ERROR("linux:rculocksched::more unlocks");

STATE USEALL Inc :
  MATCH CALL {rcu_read_lock_sched($?)} ->
```

```
    ENCODE {rcu_sched_state=rcu_sched_state+1;} GOTO Inc;
  MATCH CALL {rcu_read_unlock_sched($?)} -> ASSUME {rcu_sched_state != 1;}
    ENCODE {rcu_sched_state=rcu_sched_state-1;} GOTO Inc;
  MATCH CALL {rcu_read_unlock_sched($?)} -> ASSUME {rcu_sched_state == 1;}
    ENCODE {rcu_sched_state=rcu_sched_state-1;} GOTO Init;
  MATCH CALL {ldv_check_for_read_section($?)} ->
    ERROR("linux:rculocksched::locked at read section");
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:rculocksched::locked at exit");

END AUTOMATON
```

A.23. Требование linux:srculock

Описание: проверка корректного использования *RCU* блокировок, способных заснуть (*Sleepable RCU – SRCU*), в одном потоке.

Список типов нарушений:

- модулям запрещается использовать SRCU блокировки во время критических секций RCU (*"linux:srculock::locked at read section"*);
- модулям запрещается освобождать незахваченные ими SRCU блокировки (*"linux:srculock::more unlocks"*);
- на момент завершения работы модули должны освободить все захваченные ими SRCU блокировки (*"linux:srculock::locked at exit"*).

Автомат:

```
OBSERVER AUTOMATON linux_srculock
INITIAL STATE Init;

STATE USEFIRST Init :
  MATCH ENTRY -> ENCODE {int srcu_state = 0;} GOTO Init;
  MATCH CALL {srcu_read_lock($?)} ->
    ENCODE {srcu_state=srcu_state+1;} GOTO Inc;
  MATCH CALL {srcu_read_unlock($?)} ->
    ERROR("linux:srculock::more unlocks");

STATE USEALL Inc :
  MATCH CALL {srcu_read_lock($?)} -> ENCODE {srcu_state=srcu_state+1;}
    GOTO Inc;
  MATCH CALL {srcu_read_unlock($?)} -> ASSUME {srcu_state != 1;}
    ENCODE {srcu_state=srcu_state-1;} GOTO Inc;
  MATCH CALL {srcu_read_unlock($?)} -> ASSUME {srcu_state == 1;}
    ENCODE {srcu_state=srcu_state-1;} GOTO Init;
  MATCH CALL {ldv_check_for_read_section($?)} ->
    ERROR("linux:srculock::locked at read section");
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:srculock::locked at exit");

END AUTOMATON
```

A.24. Требование linux:completion

Описание: проверка корректного использования ожидания *completion*.

Список типов нарушений:

- модулям необходимо инициализировать *completion* перед работой с ним (*"linux:completion::wait without init"*);
- модулям запрещается инициализировать один и тот же *completion* дважды (*"linux:completion::double init"*).

Автомат:

```
// for arg_sign in completion_arg_signs
OBSERVER AUTOMATON linux_completion{{ arg_sign.id }}

INITIAL STATE Init;

STATE USEFIRST Init :
  MATCH CALL {init_completion{{ arg_sign.id }}($?)} -> GOTO Declared;
  MATCH CALL {init_completion_macro{{ arg_sign.id }}($?)} ->
    ERROR("linux:completion::double init");
  MATCH CALL {wait_for_completion{{ arg_sign.id }}($?)} ->
    ERROR("linux:completion::wait without init");

STATE USEFIRST Declared :
  TRUE -> GOTO Declared;

END AUTOMATON
// endfor
```

A.25. Требование linux:mmc:sdio_func

Описание: проверка корректного использования *MMC* хост-контроллера (*host controllers*).

Список типов нарушений:

- модулям запрещается повторно резервировать одни и те же MMC хост-контроллеры (*"linux:mmc:sdio_func::double claim"*);
- модулям запрещается освобождать незарезервированные ими MMC хост-контроллеры (*"linux:mmc:sdio_func::release without claim"*);
- модулям запрещается осуществлять операции с незарезервированными ими MMC хост-контроллеры, например, *sdio_disable_func*, *sdio_readb*, *sdio_memcpy_fromio* и т. д. (*"linux:mmc:sdio_func::wrong params"*);

- на момент завершения работы модули должны освободить все зарезервированные ими MMC хост-контроллеры (*"linux:mmc:sdio_func::unreleased at exit"*).

  Автомат:

```
// for arg_sign in sdio_func_as_arg_signs
OBSERVER AUTOMATON linux_sdio_func_as{{ arg_sign.id }}

INITIAL STATE Unlocked;

STATE USEFIRST Unlocked :
  MATCH CALL {sdio_claim_host{{ arg_sign.id }}($?)} -> GOTO Locked;
  MATCH CALL {sdio_release_host{{ arg_sign.id }}($?)} ->
    ERROR("linux:mmc:sdio_func::release without claim");
  MATCH CALL {ldv_check_mmc_context{{ arg_sign.id }}($?)} ->
    ERROR("linux:mmc:sdio_func::wrong params");

STATE USEFIRST Locked :
  MATCH CALL {sdio_claim_host{{ arg_sign.id }}($?)} ->
    ERROR("linux:mmc:sdio_func::double claim");
  MATCH CALL {sdio_release_host{{ arg_sign.id }}($?)} ->
    GOTO Unlocked;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:mmc:sdio_func::unreleased at exit");

END AUTOMATON
// endfor
```

Примечание: функция *ldv_check_mmc_context()* используется при построении модели окружения в системе LDV Tools для моделирования операций с MMC хост-контроллерами.

A.26. Требование linux:sysfs

Описание: проверка корректного использования групп для виртуальных файловых систем (*sysfs attribute group*).

Список типов нарушений:

- модулям запрещается освобождать несозданные ими группы для виртуальных файловых систем (*"linux:sysfs::less initial decrement"*);
- на момент завершения работы модули должны освободить все созданные ими группы для виртуальных файловых систем (*"linux:sysfs::more initial at exit"*).

Автомат:

```
OBSERVER AUTOMATON linux_sysfs
INITIAL STATE Init;

STATE USEALL Init :
  MATCH ENTRY -> ENCODE {int sysfs_state = 0;} GOTO Init;
  MATCH RETURN {$1=sysfs_create_group($?)} -> ASSUME {((int)$1) == 0}
    ENCODE {sysfs_state=sysfs_state+1;} GOTO Inc;
  MATCH RETURN {$1=sysfs_create_group($?)} -> ASSUME {((int)$1) <  0} GOTO Init;
  MATCH RETURN {$1=sysfs_create_group($?)} -> ASSUME {((int)$1) >  0} GOTO Stop;
  MATCH CALL {sysfs_remove_group($?)} ->
    ERROR("linux:sysfs::less initial decrement");

STATE USEALL Inc :
  MATCH RETURN {$1=sysfs_create_group($?)} -> ASSUME {((int)$1) == 0}
    ENCODE {sysfs_state=sysfs_state+1;} GOTO Inc;
  MATCH RETURN {$1=sysfs_create_group($?)} -> ASSUME {((int)$1) <  0} GOTO Inc;
  MATCH RETURN {$1=sysfs_create_group($?)} -> ASSUME {((int)$1) >  0} GOTO Stop;
  MATCH CALL {sysfs_remove_group($?)} -> ASSUME {sysfs_state >  1}
    ENCODE {sysfs_state=sysfs_state-1;} GOTO Inc;
  MATCH CALL {sysfs_remove_group($?)} -> ASSUME {sysfs_state <= 1}
    ENCODE {sysfs_state=sysfs_state-1;} GOTO Init;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:sysfs::more initial at exit");

STATE USEFIRST Stop :
  TRUE -> GOTO Stop;

END AUTOMATON
```

A.27. Требование linux:iomem

Описание: проверка корректного использования ресурсов вида отображение *IO-памяти*.

Список типов нарушений:

- модулям запрещается освобождать невыделенные ими отображения IO-памяти (*"linux:iomem::less initial decrement"*);
- на момент завершения работы модули должны освободить все выделенные ими отображения IO-памяти (*"linux:iomem::more initial at exit"*).

Автомат:

```
OBSERVER AUTOMATON linux_iomem
INITIAL STATE Init;

STATE USEALL Init :
  MATCH ENTRY -> ENCODE {int iomem_state = 0;} GOTO Init;
  MATCH RETURN {$1=io_mem_remap($?)} -> ASSUME {((void *)$1) != 0}
    ENCODE {iomem_state=iomem_state+1;} GOTO Inc;
  MATCH RETURN {$1=io_mem_remap($?)} -> ASSUME {((void *)$1) == 0} GOTO Init;
```

```
  MATCH CALL {io_mem_unmap($?)} ->
    ERROR("linux:iomem::less initial decrement");

STATE USEALL Inc :
  MATCH RETURN {$1=io_mem_remap($?)} -> ASSUME {((void *)$1) != 0}
    ENCODE {iomem_state=iomem_state+1;} GOTO Inc;
  MATCH RETURN {$1=io_mem_remap($?)} -> ASSUME {((void *)$1) == 0}
    GOTO Inc;
  MATCH CALL {io_mem_unmap($?)} -> ASSUME {iomem_state >  1}
    ENCODE {iomem_state=iomem_state-1;} GOTO Inc;
  MATCH CALL {io_mem_unmap($?)} -> ASSUME {iomem_state <= 1}
    ENCODE {iomem_state=iomem_state-1;} GOTO Init;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:iomem::more initial at exit");

END AUTOMATON
```

A.28. Требование linux:idr

Описание: проверка корректного использования ресурсов вида *IDR*.

Список типов нарушений:

- модулям запрещается использовать IDR до инициализации (*"linux:idr::not initialized"*);
- модулям запрещается инициализировать один и тот же IDR дважды (*"linux:idr::double init"*);
- на момент завершения работы модули должны освободить все выделенные IDR (*"linux:idr::more at exit"*);
- модулям запрещается использовать IDR после уничтожения (*"linux:idr::destroyed before usage"*).

Автомат:

```
// for arg_sign in idr_arg_signs
OBSERVER AUTOMATON linux_idr{{ arg_sign.id }}

INITIAL STATE Init;

STATE USEFIRST Init :
  MATCH CALL {idr_init{{ arg_sign.id }}($?)} -> GOTO Initialized;
  MATCH CALL {idr_alloc{{ arg_sign.id }}($?)} ->
    ERROR("linux:idr::not initialized");
  MATCH CALL {idr_find{{ arg_sign.id }}($?)} ->
    ERROR("linux:idr::not initialized");
  MATCH CALL {idr_remove{{ arg_sign.id }}($?)} ->
    ERROR("linux:idr::not initialized");
  MATCH CALL {idr_destroy{{ arg_sign.id }}($?)} ->
    ERROR("linux:idr::not initialized");
```

```
STATE USEFIRST Initialized :
  MATCH CALL {idr_init{{ arg_sign.id }}($?)} -> ERROR("linux:idr::double init");
  MATCH CALL {idr_alloc{{ arg_sign.id }}($?)} -> GOTO Changed;
  MATCH CALL {idr_find{{ arg_sign.id }}($?)} -> GOTO Changed;
  MATCH CALL {idr_remove{{ arg_sign.id }}($?)} -> GOTO Changed;
  MATCH CALL {idr_destroy{{ arg_sign.id }}($?)} -> GOTO Destroyed;
  MATCH CALL {ldv_check_final_state($?)} -> ERROR("linux:idr::more at exit");

STATE USEFIRST Changed :
  MATCH CALL {idr_init{{ arg_sign.id }}($?)} -> ERROR("linux:idr::double init");
  MATCH CALL {idr_alloc{{ arg_sign.id }}($?)} -> GOTO Changed;
  MATCH CALL {idr_find{{ arg_sign.id }}($?)} -> GOTO Changed;
  MATCH CALL {idr_remove{{ arg_sign.id }}($?)} -> GOTO Changed;
  MATCH CALL {idr_destroy{{ arg_sign.id }}($?)} -> GOTO Destroyed;
  MATCH CALL {ldv_check_final_state($?)} -> ERROR("linux:idr::more at exit");

STATE USEFIRST Destroyed :
  MATCH CALL {idr_init{{ arg_sign.id }}($?)} ->
    ERROR("linux:idr::double init");
  MATCH CALL {idr_alloc{{ arg_sign.id }}($?)} ->
    ERROR("linux:idr::destroyed before usage");
  MATCH CALL {idr_find{{ arg_sign.id }}($?)} ->
    ERROR("linux:idr::destroyed before usage");
  MATCH CALL {idr_remove{{ arg_sign.id }}($?)} ->
    ERROR("linux:idr::destroyed before usage");
  MATCH CALL {idr_destroy{{ arg_sign.id }}($?)} ->
    ERROR("linux:idr::destroyed before usage");

END AUTOMATON
// endfor
```

A.29. Требование linux:sock

Описание: проверка корректного использования блокировок сокетов в одном потоке.

Список типов нарушений:

- модулям запрещается освобождать незахваченные ими сокеты (*"linux:sock::double release"*);
- на момент завершения работы модули должны освободить все захваченные ими сокеты (*"linux:sock::all locked sockets must be released"*).

Автомат:

```
OBSERVER AUTOMATON linux_sock
INITIAL STATE Init;

STATE USEALL Init :
  MATCH ENTRY -> ENCODE {int sock_state = 0;} GOTO Init;
  MATCH CALL {lock_sock($?)} -> ENCODE {sock_state=sock_state+1;} GOTO Inc;
  MATCH RETURN {$1=lock_sock_fast($?)} -> ASSUME {((int)$1) != 0}
```

```
    ENCODE {sock_state=sock_state+1;} GOTO Inc;
  MATCH RETURN {$1=lock_sock_fast($?)} -> ASSUME {((int)$1) == 0} GOTO Init;
  MATCH CALL {unlock_sock($?)} -> ERROR("linux:sock::double release");

STATE USEALL Inc :
  MATCH CALL {lock_sock($?)} -> ENCODE {sock_state=sock_state+1;} GOTO Inc;
  MATCH RETURN {$1=lock_sock_fast($?)} -> ASSUME {((int)$1) != 0}
    ENCODE {sock_state=sock_state+1;} GOTO Inc;
  MATCH RETURN {$1=lock_sock_fast($?)} -> ASSUME {((int)$1) == 0} GOTO Inc;
  MATCH CALL {unlock_sock($?)} -> ASSUME {sock_state >  1}
    ENCODE {sock_state=sock_state-1;} GOTO Inc;
  MATCH CALL {unlock_sock($?)} -> ASSUME {sock_state <= 1}
    ENCODE {sock_state=sock_state-1;} GOTO Init;
  MATCH CALL {ldv_check_final_state($?)} ->
    ERROR("linux:sock::all locked sockets must be released");

END AUTOMATON
```

A.30. Требование linux:rtnl

Описание: проверка корректного использования *RTNL* блокировок в одном потоке.

Список типов нарушений:

- модулям запрещается повторно захватывать одни и те же RTNL блокировки (*"linux:rtnl::double lock"*);
- модулям запрещается освобождать незахваченные ими RTNL блокировки (*"linux:rtnl::double unlock"*);
- на момент завершения работы модули должны освободить все захваченные ими RTNL блокировки (*"linux:rtnl::lock on exit"*).

Автомат:

```
OBSERVER AUTOMATON linux_rtnl
INITIAL STATE Unlocked;

STATE USEALL Unlocked :
  MATCH CALL {rtnl_lock($?)} -> GOTO Locked;
  MATCH CALL {rtnl_unlock($?)} -> ERROR("linux:rtnl::double unlock");
  MATCH RETURN {$1=rtnl_trylock($?)} -> ASSUME {((int)$1) != 0} GOTO Locked;
  MATCH RETURN {$1=rtnl_trylock($?)} -> ASSUME {((int)$1) == 0} GOTO Unlocked;

STATE USEALL Locked :
  MATCH CALL {rtnl_lock($?)} -> ERROR("linux:rtnl::double lock");
  MATCH CALL {rtnl_trylock($?)} -> ERROR("linux:rtnl::double lock");
  MATCH RETURN {$1=rtnl_is_locked($?)} -> SPLIT {((int)$1) == 1}
    GOTO Locked NEGATION GOTO Stop;
  MATCH CALL {rtnl_unlock($?)} -> GOTO Unlocked;
  MATCH CALL {ldv_check_final_state($?)} -> ERROR("linux:rtnl::lock on exit");
```

```
STATE USEFIRST Stop :
  TRUE -> GOTO Stop;

END AUTOMATON
```

Приложение Б
Рекомендации по выбору параметров использования предложенных методов верификации

В данном приложении рассматриваются варианты выбора параметров предложенных методов верификации и их влияние на скорость, ресурсоемкость и качество получаемых результатов верификации. В дополнение к экспериментам, описанным в главе 5, рассматриваются дополнительные эксперименты, и на основе полученных данных даются рекомендации о выборе параметров использования методов верификации.

Б.1. Рекомендации по выбору параметров метода условной многоаспектной верификации

Среди настраиваемых параметров использования метода УМАВ можно выделить три группы – стратегии корректировки уровня абстракции, способ перезапуска алгоритма и используемые внутренние лимиты. Для каждой из данных групп параметров будут даны рекомендации на основе проведенных экспериментов.

Б.1.1. Стратегии корректировки уровня абстракции

Данный эксперимент был нацелен на сопоставление различных стратегий корректировки уровня абстракции в методе УМАВ. В качестве начальной конфигурации метода УМАВ был взят внутренний перезапуск и следующие значения внутренних лимитов: ВЛУ=900 секунд (т. е. базовое ограничение на проверку одного требования), ВЛПИ=20 секунд, ВЛИ=100 секунд, ВЛНИ=100 секунд. Результаты эксперимента представлены в таблице Б.1 (приведено сравнение процессорного времени только для решения задач достижимости[17]).

17 Здесь и далее процессорное время всего процесса верификации будет отсутствовать в тех случаях, если для сопоставляемых методов время подготовки задач достижимости совпадает.

Метод	*Safe*	*Unsafe*	Потери / новые (%)	Процессорное время / ускорение	Итерации / *l*
Последовательная верификация	118 703	667	–0 +0	3 889 000 1.00	121 230 30
УМАВ: стратегия *Ничего*	116 655	633	–1.78 +0.06	1 313 000 2.96	5 490 1.36
УМАВ: стратегия *ЛО/Вычитание*	116 916	630	–1.56 +0.06	1 325 000 2.94	5 352 1.32
УМАВ: стратегия *ЛО/Очистка*	116 963	630	–1.53 +0.06	1 340 000 2.9	4 438 1.1
УМАВ: стратегия *АГД/Вычитание*	116 986	633	–1.51 +0.06	1 286 000 3.02	5 408 1.34
УМАВ: стратегия *Все*	117 162	634	–1.36 +0.06	1 289 000 3.02	4 434 1.1

Табл. Б.1. Сравнение стратегий корректировки уровня абстракции в методе УМАВ (*l* – среднее число итераций для решения одной задачи).

Результаты эксперимента демонстрируют особенности предложенных стратегий. Стратегия без корректировки точности абстракции *Ничего* хуже всех справляется с предотвращением экспоненциального роста числа состояний и тем самым теряет наибольший процент результата. В данном случае точность абстракции очищается только при запуске новой итерации УМАВ, что приводит к наибольшему числу итераций для данной стратегии и, соответственно, росту накладных расходов (например, перестроение общих элементов АГД). Стратегии, нацеленные на корректировку точности абстракции только в листе ожидания, как правило, требуют меньше ресурсов на выполнение самой операции, однако при этом эффективность стратегий невысокая, что выражается в достаточно низком ускорении. Стратегия *ЛО/Вычитание* нацелена на минимизацию негативного эффекта переиспользования, ведущего к экспоненциальному росту числа состояний, однако при относительно большом количестве проверяемых требований она перестает работать из-за того, что точность абстракции для

каждого требования вычисляется эвристическим способом, поэтому возможны ситуации, в которых удаляется точность абстракции, не соответствующая отключаемому требованию. Стратегия *ЛО/Очистка* менее подвержена подобным проблемам, поэтому успешно решает несколько больше задач, но при этом требует больше ресурсов, поскольку часто удаляется лишняя точность абстракции, которую затем приходится восстанавливать. Стратегия *АГД/Вычитание* полностью предотвращает негативное влияние отключаемого требования с точностью до используемой в методе УМАВ аппроксимации, что позволяет демонстрировать наивысшее ускорение. Минусом данной стратегии является относительно большое время работы самой операции, поскольку в АГД число абстрактных состояний измеряется миллионами. Стратегия *Все* полностью сбрасывает точность абстракции, что, с одной стороны, по определению не может быть оптимальным решением (поскольку для проверки остальных требований часть точности придется восстанавливать, что приведет к дополнительным накладным расходам), но с другой стороны, достаточно эффективно предотвращается негативное влияние отключаемого требования, что позволяет успешно решить максимальное число задач. Помимо этого, данная стратегия требует минимальное количество итераций, т. е. лучше всех справляется с проблемой экспоненциального роста числа состояний. В сравнении с методом последовательной верификации все стратегии оказались быстрее примерно в 3 раза при потерях результата менее 2%. Кроме того, метод УМАВ смог решить часть задач (порядка 74-77), с которыми базовый подход верификации не мог справиться с теми же ограничениями на ресурсы.

Стоит отметить, что полученный результат отличается от описанного в [33] из-за того, что, во-первых, данный эксперимент проводился как на большем наборе задач (4041 вместо 1000), так и на большем наборе требований (30 вместо 17), во-вторых, среди имеющихся требований практически не было пересечений

(всего 5 из 17 требований имели общие части), в то время как среди используемых в данной работе требований пересечений было значительно больше (15 из 30). Данные результаты показывают, что в разных ситуациях более подходят разные стратегии корректировки уровня абстракции.

Таким образом, стратегию корректировки уровня абстракции *Ничего* не рекомендуется использовать; стратегия *ЛО/Вычитание* наиболее подойдет для относительно малого числа проверяемых требований (10-20), среди которых практически нет пересечений; стратегия *ЛО/Очистка* наиболее подойдет для проверки также относительно малого числа требований с достаточно большим количеством пересечений; стратегия *АГД/Вычитание* более приспособлена для проверки большего числа требований (20 и более) с относительно малым количеством пересечений; стратегия *Все* наилучший результат демонстрирует для проверки большего числа требований с относительно большим количеством пересечений.

Б.1.2. Способ перезапуска алгоритма

Данный эксперимент был нацелен на сравнение внешнего и внутреннего перезапусков метода УМАВ. В качестве дополнительной эвристики для раннего обнаружения экспоненциального роста числа состояний во внешней УМАВ дополнительно использовалось внешнее ограничение по времени на каждую итерацию, поскольку известны случаи, в которых существующие эвристики не позволяли полностью решить данную проблему. Каждая итерация была ограничена $\sqrt{N} * t$ секундами, где t – базовое ограничение на проверку одного требования, N – число требований (т. е. 4930 секунд процессорного времени для 30 требований). Данное ограничение было получено эвристически (существенное уменьшение ограничения вело к существенным потерям результата, а его увеличение – к увеличению требуемых ресурсов). При этом ограничение на все

итерации в обоих случаях составляло 27 000 секунд. Результаты эксперимента представлены в таблице Б.2.

Метод	*Safe*	*Unsafe*	Потери / новые (%)	Процессорное время / ускорение	
				CPAchecker	LDV Tools
Последовательная верификация	118 703	667	–0 +0	3 889 000 1	6 742 000 1
Внешняя УМАВ	116 641	622	–1.79 +0.06	1 242 000 3.13	1 514 000 4.45
Внутренняя УМАВ	117 162	634	–1.36 +0.06	1 289 000 3.02	1 514 000 4.45

Табл. Б.2. Сравнение внешнего и внутреннего перезапусков УМАВ.

С одной стороны, метод внутренней УМАВ благодаря большему переиспользованию верификационных фактов способен успешно решать большее количество задач. С другой стороны, упрощение задач достижимости между итерациями и дополнительная эвристика в методе внешней УМАВ позволяют ему демонстрировать большее ускорение при решении задач достижимости. Однако стоит также учитывать, что подобное упрощение задач достижимости также требует дополнительных ресурсов, поэтому время подготовки задач достижимости существенно больше во внешней УМАВ. Следовательно, суммарное время подготовки и решения задач достижимости практически совпадает для рассматриваемых методов.

Одним из минусов внутренней УМАВ является большая зависимость от ограничения на оперативную память, поскольку все итерации производятся в рамках одного запуска инструмента верификации, что может привести как к переполнению кэшей инструмента, так и к исчерпанию памяти при построении АГД. Для оценки подобного эффекта данный эксперимент был повторен с различными ограничениями на оперативную память: 7.5 GB, 10 GB и 20 GB. При этом дополнительное ограничение по памяти на размер кучи для виртуальной

машины Java устанавливалось равным 13/15 от базового ограничения на оперативную память. Зависимость ускорения от ограничения на оперативную память для обоих методов представлено на рисунке Б.1, зависимость процента потерь от ограничения на оперативную память – на рисунке Б.2.

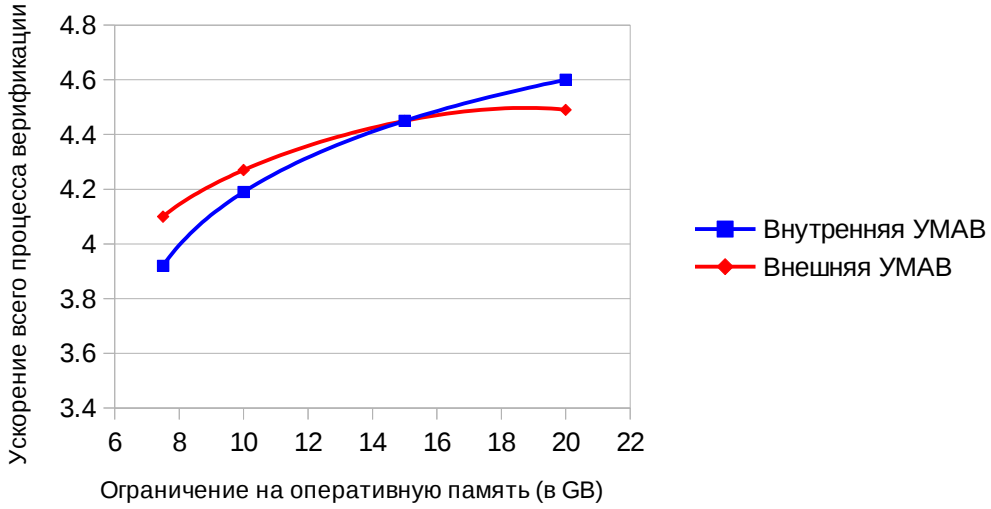

Рис. Б.1. Зависимость ускорения всего процесса верификации от ограничения на оперативную память для методов внешней и внутренней УМАВ.

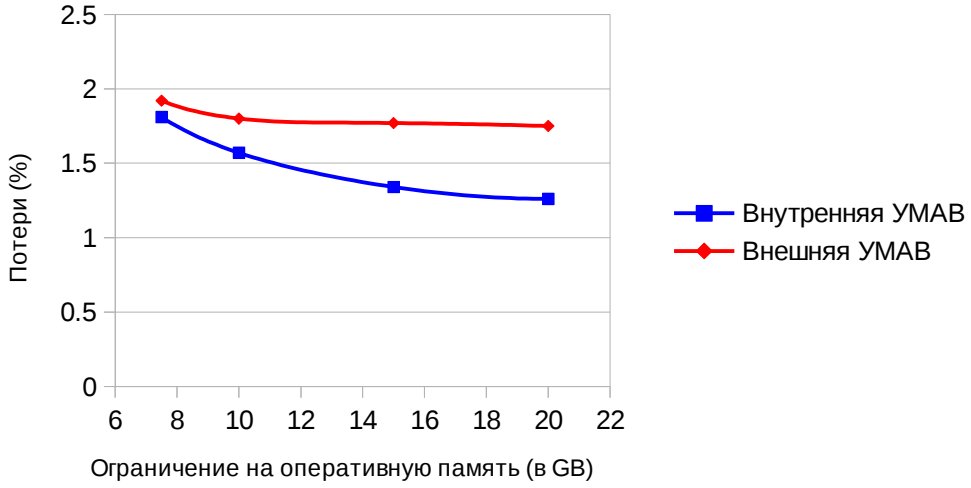

Рис. Б.2. Зависимость процента потерь от ограничения на оперативную память для методов внешней и внутренней УМАВ.

Таким образом, внешний перезапуск УМАВ более подходит для решения

задач, в которых достаточно часто возникает экспоненциальный рост числа состояний или некорректное завершение работы верификатора, вызванное проверкой только части требований, а также при использовании относительно малых ограничений на оперативную память (порядка 10 GB). Внутренний перезапуск УМАВ рекомендуется в качестве основного, если нет (или очень мало) задач, приводящих к некорректному завершению верификатора или исчерпанию доступной памяти только на части из проверяемых требований, а также имеется возможность выделять несколько больше оперативной памяти для решения задач достижимости (15 GB и выше).

Б.1.3. Внутренние лимиты

Внутренние лимиты в методе УМАВ являются основой алгоритма, поскольку именно их нарушение приводит к корректировке уровня абстракции и перезапуску алгоритма. Помимо этого, реализованы 2 эвристических лимита (потенциально могут быть реализованы и другие лимиты), нацеленные на более раннее обнаружение ситуаций, приводящих к экспоненциальному росту числа состояний.

Внутренний лимит утверждения (ВЛУ) ограничивает ресурсы на проверку каждого требования и с точностью до используемой аппроксимации в методе УМАВ полностью соответствует базовому ограничению на проверку одного требования в методе последовательной верификации. Отключение или изменение данного лимита нарушает основное положения метода УМАВ (т. е. ограничение ресурсов на проверку каждого требования в отдельности при проверки их композиции).

Внутренний лимит пустых интервалов (ВЛПИ) служит для перезапуска итераций УМАВ и непосредственно влияет на переиспользование АГД – чем больше данный лимит, тем больше времени будет выделяться для продолжения

верификации после отключения утверждения. Отключение данного лимита приведет к отсутствию перезапусков в методе УМАВ, т. е. будет использоваться метод УМАВ без перезапусков (на практике потери возрастают примерно до 4%, а ускорение уменьшается до 1.5, поэтому данная конфигурация не рекомендуется). По умолчанию рекомендуется использовать эвристически полученное значение в 20 процессорных секунд (при увеличении или уменьшении значения наблюдалось незначительное увеличение числа потерь и требуемых ресурсов, поскольку переиспользование слишком сложных АГД, как и отсутствие переиспользования достаточно простых, не рекомендуется – см. рис. Б.3 и Б.4).

Внутренний лимит интервала (ВЛИ) – эвристический внутренний лимит, нацеленный на более быстрое выявление требований, которые приводят к экспоненциальному росту числа состояний и тем самым «мешают» проверке остальных требований. В общем случае ведет к потере результата. Эксперименты показали (см. рис. Б.5 и Б.6), что увеличение данного лимита будет вести к увеличению требуемых ресурсов и уменьшению потерь, однако при достаточно больших значениях (близких к базовому ограничению на проверку одного требования) метод УМАВ существенно деградирует за счет того, что во многих случаях вовремя не удается предотвратить экспоненциальный рост числа состояний. Наибольшее ускорение с относительно небольшим уровнем потерь достигается при 100 секундах, минимальный процент потерь был получен при 225 секундах.

Внутренний лимит начального интервала (ВЛНИ) – эвристический внутренний лимит, нацеленный на выявление потенциально неразрешимых задач. В общем случае ведет к потере результата. Увеличение данного лимита будет вести к увеличению требуемых ресурсов и уменьшению потерь (см. рис. Б.7 и Б.8). Полное отключение данного лимита приводит к относительно малому снижению потерь (менее 0.1%) при достаточно большом увеличении требуемых ресурсов

(примерно в 1.3 раза). Для наиболее точной верификации ВЛНИ рекомендуется отключать, а для более быстрой верификации при минимальных потерях выбирать значение порядка 100 секунд.

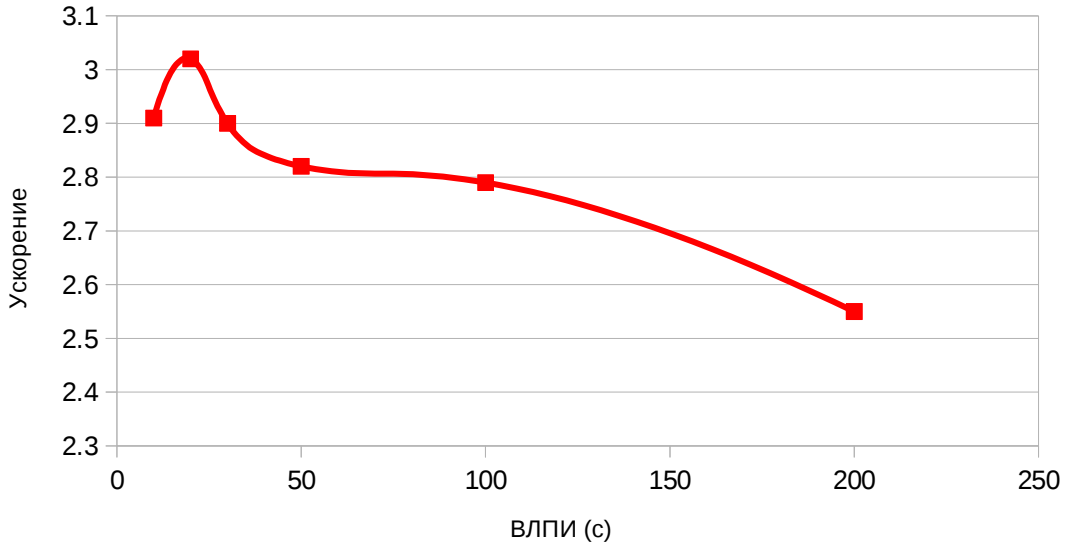

Рис. Б.3. Зависимость ускорения решения задач достижимости методом УМАВ от значения ВЛПИ.

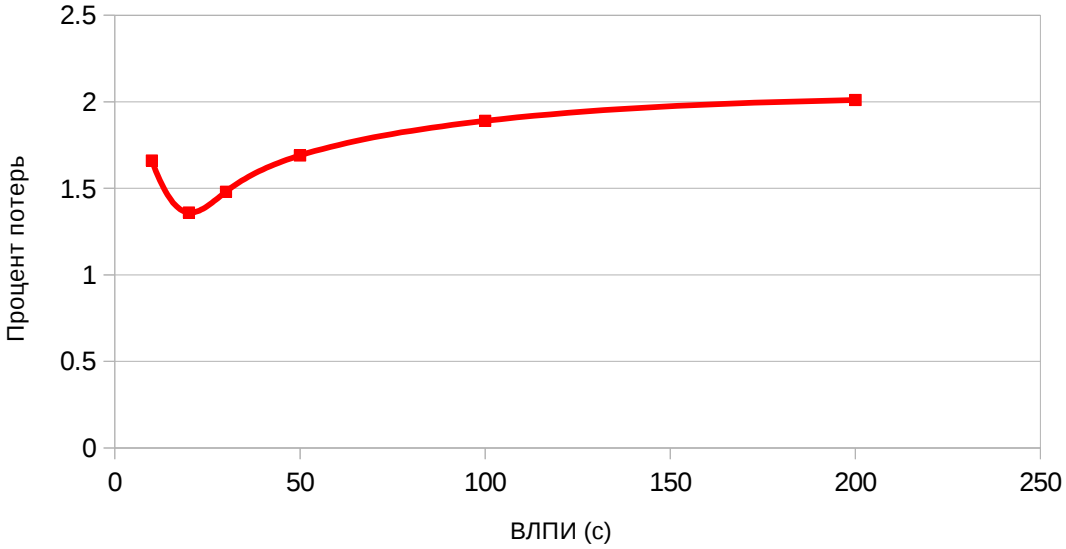

Рис. Б.4. Зависимость процента потерь метода УМАВ от значения ВЛПИ.

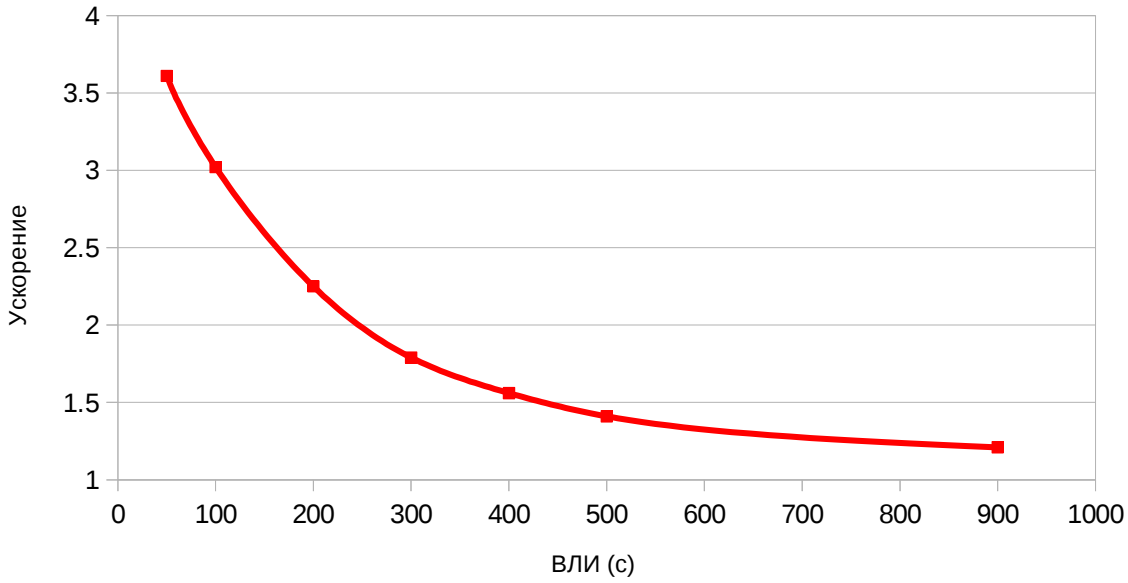

Рис. Б.5. Зависимость ускорения решения задач достижимости методом УМАВ от значения ВЛИ.

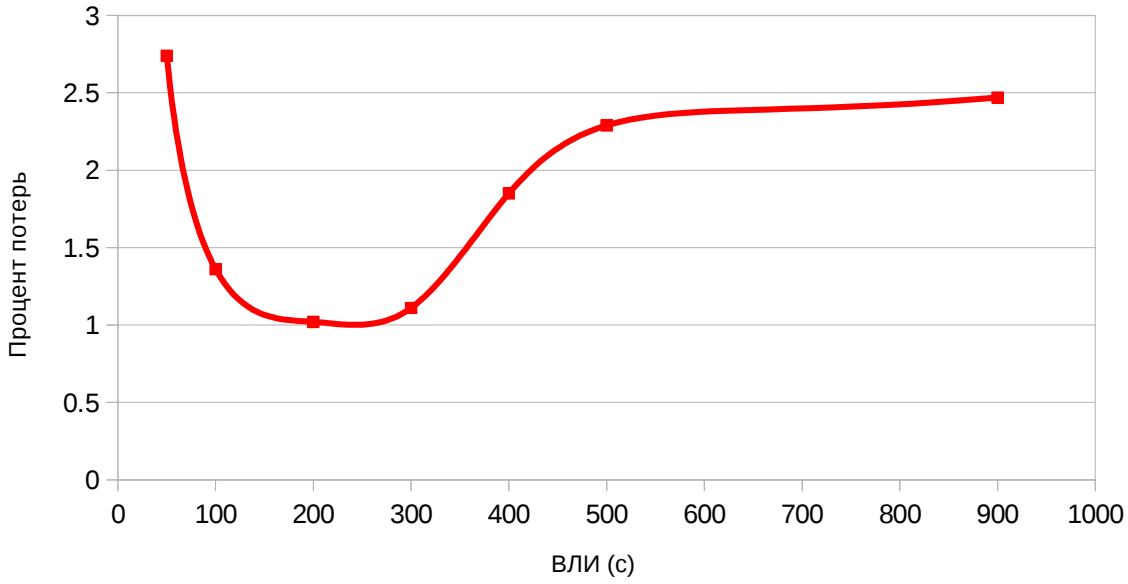

Рис. Б.6. Зависимость процента потерь метода УМАВ от значения ВЛИ.

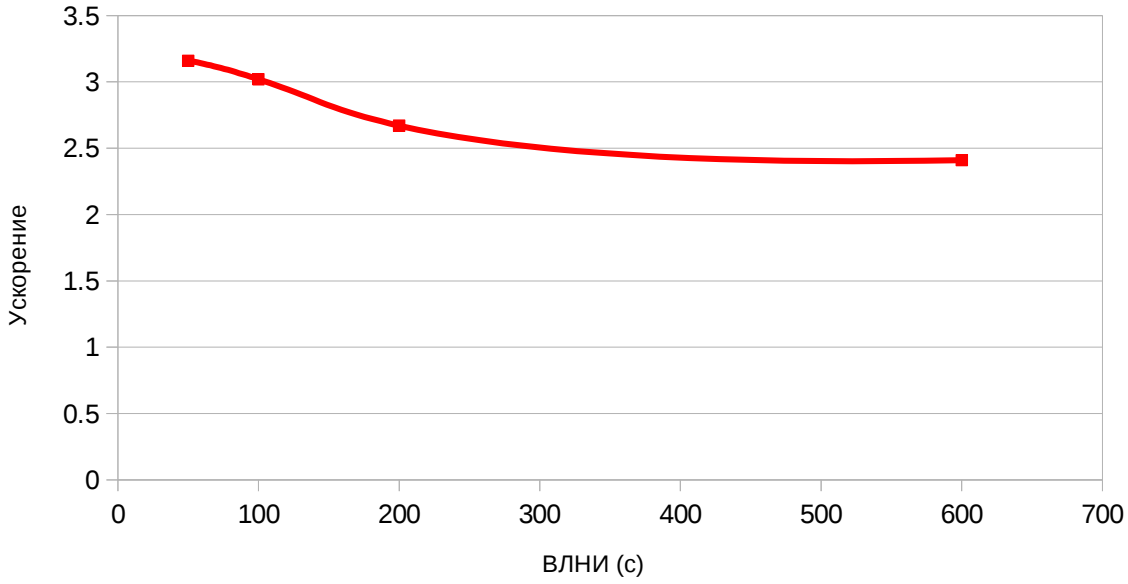

Рис. Б.7. Зависимость ускорения решения задач достижимости методом УМАВ от значения ВЛНИ.

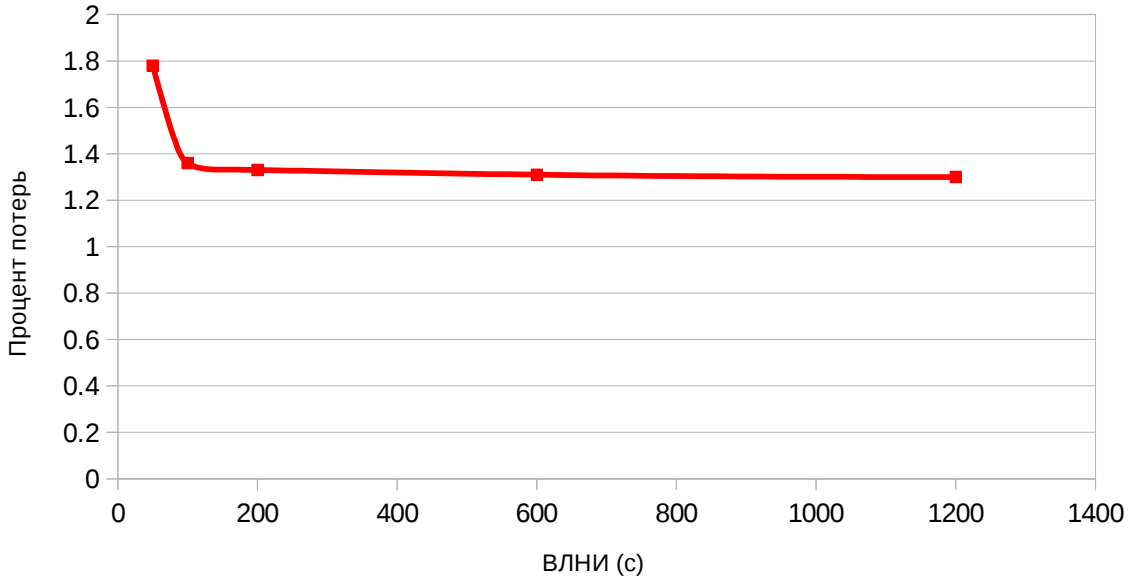

Рис. Б.8. Зависимость процента потерь метода УМАВ от значения ВЛНИ.

Для более удобного использования были предложены 4 различных конфигурации внутренних лимитов, в которых используются некоторые рекомендованные значения:

Л1. Базовая (ВЛУ=900, ВЛПИ=20, ВЛИ=100, ВЛНИ=100).

Л2. Без эвристики на ограничение первого интервала (ВЛУ=900, ВЛПИ=20, ВЛИ=100). Конфигурация повышает точность верификации за счет отключения эвристики.

Л3. С увеличенным лимитом для всех интервалов ППУ (ВЛУ=900, ВЛПИ=20, ВЛИ=225). Конфигурация повышает точность верификации за счет использования наиболее оптимального с точки зрения точности верификации значения ВЛИ.

Л4. Без эвристики на ограничение всех интервалов ППУ (ВЛУ=900, ВЛПИ=20). Конфигурация отключает эвристику, относящуюся к ВЛИ.

Результаты экспериментов с предложенными конфигурациями представлены в таблице Б.5 (приведено сравнение процессорного времени только для решения задач достижимости).

Метод	*Safe*	*Unsafe*	Потери / новые (%)	Процессорное время / ускорение
Последовательная верификация	118 703	667	–0 +0	3 889 000 1.00
УМАВ: конфигурация *Л1*	117 162	634	–1.36 +0.06	1 289 000 3.02
УМАВ: конфигурация *Л2*	117 241	634	–1.3 +0.07	1 655 000 2.35
УМАВ: конфигурация *Л3*	117 708	640	–0.92 +0.08	2 197 000 1.77
УМАВ: конфигурация *Л4*	115 844	640	–2.47 +0.09	3 221 000 1.21

Табл. Б.3. Сравнение различных конфигураций внутренних лимитов метода УМАВ.

Для конфигураций *Л1*, *Л2*, *Л3* наблюдается следующая зависимость: использование меньшего количества эвристик ведет к уменьшению процента потерь результата и уменьшению ускорения решения задач. Однако стоит

отметить, что ускорение при этом уменьшается гораздо быстрее, т. е. использование эвристик ведет к существенному уменьшению требуемых ресурсов при несущественных потерях во многом потому, что данные эвристики нацелены на раннее обнаружение ситуаций, при которых требование приводит к экспоненциальному росту числа состояний при построении абстракции.

УМАВ без эвристических внутренних лимитов (конфигурация *Л4*) демонстрирует существенное уменьшение ускорения верификации в сравнении с остальными конфигурациями и, как следствие, относительно высокий процент потерь результата. В данном случае без использования эвристик метод УМАВ не способен справиться с экспоненциальным ростом числа состояний и деградирует, поэтому его использование не рекомендуется.

Таким образом, конфигурация внутренних лимитов *Л1* для метода УМАВ является наиболее эффективной и масштабируемой; конфигурации *Л2* и *Л3* позволяют повысить точность верификации за счет снижения ее ускорения относительно конфигурации *Л1*. При задании внутренних лимитов в методе УМАВ вручную необходимо иметь в виду следующие рекомендации: ВЛУ всегда должен соответствовать базовому ограничению на проверку одного требования в методе последовательной верификации; ВЛПИ рекомендуется задавать порядка 20 процессорных секунд; ВЛИ должен быть в несколько раз меньше базового ограничения, но не меньше 100 процессорных секунд; ВЛНИ можно отключать для повышения точности, однако его использование приблизительно до 100 процессорных секунд ведет к существенному увеличению ускорения при незначительных потерях результата.

Б.2. Рекомендации по выбору параметров метода декомпозиции автоматной спецификации

Данный эксперимент был нацелен на сопоставление предложенных стратегий разбиения автоматной спецификации методом ДАС и на их сравнение с

методом АС. Для стратегии С*овместно-последовательная проверка* первый шаг был ограничен аналогично базовому ограничению по времени (т. е. 900 секунд процессорного времени), для стратегии *Релевантность* первый шаг был ограничен в 200 секунд (чем больше ограничение, тем больше накладных расходов и тем точнее будут выполняться следующие шаги), а второй – в 1 200 секунд (ограничение должно превосходить базовое ограничение на проверку одного требования для учета накладных расходов метода ДАС, однако слишком сильно увеличивать его не следует, поскольку второй шаг может занять все время верификации для потенциально неразрешимых задач). Результаты эксперимента представлены в таблице Б.4 (приведено сравнение процессорного времени только для решения задач достижимости). Стратегия без разбиения спецификации на группы требований (*Совместная проверка*) не использовалась, т. к. представляет собой вариацию метода пакетной верификации с автоматными спецификациями.

Метод	*Safe*	*Unsafe*	Потери / новые (%)	Процессорное время / ускорение	Разбиения / *l*
Метод АС	118 929	680	–0 +0	3 434 000 1	121 230 30
Метод ДАС (*ПП*)	117 830	659	–0.96 +0.04	3 484 000 0.99	121 230 30
Метод ДАС (*СПП*)	117 915	672	–0.91 +0.07	3 508 000 0.98	16 497 4.08
Метод ДАС (*Релевантность*)	118 386	673	–0.49 +0.04	1 373 000 2.5	6 982 1.73

Табл. Б.4. Результаты сравнения различных стратегий разбиения в методе декомпозиции автоматной спецификации (ПП – *Последовательная проверка*, СПП – *Совместно-последовательная проверка*, *l* – среднее число разбиений для решения одной задачи).

Стратегия *Последовательная проверка*, которая отличается от метода АС только переиспользованием ГПУ программы и кэшей верификатора и, следовательно, должна быть эффективнее, на практике демонстрирует

незначительное замедление и теряет порядка 1% результата. Данный результат объясняется тем, что накладные расходы на создание и поддержание достаточно большого числа разбиений в методе ДАС превосходят преимущества от переиспользования ГПУ и кэшей верификатора. Данную стратегию рекомендуется использовать для небольшого количества требований (5-10) вместо метода АС (и базового метода верификации) для более эффективной верификации без потери результата за счет переиспользования кэшей верификатора и ГПУ программы. При увеличении количества требований накладные расходы метода ДАС превосходят преимущества переиспользования в данной стратегии разбиения.

Стратегия *Совместно-последовательная проверка* позволяет решить несколько больше задач, чем стратегия *Последовательная проверка*, однако требует для этого несколько больше ресурсов. Данный результат объясняется тем, что стратегия *Совместно-последовательная проверка* может быстрее стратегии *Последовательная проверка* решить большое количество задач в одном разбиении (что характеризуется существенным снижением числа разбиений), однако если задача не была решена за первый шаг, то соответствующие ему ресурсы были потрачены напрасно, а сам результат забывается. Данную стратегию не рекомендуется использовать.

Таким образом, по умолчанию для метода ДАС рекомендуется использовать стратегию разбиения *Релевантность*; для более точной верификации относительно малого количества требований (5-10), часто приводящих к экспоненциальному росту числа состояний, рекомендуется стратегия *Последовательная проверка*.

Б.3. Рекомендации по разбиению спецификации на группы требований

Предлагается 2 эвристических способа разбиения спецификации, состоящей из достаточно большого количества требований, для того, чтобы уменьшить

количество потерь и время верификации. Данное разбиение главным образом имеет смысл для метода УМАВ, поскольку в методе ДАС данная проблема решается автоматически.

Наиболее простой способ заключается в построении случайного разбиения. Пусть, например, имеется N требований для верификации и K>1 узлов (вычислительных машин или ядер процессора). В данном случае можно на каждом узле создать задачу достижимости для проверки не более чем N/K произвольных требований. С одной стороны, в большинстве случаев такое разбиение поможет предотвратить большинство конфликтов между проверкой различных требований и улучшить результат. С другой стороны, при увеличении K суммарное количество требуемых ресурсов будет увеличиваться пропорционально K. В вырожденном случае (т. е. при N=K) метод превращается в последовательную верификацию.

Другой способ разбиения основан на вычислении «сложности» каждого проверяемого требования. Под сложностью требования в данном случае понимается некоторая оценка, которая вычисляется для определенного набора задач. Для получения данной оценки в общем случае необходимы результаты верификации, возможно, предыдущих версий той же программы (например, полученные при проведении регрессионной верификации). Например, в качестве сложности можно взять оценку числа релевантных требованию задач или общее время верификации. После получения подобной оценки для каждого требования устанавливается некоторый порог сложности и все требования разбиваются на 2 группы – простые (т. е. для которых оценочная сложность ниже заданного порога) и сложные. Сам порог необходимо выбирать таким образом, чтобы выделить максимальное количество простых требований, для которых возможна минимизация конфликтов при совместной верификации. При верификации «простой» группы требований с помощью метода УМАВ будет достигнуто максимальное ускорение и минимально возможные потери результата, при

верификации «сложной» группы требований – относительно большие потери и минимальное ускорение. Однако суммарно при достаточно высоком проценте «простых» требований (порядка 75% и выше) будет получено большее ускорение и меньшее количество потерь, нежели при верификации всех требований методом УМАВ без подобного разбиения спецификации (см. п. 5.8). В вырожденном случае возможна верификация «сложной» группы требований с помощью последовательной верификации, что дополнительно уменьшит количество потерь, и потребует значительно больше ресурсов.

Результаты последовательной верификации каждого из проверяемых требований представлены в таблице Б.5. Данная статистика помимо числа успешно решенных задач (вердикты *Safe* и *Unsafe*) показывает также и число нерешенных задач (*Unknown*), затраченные ресурсы и оценку[18] релевантности требований (см. п. 3.1), что может быть использовано для оценки сложности каждого из требований на выбранном наборе задач. В эксперименте из п. 5.8 требования были разбиты на основе оценки их релевантности (см. табл. Б.5) на две группы, которые содержали 22 «простых» требования (значение релевантности меньше 160) и 8 «сложных», что позволило понизить как потери метода, так и требуемые ресурсы. Стоит отметить, что любое подобное разбиение будет иметь смысл только для рассматриваемого набора задач и в общем случае не гарантирует улучшения результата. Поэтому данный метод имеет смысл использовать только при заранее известных характеристиках проверяемых требований и верифицируемых программ (в частности, в регрессионной верификации).

18 Только для успешно решенных задач (т. е. для которых были получены вердикты *Safe* или *Unsafe*).

Требование	Релевантность	*Safe*	*Unsafe*	*Unknown*	Процессорное время
linux:alloc:irq	783	3 985	0	56	116 000
linux:alloc:spin lock	1 048	3 970	5	66	134 000
linux:alloc:usb lock	4	3 999	0	42	97 000
linux:bitops	131	3 970	6	65	121 000
linux:blk:queue	25	3 976	15	50	117 000
linux:blk:request	11	3 983	9	49	114 000
linux:chrdev	63	3 975	10	56	120 000
linux:class	91	3 962	3	76	143 000
linux:completion	214	3 924	63	54	108 000
linux:gendisk	30	3 969	19	53	119 000
linux:idr	36	3 961	24	56	123 000
linux:iomem	356	3 756	132	153	222 000
linux:mmc:sdio_func	11	3 990	4	47	112 000
linux:module	118	3 906	75	60	128 000
linux:mutex	970	3 916	56	69	157 000
linux:netdev	178	3 945	1	95	159 000
linux:rculock	113	3 983	6	52	118 000
linux:rculockbh	17	3 992	2	47	113 000
linux:rculocksched	2	3 994	0	47	114 000
linux:rtnl	55	3 990	4	47	112 000
linux:rwlock	83	3 989	2	50	127 000
linux:sock	18	3 994	0	47	113 000
linux:spinlock	1 048	3 909	22	110	210 000
linux:srculock	1	3 992	0	49	115 000
linux:sysfs	134	3 926	46	69	135 000
linux:usb:coherent	68	3 943	38	60	133 000
linux:usb:dev	60	3 939	35	67	142 000
linux:usb:gadget	111	3 993	0	48	113 000
linux:usb:register	302	3 991	0	50	109 000
linux:usb:urb	155	3 881	90	70	145 000
Всего	6 236	118 703	667	1 860	3 889 000

Табл. Б.5. Результаты верификации всех модулей ядра Linux относительно каждого из 30 требований методом последовательной верификации.

Кроме того, используя разбиение спецификации на группы требований, можно ослабить ограничения из п. 2.1.1, убрав пункт 4. В данном случае в одну группу помещаются те требования, для которых пункт 4 ограничений из п. 2.1.1 выполнен. Это позволяет расширить круг требований, для которых применимы все предложенные методы.

Таким образом, разбиение спецификации на группы требований рекомендуется использовать только при регрессионной верификации (например, если известна оценка сложности требований для предыдущих версий проверяемых программ) или при нарушении пункта 4 ограничений из п. 2.1.1 для части требований.

Приложение В
Доказательство теорем и утверждений

В данном приложении приведены доказательства используемых вспомогательных утверждений и теорем.

Утверждение 1. Полнота и корректность верификации требований, удовлетворяющих ограничениям из п. 2.1.1, не изменяются при подготовке задач достижимости относительно композиции требований в сравнении с подготовкой задач достижимости относительно каждого требования в отдельности.

Доказательство. Данное утверждение состоит в том, что после выполнения объединения моделей требований и подготовки задачи достижимости относительно объединенной модели результат верификации будет таким же, так и до объединения (т. е. если программа была корректна относительно требования, то она и останется корректной, если в программе содержалось нарушение требования, то оно также может быть найдено и относительно объединенной модели). Заметим, что сама модель (согласно ограничениям из п. 2.1.1) не изменяет пути выполнения исходной программы. Кроме того, алгоритм объединения моделей требований оставляет все добавляемые в моделях проверки и не способен добавлять или отсекать пути выполнения программы. Поэтому добавление в задачу достижимости дополнительных проверок (относительно других требований) не изменит исходные пути выполнения программы. Таким образом, корректность программы относительно требования (т. е. отсутствие пути выполнения программы, содержащего метку ошибки) или его нарушение в программе (т. е. наличие пути выполнения программы, содержащего метку ошибки) инвариантны относительно объединения моделей требований.

Теорема 1. При верификации требований, удовлетворяющих ограничениям из п. 2.1.1, полнота и корректность алгоритма MAB не изменяются относительно

метода последовательной верификации.

Доказательство. Пусть дано N требований для верификации в программе P. Для доказательства данной теоремы необходимо показать, что алгоритм MAB относительно метода последовательной верификации не может привести к пропуску ошибок (т. е. вердикт *Unsafe* в методе последовательной верификации не переходит в вердикт *Safe* в алгоритме MAB) и появлению новых ложных сообщений об ошибках (т. е. вердикт *Safe* в методе последовательной верификации не переходит в вердикт *Unsafe* в алгоритме MAB).

Из условия для проверяемых требований справедливо утверждение 1, т. е. объединение их моделей не влияет на полноту и корректность верификации.

Из получения вердикта *Safe* в алгоритме MAB (т. е. доказательство того, что не существует такого пути выполнения программы, содержащего хотя бы одну из все еще проверяемых меток ошибок) следует доказательство корректности каждого из проверяемых утверждений (т. е. недостижимости одной из меток ошибок). Из нахождения вердикта *Unsafe* в алгоритме MAB (т. е. нахождение пути выполнения программы, содержащего метку ошибки) также следует возможность нахождения точно такого же пути выполнения программы при проверки только одной метки ошибки (метод последовательной верификации).

Кроме того, стоит заметить, что все добавляемые в алгоритме MAB действия не способны добавлять или удалять пути выполнения исходной программы. Действительно, действия *создание утверждений*, *проверка ВЛУ*, *смена ППУ* изменяют только внутреннее представление утверждений, а *отключение ППУ* удаляет те пути выполнения программы, в которых осуществлялась проверка отключенного утверждения (иначе говоря, содержала добавляемые моделью требования действия). Однако стоит заметить, что подобные действия не нужны для верификации остальных требований, поскольку проверяемые модели требований удовлетворяют ограничениям из п. 2.1.1.

Утверждение 2. Для требований, удовлетворяющих ограничениям из п. 2.1.1, метод АС полностью совпадает по возможностям их формализации с методом инструментирования.

Доказательство. Для доказательства данного утверждения необходимо показать, что все возможности инструментирования (например, на основе аспектно-ориентированного расширения языка C [47]) с учетом ограничений из п. 2.1.1 могут быть реализованы в методе АС.

Возможность выполнять переход в автомате на основе любого имени в программе (переменной, функции, макрофункции и т. д.) позволяет моделировать привязку некоторых действий (т. е. модельных функций из инструментирования) к точкам использования в программе. В качестве подобных действий может быть добавление и изменение значений добавленных переменных. Проверка утверждений моделируется с помощью переходов в состояние ERROR при выполнении условия из конструкции ASSUME, которая может содержать сколь угодно сложное выражение на языке C, состоящее из добавленных переменных, возвращаемого значения и параметров исходной функции (макрофункции), которое должно поддерживаться статическим верификатором (соответствует третьему требованию из п. 2.1.1). Отсечение невыполнимых путей в программе (например, моделирование возвращаемого значения, которое может принимать только неположительные значения) производится с помощью перехода во вспомогательное состояние, из которого нет переходов:

```
STATE USEFIRST Stop:
  TRUE -> GOTO Stop;
```

(при условии удовлетворения требования 2 из п. 2.1.1).

Помимо этого, аналогично заданию составных моделей требований с помощью инструментирования возможно использование метода АС. В частности, автомат может быть параметризован по заданному шаблону для генерации

нескольких автоматов, представляющих одно требование (например, для проверки корректности вызовов функций из заданного списка).

Утверждение 3. При верификации требований, удовлетворяющих ограничениям из п. 2.1.1, полнота и корректность метода АС не изменяются относительно метода последовательной верификации верификации на основе инструментирования.

Доказательство. Согласно утверждению 3, возможности по формализации требований у обоих методов эквивалентны. Помимо этого, метод АС основан на наблюдательных автоматах, для которых данное утверждение выполняется. При этом все расширения наблюдательных автоматов укладываются в ограничения из п. 2.1.1 (конструкция ENCODE не может модифицировать пути выполнения исходной программы, а конструкция ASSUME должна содержать только те операторы, которые поддерживаются верификатором). Следовательно, данное утверждение выполняется и для расширенных наблюдательных автоматах, т. е. для метода АС.

Теорема 2. При верификации требований, удовлетворяющих ограничениям из п. 2.1.1, полнота и корректность метода ДАС не изменяются относительно метода АС.

Доказательство. Как и в теореме 1 необходимо показать, что в методе ДАС невозможны появление новых ложных сообщений об ошибках и пропуск ошибок относительно метода АС.

Пусть спецификация состоит из N требований и дана некоторая произвольная стратегия разбиения S, для которой известно, что в каждом новом множестве разбиений максимальное количество требований в одном разбиении уменьшается. Обозначим данное максимальное количество требований в разбиении через R. Докажем данную теорему методом математической индукции.

1. Базис индукции R=1. В данном случае каждое требование помещается в

отдельное разбиение, т. е. верификация каждого разбиения (шаг алгоритма ДАС) полностью аналогична методу АС. На третьем шаге алгоритма ДАС либо для требования доказывается корректность, либо находится его нарушение, либо исчерпываются выделенные ресурсы, т. е. данный шаг полностью аналогичен методу АС. При этом каждое требование обязательно получит вердикт и после верификации каждого разбиения алгоритм ДАС завершается, предоставив результат, аналогичный методу АС.

2. Шаг индукции. Пусть для R=k-1 (1<k<N) утверждение теоремы верно, докажем его для R=k.

Согласно шагу 3 алгоритма ДАС для каждого из верифицируемых разбиений возможно 3 варианта:

- Доказана корректность всех требований. С одной стороны, из этого следует корректность каждого из требований (т. е. из доказательства корректности в методе ДАС следует корректность в методе АС). С другой стороны, при удалении любого из данных требований также будет доказана их корректность (т. е. по предположению индукции, из доказательства корректности в методе АС следует возможность доказательства корректности в методе ДАС).
- Найдено нарушение требований. Из этого следует нарушение данных требований в методе АС (если при проверке нескольких автоматов один нарушен, то и при проверке отдельно одного автомата он будет нарушен). Аналогично и при комбинации данного автомата с любыми другими может быть найдено его нарушение (т. е. из предположения индукции следует, что нахождение нарушения требования в методе АС влечет за собой возможность нахождения данного нарушения и в методе ДАС). Если для всех остальных требований не была доказана корректность, то все они остаются в списке проверяемых требований и будут использоваться при

построении нового множества разбиений с R=k-1, для которого справедливо предположение индукции.

- Верификация требований исчерпала выделенные ресурсы. Если данное разбиение состояло только из одного требования, то данный результат аналогичен тому, что в методе AC данное требование исчерпывает ресурсы. В противном случае все требования остаются в списке проверяемых и будут использоваться при построении нового множества разбиений с R=k-1, для которого справедливо предположение индукции.