Моделирование и верификация политик безопасности управления доступом в операционных системах

Москва
Горячая линия – Телеком
2019

УДК 004.451.9:004.056
ББК 32.973.2-018.2я73
Б40

Р е ц е н з е н т ы : доцент кафедры защиты информации и криптографии Томского государственного университета, кандидат техн. наук, доцент *Д. Н. Колегов*; старший системный аналитик департамента перспективных технологий Лаборатории Касперского, кандидат техн. наук *Е. А. Рудина*; сотрудник Академии ФСО России, кандидат техн. наук, доцент *А. Н. Цибуля*.

А в т о р ы : П. Н. Девянин, Д. В. Ефремов, В. В. Кулямин, А. К. Петренко, А. В. Хорошилов, И. В. Щепетков

Б40 **Моделирование** и верификация политик безопасности управления доступом в операционных системах / П. Н. Девянин, Д. В. Ефремов, В. В. Кулямин и др. – М.: Горячая линия – Телеком, 2019. – 214 с.: ил.

ISBN 978-5-9912-0787-4.

Описан процесс разработки и верификации формальных моделей безопасности управления доступом в операционных системах и реализующих их программных компонентов на примере отечественной защищенной операционной системы специального назначения Astra Linux Special Edition. Этот процесс направлен на получение адекватной оценки характеристик защищенности и безопасности операционных систем и достижение высокого уровня доверия к полученной оценке. Помимо этого, монография знакомит читателя с современными технологиями и инструментами моделирования и верификации, используемыми в подобных процессах. Представленный процесс направлен на обеспечение выполнения требований ГОСТ Р ИСО/МЭК 15408 «Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий» и профилей защиты операционных систем общего назначения (типа «А») третьего и второго классов защиты.

Для специалистов в области защиты информации, преподавателей, аспирантов; будет полезна студентам, обучающимся по направлениям подготовки и специальностям УГНПС «Информационная безопасность».

ББК 32.973.2-018.2я73

Адрес издательства в Интернет WWW.TECHBOOK.RU

ISBN 978-5-9912-0787-4

Предисловие

Задачи обеспечения информационной безопасности в современном мире приобретают огромное значение по причинам все большего проникновения информационных технологий в жизнь общества и роста угроз экономике и безопасности от возможных потерь, к которым может привести намеренное или ненамеренное нарушение функционирования как отдельных устройств, так и информационной инфраструктуры на локальном или даже глобальном уровнях.

Важнейшим направлением исследований и разработок, которые призваны сократить риски реализации таких угроз, является создание специальных средств защиты информации и построения защищенных систем, которые, в первую очередь, нацелены на защиту и обеспечение надежной и бесперебойной работы элементов критической инфраструктуры страны. Одним из ключевых элементов инфраструктуры любой программной системы является операционная система, поскольку она лежит в основании программного стека в любом программном комплексе.

В 2013 году компания АО «НПО РусБИТех», которая уже более 10 лет производит и внедряет специальные защищенные версии операционной системы (ОС) семейства Linux — операционную систему специального назначения (ОССН) Astra Linux Special Edition, предложила объединить усилия специалистов из двух, достаточно далеких в то время областей исследований для решения практической задачи — выведения специальной операционной системы на самый высокий уровень требований к обеспечению защиты информации, который определяется современными российскими и международными стандартами. С этой целью была сформирована группа из специалистов по информационной безопасности (под руководством проф. П.Н. Девянина) и специалистов ИСП РАН, имеющих опыт как в теоретических вопросах формальной верификации программных моделей, так и в применении различных техник верификации к крупным программным системам, в том числе к различным ОС общего назначения и реального времени.

Изначально было ясно, что придется параллельно решать научные, методические и технические задачи проекта и координировать исследовательские работы с планами АО «НПО РусБИТех» по подготовке к сертификации новых версий ОССН. Такая координация

одновременно затрудняла систематическое развитие методов, инструментов моделирования и верификации и помогала выявлять наиболее сложные, практически значимые задачи в контексте проблем верификации и сертификации большого программного продукта.

За счет тесной интеграции исследовательских и практических работ к настоящему времени удалось разработать комплекс методик и их инструментальную поддержку, что, в свою очередь, позволило внедрить результаты этих работ в процесс производства и сертификации ОССН и осенью 2017 года получить сертификат ФСТЭК России на соответствие ОССН требованиям профиля защиты ОС общего назначения (типа «А») второго класса защиты [1–3]. Это первый и пока единственный в России опыт сертификации ОС общего назначения в соответствии с требованиями такого высокого уровня.

Данная монография является плодом коллективного труда. При ее написании активно использовались материалы статей авторов по отдельным вопросам. Достаточно много времени пришлось уделить вопросам согласования терминологии и структуры изложения, так как описывается не отдельная научная проблема, а достаточно сложный процесс, состоящий из нескольких этапов работ, на каждом их которых используются разные методы и разные инструменты. Описанный процесс не является эталоном, он может пересматриваться и оптимизироваться в том или ином измерении, его можно позиционировать как «референсный», т. е. его внедрение позволяет утверждать, что концепция строгого и даже формального подхода к построению политики безопасности управления доступом таких крупных систем, как ОС общего назначения, реализуема. Кроме того, референсный процесс в дальнейшем можно использовать как некоторую базу для сравнения и выбора при решении отдельных задач, возникающих в процессах производства и сертификации защищенных программных систем.

Авторы выражают благодарность компании «Акционерное общество «НПО РусБИТех» за постоянную поддержку данной работы. Особенно важный вклад в работу внес заместитель начальника департамента этой компании А.Л. Оружейников, который на протяжении всего этого времени осуществлял решение сложных задач координации работ, определения приоритетов и предоставления необходимой технической документации на ОССН.

Также необходимо отметить важный вклад сотрудников ИСП РАН М.У. Мандрыкина (развитие инструментов дедуктивной верификации Си-программ) и Н.Ю. Комарова (верификация компонентов механизма безопасности в ядре ОССН).

Неоценимое влияние на работу оказал академик В.П. Иванников, который до последних дней следил за ходом работ, увлекал своим энтузиазмом и торопил с решением центральных проблем верификации сложнейших механизмов защиты современных ОС.

1 Формальные методы в разработке и сертификации средств защиты информации

По мере развития и широкого проникновения информационных технологий во все сферы жизни все более острыми становятся проблемы обеспечения информационной безопасности. Гражданам, бизнесу, государственным учреждениям, обществу в целом все труднее защититься от угроз, связанных с отказами или взломом в результате преднамеренных атак разнообразных информационных систем.

Современный уровень науки и техники не предлагает никакого универсального средства для выявления и парирования всех возможных угроз информационной безопасности. Скорее всего, такое универсальное средство не удастся изобрести и в будущем. Вместе с тем за последние 50 лет было немало попыток создания технологий разработки «идеального» программного обеспечения (ПО), отвечающего всем предъявляемым к нему требованиям, в том числе и требованиям информационной безопасности, использование которого позволило бы обеспечить безопасное и надежное функционирование информационных систем. В частности, предлагались концепции «разработки программ, правильных по построению», методы строгого и формального анализа программ, методы математического доказательства корректности программ и многие другие.

Хотя «идеальное» ПО создавать до сих пор не удается, прогресс в этом направлении компьютерных наук значителен. Если еще 10–15 лет назад потенциал инновационных технологий был понятен только небольшим группам ученых, то в последнее время возможности новых методов разработки и анализа программ становятся доступны практикам. Современное состояние этой области отличается от состояния пятнадцатилетней давности по многим характеристикам. Во-первых, разработаны новые методы и инструменты анализа программ, которые (при современных, существенно возросших возможностях компьютеров) позволяют анализировать на предмет защищенности не только «учебные» примеры длиной до 100 строк программного кода, но и реальные сложные программы и даже программные комплексы объемом в миллионы строк кода.

Во-вторых, за это время в рассматриваемой области был выработан систематический подход к оценке информационной безопасности в целом. Общая идеология повышения уровня информационной безопасности сводится к тому, что защита целевой системы обеспечивается совокупностью мер, реализованных по протяжении всего жизненного цикла разрабатываемой, а потом эксплуатируемой системы. То есть, начиная с определения требований к системе, выбора архитектуры и проектирования, выбора инструментов разработки и поддержки жизненного цикла, должны выполняться работы, направленные на повышение уровня защищенности, уровня доверия к целевой системы. При этом важно, что качество и полнота работ, предписанных в соответствующих регламентах, должны быть проверяемы на соответствие объективным критериям, желательно при помощи программных инструментов, чтобы минимизировать влияние субъективных человеческих факторов на оценку достигнутого уровня доверия.

Основным стандартом в этой области является ГОСТ Р ИСО/МЭК 15408 «Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий», состоящий из трех частей:

Часть 1. Введение и общая модель [4].

Часть 2. Функциональные компоненты безопасности [5].

Часть 3. Компоненты доверия к безопасности [6].

С ростом уровня требований к надежности и безопасности системы естественным образом расширяется и набор техник ее защиты, и сложность аргументации, необходимой для достижения нужной степени доверия к ним. Набор требований к мерам защиты и подтверждению их корректности в части 3 ГОСТ Р ИСО/МЭК 15408 разбит на так называемые оценочные уровни доверия (ОУД). Минимальный набор требований, базирующийся на самых общих, неформальных подходах к подтверждению безопасности оцениваемой системы, содержится в первом оценочном уровне доверия (ОУД1), а наиболее широкий и строгий набор требований к удостоверению безопасности систем изложен в седьмом оценочном уровне доверия (ОУД7). Он, в частности, предусматривает формальное (в некоторых случаях полуформальное) обоснование корректности реализации механизма управления доступом, важнейшей части средств защиты информации (СЗИ), в рамках объекта оценки (ОО).

Данная монография посвящена вопросам моделирования и верификации политики безопасности в ОС. Операционные системы, в которых имеются дополнительные средства защиты информации по

сравнению со стандартными версиями ОС, будем называть «защищенными ОС» (понимая, что разные защищенные ОС могут существенно отличаться друг от друга по степени защиты). ОС служит базовым слоем ПО, предоставляя для остальных программ среду выполнения и интерфейсы для взаимодействия с аппаратной платформой и другими программами и определяя тем самым множество элементов архитектуры и технологий разработки приложений, работающих на данной ОС. Уровень защищенности ОС не может быть ниже, чем уровень защищенности всей системы в целом. Политика безопасности ОС определяет правила управления доступом субъектов — пользователей и работающих от их имени программ — к различным объектам — информационным ресурсам, файлам, каталогам, устройствам и т. д. При этом для систем, отвечающих самым высоким требованиям по защите информации, например соответствующим ОУД6 или ОУД7, требования стандарта ГОСТ Р ИСО/МЭК 15408 предписывают необходимость формального описания политики безопасности (в виде модели политики безопасности) и формального доказательства ее корректности.

Моделирование безопасности управления доступом и информационными потоками можно считать исторически первым научным направлением в современной теории компьютерной безопасности [7–9]. В рамках этого направления разработаны десятки, если не сотни формальных моделей, а наиболее известная из них модель Белла–ЛаПадулы [7] более 40 лет назад была реализована в механизме управления доступом ОС *Multics*. Вместе с тем до сих пор научным сообществом и практиками в области информационной безопасности в полной мере не определен ни сам термин «формальная модель политики безопасности», ни тем более не сформировано четких критериев наличия представления такой модели, ее верификации при сертификации средств защиты информации [10].

Впервые требование представления формальной модели механизма управления доступом было включено в 1985 г. в требования классов защиты, начиная с B2, стандарта *Trusted Computer System Evaluation Criteria* (*TCSEC*, «Оранжевая книга») [11]. Тогда в качестве примера такой модели в *TCSEC* была указана модель Белла–ЛаПадулы. В 1992 г. аналогичное требование вошло в раздел «Гарантии проектирования», начиная с третьего класса защищенности по Руководящим документам (РД) Гостехкомиссии (ныне ФСТЭК) России «Средства вычислительной техники. Защита от несанкционированного доступа к информации. Показатели защищенности от несанкционированного доступа к информации» [12]. Однако в до-

ступных источниках авторами не найдено свидетельств применения этого требования по существу.

Ситуация существенно изменилась с февраля 2017 года, когда ФСТЭК России были утверждены профили защиты ОС общего назначения (типа «А») [13, 14], основанные на «Требованиях безопасности информации к операционным системам» [15] и ГОСТ Р ИСО/МЭК 15408. Эти профили включают требования доверия (например, содержащиеся в компонентах доверия ADV_FSP.6 «Полная полуформальная функциональная спецификация с дополнительной формальной спецификацией», ADV_SPM.1 «Формальная модель политики безопасности», AVA_CCA_EXT.1 «Анализ скрытых каналов» и др.), которые ставят перед научным сообществом задачи по разработке соответствующих технологий их реализации, без чего успешное применение новых профилей защиты будет крайне затруднено. Кроме того, ожидается утверждение ФСТЭК России профилей защиты СУБД, содержащих аналогичные требования.

Разработка модели политики безопасности и ее верификация — это только часть сложного процесса оценки защищенности операционной системы и представления ее на сертификацию. В данной монографии авторы описывают опыт проведения работ по подготовке к сертификации на соответствие требованиям профиля защиты ОС общего назначения (типа «А») второго класса защиты ОССН Astra Linux Special Edition [1–3]. Представленный материал не описывает весь процесс такой подготовки, а концентрируется на вопросах систематического, строгого анализа, построения модели политики безопасности управления доступом и информационными потоками, верификации механизмов управления доступом, которые реализованы в конкретной ОССН. Тем не менее представленная схема процесса моделирования и верификации дает достаточно полную картину проблем и арсенала средств верификации, которые позволяют подготовить все необходимые материалы (свидетельства) для сертификации на соответствие профилям защиты, включающим соответствующие требования доверия.

Хотя в монографии описывается модель политики безопасности управления доступом Linux, собственно Linux не вносит существенной специфики (за исключением учета некоторых конкретных особенностей Linux, например наличия «жестких» ссылок на файлы и др.). Практически любая ОС может брать предложенную модель за основу без значительной доработки. Например, для пополнения модели описанием дискреционного управления доступом можно ввести соответствующие роли на основе уже описанного ролевого управле-

ния доступом. Привязка к Linux позволяет дать конкретные примеры и показать, как процесс моделирования и верификации может выполняться в реальных условиях.

Содержание монографии выстроено в соответствии с этапами подготовки ОССН к сертификации. Сначала описывается фаза разработки модели политики безопасности в строгой математической нотации, затем описывается схема ее перевода на формальный язык моделирования и разработки спецификации формальной модели, после чего излагается подход к верификации полученной модели, а также верификации и тестирования ее реализации непосредственно в программном коде ОССН.

В реальности, конечно, на каждой фазе этой работы могут встречаться проблемы, в частности могут обнаруживаться некоторые ошибки в спецификациях, что вызывает необходимость менять отдельные решения или даже возвращаться и пересматривать спецификации или доказательства корректности, которые были сделаны на предыдущих фазах процесса. Важно отметить, что в данном случае приходится работать как минимум с тремя различными нотациями: математической, формальной (в нашем случае Event-B) и на языке программирования Си (его диалектом, который используется при программировании компонентов ядра ОС Linux). Переход от одной нотации к другой это непростая работа, которая не всегда может быть полностью формализована и, соответственно, автоматизирована. Кроме того, нужно понимать, что, двигаясь от модели к реализации ядра ОС Linux, мы переходим от описания модели в математической нотации (около 300 страниц текста) к формальной на Event-B (около 5 тысяч строк кода) и далее к спецификации поведения реализации ядра, размер которого приближается к 20 млн строк кода на Си, из которых значительная часть явно или косвенно связана с реализацией механизма управления доступом ОССН. Последний «переход» самый трудный в плане его формализации и автоматизации.

Цель формализации требований к механизму управления доступом и систематической верификации как моделей, так и его реализации в ОССН — это повышение уровня доверия к данному механизму в ОО. Элементы технологической цепочки процесса верификации, представленные в данной книге, различаются по степени точности и гарантиям обеспечения информационной безопасности. Для такой большой и сложной системы, как ОССН, современный уровень технологий не позволяет дать гарантии полной защищенности, тем не менее систематичный и, по возможности, строгий или даже фор-

мальный подход к верификации моделей и программ существенно повышает уровень уверенности в обеспечении информационной безопасности, именно этого и требуют регламенты и стандарты, на которые опираются процедура сертификации операционных систем, «работающих» с государственной тайной.

Все фазы процесса представлены и описаны в данной монографии. Конечно, ее нельзя рассматривать как учебное пособие, изучения которого достаточно для того, чтобы специалист по информационной безопасности смог без каких-либо затруднений выполнить аналогичный проект применительно к собственному программному продукту.

Цель авторов состоит в том, чтобы показать пример использования современных достижений компьютерных наук для решения практической задачи, с которой все больше приходится сталкиваться специалистам по информационной безопасности и другим разработчикам, которые планируют сертификацию своих продуктов на соответствие требованиям с высоким уровнем доверия к их безопасности. Вместе с тем технологии верификации, которые описываются в книге, постоянно развиваются и уже сейчас являются достаточно зрелыми, чтобы ими могли пользоваться не только математики, но и инженеры, и программисты, а это значит, что эти технологии уже можно использовать в крупных программных проектах.

Модели и спецификации, о которых пойдет речь далее, а также используемые для их создания и верификации инструменты выложены в открытом доступе на сайте ИСП РАН* и могут быть скачаны для подробного изучения.

* http://www.ispras.ru/publications/2018/security_policy_modeling_and_verification/

2 Описание процесса моделирования и верификации механизма управления доступом операционной системы

Рассматриваемый процесс состоит из нескольких этапов, которые далее будут описываться последовательно. Изложение в целом следует логике регламентирующего документа ГОСТ Р ИСО/МЭК 15408-3 в той части, где описывается класс доверия ADV «Разработка». При этом сначала рассматриваются более абстрактные представления модели политики безопасности управления доступом, затем более детальное представление модели и в конечном счете рассматривается задача верификации механизма управления доступом в ядре ОС, где мы имеем дело уже с конкретной программной системой, реализованной на языке программирования Си.

Напомним читателю требования класса доверия «Разработка», в котором в соответствии с ГОСТ Р ИСО/МЭК 15408-3 [6] для функций безопасности объекта оценки (ФБО) и функциональных требований безопасности (ФТБ) указывается, что класс «Разработка» содержит шесть семейств доверия для структурирования и представления ФБО на различных уровнях детализации. Эти семейства включают в себя:

- требования к описанию (на различных уровнях детализации) проекта и реализации ФТБ (ADV_FSP «Функциональная спецификация», ADV_TDS «Проект ОО», ADV_IMP «Представление реализации»);
- требования к описанию архитектурно-ориентированных особенностей разделения доменов, обеспечения собственной защиты ФБО и невозможности обхода ФБО (ADV_ARC «Архитектура безопасности»);
- требования к модели политики безопасности и к прослеживанию соответствия между моделью политики безопасности и функциональной спецификацией (ADV_SPM «Моделирование политики безопасности»);
- требования к внутренней структуре ФБО, которые охватывают такие аспекты, как модульность, деление на уровни и минимизацию сложности (ADV_INT «Внутренняя структура ФБО»).

Далее в ГОСТ Р ИСО/МЭК 15408-3 отмечается, что при документировании функциональных возможностей безопасности ОО необходимо продемонстрировать два основных свойства. Первое свойство заключается в том, что определенная функциональная возможность выполняется правильно, согласно спецификации. Второе свойство, которое несколько сложнее продемонстрировать, заключается в том, что невозможно использовать ОО так, чтобы это привело к искажению или обходу функциональных возможностей безопасности. Два этих свойства требуют применения различных подходов к их анализу.

В рамках книги мы в первую очередь концентрируемся на исследовании первого свойства — спецификации функциональных возможностей безопасности, поэтому рассматриваем семейства «Функциональная спецификация» (ADV_FSP), «Проект ОО» (ADV_TDS) и «Моделирование политики безопасности» (ADV_SPM).

Изложение начинается с проблем построения и верификации модели политики безопасности управления доступом, затем рассматривается вопрос построения функциональной спецификации ОО и вопрос ее соответствия модели политики безопасности управления доступом.

После того как построена формальная модель, мы исследуем ее «формальным» образом, уже не рассматривая каких бы то ни было специфических особенностей из области информационной безопасности. Нас интересует структурная и семантическая согласованность отдельных частей формальной модели между собой, в частности то, что выполняются все инварианты при любой комбинации условий, которые могут возникать при выполнении того или иного правила перехода системы из состояния в состояние модели. Согласованность или консистентность модели в свою очередь означает, что политика безопасности, которую мы представили в виде формальной модели, отвечает требованиям информационной безопасности.

В [6] отмечается, что в функциональной спецификации ОО не описывается, каким образом реализуются функции безопасности, этот вопрос с той или иной степенью детальности рассматривается в семействе «Проект ОО» (ADV_TDS). В рамках ADV_TDS особенно важно исследовать структуру реализации механизма управления доступом. В связи с этим в главе 6 описываются модуль безопасности (Linux Security Module — LSM) — ключевой компонент механизма управления доступом ОССН и методы его формальной спецификации и верификации.

В последней, 7-й главе мы переключаемся на исследование вопроса о согласованности функциональных требований, в частности требований политики безопасности управления доступом в ОО с реальным наблюдаемым поведением ОО. Такого рода интегральная проверка сама по себе не может дать гарантий корректности реализации средств защиты информации в силу сложности и размера современной операционной системы, но она необходима, так как реализация сквозного способа проверки такого большого и сложного программного комплекса как операционная система в совокупности с механизмами управления доступом, конечно, существенно повышает доверие к надежности обеспечения информационной безопасности.

2.1. Этап I. Формализация модели политики безопасности управления доступом и ее верификация

В нашем случае ОО [4] служит механизм управления доступом ОС*, являющейся важнейшей частью СЗИ ОССН. Правила ограничения доступа к тем или иным информационным ресурсам составляют политику безопасности управления доступом. Ответственность за полную и корректную реализацию этих требований возлагается на механизмы, встроенные в ядро ОС. Можно считать, что разработка некоторого документа верхнего уровня, который строго и полно описывает требования к механизму управления доступом, это один из первых этапов создания (или сборки) системы в рамках классического подхода к проектированию программ «сверху-вниз», о полезности которого было много написано еще 70–80-е годы 20-го века. Заметим, как теоретики, так и практики, приверженцы этого подхода уже в те годы отмечали, что собственно проектирование «сверху-вниз» не дает гарантий построения «правильных» программ. Такие «гарантии», точнее, свидетельства, демонстрирующие или даже доказывающие «правильность» или «корректность» получаемого программного решения, может дать только тщательная проверка, ве-

* Как уже отмечалось, описывается конкретный опыт проведения работ по подготовке к сертификации ОССН Astra Linux Special Edition, поэтому, хотя по тексту может быть написано просто «операционная система», как правило, имеется в виду архитектура ОС Linux, а иногда и специфические решения, которые используются именно в ОССН. Вместе с тем общая схема процесса верификации достаточно мало зависит от специфики ОС.

рификация всех проектных и рабочих документов (артефактов) и проверка их взаимной согласованности.

Требование тотальной верификации приводит к тому, что и исходные требования как один из документов проекта также должны быть верифицированы. Заметим, что опыт разработки и верификации сложных программных систем как у нас в стране, так и за рубежом показывает, что на всех фазах создания ПО и во всех видах документации, включая сами требования к ПО, могут встречаться дефекты или, как обычно говорят в англоязычной литературе, несогласованности — отсутствие консистентности (inconsistencies). В нашем случае верификации собственно ОО необходимым образом должна предшествовать верификация и валидация требований к ОО, т. е. описание политики безопасности управления доступом и проверка консистентности этого описания.

По мере роста объема документации ОО, в частности требований к механизму управления доступом, степень уверенности в том, что в нем нет пропусков и несогласованностей, снижается. Если размер документа, описывающего требования, превышает размер 5–10 страниц, то, если не провести тщательную проверку, можно гарантировать, что ошибки в нем есть. В нашем случае такой документ включает в себя около 300 страниц математического текста с пояснениями на русском языке. Единственной возможностью существенно повысить согласованность (консистентность) документа является представление его содержания в некотором виде, которое можно проанализировать при помощи современных средств верификации. Таким представлением может быть формальная модель, изложенная на подходящем языке формальных спецификаций (далее будем говорить «на формальном языке»). При этом от «формальной» спецификации мы ожидаем, что одновременно она написана на формальном языке, семантика которого строго и однозначно определена, и имеются программные инструменты, которые позволяют анализировать и верифицировать спецификации на таком формальном языке.

Тем самым первым этапом процесса является построение сначала строгой модели политики безопасности управления доступом и затем приведение этой модели к «формальному» виду, т. е. разработке спецификации этой модели на формальном языке и далее верификации этой спецификации при помощи современных инструментов верификации для доказательства того, что ОО не может перейти в небезопасное состояние.

В принципе модель сразу может конструироваться на некотором

формальном языке, но опыт показывает, что специалистам по информационной безопасности удобнее сначала описать модель, пользуясь традиционной математической нотацией, и только потом переводить ее на формальный язык, понятный компьютеру. В целом этот вопрос — начинать с математической нотации или сразу писать на формальном языке — не принципиален. Но авторы склоняются к тому, что наличие двух представлений модели — математической и формальной — это полезная избыточность.

Первый этап завершается верификацией формальной модели при помощи, например, метода дедуктивной верификации и инструментов, которые помогают провести математическое, дедуктивное доказательство корректности/консистентности модели, представленной на выбранном формальном языке.

2.2. Этап II. Разработка спецификации системных вызовов ОС и доказательство ее соответствия с формальной моделью политики безопасности

На втором этапе необходимо связать требования к механизму управления доступом (модель политики безопасности управления доступом) с функциональной спецификацией ОО (рис. 2.1). Это требуется для того, чтобы решить еще одну задачу, предусмотренную требованиями ГОСТ Р ИСО/МЭК 15408, а именно, доказательство соответствия между формальной функциональной спецификацией и формальной моделью политики безопасности (доказательство «соответствия между моделью политики безопасности и функциональной спецификацией (ADV_SPM «Моделирование политики безопасности»)» [6]).

В нашем случае функциональной спецификацией ОО является интерфейс API (Application Program Interface) операционной системы, т. е. интерфейс системных вызовов.

Для спецификации API операционной системы можно использовать различные формальные языки. В нашем проекте мы выбра-

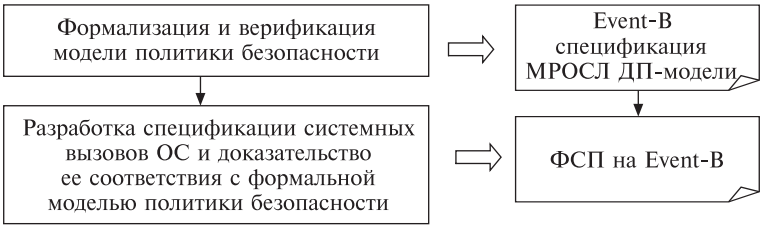

Рис. 2.1. Первые два этапа верификации механизма управления доступом ОС

ли тот же язык, который использовался для спецификации модели политики безопасности управления доступом. Если бы был выбран другой язык, то потребовался бы еще один переход от спецификации на одном языке к спецификации на другом. При этом потребовалось бы еще доказывать, что перевод выполнен корректно.

Сделаем три замечания:

- использование второго языка спецификаций и перевод с первого на второй имеют смысл, если спецификации на втором языке будут использоваться далее в процессе верификации ОО;
- спецификация системных вызовов может описывать не полную функциональность ОС, а лишь те требования к API, которые важны в контексте управления доступом, это так называемые частичные спецификации;
- после того как построена формальная функциональная спецификация API ОС, должно быть продемонстрировано соответствие данной спецификации требованиям модели политики безопасности. Если соответствие доказано, то можно гарантировать, что выполнение требований функциональной спецификации ОО гарантирует выполнение требований модели политики безопасности управления доступом. При этом нужно принять во внимание, что структура формальной модели политики безопасности управления доступом (набор типов данных и операций) отличается от структуры API ОС, поскольку системные вызовы оперируют с конкретными файлами, идентификаторами процессов и другими сущностями реальной ОС, а формальная модель политики безопасности оперирует с абстрактными информационными ресурсами, субъектами и объектами, между которыми определены правила управления доступом и его администрирования. Для формального анализа соответствия две спецификации нужно привести к некоторому «общему знаменателю». Эта задача не простая, но решаемая.

2.3. Этапы III и IV. Исследование механизма управления доступом ОС

После того как выполнены работы первого и второго этапа, мы имеем формальное, математическое доказательство того, что, во-первых, модель политики безопасности управления доступом консистентна и, во-вторых, функциональные требования к ОО согласованы с требованиями к политике безопасности управления доступом. И то, и другое необходимо для обеспечения соответствия уровням доверия ОУД6 и ОУД7. Однако пока не был рассмотрен вопрос

о том, отвечает ли реализация ОО тем требованиям, которые зафиксированы в спецификациях системных вызовов и/или в модели политики безопасности управления доступом.

Получить ответ на этот вопрос существенно сложнее, чем решить проблемы на первых двух этапах, поскольку размер и сложность реализации системных вызовов несоизмеримо больше в сравнении с моделями и их спецификациями, которые мы рассматривали выше. Современный уровень технологий позволяет математически строго верифицировать программы из класса ОС размером 10–20 тысяч строк на языке типа Си. Ядро ОС Linux сейчас содержит около 20 млн строк на Си. По этой причине для обеспечения уверенности в корректности реализации механизма управления доступом в ядре ОС Linux приходится искать некоторое компромиссное решение (что, в действительности, не противоречит требованиям стандартов).

Для того чтобы описать суть предлагаемого компромиссного решения, уточним, что мы понимаем под механизмом управления доступом в ОО, о какой архитектуре ОО идет речь.

Общая схема механизма управления доступом, которая рассматривается в данном случае, была предложена в проекте *Security-Enhanced Linux* (SELinux) [16], предполагающем встраивание специальных дополнительных точек контроля доступа ко всем информационным ресурсам, контролируемым ядром ОС. Эти точки не выполняют собственно контроль, а лишь играют роль «перехватчиков», они обращаются к специальному модулю ядра *Linux Security Modules* (LSM), где, собственно, и выполняется проверка корректности доступа. В случае некорректного доступа LSM выдает запрет на доступ, что обеспечивает безопасность доступа к защищаемым ресурсам. Логика проверок, которые выполняются модулем LSM, должна полностью отвечать конкретной политике безопасности управления доступом, установленной для конкретного механизма управления доступом.

Таким образом, корректность механизма управления доступом должна достигаться за счет того, что, во-первых, все функции модуля LSM, к которым они обращаются, реализованы корректно и, во-вторых, в ядре ОС правильно и в полной мере расставлены «перехватчики», вызывающие функции LSM.

Поскольку мы концентрируемся на решении только первой части задачи, схема процесса верификации механизма управления доступом будет выглядеть так, как показано на рис. 2.2.

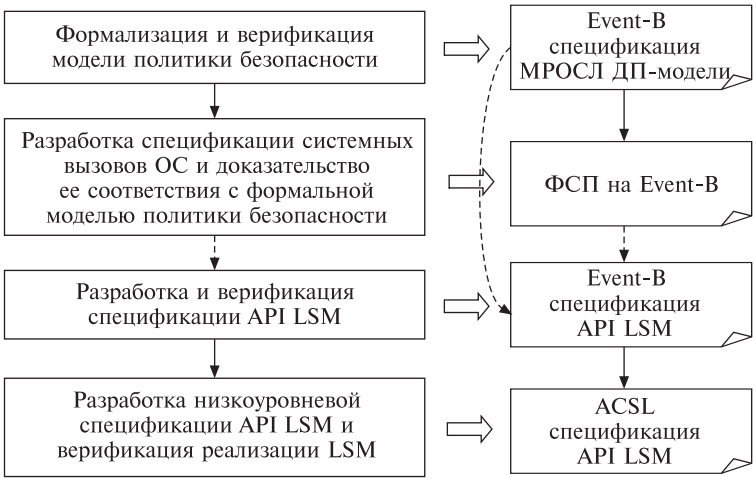

Рис. 2.2. Процесс верификации механизма управления доступом ОС. Этапы I–IV

Финальный этап представленного выше процесса в качестве результата дает доказательство того, что реализация модуля LSM корректна, т. е. она отвечает требованиям к спецификациям этого модуля. Однако, как показано на рис. 2.2, финальному этапу предшествует этап разработки спецификаций и верификации собственно спецификаций. Необходимость этого промежуточного этапа процесса объясняется следующими обстоятельствами.

Хотя модуль LSM, в принципе, выполняет те же проверки, что и проверки, включенные в модель политики безопасности управления доступом, набор операций, составляющих API модуля, и структуры данных, с которыми работают эти операции, существенно отличаются от операций и данных как в модели политики безопасности, так и в API ОС. Если в случае описания соответствия между моделью политики безопасности и API ОС мы отметили, что задача анализа соответствия сложная, но решаемая, то в случае лишь разработки описания соответствия API LSM с одной из спецификаций, рассмотренных выше, эту задачу можно назвать неразрешимой, размер формального описания такого соответствия составлял бы миллионы строк.

Из этого следует вывод, что спецификацию API LSM нужно строить на основе понимания требований к этому модулю. «Понимание» — объект не формальный. По этой причине на рис. 2.2 связь между этапом II и этапом III обозначена при помощи пунктирной линии. В силу неформальности перехода от второго этапа к третьему

без дополнительных исследований мы не можем утверждать, что из выполнения требований спецификации к каждой из функций из API LSM будет следовать выполнение общих требований по защите информации, инвариантов безопасности (что мы доказывали при исследовании модели политики управления доступом).

Для решения это проблемы к задаче построения формальной спецификации API LSM добавляется задача ее верификации, т. е. доказательства ее консистентности и доказательства сохранения инвариантов безопасности при выполнении произвольных обращений к API LSM. Такого рода доказательство можно выполнить при помощи тех же инструментов, что использовались на первом и втором этапах, но это означает, что спецификацию API LSM придется писать на языке Event-B. Собственно в этом и состоит содержание III этапа: разработка спецификаций API LSM на Event-B и ее верификация.

Содержание IV этапа — разработка спецификации API LSM на уровне языка программирования и верификация уже не модели, а реализации модуля LSM. Спецификация в этом случае уже разрабатывается на языке спецификаций ACSL, специально предназначенного для описания требований к интерфейсам программ, реализованных на языке Си. Для верификации реализации модуля LSM, естественно, также нужно использовать другие инструменты, которые предназначены для верификации Си-программ.

Переход от спецификаций API LSM на Event-B к спецификациям на ACSL трудно сделать совсем формальным, однако здесь риск получить неадекватную спецификацию невелик, поэтому на рис. 2.2 связь между III и IV этапами обозначена не пунктирной, а сплошной линией.

Для того чтобы парировать возможные риски, вызванные перечисленными выше ограничениями предложенной схемы, можно использовать технику динамической верификации (мониторинг или run-time verification). В этом случае нужно проверить на реальных сценариях использования ОС, соответствует ли функционирование модуля LSM и весь механизм управления доступом требованиям к нему. В этом случае придется сопоставлять последовательности вызовов функций в формальной модели политики безопасности управления доступом или в модели, специфицирующей системные вызовы (функциональная спецификация ОО), с последовательностью вызовов функций LSM. При сопоставлении последовательностей анализируется порядок вызовов, их аргументы и результаты вызовов (рис. 2.3).

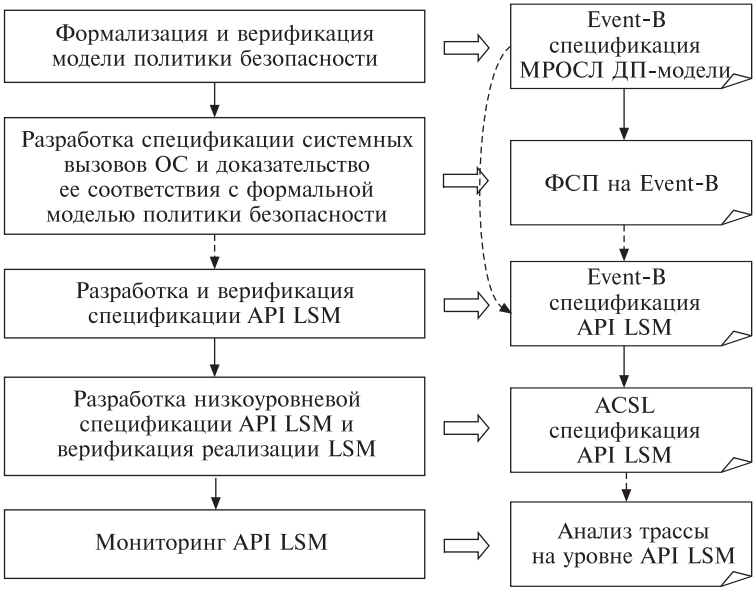

Рис. 2.3. Схема процесса верификации механизма управления доступом ОС. Вариант 1

Однако у данной схемы имеется недостаток — сопоставить последовательность выполнения системных вызовов на уровне API ОС с последовательностью вызовов LSM-функций в каждом конкретном случае и тем более в общем случае чрезвычайно трудно. Кроме того, проверки на уровне вызовов модуля LSM принципиально неполны, так как теоретически, что в некоторых случаях ядро ОС не обращается к нужной функции LSM.

В связи с этим приходится выбирать другой вариант организации сквозной проверки соответствия поведения ОО требованиям модели политики безопасности управления доступом. Эта схема представлена на рис. 2.4. В этой схеме трассы наблюдаются на уровне системных вызовов, а не на уровне модуля LSM. Критерий корректности трассы — выполнение требований функциональной спецификации системных вызовов (ФСП). В силу результата второго этапа процесса верификации выполнение требований ФСП гарантирует выполнение требований модели политик безопасности управления доступом в целом. Тем самым, если хотя бы одну трассу не удается сопоставить с ФСП, нужно делать вывод о том, что требования модели политики безопасности нарушаются. Если все трассы удается сопоставить с ФСП, то мы получаем набор объективных свидетельств о выполнении требований политики безопасности.

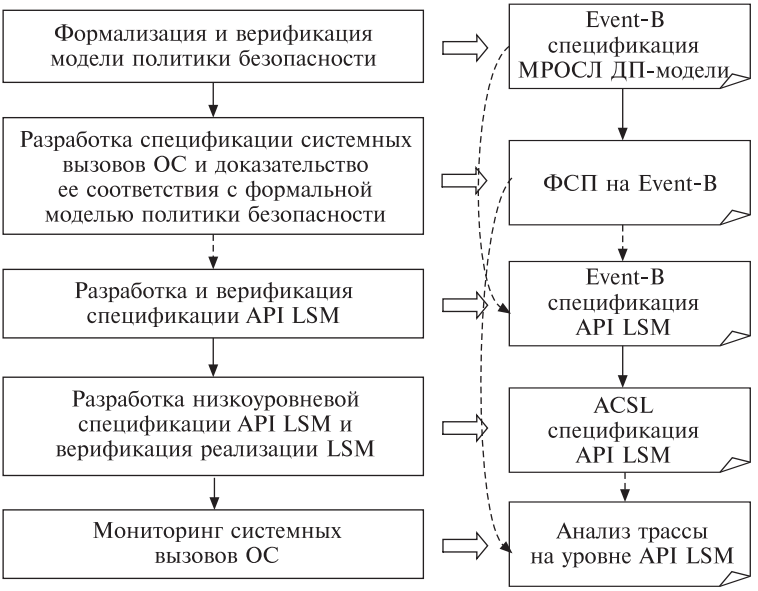

Рис. 2.4. Схема процесса верификации механизма управления доступом ОС. Вариант 2

Применение последней схемы проверки позволяет проверить соответствие всех артефактов, включенных в описываемый процесс верификации. При обнаружении несоответствия трассы требованиям функциональной спецификации ОО следует делать вывод о том, что обнаружен дефект либо в артефактах (компонентах ОО), либо в процедурах/инструментах верификации и/или артефактах, создаваемых при верификации. Каждый случай выявленного несоответствия следует анализировать, а затем устранять причину несоответствия. К сожалению, анализ и коррекцию здесь нельзя сделать автоматически, но дополнительная, сквозная проверка дает дополнительную уверенность в достоверности проведенной верификации и заодно является средством валидации самой формальной модели политики безопасности управления доступом.

В рамках исследований ОССН Astra Linux Special Edition был выбран последний вариант схемы процесса верификации механизма управления доступом, представленный на рис. 2.4.

2.4. Замечание об используемой терминологии

Здесь можно сказать, что основные понятия неформально уже введены или, по крайней мере, упомянуты. Однако авторы понима-

ют, что терминология для многих читателей непривычна, поэтому здесь следует дать более систематическое и более строгое определения использующихся понятий. Основной причиной трудности восприятия содержания этой главы является перегруженность терминов «модель» и «спецификация», которые приходится употреблять в разных контекстах, имея в виду разные сущности. Постараемся разъяснить трактовку этих терминов.

Термин «модель» употребляется там, где мы строим некоторое ментальное представление или некоторый артефакт (например, документ на естественном, полуформальном или формальном и машиночитаемом языке), которые позволяют нам представить отдельные стороны, характеристики некоторого реального программного объекта (например, функции, реализующей системный вызов ядра ОС). Этот же термин используется там, где мы рассматриваем не реальный программный объект, а некоторую абстракцию, некоторое обобщение таких объектов (например, действия по получению права доступа к некоторой сущности, которая может быть файлом, каталогом, портом канала связи и так далее).

В профессиональной речи часто термин «модель» понимается как синоним термина «спецификация», хотя буквально «спецификация» это лишь точное, детальное описание чего-либо, в частности описание модели или описание реальной программной сущности или программного процесса. Понятно, что описание программной сущности (например, некоторой функции на языке Си), это не то же самое, что сама сущность. Однако в случае рассмотрения спецификаций на Event-B «модель» и «спецификация» (модели) может значить одно и то же.

Спецификацию функции можно трактовать как требование к реализации функции. Если в спецификации функции мы описываем не все, что можно узнать о ней, можно говорить, что спецификация представляет некоторую модель функции (например, спецификация определяет каков должен быть результат при вызове функции с тем или иным аргументом, но не определяет время вычисления, необходимые ресурсы и даже алгоритм вычисления).

Математическая модель — модель, например политики безопасности управления доступом, представленная в математической нотации. Она формальна с точки зрения математиков, но неформальна с точки зрения программистов, которые под формальной нотацией понимают такую нотацию, которая является машиночитаемой и однозначно интерпретируемой соответствующими программными инструментами.

Формальная модель (или формальная спецификация) — модель, представленная в формальной машиночитаемой нотации, например в Event-B. Термин «формальный» здесь означает, что для такого представления моделей могут быть разработаны или уже разработаны инструменты, которые анализируют такие модели, проводят их верификацию, а иногда и трансформацию в программы на языках программирования.

Функциональная спецификация ОО, в соответствии с определением, приведенном в стандарте ГОСТ Р ИСО/МЭК 15408, — это «полная полуформальная функциональная спецификация с дополнительной формальной спецификацией». В нашем подходе полуформальные спецификации мы используем как некоторый исходный артефакт, на основе которого строится система формальных спецификаций. Формализация спецификаций нужна для того, чтобы можно было использовать те или иные программные инструменты для анализа/верификации самих спецификаций/моделей или для анализа соответствия между спецификациями/моделями различного уровня детальности или между спецификациями (требований) и программных реализаций.

ФСП. В регламентирующих документах FSP означает полуформальную спецификацию интерфейсов ОО. В данной книге термин ФСП, как правило, используется для обозначения формальной функциональной спецификации ОО и при этом иногда мы пишем «формальная ФСП», чтобы подчеркнуть возможность использования ФСП в качестве исходных данных для инструментов моделирования и анализа.

3 Модель политики безопасности управления доступом — требования к составу и структуре модели. Базовый уровень МРОСЛ ДП-модели

3.1. Требования к составу и структуре модели

Основу системы требований по созданию на основе ОС общего назначения механизма управления доступом, сертификации ОС на соответствие «Требованиям безопасности информации к операционным системам», утвержденным приказом ФСТЭК России от 19 августа 2016 г. № 119 [15], составляют шесть профилей защиты ОС типа «А» [17]. Эти профили защиты, основываясь на ГОСТ Р ИСО/МЭК 15408, а также дополняя его, включают иерархическую систему функциональных требований и требований доверия, состав которых постепенно усиливается от шестого к первому классу защиты ОС.

Наибольший интерес, с учетом сложившейся практики создания и сертификации отечественных защищенных ОС, представляют профили защиты для третьего и второго классов [13, 14], ориентированные на информационные системы, в которых обрабатывается информация, содержащая секретные или совершенно секретные сведения соответственно.

Начиная с третьего класса защиты профили защиты включают компонент доверия ADV_SPM.1 «Формальная модель политики безопасности». В соответствии с ГОСТ Р ИСО/МЭК 15408 в этом компоненте указывается, что формальная модель должна быть изложена в формальном стиле (с использованием, например, математического языка), должно быть определено понятие «безопасность» для ОО и должно быть представлено формальное доказательство того, что ОО не может перейти в небезопасное состояние, а также должно быть продемонстрировано соответствие между какой-либо функциональной спецификацией, используемой ОО, и моделью. При этом непосредственно в замечаниях по применению компонента ADV_SPM.1 в профилях защиты испытательной лаборатории при выполнении соответствующей проверки предписывается руководствоваться п. 10.7.1 ГОСТ Р ИСО/МЭК 18045 [18]. Однако

в нем лишь говорится, что «общее руководство отсутствует; за консультациями по выполнению данного подвида деятельности следует обращаться в конкретную систему оценки», т. е. не дается содержательных пояснений, как выполнить данную проверку.

Кроме того, выполнение требований компонента доверия ADV_SPM.1 тесно связано еще, как минимум, с двумя компонентами. В самом компоненте доверия указывается на наличие зависимости от компонента доверия ADV_FSP.4 «Полная функциональная спецификация». Более того, в профилях защиты для ОС третьего и второго классов это требование усилено включением в них компонента доверия ADV_FSP.5 «Полная полуформальная функциональная спецификация с дополнительной информацией об ошибках» и ADV_FSP.6 «Полная полуформальная функциональная спецификация с дополнительной формальной спецификацией» соответственно. Разработка полуформальной функциональной спецификации, тем более формальной спецификации без «привязки» к формальной модели политики безопасности вряд ли представляется возможной.

Также в эти профили защиты включен отсутствующий в ГОСТ Р ИСО/МЭК 15408 компонент доверия AVA_CCA_EXT.1 «Анализ скрытых каналов». Для определения этого компонента в профилях защиты указано, что «анализ скрытых каналов является частью анализа уязвимостей и проводится с целью сделать заключение о существовании и потенциальной пропускной способности каналов передачи сигналов (коммуникационных каналов), которые не предусмотрены для передачи защищаемой информации (данных пользователя и иной информации) или неразрешенных сигналов, но которые могут быть для этого использованы потенциальными нарушителями (в нарушение установленных политик управления информационными потоками, управления доступом или иных установленных ограничений)».

В качестве скрытых каналов в профилях защиты рассматриваются:

- каналы передачи, предназначенные для управления, но которые (в нарушение политики управления доступом или политики управления потоками) потенциально могут использоваться для передачи данных пользователя;
- каналы передачи, предназначенные для передачи данных пользователя, но которые потенциально могут использоваться для передачи сигналов нарушителя (в том числе с использованием модуляции передачи данных);

- каналы передачи, которые (в нарушение установленных ограничений) потенциально могут использоваться для наблюдения одним пользователем за действиями другого пользователя;
- иные типы скрытых каналов.

Это значит, что в качестве скрытых каналов должны быть рассмотрены информационные потоки по времени [19], возникающие в результате действий над объектами доступа (получение доступов, изменение прав доступа, создание, удаление, переименование и т. д.), осуществляемых кооперирующими субъектами доступа ОС, примеры которых рассмотрены в [9, 20]. В результате анализ таких скрытых каналов вне рамок формальной модели политики безопасности, в первую очередь политики управления доступом, будет трудно признать адекватным.

Для выполнения требований компонента доверия ADV_SPM.1 необходимо, чтобы язык представления формальной модели был либо математическим [9], либо формальным (например, [21]). Однако не менее важным является содержательная сторона дела, каковы требования к свидетельствам, предоставляемым при выполнении компонента доверия, какова «глубина» проработки формальной модели. Можно ли считать удовлетворительным, например, следующее «математическое» представление механизма управления доступом ОС в рамках модели в виде кортежа (V, T), где V — множество состояний системы, как-то задающее доступы (текущие доступы или права доступа) субъектов из множества S к объектам из множества O, а T — какая-то функция переходов системы из состояния в состояние, без какой-либо детализации?

Довольно популярной среди разработчиков отечественных защищенных ОС до сих пор является модель Белла–ЛаПадулы. Можно ли признать ее адекватной современным ОС и достаточной для представления в профиле защиты, например механизма мандатного управления доступом, реализуемого в защищенных ОС, принадлежащих семейству *Linux*? При том, что в этой модели не содержится средств описания информационных потоков по времени, иерархии сущностей (адекватной файловым системам ОС), функционально ассоциированных с субъектами сущностей, мандатного контроля целостности, различий в условиях функционирования доверенных и недоверенных субъектов и др. Ответ, очевидно, отрицательный.

Здесь целесообразно отметить, что если принципы реализации мандатного управления доступом хотя бы в том объеме, в котором они были изложены в классической модели Белла–ЛаПадулы, известны многим разработчикам отечественных защищенных ОС, то

в случае с мандатным контролем целостности ситуация часто обстоит иначе. С одной стороны, мандатный контроль целостности был впервые теоретически описан еще в 1975 г. в рамках во многом похожей на модель Белла–ЛаПадулы модели Биба [22]. Он с 2007 г. реализуется в механизме *MIC* (*Mandatory Integrity Control*) всех ОС семейства *Microsoft Windows*, где показал свою высокую эффективность при противодействии компьютерным вирусам и атакам, направленным на несанкционированное повышение привилегий. С другой стороны, среди отечественных ОС мандатный контроль целостности применен только ОССН *Astra Linux Special Edition* версии 1.5. По этой причине напомним читателю основные свойства мандатного контроля целостности.

Так же, как мандатное управление доступом, мандатный контроль целостности — это политика безопасности, при реализации которой задается решетка классификационных иерархических меток. Решетка уровней целостности в самом простом варианте состоит из двух уровней целостности: высокого (привилегированного, доверенного, системного) и низкого (непривилегированного, недоверенного, пользовательского). Каждой сущности и каждому субъекту ОС присваиваются уровни целостности, при этом субъект может получить доступ к сущности только в случае, когда выполняются следующие правила управления доступом:

- при получении доступа на запись к сущности уровень целостности субъекта должен быть не ниже уровня целостности сущности;
- доступ субъекта к сущности не приводит к несанкционированному получению субъектом контроля над другим субъектом, уровень целостности которого не сравним или выше уровня целостности первого субъекта (в данном случае под контролем понимается возможность одним субъектом управлять или существенно изменять функциональность другого субъекта, примером такой ситуации является заражение компьютерным вирусом).

В связи с изложенным для удовлетворения требованиям компонента доверия ADV_SPM.1 на основе опыта авторов по разработке, верификации и внедрению математической модели для обеспечения должной «глубины» и детализации целесообразно предложить следующие требования к ее содержанию. Модель должна включать в себя строго описанные:

- множества учетных записей пользователей, субъектов, объектов (сущностей), устанавливающих классификацию элементов этих множеств, связи между этими множествами или внутри

них функций (отношений), заданных на этих множествах отношений иерархии;

- множества реализуемых прав доступа и доступов субъектов к сущностям, используемых для задания прав доступа и доступов (непосредственно, с использованием групп, ролей, типов, атрибутов) множеств, функций (отношений);
- решетки уровней целостности (для большинства современных ОС без мандатного контроля целостности трудно достичь необходимого уровня защищенности), используемых для задания уровней целостности учетных записей пользователей, субъектов, сущностей функций (отношений);
- решетки уровней конфиденциальности (при необходимости реализации в ОС мандатного управления доступом), используемых для задания уровней доступа учетных записей пользователей и субъектов, уровней конфиденциальности сущностей функций (отношений);
- множества функций (отношений), используемых для задания сущностей, функционально ассоциированных с доверенными субъектами или параметрически ассоциированных с учетными записями пользователей;
- множества функций (отношений), используемых для задания сущностей-контейнеров, доступ к содержащимся в которых сущностям субъектами может быть разрешен без учета уровней целостности или без учета уровней конфиденциальности (при необходимости) таких сущностей-контейнеров;
- виды информационных потоков (как минимум по памяти), используемых для задания информационных потоков между сущностями и субъектами множеств, функций (отношений);
- элементы состояний, моделирующей ОС абстрактной системы, используемых для этого множеств, функций (отношений);
- условия предоставления субъектам прав доступа и доступов к сущностям или субъектам и условий выполнения иных правил перехода системы из состояния в состояние (команд, операций, функций перехода) над учетными записями пользователей, субъектами и сущностями (создание, удаление, переименование, получение параметров), заданных для этого специальных элементов ОС (привилегией, ролей, административных ролей);
- условия возникновения информационных потоков за счет реализации субъектами доступов к сущностям или субъектам или получения субъектами контроля над другими субъектами;

- условия получения субъектами контроля над другими субъектами за счет использования сущностей, функционально ассоциированных с субъектами или параметрически ассоциированных с учетными записями пользователей, и информационных потоков между ними;
- правила перехода системы из состояния в состояние (команды, операции, функции перехода), моделирующей ОС абстрактной системы, включая параметры каждого правила, условия и результаты его применения. Как минимум должны быть описаны: правила администрирования (создания, удаления, переименования, изменения прав доступа, уровней целостности, доступа или конфиденциальности (при необходимости), получения параметров) учетных записей пользователей, субъектов и сущностей; правила предоставления доступов субъектов к сущностям и субъектам; правила создания информационных потоков и получения субъектами контроля над другими субъектами;
- доказательства выполнения условий консистентности модели политики безопасности управления доступом: при применении правил перехода системы из состояния в состояние (команд, операций, функций перехода), моделирующей ОС абстрактной системы, верны условия предоставления субъектам прав доступа и доступов к сущностям или субъектам, условия выполнения иных правил перехода системы из состояния в состояние (команд, операций, функций перехода) над учетными записями пользователей, субъектов и сущностей (создание, удаление, переименование, получение параметров);
- доказательства в рамках моделирующей ОС абстрактной системы того, что реализованный мандатный контроль целостности позволяет обеспечить защиту от несанкционированного изменения субъектом-нарушителем параметров или данных в сущностях, параметров или функциональности субъектов (захватить контроль над субъектом) с более высоким, чем у него, уровнем целостности, и в результате нарушить целостность программно-аппаратной среды ОС;
- доказательства в рамках моделирующей ОС абстрактной системы того, что реализованные мандатные контроль целостности и управление доступом позволяют обеспечить защиту от запрещенных информационных потоков (как минимум по памяти) от сущностей с более высоким уровнем конфиденциальности к сущностям с более низким уровнем конфиденциальности (защиту от информационных потоков «сверху-вниз»).

Примером представления элементов такого описания является рассмотренный ниже базовый уровень мандатной сущностно-ролевой модели управления доступом и информационными потоками в ОС семейства Linux (МРОСЛ ДП-модели). В своей работе авторы используют именно ее в качестве исходной, математической модели политики безопасности управления доступом. Базовый уровень МРОСЛ ДП-модели описывает механизм управления доступом на основе ролей. Основное свойство этого механизма состоит в том, что субъект может получить доступ определенного вида к какой-либо сущности только тогда, когда он обладает ролью, имеющей право на этот вид доступа к этой сущности. Каждая роль представляет собой набор таких пар из сущности и права определенного доступа к ней, а назначение ролей субъектам осуществляет администратор безопасности. Другие уровни МРОСЛ ДП-модели описывают механизмы мандатного контроля целостности и мандатного управления доступом и в настоящей монографии детально не рассматриваются. По этой причине в монографии не излагаются подходы к анализу условий реализации скрытых каналов (информационных потоков) по памяти и по времени, как относящиеся в первую очередь к мандатному управлению доступом и выполненные на последующих после базового уровнях модели [9, 20]. В то же время в силу отмеченной важности осуществления такого анализа в рамках формальной модели безопасности управления доступом для выполнения требований компоненты доверия AVA_CCA_EXT.1 там, где это возможно, при описании базового уровня МРОСЛ ДП-модели даются пояснения, раскрывающие пути дальнейшего осуществления анализа скрытых каналов.

Кроме того, с учетом сложности математической модели политики безопасности управления доступом целесообразно в соответствии с требованиями компонента доверия ADV_SPM.1 для формального доказательства того, что ОС не может перейти в небезопасное состояние, а также для демонстрации соответствия между какой-либо функциональной спецификацией ОС и моделью требовать ее верификации с применением инструментальных средств. Для этого представление модели должно включать описание:

- основных функциональных возможностей, формализованного языка и порядка применения использованных для верификации модели политики безопасности управления доступом инструментальных средств;
- представления модели политики безопасности управления доступом с использованием формализованного языка инструмен-

тальных средств верификации. При этом на формализованном языке должны быть выражены: элементы состояний, моделирующей ОС абстрактной системы, используемые для этого множества, функции (отношения); правила перехода системы из состояния в состояние (команды, операции, функции перехода), включая параметры каждого правила, условия и результаты его применения; условия выполнения мандатного контроля целостности и мандатного управления доступом (при необходимости);

- если формализованный язык инструментальных средств не может точно выразить некоторые элементы модели политики безопасности управления доступом, то описание всех таких элементов и полуформальное обоснование того, что это не влияет на итоговый результат верификации;
- результатов верификации модели политики безопасности управления доступом с использованием инструментальных средств верификации с указанием того, какие элементы модели были верифицированы в автоматическом режиме, а какие в полуавтоматическом (ручном) режиме;
- с применением инструментальных средств результатов верификации выполнения условий консистентности модели политики безопасности управления доступом: при применении правил перехода системы из состояния в состояние (команд, операций, функций перехода), моделирующей ОС абстрактной системы, верны условия предоставления субъектам прав доступа и доступов к сущностям или субъектам, условия выполнения иных правил перехода системы из состояния в состояние (команд, операций, функций перехода) над учетными записями пользователей, субъектами и сущностями (создание, удаление, переименование, получение параметров);
- с применением инструментальных средств результатов верификации в рамках моделирующей ОС абстрактной системы того, что реализованный мандатный контроль целостности позволяет обеспечить защиту от несанкционированного изменения субъектом-нарушителем параметров или данных в сущностях, параметров или функциональности субъектов (захватить контроль над субъектом) с более высоким, чем у него уровнем целостности, и в результате нарушить целостность программно-аппаратной среды ОС;
- при необходимости с применением инструментальных средств результатов верификации в рамках моделирующей ОС абстрактной системы того, что реализованные мандатные контроль

целостности и управление доступом позволяют обеспечить защиту от запрещенных информационных потоков (как минимум по памяти) от сущностей с более высоким уровнем конфиденциальности к сущностям с более низким уровнем конфиденциальности (защиту от утечки конфиденциальных данных, от информационных потоков «сверху-вниз»).

Возможность практического выполнения этих условий подтверждается опытом верификации МРОСЛ ДП-модели, представленной на формальном языке Event-B в инструментальной среде Rodin. Примерам такой верификации для базового уровня модели посвящена глава 3 настоящей монографии, а подходу к представлению на основе модели формальной функциональной спецификации элементов механизма управления доступом — глава 4.

3.2. Базовый уровень МРОСЛ ДП-модели в математической нотации

Как было показано в предыдущей главе, отправной точкой выполнения требований компонента доверия ADV_SPM.1 является либо разработка модели политики безопасности управления доступом в математической нотации с ее дальнейшим переводом формализованную нотацию, либо изначально представление модели в формализованной нотации. Первый путь базируются на отработанной десятилетиями опыте формирования классических математических моделей, начиная с модели Белла–ЛаПадулы, *Take-Grant* и др. [9]. Хотя перевод в формализованную нотацию представляет собой достаточно сложную с научной и практической точек зрения задачу, ее решение при наличии уже разработанной математической модели все же проще, чем второй путь — разработка модели политики безопасности управления доступом в формализованной нотации с нуля.

В связи с этим авторами настоящей монографии был выбран первый путь — разработка в математической нотации мандатной сущностно-ролевой ДП-модели (МРОСЛ ДП-модели) как теоретической основы для реализации механизма управления доступом ОССН. Этому, в свою очередь, предшествовал достаточно длительный этап формирования на основе классических моделей семейства ДП-моделей безопасности управления доступом в компьютерных системах с дискреционным, мандатным или ролевым управлением доступом. Рассмотрим его подробнее.

В каждой из классических математических моделей безопасности управления доступом, как правило, используются оригинальные

определения основных элементов и механизмов современных компьютерных систем и не учитываются многие существенные особенности их функционирования (наличие иерархии сущностей, доверенных и недоверенных субъектов, различия в условиях возникновения информационных потоков по памяти или по времени и т. д.). В связи с этим с целью обеспечения возможности теоретического анализа условий утечки прав доступа и реализации запрещенных информационных потоков по памяти или по времени было построено базовое семейство ДП-моделей [23]. Первоначально в состав этого семейства вошло десять ДП-моделей компьютерных систем с дискреционным или мандатным управлением доступом, наиболее существенными из которых являлись следующие модели.

Основой всех моделей семейства стала дискреционная базовая ДП-модель, построенная с применением расширенной модели *Take-Grant*, модели Белла–ЛаПадулы, модели систем военных сообщений и субъектно-ориентированной модели изолированной программной среды [9]. При этом был использован классический субъект-сущностный подход. В рамках базовой ДП-модели были обоснованы необходимые и достаточные условия передачи в прав доступа или реализации информационных потоков по памяти или по времени для самого простого случая, когда все субъекты идеально кооперируют друг с другом при передаче прав доступа и создании информационных потоков.

Для анализа компьютерных систем, в которых все субъекты являются либо доверенными, либо недоверенными, когда доверенные субъекты не кооперируют с недоверенными при передаче прав доступа или реализации информационных потоков, были построены ДП-модель без кооперации доверенных и недоверенных субъектов (БК ДП-модель), ДП-модель с блокирующими доступами доверенных субъектов (БД ДП-модель) и ДП-модель с функционально ассоциированными с субъектами сущностями (ФАС ДП-модель), которая позволяет анализировать условия получения недоверенным субъектом права доступа владения к доверенному субъекту с использованием реализации недоверенным субъектом информационного потока по памяти к сущности, функционально ассоциированной с доверенным субъектом.

Для анализа безопасности компьютерных систем с мандатным управлением доступом были построены мандатная ДП-модель, мандатная ДП-модель с блокирующими доступами доверенных субъектов (БДМ ДП-модель), мандатная ДП-модель с отождествлением порожденных субъектов (ОСМ ДП-модель) и мандатная ДП-модель

КС, реализующих политику строгого мандатного управления доступом (ПСМ ДП-модель).

Дальнейшее развитие семейства ДП-моделей пошло по следующим двум направлениям:

- построение ДП-моделей типовых практически значимых или перспективных компьютерных систем;
- построение ДП-моделей, содержащих элементы принципиально новые по сравнению с элементами уже разработанных ДП-моделей.

По первому направлению построены ДП-модели для нескольких реальных компьютерных систем, в том числе ДП-модель веб-системы и веб-системы на основе СУБД, в рамках которых осуществлен анализ защищенности типовых веб-систем от угроз реализации атак вида межсайтового скриптинга (*Cross Site Scripting*, *XSS*) [24] и *SQL*-инъекции [25].

По второму направлению с использованием ДП-модели с функционально или параметрически ассоциированными с субъектами сущностями (ФПАС ДП-модели) [26] был рассмотрен случай, когда в компьютерной системе могут существовать параметрически ассоциированные с субъектами сущности (файлы паролей, файлы *cookies*), реализация от которых информационных потоков по памяти (например, их чтение) к недоверенным субъектам позволяет им получить контроль над другими субъектами системы, в том числе доверенными. Кроме того, описана ДП-модель файловых систем (ФС ДП-модель) [27], в которой анализируется характерный для файловых систем новый вид доверенных субъектов — потенциальных доверенных субъектов, из которых могут быть созданы доверенные субъекты, реализующие доступ к сущностям, защищенным механизмами файловых систем (например, механизмами файловой системы *EFS* в среде ОС семейства *Windows XP*/2003/*Vista*).

Также по второму направлению для исследования безопасности компьютерных систем с учетом особенностей ролевого управления доступом на основе семейства ролевых моделей *RBAC* и семейства дискреционных и мандатных ДП-моделей были построены две ролевые ДП-модели (БР ДП-модель и РОСЛ ДП-модель) [28, 29], ставшие «промежуточными», заложившими фундамент формирования современной МРОСЛ ДП-модели.

В МРОСЛ ДП-модели было обеспечено сочетание мандатного управления доступом, мандатного контроля целостности с перспективным ролевым управлением доступом. При этом особое внимание

уделялось детальному описанию правил переходов системы из состояние в состояние, которые классифицированы на де-юре и де-факто правила.

В теории компьютерной безопасности важнейшим результатом исследования математической модели безопасности управления доступом и информационными потоками, как правило, считается обоснование в ее рамках условий безопасности или, наоборот, нарушения безопасности рассматриваемых систем. Для соответствия этому в МРОСЛ ДП-модели были приведены определения безопасного начального состояния системы (состояния, в котором отсутствуют запрещенные информационные потоки по памяти или по времени, фактическое владение субъект-сессиями друг другом, информационные потоки по памяти или доступы к сущностям, параметрически или функционально ассоциированным с субъект-сессиями, не нарушают правил мандатного контроля целостности) и определение трех смыслов нарушения безопасности системы:

- в смысле мандатного контроля целостности, позволяющее недоверенной субъект-сессии с низким уровнем целостности захватить контроль (фактическое владение) над доверенной субъект-сессии с высоким уровнем целостности;
- в смысле Белла–ЛаПадулы, результатом которого является создание запрещенного информационного потока по памяти «сверху-вниз»;
- в смысле контроля информационных потоков по времени — создание запрещенного информационного потока по времени «сверху-вниз» между сущностями.

С использованием этих определений в модели были сформулированы и обоснованы достаточные условия безопасности системы во всех трех смыслах.

Однако такое описание МРОСЛ ДП-модели имело достаточно существенный объем и являлось «монолитным», т. е. элементы модели давались в порядке, удобном для описания модели в целом. Из-за большого объема и монолитности модели, невозможности в таком виде ее поэтапной реализации затрудняется использование модели разработчиками ОССН, а также создание на ее основе новых моделей.

Все выше перечисленное стало причиной переработки МРОСЛ ДП-модели в ее иерархическое представление. Первоначально при его разработке в модель (по сравнению с ее «монолитным» представлением) не добавлялись новые элементы, а основной целью являлось формирование следующих четырех упорядоченных уровней

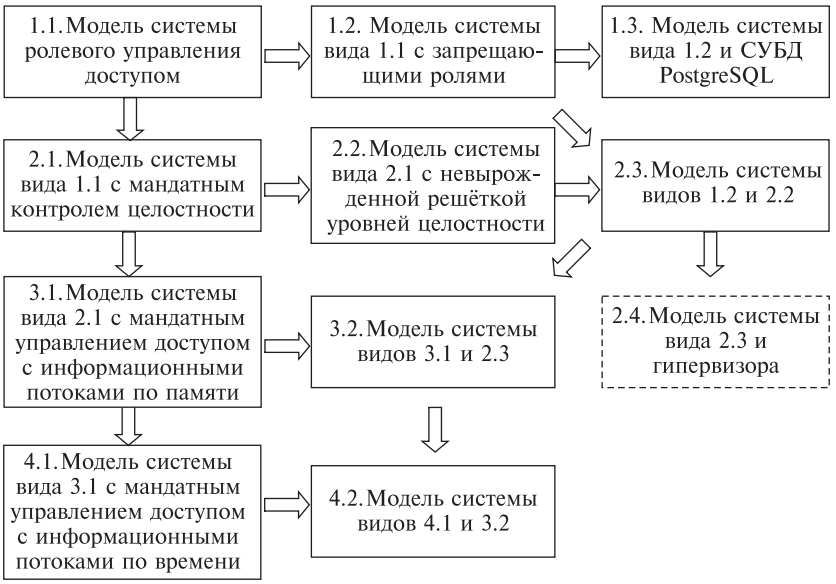

Рис. 3.1. Актуальное иерархическое представление МРОСЛ ДП-модели

[30] (рис. 3.1):

- первый уровень (базовый) — модель системы ролевого управление доступом (1.1);
- второй уровень — модель системы ролевого управление доступом и мандатного контроля целостности (2.1);
- третий уровень — модель системы ролевого управление доступом, мандатного контроля целостности и мандатного управления доступом только с информационными потоками по памяти (3.1);
- четвертый уровень — модель системы ролевого управление доступом, мандатного контроля целостности и мандатного управления доступом с информационными потоками по памяти и по времени (4.1).

Каждый нижний уровень иерархического представления модели соответствует абстрактной системе, элементы которой не зависят от новых элементов, принадлежащих более высокому уровню модели, который, в свою очередь, наследует, а при необходимости корректирует или дополняет элементы нижнего уровня. Такой подход позволяет постепенно усложнять формулировки определений и утверждений модели по мере включения в нее соответствующих очередному рассматриваемому уровню элементов.

Следующим шагом стало включение в иерархическое представ-

ление МРОСЛ ДП-модели новых уровней, содержащих элементы, до этого не использованные в «монолитном» представлении.

Практика использования МРОСЛ ДП-модели при реализации ОССН показала, что в ряде случаев применение ролей, обладающих только правами доступа к сущностям, разрешающим субъектам получение соответствующих доступов к ним, является недостаточным и создает неудобство при администрировании ОССН. В [31] описываются ситуации, когда, например, только одну учетную запись пользователя системы необходимо лишить права доступа, предоставляемого ей через роль, которой обладают все учетные записи пользователей системы, при этом все другие права доступа этой роли должны быть оставлены без изменений. В связи с этим в иерархическое представление модели был добавлен уровень запрещающих ролей (1.2). Запрещающие роли в отличие от «обычных» ролей содержат права доступа к сущностям или субъектам, которые не разрешают, а, наоборот, запрещают получение соответствующих доступов. Этот уровень основан на базовом уровне модели, на нем не реализованы мандатное управление доступом и мандатный контроль целостности, а для задания запрещающих ролей используется описанный еще в ролевых моделях семейства *RBAC* механизм ограничений (*constraint*).

Следует отметить, что мандатный контроль целостности — важнейший, хорошо зарекомендовавший себя механизм безопасности, который аналогично мандатному управлению доступом направлен на задание и применение четких, понятных для пользователей и администраторов современных операционных систем правил обеспечения целостности их программной среды. Изначально в МРОСЛ ДП-модели для задания мандатного контроля целостности применялись всего два уровня целостности: высокий (*i_high*) и низкий (*i_low*). Этого было вполне достаточно для разделения всех элементов ОССН на системные (доверенные компоненты ОССН, обеспечивающие функционирование процессов ее ядра и системных процессов, в том числе механизмов защиты и администрирования), обладающие высоким уровнем целостности, и пользовательские (недоверенные компоненты ОССН, выполняющие функции процессов непривилегированных пользователей, в том числе нарушителей), обладающие низким уровнем целостности.

Именно в таком виде мандатный контроль целостности реализован в версиях 1.4 и 1.5 ОССН. Но при использовании в ОССН технологий виртуализации (гипервизора), например на основе ПК «ВИУ» [32], а также при реализации сетевой доменной архитектуры, тре-

буются дополнительные уровни целостности. Например, одним из возможных решений здесь может быть использование трех уровней целостности: высокого, соответствующего системному для основной ОССН, среднего, соответствующего системному для ОССН, запущенной в среде виртуализации (виртуализированной), и низкому — пользовательскому для основной и виртуализированной ОССН. Кроме того, в перспективе уровней целостности может потребоваться больше, в том числе для реализации невырожденной (состоящей из более чем двух уровней, где не каждый уровень сравним с каждым) решетки уровней целостности. В связи с изложенным был разработан уровень мандатного контроля целостности с невырожденной решеткой уровней целостности (2.2) [33].

Наиболее интересным было формирование уровня ролевого управления доступом с запрещающими ролями и мандатного контроля целостности с невырожденной решеткой уровней целостности (2.3) МРОСЛ ДП-модели, основанном не на одном, а впервые на двух предшествующих уровнях 1.2 и 2.2. В результате было показано, что структура иерархического представления модели может развиваться не только «древовидно». Она позволяет после разработки уровня, содержащего существенные новые элементы, далее дополнить этими элементами существующие «верхние» уровни модели, ориентированные на мандатное управление доступом с контролем информационных потоков по памяти и по времени.

Именно это было осуществлено далее при создании уровней ролевого управления доступом с запрещающими ролями, мандатного контроля целостности с невырожденной решеткой уровней целостности и мандатного управления доступом с информационными потоками по памяти (3.2) и по времени (4.2).

Поскольку полное описание МРОСЛ ДП-модели имеет значительный объем, в настоящей монографии детально приводится только первый базовый уровень этой математической модели. Однако используемая при этом последовательность изложения содержания базового уровня полностью соответствует применяемым для всех последующих уровней иерархического представления модели и достаточна для того, чтобы проиллюстрировать процесс разработки математической модели как часть выполненного авторами общего процесса по моделированию и верификации управления доступом в ОССН при ее сертификации.

Структура описания базового уровня иерархического представления МРОСЛ ДП-модели состоит из четырех частей, следующих далее. Сначала определяются структуры данных, составляющие

модель, в первую очередь необходимые для описания элементов состояний рассматриваемой в рамках модели абстрактной системы. Затем приводятся ограничения на связи между элементами разных видов, регламентируются условия консистентности состояний системы, а также переходов системы из состояние в состояние. Далее описываются правила перехода системы из состояния в состояние, которые соответствуют выполнению тех или иных действий в механизме управления доступом реальной ОССН, инициированные непривилегированными пользователями, администраторами безопасности, а также функционирующими от их имени процессами. В заключении обосновывается корректность правил перехода системы из состояния в состояние, т. е. выполнение при их применении условий консистентности состояний системы и самих переходов из состояния в состояние.

По мере описания модели также даются пояснения возможных способов или особенностей реализации ее элементов в реальной ОССН.

3.3. Состояние системы: структуры данных модели

3.3.1. Элементы состояния системы

Выбранный подход к моделированию политики безопасности управления доступом в литературе принято называть как state-based modeling, т. е. построение модели в форме системы переходов из состояния в состояние. Синонимом «системы переходов» в данном случае может быть термин «автоматная модель».

Основным элементам состояния системы в рамках модели, образующим конструкцию базового уровня иерархического представления МРОСЛ ДП-модели, соответствуют учетные записи пользователей, представляющие пользователей ОССН, функционирующих от их имени субъект-сессий — «активных» компонентов (процессов) ОССН, так называемые «пассивные» сущности, представляющие файлы, каталоги, сокеты и другие ресурсы ОССН, представляющие связи между пользователями и пассивными сущностями. Все эти элементы применяются для описания состояний рассматриваемой в рамках модели абстрактной системы, для чего используем следующие обозначения:

U — конечное непустое множество учетных записей пользователей (в любой ОССН задана хотя бы одна учетная запись пользователя);

S — конечное непустое множество субъект-сессий учетных записей пользователей (всех «активных» компонентов защищенной ОССН, при этом в любой ОССН есть хотя бы одна субъект-сессия, например, соответствующая системному процессу, реализующему процедуру входа пользователя в систему и запуска от имени его учетной записи процессов);

user: $S \to U$ — функция принадлежности субъект-сессии учетной записи пользователя, задающая для каждой субъект-сессии учетную запись пользователя, от имени которой она активизирована;

$E = O \cup C$ — конечное непустое множество сущностей (всех «пассивных» компонентов защищенной ОС, к которым назначаются права доступа, включающее файлы, каталоги, порты, сокеты, очереди, семафоры и другие объекты хранения данных, сетевого и межпроцессного взаимодействия), где O — множество объектов (например, файлов), C — множество контейнеров (например, каталогов) и $O \cap C = \varnothing$.

Также определим:

$NAMES$ — множество допустимых имен сущностей, ролей и административных ролей;

entity_name: $C \times E \to 2^{NAMES}$ — функция имен сущностей в составе сущностей-контейнеров. При этом для любых контейнеров c, $cx \in C$ по определению выполняются условия:

- $|\mathit{entity\_name}(c, cx)| \leqslant 1$;
- если $c \neq cx$ и существует $cy \in C$ такой, что $\mathit{entity\_name}(c, cy) \neq \varnothing$, то $\mathit{entity\_name}(cx, cy) = \varnothing$;
- существует единственная сущность — «корневой контейнер» $ROOT \in C$ такая, что $\mathit{entity\_name}(c, ROOT) = \varnothing$ и, если $c \neq ROOT$, то существует единственная последовательность контейнеров $c_1 = c, c_2, \ldots, c_n = ROOT \in C$ такая, что $n \geqslant 2$ и $\mathit{entity\_name}(c_i, c_{i-1}) \neq \varnothing$, где $1 < i \leqslant n$.

В большинстве ОС сущности образуют иерархические, древовидные структуры, для представления которых в модели вводится определение иерархии сущностей.

Определение 3.1. Иерархией сущностей называется заданное на множестве сущностей E бинарное отношение «$\leqslant$», удовлетворяющее условию: для двух сущностей $e, ex \in E$, выполняется отношение $e \leqslant ex$, когда либо $e = ex$, либо существует последовательность сущностей $e_1 = e$, $e_2, \ldots, e_n \neq ex \in E$ такая, что $n \geqslant 2$ и $\mathit{entity\_name}(e_i, e_{i-1}) \neq \varnothing$, где $1 < i \leqslant n$. В случае, когда для двух

сущностей $e_1, e_2 \in E$ выполняются условия $e_1 \leqslant e_2$ и $e_1 \neq e_2$, будем говорить, что сущность e_1 содержится в сущности-контейнере e_2, и будем использовать обозначение $e_1 < e_2$. Определим функцию иерархии сущностей H_E: $E \to 2^E$, где для $e \in E$ выполняется $H_E(e) = \{ex \in E \mid entity\_name(e, ex) \neq \varnothing\}$.

Аналогичная иерархия необходима и для построения структур из субъект-сессий, так как в ОССН для процесса может быть определен его родительский процесс и, если существуют, дочерние процессы, что в совокупности образует множество древовидных структур (лес).

Определение 3.2. Иерархией субъект-сессий называется заданное на множестве S отношение частичного порядка «$\leqslant$», удовлетворяющее условию: если для субъект-сессии $s \in S$ существуют субъект-сессии $s_1, s_2 \in S$ такие, что $s \leqslant s_2$, $s \leqslant s_1$, то $s_1 \leqslant s_2$ или $s_2 \leqslant s_1$. В случае, когда для двух субъект-сессий $s_1, s_2 \in S$ выполняются условия $s_1 \leqslant s_2$ и $s_1 \neq s_2$, будем говорить, что субъект-сессия s_1 является потомком s_2, и будем использовать обозначение $s_1 < s_2$. Определим H_S: $S \to 2^S$ — функцию иерархии субъект-сессий (сопоставляющую каждой субъект-сессии $s \in S$ множество субъект-сессий $H_S(s) \subset S$, непосредственно в ней содержащихся), удовлетворяющую условиям:

- если субъект-сессия $s_1 \in H_S(s_2)$, то $s_1 < s_2$, и не существует субъект-сессии $s \in S$ такой, что $s_1 < s < s_2$.
- для любых субъект-сессий s_1, $s_2 \in S$, $s_1 \neq s_2$ выполняется равенство $H_S(s_1) \cap H_S(s_2) = \varnothing$.

Заметим, что в рамках МРОСЛ ДП-модели иерархия сущностей задана не «абстрактной» функцией H_E, а реализуемой явно в ОССН функцией имен сущностей в составе сущностей-контейнеров *entity_name*, что, кроме того, позволит в дальнейшем описать правила предоставления содержимого контейнеров (например, списков файлов и каталогов, выдаваемых в реальной ОССН по команде *ls*) с учетом параметров мандатного управления доступом. При этом учтено наличие механизма создания «жестких» ссылок (*hard link*) в файловой системе ОССН, обеспечивающего возможность размещения сущностей-объектов одновременно в нескольких сущностях-контейнерах, в том числе несколько «жестких» ссылок на одну сущность-объект в составе одной сущности-контейнере. Также описаны свойства корневой сущности-контейнера $ROOT$, как правило, соответствующей в реальных защищенных ОС корневому каталогу «/».

Еще одним важным отличием МРОСЛ ДП-модели от других ДП-моделей и классических моделей является то, что множество

субъект-сессий не входит во множество сущностей. Это связно с тем, что в реальной ОССН функции управления доступом к субъект-сессиям (процессам) и сущностям (файлам, каталогам) реализуются раздельно. Таким образом, иерархия на множестве субъект-сессий определяется независимо от иерархии сущностей и при реализации в ОССН может быть задана двумя способами. Первый способ, когда существует явная связь между родительскими субъект-сессиями и их потомками (например, когда при завершении работы родительской субъект-сессии завершают работу ее субъект-сессии потомки, подчиненные родительской в иерархии). Второй способ, когда порожденная субъект-сессией другая субъект-сессия функционирует независимо, в этом случае подчинение ее в иерархии родительской субъект-сессии нецелесообразно.

Ролевое управление доступом является развитием политики безопасности дискреционного управления доступом, при этом права доступа субъектов (сессий) системы к сущностям группируются с учетом специфики их применения, образуя роли. При этом учитываются либо информационная технология, для реализации которой необходим соответствующий набор прав доступа, либо принадлежность роли пользователю, занимающему определенное место в должностной иерархии. Задание ролей позволяет определить более четкие и понятные для пользователей компьютерных систем правила управления доступом, а также предоставлять возможность задать гибкие, изменяющиеся динамически в процессе функционирования компьютерной системы правила управления доступом.

Для введения ролей на базовом уровне иерархического представления МРОСЛ ДП-модели используем следующие обозначения:

R — множество ролей;

AR — множество административных ролей, при этом по определению $AR \cap R = \varnothing$ (административные роли — особый вид ролей, предназначенный для изменения множеств прав доступа ролей, авторизации на роли, а также выполнения функций по администрированию системы, например управления мандатными уровнями конфиденциальности сущностей и субъект-сессий);

$SAR \subset AR$ — множество специальных административных ролей, которые не могут создаваться, удаляться, переименовываться, менять свои параметры в процессе функционирования системы;

$R_r = \{read_r, write_r, execute_r, own_r\}$ — множество видов прав доступа;

$R_a = \{read_a, write_a\}$ — множество видов доступа.

Поскольку в ОССН реализованы три вида прав доступа к сущностям: на чтение, на запись и на выполнение (или на использование контейнера-каталога), а также при управлении доступом к сущностям и субъект-сессиям учитывается наличие у каждой из них уникального владельца, имеющего право передавать права доступа к ним другим учетным записям пользователей, то соответственно в рамках МРОСЛ ДП-модели будем использовать виды прав доступа $read_r$, $write_r$, $execute_r$, own_r. Кроме того, так как в ОССН при получении субъект-сессиями доступов к сущностям они реализуют одну из двух (или обе) основных возможностей: читать или записывать в сущности данные (например, когда процесс открывает доступ к файлу на чтение или на запись), то в модели заданы следующие виды доступов $read_a$ и $write_a$:

$P \subseteq (E \times R_r) \cup (S \times \{own_r\})$ — множество прав доступа к сущностям и субъект-сессиям;

$AP \subseteq (R \cup AR) \times R_r$ — множество административных прав доступа к ролям или административным ролям;

$A \subseteq S \times E \times R_a$ — множество доступов субъект-сессий к сущностям;

$AA \subseteq S \times (R \cup AR) \times R_a$ — множество доступов субъект-сессий к ролям или административным ролям;

PA: $R \cup AR \to 2^P$ — функция прав доступа к сущностям ролей и административных ролей, при этом для каждого права доступа $p \in P$ существует роль $r \in R \cup AR$ такая, что выполняется условие $p \in PA(r)$;

APA: $AR \to 2^{AP}$ — функция административных прав доступа к ролям и административным ролям административных ролей, при этом для каждого административного права доступа $ap \in AP$ существует административная роль $ar \in AR$ такая, что выполняется условие $ap \in APA(ar)$;

shared_container: $C \cup R \cup AR \to \{true, false\}$ — функция разделяемых контейнеров такая, что сущность-контейнер, роль или административная роль $c \in C \cup R \cup AR$ является разделяемой, когда *shared_container*$(c) =$ *true*, в противном случае *shared_container*$(c) =$ *false*;

V: $E \cup R \cup AR \to \{(a_1, \ldots, a_n)$: $a_i \in \{0,1\}, 1 \leqslant i \leqslant n, n \geqslant 1\} \cup \cup \{\varnothing\}$ — функция значений сущностей, ролей и административных ролей (как аналогов сущностей-контейнеров), задающая возможность для каждой из них принимать значение любой (в том числе пустой) конечной последовательности битов;

role_name: $(R \cup AR) \times (R \cup AR) \to NAMES$ — функция имен ролей и административных ролей в составе роли или административной роли (как контейнера). При этом для любых ролей или административных ролей $r, rx \in R \cup AR$ по определению выполняются условия:

- $|role\_name(r, rx)| \leqslant 1$;
- если $role\_name(r, rx) \neq \varnothing$, то либо $r, rx \in R$, либо $r, rx \in AR$;
- для роли или административной роли $ry \in R \cup AR$ справедливо *role_name*$(r, ry) = role\_name(rx, ry)$;
- если существует последовательность ролей или административных ролей $r_1 = r, r_2, \ldots, r_n = rx \in R \cup AR$ такая, что $n \geqslant 2$ и *role_name*$(r_i, r_{i-1}) \neq \varnothing$, где $1 < i \leqslant n$, то не существует последовательности ролей или административных ролей $rx_1 = rx, rx_2, \ldots, rx_m = r \in R \cup AR$ такой, что $m \geqslant 2$ и *role_name*$(rx_i, rx_{i-1}) \neq \varnothing$, где $1 < i \leqslant m$;
- для административной роли $ar \in AR$, если $(r, read_r) \in APA(ar)$ и *role_name*$(r, rx) \neq \varnothing$, то выполняется условие $(rx, read_r) \in APA(ar)$.

Роли также образуют иерархическую структуру, что было предусмотрено еще в классических ролевых моделях семейства $RBAC$. Это упрощает администрирование политики безопасности, делает ее более понятной для пользователей и администраторов компьютерной системы. Например, довольно часто ролевая иерархия задается в соответствии с должностной иерархией пользователей компьютерной системы.

Определение 3.3. Иерархией ролей или административных ролей называется заданное на множестве ролей R или AR соответственно бинарное отношение «$\leqslant$», удовлетворяющее условию: для ролей или административных ролей $r, rx \in R \cup AR$ выполняется отношение $r \leqslant rx$, когда либо $r = rx$, либо существует последовательность ролей или административных ролей $r_1 = r, r_2, \ldots, r_n = rx \in R \cup AR$ такая, что $n \geqslant 2$ и *role_name*$(r_i, r_{i-1}) \neq \varnothing$, где $1 < i \leqslant n$. В случае, когда для двух ролей или административных ролей $r_1, r_2 \in R \cup AR$ выполняются условия $r_1 \leqslant r_2$ и $r_1 \neq r_2$, будем говорить, что роль или административная роль r_1 содержится в роли или административной роли r_2, и будем использовать обозначение $r_1 < r_2$. Определим функцию иерархии ролей или административных ролей H_R: $R \cup AR \to 2^R \cup 2^{AR}$, где для $r \in R \cup AR$ выполняется $H_R(r) = \{rx \in R \cup AR \mid role\_name(r, rx) \neq \varnothing\}$.

В отличие от других ролевых моделей МРОСЛ ДП-модель предлагает рассматривать роли как аналог сущностей-контейнеров

[34], к которым субъект-сессии могут иметь (через административные роли) права доступа и получать доступы. При этом в рамках МРОСЛ ДП-модели иерархии ролей и административных ролей заданы (по аналогии с иерархией сущностей) не функцией H_R, а функцией имен ролей в составе ролей-контейнеров *role_name*. Таким образом, право доступа own_r — владелец роли, $read_r$ — право получать роль как текущую, просматривать ее параметры, $write_r$ — право изменять множество прав доступа роли, $execute_r$ — право обращаться к ролям, подчиненным данной роли в иерархии ролей (по умолчанию предполагается, что такое право доступа к ролям имеется всегда); доступ $read_a$ — получение субъект-сессией роли как текущей, доступ $write_a$ — изменение прав доступа роли или состава ролей, подчиненных ей в иерархии. Имеющиеся в ОС привилегии целесообразно задать административными или «обычными» ролями, это обеспечит целостность механизма управления доступом в ОССН и, кроме того, позволит в дальнейшем присваивать ролям-привилегиям уровни конфиденциальности и уровни целостности. В результате закладывается основа единого механизма мандатного управления доступом и мандатного контроля целостности для доступов к сущностям, получения в качестве текущих и администрирования ролей субъект-сессиями, с возможностью противодействия в дальнейшем запрещенным информационным потокам по памяти или по времени.

Кроме того, для сущностей-контейнеров, ролей и административных ролей задана функция разделяемых контейнеров (помечаемых в ОССН атрибутом «t») и функция их значений. При реализации иерархий ролей в реальной ОССН по аналогии с файловой системой можно использовать виртуальную структуру сущностей-ролей, отличающуюся от структур для файлов наличием «жестких» ссылок не только на роли-«объекты» (роли, которым в иерархии не подчинена ни одна другая роль), но и на роли-«контейнеры» (роли, которым в иерархии подчинена хотя бы одна роль).

В моделях семейства $RBAC$ и других ролевых ДП-моделях предполагалось, что функция авторизованных ролей учетных записей пользователей UA обладает свойством «наследования» подчиненных ролей, т. е. выполняется следующее условие: для учетной записи пользователя $u \in U$, если роли $r, r' \in R$ такие, что $r \in UA(u)$ и $r' \leqslant r$, то выполняется условие $r' \in UA(u)$. Это обеспечивается «наследованием» административной ролью права доступа $read_r$ от данной роли ко всем подчиненным ей ролям в иерархии. При этом права доступа $write_r$ и own_r таким свойством не обладают, что мо-

жет позволить гибко задавать «диапазоны» администрируемых ролей по аналогии с моделью ролевого администрирования $ARBAC$.

Ранее в моделях семейства $RBAC$ и ролевых ДП-моделях для задания текущих ролей использовалась функция *roles* [35, 36], а для администрирования ролей функции вида *can_assign*, *can_revoke* и *can_manage_rights*, которые потребовали бы отдельной реализации в ОССН. В МРОСЛ ДП-модели для этого используются контролируемые системой мандатного управления доступом доступы субъект-сессий к ролям (задаются множество AA) и административные права доступа административных ролей (задаются функцией APA). При этом для обеспечения большего быстродействия ОССН при проверке прав доступа субъект-сессий к сущностям целесообразно для каждой субъект-сессии хранить списки (по аналогии со списком привилегий) текущих ролей, к которым она имеет соответственно доступы $read_a$ или $write_a$.

Поскольку в реальной ОССН любой доступ субъект-сессии к сущности сопровождается последовательной проверкой наличия у субъект-сессии прав доступа на выполнение $execute_r$ ко всем контейнерам, начиная с корневого, в котором содержится сущность, то для удобства и краткости дальнейшего описания таких ситуаций определим функцию:

execute_container: $S \times C \times E \to \{true, false\}$ — функция доступа субъект-сессии к сущностям в контейнерах такая, что по определению для субъект-сессии $s \in S$, контейнера $c \in C$ и сущности $e \in E$ справедливо равенство *execute_container*$(s, \mathrm{c}, e) =$ *true* тогда и только тогда, когда существует последовательность сущностей $e_1, \ldots, e_n \in E$, где либо $n =1$ и $c = e = e_1$, либо $n \geqslant 2$, $c = e_{n-1}$, $e = e_n$ и выполняются следующие условия:

- не существует сущности-контейнера $e_0 \in E$ такой, что $e_1 \in \in H_E(e_0)$;
- верно $e_i \in H_E(e_{i-1})$, где $1 < i \leqslant n$;
- существует $r_i \in R \cup AR$ такая, что $(s, r_i, read_a) \in AA$ и $(e_i, execute_r) \in PA(r_i)$, где $1 \leqslant i < n$.

Также используем следующее обозначение для состояния системы в целом:

$$G = (APA, PA, user, A, AA, H_R, H_E, H_S),$$

где:

- множества учетных записей пользователей U, сущностей E, субъект-сессий S, прав доступа к сущностям P, доступов субъект-сессий к сущностям A, доступов субъект-сессий к ролям и административным ролям AA;

- функции административных прав доступа к ролям административных ролей APA, прав доступа ролей и административных ролей PA, принадлежности субъект-сессий учетным записям пользователей *user*, иерархии ролей H_R, иерархии сущностей H_E, иерархии субъект-сессий H_S.

  Используем обозначение:

  $\Sigma(G^*, OP)$ — система, при этом:
- G^* — множество всех возможных состояний;
- OP — множество правил перехода системы из состояния в состояние, заданных в табл. 3.1 (см. разд. 3.3.3).

Также по традиции с классическими моделями используем следующее обозначение:

$G \vdash_{op} G'$ — переход системы $\Sigma(G^*, OP)$ из состояния G в состояние G' с использованием правила перехода системы из состояния в состояние $op \in OP$, при этом, если условия применения правила op не выполняются в состоянии G, то по определению справедливо равенство $G = G'$.

Если для системы $\Sigma(G^*, OP)$ определено начальное состояние, то будем использовать обозначение $\Sigma(G^*, OP, G_0)$ — система $\Sigma(G^*, OP)$ с начальным состоянием G_0.

Таким образом, определены все основные элементы, используемые для описания состояний рассматриваемой в рамках базового уровня иерархического представления МРОСЛ ДП-модели абстрактной системы.

3.3.2. Условия консистентности модели

Связи между элементами, задающими каждое состояние абстрактной системы в рамках МРОСЛ ДП-модели и определяющими условия функционирования механизма управления доступом реальной ОССН, строятся не произвольным образом. Имеются ограничения, которые по сути являются критериями корректного, консистентного состояния МРОСЛ ДП-модели, а также переходов из состояния в состояние. Кратко будем называть эти ограничения условиями консистентности модели. Таким образом, консистентность состояний и переходов системы на базовом уровне иерархического представления МРОСЛ ДП-модели состоит в выполнении следующих условий.

Условие 1 (права доступа ролей или административных ролей к субъект-сессиям и доступы субъект-сессий друг к другу):

- роли или административные роли могут обладать к субъект-сессиям только правом доступа владения: для роли или адми-

нистративной роли $r \in R \cup AR$ и субъект-сессии $s \in S$, если $(s, \alpha_r) \in PA(r)$, то $\alpha_r = own_r$;

- субъект-сессии не могут иметь друг к другу никаких доступов: для субъект-сессий $s, s' \in S$ выполняется $\{(s, s', \alpha_a)\colon\ \alpha_a \in R_a\} \cap \cap A = \varnothing$.

Условие 2 (административные права доступа и иерархия ролей или административных ролей):

- все роли и административные роли являются «разделяемыми контейнерами»: для каждой роли или административной роли $r \in R \cup AR$ справедливо равенство *shared_container*$(r) =$ *true*;
- у каждой административной роли есть права доступа $execute_r$ ко всем ролям и административным ролям: для каждой административной роли $ar \in AR$, роли или административной роли $r \in R \cup AR$ выполняется условие $(r, execute_r) \in APA(ar)$.

Условие 3 (доступы и права доступа, право доступа владения, администрирование параметров прав доступа сущностей, ролей, административных ролей):

- к ролям, административным ролям и сущностям субъект-сессии могут иметь любые виды доступа из множества R_a;
- роли и административные роли могут иметь к сущностям любые права доступа из множества R_r, только административные роли могут иметь к ролям или административным ролям права доступа из множества R_r;
- для управления доступом к сущности субъект-сессия должна иметь к ней через соответствующую текущую роль или административную роль право доступа владения;
- для каждой сущности или субъект-сессии, если существует, то единственная роль или административная роль, обладающая к ней правом доступа владения, при этом для изменения роли или административной роли, обладающей правом доступа владения к сущности или субъект-сессии, необходимо наличие у субъект-сессии доступа на чтение к административной роли *entities_admin_role* или *subjects_admin_role* соответственно: для каждой сущности $e \in E$ выполняется условие $|\{r \in R \cup AR\colon (e, own_r) \in PA(r)\}| \leqslant 1$, для каждой субъект-сессии $s \in S$ выполняется условие $|\{r \in R \cup AR\colon\ (s, own_r) \in PA(r)\}| \leqslant 1$, где $entities\_admin\_role, subjects\_admin\_role \in SAR$;
- для каждой роли существует единственная административная роль *roles_admin_role*, обладающая к ней правом доступа владения, для каждой административной роли существует единственная административная роль *admin_roles_admin_role*, обла-

дающая к ней правом доступа владения: для каждой роли $r \in$ $\in R$ выполняется условие $\{ar' \in AR\colon (r, own_r) \in APA(ar')\} =$ $= \{roles\_admin\_role\}$, для каждой административной роли $ar \in$ $\in AR$ выполняется условие $\{ar' \in AR\colon (ar, own_r) \in APA(ar')\} =$ $= \{admin\_roles\_admin\_role\}$, где *roles_admin_role*, *admin_roles_admin_role* $\in$ *SAR*;

- для управления доступом к роли или административной роли субъект-сессия должна иметь доступ на чтение к административной роли *roles_admin_role* или *admin_roles_admin_role* соответственно.

Условие 4 (доступ к сущностям в иерархии сущностей). Для получения субъект-сессией любого доступа к сущности, создания «жесткой» ссылки на нее, получения или изменения параметров, прав доступа к ней, активизации из нее субъект-сессии требуется существование последовательности непосредственно вложенных друг в друга сущностей-контейнеров, начинающейся с некоторой сущности-«корневой контейнер» (например, корневой контейнер «/» в ОССН) и заканчивающейся сущностью-контейнером, в состав которой непосредственно входит сама сущность, и наличие у субъект-сессии текущих ролей или административных ролей, обладающих в совокупности правами доступа $execute_r$ ко всем сущностям-контейнерам этой последовательности: для состояний системы G и G', правила перехода системы из состояния в состояние $op \in OP$ таких, что $G \vdash_{op} G'$, если субъект-сессия $s \in S$, сущность $e \in E$, $(s, e, \alpha_a) \notin A$ и $(s, e, \alpha_a) \in A'$, где $\alpha_a \in R_a$, то существует контейнер $c \in C$ и $execute\_container(s, c, e) = true$.

Условие 5 (создание, переименование или удаление сущности, роли или административной роли или «жесткой» ссылки на нее, получение ее параметров):

- для создания, переименования или удаления сущности, роли или административной роли или «жесткой» ссылки на нее в сущности-контейнере, роли или административной роли соответственно субъект-сессии необходимо иметь к последней доступ на запись и текущую роль или административную роль, обладающую к последней правом доступа на выполнение $execute_r$;
- для изменения прав доступа роли или административной роли субъект-сессии необходимо иметь к ней доступ на запись, за исключением случаев, когда либо административным ролям назначаются административные права доступа $read_r$ или $execute_r$ к ролям или административным ролям при изменении их иерархии в соответствии условием 2 и определением 3.3, либо уда-

ляются сущности, субъект-сессии, роли или административные роли и, соответственно, удаляются имеющиеся к ним у ролей или административных ролей права доступа;

- для переименования, удаления сущности или «жесткой» ссылки на сущность e в сущности-контейнере c ($e \in H_E(c)$), помеченной как разделяемая ($shared\_container(c) = true$), требуется наличие у субъект-сессии доступа на чтение к роли или административной роли, обладающей правом доступа владения own_r к сущности e;
- сущности-контейнеры при создании помечаются как неразделяемые. Для изменения метки разделяемости сущности-контейнера субъект-сессии необходимо обладать текущей ролью или административной ролью, имеющей право доступа владения к этой сущности-контейнеру или доступом на чтение к административной роли *entities_admin_role*;
- для получения субъект-сессией данных о ролях или административных ролях, обладающих правами доступа к сущности, требуется либо наличие у субъект-сессии доступа на чтение к административной роли *entities_admin_role*, либо роли или административной роли, обладающей правом доступа владения own_r к этой сущности;
- для получения субъект-сессией данных о ролях или административных ролях, обладающих правом доступа владения к другой субъект-сессии или имеющихся у этой субъект-сессии текущих ролях или административных ролях, требуется либо наличие у первой субъект-сессии доступа на чтение к административной роли *subjects_admin_role*, либо к роли или административной роли, обладающей правом доступа владения own_r ко второй субъект-сессии;
- для создания, переименования, удаления, получения параметров роли или административной роли, «жесткой» ссылки на нее, числа «жестких» ссылок к ней, множества административных ролей, обладающих к ней правами доступа, требуется наличие у субъект-сессии доступа на чтение к административной роли *roles_admin_role* или *admin_roles_admin_role* соответственно.

Условие 6 (администрирование учетных записей пользователей):

- для создания или удаления учетной записи пользователя требуется наличие у субъект-сессии доступа на чтение к административной роли $users\_admin\_role \in SAR$, доступов на чтение, а при

создании и на запись, к административным ролям *roles_admin_role* и *admin_roles_admin_role*, при этом множество субъект-сессий, функционирующих от имени данной учетной записи пользователя, должно быть пустым;

- для получения субъект-сессией параметров учетной записи пользователя она либо должна функционировать от ее имени, либо иметь текущую административную роль *users_admin_role*.

Условие 7 (создание и удаление субъект-сессий):

- субъект-сессия может активизировать из сущности новую субъект-сессию от имени некоторой учетной записи пользователя только при наличии к сущности права доступа на выполнение у хотя бы одной из ролей, к которой активизирующая субъект-сессия имеет доступ на чтение;
- субъект-сессия может удалить субъект-сессию, только обладая текущей ролью или административной ролью, имеющей к ней право доступа владения.

Условие 8 (вид метки):

- для каждой сущности, роли или административной роли задается вид ее метки: прямая или косвенная, который не изменяется в процессе функционирования системы: задается функция *direct*: $E \cup R \cup AR \to \{true, false\}$, где для $e \in E \cup R \cup AR$, если *direct*$(e) =$ *true*, то метка прямая, иначе — косвенная;
- метка каждой роли или административной роли является прямой: для $r \in R \cup AR$ верно равенство *direct*$(r) =$ *true*;
- если у некоторой сущности-контейнера метка косвенная, то у всех сущностей ниже ее в иерархии она также косвенная: если для сущности-контейнера $c \in C$ выполняется *direct*$(c) =$ *false*, то для всех сущностей $e \in E$ таких, что $e \leqslant c$, выполняется *direct*$(e) =$ *false*;
- для каждой сущности с косвенной меткой существует единственная старшая ее в иерархии сущность-контейнер с прямой меткой, для которой все сущности, находящиеся ниже ее в иерархии, имеют косвенные метки, а права доступа всех ролей или административных ролей к ним равны правам доступа к этой сущности-контейнеру: для каждой сущности $e \in E$ такой, что *direct*$(e) =$ *false*, существует единственная сущность-контейнер $\mathrm{c} \in C$ такая, что *direct*$(c) =$ *true*, $e < \mathrm{c}$ и для любой сущности $e' \in E$ такой, что $e' < \mathrm{c}$, верно *direct*$(e') =$ *false* и выполняется $(e', \alpha_r) \in PA(r)$ тогда и только тогда, когда $(c, \alpha_r) \in PA(r)$;
- в сущностях-контейнерах с косвенной меткой нельзя создавать сущности с прямой меткой или «жесткие» ссылки на них.

В сущности-контейнере с прямой меткой могут создаваться сущности или «жесткие» ссылки на них с метками одного вида. «Жесткая» ссылка на сущность-объект с косвенной меткой может создаваться в сущности-контейнере, подчиненной в иерархии той же самой единственной сущности-контейнеру, которой подчинена в иерархии сама сущность-объект.

Условие 9 (индивидуальная административная и индивидуальная роли учетной записи пользователя, общая роль):

- для каждой учетной записи пользователя $u \in U$ задается индивидуальная административная роль $u\_admin \in AR$, не находящаяся в иерархии других ролей, при этом задано множество всех индивидуальных административных ролей $U\_ADMIN = \{u\_admin\colon\ u \in U\}$. Множество других ролей учетной записи пользователя u задается с использованием административных прав доступа этой административной роли, определяемых функцией АРА. На траекториях функционирования системы у этой административной ролей не изменяется имя;
- для каждой учетной записи пользователя задается индивидуальная роль, не находящаяся в иерархии других ролей, административными правами доступа на чтение, запись и выполнение к которой обладает ее индивидуальная административная роль: для каждых учетной записи пользователя $u \in U$ задается роль $u\_c \in R$ такая, что $(u\_c,\ \alpha_r) \in APA(u\_admin)$, где $\alpha_r \in \{read_r, write_r, execute_r\}$, при этом задано множество всех индивидуальных ролей $U\_ROLES = \{u\_c : u \in U\}$;
- задается общая роль $common\_role \in R$, не находящаяся в иерархии других ролей, административными правами доступа на чтение, запись и выполнение к которой обладают все индивидуальные административные роли всех учетных записей пользователей, и задано множество $COMMON\_ROLES = \{common\_role\}$: для каждой учетной записи пользователя $u \in U$ выполняется $(common\_role,\ \alpha_r) \in APA(u\_admin)$, где $\alpha_r \in \{read_r, write_r, execute_r\}$;
- административные роли *roles_admin_role* и *admin_roles_admin_role* не используются для нарушения правил назначения административных прав доступа к индивидуальным административным ролям, индивидуальным ролям учетных записей пользователей и общим ролям.

Условие 10 (доступы субъект-сессии к индивидуальным ролям и общим ролям):

- при создании каждой субъект-сессии она получает доступ на чтение к индивидуальной административной роли и доступ на чтение и запись к индивидуальной роли ее учетной записи пользователя и общей роли: для субъект-сессии $s \in S$ выполняются условия $(s, user(s)\_admin, read_a)$, $(s, user(s)\_c,\ read_a)$, $(s, common\_role,\ read_a)$, $(s, user(s)\_c,\ write_a)$, $(s, common\_role,\ write_a) \in AA$;
- при создании каждой субъект-сессии индивидуальная роль ее учетной записи пользователя получает право доступа владения к этой субъект-сессии: для субъект- сессии $s \in S$ выполняется условие $(s, own_r) \in PA(user(s)\_c)$;
- при удалении субъект-сессии удаляются все ее административные доступы к индивидуальной, индивидуальной административной и общей ролям.

Условия консистентности модели сформулированы с учетом технологий разработки механизма управления доступом реальной ОССН, которые уже либо были в ней реализованы, либо могли быть практически реализованы при доработке этого механизма. Вместе с тем как часть математической модели условия консистентности изложены достаточно абстрактно. Поэтому для большей ясности целесообразно пояснить, каким образом они могут быть непосредственно выполнены в ОССН.

В условии 1 предполагается, что субъект-сессии не могут иметь доступов друг к другу, что соответствует условиям функционирования реальных защищенных ОССН. При реализации условия потребуется уточнить, что предоставляет право доступа владения к субъект-сессии, например дает ли такое право возможность ее отладки или возможность выполнить от ее имени какое-либо действие в системе.

Условие 2 обеспечивает потенциальную возможность получения доступа к роли вне зависимости от ее положения в иерархии ролей. Кроме того, оно не позволяет субъект-сессии, не обладающей соответствующими административными ролями, изменить иерархию ролей. Получая доступ на запись к некоторой роли (дающий возможность изменять множество ее прав доступа), такая субъект-сессия не может удалить роли, подчиненные данной роли в иерархии, так как она помечена как «разделяемый контейнер».

Условие 3 задает общий порядок получения доступов и назначения прав доступа ролей или административных ролей, ролей-«владельцев» к сущностям, субъект-сессиям, ролям или административным ролям, при этом вводятся специальные административ-

ные роли *entities_admin_role*, *subjects_admin_role*, *roles_admin_role* и *admin_roles_admin_role*, используемые для администрирования сущностей, субъект-сессий, ролей или административных ролей соответственно.

Условия 4 и 5 задают порядок получения доступа к сущностям, их создания, переименования или удаления, получения параметров типичный для ОССН, при этом эти условия основаны на ролевом управлении доступом и использовании функции *execute_container*. Соответствующие условия задаются для осуществления аналогичных действий с ролями и административными ролями, при этом требуются специальные административные роли *roles_admin_role* и *admin_roles_admin_role*.

В условии 6 указываются роли, требуемые для администрирования учетных записей пользователей, что в предшествующих ДП-моделях не допускалось. Типичные для ОССН условия создания или удаления субъект-сессий заданы в условии 7.

Для обеспечения возможности задания в ОССН прав доступа к сущностям, находящимся в архивах или на внешних устройствах (файловая система которых часто не позволяет хранить права доступа или другие параметры механизма управления доступом, например уровни конфиденциальности или целостности, в соответствии с требованиями МРОСЛ ДП-модели), в условии 8 используются косвенные метки сущностей, с помощью которых указывается, что права доступа ролей или административных ролей к сущности (в дальнейшем уровни конфиденциальности или целостности) наследуются от сущности-контейнера, являющейся «точкой монтирования» к файловой системе архива или внешнего устройства. Так как файловая система ОССН не позволяет создавать «жесткие» ссылки на сущности между файловыми системами различных устройств, то в соответствии с условием 2 такая «точка монтирования» определяется однозначно для каждой сущности с косвенной меткой, а следовательно, однозначно задаются права доступа к ней.

В условиях 9 и 10 в отличие от моделей семейства $RBAC$ и других ролевых ДП-моделей впервые вместо функций UA и AUA для задания авторизованных ролей и административных ролей учетных записей пользователей используются права доступа их индивидуальных административных ролей, задаваемых функцией APA. Такой подход позволяет реализовать ролевое управление доступом — назначение прав доступа учетным записям пользователей только через роли. Также достигается большая совместимость со штатными механизмами управления доступом реальной ОССН. Для обеспе-

чения возможности функционирования в реальной ОССН субъект-сессии (процессу) необходимо предоставить хотя бы одну индивидуальную роль, обладающую или имеющую возможность получения прав доступа к сущностям (особенно при их создании), «принадлежащим» только учетной записи пользователя, от имени которой функционирует субъект-сессия. Например, такие сущности могут располагаться в «домашнем» каталоге («/*home*/*user*»). Кроме того, для представления прав доступа ролей, которые в реальных дискреционных ОС семейства *Linux* задаются «для всех остальных» учетных записей пользователей, аналогично индивидуальным ролям учетных записей пользователей используется общая роль. На базовом уровне модели она единственная, на последующих уровнях модели их может быть несколько, поэтому для общих ролей задано соответствующее множество *COMMON_ROLES*. Таким образом, с использованием индивидуальных административных, индивидуальных ролей учетных записей пользователей и общей роли легко выразить традиционный для ОС семейства *Linux* подход к реализации дискреционного управления доступом.

3.3.3. Де-юре правила перехода системы из состояния в состояние

МРОСЛ ДП-модель, как и большинство классических моделей политик безопасности управления доступом [9], является автоматной, а значит, после задания состояний моделируемой абстрактной системы должна быть задана функция переходов, которая в модели по традиции определяется через описание правил перехода системы из состояния в состояние.

В рамках базового уровня иерархического представления МРОСЛ ДП-модели используются правила перехода системы из состояния в состояние из множества *OP*, которые по аналогии с моделью *Take-Grant* классифицированы на *де-юре правила* — правила, которые требуют реализации в ОССН, т. е. приводящие к «реальным» изменениям ее параметров: изменению множеств прав доступа ролей, получению доступов субъект-сессий к сущностям или ролям и т. д.; и *де-факто правила* — правила, которые не требуют реализации в ОССН, так как используются в модели для отражения факта получения субъект-сессией де-факто владения субъект-сессиями или факта реализации информационного потока по памяти или по времени. В связи с этим на базовом уровне иерархического представления МРОСЛ ДП-модели задаются только де-юре правила перехода системы из состояния в состояние (табл. 3.1). При

Таблица 3.1

Де-юре правила перехода системы из состояния в состояние
на базовом уровне иерархического представления МРОСЛ ДП-модели

Исходное состояние G	Результирующее состояние G'
	***create_user*(*x*, *u*)**
$x \in S$, $u \notin U$, $(x, users\_admin\_role, read_a) \in AA$; $(x, roles\_admin\_role, \alpha_a) \in AA$; $(x, admin\_roles\_admin\_role, \alpha_a) \in AA$, где $\alpha_a \in \{read_a, write_a\}$	$U' = U \cup \{u\}$, $AR' = AR \cup \{u\_admin\}$; $PA'(u\_admin) = \varnothing$; $direct'(u\_admin) = shared\_container'(u\_admin) = true$; $role\_name'(u\_admin) =$ "u_admin"; $H'_R(u\_admin) = \varnothing$; $R' = R \cup \{u\_c\}$; $PA'(u\_c) = \varnothing$; $direct'(u\_c) = shared\_container'(u\_c) = true$; $role\_name'(u\_c) =$ "u_c"; $H'_R(u\_c) = \varnothing$; $APA'(admin\_roles\_admin\_role) = APA(admin\_roles\_admin\_role) \cup \{(u\_admin, own_r)\}$; $APA'(roles\_admin\_role) = APA(roles\_admin\_role) \cup \{(u\_c, own_r)\}$, для $ar \in AR$ выполняется $APA'(ar) = APA(ar) \cup \{(u\_admin, execute_r)\} \cup \{(u\_c, execute_r)\}$; $APA'(u\_admin) = \{(u\_admin, \alpha_r)\colon\ \alpha_r \in \{read_r, write_r, execute_r\}\} \cup \{(u\_c, \alpha_r), (common\_role, \alpha_r)\colon\ \alpha_r \in \{read_r, write_r, execute_r\}\} \cup \{(r, execute_r)\colon\ r \in R' \cup AR'\}$
	***delete_user*(*x*,*u*)**
$x \in S$, $u \in U$, $user^{-1}(u) = \varnothing$; $(x, users\_admin\_role, read_a) \in AA$; $(x, admin\_roles\_admin\_role, read_a) \in AA$; $(x, roles\_admin\_role, read_a) \in AA$	$U' = U \setminus \{u\}$; $AR' = AR \setminus \{u\_admin\}$; $R' = R \setminus \{u\_c\}$, для $ar \in AR'$ выполняется $APA'(ar) = APA(ar) \setminus (\{(u\_admin, \alpha_r)\colon\ \alpha_r \in R_r\} \cup \{(u\_c, \alpha_r)\colon\ \alpha_r \in R_r\})$
	***get_user_attr*(*x*, *u*, *z*)**
$x \in S$, $u \in U$, $z \in O$, $(x, z, write_a) \in A$	$V'(z) =$ ((если $user(x) = u$ или $(x, users\_admin\_role, read_a) \in AA$, то $\{(r', \alpha_r)\colon\ (r', \alpha_r) \in APA(u\_admin)\}$, иначе "$\varnothing$"), (если $user(x) = u$ или $(x, users\_admin\_role, read_a) \in AA$, то $\{s \in S\colon\ user(s) = u\}$, иначе "$\varnothing$"))
	***access_write*(*x*, *y*)**
$x \in S$, $y \in E \cup R \cup AR$, существует $r \in R \cup AR$: $(x, r, read_a) \in AA$, [если $y \in E$, то $(y, write_r) \in PA(r)$ и существует контейнер $c \in C$ такой, что $execute\_container(x, c, y) = true$], [если $y \in R \cup AR$, то $(y, write_r) \in APA(r)$]	если $y \in E$, то $A' = A \cup \{(x, y, write_a)\}$; $AA' = AA$, если $y \in R \cup AR$, то $AA' = AA \cup \{(x, y, write_a)\}$; $A' = A$

Продолжение табл. 3.1

Исходное состояние G	Результирующее состояние G'
***access_read*(x, y)**	
$x \in S$, $y \in E \cup R \cup AR$, существует $r \in R \cup AR$: $(x, r, read_a) \in AA$, [если $y \in E$, то $(y, read_r) \in PA(r)$ и существует контейнер $c \in C$ такой, что $execute\_container(x, c, y) = true$], [если $y \in R \cup AR$, то $(y, read_r) \in APA(r)$]	если $y \in E$, то $A' = A \cup \{(x, y, read_a)\}$; $AA' = AA$, если $y \in R \cup AR$, то $AA' = AA \cup \{(x, y, read_a)\}$; $A' = A$
***delete_access*(x, y, α_a)**	
$x \in S$, $y \in E \cup R \cup AR$, $(x, y, \alpha_a) \in A \cup AA$	если $y \in E$, то $A' = A \setminus \{(x, y, \alpha_a)\}$, $AA' = AA$, если $y \in R \cup AR$, то $AA' = AA \setminus \{(x, y, \alpha_a)\}$, $A' = A$
***grant_rights*(x, r, $\{(y, \alpha_{rj})$: $1 \leqslant j \leqslant k\}$)**	
$x \in S$, $y \in E$, $r \in R \cup AR$, $\alpha_{rj} \in \{write_r, read_r, execute_r\}$; $(x, r, write_a) \in AA$; $direct(y) = true$; [существует $r' \in R \cup AR$: $(x, r', read_a) \in AA$; $(y, own_r) \in PA(r')$], [существует контейнер $c \in C$ такой, что $execute\_container(x, c, y) = true$], где $1 \leqslant j \leqslant k$	$PA'(r) = PA(r) \cup \{(y, \alpha_{rj})$: $1 \leqslant j \leqslant k\}$, если $H_E(y) = \{y' \in H_E(y)$: $direct(y') = false\}$, то для всех $y' < y$ верно $PA'(r) = PA(r) \cup \{(y', \alpha_{rj})$: $1 \leqslant j \leqslant k\}$
***remove_rights*(x, r, $\{(y, \alpha_{rj})$: $1 \leqslant j \leqslant k\}$)**	
$x \in S$; $y \in E$; $r \in R \cup AR$; $\alpha_{rj} \in \{write_r, read_r, execute_r\}$; $\{(y, \alpha_{rj})$: $1 \leqslant j \leqslant k\} \subset PA(r)$; $(x, r, write_a) \in AA$; $direct(y) = true$, [существует $r' \in R \cup AR$: $(x, r', read_a) \in AA$, $(y, own_r) \in PA(r')$], [существует контейнер $c \in C$ такой, что $execute\_container(x, c, y) = true$], где $1 \leqslant j \leqslant k$	$PA'(r) = PA(r) \setminus \{(y, \alpha_{rj})$: $1 \leqslant j \leqslant k\}$, если $H_E(y) = \{y' \in H_E(y)$: $direct(y') = false\}$, то для всех $y' < y$ верно $PA'(r) = PA(r) \setminus \{(y', \alpha_{rj})$: $1 \leqslant j \leqslant k\}$
***set_entity_owner*(x, r, r', y)**	
$x \in S$; $y \in E$; $r, r' \in R \cup AR$, $\{(x, r', write_a), (x, entities\_admin\_role, read_a)\} \subset AA$, $[(\{(x, r, read_a), (x, r, write_a)\} \subset AA$, $(y, own_r) \in PA(r))$ или (для всех $r'' \in R \cup AR$ выполняется $(y, own_r) \notin PA(r''))]$, $direct(y) = true$, [существует контейнер $c \in C$ такой, что $execute\_container(x, c, y) = true$]	$PA'(r) = PA(r) \setminus \{(y, own_r)\}$, $PA'(r') = PA(r') \cup \{(y, own_r)\}$, если $H_E(y) = \{y' \in H_E(y)$: $direct(y') = false\}$, то для всех $y' < y$ верно $PA'(r) = PA(r) \setminus \{(y', own_r)\}$, $PA'(r') = PA(r') \cup \{(y', own_r)\}$
***set_subject_owner*(x, r, r', y)**	
$x \in S$; $y \in S$; $r, r' \in R \cup AR$, $\{(x, r', write_a), (x, subjects\_admin\_role, read_a)\} \subset AA$, $[(\{(x, r, read_a), (x, r, write_a)\} \subset AA$, $(y, own_r) \in PA(r))$ или (для всех $r'' >\in R \cup AR$ выполняется $(y, own_r) \notin PA(r''))]$	$PA'(r) = PA(r) \setminus \{(y, own_r)\}$, $PA'(r') = PA(r') \cup \{(y, own_r)\}$

Продолжение табл. 3.1

Исходное состояние G	Результирующее состояние G'
***grant_admin_rights*$(x, ar, \{(r, \alpha_{rj})$: $1 \leqslant j \leqslant k\})$**	
$x \in S$, $r \in R \cup AR$, $ar \in AR$, $\alpha_{rj} \in \{write_r, read_r\}$, $(x, ar, write_a) \in AA$, [если $r \in R$, то $(x, roles\_admin\_role, read_a) \in AA$, если $r \in AR$, то $(x, admin\_roles\_admin\_role, read_a) \in AA$], где $1 \leqslant j \leqslant k$	$APA'(ar) = APA(ar) \cup \{(r, \alpha_{rj})$: $\alpha_{rj} = write_r$, $1 \leqslant j \leqslant k\} \cup \{(r', \alpha_{rj})$: $\alpha_{rj} = read_r$, $r' \leqslant r$, $1 \leqslant j \leqslant k\}$
***remove_admin_rights*$(x, ar, \{(r, \alpha_{rj})$: $1 \leqslant j \leqslant k\})$**	
$x \in S$, $r \in R \cup AR$, $ar \in AR$, $\alpha_{rj} \in \{write_r, read_r\}, \{(r, \alpha_{rj})$: $1 \leqslant j \leqslant k\} \subset APA(ar), (x, ar, write_a) \in AA$, [если $r \in R$, то $(x, roles\_admin\_role, read_a) \in AA$, если $r \in AR$, то $(x, admin\_roles\_admin\_role, read_a) \in AA$], [не существует $u \in U$ таких, что $ar = u\_admin$ и $r \in \{u\_admin, u\_c, common\_role\}$]	$APA'(ar) = APA(ar) \setminus (\{(r, \alpha_{rj})$: $\alpha_{rj} = write_r$, $1 \leqslant j \leqslant k\} \cup \{(r', \alpha_{rj})$: $\alpha_{rj} = read_r$, $r \leqslant r'$, $1 \leqslant j \leqslant k\})$
***create_object*$(x, y, yd, name, z)$**	
$x \in S$, $y \notin E$, $z \in C$, [$(x, z, write_a) \in A$, существует $r \in R \cup AR$ такая, что $(x, r, read_a) \in AA$ и $(z, execute_r) \in PA(r)$], $name \in NAME \setminus (\{""\} \cup \{entity\_name(z, y')$: $y' \in H_E(z)\})$, $(x, user(x)\_c, write_a) \in AA$, $H_E(z) = \{y' \in H_E(z)$: $direct(y') = yd\}$, [если $yd = true$, то $direct(z) = true$]	$E' = E \cup \{y\}$ $(O' = O \cup \{y\}, C' = C)$, $entity\_name'(z, y) = \{name\}$, $direct'(y) = yd$, если $yd = true$, то $PA'(user(x)\_c) = PA(user(x)\_c) \cup \cup \{(y, own_r)\}$, если $yd = false$ и $c \in C$ такая, что $direct(c) = true$, $H_E(c) = \{y' \in H_E(y)$: $direct(y') = false\}$ и $y < c$, то для всех $r' \in R \cup AR$ выполняется $PA'(r') = PA(r') \cup \{(y, \alpha_{rj})$: $(c, \alpha_{rj}) \in PA(r')\}$, $V'(y) = \varnothing$, $H'_E(z) = H_E(z) \cup \{y\}$, $H'_E(y) = \varnothing$
***create_container*$(x, y, yd, name, z)$**	
$x \in S$, $y \notin E$, $z \in C$, [$(x, z, write_a) \in A$, существует $r \in R \cup AR$ такая, что $(x, r, read_a) \in AA$ и $(z, execute_r) \in PA(r)$], $name \in NAME \setminus (\{""\} \cup \{entity\_name(z, y')$: $y' \in H_E(z)\})$, $(x, user(x)\_c, write_a) \in AA$, $H_E(z) = \{y' \in H_E(z)$: $direct(y') = yd\}$, [если $yd = true$, то $direct(z) = true$]	$E' = E \cup \{y\}$ $(C' = C \cup \{y\}, O' = O)$, $entity\_name'(z, y) = \{name\}$, $shared\_container'(y) = false$, $direct'(y) = yd$, если $yd = true$, то $PA'(user(x)\_c) = PA(user(x)\_c) \cup \{(y, own_r)\}$, если $yd = false$ и $c \in C$ такая, что $direct(c) = true$, $H_E(c) = \{y' \in H_E(y)$: $direct(y') = false\}$ и $y < c$, то для всех $r' \in R \cup AR$ выполняется $PA'(r') = PA(r') \cup \{(y, \alpha_{rj})$: $(c, \alpha_{rj}) \in PA(r')\}$, $V'(y) = \varnothing$, $H'_E(z) = H_E(z) \cup \{y\}$, $H'_E(y) = \varnothing$

Продолжение табл. 3.1

Исходное состояние G	Результирующее состояние G'
$delete\_entity(x, y, z)$	
$x \in S$, $y \in E$, $z \in C$, $y \in H_E(z)$, $H_E(y) = \varnothing$, [не существует $z' \in C$ такой, что $z' \neq z$ и $y \in H_E(z')$, и $\lvert entity\_name(z, y)\rvert = 1$], [$(x, z, write_a) \in A$, существует $r \in R \cup AR$ такая, что $(x, r, read_a) \in AA$ и $(z, execute_r) \in PA(r)$], [если $shared\_container(z) = true$, то существует $r \in R \cup AR$ такая, что $(x, r, read_a) \in AA$ и $(y, own_r) \in PA(r)$]	$E' = E \setminus \{y\}$, $H'_E(z) = H_E(z) \setminus \{y\}$, для $r \in R$ выполняются равенства $PA'(r) = PA(r) \setminus \{(y, \alpha_r)\colon\ \alpha_r \in R_r\}$, $A' = A \setminus \{(s, y, \alpha_a)\colon\ s \in S, \alpha_a \in R_a\}$
$create\_hard\_link(x, y, name, z)$	
$x \in S$, $y \in O$, $z \in C$, [существует $c \in C$: $execute\_container(x, c, y) = true$], [$(x, z, write_a) \in A$, существует $r \in R \cup AR$ такая, что $(x, r, read_a) \in AA$ и $(z, execute_r) \in PA(r)$], $name \in NAME \setminus (\{""\} \cup \cup \{entity\_name(z, y')\colon\ y' \in H_E(z)\})$, $H_E(z) = \{y' \in H_E(z)\colon\ direct(y') = direct(y)\}$, [если $direct(y) = true$, то $direct(z) = true$], [если $direct(y) = false$, то существует $z' \in C$ такая, что $direct(z) = true$, $y < z'$, $z \leqslant z'$ и для всех $y' < z'$ верно $direct(y') = false$]	$entity\_name'(z, y) = entity\_name(z, y) \cup \{name\}$, $H'_E(z) = H_E(z) \cup \{y\}$
$delete\_hard\_link(x, y, name, z)$	
$x \in S$, $y \in O$, $z \in C$, $y \in H_E(z)$, $name \in entity\_name(z, y)$, [существует $z' \in C$ такой, что $z' \neq z$ и $y \in H_E(z')$], [$(x, z, write_a) \in A$, существует $r \in R \cup AR$ такая, что $(x, r, read_a) \in AA$ и $(z, execute_r) \in PA(r)$], [если $shared\_container(z) = true$, то существует $r \in R \cup AR$ такая, что $(x, r, read_a) \in AA$ и $(y, own_r) \in PA(r)$]	$entity\_name'(z, y) = entity\_name(z, y) \setminus \{name\}$, если $entity\_name'(z, y) = \varnothing$, то $H'_E(z) = H_E(z) \setminus \{y\}$
$create\_role(x, r, name, rz)$	
$x \in S$, $r \notin R \cup AR$, $rz \in (R \cup AR) \setminus (U\_ADMIN \cup U\_ROLES \cup \cup COMMON\_ROLES \cup SAR)$, [если $rz \in R$, то $(x, roles\_admin\_role, \alpha_a) \in AA$, если $rz \in AR$, то $(x, admin\_roles\_admin\_role, \alpha_a) \in AA$, где $\alpha_a \in \{read_a, write_a\}$], $(x, rz, write_a) \in AA$, $name \in NAME \setminus \setminus (\{""\} \cup \{role\_name(r', r'')\colon\ r', r'' \in R \cup AR\})$	если $rz \in R$, то $R' = R \cup \{r\}$, $APA'(roles\_admin\_role) = = APA(roles\_admin\_role) \cup \{(r, own_r), (r, execute_r)\} \cup \{(r, read_r)\colon\ (rz, read_r) \in APA(roles\_admin\_role)\}$, если $rz \in \in AR$, то $AR' = AR \cup \{r\}$, $APA'(admin\_roles\_admin\_role) = = APA(admin\_roles\_admin\_role) \cup \{(r, own_r), (r, execute_r)\} \cup \cup \{(r, read_r)\colon\ (rz, read_r) \in APA(admin\_roles\_admin\_role)\}$,

Продолжение табл. 3.1

Исходное состояние G	Результирующее состояние G'
	$APA'(r) = \{(r', execute_r)\colon\ r' \in R \cup AR'\}$, для $ar \in AR \setminus \setminus \{admin\_roles\_admin\_role, roles\_admin\_role\}$ выполняется $APA'(ar) = APA(ar) \cup \{(r, execute_r)\} \cup \{(r, read_r)\colon (rz, read_r) \in APA(ar)\}$, $role\_name'(rz, r) = name$, $direct(r) = true$, $shared\_container'(r) = true$, $PA'(r) = \varnothing$, $H'_R(rz) = H_E(z) \cup \{r\}$, $H'_R(r) = \varnothing$
$delete\_role(x, r, rz)$	
$x \in S$, $r \in (R \cup AR) \setminus (U\_ADMIN \cup U\_ROLES \cup COMMON\_ROLES \cup \cup SAR)$, [если $r \in R$, то $(x, roles\_admin\_role, \alpha_a) \in AA$, если $r \in AR$, то $(x, admin\_roles\_admin\_role, \alpha_a) \in AA$, где $\alpha_a \in \{read_a, write_a\}$], $r \in H_R(rz)$, $H_R(r) = \varnothing$, [не существует $rz' \in R \cup AR$ такой, что $rz' \neq rz$ и $r \in H_R(rz')$], $(x, rz, write_a) \in AA$	$R' = R \setminus \{r\}$, $AR' = AR \setminus \{r\}$, $H'_R(rz) = H_R(rz) \setminus \{r\}$, для $ar \in AR'$ выполняются равенства $APA'(ar) = APA(ar) \setminus \setminus \{(r, \alpha_r)\colon\ \alpha_r \in R_r\}$, $AA' = AA \setminus \{(s, r, \alpha_a)\colon\ s \in S,\ \alpha_a \in R_a\}$
$create\_hard\_link\_role(x, r, rz)$	
$x \in S$, $r \in (R \cup AR) \setminus (U\_ADMIN \cup U\_ROLES \cup COMMON\_ROLES \cup \cup SAR)$, $rz \in (R \cup AR) \setminus (U\_ADMIN \cup U\_ROLES \cup COMMON\_ROLES)$, не выполняется условие $rz \leqslant r$, [если $rz \in R$, то $(x, roles\_admin\_role, \alpha_a) \in AA$, если $rz \in AR$, то $(x, admin\_roles\_admin\_role, \alpha_a) \in AA$, где $\alpha_a \in \{read_a, write_a\}$], $(x, rz, write_a) \in AA$	$role\_name'(rz, r) = role\_name(rz', r)$, где $rz' \in R \cup AR$ и $r \in H_R(rz')$, $H'_R(rz) = H_R(rz) \cup \{r\}$, для $ar \in AR$ таких, что $(rz, read_r) \in APA(ar)$ выполняется $APA'(ar) = = APA(ar) \cup \{(r', read_r)\colon\ r' \leqslant r\}$
$delete\_hard\_link\_role(x, r, rz)$	
$x \in S$, $r \in (R \cup AR) \setminus (U\_ADMIN \cup U\_ROLES \cup COMMON\_ROLES \cup \cup SAR)$, [если $r \in R$, то $(x, roles\_admin\_role, \alpha_a) \in AA$, если $r \in AR$, то $(x, admin\_roles\_admin\_role, \alpha_a) \in AA$, где $\alpha_a \in \{read_a, write_a\}$], $r \in H_R(rz)$, [существует $rz' \in R \cup AR$ такая, что $rz' \neq rz$ и $r \in H_R(rz')$], $(x, rz, write_a) \in AA$	$H'_R(rz) = H_R(rz) \setminus \{r\}$
$rename\_entity(x, y, old\_name, name, z)$	
$x \in S$, $y \in E$, $z \in C$, $y \in H_E(z)$, $old\_name \in entity\_name(z, y)$, $name \in NAME \setminus (\{\text{""}\} \cup \{entity\_name(z, y')\colon\ y' \in H_E(z)\})$, [$(x, z, write_a) \in A$, существует $r \in R \cup AR$ такая, что $(x, r, read_a) \in AA$ и	$entity\_name(z, y) = (entity\_name(z, y) \cup \{name\}) \setminus \{old\_name\}$

Продолжение табл. 3.1

Исходное состояние G	Результирующее состояние G'
$(z, execute_r) \in PA(r)$], [если $shared\_container(z) = true$, то существует $r \in R \cup AR$ такая, что $(x, r, read_a) \in AA$ и $(y, own_r) \in PA(r)$]	
***rename_role*(*x*, *ry*, *name*)**	
$x \in S$, $ry \in (R \cup AR) \setminus (U\_ADMIN \cup U\_ROLES \cup COMMON\_ROLES \cup \cup SAR)$, $name \in NAME \setminus (\{\text{""}\} \cup \{role\_name(rz, ry')\colon\ ry', rz \in R \cup AR, ry' \in H_R(rz)\})$, [если $ry \in R$, то $(x, roles\_admin\_role, read_a) \in AA$, если $ry \in AR$, то $(x, admin\_roles\_admin\_role, read_a) \in AA$], [для $rz \in R \cup AR\colon\ ry \in H_R(rz)$, выполняется $(x, rz, write_a) \in AA$]	для $rz \in R \cup AR$ таких, что $ry \in H_R(rz)$, выполняется $role\_name'(rz, ry) = name$
***read_container*(*x*, *y*, *z*)**	
$x \in S$, $y \in C \cup R \cup AR$, $z \in O$, $(x, z, write_a) \in A$, существует $r \in R \cup AR\colon\ (x, r, read_a) \in AA$, [если $y \in C$, то $(y, read_r) \in PA(r)$, существует $r' \in R \cup AR\colon\ (x, r', read_a) \in AA$, $(y, execute_r) \in PA(r')$, и существует контейнер $c \in C\colon\ execute\_container(x, c, y) = true$], [если $y \in R \cup AR$, то $(y, read_r) \in APA(r)$]	если $y \in C$, то $V'(z) = \{entity\_name(y, e)\colon\ e \in H_E(y)\}$, если $y \in R \cup AR$, то $V'(z) = \{role\_name(y, r)\colon\ r \in H_R(y)\}$
***get_entity_attr*(*x*, *y*, *z*)**	
$x \in S$, $y \in E$, $z \in O$, [существует контейнер $c \in C$ такой, что $execute\_container(x, c, y) = true$], $(x, z, write_a) \in A$	$V'(z) =$ ($direct(y)$, (если $y \in C$, то $shared\_container(y)$, иначе "*false*"), (если $y \in O$, то $\lvert\{e \in C\colon\ y \in H_E(e)\}\rvert$, иначе "0"), (если [существует $r \in R \cup AR\colon\ (x, r, read_a) \in AA$, $(y, own_r) \in PA(r)$] или [$(x, entities\_admin\_role, read_a) \in \in AA$], то $\{(r', \alpha_r)\colon\ r' \in R \cup AR,\ (y, \alpha_r) \in PA(r')\}$, иначе "$\varnothing$"))
***get_subject_attr*(*x*, *y*, *z*)**	
$x \in S$, $y \in S$, $z \in O$, $(x, z, write_a) \in A$	$V'(z) =$ ($user(s)$, (если [существует $r \in R \cup AR\colon\ (x, r, read_a) \in AA$, $(y, own_r) \in PA(r)$] или [$(x, subjects\_admin\_role, read_a) \in AA$], то $\{(r', own_r)\colon\ r' \in R \cup AR,\ (y, own_r) \in \in PA(r')\}$, иначе "$\varnothing$"), (если [существует $r \in R \cup AR\colon\ (x, r, read_a) \in AA$, $(y, own_r) \in PA(r)$] или [$(x, subjects\_admin\_role, read_a) \in AA$], то $\{(r', \alpha_a)\colon\ (y, r', \alpha_a) \in AA\}$, иначе "$\varnothing$"))

Окончание табл. 3.1

Исходное состояние G	Результирующее состояние G'
	***get_role_attr*(x, y, z)**
$x \in S$, $y \in R \cup AR$, $z \in O$, $(x, z, write_a) \in A$	$V'(z) = (direct(y), shared\_container(y)$, (если [$y \in R$ и $(x, roles\_admin\_role, read_a) \in AA$] или [$y \in AR$ и $(x, admin\_roles\_admin\_role, read_a) \in AA$], то $\lvert\{r \in R \cup AR: y \in H_R(r)\}\rvert$, иначе "0"), (если [$y \in R$ и $(x, roles\_admin\_role, read_a) \in AA$] или [$y \in AR$ и $(x, admin\_roles\_admin\_role, read_a) \in AA$], то $\{(r, \alpha_r)$: $r \in AR$, $(y, \alpha_r) \in APA(r)\}$, иначе "$\varnothing$"))
	***set_container_attr*(x, y, t)**
$x \in S$, $y \in C$, $t \in \{true, false\}$, [либо существует $r \in R \cup AR$: $(x, r, read_a) \in AA$, $(y, own_r) \in PA(r)$, либо $(x, entities\_admin\_role, read_a) \in AA$], [существует контейнер $c \in C$ такой, что $execute\_container(x, c, y) = true$]	$shared\_container'(y) = t$
	***create_first_subject*(x, u, y, z)**
$x \in S$, $u \in U$, $y \in E$, $z \notin S$, существует $r \in R \cup AR$ такая, что $(x, r, read_a) \in AA$ и $(y, execute_r) \in PA(r)$, существует контейнер $c \in C$ такой, что $execute\_container(x, c, y) = true$	$S' = S \cup \{z\}$, $user'(z) = u$, $AA' = AA \cup \{(z, u\_admin, read_a), (z, u\_c, write_a), (z, common\_role, write_a, (z, u\_c, read_a), (z, common\_role, read_a)\}$, $H'_S(z) = \varnothing$, $PA'(u\_c) = PA(u\_c) \cup \cup \{(z, own_r)\}$
	***create_subject*(x, y, z)**
$x \in S$, $y \in E$, $z \notin S$, существует $r \in R \cup AR$ такая, что $(x, r, read_a) \in AA$ и $(y, execute_r) \in PA(r)$, существует контейнер $c \in C$ такой, что $execute\_container(x, c, y) = true$	$S' = S \cup \{z\}$, $user'(z) = user(x)$, $AA' = AA \cup \{(z, user(x)\_admin, read_a), (z, user(x)\_c, write_a), (z, common\_role, write_a), (z, user(x)\_c, read_a), (z, common\_role, read_a)\}$, $PA'(user(x)\_c) = PA(user(x)\_c) \cup \cup \{(z, own_r)\}$; $H'_S(x) = H_S(x) \cup \{z\}$, $H'_S(z) = \varnothing$
	***delete_subject*(x, z)**
$x, z \in S$, $H_S(z) = \varnothing$, существует $r \in R \cup AR$: $(x, r, read_a) \in \in AA$, $(z, own_r) \in PA(r)$	$S' = S \setminus \{z\}$, $AA' = AA \setminus \{(z, r, \alpha_a)$: $r \in R \cup AR$, $\alpha_a \in R_a\}$, $A' = A \setminus \{(z, e, \alpha_a)$: $e \in E$, $\alpha_a \in R_a\}$, для $r \in R \cup AR$ выполняется $PA'(r) = PA(r) \setminus \{(z, own_r)\}$, для $z' \in S$ такой, что $z \in H_S(z')$, справедливо равенство $H'_S(z') = H_S(z') \setminus \{z\}$

этом, если в упомянутой таблице для некоторого элемента исходного состояния используется обозначение вида X, то аналогичный элемент результирующего состояния будет иметь обозначение вида X'. Кроме того, в результирующем состоянии не указываются не изменяющиеся элементы состояний системы.

В рамках базового уровня МРОСЛ ДП-модели задано 31 де-юре правило перехода системы из состояния в состояние, условия и результаты применения которых соответствуют условиям консистентности модели. Эти правила предназначены для формального описания (спецификации) следующих основных функций механизма управления доступом защищенной ОС:

- создание, удаление, переименование, получение или изменение параметров учетных записей пользователей, ролей, административных ролей, сущностей или «жестких» ссылок на них, субъект-сессий;
- получение доступов субъект-сессий к сущностям, ролям или административным ролям;
- изменение прав доступа ролей или административных ролей к сущностям, субъект-сессиям, ролям или административным ролям;
- изменение иерархии сущностей, ролей или административных ролей.

Приведенные правила, в основном, соответствуют применяемым в ОС семейства *Linux* подходам к реализации механизма дискреционного управления доступом, некоторые параметры которого выражены с помощью ролей.

Де-юре правила вида *create_user*(x, u), *delete_user*(x, u) и *get_user_attr*(x, u, z) позволяют субъект-сессии x создать, удалить или получить параметры учетной записи пользователя u. Во всех случаях, кроме получения параметров, требуется наличие у субъект-сессии x доступа на чтение к специальной административной роли *users_admin_role*. При этом в последующем состоянии иерархия ролей модифицируется (создаются или удаляются индивидуальная роль и индивидуальная административная роль, назначаются или удаляются административные права доступа к ним). Для создания или удаления требуется наличие у субъект-сессии x доступов на чтение, а в случае создания и на запись, к специальным административным ролям *roles_admin_role* и *admin_roles_admin_role*, так как именно эти роли получают права доступа владения к создаваемым при применении правил индивидуальным административным ролям

и индивидуальным ролям учетной записи пользователя u. В случае удаления учетной записи пользователя требуется, чтобы в этот момент времени от ее имени в системе не функционировала ни одна субъект-сессия. При получении параметров при условии, что x функционирует от имени u или обладает текущим административным доступом на чтение к роли *users_admin_role*, в сущность z (к которой субъект-сессия x должна иметь доступ на запись) записываются роли и права доступа к ним, которыми обладает индивидуальная административная роль u, и записываются субъект-сессии (в реальной защищенной ОС идентификаторы субъект-сессий), функционирующие от имени u (в этом случае полные данные об учетной записи пользователя предоставляются либо субъект-сессии, функционирующей от ее имени, либо субъект-сессии, которая может администрировать эту учетную запись).

Де-юре правила вида *access_read*(x, y) и *access_write*(x, y) позволяют субъект-сессии x, обладающей доступом на чтение к некоторой роли или административной роли r (текущей роли), содержащей соответствующее право доступа к сущности или административное право доступа к роли или административной роли y, получить к y соответствующий доступ. При этом требуется, чтобы доступ к y был предоставлен с учетом прав доступа субъект-сессии x к сущностям-контейнерам или ролям-контейнерам, содержащим y. Де-юре правило *delete_access*(x, y, α_a) позволяет субъект-сессии x, обладающей доступом α_a к сущности или административным доступом к роли или административной роли y, удалить этот доступ.

Де-юре правила вида *grant_rights*$(x, r, \{(y, \alpha_{rj})\colon\ 1 \leqslant j \leqslant k\})$ и *remove_rights*$(x, r, \{(y, \alpha_{rj})\colon\ 1 \leqslant j \leqslant k\})$ позволяют субъект-сессии x добавить или удалить соответственно права доступа к сущности y из множества прав доступа (за исключением права доступа владения) роли или административной роли r. Возможность с использованием правил одновременного изменения нескольких прав доступа к одной сущности соответствует возможностям типовых для реальных защищенных ОС функций администрирования прав доступа (например, функции *chmod*). Непосредственно изменять права доступа разрешено только к сущностям с прямой меткой. Изменение прав доступа к сущностям с косвенной меткой осуществляется одновременно с изменением прав доступа к единственной существующей по условию 8 соответствующей сущности-контейнеру. Для применения правил необходимо наличие у субъект-сессии x доступа на запись к роли r и наличие текущей роли, обладающей правом доступа владения к сущности y. При этом требуется, чтобы x могла получить до-

ступ к сущности y с учетом прав доступа к сущностям-контейнерам, содержащим y.

Де-юре правила вида *set_entity_owner*(x, r, r', y) и *set_subject_owner*(x, r, r', y) позволяют субъект-сессии x либо изменить, либо задать единственную роль-«владелец» (имеющую право доступа владения) к сущности или субъект-сессии y соответственно с роли или административной роли r на роль или административную роль r'. Для этого субъект-сессии x необходимо иметь административные доступы на чтение и запись к r и на запись к r' (чтобы иметь возможность менять права доступа данных ролей), а также иметь административный доступ на чтение соответственно либо к административной роли *entities_admin_role*, либо к *subjects_admin_role*. Непосредственно изменять роль-«владельца» разрешено только к сущностям с прямой меткой. Изменение роли-«владельца» к сущностям с косвенной меткой осуществляется одновременно с изменением роли-«владельца» к единственной существующей по условию 8 соответствующей сущности-контейнеру. Когда изменяется роль-«владелец» сущности y требуется, чтобы x могла получить доступ к сущности y с учетом прав доступа к сущностям-контейнерам, содержащим y.

Де-юре правила вида *grant_admin_rights*$(x, ar, \{(r, \alpha_{rj})\colon 1 \leqslant j \leqslant k\})$ и *remove_admin_rights*$(x, ar, \{(r, \alpha_{rj})\colon 1 \leqslant j \leqslant k\})$ позволяют субъект-сессии x добавить или удалить соответственно права доступа на чтение или запись к роли или административной роли r из множества прав доступа административной роли *ar*, к которой x должна иметь административный доступ на запись. Для изменения прав доступа к роли r требуется наличие у x текущей административной роли *roles_admin_role*, а если r является административной ролью, то к роли *admin_roles_admin_role*. При удалении административных прав доступа не должны нарушаться требования к правам доступа индивидуальных административных ролей, индивидуальных ролей учетных записей пользователей и общей роли.

Де-юре правила вида *create_object*$(x, y, yd, name, z)$, *create_container*$(x, y, yd, name, z)$ и *delete_entity*(x, y, z) позволяют субъект-сессии x создать или удалить сущность-объект или сущность-контейнер y, входящую в состав сущности-контейнера z, к которой субъект-сессия x должна иметь доступ на запись и обладать текущей ролью или административной ролью, имеющей к z право доступа на выполнение *execute*$_r$. В соответствии со спецификой файловой системы ОССН нельзя создавать одноименные сущности в одной сущности-контейнере или удалять непустые сущности-контейнеры

(удаление таких сущностей-контейнеров реализуется в этих ОС рекурсивно, с использованием удаления сущностей-объектов и пустых сущностей-контейнеров), а при удалении сущности объекта должно быть проверено отсутствие на него других «жестких» ссылок. При создании сущности с прямой меткой (это возможно в сущности-контейнере только с прямой меткой, все входящие в состав которого сущности обладают также прямой меткой) право доступа владения добавляется к индивидуальной роли учетной записи пользователя, от имени которой функционирует субъект-сессия x, к которой она имеет доступ на запись. При создании сущности с косвенной меткой (это возможно в сущности-контейнере, содержащей сущности только с косвенной меткой) все роли или административные роли, имеющие права доступа к единственной существующей условию 8 соответствующей сущности-контейнеру, получают эти права доступа к создаваемой сущности. При удалении сущности в разделяемой сущности-контейнере требуется наличие у субъект-сессии доступа на чтение к роли или административной роли, обладающей правом доступа владения к удаляемой сущности.

Де-юре правила вида *create_hard_link*$(x, y, name, z)$ и *delete_hard_link*$(x, y, name, z)$ позволяют субъект-сессии x создать или удалить соответственно в составе сущности-контейнера z (к которой субъект-сессия x должна иметь доступ на запись и обладать текущей ролью или административной ролью, имеющей к z право доступа на выполнение *execute*$_r$) «жесткую» ссылку на сущность-объект y. При создании «жесткой» ссылки на сущность y учитываются права доступа к сущностям-контейнерам, ее содержащим. Так как в ОС возможно создание в одной сущности-контейнере нескольких «жестких» ссылок (с разными именами) на одну сущность-объект, то в условиях применения правила *create_hard_link* не требуется отсутствие сущности-объекта y в составе сущности-контейнера z, а в правиле *delete_hard_link* указывается имя «жесткой» ссылки, с которым она будет удалена. Поскольку «жесткие» ссылки создаются только на объекты, то не требуется проверки на появление циклов в иерархии сущностей. При создании «жесткой» ссылки учитывается вид ее метки, в результате, если у сущности y метка прямая, то она прямая у сущности-контейнера z, а если у сущности y метка косвенная, то «жесткая» ссылка на нее может быть осуществлена только когда сущность-контейнер z подчинена в иерархии единственной удовлетворяющей условию 8 сущности-контейнеру с прямой меткой старшей в иерархии z и y. При удалении проверяется, действительно ли удаляется «жесткая» ссылка (т. е. есть еще другие ссылки на

нее), и учитывается, осуществляется ли удаление «жесткой» ссылки в разделяемой сущности-контейнере z или нет. В реальных защищенных ОС удаление сущности-объекта и «жесткой» ссылки на него реализуются, как правило, одной функцией. В то же время, так как формально результаты таких удалений существенно отличаются, то для удобства в рамках модели заданы два правила.

Де-юре правила вида *create_role*$(x, r, name, rz)$, *create_hard_link_role*(x, r, rz), *delete_role*(x, r, rz) и *delete_hard_link_role*(x, r, rz) позволяют субъект-сессии x создать или удалить роль или административную роль r или «жесткую» ссылку на нее (изменить иерархию ролей), входящую в состав роли или административной роли rz, к которой субъект-сессия x должна иметь доступ на запись и не являющейся индивидуальной административной, индивидуальной ролью учетной записи пользователя или общей ролью. При этом нельзя удалять или создавать роли или административные роли, «жесткие» ссылки на них, являющиеся индивидуальными ролями и индивидуальными административными ролями учетных записей пользователей, а также общей ролью и специальными административными ролями из множества SAR. Для применения правил необходимо наличие у субъект-сессии x административных доступов на чтение и запись к административным ролям *roles_admin_role* или *admin_roles_admin_role* для действий с ролями или административными ролями соответственно. При этом доступ на запись к этим административным ролям следует требовать, так как создание, удаление ролей или «жестких» ссылок на них являются существенными преобразованиями параметров безопасности системы. При реализации правил в последующем состоянии иерархия ролей модифицируется (назначаются или удаляются административные права доступа) в соответствии с условиями консистентности. Аналогично правилам администрирования учетных записей пользователей при реализации правил создания или удаления ролей права доступа к ним могут изменяться у административных ролей без явного получения к ним субъект-сессией x административного доступа на запись. Так же, как при удалении сущностей или «жестких» ссылок на них, при удалении роли или административной роли проверяется, что ей в иерархии не подчинены другие роли, а при удалении «жесткой» ссылки на роль, что эта ссылка не последняя. Однако при создании «жесткой» ссылки на роль (так как в отличие от сущностей могут создаваться «жесткие» ссылки на роли-контейнеры, содержащие подчиненные роли) проверяется, что это не приведет к

возникновению в иерархии циклов, т. е. нарушению отношения частичного порядка.

Де-юре правила вида *rename_entity*(x, y, *old_name*, *name*, z) и *rename_role*(x, ry, *name*) позволяют субъект-сессии x переименовать сущность y, входящую в состав сущности-контейнера z, к которой субъект-сессия x должна иметь доступ на запись и обладать текущей ролью или административной ролью, имеющей к ней право доступа на выполнение $execute_r$, или соответственно переименовать роль или административную роль ry во всех ролях-контейнерах, в которые она входит (так как каждая роль или административная роль имеет в отличие от сущностей уникальное имя) и к которым субъект-сессия x должна иметь административные доступы на запись. Поскольку на сущность-объект может быть несколько «жестких» ссылок в одной сущности-контейнере и, следовательно, несколько имен, то в правило добавлен параметр, указывающей старое имя сущности, подлежащее замене. Для ролей такой параметр не требуется, так как роль должна иметь в системе уникальное имя. Правило переименования роли не использовалось в предыдущих ДП-моделях и включено в МРОСЛ ДП-модель с учетом реализованного в ней представления ролей как аналогов сущностей-контейнеров. В связи с этим для переименования роли или административной роли требуется наличие у субъект-сессии доступа на чтение к административной роли *roles_admin_role* или *admin_roles_admin_role* соответственно, при этом нельзя переименовывать роли, являющиеся индивидуальными ролями и индивидуальными административными ролями учетных записей пользователей, а также общей ролью и специальными административными ролями из множества SAR. При переименовании сущности учитывается, осуществляется ли оно в разделяемой сущности-контейнере z или нет.

Де-юре правило вида *read_container*(x, y, z) позволяет субъект-сессии x «считать» в сущность-объект z (например, в реальной защищенной ОС в сущность-«рабочий стол»), к которой она должна иметь доступ на запись, содержимое (имена входящих в нее непосредственно сущностей или ролей) сущности-контейнера, роли или административной роли y, к которой x должен иметь права доступа на чтение и выполнение, с учетом прав доступа к сущностям-контейнерам, содержащим y.

Де-юре правила вида *get_entity_attr*(x, y, z), *get_subject_attr*(x, y, z) и *get_role_attr*(x, y, z) позволяют субъект-сессии x «считать» в сущность-объект z (например, в реальной защищенной ОС в сущность-«рабочий стол»), к которой она должна иметь доступ на за-

пись, атрибуты сущности, субъект-сессии, роли или административной роли y соответственно. В случае, когда y является сущностью, требуется, чтобы субъект-сессия x могла получить к ней доступ с учетом прав доступа x к сущностям-контейнерам, содержащим y. Для сущности y выдаются следующие атрибуты:

- вид метки;
- для сущности-контейнера является ли она разделяемой;
- для сущности-объекта число «жестких» ссылок на нее;
- роли или административные роли и имеющиеся у них права доступа к сущности (в случае, когда x имеет либо административный доступ на чтение к роли-«владельцу» сущности y, либо к административной роли *entities_admin_role*).

Для субъект-сессии y выдаются следующие атрибуты:

- учетная запись пользователя, от имени которой она функционирует;
- роль-«владелец» субъект-сессии (в случае, когда x имеет либо административный доступ на чтение к роли-«владельцу» субъект-сессии y, либо к административной роли *subjects_admin_role*);
- текущие административные доступы субъект-сессии к ролям или административным ролям (в случае, когда x имеет либо административный доступ на чтение к роли-«владельцу» субъект-сессии y, либо к административной роли *subjects_admin_role*).

Для роли или административной роли y выдаются следующие атрибуты:

- вид метки (всегда *true*);
- является ли она разделяемой (всегда *true*, выдается для общности формата данных с аналогичными данными для сущностей);
- число «жестких» ссылок на роль или административную роль (в случае, когда x имеет административный доступ на чтение к административной роли *roles_admin_role* или *admin_roles_admin_role* соответственно);
- административные роли и имеющиеся у них права доступа к роли или административной роли y (в случае, когда x имеет либо административный доступ на чтение к административной роли *roles_admin_role* или *admin_roles_admin_role* соответственно).

Де-юре правило вида *set_container_attr*(x, y, t) позволяет субъект-сессии x задать сущности-контейнеру y является ли она разделяемой или нет. При этом субъект-сессия x должна иметь либо

административный доступ на чтение к роли-«владельцу» контейнера y, либо к административной роли *entities_admin_role*, а также требуется, чтобы доступ к y мог быть предоставлен x с учетом ее прав доступа к сущностям-контейнерам, содержащим y.

Де-юре правило вида *create_first_subject*(x, u, y, z) позволяет субъект-сессии x с использованием сущности y и учетной записи пользователя u создать от имени u новую субъект-сессию z. Для этого требуется наличие у субъект-сессии x доступа на чтение к роли, обладающей правом доступа на выполнение к сущности y (к которой субъект-сессия x может получить доступ с учетом прав доступа к сущностям-контейнерам, содержащим сущность y). После создания субъект-сессии z она (в отличие от предшествующих ДП-моделей) получает доступы на запись и чтение к индивидуальной роли *u_c* и общей роли *common_role* и доступ на чтение к индивидуальной административной роли *u_admin*. При этом индивидуальная роль*u_c* получает право доступа владения к субъект-сессии z.

Де-юре правила вида *create_subject*(x, y, z) и *delete_subject*(x,z) позволяют субъект-сессии x создать или удалить соответственно субъект-сессию z. При создании требуется наличие у субъект-сессии x доступа на чтение к роли, обладающей правом доступа на выполнение к сущности y (к которой субъект-сессия x может получить доступ с учетом прав доступа к сущностям-контейнерам, содержащим сущность y). После создания субъект-сессии z она получает доступы на запись и чтение к индивидуальной роли *user*(x)*_c* и общей роли *common_role* и доступ на чтение к индивидуальной административной роли *user*(x)*_admin*, соответствующим ее учетной записи пользователя (у субъект-сессий x и z общая учетная запись пользователя), и непосредственно подчиняется в иерархии субъект-сессии x. При этом индивидуальная роль *user*(x)*_c* получает право доступа владения к субъект-сессии z. При удалении субъект-сессии z требуется, чтобы ей в иерархии не подчинялись другие субъект-сессии (в противном случае удаление надо начинать с них), и требуется наличие у субъект-сессии x доступа на чтение к роли-«владельцу» (обладающей правом доступа владения) к субъект-сессии z.

Таким образом, полностью определена рассматриваемая в рамках базового уровня иерархического представления МРОСЛ ДП-модели абстрактная система (автомат), а именно заданы элементы, используемые для описания ее состояний, и правила перехода системы из состояния в состояние. При этом при задании правил так же, как для состояний, учитывалась специфика функционирования механизма управления доступом реальной ОССН.

3.3.4. Обоснование выполнения условий консистентности модели

При разработке базового уровня иерархического представления МРОСЛ ДП-модели задание состояний абстрактной системы, правил перехода между ними, несомненно, велось с целью обеспечения выполнения условий консистентности модели. То есть таким образом, чтобы при условии нахождения системы $\Sigma(G^*,\ OP,\ G_0)$ в начальном состоянии G_0, удовлетворяющем условиям консистентности, каждое последующее состояние, полученное при применении любого де-юре правила перехода системы, определенного в табл. 3.1, как и сам такой переход, также удовлетворяли условиям консистентности модели. Иными словами, должны удовлетворять условиям консистентности модели все состояния и все переходы любой траектории функционирования системы, полученной из ее начального состояния, удовлетворяющего условиям консистентности, путем применения конечной последовательности правил перехода системы из состояния в состояние.

Однако для того чтобы строго обосновать выполнение условий консистентности модели на траекториях функционирования системы, только описания состояний системы и правил перехода системы из состояния в состояние явно недостаточно. Для этого требуется доказать соответствующее утверждение, что также по сути является верификацией описания базового уровня иерархического представления МРОСЛ ДП-модели в рамках математической нотации.

Утверждение 3.1. Пусть G_0 — начальное состояние системы $\Sigma(G^*,\ OP,\ G_0)$, удовлетворяющее условиям консистентности модели. Тогда для любой траектории $G_0 \vdash_{op_1} G_1 \vdash_{op_2} \ldots \vdash_{op_N} G_N$, где $N \geqslant 1$, в состоянии G_N выполняются условия консистентности модели, а также переход $G_{N-1} \vdash_{op_N} G_N$ удовлетворяет условиям этого предположения.

Доказательство. Докажем утверждение индукцией по длине N траектории функционирования системы.

Пусть $N = 0$, тогда по условию утверждения состояние G_0 удовлетворяет условиям консистентности модели.

Пусть $N > 0$ и утверждение верно для всех траекторий длины $0 \leqslant L < N$. Пусть $G_0 \vdash_{op_1} G_1 \vdash_{op_2} \ldots \vdash_{op_N} G_N$ — траектория функционирования системы длины N. По предположению индукции состояние G_{N-1} удовлетворяет условиям консистентности модели, а также каждый переход $G_{i-1} \vdash_{op_i} G_i$ удовлетворяет этим условиям, где $1 \leqslant i < N$.

Рассмотрим правило перехода системы из состояния в состояние op_N. Если условия его применения не выполняются в состоянии G_{N-1}, то по определению правил справедливо равенство $G_{N-1} = G_N$, и по предположению индукции состояние G_N удовлетворяет условиям консистентности модели и переход $G_{N-1} \vdash_{op_N} G_N$ удовлетворяет этим условиям. Пусть условия применения правила op_N выполняются в состоянии G_{N-1}.

Обоснуем выполнение условий консистентности модели в состоянии G_N и при переходе $G_{N-1} \vdash_{op_N} G_N$.

Роли или административные роли получают права доступа к субъект-сессиям только в случае, когда op_N является одним из правил вида *set_subject_owner*(x, r, r', y), *create_first_subject*(x, u, y, z) и *create_subject*(x, y, z), в результате применения которых может быть дано только право доступа владения own_r, при этом отсутствуют правила, в результате применения которых субъект-сессии могли бы получать доступы к субъект-сессиям. Следовательно, с учетом предположения индукции в состоянии G_N выполнено условие 1 консистентности модели (требований к переходу $G_{N-1} \vdash_{op_N} G_N$ в этом условии не содержится).

Новые роли или административные роли создаются в результате применения правил вида *create_user*(x, u) и *create_role*$(x, r, name, rz)$, в результатах которых все создаваемые роли или административные помечаются как «разделяемые контейнеры» и к ним всем административным ролям дается права доступа $execute_r$, а также созданной индивидуальной административной роли учетной записи пользователя дается это право доступа ко всем ролям и административным ролями. При этом в МРОСЛ ДП-модели отсутствуют правила, позволяющие изменить эти параметры ролей или административных ролей. Следовательно, с учетом предположения индукции в состоянии G_N выполнено условие 2 консистентности модели (требований к переходу $G_{N-1} \vdash_{op_N} G_N$ в этом условии не содержится).

Административные права доступа к ролям или административным ролям даются только административным ролям в результате применения правил *create_user*(x, u), *grant_admin_rights*$(x, ar, \{(r, \alpha_{rj})\colon\ 1 \leqslant j \leqslant k\})$, *create_role*$(x, r, name, rz)$, *create_hard_link_role*(x, r, rz). Управление доступом к сущности может быть осуществлено субъект-сессией только при использовании правил вида *grant_rights* $(x, r, \{(y, \alpha_{rj})\colon\ 1 \leqslant j \leqslant k\})$, *remove_rights*$(x, r, \{(y, \alpha_{rj})\colon\ 1 \leqslant j \leqslant \leqslant k\})$, *set_entity_owner*$(x, r, r', y)$ и *set_container_attr*(x, y, t) при наличии у нее в состоянии G_{N-1} текущей роли или административной роли, обладающей правом доступа владения к этой сущности.

Право доступа владения к создаваемой сущности или субъект-сессии дается соответствующей индивидуальной роли учетной записи пользователя в результате применения правил вида *create_object* $(x, y, yd, name, z)$, *create_container*$(x, y, yd, name, z)$, *create_first_subject*(x, u, y, z) и *create_subject*(x, y, z), изменение такой роли-«владельца» возможно только с применением правил вида *set_entity_owner*(x, r, r', y), и *set_subject_owner*(x, r, r', y) при наличии у инициирующей их выполнение субъект-сессии x текущих административных ролей *entities_admin_role* или *subjects_admin_role* соответственно. Право доступа владения к ролям или административным ролям дается только административным ролям *roles_admin_role* или *admin_roles_admin_role* в результате применения правил вида *create_user*(x, x', u) и *create_role*$(x, r, name, rz)$, и нет правил, позволяющих изменить такую роль-«владельца» роли или административной роли.

Управление доступом к роли или административной роли осуществляется в результате применения правил вида *create_user*(x, u), *delete_user*(x, u), *create_role*$(x, r, name, rz)$, *delete_role*(x, r, rz), *create_hard_link_role*(x, r, rz), *delete_hard_link_role*(x, r, rz), *grant_admin_rights*$(x, ar, \{(r, \alpha_{rj})\colon\ 1 \leqslant j \leqslant k\})$ и *remove_admin_rights*$(x, ar, \{(r, \alpha_{rj})\colon\ 1 \leqslant j \leqslant k\})$, в условиях применения которых также проверяется наличие у субъект-сессии x текущих административных ролей *roles_admin_role* или *admin_roles_admin_role* соответственно.

Следовательно, с учетом предположения индукции в состоянии G_N и при переходе $G_{N-1} \vdash_{op_N} G_N$ выполнено условие 3 консистентности модели (требований к переходу в этом условии не содержится).

Субъект-сессия может получить доступ к сущности, создать «жесткую» ссылку на нее, получить или изменить ее параметры, права доступа к ней, активизировать из нее субъект-сессию только в случае, когда op_N является одним из правил вида *access_read*(x, y), *access_write*(x, y), *grant_rights*$(x, r, \{(y, \alpha_{rj})\colon\ 1 \leqslant j \leqslant k\})$, *remove_rights*$(x, r, \{(y, \alpha_{rj})\colon\ 1 \leqslant j \leqslant k\})$, *set_entity_owner*$(x, r, r', y)$, *create_hard_link*$(x, y, name, z)$, *read_container*$(x, y, z)$, *get_entity_attr*$(x, y, z)$, *set_container_attr*$(x, y, t)$, *create_first_subject*$(x, u, y, z)$ и *create_subject*(x, y, z), в которых проверяется выполнение равенства *execute_container*$_{N-1}(x, c, y) =$ *true*, где контейнер $c \in C_{N-1}$. Таким образом, с учетом предположения индукции в состоянии G_N и при переходе $G_{N-1} \vdash_{op_N} G_N$ выполнено условие 4 консистентности модели.

Для создания, переименования или удаления сущности, роли или административной роли или «жесткой» ссылки на нее в сущнос-

ти-контейнере, роли или административной роли субъект-сессия может использовать только правила вида *create_object*$(x, y, yd, name, z)$, *create_container*$(x, y, yd, name, z)$, *delete_entity*(x, y, z), *create_hard_link*$(x, y, name, z)$, *delete_hard_link*$(x, y, name, z)$, *create_role*$(x, r, name, rz)$, *delete_role*(x, r, rz), *create_hard_link_role*(x, r, rz), *delete_hard_link_role*(x, r, rz), *rename_entity*$(x, y, old\_name, name, z)$ и *rename_role*$(x, ry, name)$, в условиях применения которых требуется наличие у субъект-сессии в состоянии G_{N-1} доступа или административного доступа на запись к этой сущности-контейнеру, роли или административной роли соответственно и требуется наличие у x текущей роли или административной роли, обладающей к последней правом доступа на выполнение *execute*$_r$. Для случая, когда эти правила применяются для переименования, удаления сущности или «жесткой» ссылки на сущность в сущности-контейнере, помеченной как разделяемая, в условиях применения правил проверяется наличие у субъект-сессии x текущей роли или административной роли, обладающей правом доступа владения *own*$_r$ к сущности y.

Сущность-контейнер может быть создана только с применением правила вида *create_container*$(x, y, yd, name, z)$, в результатах применения которого обеспечивается равенство *false* для нее функции разделяемых контейнеров. Изменение метки разделяемости сущности-контейнера возможно только с использованием правила вида *set_container_attr*(x, x', t), в условиях которого проверяется наличие у субъект-сессии x текущей роли, обладающей правом доступа владения к этой сущности-контейнеру y или доступа на чтение к административной роли *entities_admin_role*.

Для получения субъект-сессией x данных о ролях или административных ролях, обладающих правами доступа к сущности y, используется правило вида *get_entity_attr*(x, y, z), в условиях применения которого проверяется либо наличие у субъект-сессии x доступа на чтение к административной роли *entities_admin_role*, либо роли или административной роли, обладающей правом доступа владения *own*$_r$ к сущности y. Для получения субъект-сессией x данных о ролях или административных ролях, обладающих правом доступа владения к субъект-сессии y, или имеющихся у этой субъект-сессии текущих ролях или административных ролях используется правило вида *get_subject_attr*(x, y, z), в условиях применения которого проверяется наличие у субъект-сессии x либо доступа на чтение к административной роли *subjects_admin_role*, либо к роли или административной роли, обладающей правом доступа владения *own*$_r$ к субъект-сессии y.

Права доступа ролей или административных ролей могут быть изменены в результате применения правил четырех групп видов. К первой группе относятся правила вида: *grant_rights*$(x, r, \{(y, \alpha_{rj})$: $1 \leqslant j \leqslant k\})$, *remove_rights*$(x, r, \{(y, \alpha_{rj})$: $1 \leqslant j \leqslant k\})$, *set_entity_owner*$(x, r, r', y)$, *set_subject_owner*$(x, r, r', y)$, *grant_admin_rights*$(x, ar,$ $\{(r, \alpha_{rj})$: $1 \leqslant j \leqslant k\})$, *remove_admin_rights*$(x, ar, \{(r, \alpha_{rj})$: $1 \leqslant j \leqslant$ $\leqslant k\})$, *create_object*$(x, y, yd, name, z)$, *create_container*$(x, y, yd, name,$ $z)$, в условиях применения которых проверяется наличие у субъект-сессии x административного доступа на запись к соответствующей роли или административной роли, права доступа которой изменяются при применении правил. При реализации правил второй группы видов: *create_first_subject*(x, u, y, z) и *create_subject*(x, y, z) после создания субъект-сессии z ей дается административный доступ на запись к существующей по условию 9 консистентности модели индивидуальной роли ее учетной записи пользователя, после чего этой роли дается право доступа владения к субъект-сессии z. В условиях правил третьей группы видов: *create_user*(x, u), *create_role*$(x, r, name,$ $rz)$, *create_hard_link_role*(x, r, rz) проверяется (за исключением последнего правила, когда административным ролям назначаются административные права доступа $read_r$ к ролям или административным ролям с учетом изменения их иерархии) наличие у субъект-сессии x административного доступа на запись к административным ролям *roles_admin_role* и *admin_roles_admin_role*, которым даются права доступа владения к создаваемым в результате применения правил ролям или административным ролям. Кроме того, автоматически административным ролям назначаются административные права доступа $read_r$ или $execute_r$ к ролям или административным ролям при изменении их иерархии в соответствии условием 2 консистентности модели и определением 2.3. При реализации правил четвертой группы видов: *delete_user*(x, u), *delete_entity*(x, y, z), *delete_role*(x, r, rz), *delete_subject*(x, z) у ролей и административных ролей удаляются права доступа к соответствующим удаляемым сущностям, субъект-сессиям, ролям или административным ролям.

Создание, переименование, удаление, получение параметров роли или административной роли, «жесткой» ссылки на нее, числа «жестких» ссылок к ней, множества административных ролей, обладающих к ней правами доступа, может быть реализовано с использованием правил вида *create_role*$(x, r, name, rz)$, *delete_role*(x, r, rz), *create_hard_link_role*(x, r, rz), *delete_hard_link_role*(x, r, rz), *rename_role*$(x, ry, name)$, *get_role_attr*(x, y, z), в условиям применения которых (в последнем правиле в результатах применения) проверяется

наличие у субъект-сессии x текущей административной роли *roles_admin_role* или *admin_roles_admin_role* соответственно.

Таким образом, с учетом предположения индукции в состоянии G_N и при переходе $G_{N-1} \vdash_{op_N} G_N$ выполнено условие 5 консистентности модели.

Для создания или удаления учетной записи пользователя используются правила вида *create_user*(x, u) и *delete_user*(x, u), в условиях применения которых проверяется наличие у субъект-сессии x текущих административных ролей *users_admin_role*, *roles_admin_role* и *admin_roles_admin_role*, а при использовании первого правила административного доступа на запись к административным ролям *roles_admin_role* и *admin_roles_admin_role*. Также проверяется, что множество субъект-сессий, функционирующим от имени данной учетной записи пользователя u, является пустым.

Для получения параметров учетной записи пользователя используется правило вида *get_user_attr*(x, u, z), в условиях применения которого проверяется, что либо субъект-сессия x функционирует от имени учетной записи пользователя u, либо x имеет текущую административную роль *users_admin_role*. Таким образом, с учетом предположения индукции в состоянии G_N и при переходе $G_{N-1} \vdash_{op_N} G_N$ выполнено условие 6 консистентности модели.

Субъект-сессия может активизировать новую субъект-сессию только с использованием правил вида *create_first_subject*(x, u, y, z) и *create_subject*(x, y, z), в которых проверяется наличие у субъект-сессии x текущей роли или административной роли, обладающей правом доступа на выполнение к сущности y. Субъект-сессия x может удалить субъект-сессию z только с использованием правила *delete_subject*(x, z), в условиях применения которого проверяется наличие у x текущей роли или административной ролью, имеющей право доступа владения к z. Таким образом, с учетом предположения индукции в состоянии G_N и при переходе $G_{N-1} \vdash_{op_N} G_N$ выполнено условие 7 консистентности модели.

В рамках базового уровня МРОСЛ ДП-модели отсутствуют правила, позволяющие изменять вид метки у сущностей. Непосредственное изменение прав доступа ролей или административных ролей к сущности может быть осуществлено только с применением правил вида *grant_rights*$(x, r, \{(y, \alpha_{rj})\colon 1 \leqslant j \leqslant k\})$, *remove_rights*$(x, r, \{(y, \alpha_{rj})\colon 1 \leqslant j \leqslant k\})$, *set_entity_owner*$(x, r, r', y)$, в условиях которых проверяется, что сущность обладает прямой меткой. При этом в результатах применения этих правил задано, что при изменение прав доступа к сущности-контейнеру с прямой меткой, содержащей

только сущности с косвенными метками, соответственно изменяются права доступа к этим сущностям и сущностям, им подчиненным в иерархии. Новые сущности создаются только с использованием правил вида *create_object*$(x, y, yd, name, z)$ и *create_container*$(x, y, yd, name, z)$, в результатах применения которых задается вид их метки. Кроме этих правил иерархия сущностей может быть изменена с использованием правила вида *create_hard_link*$(x, y, name, z)$. При этом в условиях всех этих правил проверяется, что если у создаваемой сущности y либо «жесткой» ссылки на нее метка прямая, то у сущности-контейнера z, в котором она создается, метка также прямая, а значит, если у некоторой сущности-контейнера метка косвенная, то у всех сущностей ниже ее в иерархии она также косвенная. Также проверяется, что все сущности внутри сущности-контейнера должны иметь метку одного вида. При создании сущности y с косвенной меткой с применением первых двух правил права доступа всех ролей и административных ролей к ней задаются равными соответствующим правам доступа ролей и административных ролей к сущности-контейнеру z, которая либо имеет прямую метку, либо по предположению индукции права доступа к ней совпадают с правами к единственной существующей по предположению индукции старшей ее в иерархии сущности-контейнера с прямой меткой. В правиле вида *create_hard_link*$(x, y, name, z)$ проверяется, что «жесткая» ссылка на сущность с косвенной меткой создается в иерархии этой единственной сущности-контейнера. Также в рамках базового уровня МРОСЛ ДП-модели отсутствуют правила, позволяющие изменять вид метки у ролей или административных ролей. Новые роли создаются только с использованием правил вида *create_user*(x, u) и *create_role*$(x, r, name, rz)$, в результатах применения которых для всех новых ролей указывается, что их метки прямые. Таким образом, с учетом предположения индукции в состоянии G_N и при переходе $G_{N-1} \vdash_{op_N} G_N$ выполнено условие 8 консистентности модели.

Индивидуальная административная роль и индивидуальная роль учетной записи пользователя u создаются только с использованием правила вида *create_user*(x, u), при этом административные права доступа на чтение, запись и выполнение к индивидуальной роли даются индивидуальной административной роли учетной записи пользователя u. Данные роли удаляются только при удалении учетной записи пользователя с использованием правила вида *delete_user*(x, u). Также при использовании правила вида *create_user*(x, u) индивидуальной административной роли учетной записи пользователя u даются права доступа на чтение, запись и вы-

полнение к общей роли *common_role*. Кроме того, по условию 9 консистентности модели административные роли *roles_admin_role* и *admin_roles_admin_role* с применением правила вида *remove_admin_rights*$(x, ar, \{(r, \alpha_{rj})\colon 1 \leqslant j \leqslant k\})$ нельзя использовать для нарушения заданной иерархии и правил назначения административных прав доступа к индивидуальным административным ролям, индивидуальным ролям учетных записей пользователей и общей роли. В соответствии с условиями применения правил вида *delete_role*(x, r, rz), *create_hard_link_role*(x, r, rz), *delete_hard_link_role*(x, r, rz) и *rename_role*$(x, ry, name)$ эти роли и административные роли нельзя удалить, переименовать, создать или удалить «жесткую» ссылку на них.

Следовательно, с учетом предположения индукции в состоянии G_N и при переходе $G_{N-1} \vdash_{op_N} G_N$ выполнено условие 9 консистентности модели.

Создание субъект-сессии может быть осуществлено с использованием правил вида *create_first_subject*(x, u, y, z) и *create_subject*(x, y, z), в результатах применения которых обеспечивается, что эта субъект-сессия получает доступ на чтение к индивидуальной административной роли, доступы на запись и чтение к индивидуальной роли ее учетной записи пользователя и общей роли. Также индивидуальная роль получает право доступа владения к этой субъект-сессии.

При удалении субъект-сессии с использованием правила вида *delete_subject*(x, z) удаляются все ее административные доступы, в том числе к индивидуальной, индивидуальной административной и общей роли.

Следовательно, с учетом предположения индукции в состоянии G_N и при переходе $G_{N-1} \vdash_{op_N} G_N$ выполнено условие 10 консистентности модели.

Шаг индукции доказан.

Утверждение доказано.

Таким образом, обосновано, что заданные в рамках базового уровня иерархического представления МРОСЛ ДП-модели правила перехода системы из состояния в состояние соответствуют условиям консистентности модели.

4 Event-B спецификация базового уровня МРОСЛ ДП-модели

Как отмечено в главе 1, компонент доверия ADV_SMP.1 «Формальная модель политики безопасности» требует наличия формальной модели политики безопасности, причем язык представления формальной модели может быть либо математическим, либо формализованным. Базовый уровень МРОСЛ ДП-модели был изложен на математическом языке в главе 3. В данной главе рассматривается перевод базового уровня модели на формализованный язык формального метода Event-B [21], который позволит использовать автоматические и интерактивные инструменты для ее верификации. Глава рассчитана на читателя, который впервые сталкивается с текстом на языке Event-B, так что все основные используемые конструкции будут поясняться по ходу их появления. При желании ознакомиться с Event-B предлагается обратить внимание на Приложение Б либо ознакомиться с официальной документацией [37].

Имеется несколько формальных методов, при помощи которых можно было бы представить МРОСЛ ДП-модель в формализованном виде. Примерами таких методов являются ASM [38], Alloy [39], B [40], Event-B, TLA+ [41], VDM [42], Z [43]. Нами был выбран формальный метод Event-B, так как он отличается простым и понятым языком, спецификации на котором разрабатываются и верифицируются с помощью хорошо апробированной платформы Rodin [44]. Кроме того, Event-B хорошо себя зарекомендовал как средство специфицирования так называемых систем, управляемых событиями (event-driven systems). Так как ОС может рассматриваться как подобная система, то выбор Event-B для специфицирования механизма управления доступом ОС представляется весьма разумным. При этом структура спецификаций на Event-B хорошо соответствует компонентам МРОСЛ ДП-модели, что упрощает задачу перевода модели на данный формализованный язык.

4.1. Формальные спецификации

Под формальной спецификацией некоторой системы далее подразумевается автоматная модель (здесь термины «модель» и «спе-

цификация» используются как синонимы), которая состоит из следующих частей:

- набора состояний, описываемых некоторыми внутренними данными или переменными системы (возможные комбинации значений переменных задают возможные состояния);
- выделенного непустого множества возможных начальных состояний (может быть, включающего только одно состояние);
- набора событий и правил, определяющих переходы между состояниями (изменения значений переменных) по произошедшим событиям;
- набора свойств или требований, которые должны выполняться во всех достижимых состояниях системы.

Задача доказательства корректности спецификации или ее верификация заключается в подтверждении при помощи симуляции, статического анализа или формального математического доказательства того, что сформулированные в рамках спецификации свойства и требования действительно выполняются во всех достижимых состояниях, т. е. в тех состояниях, в которые система может перейти из некоторого начального состояния в результате произвольных последовательностей событий в соответствии с описанными в спецификации правилами. Благодаря своей формальной математической природе, спецификации могут быть верифицированы с помощью различных автоматических инструментов. В случае Event-B можно воспользоваться, например, инструментами, которые являются частью платформы Rodin.

В Event-B каждая спецификация состоит из компонентов двух типов: *контекстов* и *машин* (рис. 4.1). Контексты содержат статическую, неизменяемую часть спецификации: определения *множеств* и *констант*, а также *аксиомы*, являющиеся предикатами, в виде которых описываются типы и свойства констант и множеств. Аксиомы принимаются истинными без доказательства.

В отличие от контекстов машины содержат динамическую часть спецификации. Машины имеют доступ к элементам контекста через механизм «видения», что позволяет использовать определенные там константы и множества. В машинах содержатся *переменные*, *инварианты*, *события*. Переменные, как и константы, соответствуют математическим объектам: они могут быть множествами, бинарными отношениями, функциями, числами, принимать значения логического типа и т. д. Значение переменных формируют текущее состояние спецификации, а инварианты — предикаты, определенные на множестве переменных спецификации, — ограничивают его.

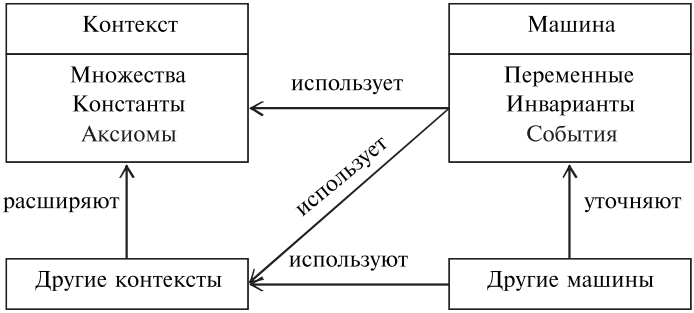

Рис. 4.1. Структура спецификации на Event-B

Текущее состояние спецификации может быть изменено событием. Каждое событие обычно состоит из названия, *параметров*, *охранных условий* и *действий*. Охранные условия являются обязательными условиями в виде набора предикатов, ограничивающих множество возможных состояний спецификации, в которых данное событие может случиться. Типы параметров события также задаются в блоке охранных условий. Действия изменяют текущее состояние спецификации за счет модификации значения переменных спецификации. Определенные в машине инварианты должны сохраняться в результате любых модификаций значений переменных, поэтому корректность каждого изменения состояния необходимо доказывать.

Верификация спецификаций на Event-B осуществляется при помощи формального математического доказательства с использованием включенных в состав платформы Rodin автоматических и интерактивных средств. Интерактивные средства позволяют проводить доказательства вручную, причем их корректность затем проверяется одним из компонентов платформы. Автоматическое же доказательство осуществляется средствами встроенных инструментов, а также SMT-решателями, которые добавляются в Rodin с помощью плагина, входящего в состав AstraVer Toolset. Возможна и комбинация интерактивного и автоматического доказательств, при которой утверждение для доказательства перед подачей на вход автоматическим инструментам сначала вручную разбивается на несколько более простых частных случаев.

Для каждого требующего доказательства случая — неоднозначность выражений, сохранность инвариантов, корректность проведенного пошагового уточнения (если данная техника была использована) — Rodin генерирует соответствующие утверждения для доказательства, причем платформа решает проблему поддержки ак-

туальности сгенерированных утверждений и выполненных доказательств в случае изменений в спецификации. Полное доказательство спецификации означает, что доказаны все сгенерированные утверждения.

4.2. Связь элементов МРОСЛ ДП-модели и элементов Event-B спецификации

Основными компонентами МРОСЛ ДП-модели, которые необходимо перевести на формализованный язык Event-B, являются определяемые в ней функции и множества (а также элементы этих множеств), условия консистентности и правила перехода системы из состояния в состояние. Функции и множества определяют множество всех возможных состояний системы G^*, переход между которыми (то есть изменение значений данных функций и множеств) задается соответствующими правилами. Условия и требования формируют свойства, которые должны выполняться в каждом состоянии системы.

Данные компоненты модели хорошо соответствуют конструкциям языка Event-B (рис. 4.2): требования и условия могут быть выражены в виде инвариантов, правила перехода системы из состояния в состояние — в виде событий, а определенные в модели множества и функции — в виде констант, множеств, переменных. Однако из-за особенностей Event-B некоторые компоненты МРОСЛ ДП-модели реализованы в спецификации иначе, а другие не реализованы совсем. Кроме того, в спецификации также присутствуют дополнительные элементы, введенные для упрощения записи некоторых свойств. Причины подобных исключений будут описаны далее в тексте данной главы.

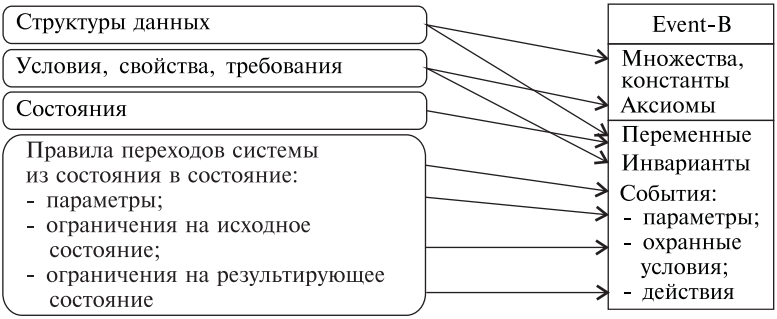

Рис. 4.2. Соответствие между компонентами МРОСЛ ДП-модели и Event-B

4.3. Контекст

Начнем рассмотрение Event-B спецификации базового уровня МРОСЛ ДП-модели с контекста — части спецификации, в которой содержатся неизменяемые в процессе функционирования системы сущности — определения констант, множеств, аксиом.

Первым определенным в контексте элементом является конечное (ключевое слово finite Event-B) множество Union, которое не имеет аналога в модели. Оно служит общим множеством, элементы которого могут быть сущностями, субъект-сессиями, ролями или административными ролями МРОСЛ ДП-модели. Другими словами, множество Union является базовым типом, который имеют элементы упомянутых выше множеств (в некотором смысле это аналог базового класса Object из языков программирования Java, C#):

```
sets
  Union
axioms
  @UnionIsFinite
    finite(Union)
```

Каждой аксиоме поставлена в соответствие метка (в данном случае меткой является @UnionIsFinite), которая служит в качестве ее идентификатора. В Event-B метки присутствуют также у инвариантов, охранных условий и действий событий. Мы будем использовать данные метки, чтобы ссылаться на интересующие нас элементы спецификации.

Следующим определяемым множеством является множество Names, которое соответствует множеству возможных имен сущностей, ролей, административных ролей $NAMES$ модели NAMES модели (здесь и далее курсивом отмечаются элементы математической нотации МРОСЛ ДП-модели):

```
sets
  Names
```

Множество Accesses соответствует множеству видов прав доступа R_a. Элементы этого множества — это доступ на чтение ReadA ($read_a$ в модели) и доступ на запись WriteA ($write_a$), выражаемые в виде констант:

```
sets
  Accesses
constants
  ReadA
  WriteA
```

```
axioms
  @AccessesPartition
    partition(Accesses, {ReadA}, {WriteA})
```

Предикат $partition(S, x, y)$ является сокращенной формой записи утверждения, что некоторое множество S состоит из *непересекающихся* подмножеств x и y:

$$S = x \cup y$$
$$x \cap y = \varnothing$$

То есть вместо одной аксиомы с меткой @AccessesPartition можно было бы написать следующие две аксиомы, которые были бы ей эквивалентны (где {ReadA} — множество из одного элемента):

```
axioms
  @AccessesType1
    Accesses = { ReadA}∪{WriteA}
  @AccessesType2
    {ReadA}∩{WriteA} = ∅
```

Множество AccessRights соответствует множеству видов доступа R_r. Элементы этого множества — это право доступа на чтение Read ($read_r$), право доступа на запись Write ($write_r$), право доступа на выполнение Execute ($execute_r$), право доступа владения Own (own_r):

```
sets
  AccessRights
constants
  Read
  Write
  Execute
  Own
axioms
  @AccessRightsPartition
    partition(AccessRights, {Read},{Write},{Execute},{Own})
```

Константа Root соответствует корневой сущности-контейнеру $ROOT$ модели. Данная константа является элементом общего множества Union:

```
constants
  Root
axioms
  @RootType
    Root ∈ Union
```

Константа SRoot не имеет прямого аналога в модели. Она соответствует корневой субъект-сессии, т. е. субъект-сессии, которая является первой (или самой верхней) субъект-сессией в иерархии:

```
constants
  SRoot
axioms
  @SRootType
    SRoot ∈ Union
```

Константа SpecialAdmRoles соответствует множеству специальных административных ролей SAR. Элементами данного множества являются константы EntitiesAR (административная роль *entities_admin_role*), SubjectsAR (*subjects_admin_role*), UsersAR (*users_ admin_role*), RolesAR (*roles_ admin_role*), ARolesAR (*admin_ roles_admin_role*).

```
constants
  SpecialAdmRoles
  EntitiesAR
  SubjectsAR
  UsersAR
  RolesAR
  ARolesAR
axioms
  @SpecialAdmRolesType
    SpecialAdmRoles ⊆ Union
  @SpecialAdmRolesAreFinite
    finite(SpecialAdmRoles)
  @SpecialAdmRolesContent
    partition(SpecialAdmRoles, {EntitiesAR}, {SubjectsAR},
      {UsersAR}, {RolesAR}, {ARolesAR})
```

Хотя SpecialAdmRoles и является множеством, в спецификации оно объявлено как константа, так как множества (определяемые в блоке sets контекста) в Event-B по определению непересекающиеся и могут рассматриваться как аналоги типов. SpecialAdmRoles не является новым типом — далее можно будет увидеть, что оно является подмножеством множества административных ролей, которые в свою очередь являются подмножеством множества ролей, а множество ролей — подмножеством общего типа Union.

Константа CommonRole описывает общую роль учетных записей пользователей *COMMON_ROLE* модели:

```
constants
  CommonRole
axioms
```

@CommonRoleType
CommonRole ∈ Union

Также в контексте присутствует одна из аксиом Пеано для натуральных чисел, которая используется в спецификации при доказательстве по индукции:

axioms
@InductionAxiom
$\forall s \cdot s \subseteq \mathbb{N} \land 0 \in s \land (\forall n \cdot n \in s \Rightarrow n + 1 \in s) \Rightarrow \mathbb{N} \subseteq s$

4.4. Машина

На этом разбор контекста спецификации завершен. Рассмотрим теперь динамическую часть спецификации, которая в Event-B называется машиной, а именно определенные в ней переменные, инварианты и события.

4.4.1. Переменные и инварианты

Большинство переменных спецификации прямо соответствуют переменным, определенным в МРОСЛ ДП-модели, но кроме них в машине также присутствуют дополнительные переменные, заданные для удобства выражения некоторых свойств. Одной из подобных дополнительных переменных является CurrUnion, которая объединяет все текущие учетные записи пользователей (переменная UserAccs), субъект-сессии (Subjects), сущности (Entities) и роли (Roles) в единое общее множество, которое, в свою очередь, имеет тип Union. Данная переменная позволяет использовать в спецификации множество всех еще не созданных элементов Union\CurrUnion. В качестве примера, немного забегая вперед, отметим, что данное множество используется в событии по созданию нового объекта create_object в качестве его типа.

Каждой переменной обязательно соответствует один инвариант, в котором задается ее тип. Прочие инварианты, в тексте которых используется переменная, как правило, описывают ее свойства. Инварианты должны быть выполнены в каждом состоянии системы, и это является одним из основных свойств спецификации, которое требуется доказать на этапе ее верификации. В данном случае инвариант с меткой @CurrUnionType задает тип переменной CurrUnion (подмножество определенного в контексте множества Union), а инвариант с меткой @CurrUnionPartition — ее свойство (множество CurrUnion состоит из четырех непересекающихся подмножеств UserAccs, Subjects, Entities, Roles). При этом второй ин-

вариант также является инвариантом, задающим тип для этих четырех переменных:

```
variables
  CurrUnion
  UserAccs // U: множество учетных записей пользователей
  Subjects // S: множество субъект-сессий учетных записей
  пользователей
  Entities // E: множество сущностей
  Roles // Общее множество для обычных и административных ролей,
  не имеет прямого аналога в модели
invariants
  @CurrUnionType
    CurrUnion ⊆ Union
  @CurrUnionPartition
    partition(CurrUnion, UserAccs, Subjects, Entities, Roles)
```

В виде комментариев рядом с именами переменных в секции variables приведены соответствующие им сущности из МРОСЛ ДП-модели.

Множество сущностей Entities состоит из сущностей-объектов (переменная Objects) и сущностей-контейнеров (Containers). Аналогичным образом разделены роли: имеется общее множество ролей Roles, которое состоит из обычных (OrdRoles) и административных ролей (AdmRoles):

```
variables
  Objects // O: множество объектов
  Containers // C: множество контейнеров
  OrdRoles // R: множество ролей
  AdmRoles // AR: множество административных ролей
invariants
  @EntitiesPartition
    partition(Entities, Objects, Containers)
  @RolesPartition
    partition(Roles, AdmRoles, OrdRoles)
```

В соответствии с текстом МРОСЛ ДП-модели множества учетных записей пользователей и субъект-сессий должны быть конечными (это учтено еще в контексте в аксиоме с меткой @UnionIsFinite) и непустыми:

```
invariants
  @UserAccsAreNotEmpty
    UserAccs ≠ ∅
  @SubjectsAreNotEmpty
    Subjects ≠ ∅
```

Функция Direct ставит в соответствие каждой сущности и роли ее метку (@DirectType), которая может быть прямой (значение TRUE) или косвенной (FALSE).

Для упрощения описания свойств функции Direct для каждой сущности определяется вспомогательная функция EntityMP, ставящая им в соответствие сущность-контейнер, которая называется точкой монтирования (@EntityMPType). Если метка сущности прямая, то ее точкой монтирования является корневой каталог (@Direct1), иначе — сущность-контейнер с прямой меткой (@Direct2), причем точка монтирования должна находиться выше в иерархии сущностей по отношению к ней (@Direct6).

Если метка сущности-контейнера косвенная, то у всех сущностей, содержащихся в этой сущности-контейнере, должна быть косвенная метка (@Direct4). Если у некоторой сущности-контейнера mp метка прямая, а у одной из сущностей, содержащейся в mp, косвенная, то у всех сущностей, содержащихся в mp, метка косвенная (@Direct5), при этом mp является точкой монтирования для всех этих сущностей.

Метка корневого каталога прямая (@Direct7).

Если у роли имеется право доступа к сущности с косвенной меткой, то у нее должно быть такое же право доступа и к точке монтирования этой сущности, и наоборот (@Direct8, @Direct9).

Точкой монтирования сущности с косвенной меткой является либо точка монтирования ее родителя (@Direct10), либо сам родитель (@Direct11).

При этом метка каждой роли является прямой (@Direct12).

```
variables
  Direct // direct: функция, задающая метку сущности - прямая
    или косвенная
  EntityMP // вспомогательная функция, которая для каждой сущности
    с косвенной меткой ставит в соответствие точку монтирования
    (mount point)
invariants
  @DirectType
    Direct ∈ Entities ∪ Roles → BOOL
  @EntityMPType
    EntityMP ∈ Entities → Containers
  @Direct1
    ∀e · e ∈ Entities ∧ Direct(e) = TRUE ⇒ EntityMP(e) = Root
  @Direct2
    ∀e·e ∈ Entities ∧ Direct(e) = FALSE ⇒ Direct(EntityMP(e)) = TRUE
  @Direct3
```

$\forall c \cdot c \in Containers \wedge Direct(c) = FALSE \Rightarrow c \notin ran(EntityMP)$

@Direct4

$\forall e, p \cdot e \in dom(EntityNames) \wedge p \in dom(EntityNames(e))$
$\wedge\ Direct(p) = FALSE$
$\Rightarrow Direct(e) = FALSE$

@Direct5

$\forall e, mp \cdot e \in dom(EntityNames) \wedge mp \in dom(EntityNames(e))$
$\wedge\ Direct(mp) = TRUE \wedge Direct(e) = FALSE$
$\Rightarrow (\forall\ child \cdot child \in dom(EntityNames)$
$\wedge\ mp \in dom(EntityNames(child))$
$\Rightarrow Direct(child) = FALSE)$

@Direct6

$\forall e, p \cdot e \in dom(EntityNames) \wedge p \in dom(EntityNames(e))$
$\wedge\ Direct(e) = FALSE$
$\Rightarrow (\exists\ E \cdot E \subseteq Containers \wedge Root \notin E \wedge Parent[E] \cup \{p\} =$
$E \cup \{Root\} \wedge EntityMP(e) \in E \cup \{Root\})$

@Direct7

$Direct(Root) = TRUE$

@Direct8

$\forall e, a, r \cdot e \in Entities \wedge Direct(e) = FALSE \wedge r \in Roles$
$\wedge\ a \in AccessRights \wedge e \mapsto a \in RoleRights(r)$
$\Rightarrow EntityMP(e) \mapsto a \in RoleRights(r)$

@Direct9

$\forall e, a, r \cdot e \in Entities \wedge Direct(e) = FALSE \wedge r \in Roles$
$\wedge\ a \in AccessRights \wedge EntityMP(e) \mapsto a \in RoleRights(r)$
$\Rightarrow e \mapsto a \in RoleRights(r)$

@Direct10

$\forall e, p \cdot e \in dom(EntityNames) \wedge p \in dom(EntityNames(e))$
$\wedge\ Direct(e) = FALSE \wedge Direct(p) = FALSE$
$\Rightarrow EntityMP(e) = EntityMP(p)$

@Direct11

$\forall e, p \cdot e \in dom(EntityNames) \wedge p \in dom(EntityNames(e))$
$\wedge\ Direct(e) = FALSE \wedge Direct(p) = TRUE$
$\Rightarrow EntityMP(e) = p$

@Direct12

$\forall r \cdot r \in Roles \Rightarrow Direct(r) = TRUE$

Функция EntityNames определена для всех сущностей, за исключением корневого каталога, и ставит им в соответствие множество пар вида сущность-контейнер — имя, под которым сущность хранится в данной сущности-контейнере. Множество необходимо для поддержки механизма жестких ссылок: сущности-объекты могут храниться одновременно в нескольких сущностях-контейнерах или же в одной сущности-контейнере с разными именами.

Функция Parent определена для всех сущностей-контейнеров, за исключением корневого каталога, и ставит им в соответствие другую сущность-контейнер, которая находится непосредственно выше в иерархии сущностей, т. е. родительский каталог.

EntityNames должна удовлетворять следующим свойствам, выраженным в виде инвариантов:

- множество не может быть пустым, так как только у корневого каталога нет соответствующего родительского каталога (инвариант с меткой @EntityNames1);
- множество, соответствующее сущностям-контейнерам, должно состоять только из одной пары, так как жесткие ссылки на сущности-контейнеры не поддерживаются (@EntityNames2);
- две разные сущности не могут храниться в одной и той же сущности-контейнере с одинаковым именем (@EntityNames3).

Инварианты с метками @EntityNames4 и @EntityNames5 устанавливают связь между функциями EntityNames и Parent. Без функции Parent вполне можно было бы обойтись, так как вся хранящаяся в ней информация об иерархии сущностей-контейнеров также присутствует и в функции EntityNames, однако функция Parent проще и с ее помощью многие свойства выражаются легче.

Так, функцию Parent удобно использовать для выражения свойства отсутствия циклов в иерархии сущностей: если бы циклы присутствовали, то существовало бы такое множество сущностей-контейнеров, что разность этого множества и множества родительских каталогов этих сущностей была бы пустым множеством. Свойство отсутствия циклов является отрицанием данного утверждения и выражается в инварианте (@NoCyclesForContainers).

```
variables
  EntityNames // entity_name: функция имен сущностей в составе
  сущностей-контейнеров
  Parent
invariants
  @EntityNamesType
    EntityNames ∈ Entities \ {Root} → (Containers ↔ Names)
  @ParentType
    Parent ∈ Containers \ {Root} → Containers
  @EntityNames1
    ∀e · e ∈ dom(EntityNames) ⇒ EntityNames(e) ≠ ∅
  @EntityNames2
    ∀c · c ∈ Containers ∧ c ≠ Root
      ⇒ (∃p, n · p ∈ Containers ∧ n ∈ Names ∧ EntityNames(c) =
        {p ↦ n})
```

```
  @EntityNames3
    ∀e1, e2 · e1 ∈ dom(EntityNames) ∧ e2 ∈ dom(EntityNames) ∧ e1 ≠ e2
      ⇒ EntityNames(e1) ∩ EntityNames(e2) = ∅
  @EntityNames4
    ∀c1, c2 · c1 ∈ Containers ∧ c1 ≠ Root ∧ c2 ∈ dom(EntityNames(c1))
      ⇒ c2 = Parent(c1)
  @EntityNames5
    ∀c1, c2 · c1 ∈ Containers ∧ c1 ≠ Root ∧ c2 = Parent(c1)
      ⇒ c2 ∈ dom(EntityNames(c1))
  @NoCyclesForContainers
    ∀ C · C ⊆ Containers ∧ C ≠ ∅∧ Root ∉ C ⇒ C \ Parent[C] ≠ ∅
```

Функция RoleAdmRights ставит в соответствие каждой административной роли множество пар вида роль — право доступа, которое имеет административная роль к данной роли. Данная функция имеет следующие свойства, выраженные в виде инвариантов:

- каждая административная роль имеет право доступа на выполнение к любой существующей роли (@ExecuteToEverything);
- специальная административная роль RolesAR является владельцем для каждой обычной роли (@RolesAR1), причем она является единственным владельцем (@RolesAR2);
- специальная административная роль ARolesAR является владельцем для каждой административной роли (@ARolesAR1), причем она является единственным владельцем (@ARolesAR2);
- если административная роль обладает правом доступа на чтение к роли, то она обладает правом доступа на чтение и ко всем ролям ниже в ее иерархии (@ReadSpreads).

```
variables
  RoleAdmRights // APA: функция административных прав доступа
  к ролям и административным ролям административных ролей
invariants
  @RoleAdmRightsType
    RoleAdmRights ∈ AdmRoles → (Roles ↔ AccessRights)
  @ExecuteToEverything
    ∀ ar, r · ar ∈ AdmRoles ∧ r ∈ Roles
      ⇒ r ↦ Execute ∈ RoleAdmRights(ar)
  @RolesAR1
    ∀ r · r ∈ OrdRoles
      ⇒ r ↦ Own ∈ RoleAdmRights(RolesAR)
  @RolesAR2
    ∀ r, ar · r ∈ OrdRoles ∧ ar ∈ AdmRoles
      ∧ r ↦ Own ∈ RoleAdmRights(ar)
        ⇒ ar = RolesAR
  @ARolesAR1
```

```
    ∀ r · r ∈ AdmRoles
      ⇒ r ↦ Own ∈ RoleAdmRights(ARolesAR)
  @ARolesAR2
    ∀ r, ar · r ∈ AdmRoles ∧ ar ∈ AdmRoles
      ∧ r ↦ Own ∈ RoleAdmRights(ar)
        ⇒ ar = ARolesAR
  @ReadSpreads
    ∀ ar, r, p · ar ∈ AdmRoles ∧ r ∈ Roles
      ∧ p ∈ Roles ∧ p ∈ RParents(r)
      ∧ p ↦ Read ∈ RoleAdmRights(ar)
        ⇒ r ↦ Read ∈ RoleAdmRights(ar)
```

Функция RoleName определена для всех ролей и ставит им в соответствие их имя. Данное имя должно быть уникально: не должно существовать двух различных ролей с одинаковым именем, так что в Event-B данная функция определена как тотальная инъекция (символ $\mapsto$). Совместно с функцией RParents функция RoleName реализует функцию МРОСЛ ДП-модели role_name: функцию имен ролей и административных ролей в составе роли или административной роли.

```
variables
  RoleName
invariants
  @RoleNameType
    RoleName ∈ Roles Names
```

Функция RoleRights ставит в соответствие для каждой роли множество пар вида сущность — право доступа, которое имеет роль к данной сущности. С переменной RoleRights связано следующее свойство: у каждой сущности может быть только одна роль-владелец:

```
variables
  RoleRights // PA: функция прав доступа к сущностям ролей
    и административных ролей
invariants
  @RoleRightsType
    RoleRights ∈ Roles → (Entities ↔ AccessRights)
  @NoMultipleOwners
    ∀ r1, r2, e · r1 ∈ Roles ∧ r2 ∈ Roles ∧ e ↦ Own ∈ RoleRights(r1)
      ∧ e ↦ Own ∈ RoleRights(r2)
        ⇒ r1 = r2
```

Константа Root, соответствующая корневому каталогу, является элементом множества текущих сущностей-контейнеров:

```
invariants
```

```
  @RootType
    Root ∈ Containers
```

Функция RParents определена для всех ролей и ставит им в соответствие множество родительских ролей, в которых они непосредственно содержатся. Множество необходимо по той причине, что одна роль может одновременно содержаться в нескольких других ролях. Не у всех ролей есть родительская роль, так что поставленное в соответствие множество может быть пустым. С переменной RParents связаны следующие инварианты:

- иерархии обычный ролей и административных ролей независимы (@RParents1 и @RParents2);
- в иерархии ролей нет циклов (@NoCyclesForRoles).

```
variables
  RParents
invariants
  @RParentsType
    RParents ∈ Roles → P(Roles)
  @RParents1
    ∀ r · r ∈ AdmRoles ⇒ RParents(r) ⊆ AdmRoles
  @RParents2
    ∀ r · r ∈ OrdRoles ⇒ RParents(r) ⊆ OrdRoles
  @NoCyclesForRoles
    ∀ R · R ⊆ Roles ∧ R ≠ ∅ ⇒ (∃ r · r ∈ R ∧ (∀ p · p ∈ R
      ⇒ p ∉ RParents(r)))
```

Функция Shared определена на множествах сущностей-контейнеров и субъект-сессий и задает, являются ли элементы этих множеств разделяемыми контейнерами. Если бы множества Containers и Roles не были бы подмножествами общего множества-типа Union, определенного в контексте, то Event-B не позволил бы использовать в записи инварианта операцию объединения этих множеств. Каждая роль должна являться разделяемым контейнером:

```
variables
Shared // shared_container: функция разделяемых записей контейнеров
invariants
  @SharedType
    Shared ∈ Containers ∪ Roles → BOOL
  @RolesAreShared
    ∀ r · r ∈ Roles ⇒ Shared(r) = TRUE
```

Функция SParent определена для всех субъект-сессий, за исключением корневой сессии, и ставит им в соответствие другую

субъект-сессию, которая находится непосредственно выше в иерархии субъект-сессий. Также на иерархии субъект-сессий должны отсутствовать циклы:

```
variables
  SParent
invariants
  @SParentType
    SParent ∈ Subjects \ {SRoot} → Subjects
  @NoCyclesForSubjects
    ∀ S · S ⊆ dom(SParent) ∧ S ≠ ∅ → S \ SParent[S] ≠ ∅
```

Определенное в контексте в виде константы множество SpecialAdmRoles должно быть подмножеством множества административных ролей:

```
invariants
  @SpecialAdmRolesType
    SpecialAdmRoles ⊆ AdmRoles
```

Константа SRoot, в свою очередь, является элементом множества текущих субъект-сессий:

```
invariants
  @SRootType
    SRoot ∈ Subjects
```

Функция SubjectAccesses описывает доступы субъект-сессий к сущностям: для каждой субъект-сессии ставится в соответствие множество упорядоченных пар вида сущность — доступ, который субъект-сессия имеет к данной сущности:

```
variables
  SubjectAccesses // A: функция доступов субъект-сессий к сущностям
invariants
  @SubjectAccessesType
    SubjectAccesses ∈ Subjects → (Entities ↔ Accesses)
```

Функция SubjectAdmAccesses описывает доступы субъект-сессий к ролям: для каждой субъект-сессии ставится в соответствие множество упорядоченных пар вида роль — доступ, который субъект-сессия имеет к данной роли:

```
variables
  SubjectAdmAccesses // AA: функция доступов субъект-сессий к ролям
или административным ролям
invariants
  @SubjectAdmAccessesType
    SubjectAdmAccesses ∈ Subjects → (Roles ↔ Accesses)
```

Частичная функция SubjectOwner ставит в соответствие для некоторых субъект-сессий роль, которая имеет право доступа владения к данной сессии. Функция определена не для всех субъект-сессий, так как не у каждой субъект-сессии есть владелец:

```
variables
  SubjectOwner // PA: частный случай функции PA, только для сессий и
правда доступа владения
invariants
  @SubjectOwnerType
    SubjectOwner ∈ Subjects ↦ Roles
```

Функция SubjectUser ставит в соответствие каждой субъект-сессии учетную запись пользователя, от имени которой она действует:

```
variables
  SubjectUser // user: функция принадлежности субъект-сессии
    учетной записи пользователя
invariants
  @SubjectUserType
    SubjectUser ∈ Subjects → UserAccs
```

Функции UserAdmRole и UserOrdRole определены для каждой учетной записи пользователя и ставят им в соответствие индивидуальную административную роль и индивидуальную обычную роль соответственно (@UserAdmRoleType, @UserOrdRoleType). Индивидуальные роли пользователя должны содержаться в иерархии ролей независимо, т. е. у них не должно быть ни родителей, ни потомков (@UserAdmRole1, @UserAdmRole2, @UserOrdRole1, @UserOrdRole2). Одна роль не может быть индивидуальной ролью сразу нескольких учетных записей пользователей (@UserAdmRole3, @UserOrdRole3). Индивидуальная административная роль пользователя не должна быть специальной административной ролью (@UserAdmRole4). Кроме того, индивидуальная административная роль учетной записи пользователя должна обладать правами доступа на чтение и на запись как к самой себе, так и к индивидуальной обычной роли учетной записи пользователя (@UserAdmRole5, @UserAdmRole6, @UserOrdRole4, @UserOrdRole5).

```
variables
  UserAdmRole // u_admin: индивидуальная административная роль
    учетной записи пользователя
  UserOrdRole // u_c: индивидуальная роль учетной записи
      пользователя
invariants
```

```
  @UserAdmRoleType
    UserAdmRole ∈ UserAccs → AdmRoles
  @UserOrdRoleType
    UserOrdRole ∈ UserAccs → OrdRoles
  @UserAdmRole1
    ∀ u · u ∈ UserAccs ⇒ RParents(UserAdmRole(u)) = ∅
  @UserAdmRole2
    ∀ u, r · u ∈ UserAccs ∧ r ∈ Roles ⇒ UserAdmRole(u) ∉ RParents(r)
  @UserAdmRole3
    ∀ u1, u2 · u1 ∈ UserAccs ∧ u2 ∈ UserAccs ∧ u1 ≠ u2
      ⇒ UserAdmRole(u1) ≠ UserAdmRole(u2)
  @UserAdmRole4
    ∀ u · u ∈ UserAccs ⇒ UserAdmRole(u) ∉ SpecialAdmRoles
  @UserAdmRole5
    ∀ u · u ∈ UserAccs
      ⇒ UserAdmRole(u) ↦ Read ∈ RoleAdmRights(UserAdmRole(u))
  @UserAdmRole6
    ∀ u · u ∈ UserAccs
      ⇒ UserAdmRole(u) ↦ Write ∈ RoleAdmRights(UserAdmRole(u))
  @UserOrdRole1
    ∀ u · u ∈ UserAccs ⇒ RParents(UserOrdRole(u)) = ∅
  @UserOrdRole2
    ∀ u, r · u ∈ UserAccs ∧ r ∈ Roles ⇒ UserOrdRole(u) ∉ RParents(r)
  @UserOrdRole3
    ∀ u1, u2 · u1 ∈ UserAccs ∧ u2 ∈ UserAccs ∧ u1 ≠ u2
      ⇒ UserOrdRole(u1) ≠ UserOrdRole(u2)
  @UserOrdRole4
    ∀ u · u ∈ UserAccs
      ⇒ UserOrdRole(u) ↦ Read ∈ RoleAdmRights(UserAdmRole(u))
  @UserOrdRole5
    ∀ u · u ∈ UserAccs
      ⇒ UserOrdRole(u) ↦ Write ∈ RoleAdmRights(UserAdmRole(u))
```

Общая роль CommonRole является обычной ролью (@CommonRoleType). Общая роль находится в иерархии ролей независимо от прочих ролей (@CommonRole1, @CommonRole2). Общая роль не может быть индивидуальной ролью какой-либо учетной записи пользователя (@CommonRole3). Кроме того, индивидуальная административная роль каждой учетной записи пользователя должна обладать правами доступа на чтение и на запись к общей роли CommonRole (@CommonRole4, @CommonRole5).

```
invariants
  @CommonRoleType
    CommonRole ∈ OrdRoles
```

@CommonRole1
  RParents(CommonRole) = ∅
@CommonRole2
  ∀ r · r ∈ Roles ⇒ CommonRole ∉ RParents(r)
@CommonRole3
  ∀ u · u ∈ UserAccs ⇒ CommonRole ≠ UserOrdRole(u)
@CommonRole4
  ∀ u · u ∈ UserAccs
    ⇒ CommonRole ↦ Read ∈ RoleAdmRights(UserAdmRole(u))
@CommonRole5
  ∀ u · u ∈ UserAccs
    ⇒ CommonRole ↦ Write ∈ RoleAdmRights(UserAdmRole(u))

4.4.2. Событие инициализации

В Event-B достижимыми называются состояния, в которых система может оказаться при переходах из некоторого начального состояния в результате выполнения произвольных последовательностей описанных в спецификации событий. Начальное состояние при этом задается событием инициализации. В спецификации базового уровня МРОСЛ ДП-модели событие инициализации на данный момент не реализовано. Поэтому для подтверждения корректности спецификации вместо доказательства сохранности инвариантов в каждом достижимом состоянии проводится доказательство сохранности инвариантов при каждом переходе от одного возможного состояния к другому возможному, где под возможным состоянием подразумевается любое состояние, при котором определенные в спецификации инварианты были бы выполнены.

4.4.3. Реализация правила перехода системы из состояния в состояние в виде события

Рассмотрим пример реализации одного из правил перехода системы из состояния в состояние МРОСЛ ДП-модели в виде Event-B события — правила *delete_subject* (табл. 4.1).

Правило delete_subject обладает двумя параметрами (x и z), пронумерованным множеством предусловий (левая часть таблицы) и пронумерованным множеством постусловий (правая часть таблицы). События Event-B устроены схожим образом: событие может иметь параметры, блок охранных условий (аналог предусловий) и блок действий, в котором осуществляется изменение значений определенных в спецификации переменных.

Второе предусловие требует отсутствия у удаляемой субъект-сессии z потомков в иерархии субъект-сессий (в противном случае

Таблица 4.1

Правило перехода системы из состояния в состояние *delete_subject*

$\boldsymbol{delete\_subject(x, z)}$	
1) $x, z \in S$; 2) $H_S(z) = \varnothing$; 3) существует $r \in R \cup AR$: $(x, r, \mathit{read}_a) \in AA$; $(z, \mathit{own}_r) \in PA(r)$	1) $S' = S \setminus \{z\}$; 2) $AA' = AA \setminus \{(z, r, \alpha_a)$: $r \in R \cup AR$, $\alpha_a \in R_a\}$; 3) $A' = A \setminus \{(z, e, \alpha_a)$: $e \in E$, $\alpha_a \in R_a\}$; 4) для $r \in R \cup AR$ выполняется $PA'(r) = PA(r) \setminus \{(z, \mathit{own}_r)\}$; 5) для $z' \in S$ такой, что $z \in H_S(z')$, справедливо равенство $H'_S(z') = H_S(z') \setminus \{z\}$

удаление надо начинать с них). На Event-B спецификации иерархия субъект-сессий реализована обратной функцией SParent, так что соответствующее условие @grd4 требует, чтобы удаляемая субъект-сессия delSubject не являлась родителем ни для какой другой субъект-сессии.

Третье предусловие требует наличия у субъект-сессии x доступа на чтение к роли с правом доступа владения к удаляемой субъект-сессии z. На Event-B спецификации права доступа владения к субъект-сессиям реализуются отдельной частичной функцией Subject-Owner. Так как функция частичная, то требование существования роли-владельца сводится к требованию принадлежности субъект-сессии к ее множеству определения (охранное условие @grd5). Требование наличия доступа на чтение выражается в условии @grd6.

В правиле перехода системы из состояния в состояние *delete_subject* также присутствует пять постусловий, которые описывают удаление всякого упоминания субъект-сессии z из переменных состояния. Первому постусловию соответствуют действия события с метками @act1 и @act2 (все элементы множества Subjects также являются элементами общего множества CurrUnion, которого нет в МРОСЛ ДП-модели). Второе и третье постусловия описывают удаление всех доступов, которые имела удаляемая субъект-сессия к ролям и сущностям — @act4 и @act6. Удаление информации о роли-владельце из четвертого постусловия осуществляется в действии @act5. Обновление иерархии субъект-сессий из пятого постусловия — в @act6.

Необъясненным осталось только охранное условие с меткой @grd3, которое требует, чтобы удаляемая субъект-сессия не была корневой субъект-сессией. Понятие корневой субъект-сессии отсутствует в МРОСЛ ДП-модели, оно было добавлено в Event-B спецификацию исключительно для облегчения выражения некоторых связанных с иерархией субъект-сессий свойств.

```
event delete_subject
  any subject delSubject
  where
    @grd1 subject ∈ Subjects
    @grd2 delSubject ∈ Subjects
    @grd3 delSubject ≠ SRoot
    @grd4 ∀ s · s ∈ dom(SParent) ⇒ SParent(s) ≠ delSubject
    @grd5 delSubject ∈ dom(SubjectOwner)
    @grd6 SubjectOwner(delSubject) ↦ ReadA ∈
      SubjectAdmAccesses(subject)
  then
    @act1 CurrUnion := CurrUnion \{delSubject}
    @act2 Subjects := Subjects \{delSubject}
    @act3 SubjectUser := {delSubject} ⩤ SubjectUser
    @act4 SubjectAccesses := {delSubject} ⩤ SubjectAccesses
    @act5 SubjectOwner := {delSubject} ⩤ SubjectOwner
    @act6 SubjectAdmAccesses := {delSubject} ⩤ SubjectAdmAccesses
    @act7 SParent := {delSubject} ⩤ SParent
end
```

4.4.4. Использование вспомогательных параметров

Предусловия и постусловия правила перехода системы из состояния в состояние МРОСЛ ДП-модели *delete_subject* достаточно просты и транслируются на Event-B без заметных изменений. Однако ситуация обстоит несколько иначе с некоторыми другими правилами. Рассмотрим пример правила перехода системы из состояния в состояние *create_object* (табл. 4.2).

Таблица 4.2

Правило перехода системы из состояния в состояние *create_object*

$\boldsymbol{create\_object(x, y, yd, name, z)}$	
1) $x \in S$; 2) $y \notin E$; 3) $z \in C$; 4) [$(x, z, write_a) \in A$, существует $r \in R \cup AR$ такая, что $(x, r, read_a) \in AA$ и $(z, execute_r) \in PA(r)$]; 5) $name \in NAME \setminus (\{<<>>\} \cup \cup \{entity\_name(z, y')$: $y' \in H_E(z)\})$; 6) $(x, user(x)\_c, write_a) \in AA$; 7) $H_E(z) = \{y' \in H_E(z)$: $direct(y') = = yd\}$; 8) [если $yd = true$, то $direct(z) = true$]	1) $E' = E \cup \{y\}$ $(O' = O \cup \{y\}$, $C' = C)$; 2) $entity\_name'(z, y) = \{name\}$; 3) $direct'(y) = yd$; 4) если $yd = true$, то $PA'(user(x)\_c) = = PA(user(x)\_c) \cup \{(y, own_r)\}$, если $yd = false$ и $c \in C$ такая, что $direct(c) = true$, $H_E(c) = \{y' \in H_E(y)$: $direct(y') = false\}$ и $y < c$, то для всех $r' \in R \cup AR$ выполняется $PA'(r') = = PA(r') \cup \{(y, \alpha_{rj})$: $(c, \alpha_{rj}) \in PA(r')\}$; 5) $H'_E(z) = H_E(z) \cup \{y\}$, $H'_E(y) = \varnothing$

Постусловие номер 4 правила *create_object* описывает изменение функции PA. Данное описание включает в себя большое число различных условий и достаточно сложно, чтобы его можно было выразить в виде действий Event-B события, которые в основном выглядят как простые присваивания. Существует два способа решения данной проблемы. Первый — использование нотации задания множества вида $\{x \mid P\}$, с помощью которой можно задать множество, состоящее из всех элементов x, удовлетворяющих предикату P. Пример использования — присваивание переменной S множества, состоящего из всех таких элементов x, которые либо принадлежат множеству A, либо множеству B:

$$S := \{x \mid x \in A \vee x \in B\}.$$

В данном конкретном случае можно было вполне обойтись без нотации задания множества и напрямую присвоить переменной S объединение множеств A и B:

$$S := A \cup B.$$

В случае с четвертым постусловием правила *create_object* подобной простой альтернативы нет, а предикат P будет заметно сложней: на самом деле он станет настолько сложным, что в нем будет весьма затруднительно разобраться, и особенно заметно усложнится процесс доказательства сохранности инвариантов, в которых присутствует измененная переменная.

Второй способ заключается в объявлении вспомогательного параметра события, значение которого присваивается нужной переменной в виде действия Event-B события. При этом в виде охранных условий на значение объявленного параметра описываются свойства, которым должно соответствовать новое значение переменной, а также связь между текущим и новым значениями. В данном способе единый предикат P, о котором шла речь выше, разбивается на отдельные охранные условия, работать с которыми становится проще.

Второй способ хорошо себя зарекомендовал и широко применяется в Event-B спецификации МРОСЛ ДП-модели. Рассмотрим его на примере параметра roleRights события *create_object*, который был добавлен для описания постусловия номер 4 правила *create_object*.

Охранное условие с меткой @grd17 задает тип параметра. Так как значение параметра впоследствии будет присвоено переменной RoleRights в действии @act5, то задаваемый тип должен соответствовать типу переменной. Тип переменной RoleRights — функция, ставящая в соответствие для каждой роли множество пар вида сущность — право доступа. Так как в результате выполнения события

create_object множество сущностей Entities изменится — в него добавится только что созданный объект, а множество сущностей Entities присутствует в описании типа переменной RoleRights, то в описании типа параметра roleRights нужно учесть данное изменение: вместо множества Entities использовать его новое значение, а именно объединение множества Entities и создаваемого объекта object. Все это учтено в задающем тип параметра охранном условии @grd17.

Все изменения переменной RoleRights, описанные в постусловии номер 4, касаются добавления некоторым ролям прав доступа к создаваемому объекту object. Данные изменения выражаются в виде охранных условий @grd19-21 на параметр roleRights.

Права доступа ко всем сущностям, за исключением объекта object, имеющиеся у ролей на момент до события create_object, должны остаться неизменны после присваивания переменной RoleRights значения параметра roleRights. Данная связь описывается в охранном условии @grd18.

В действии @act5 происходит присваивание переменной RoleRights значения параметра roleRights, который был подробно описан в охранных условиях.

```
event create_object
  any subject object parent name role dLabel mountPoint roleRights
depth
  where
    @grd1 object ∈ Union ∖ CurrUnion
    @grd2 subject ∈ Subjects
    @grd3 parent ∈ Containers
    @grd4 parent ↦ WriteA ∈ SubjectAccesses(subject)
    @grd5 ∃ r · r ∈ Roles ∧ r ↦ ReadA ∈ SubjectAdmAccesses(subject)
      ∧ parent ↦ Execute ∈ RoleRights(r)
    @grd6 name ∈ Names
    @grd7 ∀ e · e ∈ dom(EntityNames)
      ⇒ parent ↦ name ∉ EntityNames(e)
    @grd8 role = UserOrdRole(SubjectUser(subject))
    @grd9 role ↦ WriteA ∈ SubjectAdmAccesses(subject)
    @grd10 mountPoint ∈ Containers
    @grd11 dLabel ∈ BOOL
    @grd12 ∀ e · e ∈ dom(EntityNames) ∧ parent ∈ dom(EntityNames(e))
      ⇒ Direct(e) = dLabel
    @grd13 dLabel = TRUE ⇒ Direct(parent) = TRUE
    @grd14 dLabel = TRUE ⇒ mountPoint = Root
    @grd15 dLabel = FALSE ∧ Direct(parent) = FALSE
      ⇒ mountPoint = EntityMP(parent)
    @grd16 dLabel = FALSE ∧ Direct(parent) = TRUE
```

```
      ⇒ mountPoint = parent
    @grd17 roleRights ∈ Roles → (Entities ∪{object} ↔
      AccessRights)
    @grd18 ∀ e, a, r · e ∈ Entities ∧ a ∈ AccessRights ∧ r ∈ Roles
      ⇒ (e ↦ a ∈ roleRights(r) ↔ e ↦ a ∈ RoleRights(r))
    @grd19 dLabel = TRUE ⇒ object ↦ Own ∈ roleRights(role)
    @grd20 dLabel = TRUE
      ⇒ (∀ a, r · a ∈ AccessRights ∧ r ∈ Roles
        ∧ object ↦ a ∈ roleRights(r)
          ⇒ a = Own ∧ r = role)
    @grd21 dLabel = FALSE
      ⇒ (∀ a, r · a ∈ AccessRights ∧ r ∈ Roles
        ⇒ (mountPoint ↦ a ∈ RoleRights(r)
          ⇔ object ↦ a ∈ roleRights(r)))
    @grd22 depth ∈ ℕ → ℙ(Containers)
    @grd23 ∀ c · c ∈ Containers ⇒ (∃ i · i ∈ ℕ ∧ c ∈ depth(i))
    @grd24 depth(0) = {Root}
    @grd25 ∀ i · i ∈ ℕ ∧ i ≠ 0 ⇒ (∀ c · c ∈ depth(i) ⇒ c ≠ Root)
    @grd26 ∀ i · i ∈ ℕ
      ⇒ (∀ c · c ∈ depth(i + 1)
        ⇒ (∃ p · p ∈ depth(i) ∧ p = Parent(c)))
    theorem @grd27
      ∀ i · i ∈ ℕ ∧(∀ c · c ∈ depth(i)
        ⇒ (∃ E · E ⊆ Containers ∧ Root ∉ E ∧ Parent[E] ∪ {c} =
        E ∪ {Root}))
          ⇒ (∀ c · c ∈ depth(i + 1)
            ⇒ (∃ E · E ⊆ Containers ∧ Root ∉ E ∧ Parent[E] ∪ {c} =
            E ∪ {Root}))
    theorem @grd28
      ∀ i · i ∈ ℕ ⇒ (∀ c · c ∈ depth(i)
        ⇒ (∃ E · E ⊆ Containers ∧ Root ∉ E ∧ Parent[E] ∪ {c} = E ∪
{Root}))
  then
    @act1 CurrUnion := CurrUnion ∪ {object}
    @act2 Entities := Entities ∪ {object}
    @act3 Objects := Objects ∪ {object}
    @act4 EntityNames(object) := {parent ↦ name}
    @act5 RoleRights := roleRights
    @act6 Direct(object) := dLabel
    @act7 EntityMP(object) := mountPoint
end
```

4.4.5. Использование математической индукции

При доказательстве сохранности инвариантов иногда требуется использовать пятую аксиому Пеано натуральных чисел, а именно аксиому индукции: если какое-либо предложение доказано для 0 (база индукции) и если из допущения, что оно верно для натурального числа n, вытекает, что оно верно для следующего за n натурального числа (индукционное предположение), то это предложение верно для всех натуральных чисел. Рассмотрим использование доказательства по индукции в Event-B на примере события *create_object*.

Для доказательства сохранности одного из инвариантов для события *create_object* потребовалось использовать следующее свойство: для каждой сущности-контейнера существует путь до корневого каталога. На Event-B данное свойство можно выразить так: для каждой сущности-контейнера c существует множество сущностей-контейнеров E (которое и задает требуемый путь), не включающее в себя корневой каталог, такое, что множество непосредственных родителей в иерархии сущностей для всех сущностей-контейнеров из множества E в объединении с сущностью-контейнером c равно объединению множества E и корневого каталога *Root*:

$$\forall\, \mathtt{c} \cdot \mathtt{c} \in \mathtt{Containers} \Rightarrow (\exists\, \mathtt{E} \cdot \mathtt{E} \subseteq \mathtt{Containers} \land \mathtt{Root} \notin \mathtt{E} \land \mathtt{Parent[E]} \cup \{\mathtt{c}\} = \mathtt{E} \cup \{\mathtt{Root}\})$$

Рассмотрим данное свойство с помощью рис. 4.3 на примере пути от сущности-контейнера c до корневого каталога. Тогда множество E, которое будет соответствовать данному пути, будет состоять из элементов $\{x, p, c\}$. Множество непосредственных родителей в иерархии сущностей для всех сущностей-контейнеров из множест-

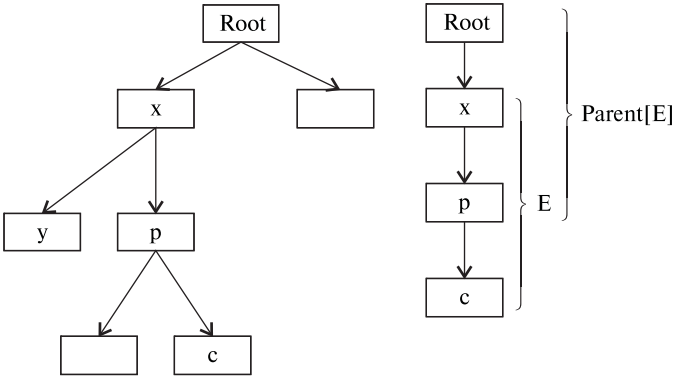

Рис. 4.3. Путь до корневого каталога от сущности-контейнера c

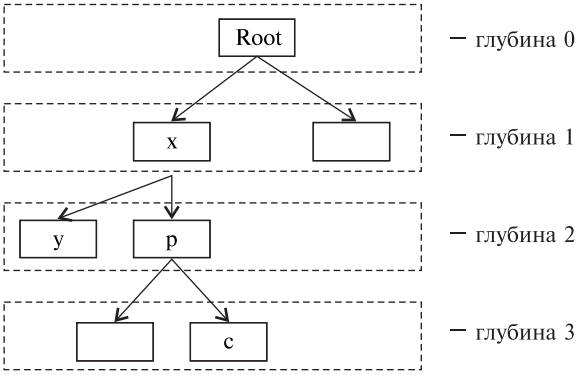

Рис. 4.4. Глубина сущностей-контейнеров в иерархии сущностей

ва E, выражаемое в виде $Parent(E)$, можно вычислить следующим образом:

$$Parent[E] = Parent(x) \cup Parent(p) \cup Parent(c) = \{Root\} \cup \{x\} \cup \{p\}$$

Выбранное нами множество E действительно удовлетворяет свойству

$$Parent[E] \cup \{c\} = \{x, p, c\} \cup \{Root\},$$

так как

$$\{Root, x, p\} \cup \{c\} = \{x, p, c\} \cup \{Root\}.$$

Однако если бы мы выбрали для сущности-контейнера c другое множество E, например включающее в себя сущность-контейнер y, то данное свойство уже бы не выполнялось, что означало бы, что данное множество не является отображением пути до корневого каталога.

Чтобы использовать данное свойство, его нужно сначала доказать. Для этого можно использовать математическую индукцию: поставим в соответствие для каждой сущности-контейнера ее глубину (наибольшую длину пути от корневого каталога до данной сущности-контейнера) (рис. 4.4). Тогда базе индукции будет соответствовать сам корневой каталог, а свойство существования пути до корневого каталога будет доказано следующим образом: пусть $c = Root$, тогда нужно найти такое E, что

$$\exists E \cdot E \subseteq Containers \wedge Root \notin E \wedge Parent[E] \cup \{Root\} = E \cup \{Root\}$$

Таким E является $\varnothing$. Проверим это:

$$\varnothing \subseteq Containers \wedge Root \notin \varnothing \wedge Parent[\varnothing] \cup \{Root\} = \varnothing \cup \{Root\}$$

Первые два конъюнкта очевидно истинны. Так как $Parent[\varnothing] = \varnothing$, то третий конъюнкт на самом деле является тождеством. База индукции доказана.

Далее следует доказать индукционное предположение: при условии существовании пути до корневого каталога от любой сущности-контейнера, расположенной на глубине i, доказать, что существует путь до корневого каталога любой сущности-контейнера, расположенной на глубине $i + 1$. Это достаточно просто: выберем случайную сущность-контейнер, расположенную на глубине $i + 1$, и назовем ее c. По определению используемой нами функции глубины, у каждой сущности на глубине $i + 1$ существует родительская сущность контейнер, расположенная на глубине i, — назовем ее p. Из первой части индукционного предположения известно, что для любой сущности-контейнера на глубине i существует путь до корневого каталога, назовем этот путь E. Тогда требуемый для доказательства путь до корневого каталога от сущности-контейнера для глубине $i + 1$ будет $E \cup \{p\}$.

Теперь перенесем данные рассуждения на событие *create_object* на Event-B. Объявим функцию глубины как дополнительный параметр события depth. В охранном условии @grd22 зададим тип данного параметра: функция, ставящая для каждого натурального числа (собственно глубины) множество сущностей-контейнеров, которые находятся на данной глубине. Последующие охранные условия описывают функцию глубины более полным образом:

- атрибут глубины должен быть у каждой сущности-контейнера (@grd23);
- на нулевой глубине должен находиться только корневой каталог (@grd24);
- корневой каталог не может находиться на глубине отличной от 0 (@grd25);
- для любой сущности-контейнера на глубине $i + 1$ существует сущность-контейнер на глубине i, которая является его родителем (@grd26).

Используя функцию глубины, можно задать индукционное предположение в виде теоремы @grd27: если существует путь до корневого каталога от любой сущности-контейнера на глубине i, то существует и для любой сущности-контейнера на глубине $i+1$. Данную теорему нужно доказать (мы это уже сделали неформальным образом выше), а затем совместно с базой индукции @grd24 использовать для доказательства исходного свойства существования пути до корневого каталога для любой сущности-контейнера, которое мы

выразили в виде теоремы @grd28. Аналогичным образом выполняется доказательство по индукции еще для нескольких событий в спецификации МРОСЛ ДП-модели.

4.4.6. Разделение правила перехода системы из состояния в состояние на несколько событий

Следующее, о чем целесообразно упомянуть в контексте различий между МРОСЛ ДП-моделью и ее спецификацией на Event-B, это разделение некоторых правил перехода системы из состояния в состояние на два и более события. Данное действие требуется обычно в случаях, когда правило описывает логически общее действие — например, получение доступа на чтение — но направленное на разные объекты — например, доступ на чтение к сущности или роли. Рассмотрим правило перехода системы из состояния в состояние *access_read* (табл. 4.3).

Таблица 4.3

Правило перехода системы из состояния в состояние access_read

$access\_read(x, y)$	
1) $x \in S$; 2) $y \in E \cup R \cup AR$; 3) существует $r \in R \cup AR$: $(x, r, read_a) \in AA$, [если $y \in E$, то $(y, read_r) \in PA(r)$ и существует контейнер $c \in C$ такой, что $execute\_container(x, c, y) = true$], [если $y \in R \cup AR$, то $(y, read_r) \in APA(r)$]	1) если $y \in E$, то $A' = A \cup \cup\{(x, y, read_a)\}$, $AA' = AA$; 2) если $y \in R \cup AR$, то $AA' = AA \cup \{(x, y, read_a)\}$, $A' = A$

Параметр y может быть как сущностью, так и ролью, и в зависимости от этого различаются пред- и постусловия правила. В принципе, это не препятствует прямому их переносу на Event-B, однако итоговое событие получится неоправданно усложненным. Вместо этого предлагается в данном случае разбить одно событие на два независимых, каждое из которых будет описывать отдельный частный случай — получение доступа к сущности (событие *access_read_entity*) и получение доступа к роли (событие *access_read_role*):

```
event access_read_entity
  any subject entity
  where
    @grd1 subject ∈ Subjects
    @grd2 entity ∈ Entities
    @grd3 ∃ r · r ∈ Roles ∧ r ↦ ReadA ∈ SubjectAdmAccesses(subject)
      ∧ entity ↦ Read ∈ RoleRights(r)
    @grd4 ∃ E, c · E ⊆ Containers ∧ Root ∉ E
      ∧ ((entity ∈ dom(EntityNames) ∧ c ∈ dom(EntityNames(entity))
```

```
        ∧ Parent[E] ∪ {c} = E ∪ {Root}) ∨ (E = ∅∧ entity = Root))
      ∧ (∀ o · o ∈ E ∪{entity} ∪ {Root} ⇒ (∃ r · r ∈ Roles
        ∧ r ↦ ReadA ∈ SubjectAdmAccesses(subject)
        ∧ o ↦ Execute ∈ RoleRights(r)))
  then
    @act1 SubjectAccesses(subject) := SubjectAccesses(subject)
        ∪ {entity ↦ ReadA}
end
event access_read_role
  any subject role
  where
    @grd1 subject ∈ Subjects
    @grd2 role ∈ Roles
    @grd3 ∃ r · r ∈ AdmRoles ∧ role ↦ ReadA ∈ RoleAdmRights(r)
      ∧ r ↦ Read ∈ SubjectAdmAccesses(subject)
  then
    @act1 SubjectAdmAccesses(subject) := SubjectAdmAccesses
      (subject) ∪ {role ↦ ReadA}
end
```

4.4.8. Отсутствующие в спецификации элементы МРОСЛ ДП-модели

В тексте МРОСЛ ДП-модели приводится определение функции значений сущностей, ролей и административных ролей V, задающей возможность для каждой из них принимать значение любой конечной последовательности бит. Значение данной функции изменяется исключительно в следующих правилах перехода системы из состояния в состояние: get_user_attr, read_container, get_entity_attr, get_subject_attr, get_role_attr. Приведенные правила не изменяют значений никаких других функций и множеств, определенных в модели, и в модели не содержится никаких условий и требований, которым должна удовлетворять функция V. Как следствие, данная функция и связанные с ней правила перехода системы из состояния в состояние не были реализованы в Event-B спецификации, так как на данный момент они малополезны с точки зрения верификации спецификации.

4.4.9. Формализация свойств безопасности МРОСЛ ДП-модели

В представленной выше спецификации отсутствуют инварианты, соответствующие содержательным свойствам безопасности. Например, основное правило управления доступом, использующего роли, могло бы быть представлено так: если субъект-сессия имеет

доступ на чтение или запись к какой-либо сущности, значит она владеет ролью (имеет к ней доступ на чтение), дающей право на выполнение такого же вида действий (соответственно, чтения или записи) над этой сущностью. В виде инварианта такое правило для доступа на чтение могло бы выглядеть следующим образом.

```
invariants
  @AccessesAreOnlyThroughRoles
    ∀ e, s · e ∈ Entities ∧ s ∈ Subjects ∧ e ↦ ReadA ∈
    SubjectAccesses(s)
      ⇒ (∃ r · r ∈ Roles ∧ r ↦ ReadA ∈ SubjectAdmAccesses(s)
        ∧ e ↦ Read ∈ RoleRights(r))
```

Однако можно заметить, что такой инвариант не может быть выполнен в представленной спецификации, поскольку в ней есть события, изменяющие как права доступа ролей, так и доступы сессий к ролям. Сессия, получившая доступ к какой-либо сущности благодаря владению ролью, имеющей на это право, может сохранять этот доступ даже тогда, когда эта роль будет модифицирована и потеряет соответствующее право, или сессия потеряет доступ к этой роли. Таким образом, в описанной спецификации ограничения, связанные с предоставлением доступа только в соответствии с правами ролей, гарантированно выполняются лишь в сам момент получения доступа — в охранных условиях событий получения доступа эти правила проверяются, и доступ предоставляется согласно имеющимся у сессии ролям, однако последующие изменения прав ролей не отслеживаются и не влияют на уже полученные доступы.

Проверка ограничений доступа лишь в момент его получения может быть недостаточна при описании более строгих ограничений, например ограничений мандатного управления доступом. В спецификациях, формализующих такие ограничения, крайне желательно иметь соответствующие им инварианты, обеспечивающие выполнение ограничений во всех достижимых состояниях. При этом трудности, связанные с возможными изменениями уровней доступа (аналогичные продемонстрированным выше в связи с изменениями прав ролей), могут быть разрешены за счет ограничений на такие изменения в охранных условиях событий или разделением режимов работы системы на обычный (в рамках которого никаких изменений уровней доступа не происходит) и административный (в котором их разрешено менять, соответственно, инварианты безопасности могут нарушаться при таких изменениях, однако должны быть выполнены при переходе в обычный режим).

4.5. Верификация

Комплекс инструментальных средств AstraVer Toolset [45] включает в себя среду для разработки и верификации спецификаций на Event-B, основанную на платформе Rodin [44]. Спецификации на Event-B разрабатываются и верифицируются с помощью платформы Rodin. Для каждого требующего доказательства свойства спецификации Rodin автоматически генерирует соответствующие утверждения для доказательства, причем платформа решает проблему поддержки актуальности сгенерированных утверждений и выполненных доказательств в случае изменений в спецификации. Полное доказательство спецификации означает, что доказаны все сгенерированные утверждения.

Для спецификации базового уровня МРОСЛ ДП-модели было сгенерировано 831 утверждение для доказательства, часть из которых была доказана полностью автоматически, а оставшаяся часть — интерактивным образом, при котором доказательство сначала упрощалось и разбивалось на составные части вручную, а уже затем подавалось на вход инструментам автоматического доказательства теорем.

Рассмотрим следующее условие МРОСЛ ДП-модели: для каждой роли существует единственная административная роль *roles_admin_role*, обладающая к ней правом доступа владения, для каждой административной роли существует единственная административная роль *admin_roles_admin_role*, обладающая к ней правом доступа владения. Данное условие выполняется в результате выполнения каждого правила перехода системы из состояния в состояние, так как право доступа владения к ролям или административным ролям дается только административным ролям *roles_admin_role* или *admin_roles_admin_role* в результате применения правила вида *create_user*(x, x', u) и *create_role*$(x, r, name, rz)$, и нет правил, позволяющих изменить такую роль-владельца роли или административной роли.

Так выглядит математическое доказательство. Рассмотрим теперь тоже самое условие, однако выраженное в виде инвариантов на Event-B:

```
invariants
  @RolesAR1
    ∀ r · r ∈ OrdRoles ⇒ r ↦ Own ∈ RoleAdmRights(RolesAR)
  @ARolesAR1
    ∀ r · r ∈ AdmRoles ⇒ r ↦ Own ∈ RoleAdmRights(ARolesAR)
```

Для каждого изменения входящих в эти инварианты переменных в результате выполнения какого-либо события, а именно переменных OrdRoles, AdmRoles, RoleAdmRights, система верификации сгенерирует утверждения для доказательства. В данном случае утверждения будут сгенерированы для событий *create_user, delete_user, create_role, create_hard_link_role, delete_role, grant_admin_rights, remove_admin_rights*, так как это полный список событий, в которых каким-либо образом изменяются интересующие нас переменные. Рассмотрим данные события:

- в событиях *create_user* и *create_role* происходит создание новых ролей (изменение множеств AdmRoles и OrdRoles), а также изменение функции доступов RoleAdmRights. Инварианты сохраняются, так как в условиях событий явно прописано добавление доступа владения к создаваемым ролям ARolesAR и RolesAR;
- в событиях *delete_user* и *delete_role* удаляются индивидуальные роли пользователя, а также происходит удаление связанных с ними доступов в функции RoleAdmRights. Это может нарушить инварианты в случае, когда удаляемой ролью становится роль ARolesAR или RolesAR. Однако эта ситуация исключена в предусловиях данных событий и инвариантах;
- в событиях *create_hard_link_role*, *grant_admin_rights*, *remove_admin_rights* изменяется функция доступов RoleAdmRights, однако в ходе доказательства можно показать, что данные изменения никак не затрагивают права доступа владения.

Следует напомнить, что доказательство на Event-B на самом деле проводится с использованием автоматических и интерактивных инструментов, которые следят за его корректностью и не позволяют допускать ошибки, а данный пример является его примерным словесным описанием.

В итоге получается, что доказательство на Event-B в целом соответствует математическому доказательству (выполняемому «вручную» аналитиком), однако является более достоверным, что иногда позволяет находить несоответствия и нестыковки в описании специфицируемой системы.

4.6. Использование уточнения

В настоящее время МРОСЛ ДП-модель имеет иерархическое представление, состоящее из серии уровней. Нижний уровень модели — базовый уровень, который используется в качестве примера в данной работе, — не зависит от новых элементов, принадлежащих более высоким уровням модели. В свою очередь, все остальные

уровни модели наследуют элементы предыдущих уровней, при необходимости корректируя, а также дополняя их новыми элементами.

Event-B позволяет разрабатывать спецификации, отражающие подобную иерархическую структуру. Для этого предлагается использовать технику пошагового уточнения [46, 47]. Вместо создания единой монолитной спецификации, которая будет содержать в себе все детали моделируемой системы, уточнение предлагает разрабатывать серию связанных между собой спецификаций. В такой серии первая спецификация представляет собой некоторую базовую версию системы, которая содержит в себе только основные детали. Дополнительные детали системы шаг за шагом добавляются в остальных спецификациях серии таким образом, что каждая последующая спецификация в серии является уточнением предыдущих. Подробней с техникой уточнения можно ознакомиться в Приложениях A и B.

Как правило, на каждом уровне уточнения добавляются новые переменные, новые события, которые изменяют значения новых переменных, а также модифицируются старые события, например вводятся дополнительные охранные условия, учитывающие значения новых переменных. При этом уточнение позволяет использовать определенные ранее (на предыдущих уровнях уточнения) переменные, но не позволяет изменять их значения. Запрет на изменение старых переменных позволяет быть уверенным, что определенные и доказанные на прошлых уровнях инварианты будут выполняться и на всех последующих. Кроме этого, новые детали системы не должны противоречить старым.

Данные свойства уточнения входят в некоторое противоречие с иерархической структурой МРОСЛ ДП-модели, которая позволяет более высоким уровням при необходимости корректировать элементы предыдущих уровней. Так, например, на базовом уровне в правиле перехода системы из состояния в состояние *create_user* создаются ровно две индивидуальные роли пользователя. На уровне, в котором реализуется мандатное управление доступом, число создаваемых индивидуальных ролей будет другим: оно будет зависеть от мандатной метки конфиденциальности создаваемой учетной записи пользователя. На Event-B же, если на базовом уровне спецификации сказано, что в событии создается только две роли, то и на всех последующих уровнях в данном событии будут создаваться ровно две роли.

Такое явное противоречие (с точки зрения уточнения и Event-B) просто не позволяет использовать уточнение для добавления но-

вых уровней к приведенной в данной главе Event-B спецификации базового уровня МРОСЛ ДП-модели. Для того чтобы это было возможно, подобные противоречия следует предварительно устранить. В приведенном выше примере для устранения противоречия между двумя уровнями в событии *create_user* спецификации базового уровня следует отметить, что создаваемых ролей будет несколько, но отдельно описать свойства только двух из них. При уточнении свойства остальных создаваемых индивидуальных ролей будут добавлены в спецификацию в виде дополнительных охранных условий соответствующих событий на уровне мандатного управления доступом.

Рассмотренный метод удовлетворяет требованиям компонента доверия ADV_SPM.1 «Формальная модель политики безопасности» наличия формальной модели политики безопасности (спецификация на формализованном языке Event-B) и ее верификации (формальное доказательство невозможности перехода спецификации в небезопасное состояние). В следующей главе будет предложен подход к формальному доказательству соответствия между функциональной спецификацией объекта оценки и формальной моделью политики безопасности.

5 Формальная функциональная спецификация

В данной главе рассматривается второй этап процесса моделирования и верификации механизма управления доступом операционной системы, а именно разработка и верификация формальной функциональной спецификации ОО. Требования к этой работе сформулированы в компонентах доверия ADV_FSP.6 «Полная полуформальная функциональная спецификация с дополнительной формальной спецификацией» и ADV_SPM.1 «Формальная модель политики безопасности ОО»:

- ADV_FSP.6.9C. В формальном представлении функциональной спецификации ФБО должно быть изложено формальное описание интерфейса ФБО, дополненное, где это необходимо, неформальным пояснительным текстом;
- ADV_SPM.1.3C. Соответствие между моделью политики безопасности и функциональной спецификацией должно быть представлено на соответствующем уровне формализации;
- ADV_SPM.1.4C. В соответствии между моделью политики безопасности объекта оценки (ПБО) и функциональной спецификацией должно быть продемонстрировано, что функциональная спецификация является непротиворечивой и полной относительно модели ПБО.

В данных требованиях под интерфейсом функций безопасности понимается совокупность всех способов, через которые пользователи могут получить доступ к защищенным ресурсам системы. В случае ядра ОС интерфейсом, доступным приложениям, а через них и пользователям ОС, являются системные вызовы. Напрямую пользователь не взаимодействует со средствами защиты информации, так как они функционируют внутри ядра ОС. Пользователь может обнаружить эффект их действия только в ходе выполнения некоторых системных вызовов, которые исходят от исполняемого приложения пользователя. Например, если приложение пользователя попытается открыть на запись файл, к которому в соответствии с заданной политикой безопасности ему разрешен доступ только на чтение, оно получит код ответа, сообщающий, что при попытке открытия файла произошла ошибка, и файл открыть не удалось.

То есть пользователь может наблюдать поведение системы защиты информации, но его взаимодействие с системой защиты происходит опосредованно.

Мы можем заключить, что в нашем случае интерфейсом ОО является интерфейс системных вызовов и именно для этого интерфейса должна быть представлена формальная функциональная спецификация, которую затем необходимо исследовать на предмет того, является ли она полной и непротиворечивой по отношению к модели политики безопасности управления доступом.

Начнем с задачи разработки формальной функциональной спецификации системных вызовов. В идеальном случае можно было бы ожидать, что разработчики ОС сами предоставляют полную и строгую спецификацию интерфейса разрабатываемого программного продукта. Заметим, что здесь речь идет не только о строгом описании структуры (синтаксиса) интерфейсных сущностей, но и об их функциональности, т. е. об описании наблюдаемого извне поведения ОС в ходе взаимодействия с ней приложений посредством этих интерфейсных сущностей.

К сожалению, в практике промышленного программирования разработка строгих или формальных спецификаций выполняется крайне редко (хотя заметим, что разработчики микропроцессоров всегда создают как минимум строгие спецификации системы команд процессора). В случае ОС принципиальных технических трудностей для разработки спецификации системных вызовов нет. Полная формальная спецификация программного интерфейса ОС реального времени была разработана в 1994–1997 годах [48], в 2005–2010 годах были разработаны спецификации для ОС Windows [49] и для базового набора системных библиотек ОС Linux [50]. Все три проекта использовали близкие по идеологии техники спецификации, которые сводились к разработке программных контрактов в форме пред- и постусловий и/или описания ожидаемого поведения функций в форме некоторой исполнимой модели (в роли модели могла быть программа на языке спецификации или на некотором диалекте C или C#). Природа моделей, которые использовались в этих проектах, относится к классу так называемых моделей на основе состояний (state-based models). Такие модели состоят из набора операций (функций, событий), входом к которым служат некоторые данные (аргументы, фактические параметры), а выходом некоторый результат операции (значение функции). Также операции имеют доступ к некоторым глобальным структурам данных (состоянию). Результат операции зависит от состояния, в котором она вызывается. Кроме

того, изменение данных состояния может быть частью результата, в таком случае говорят, в результате выполнения операции система перешла из одного состояния в другое.

По сути при разработке формальной спецификации МРОСЛ ДП-модели на Event-B мы использовали близкий подход. Таким же образом технически возможно с использованием уже известного нам инструментария разработать и формальную функциональную спецификацию системных вызовов ядра ОС Linux.

Далее после задачи разработки формальной функциональной спецификации стоит задача демонстрации (а еще лучше, доказательства) соответствия между ней и спецификацией модели политики безопасности. Здесь и далее под свойством «соответствия» мы будем понимать выполнение требований к ОО, о которых говорилось выше, — требований полноты и непротиворечивости функциональной спецификации ОО по отношению к модели политики безопасности.

Техникам доказательства соответствия между формальными спецификациями некоторой программной системы, которые отличаются друг от друга степенью детализации, посвящено достаточно много исследований. Техника пошагового уточнения [46] является одной из наиболее известных и широко используемых, в том или ином виде она поддерживается большинством формальных методов, включая Event-B. В рамках этого подхода рекомендуется вести разработку поэтапно, начиная с базового уровня спецификации, содержащей максимально простую и компактную модель системы, и затем постепенно добавлять новые детали в последующих уровнях, например, путем ввода новых операций или уточнения структур данных. В ходе каждого этапа проводится две группы доказательств корректности. Первая группа нацелена на доказательство корректности каждого уровня спецификации отдельно, самого по себе. Вторая группа доказательств должна подтвердить, что каждый новый уточненный уровень полон и не входит в противоречие с предыдущим, более абстрактным уровнем, то есть все свойства системы (и вся функциональность), описанные на более абстрактном уровне, сохранены и на более детальном уровне. Общая идеология данного подхода сводится к тому, что, если на каждом этапе разработки объем «приращения» функциональности небольшой и поддается формальной верификации, и, кроме того, доказано соответствие между уровнями спецификации, то можно гарантировать, что система в целом корректна и в ней выполнены все требования, которые были сформулированы по ходу проектирования.

Таким образом, для доказательства соответствия между двумя спецификациями некоторой системы — между абстрактной спецификацией M1 и детальной спецификацией M2 (рис. 5.1) — достаточно показать, что между ними существует такое отношение уточнения R, что для каждого перехода между состояниями из спецификации M2, который ведет из некоторого состояния s в состояние s', существует соответствующий переход в абстрактной спецификации M1 из абстрактного состояния σ в состояние σ'. При этом переход называется соответствующим, если выполняется соотношения $R(s) = \sigma$ и $R(s') = \sigma'$.

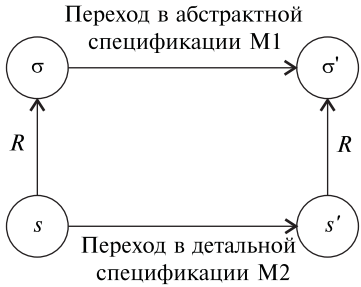

Рис. 5.1. Диаграмма отношения уточнения между абстрактной и детальной спецификациями

В общем случае доказывать требование соответствия для двух разных спецификаций одной системы достаточно трудно. По этой причине в методологиях и технологиях, которые развивали данный подход, вводились достаточно жесткие ограничения на то, как описываются новые уровни уточнения, т. е. правила детализации проектирования системы жестко ограничивались, чтобы предоставить возможность формальной проверки сохранения соответствия между более абстрактными и более детальными слоями проекта. Например, в методологии RAISE [51] нельзя было переопределять структуры данных, т. е. для каждого типа данных требовалось дать сразу полное и окончательное его описание, что для сложных и тем более развивающихся систем является очень неприятным ограничением. В Event-B нельзя изменять значения переменных, определенных ранее (на предыдущих уровнях уточнения). Хотя благодаря этому ограничению можно быть уверенным, что определенные и доказанные на прошлых уровнях инварианты будут выполняться и на всех последующих, оно существенно усложняет процесс планирования уровней уточнения при разработке спецификации.

Возвращаясь к доказательству соответствия между моделью политики безопасности и формальной функциональной спецификацией, мы должны показать существование отношения уточнения R между их состояниями и операциями, причем на уровне формальной функциональной спецификации за переход между состояниями отвечают системные вызовы, а на уровне модели политики безопасности — де-юре правила МРОСЛ ДП-модели. Однако оказывается, что в нашем случае построение соответствия является весьма

непростой задачей. Если набор структур данных и пространство состояний формальной функциональной спецификации и можно рассматривать как расширение структур данных и пространства состояния МРОСЛ ДП-модели, то операции формальной функциональной спецификации — системные вызовы — не удается прямо отобразить на правила МРОСЛ ДП-модели. Далее на примере показывается, как можно решить данную проблему и построить соответствие в нашем случае.

5.1. Построение соответствия между системными вызовами и правилами МРОСЛ ДП-модели

Если подробно описать контекст, в котором вызывается тот или иной системный вызов, т. е. описать текущее состояние ОС и значение параметров вызова, то можно свести его выполнение к последовательности (цепочке) определенных действий и проверок. Часть из этих действий будет прямо соответствовать правилам перехода системы из состояния в состояние МРОСЛ ДП-модели, а для прочих можно показать, что они не влияют на выполнимость установленных в ней требований.

Рассмотрим системный вызов open. Этот системный вызов реализуется функцией int open(const char *pathname, int flags), которая выполняет действия, необходимые для открытия файла. У функции open два параметра: строка pathname содержит путь до открываемого файла, а flags определяет вид доступа, который требуется получить — на чтение, на запись или на чтение и на запись одновременно. За это отвечают флаги O_RDONLY, O_WRONLY, O_RDWR. flags также может содержать дополнительные, необязательные флаги. Функция возвращает ассоциированный с файлом файловый дескриптор, который затем может использоваться в последующих системных вызовах (например, в read, write, lseek, fcntl).

Рассмотрим частный случай вызова open, при котором файл из параметра pathname не существует, а параметр flags содержит флаг O_WRONLY, что означает открытие файла на запись, и дополнительный флаг O_CREAT, что позволит создать файл. Если процесс, от имени которого вызывается open, обладает всеми нужным правами доступа, то open должен выполнить следующую цепочку действий:

- разбор значений параметров;
- проверка наличия необходимых для создания файла прав доступа — в рассматриваемом частном случае проверка проходит успешно;

- получение доступа на запись к каталогу, в котором будет создан файл;
- создание файла;
- получение права доступа на запись к созданному файлу;
- получение доступов к созданному файлу;
- возвращение файлового дескриптора созданного и открытого на запись файла.

Каждое из этих действий определенным образом изменяет состояние формальной функциональной спецификации (ФСП). Например, действие «возврат файлового дескриптора» изменяет множество всех открытых данным процессом файловых дескрипторов. Можно заметить, что три действия — а именно действия «получение доступа на запись», «получение права доступа на запись» и «создание файла» — прямо соответствуют правилам перехода между состояниями МРОСЛ ДП-модели *access_write*, *grant_rights* и *create_object*. Если рассматривать состояние ФСП как расширение состояния МРОСЛ ДП-модели, то данные действия изменяют исключительно ту ее часть, которая является общей с состоянием МРОСЛ ДП-модели. Для доказательства соответствия и непротиворечивости достаточно показать, что данные действия не нарушают определенных в МРОСЛ ДП-модели инвариантов через доказательство наличия отношения уточнения между действиями системного вызова open и правилами МРОСЛ ДП-модели. Остальные действия (назовем их вспомогательными) изменяют исключительно переменные из расширенного, дополненного состояния, т. е. те переменные, аналогов которых нет в состоянии МРОСЛ ДП-модели, так что они никак не влияют на существование отношения уточнения. Для демонстрации непротиворечивости таких действий относительно МРОСЛ ДП-модели следует сформулировать инварианты, которые свяжут значения переменных дополненного состояния формальной функциональной спецификации и переменных общего с МРОСЛ ДП-моделью состояния, и затем показать их выполнимость. Например, в данном случае следует сформулировать инвариант, который свяжет файловый дескриптор, открытый на запись (переменная дополненного состояния), с полученным доступом на запись (переменная общего с МРОСЛ ДП-моделью состояния).

Конкретный путь выполнения, или конкретная последовательность действий, которая будет выполнена в результате вызова системного вызова open, зависит от контекста, в котором происходит вызов, а именно от текущего состояния ОС и значений параметров вызова. Если описать все возможные пути выполнения для

каждого системного вызова, то доказательство того, что ФСП является непротиворечивой относительно МРОСЛ ДП-модели, может быть достигнуто за счет доказательства выполнимости инвариантов МРОСЛ ДП-модели и дополнительных связующих инвариантов при любом выполняемом внутри системного вызова действии. А для доказательства того, что ФСП является полной относительно МРОСЛ ДП-модели, необходимо показать, что каждое правило перехода системы из состояния в состояние МРОСЛ ДП-модели связано хотя бы с одним действием функциональной спецификации.

Итак, мы показали, что в конкретном контексте системному вызову соответствует цепочка действий, которую можно отобразить на правила МРОСЛ ДП-модели. Подобные цепочки можно построить для каждого контекста и так для каждого системного вызова, тем самым решив проблему их отображения на правила МРОСЛ ДП-модели.

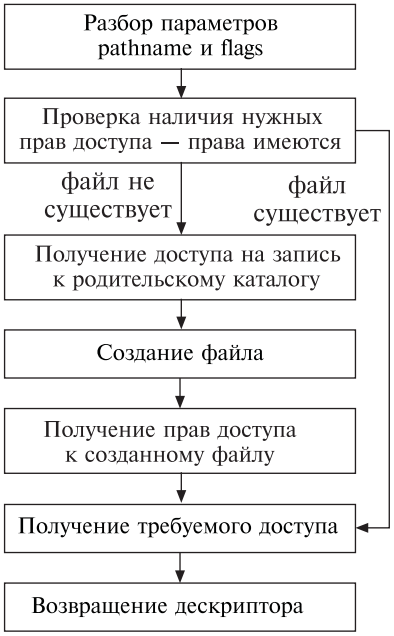

Рис. 5.2. Графовая модель небольшого подмножества цепочек действий системного вызова open

Возникает вопрос, как перебрать все необходимые контексты и как это сделать без чрезмерных затрат труда. Можно заметить, что большинство цепочек, относящихся к одному системному вызову, похожи друг на друга и отличаются только в некоторых деталях. Так, если вместо примера открытия несуществующего файла на запись, рассмотренного выше, мы бы рассмотрели открытие несуществующего файла и на чтение, и на запись, то большинство действий в обоих событиях были бы общими. Или, если бы мы рассмотрели пример открытия уже существующего файла на запись (или на чтение и запись, или просто на чтение), то соответствующая данному контексту цепочка оказалась бы просто сокращенной версией цепочки из примера.

Из-за подобного сходства цепочки, относящиеся к одному системному вызову, можно объединить друг с другом для получения графовой модели системного вызова. Пример подобной модели для

нескольких контекстов вызова open, рассмотренных выше, приводится на рис. 5.2. В данном примере есть две основных цепочки.

Графовое представление эквивалентно набору цепочек, т. е. оно так же решает проблему отображения системных вызовов на правила модели политики безопасности, однако оно заметно компактней и наглядней, а также проще в реализации. Формальным критерием консистентности данного представления для отдельного системного вызова является выполнимость требования, состоящего в том, что из текущего состояния ОС, параметров системного вызова, а также условий из всех предыдущих точек ветвления следует дизъюнкция условий каждой точки ветвления.

5.2. Пример формализации функциональной спецификации

МРОСЛ ДП-модель представлена на формализованном языке Event-B с использованием техники пошагового уточнения для отображения ее иерархической структуры. Мы будем использовать разновидность данной техники под названием *расширенное* уточнение и для формализации функциональной спецификации, что значительно упростит доказательство ее непротиворечивости и полноты. В расширенном уточнении благодаря представлению событий уточненной спецификации (формальной функциональной спецификации) в виде графовой модели можно провести соответствие между событиями уточняемой спецификации (Event-B спецификации МРОСЛ ДП-модели) и некоторыми узлами графов событий уточненной спецификации (ФСП). Рассмотрим данный метод на примере.

В соответствии с предлагаемым методом узлы графов системных вызовов, которые соответствуют правилам МРОСЛ ДП-модели (узел создания файла в open является правилом перехода системы из состояния в состояние create_object МРОСЛ ДП-модели), будут представлены в ФСП как уточнения описывающих эти правила Event-B событий. Оставшиеся вспомогательные узлы (такие как разбор параметров pathname и flags системного вызова open) будут представлены как вспомогательные события, изменяющие исключительно дополненное состояние ФСП, которое затем будет связано инвариантами с состоянием Event-B спецификации МРОСЛ ДП-модели. Используя средства языка Event-B, данные события затем будут объединены в виде цепочек, где цепочка является одним из возможных путей в представляющем системный вызов графе.

Покажем как это будет выглядеть на примере рассмотренного ранее частного случая обращения к системному вызову open, при котором файл с именем pathname не существует, параметр flags содержит флаги O_WRONLY и O_CREAT и у процесса, от имени которого вызывается open, есть все необходимые права доступа. Этому случаю в ФСП на Event-B будет соответствовать цепочка из восьми событий: *open_start* → *open_check_p* → *open_write_p* → *open_create* → *open_grant* → *open_check* → *open_write* → *open_finish*, где:

- в событии *open_start* находятся предусловия, описывающие разбор параметров системного вызова open и принимается решение, какое событие должно произойти следующим. В рассматриваемом частном случае открываемый файл не существует, так что следующим событием будет *open_check_p*. Если бы файл существовал, то следующим событием был бы *open_check* и в целом цепочка событий была бы другой;
- в событии *open_check_p* осуществляется проверка наличия требуемых прав доступа — в данном примере у вызывающего open процесса все нужные права доступа имеются;
- событие *open_write* является уточнением события *access_write_entity* Event-B спецификации МРОСЛ ДП-модели, которое описывает получение доступа на запись. В данном событии получается доступ на запись к каталогу, в котором будет создан файл;
- событие *open_create* является уточнением события *create_object* из Event-B спецификации МРОСЛ ДП-модели, которое описывает создание нового объекта (файла);
- событие *open_grant* является уточнением события *grant_rights* Event-B спецификации МРОСЛ ДП-модели, которое описывает получение прав доступа;
- в событии *open_check* осуществляется проверка наличия требуемых прав доступа и принимается решение, какое событие должно произойти следующим. В данном случае требуется получить доступ на запись к созданному файлу, следовательно, следующим событием будет *open_write*;
- событие *open_write* является уточнением события *access_write_entity* Event-B спецификации МРОСЛ ДП-модели, которое описывает получение доступа на запись;
- событие *open_finish* в своем постусловии возвращает запрашиваемый файловый дескриптор.

Данная цепочка событий соответствует частному случаю системного вызова open. Для демонстрации метода на рис. 5.3 приводится граф, соответствующий еще нескольким частным случаям.

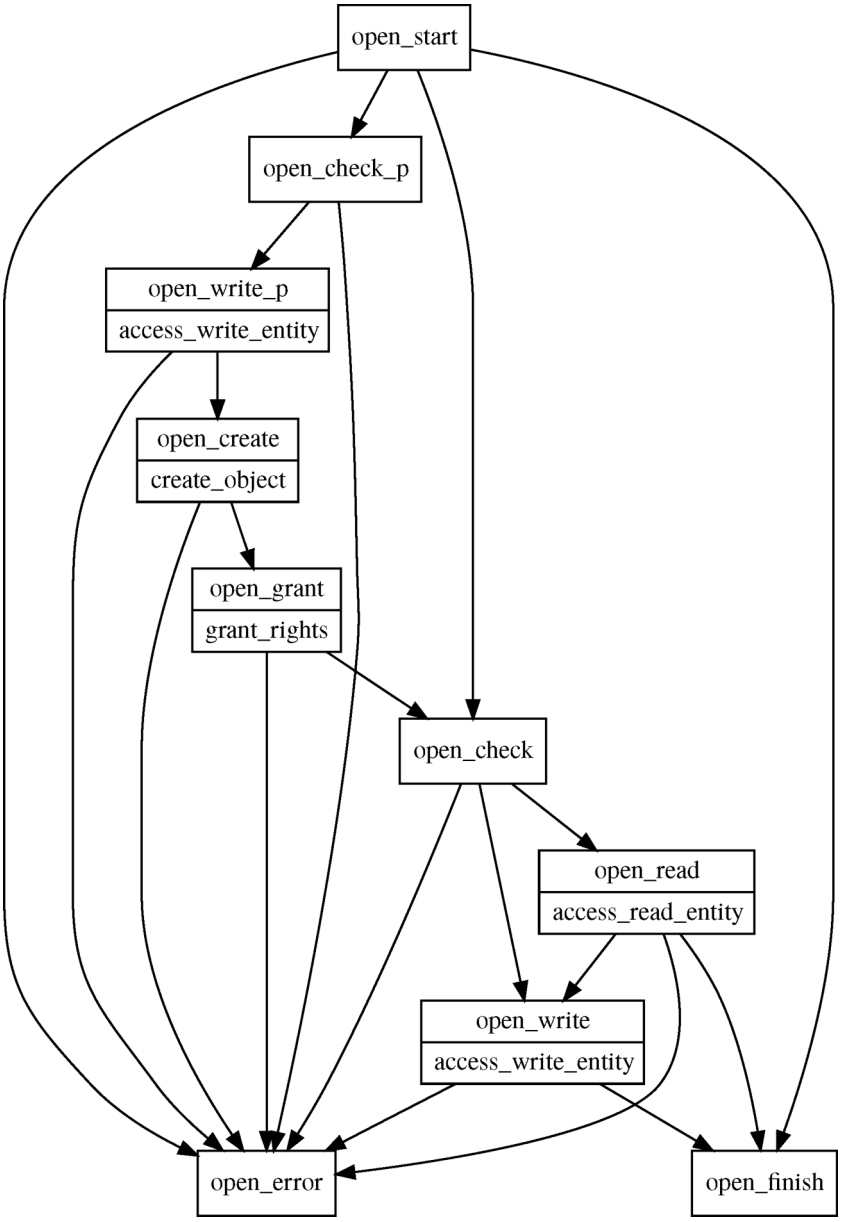

Рис. 5.3. Граф Event-B событий нескольких частных случаев системного вызова open

События ФСП, представляющие граф системного вызова open, изображены прямоугольниками; если событие вспомогательное, то в прямоугольнике показано только его имя, если же событие соответствует событию Event-B спецификации МРОСЛ ДП-модели, то в его прямоугольнике снизу показано имя уточняемого события.

Если формализовать все оставшиеся частные случаи, то полученный граф будет являться формальной спецификацией поведения системного вызова open. При этом благодаря использованию уточнения данная спецификация будет корректна по построению и полностью соответствовать правилам МРОСЛ ДП-модели, что в свою очередь будет означать, что при любом сочетании параметров и состояния системы выполнение системного вызова open сохранит все установленные в МРОСЛ ДП-модели требования и свойства. После этого останется только доказать выполнимость инвариантов, которые связывают дополненное состояние и состояние МРОСЛ ДП-модели. В результате формализации подобным образом всех системных вызовов будет получена формальная функциональная спецификация с доказанными свойствами непротиворечивости и полноты.

6 Дедуктивная верификация модуля безопасности ядра ОС Linux

Как уже отмечалось, модуль LSM является ключевым звеном механизма управления доступом ОССН в целом. По этой причине его верификация является одной из центральных задач верификации механизма управления доступом. Дедуктивная верификация — это наиболее точный метод статического анализа программ. Для его применения к некоторому программному модулю помимо соответствующих инструментов верификации требуется наличие формальных функциональных спецификаций этого модуля. В контексте данной работы нам важно иметь не только полные и корректные спецификации модуля LSM, данные спецификации должны соответствовать исходной модели политики безопасности управления доступом в ОССН. Как уже отмечалось в первых главах, автоматический переход от формальной спецификации модели политики безопасности (Event-B спецификации МРОСЛ ДП-модели) к спецификации модели модуля LSM невозможен. В связи с этим предлагается цепочка работ, которая в результате позволяет верифицировать модуль LSM и продемонстрировать его близость к требованиям спецификации исходной модели политики безопасности управления доступом. Эта цепочка выглядит следующим образом:

- на Event-B вручную создается дополнительная модель механизмов безопасности на уровне LSM, содержащая контракты операций интерфейса LSM и инварианты, описывающие гарантируемые свойства безопасности (помимо обычных инвариантов, описывающих корректность данных, используемых операциями LSM);
- соответствие между созданной Event-B спецификацией LSM и Event-B спецификацией МРОСЛ ДП-модели контролируется с помощью экспертизы — специалисты по безопасности могут проверить содержательное соответствие между инвариантами безопасности двух спецификаций (эти инварианты должны содержать формулировку по сути одинаковых ограничений с учетом различий в используемых структурах данных). Также проверяется наличие в Event-B спецификации LSM всех механизмов, описанных Event-B спецификации МРОСЛ ДП-модели

(быть может, за исключением ограничений, реализуемых с помощью стандартного для Unix-систем контроля доступа пользователей и групп пользователей), и отсутствие каких-либо значимых способов обхода контроля доступа в спецификации LSM по сравнению со спецификацией МРОСЛ ДП-модели;

- далее осуществляется формальная верификация Event-B спецификации LSM, т. е. проверяется выполнение контрактов и сохранение инвариантов безопасности при выполнении моделирующих интерфейс LSM Event-B событий;
- в дальнейшем контракты событий Event-B спецификации LSM вручную транслируются в контракты этих же операций на языке спецификаций ACSL [52], что позволит верифицировать их выполнение в коде реализации модуля LSM. Корректность трансляции также проверяется при помощи экспертизы;
- в завершение верифицируется код на языке Си модуля LSM, доказывается соответствие реализации модуля LSM его ACSL-спецификации.

Верификация моделирующих интерфейс LSM Event-B событий доказывает выполнение требований соответствующих программных контрактов и выполнение общих требований по защите информации (инвариантов безопасности). Далее после трансляции Event-B спецификации LSM в ACSL появляется возможность провести верификацию кода модуля LSM. Результатом такой верификации будет не только доказательство того, что функции LSM выполняют требования, заданные в спецификациях, но и доказательство выполнения общих требований к защите информации (инвариантов безопасности) модуля LSM, поскольку ACSL-спецификация соответствует Event-B спецификации LSM, а из свойства «соответствия» следует сохранение в ACSL-спецификации свойств Event-B спецификации. Это важно, так как в противном случае пришлось бы доказывать сохранение инвариантов безопасности в ACSL-спецификации, а это существенно сложнее, чем в случае Event-B спецификации.

Данная глава посвящена описанию этапов цепочки работ, представленной выше — она раскрывает основные задачи, которые приходится решать по ходу верификации модуля LSM, но сначала дается более подробное описание работы LSM.

6.1. Подсистема LSM. Функционирование модуля безопасности ядра ОС Linux

Ядро Linux разработано на языке Си (стандарт C89 с gnu-расширениями [53]), имеет большой объем кода (более 15 миллионов

строк кода для версии v4.14 по оценке программы SLOCCount) и активно развивается на протяжении уже более двадцати лет. Ядро Linux является монолитным, т. е. все его части работают на одном уровне привилегий — в нулевом кольце. При этом ядро является модульным, т. е. позволяет загружать и выгружать отдельные драйверы/модули в процессе своей работы. Всегда остается некоторая часть ядра Linux, которая является резидентной, т. е. она не может быть выгружена: сюда относятся базовые подсистемы, например, планировщик, подсистемы управления памятью и процессами, виртуальная файловая система и др. Одной из таких резидентных подсистем (начиная с версии ядра 2.6.24 [54]) является подсистема безопасности Linux Security Modules (LSM) [55].

Подсистема LSM — это интерфейс, на основе которого могут быть реализованы разнообразные модули безопасности. В дистрибутиве ОС Linux может присутствовать несколько модулей LSM, и при конфигурировании системы администратор выбирает, какой (или какие) из модулей следует использовать. Так, в базовой версии ядра v4.14 представлены системы мандатного контроля доступа SELinux, Smack, Tomoyo, AppArmor.

Интерфейс LSM разрабатывался так, чтобы предоставлять все необходимое для реализации на его основе мандатной системы управления доступом [56]. Следует понимать, что механизм LSM — это прежде всего инструмент для дополнительного ограничения доступа, для реализации дополнительных проверок поверх основных (дискреционного управления доступом — DAC). LSM не может быть использован для того, чтобы заменять собой стандартные проверки прав доступа ядра Linux. LSM не предназначен для реализации каких-либо действий кроме проверок, например, затирания файлов на жестком диске при их удалении (secure_delete). Это является его архитектурными ограничениями. Начиная с версии ядра v4.2, вводится поддержка стековой модели работы модулей безопасности [57], что подразумевает под собой контроль доступа не одним, а несколькими модулями LSM.

Интерфейс LSM состоит из набора указателей на функции. Модуль безопасности должен реализовывать в себе соответствующие функции, зарегистрировать их в ядре при инициализации своей работы. Далее ядро вызывает функции интерфейса в тех участках своего кода, где требуется проверка прав доступа (например, при обработке системного вызова) или происходит выделение соответствующего ресурса (рис. 6.1). Правильность расстановки точек вызова функций интерфейса LSM в ядре анализировалась в [58–61].

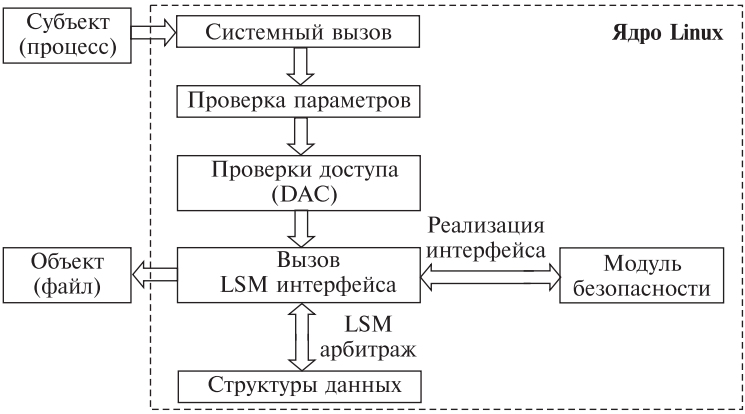

Рис. 6.1. Схема работы ядра ОС Linux с модулем LSM

Функции интерфейса делятся на два типа — оповещательные и контрольные. Первые нужны для того, чтобы модуль безопасности ядра знал, что ядро выделило какой-то ресурс, и мог, например, добавить в его поля собственную информацию о метках безопасности (атрибутах). Вторые — проверяют возможность доступа и возвращают код результата, например *EACCES*, *EPERM* (в случае, если доступ запрещен). Ядро анализирует результат обработки каждого из контрольных вызовов и продолжает обработку системного вызова или прекращает ее.

Интерфейс LSM избыточен в том смысле, что контрольные и оповещательные вызовы дублируются на разных уровнях абстракции. Например, права доступа к файлам можно проверять как права доступа к определенном объекту (inode, dentry) или как права доступа по определенному пути. Разные модули безопасности реализуют эти проверки по-своему (например, SELinux использует проверки inode, AppArmor — проверки по пути).

Интерфейс LSM содержит в себе большое число функций (более двух сотен). Модулю безопасности не обязательно реализовывать их все. Если модуль безопасности не реализует какую-то оповещательную функцию, то он не будет знать о наступлении какого-то события в ядре, если контрольную, то ядром будет считаться, что доступ модулем LSM разрешен.

Ядро может быть сконфигурировано для работы в многопроцессорном режиме, в таком случае каждый системный вызов обрабатывается в отдельном потоке ядра, и в коде модуля безопасности LSM одновременно могут вызываться разные обработчики (соответственно, требуется синхронизация для работы с разделяемыми данными).

Дополнительно необходимо заметить, что синхронизация также требуется, если ядро собрано в режиме вытеснения. В рамках данной редакции мы всегда полагаем, что ядро сконфигурировано для работы в одноядерном режиме и без механизма вытеснения ядерного кода.

Сам модуль безопасности может вызывать код подсистем ядра (рис. 6.2), запрашивая какие-то данные через функции (и в процессе работы этих функций управление опять может вернуться в модуль LSM, пройдя таким образом цикл по графу вызовов функций), вызывать какие-то библиотечные функции.

То есть модуль безопасности, реализующий интерфейс LSM, как вызывается ядром Linux, так и сам активно вызывает функции ядра. Однозначным образом предсказать статическими методами, каким будет граф потока управления при обработке, например, системного вызова, не всегда возможно. В отдельных случаях подобные «запутанные» потоки управления могут приводить к взаимным блокировкам [62]*.

При обработке одного системного вызова может вызываться несколько функций интерфейса LSM. Рассмотрим пример, где происходит системный вызов *open* для открытия файла testfile на чтение. Файл testfile расположен в том же каталоге, что и программа, его открывающая. Можно увидеть (табл. 6.1), что во время обработки данного системного вызова интерфейс LSM вызывается 6 раз (3 оповещательных и 3 контрольных вызова).

---

* По ссылке [62] идет описание ошибки взаимной блокировки в ядре в модуле распределенной файловой системы ceph, вызванной работой LSM модуля. При открытии файла выполняется операция *atomic_open* в модуле ceph. При обработке операции шлется запрос к серверу метаданных ceph MDS. Послав запрос к серверу MDS, поток ядра засыпает до тех пор, пока результаты запроса не будут обработаны и возвращены. За обработку ответов от MDS сервера в ядре отвечает специальный поток MDS dispatch. В процессе обработки вызывается оповещательный LSM-интерфейс instantiate. Он не требует возврата результата разрешения доступа, всего лишь оповещает модуль защиты о том, что inode и dentry связываются. Если в d_instantiate модуль защиты (в данном случае SELinux) запрашивает дополнительно xattrs (и они не были получены в прошлом ответе), то формируется новый запрос к MDS, что приводит к засыпанию уже MDS dispatch. После этого становится некому обрабатывать ответы от MDS сервера, так как ответственный за это поток ядра спит. Код модуля ceph построен так, что в процессе обработки ответа от MDS (в потоке MDS dispatch) нельзя посылать новый запрос.

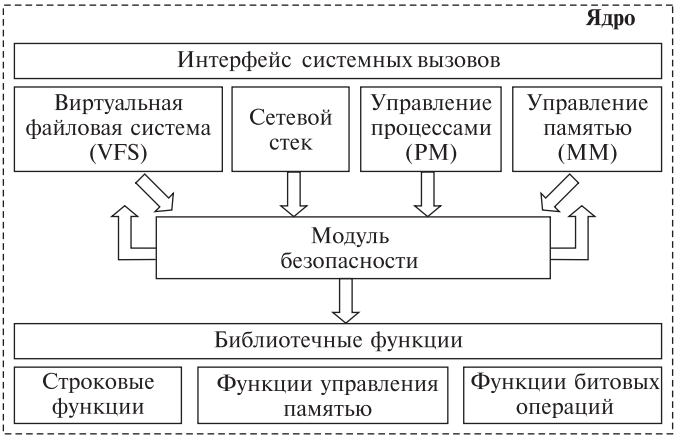

Рис. 6.2. Взаимодействие модуля безопасности с компонентами ядра ОС Linux

В ядре системный вызов open раскрывается в вызов функции do_open, из нее управление передается в do_filp_open и затем в path_openat. В последней сначала через серию вызовов контрольного интерфейса inode_permission с флагом MAY_EXEC осуществляется проверка прав доступа к родительскому каталогу файла (link_path_walk). После этого управление передается в vfs_open, do_dentry_open. В последней осуществляется вызов контрольного интерфейса file_open. Затем в зависимости от типа файла и файловой системы вызывается специальный обработчик операции. Для файлов ext4 — это ext4_file_open. В нем осуществляются специфичные для формата файловой системы действия. Для каталогов ext4 — это ext4_dir_open. В ext4_dir_open не производится никаких проверок, за исключением случаев, когда файловая система зашифрована.

На данном примере можно увидеть, что при обработке системного вызова ядро Linux сначала выделяет пустой файловый дескриптор и дает модулю безопасности записать в него свои метаданные. Затем происходит проверка прав доступа к родительскому каталогу открываемого файла (флаг *MAY_EXEC* для открытия каталога). После происходит аллокация структуры индексного дескриптора, о чем сообщается модулю безопасности. Далее происходит связывание структур индексного дескриптора со структурой dentry. Если бы данный файл к настоящему моменту использовался другими программами, то двух последних действий могло бы и не произойти. Затем происходит проверка доступа к индексному дескриптору файла testfile на чтение. То же действие повторяется на уровне файлового дескриптора.

Таблица 6.1

Системный вызов open("testfile", O_RDONLY). Последовательность вызовов функции интерфейса LSM при его обработке. Ядро Linux v4.14

Функция интерфейса LSM	Точка вызова ядра функции интерфейса	Описание функции точки вызова	Параметры LSM	Описание параметров
file_alloc_security (оповещательная)	*get_empty_filp* (*fs/file_table.c*)	Выделение структуры файла для процесса	file=0xc445fd80	Структура file представляет собой открытый файловый дескриптор
inode_permission (контрольная)	*__inode_permission* (*fs/namei.c*)	Проверка прав доступа к inode	inode=0xc6b57764 mask=0x81	Структура inode служит в ядре для внутреннего представления файлов. Может быть несколько файловых дескрипторов file, указывающих на один и тот же inode. Флаги параметра маски: *MAY_EXEC* | *MAY_NOT_BLOCK*
inode_alloc_security (оповещательная)	*inode_init_always* (*fs/inode.c*)	Инициализация структуры inode	inode=0xc69c6134	
d_instantiate (оповещательная)	*d_instantiate* (*fs/dcache.c*)	Связывание структуры inode и dentry	dentry=0xc6a98e40; inode=0xc69c6134	Структура dentry необходима для связывания структуры inode с именем файла, поддержания иерархии файлов в каталоге, и связи с родительскими каталогами
inode_permission (контрольная)	*__inode_permission* (*fs/namei.c*)	Проверка прав доступа к inode	inode=0xc69c6134; mask=0x24	Флаги параметра маски: *MAY_OPEN*| *MAY_READ*
file_open (контрольная)	*do_dentry_open* (*fs/open.c*)	Проверка прав доступа к file	file=0xc445fd80; cred=0xc44a8c40	Структура cred содержит в себе субъект/объектные метки/данные процесса в зависимости от того, в каком качестве выступает процесс. В данном случае как субъект доступа

Рис. 6.3. Схема процесса верификации кода модуля LSM

На этом небольшом примере можно увидеть, что некоторые проверки выполняются на разных уровнях представления: индексные дескрипторы служат для описания метаданных файла (есть взаимно-однозначное соответствие данных на диске и индексного дескриптора), структура file является дескриптором открытого файла у процесса и ссылается на индексный дескриптор. Также можно увидеть, что цепочка вызовов функций LSM сильно зависит от текущего состояния ядра и аргументов системного вызова. Например, если бы файл testfile был расположен в другом каталоге, то появились бы дополнительные контрольные вызовы *inode_permission* и могли бы появиться оповещательные вызовы *inode_alloc_security* и *d_instantiate*.

Исходя из описанной структуры модуля LSM и схемы его интеграции с ядром ОС Linux разработка спецификаций и верификация программного кода модуля LSM состоит из следующих работ (рис. 6.3):

- разработка спецификаций для функций верхнего уровня (интерфейсных функций модуля);
- последовательный перебор всех интерфейсных функций;
- если рассматриваемая функция не вызывает функций, которые ранее не были верифицированы, то она верифицируется;
- если такие функции имеются, то необходимо разработать для них спецификации и доказать их корректность (иногда приходится ограничиваться только разработкой спецификации), а после этого вернуться к верификации данной функции.

6.2. Дедуктивная верификация кода на языке Си

Задача дедуктивной верификации состоит в доказательстве того, что в предположении, что программа получает на вход допустимые данные (предусловие выполнено), она выдает результат, отвечающий требованиям к результату (постусловие выполнено). Кроме того, требуется также доказать, что программа завершается, т. е. не «зацикливается» и не содержит ошибок (неопределенного поведения). Ограничения на входные данные и требования к результату называются спецификацией программы. Кратко будем говорить,

что задача дедуктивной верификации состоит в доказательстве соответствия реализации функции (программы) ее спецификации. По сути для верификации Си-программ мы будем использовать тот же подход, который уже использовался для верификации Event-B спецификаций, однако в случае Си-программ в ядре ОС приходится использовать другие инструменты и учитывать особенности этого языка программирования, модели памяти Си-программ и используемых (библиотечных) функций ядра ОС.

Методы формальной верификации имеют долгую историю развития. Теоретические основы были заложены в работах Флойда (методы Флойда: метод частичной корректности и метод завершимости) [63] и Хоара [64]. Первой системой формального доказательства принято считать разработанную в 1969 году Дж. Кингом, студентом Р. Флойда, программу [65]. С тех пор появилось большое число систем формального доказательства программ, например [66–71].

Методы формальной верификации, не имевшие широкого применения в индустрии и развивавшиеся в стенах университетов и исследовательских институтов, в последнее время обретают интерес в реальных применениях [72, 73]. Это объясняется несколькими причинами: развитием инструментов верификации, развитием решателей для автоматического доказательства условий верификации, возрастающей стоимостью ошибки в массовом программном обеспечении (операционные системы, программы для работы с электронными деньгами, средства коммуникации, системы управления аппаратными комплексами, авиационной, автомобильной техникой и т. д.).

Однако из-за того, что инструменты верификации не имеют еще широкого применения за пределами академической среды, при попытках их применения к коду реальных программных систем обнаруживаются существенные недостатки и ограничения [74, 75] такие, как плохая документация, неполная поддержка языка Си, отсутствие поддержки расширений компиляторов, слабость моделей памяти, лежащих в их основе [76, 77], неудобство работы с большими объемами кода, отсутствие генерации отчетов о покрытии спецификациями кода и другого рода сопроводительной информации.

Для того чтобы провести дедуктивную верификацию нужно иметь спецификацию программы. Если спецификации нет (или она не полна), ее нужно разработать. Спецификации пишутся на специальных декларативных языках, как правило, в их математической основе лежит логика первого порядка. Существуют языки программирования (SPARK [68], Whiley [69], Dafny [70], ADA 2012 [78], Eiffel [79]), которые поддерживают синтаксические конструкции для

разработки спецификаций. Языки программирования (Си, Java), которые создавались без учета потребности в разработке формальных спецификаций на код, интегрируют их в код либо посредством специальных расширений языка (использования компиляторных атрибутов), либо с использованием препроцессорного языка (макросы, которые при компиляции раскрываются в пустые конструкции, но статически обрабатываются другими инструментами), либо посредством комментариев специального вида.

В данной работе мы следуем методу дедуктивной верификации Флойда–Хоара [63, 64]. Спецификации состоят из контрактов (предусловие и постусловие) к функциям, инвариантов и оценочных функций на циклы, инвариантов на типы данных и глобальные переменные, аксиоматических теорий. В предусловии описываются ограничения на аргументы функции и состояние глобального контекста, которому должно соответствовать выполнение программы на момент вызова данной функции, чтобы она отработала корректным образом. В постусловии описывается требование к результату функции (возвращаемому значению) и к побочному эффекту, состоянию измененных данных программы (переменных и областей памяти глобального контекста) на момент завершения данной функции.

Инструменты на основе спецификаций, исходного кода и собственных моделей (например, памяти и целых чисел) порождают набор условий верификации. Условие верификации представляет собой логическую формулу. Например, состоящую из предусловий контракта и цепочки условий, определяющих конкретный путь в коде, в предпосылках формулы и ограничения на неравенство нулю делителя в следствии. Если показать общезначимость данной формулы, то можно утверждать, что на любых входных данных в этой функции, удовлетворяющих предусловию и позволяющих дойти до деления этим путем по коду, никогда не произойдет ошибки деления на ноль. Для верификации функции требуется доказать общезначимость всех условий верификации. После этого можно утверждать, что функция полностью соответствует своему контракту и не содержит определенных классов ошибок.

Для доказательства постусловий циклы аннотируются инвариантами — логическими условиями, которые должны соблюдаться на каждой итерации цикла (доказывается статически по индукции). Для доказательства завершимости циклов им ставится в соответствие оценочная функция — монотонно убывающая функция, ограничивающая число итераций цикла, которое может произойти до гарантированного завершения данного цикла.

Хотя математический аппарат для дедуктивной верификации достаточно прост, его использование в применении к реальным программам требует инструментальной поддержки. В следующей главе описываются соответствующие инструменты.

6.2.1. Инструментарий

В данной главе описывается опыт дедуктивной верификации LSM при помощи комплекса инструментальных средств AstraVer Toolset [80], который был создан в ИСП РАН. AstraVer Toolset включает в себя набор инструментов, предназначенных для дедуктивной верификации Си-программ, основными среди которых являются Frama-C [81], Why3 [71] и AstraVer Plugin [76]. Frama-C — это платформа статического анализа программ на языке Си, разработанная совместно двумя французскими организациями: CEA LIST (Software Reliability Laboratory) и INRIA Saclay. AstraVer Plugin является ответвлением (форком) плагина Jessie [82] и использует платформу Why3. Спецификации для AstraVer пишутся на языке ACSL [52], который позволяет разрабатывать спецификации различного уровня, от абстрактных до низкоуровневых. Инструменты Frama-C и Why3 из AstraVer Toolset адаптированы для работы с AstraVer Plugin и имеют дополнительную функциональность, включающую в себя поддержку лемма-функций для упрощения доказательства лемм, поддержку переключения моделей работы с целыми числами для конкретных операций, импорт лемм, поддержку произвольного порядка определений логических сущностей, расширенную поддержку работы со строками на уровне спецификаций, поддержку offsetof и container_of, массивов нулевого размера и другие улучшения, делающие возможным использование инструментов дедуктивной верификации в работе с реальным кодом.

Why3 — это платформа для дедуктивной верификации. В связке с Frama-C она используется как интерактивная среда для работы с доказательством условий верификации. Также она предоставляет среду интеграции для запуска различных автоматических и интерактивных решателей, обрабатывает протоколы доказательств, дает возможность применять различные трансформации для логических формул (разбиения, подстановки, инверсии и т. д.). Why3 позволяет использовать большое число различных решателей: Coq, Alt-Ergo, CVC3, CVC4, Yices, Z3 и т. д.

Вся связка инструментов дедуктивной верификации работает следующим образом (рис. 6.4). Исходный код со спецификациями поступает в Frama-C, который преобразует их в промежуточное

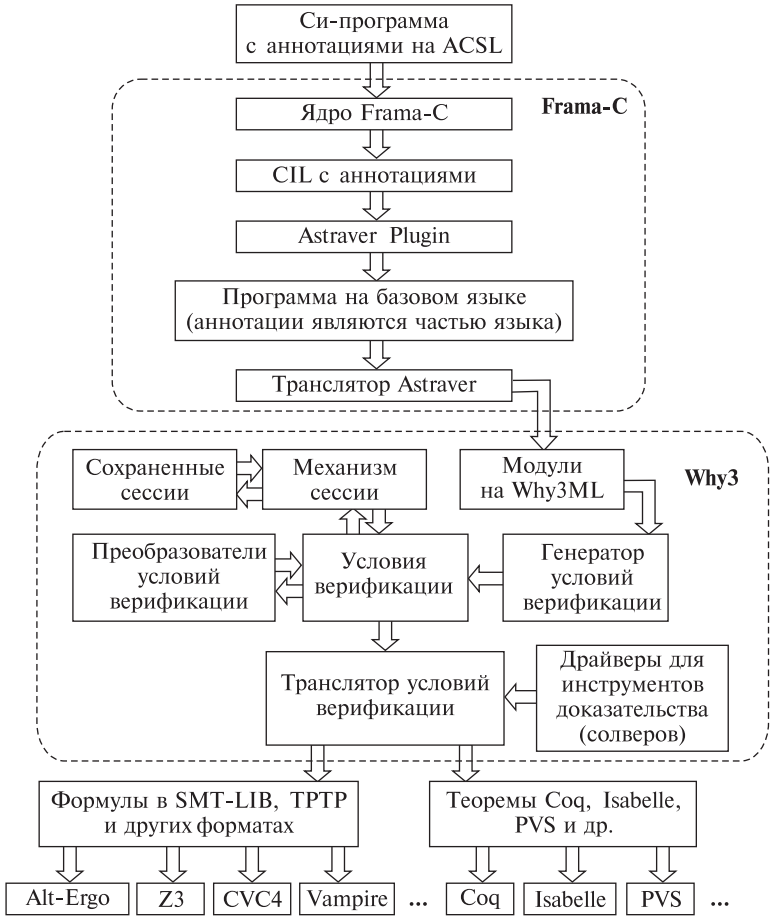

Рис. 6.4. Инструментарий дедуктивной верификации программ на языке Си

представление C Intermediate Language (CIL). Далее плагин AstraVer, подключая модели памяти и целых чисел, преобразует CIL в модель на языке WhyML. На основе этой модели Why3 генерирует условия верификации. Они выводятся автоматически на основе правил формального вывода, спецификаций и формальных моделей языка программирования (к ним относятся модели памяти, модели целых или действительных чисел, модель битовых векторов и др.). Why3 дает возможность проверять их выполнимость либо посредством запуска автоматических решателей, либо при помощи интерактивной среды доказательства.

6.2.2. Пример спецификаций на языке ACSL

Перед тем как рассмотреть пример спецификации на языке ACSL, кратко опишем общую схему разработки и размещения спецификаций в программе на Си. Спецификации располагаются вместе с кодом на языке Си и оформляются как псевдокомментарии, помеченные знаком «@»:

/* @ ...*/ или //@ ...

Спецификации могут описывать свойства типов данных, функций, констант и переменных. В спецификационных выражениях могут использоваться обращения к элементам специфицируемой программы: типы данных, константы, переменные, макросы и т. д. Помимо этого в спецификации могут использоваться дополнительные конструкции, например \forall ($\forall$) и \exists ($\exists$) — кванторы всеобщности и существования, \valid — встроенная функция для проверки корректности выделенной памяти и др. В спецификации могут быть введены спецификационные переменные ghost, логические функции и предикаты, ими можно пользоваться только в спецификационных выражениях.

Рассмотрим, как может выглядеть спецификация на функцию *strcat*. Напомним, что данная функция осуществляет конкатенацию двух строк *dest* и *src*.

Спецификационный контракт функции, как правило, состоит из следующих элементов: предусловия, постусловия и рамочных условий (которые в данном случае можно рассматривать как разновидность постусловий). Предусловия начинаются с ключевого слова requires, постусловия — ensures, рамочные условия — assigns. В спецификациях по умолчанию используется неограниченный тип integer ($\mathbb{Z}$) для целых чисел (в отличие от ограниченных типов в языке Си).

Предусловия описывают требования к значениям аргументов функции к глобальному контексту в момент времени «до выполнения функции». В предусловиях невозможно сформулировать условия на возвращаемый результат функции и на ее побочные действия.

Постусловия вычисляются на момент времени «после выполнения функции» и описывают требования к возвращаемому результату и глобальному контексту. В постусловии есть специальная конструкция \result, которая описывает возвращаемое функцией значение. В постусловиях можно обратиться к состоянию памяти «до выполнения функции» через временную метку Old и конструкцию \old.

Рамочные условия описывают память, в которую функция может осуществлять запись. Таким образом, если описываемое множество является пустым, то функция — чистая и не содержит побочных эффектов. Также на уровне рамочных условий может задаваться статус аллокации регионов памяти. В данном примере он не описывается, так как функция strcat не выделяет и не освобождает память.

Дополнительно в контракте возможно описать условия завершимости функции, но по умолчанию всегда считается что функция завершима, и порождается соответствующее условие верификации для доказательства этого.

В контракте функции *strcat* (листинг 6.1) имеется 4 предусловия. В первых двух говорится, что входящие указатели *src* и *dest* указывают на корректно выделенные участки памяти подходящего размера, имеют маркеры конца строки. Это делается через предикат *valid_str*, определение которого приведено ниже. В третьем предусловии говорится, что сумма двух строк не превышает максимально допустимый размер для типа *size_t*. В следующем предусловии говорится, что указатель *dest* указывает на корректно выделенный участок памяти, размер которого достаточен для того, чтобы вместить обе строки.

Листинг 6.1. Контракт функции strcat lib/string.c

```
/*@ requires valid_str(src);
    requires valid_str(dest);
    requires strlen(dest)+strlen(src) ⩽ SIZE_MAX;
    requires \valid(dest+(0..strlen(dest)+strlen(src)));
    assigns dest[strlen(dest)..strlen(dest)+strlen(src)];
    ensures \result ≡ dest;
    ensures valid_str(\result);
    ensures ∀ ℤ i; 0 ⩽ i < strlen{Old} (dest) ⇒
      \old(dest[i]) ≡ \result[i];
    ensures ∀ ℤ i;
      strlen{Old} (dest) ⩽ i < strlen{Old} (dest)+strlen(src) ⇒
      src[i - strlen{Old} (dest)] ≡ \result[i];
    ensures strlen(\result) ≡ strlen{Old} (dest) + strlen(src);
*/
char *strcat(char *dest, const char *src);
```

В рамочном условии assigns описывается участок памяти, за пределами которого функция *strcat* ничего не меняет. В данном случае он соответствует участку памяти, куда будет копироваться строка *src*.

Если не задавать постусловия, то по умолчанию всегда будет считаться, что постусловие равно истине при любом результате функции (вырожденные постусловия). Контракта без постусловий и рамочных условий будет достаточно, чтобы доказать отсутствие определенных ошибок в исходном коде функции (например, деление на ноль, выход за границу массива и др.). Так, если посмотреть на спецификации к функции TimSort, которые использовались для того, чтобы показать, что в реализации OpenJDK [83] есть ошибка, то можно увидеть, что в некоторых случаях используются вырожденные постусловия (ensures \true). Если же понадобится доказать корректность функции, вызывающую данную, то с большой вероятностью потребуется сформулировать постусловия для нее.

На практике встает вопрос полноты постусловий: например, можно описать, что функция возвращает отсортированный массив, но не уточнить, что содержимое входного и выходного массивов совпадает. В таком случае функция, всегда возвращающая массив «[1, 2, 3]», формально отвечает требованиям «отсортированности» результата. Не включенное в спецификацию постусловие о связи входных данных и возвращаемого результата в данном случае делает и спецификацию, и соответствующую ей реализацию бессмысленными. Полноту постусловий для верхнего уровня функций сложно гарантировать формальным образом. В рамках нашей работы полнота постусловий этих функций определяется наличием возможности выразить свойства, гарантирующие сохранение инвариантов безопасности.

Листинг 6.1 описывает 5 постусловий. В первом сообщается, что значение возвращаемого указателя совпадает с *dest*. Во втором постусловии уточняется, что этот указатель — указатель на корректную строку. В третьем условии говорится, что возвращаемая строка начинается с символов входной строки *dest* (без спецсимвола «конец строки»). Это условие является избыточным и вытекает из рамочного условия assigns. Оно было оставлено для того, чтобы дать подсказку решателям при доказательстве рамочного условия (без этого постусловия они хуже справляются с его автоматическим доказательством). Четвертое условие задает требование ко второй части строки: после символов из *dest* должны следовать символы из *src*, иными словами, строка *src* добавляется к концу *dest*. Последнее постусловие говорит о том, что длина результирующей строки равна сумме длин исходных.

Контракта функции достаточно, чтобы доказывать корректность функций, вызывающих данную, но может быть недостаточно

для того, чтобы доказывать корректность реализации самой функции, в частности, контракта недостаточно, чтобы доказать завершимость функции. Если в теле функции используются циклы, то необходимо дополнительно сформулировать инварианты на цикл и оценочную функцию. Инварианты должны быть сформулированы таким образом, чтобы из них вытекали постусловия. Оценочная функция необходима для того, чтобы доказать завершимость цикла.

Реализация функции *strcat* (см. Листинг 6.2) содержит два цикла. Первый — для определения конца строки *dest*, второй — для копирования строки *src* в конец *dest*.

Листинг 6.2. Инварианты и оценочная функция для strcat

```
char *strcat(char *dest, const char *src)
{
  char *tmp = dest;
  //@ ghost size_t dest_len = strlen(tmp);
  /*@ loop invariant tmp ⩽ dest ⩽ tmp + dest_len;
      loop invariant valid_str(dest);
      loop invariant ∀ ℤ i;
        0 ⩽ i < dest - tmp ⇒ tmp[i] ≢ '\0';
      loop variant dest_len - (dest - tmp);
    */
    while (*dest)
      dest++;
  //@ assert *dest ≡ '\0';
  //@ assert dest ≡ tmp + dest_len;
  //@ ghost char *osrc = src;
  //@ ghost char *mdest = dest;
  /*@ loop invariant osrc ⩽ src ⩽ osrc + strlen(osrc);
      loop invariant mdest ⩽ dest ⩽ mdest + strlen(osrc);
      loop invariant src - osrc ≡ dest - mdest;
      loop invariant valid_str(src);
      loop invariant ∀ ℤ i; 0 ⩽ i < src - osrc ⇒
        mdest[i] ≡ osrc[i];
      loop invariant ∀ ℤ i; 0 ⩽ i < dest - tmp ⇒
        tmp[i] ≢ '\0';
      loop assigns mdest[0..strlen(osrc)];
      loop variant strlen(osrc) - (src - osrc);
  */
  while ((*dest++ = *src++) != '\0')
    ;
  //@ assert ∀ ℤ i; 0 ⩽ i < dest_len ⇒ \at(dest[i],Pre) ≡ tmp[i];
  //@ assert dest[-1] ≡ '\0' ∧ src[-1] ≡ '\0';
  //@ assert dest - 1 ≡ tmp + dest_len + strlen(osrc);
```

```
  //@ assert strlen(osrc) ≡ src - osrc - 1;
  /*@ assert ∃ size_t n;
    tmp[n] ≡ '\0' ∧
    \valid(tmp+(0..n)) ∧
    n ≡ (size_t) (dest_len + strlen(osrc));
    */
  /*@ assert valid_str(tmp) ∧
    (tmp[(size_t)(dest_len + strlen(osrc))] ≡ '\0') ∧
      (∀ ℤ i; 0 ⩽ i < (size_t)(dest_len + strlen(osrc)) ⇒
        tmp[i] ≢ '\0');
    */
  //@ assert strlen(tmp) ≡ (size_t) (dest_len + strlen(osrc));
  return tmp;
}
```

Спецификационные конструкции ghost используются для «запоминания» значения переменных в определенный момент времени. При этом в первой спецификации в ghost-переменную сохраняется результат вычисления логической функции *strlen*. Спецификации assert в данном примере необходимы для того, чтобы дать «подсказки» решателям при автоматическом доказательстве условий верификации. На каждый assert порождаются соответствующие условия верификации, доказательство следующих за ними по коду утверждений происходит, отталкиваясь от них в предпосылках.

Для первого цикла написано три инварианта. В первом утверждается, что указатель *dest* в рамках цикла изменяется в определенных пределах. Во втором утверждается, что *dest* не перестает указывать на корректную строку. В последнем говорится, что маркер конца строки не встречался до конца массива.

Оценочная функция на цикл должна на каждом следующем шаге итерации возвращать результат меньший, чем на текущем. Ее доменом должно быть фундированное множество (множество, в котором не существует бесконечной строго убывающей последовательности) натуральных чисел с нулем.

Для второго цикла первые два инварианта описывают диапазоны изменения указателей *src* и *dest*. В третьем инварианте утверждается, что указатели *src* и *dest* на каждой итерации сдвигаются на равное число байтов. В четвертом инварианте описывается то, что *src* по-прежнему указывает на «валидную» строку на каждой итерации данного цикла. В пятом инварианте утверждается, что все «пройденные» элементы строки *src* записаны в конец оригинальной строки *dest*. Из этого инварианта вытекает предпоследнее постусловие в контракте функции. В последнем инварианте записано, что

маркер конца строки не попадает в результирующую строку до тех пор, пока не будет скопирован весь массив *src*. Этот инвариант нужен для доказательства последнего постусловия.

Описанных инвариантов и оценочных функций достаточно для доказательства полной корректности функции *strcat* относительно ее контракта, модели памяти и модели целых чисел. Для того чтобы решатели могли доказать все условия верификации в автоматическом режиме, в тело функции были добавлены логические утверждения в виде конструкций *assert*.

При разработке спецификации использовались предикат *valid_str* и логическая функция *strlen*. Листинг 6.3 содержит их определения. Помимо этого, в нем приведены некоторые леммы на поведение *valid_str* и *strlen*. Леммы используются решателями при доказательстве условий верификации. Сами леммы, как правило, доказываются с помощью среды интерактивного доказательства Coq.

Листинг 6.3. Логическая функция strlen

```
/*@ axiomatic Strlen {
    predicate valid_str(char *s) =
    ∃ size_t n;
      s[n] ≡ '\0' ∧ \valid(s+(0..n));
    lemma valid_str_shift1:
    ∀ char *s;
      *s ≢ '\0' ∧
      valid_str(s) ⇒
        valid_str(s+1);
    lemma valid_str_strend:
    ∀ char *s;
      \valid(s) ∧ *s ≡ '\0' ⇒
      valid_str(s);
    logic size_t strlen(char *s) =
      s[0] ≡ '\0' ?
        (size_t) 0 : (size_t) ((size_t)1 + strlen(s + 1));
    lemma strlen_before_null:
    ∀ char* s, ℤ i;
      valid_str(s) ∧
      0 ⩽ i < strlen(s) ⇒ s[i] ≢ '\0';
    lemma strlen_at_null:
    ∀ char* s;
      valid_str(s) ⇒ s[strlen(s)] ≡ '\0';
    lemma strlen_shift1:
    ∀ char *s;
      valid_str(s) ∧ *s ≢ '\0' ⇒
        strlen(s) ≡ 1 + strlen(s+1);
```

```
    lemma strlen_main:
    ∀ char *s, size_t n;
      valid_str(s) ∧
      s[n] ≡ '\0' ∧
        (∀ ℤ i; 0 ⩽ i < n ⇒ s[i] ≢ '\0') ⇒
          strlen(s) ≡ n;
    }
*/
```

Предикат *valid_str* утверждает, что существует некоторый индекс n, по которому в строке содержится нулевой байт (маркер конца строки). Также утверждается, что память от начала строки до этого индекса n должна ранее быть корректным образом выделена (\valid). Если этот предикат выполнен, то строка считается корректной (валидной). Дополнительно в этом предикате можно было бы потребовать, что до индекса n в строке *str* нет значений с нулевым байтом. В лемме *valid_str_shift1* утверждается, что предикат *valid_str* сохраняется для строки, сдвинутой в право на один байт, если в текущем не содержится конца строки. Эта лемма используется для доказательства второго и четвертого инвариантов в циклах *strcat*. Лемма *valid_str_strend* утверждает, что из того, что указатель указывает на валидный байт памяти, в котором расположен ноль, следует выполнение предиката *valid_str*. То есть пустая строка — валидна.

Листинг 6.3 содержит два определения для функции *strlen*. Первое (logic) описывает алгоритм вычисления *strlen* явным образом, используя рекурсию. Второе (лемма *strlen_main*) задает ее значение неявным образом, т. е. формулирует требование к правильному результату. Лемма *strlen_main* используется для доказательства последнего постусловия функции *strcat*, если посмотреть на одно из утверждений assert, то явно видно, что он повторяет структуру предпосылок данной леммы. Лемма *strlen_before_null* утверждает, что в индексах до *strlen* не может лежать нулевого байта. В лемме *strlen_at_null* утверждается, что в строке по индексу *strlen* должен лежать маркер конца строки. В лемме *strlen_shift_1* описывается, как изменяется значение логической функции *strlen*, если сдвинуть указатель строки на один байт в право.

Таким образом, мы на примере функции *strcat* из ядра Linux рассмотрели, как могут выглядеть спецификации ACSL. Формальные спецификации на эту и другие библиотечные функции ядра Linux, которые могут использовать модули ядра в своей работе, а также протоколы доказательств к ним были частично опубликованы в рамках AstraVer [84]. Необходимо отметить, что функция

strcat является функцией «нижнего» уровня (по графу вызовов) и в ее спецификации не отражены высокоуровневые инварианты безопасности, полученные из Event-B спецификации интерфейсов LSM, потому что их к данной функции нет.

6.2.3. Текущие ограничения стека инструментов Frama-C/AstraVer

Инструменты верификации на текущий момент имеют ограничения на работу со следующими конструкциями языка Си:

- goto — разрешены безусловные переходы, направленные только вперед по коду, не передающие управление во вложенные блоки кода;
- преобразование из целочисленных типов в указатели и обратное преобразование не поддерживаются;
- преобразования между произвольными указателями не поддерживаются. Исключение составляют преобразование из void * в указатель другого типа, преобзования указателей на целые числа друг в друга, преобразования указателя на поле структуры в указатель на объемлющую структуру (container_of, offsetof), преобразование указателя на union к указателю на одно из полей union структуры и другие частные случаи;
- функции, являющиеся обертками к вызову функций, которые возвращают void * (пример: void * myalloc() {return kmalloc()}), не поддерживаются;
- операции с union-структурами данных лишь частично поддерживаются — нельзя производить операции над одним типом данных в union, а после работать с этими же данными через другой тип;
- функции с переменным числом аргументов не поддерживаются;
- квалификаторы типа volatile и const принимаются, но не учитываются;
- ассемблерные вставки поддерживаются только на уровне разбора синтаксических конструкций, эффект вычислений, описанный вставкой, не учитывается.

6.3. Процесс верификации: работы, планирование, координация

Как и в любом большом проекте (каким, конечно, является ядро Linux), процесс верификации невозможно рассматривать в отрыве от развития и сопровождения целевой системы. Поэтому при верификации модуля безопасности ОССН Astra Linux Special Edition

планированию и координации работ по развитию ОССН и по верификации LSM уделялось серьезное внимание. В частности, были проведены работы по согласованию планов и правил разработки кода модуля LSM, о чем подробнее написано ниже.

Размер модуля безопасности составляет более 3 тыс. строк кода на Си. Для составления спецификаций интерфейсных функций LSM требуется не только изучение документации, но и анализ кода ядра. Также требуется разработать спецификации вызываемых из модуля (библиотечных) функций ядра ОС. Некоторые из них должны сами быть верифицированы для того, чтобы разработчик спецификаций был уверен, что их спецификации корректны. Тем самым размер подсистемы, которая должна быть специфицирована и верифицирована, оказывается существенно больше, чем размер собственно модуля безопасности, а это означает, что сроки и трудоемкость верификации измеряются годами и человеко-годами соответственно. Это требует четкого планирования работ, анализа приоритетов и выстраивания последовательности разработки спецификаций и проведения верификации. Еще одним фактором, влияющим на последовательность работ, а также на их трудоемкость, является постоянное развитие инструментов верификации. В предыдущем разделе было описано текущее состояние возможностей инструментов верификации. Так как работы по совершенствованию методов и инструментов идут постоянно, при планировании частных работ следует учитывать не только текущее состояние технологии, но и то, на что можно рассчитывать в ближайшей перспективе. Тогда некоторые работы, которые сейчас требуют чрезмерных усилий, можно отложить на несколько месяцев и получить нужный результат, возможно, несколько позже, но существенно с меньшими затратами. При этом важно понимать, что артефакты верификации в дальнейшем придется сопровождать, поэтому важно учитывать не только трудоемкость собственно верификации, но и трудоемкость дальнейшего сопровождения.

Далее мы описываем отдельные работы, отдельные аспекты планирования работ и некоторые выводы, извлеченные из опыта проекта AstraVer.

6.4. Согласование правил разработки кода

Код модуля безопасности, являясь частью самого ядра, в существенной мере наследует стиль кода, принятый в сообществе разработчиков ядра ОС Linux. Код ядра не ограничен рамками какого-либо «безопасного» подмножества языка Си, например [85]. Реали-

зация модуля написана с целью обеспечения оптимального времени работы, а не для упрощения задач верификации. В модуле LSM активно используются расширения стандарта языка Си-99, которые используются сообществом разработчиков ядра ОС Linux (диалект поддерживается компилятором GCC). Более того, так как модуль безопасности опирается на кодовую базу ядра, то использование некоторых конструкций языка, паттернов и расширений заранее навязано, и не всегда имеется возможность от них отказаться.

Трудоемкость верификации в значительной степени зависит от сложности верифицируемого кода. Анализ сложности, в частности, при помощи соответствующей оценки, позволяет указать на наиболее сложные фрагменты программ. Знание о «сверх-сложных» фрагментах дает возможность провести их анализ вместе с разработчиками и, как правило, упростить код, что дает общий выигрыш: разработчики получают код, который проще сопровождать, а инженеры по формальному анализу получают код, с которым могут работать. Некоторые подробности об использованных методах измерения сложности программ приведены ниже.

В связи с этим работам по верификации модуля LSM предшествует предварительная фаза изучения особенностей его реализации и сопоставления стиля кода с имеющимися возможностями инструментов верификации. По результатам предварительного анализа с разработчиками кода модуля безопасности устанавливается соглашение по используемым правилам разработки.

6.4.1. Работа с макросами

Конструкции препроцессора, являясь отдельным языком, требуют самостоятельного рассмотрения. Спецификации на языке ACSL не могут быть написаны на макросы языка Си: инструменты работают уже с препроцессированным кодом. При верификации кода модуля каждый макрос требует анализа, так как:

- некоторые из макросов раскрываются в простое логическое условие и могут использоваться в спецификациях функций, так как они не нарушают синтаксиса языка спецификаций ACSL;
- часть раскрывается в побочные действия, специфицировать которые не требуется (например, likely, unlikely), часть — в инициализацию структур данных (ATOMIC_INIT). Некоторые из этих макросов могут быть заменены на простое раскрытие своих аргументов (#define likely(x) (x));
- часть раскрывается в последовательность вызовов функций. В данном случае необходимо написать спецификации на эти функции;

- некоторые макросы целесообразнее переопределить как функции, для того чтобы стало возможным описать их поведение. Не для всех макросов это возможно.

После анализа макросов:

- составляются списки макросов, которые могут использоваться в спецификациях;
- переопределяются те, которые при верификации можно заменить несложными заглушками;
- переписываются на встраиваемые (inline) функции те, корректность которых нужно доказать;
- на основании анализа исходных текстов совместно с разработчиками формулируются правила написания макроопределений, которые позволят упростить верификацию последующих версий модуля LSM.

6.4.2. Разработка спецификаций

Как уже упоминалось, при дедуктивной верификации разработка спецификаций выполняется для всех функций модуля безопасности и, если необходимо, для вызываемых из них функций ядра (см. рис. 6.3). При дедуктивной верификации разработка спецификаций ведется для всех функций модуля безопасности. Спецификационные контракты разрабатываются для всех функций ядра, которые вызываются модулем. Доказательство корректности производится только для функций модуля и некоторых библиотечных функций ядра. Таким образом, результатом работы является набор спецификаций с протоколами доказательств для всех функций модуля ядра безопасности, набором контрактов для использующихся модулем функций ядра без доказательств. Также без доказательств остается соблюдение корректности предусловий ядром в точках вызова функций модуля безопасности (предусловия для LSM-интерфейса).

6.4.3. Планирование работ. Оценка сложности

Код функции не может быть верифицирован на соответствие ее спецификации, пока не специфицированы все функции, которые вызывает данная. Соответственно, процесс верификации кода идет снизу-вверх по графу вызовов.

Для планирования работ желательно иметь возможность автоматического построения графа вызовов. Для решения этой задачи авторами был разработан инструмент, который строит карту исходного кода на основе графа вызовов внутримодульных функций

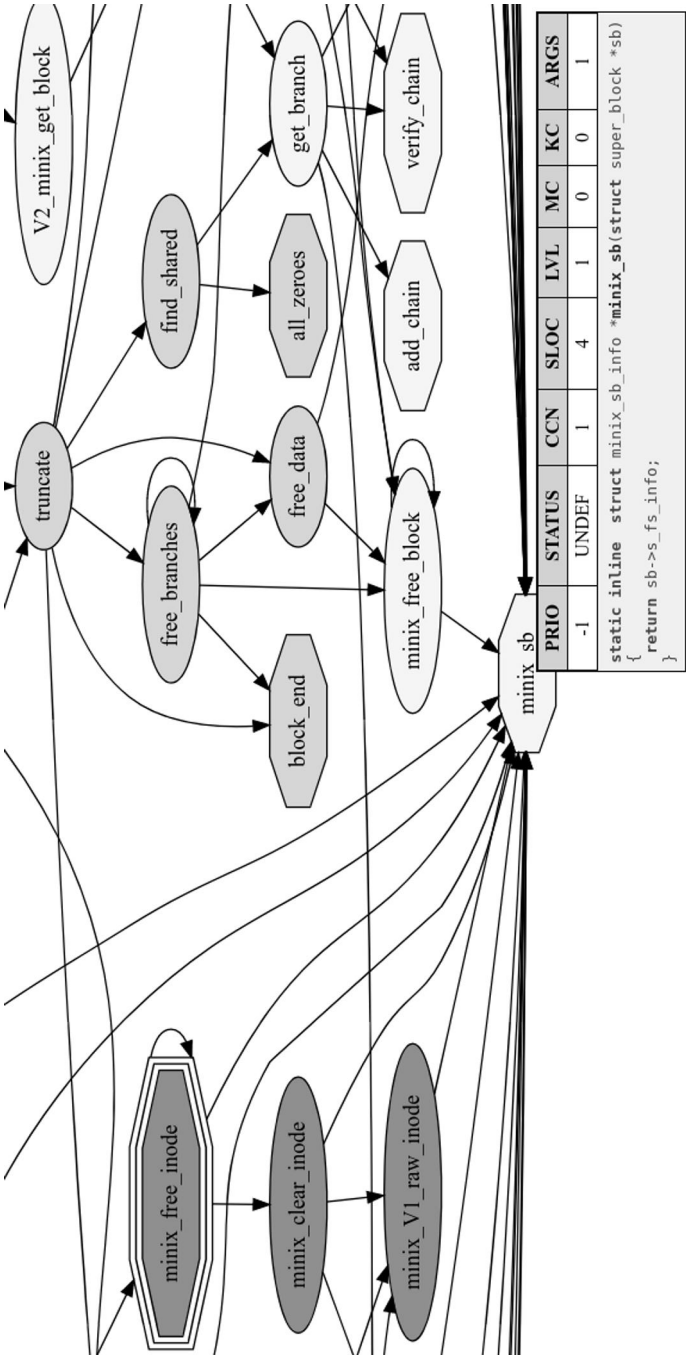

Рис. 6.5. Пример карты исходного кода драйвера файловой системы Minix

(рис. 6.5). Карта наглядно отображает характеристики кода, что упрощает оценку сроков и координацию работ. Так, на карте может отражаться информация об исходном коде функций (число строк, цикломатическая сложность) и о размере спецификаций.

При планировании и оценке сроков работ по верификации используются оценки сложности кода. Для чего учитывается размер каждой функции в строках кода (используется SLOCCount), число функций, которые она вызывает, цикломатическая сложность.

Цикломатическая сложность функции позволяет оценить, какое число тестов необходимо, чтобы покрыть все возможные пути исполнения кода в ней.

Опыт показал, что для дедуктивной верификации наибольшую сложность представляют функции с высокой цикломатической сложностью. Национальный институт стандартов и технологий США (NIST) в методике структурного тестирования рекомендует не выходить за пределы сложности в 10–15 единиц [86]. По договоренности с разработчиками подобный код упрощается.

6.4.4. Разработка спецификаций для функций ядра, которые использует модуль безопасности

Модуль безопасности может вызывать код из любых подсистем ядра, как резидентной его части, так и других модулей. Для того чтобы доказать корректность функции модуля, которая вызывает функцию ядра, необходимо иметь контракт последней. В точке вызова проверяется соблюдение предусловий, для последующего кода в формулу вставляются постусловия из контракта.

Для функций ядра, код управления которых уходит «глубоко» в подсистемы ядра, нерационально доказывать корректность: в большинстве случаев это невозможно из-за объемов кода. Для таких функций пишется только спецификационный контракт. Зачастую этот контракт не полный и не описывает всех побочных эффектов, которые производит функция: формализуются только те, которые могут оказать влияние на код модуля безопасности. Для полноты контракта потребовалось бы формализовать работу подсистем ядра: моделировать на уровне спецификаций работу подсистем планировщика, аллокатора и др. Это не представляется возможным ввиду сложности и объемов работы.

Некоторые функции, вызываемые модулем ядра, не зависят от подсистем ядра, и их корректность может быть доказана. Как правило, это библиотечные функции, например функция *strcat* из

прошлого раздела. В таком случае можно разработать полный контракт и верифицировать функцию. Для некоторых библиотечных функций были опубликованы спецификации и протоколы доказательств [84].

6.4.5. Тактика верификации — рабочий цикл процесса

При планировании работ приходится встречаться с различными трудностями, которые вызваны либо сложностью верифицируемого кода, либо ограничениями инструментов верификации, либо отсутствием соответствия между кодом и спецификациями к нему и т. д. В данной главе описывается типовой процесс верификации с изложением часто встречающихся проблем и рекомендациями по их решению или обходу.

При дедуктивной верификации инструментом Frama-C работа с функцией сводится к нескольким шагам:

1. Выборка исходного кода функции и связанного с ней кода, предназначенного для анализа. Во время разработки спецификаций требуется обозревать большое число кода единовременно. Постоянные переключения между различными файлами лишь отвлекают разработчика спецификаций. Вместе с тем функция и ее зависимости по коду/данным не всегда локализованы и расположены близко друг от друга. Помимо этого, скорость работы верификационных инструментов существенно зависит от объема кода, на котором они запускаются. Специальным образом из суммы исходных кодов модуля и подключаемых ими заголовочных файлов ядра выделяются все зависимости по коду/данным для рассматриваемой функции. При этом сохраняется оригинальная структура кода функций, включая форматирование и комментарии. Информации, содержащейся в получившихся файлах, достаточно, чтобы компилятор мог собрать объектный файл.

2. Формализация спецификаций. Зачастую, первым шагом при разработке спецификаций для уже существующего кода является «прямое» переложение кода реализации на спецификацию. Последние в дальнейшем дополняются и расширяются деталями, формализмами, соответствующими более глубоким и неявным уровням зависимостей и особенностям функционирования. Важно отметить, что разработка спецификаций при помощи простого механического отражения кода реализации без описания содержательного понимания назначения функции ведет к появлению «плохих» спецификаций. К сожалению, формальных критериев, позволяющих различать «содержательные» и «механические» спецификации в общем случае сформулировать не удается.

3. Доказательство соответствия кода спецификациям. Для того чтобы осуществить доказательство соответствия кода его спецификациям используются специальные программы — решатели. Эти программы в большинстве случаев способны автоматически доказывать условия верификации, которые генерируются из спецификаций, исходного кода и инструментальных моделей языка Си. Решателям не удается автоматически доказать условие верификации в нескольких случаях: ошибка в спецификации, неполнота спецификации, нехватка временных ресурсов и выделенной памяти для работы решателя (в этом случае можно оставить на уровне спецификаций «подсказки» решателям в виде утверждений assert), ошибка в исходном коде, ошибка в инструментах (неполнота моделей). Часто, если имеется какое-то «сложное» для автоматического вывода утверждение, то оно оформляется в виде соответствующей леммы на языке ACSL. Это делается с той же целью, чтобы помочь решателям провести доказательство автоматически. Соответственно, сформулированные леммы должны доказываться отдельно в специальных инструментах интерактивного доказательства (например, PVS, Coq). Шаги 2 и 3 зачастую «зацикливаются» в том смысле, что сначала разрабатывается спецификация на функцию, затем следует работа с инструментами верификации, попытки доказательства. Далее обнаруживаются ошибки в спецификации и/или исходном коде. И следует итеративная доработка спецификации, исходного кода до того момента, пока соответствие кода его спецификации не становится доказанным.

После верификации функции спецификации переносятся назад на исходный код модуля системы безопасности вместе с правками кода, если они были. На последнем шаге нужно проводить полноценное «передоказательство» на всем коде модуля. Это необходимо, так как спецификации зависимых от нее функций могли уточняться, как и общие с другими функциями предикаты и нужно убедиться в том, что это не влияет на соответствие остальных функций их спецификациям. Также на этом шаге проверяется непротиворечивость самих спецификаций, что нет взаимоисключающих требований к коду.

6.5. Пример

Представленный ниже пример демонстрирует переход от спецификации функции на Event-B к ее спецификации на языке ACSL, которая может быть использована для дальнейшей верификации кода функции. В качестве специфицируемой функции в этом примере

выступает функция из интерфейса модуля LSM *task_kill*, определяющая допустимость или недопустимость отправки одним процессом другому сигнала в соответствии с текущей политикой безопасности. В МРОСЛ ДП-модели этой функции соответствует правило *delete_subject*, завершающее работу сессии по команде из другой сессии.

В качестве исходной модели политики безопасности в этом примере используется специализированная версия МРОСЛ ДП-модели, в которой возможность доступа сессии (процесса) к сущности (информационному объекту, например файлу, директории, сокету и пр.) определяется на основе не ролей, а уровней доступа и целостности сессии и уровней конфиденциальности и целостности сущности. Уровни конфиденциальности и доступа могут выражаться произвольными неотрицательными целыми числами, а уровни целостности могут принимать только два значения — высокое (1) и низкое (0). Доступ сессии к сущности на запись считается возможным тогда и только тогда, когда уровень доступа сессии равен уровню конфиденциальности сущности (при этом более конфиденциальная информация не может попасть в менее конфиденциальную сущность) и уровень целостности сессии не меньше уровня целостности сущности (низкоцелостные сессии не могут изменять высокоцелостные сущности).

Спецификация функции task_kill (так же, как и соответствующего правила delete_subject) написана исходя из сформулированного правила, — поскольку возможность послать сигнал из одного процесса другому означает, что первый процесс способен изменить состояние второго процесса (например, завершить его выполнение), эта возможность должна контролироваться так же, как и доступ на запись. Таким образом, в формальной модели модуля LSM, имеющего функцию task_kill в интерфейсе, ее спецификация утверждает, что доступ на отправку сигнала из одного процесса другому предоставляется тогда и только тогда, когда первый процесс имеет тот же самый уровень доступа, что и второй, и первый процесс имеет не меньший уровень целостности, чем второй. Выражается это общим предикатом AccessIsPermitted(TaskLabel(killer), TaskLabel(task), {WriteA}), проверяющим нужные соотношения между уровнями доступа и целостности процессов killer и task при выдаче доступа на запись.

Рассмотрим Event-B спецификацию для LSM-интерфейса task_kill и ACSL-спецификацию реализации функции соответствующей task_kill в некотором модуле безопасности. Заметим, что пример

не является частью какой-либо операционной системы, многие детали, которые в реальной системе обязательно должны присутствовать, здесь опущены. Назовем функцию из модуля безопасности mylsm_kill_permission.

Контрольная функция task_kill разрешает либо запрещает посылать сигнал kill процессу (системный вызов kill) в зависимости от прав доступа процесса.

```
event task_kill
  any task killer
  where
    @grd1 task ∈ CurTasks
    @grd2 killer ∈ CurTasks
    @grd3 AccessIsPermitted((TaskLabel(killer))
                            (TaskLabel(task))
                            ({WriteA})) = TRUE
  then
    @act1 TaskTaskAccess(killer) :=
          TaskTaskAccess(killer) ∪ {task ↦ {WriteA}
end
```

Событие task_kill в спецификации Event-B имеет два входных аргумента:

- task — процесс, которому посылается сигнал kill;
- killer — процесс, который посылает сигнал kill.

В первых двух предусловиях (@grd1, @grd2) проверяется, что task и killer принадлежат множеству процессов, функционирующих в системе. В @grd3 описывается, что событие возможно, если предикат AccessIsPermitted выполняется. AccessIsPermitted имеет аргументами метку безопасности процесса killer, метку безопасности процесса task, и множество запрашиваемых доступов (в данном случае только доступ на запись WriteA).

Результат события task_kill описывается в конструкции @act1 — обновляется множество доступов для killer процесса, куда добавляется доступ WriteA к процессу task.

Неформальная документация к функции LSM-интерфейса task_kill выглядит следующим образом:

```
int (*task_kill)(struct task_struct *task, struct siginfo *info,
    int sig, const struct cred *cred);
```

Функция task_kill проверяет доступ перед посылкой сигнала sig процессу task. Параметр info может принимать значение NULL, может быть константой — 1 или же корректным указателем на структуру siginfo. Если параметр info имеет значение 1 или выполнено SI_FROMKERNEL(info), тогда считается, что сигнал посылается

ядром и в нормальном режиме функционирования доступ должен быть разрешен. Интерфейс должен возвращать 0, если доступ разрешен.

Параметры интерфейса:

- task — структура данных процесса, которому посылается сигнал;
- info — информация о сигнале;
- sig — значение сигнала (тип SIGHUP, SIGING, SIGKILL ...), 0 — если сигнал посылать не нужно, просто проверить существование процесса;
- cred — структура описывает полномочия процесса, который послал сигнал. Если параметр cred равен NULL, то это означает что сигнал послан текущим (current) процессом.

В данном примере мы отдельно не рассматриваем ситуацию, когда сигнал посылается процессу потоком ядра, а не другим процессом: в примере мы полагаем, что у потока ядра имеется такая же метка безопасности, как и у любого процесса пользовательского пространства. Ситуация, когда параметр сигнала sig равен нулю, в реальности могла бы рассматриваться как запрос доступа на чтение (ReadA). В целях упрощения примера эта ситуация также не рассматривается отдельным образом, а обрабатывается, как и все остальные сигналы.

Рассмотрим, как могла бы выглядеть ACSL-спецификация к данной функции:

```
/*@ requires valid_task_struct(task);
    requires valid_task_struct(current);
    requires task ≢ current ∧ task-> cred ≢ cred;
    requires valid_seclabel(TaskLabel(task));
    requires seclabel_get_requires(TaskLabel(task));
    requires valid_cred(cred) ∧
      (\let sbj_sec = (seclabel_t*)cred-> security;
      valid_seclabel(sbj_sec) ∧
      sbj_sec ≢ TaskLabel(task) ∧
      seclabel_get_requires(sbj_sec))
    ∨
      cred ≡ \null ∧
      (\let sbj_sec = TaskLabel(current);
      valid_seclabel(sbj_sec) ∧
      sbj_sec ≢ TaskLabel(task) ∧
      seclabel_get_requires(sbj_sec));
    assigns TaskLabel(task)-> ucnt,
      ((seclabel_t *)cred-> security)-> ucnt,
```

```
      TaskLabel(current)-> ucnt;
    behavior System:
    ensures seclabel_counter_unchanged{Pre,Post}
      (TaskLabel(task));
    ensures cred ≡ \null ⇒
      seclabel_counter_unchanged{Pre,Post} (TaskLabel(current));
    ensures valid_cred(cred) ⇒
      seclabel_counter_unchanged{Pre,Post}(cred-> security);
    ensures \result ≡ 0
        ∨\result ≡ -EPERM;
    behavior ModelCred:
    assumes cred ≢ \null;
    ensures \result ≡ 0 ⇔ AccessIsPermitted(
        i2m_Label((seclabel_t *)cred-> security),
        i2m_Label(TaskLabel(task)),
        MAY_WRITE);
    behavior ModelCurrent:
    assumes cred ≡ \null;
    ensures \result ≡ 0 ⇔ AccessIsPermitted(
        i2m_Label(TaskLabel(current)),
        i2m_Label(TaskLabel(task)),
        MAY_WRITE);
    disjoint behaviors ModelCred, ModelCurrent;
    complete behaviors ModelCred, ModelCurrent;
*/
int mylsm_kill_permission(struct task_struct *task,
      struct siginfo *info,
      int sig, const struct cred *cred);
```

В ACSL-контракте функции mylsm_kill_permission в предусловии требуется корректность (предикат valid_task_struct) структур данных struct task_struct, описывающих процессы. В целях упрощения примера мы будем считать, что процесс не может послать сигнал самому себе (это выражается третьим предусловием). Далее в предусловиях 4 и 5 требуется корректность метки безопасности (valid_seclabel) процесса task, возможность ее захвата (seclabel_get_requires) посредством увеличения счетчика использований. Логическая функция TaskLabel возвращает метку безопасности процесса. Эта функция имеет схожую семантику с отображением TaskLabel в Event-B спецификации. В последнем предусловии рассматриваются две возможные альтернативы: когда метка безопасности извлекается из аргумента cred, описывающего привилегии процесса пославшего сигнал, либо из структуры текущего (current) процесса. Также в

обоих ситуациях содержатся требования корректности структур меток (valid_seclabel), возможности захвата метки (seclabel_get_requires) и различия меток безопасности у процесса, посылающего сигнал, и процесса, которому сигнал предназначен. Последнее является следствием упрощения примера, когда процесс не может послать сигнал сам себе.

Далее в рамочных условиях (assigns) описывается участок памяти, в котором функция может изменять значения переменных. В данном случае описываются исключительно счетчики ссылок у структур данных. Другую память функция изменять не может.

Как уже было упомянуто выше, ACSL-спецификации в данной работе включают в себя сущности двух видов. Первые соответствуют типам, переменным, функциям и другим сущностям Event-B спецификации Си-функций. Вторые связаны с конкретной реализацией Си-функций, необходимы для описания деталей, представленных только на уровне реализации, и функциональности, не связанной с аспектами информационной безопасности Event-B спецификации. Для того чтобы привести требования Event-B спецификации и ACSL-спецификации к некоторому «общему знаменателю», сущности первого вида переносятся с Event-B спецификации и требования к их корректности формулируются при помощи тех же условий, которые были сформулированы в Event-B спецификации. Такое приведение выполняется при помощи функций, которые мы далее будем называть implementation-to-model (i2m). Префикс i2m используется в их именах.

В соответствии с этой схемой (разделения на реализационные спецификации и модельные) для всех функций постусловия разделяются на две части: System — требования, соответствующие семантике функции, не связанной с информационной безопасностью, и Model — здесь описываются требования, перенесенные с Event-B спецификации. Для доказательства корректности функции требуется соблюдение всех постусловий. Там, где нет разделения, считается, что нет требований, связанных с моделью.

В System постусловиях (первые три) от функции требуется, чтобы она освобождала счетчики ссылок у ресурсов после того, как они становятся ей не нужны. В последнем говорится, что результатом работы функции может быть либо 0 (доступ разрешен), либо -EPERM (доступ запрещен).

Model постусловия разбиваются на два: ModelCred и ModelCurrent. В ModelCred рассматривается ситуация, когда аргумент cred у функции имеет значение не NULL. Тогда метка безопасности процес-

са, который послал сигнал, берется из этой структуры. В ModelCurrent метка безопасности берется у текущего процесса. Постусловия из ModelCred и ModelCurrent не могут требоваться одновременно (директива disjoint), но при этом одно из них в зависимости от входных аргументов обязательно должно быть применимо к исходному коду (директива complete). В Model постусловиях утверждается, что функция корректным образом реализована тогда и только тогда, когда соблюдается предикат AccessIsPermitted. В спецификации ACSL предикат AccessIsPermitted принимает в двух первых аргументах метки безопасности модельного типа Label. Для того чтобы преобразовать структуру метки безопасности seclatel_t в модельную Label, используется специальная логическая implementation-to-model функция i2m_Label, которая будет рассмотрена подробно ниже.

В модуле LSM граф вызовов обработки task_kill выглядит следующим образом (рис. 6.6). Функция mylsm_task_permission вызывает несколько функций, одна из которых (access_is_permitted) имеет соответствие в Event-B спецификации (AccessIsPermited). Функции seclabel_get и seclabel_put отвечают за работу со счетчиком использования метки безопасности, они не несут модельной семантики. Функции atomic_inc (атомарная операция инкрементирования счетчика), atomic_dec_and_test (атомарное декрементирования счетчика с проверкой на ноль), kfree (освобождение ранее аллоцированной памяти) также являются функциями ядра.

На рис. 6.6 темно-серым цветом отмечены функции, у которых есть модельные спецификации, светло-серым — функции ядра. Прямоугольником отмечены функции модуля, к которым нет требований Event-B, к ним есть исключительно реализационные требования.

Реализации интерфейса task_kill (функция mylsm_kill_permission) в коде LSM-модуля может выглядеть следующим образом:

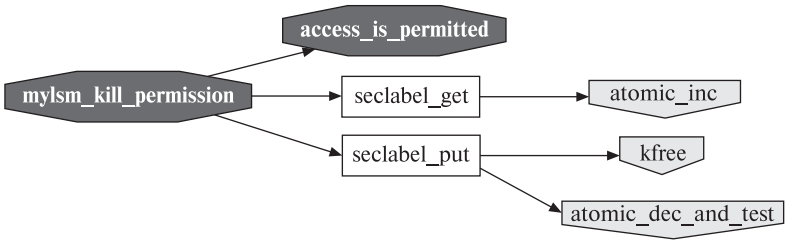

Рис. 6.6. Граф вызовов функции mylsm_kill_permission

```
int mylsm_kill_permission(struct task_struct *task,
      struct siginfo *info,
      int sig, const struct cred *cred)
{
  int rc;
  const seclabel_t *slbl_task, *slbl_killer;
  /* sig == 0 в примере не рассматривается как MAY_READ доступ */
  /* Считается, что метка безопасности всегда инициализирована */
  /* Subjective credentials: cred */
  if (cred) {
    slbl_killer = seclabel_get(cred-> security);
  } else {
    slbl_killer = seclabel_get(current_security()); /* cred */
  }
  /* Objective credentials: real_cred */
  slbl_task = seclabel_get(task_security(task));
  rc = access_is_permitted(slbl_killer, slbl_task, MAY_WRITE);
  seclabel_put(slbl_task);
  seclabel_put(slbl_killer);
  return rc;
}
```

Как уже упоминалось, в примере не рассматриваются ситуации, когда сигнал посылается ядром, когда аргумент sig имеет значение 0. В примере считается, что процесс не посылает сигнал себе сам, а метки безопасности у двух рассматриваемых процессов не совпадают на уровне указателей. Также в функции из примера не задействуется информация о сигнале (аргумент info) и типе сигнала (аргумент sig) при проверке доступа.

В функции mylsm_kill_permission сначала считываются метки безопасности процессов. Для того чтобы получить поле security структуры cred текущего процесса, используется макрос current_security. Для получения поля security структуры real_cred процесса task используется макрос task_security. В ядре Linux структура процесса (struct task_struct) имеет два поля для описания привилегий: cred и real_cred. Они имеют тип структуры (struct cred), в которой имеется поле security. В него модуль LSM записывает собственные метки безопасности. Наличие двух полей описания привилегий объясняется тем, что каждый процесс имеет субъектные (cred) привилегии (используется, когда он является субъектом доступа) и объектные (real_cred) привилегии (используется, когда процесс выступает как объект доступа) [87]. В модели нашего примера нет такого разделения для процессов. Соответственно, в реализации появляется

требование, чтобы поле security у cred и real_cred для всех процессов вело на одну и ту же метку безопасности. Это условие заложено в предикате корректности структуры valid_task_struct, определение которого будет рассмотрено далее.

Функция seclabel_get увеличивает у меток безопасности счетчик использования на единицу. В функции access_is_permitted осуществляется проверка доступа на запись к процессу task с меткой slbl_task процессом с меткой slbl_killer. Функция возвращает код rc, в котором содержится положительный либо отрицательный вердикт доступа. Функций seclabel_put освобождает ресурсы меток безопасности процессов (декрементируется счетчик использований и вызывает функцию kfree, если счетчик достигает нуля).

На уровне исходного кода видно, что AccessIsPermitted реализуется внутри функции access_is_permitted.

Далее рассмотрим TaskLabel. В Event-B TaskLabel реализуется как отображение:

$$@\texttt{TaskLabel\_type}\ \texttt{TaskLabelSubj} \in \texttt{CurTasks} \rightarrow \texttt{Labels}$$

Рассмотрим определение логической функции TaskLabel в ACSL:

```
/*@ axiomatic TaskLabel {
    predicate valid_cred(struct cred *cred) =
      \valid(cred) ∧
      (\typeof(cred-> security) <: \type(seclabel_t *)) ∧
      valid_seclabel((seclabel_t *)cred-> security) ∧
      0 < cred-> usage.counter;
    predicate valid_task_struct(struct task_struct* tsk) =
      \valid(tsk) ∧
      valid_cred(tsk-> cred) ∧
      valid_cred(tsk-> real_cred) ∧
      tsk-> cred-> security ≡ tsk-> real_cred-> security;
    logic seclabel_t *TaskLabel{L} (struct task_struct *t) =
      (seclabel_t *)(t-> cred-> security);
}
*/
```

Логическая функция TaskStruct имеет простое строение за счет того, что есть требование равенства меток безопасности у субъектных (cred) и объектных (real_cred) привилегий процесса. Помимо этого в последнем листинге приведены определения предикатов valid_cred и valid_task_struct. В предикате valid_cred требуется, чтобы поле security структуры cred всегда в коде рассматривалось как указатель на структуру seclabel_t, несмотря на то, что оно имеет тип

void *, а счетчик использований никогда не был равен нулю. Предикат valid_task_struct требует корректности структур cred и real_cred процесса, а также содержит требование равенства субъектных и объектных меток процесса.

Давайте рассмотрим спецификацию функции AccessIsPermitted Event-B:

```
@AccessIsPermitted_type
    AccessIsPermitted ∈ Labels → (Labels → (ℙ(Accesses) → BOOL))
@AccessIsPermitted_definition
    ∀ la,lo,m · la ∈ Labels ∧ lo ∈ Labels ∧ m ⊆ Accesses ⇒
    (((WriteA ∈ m)
    ∧ LabelCnfLevel(la) = LabelCnfLevel(lo)
    ∧ LabelCnfCategories(la) = LabelCnfCategories(lo)
    ∧ LabelIntLevel(la) ⩾ LabelIntLevel(lo))
      ⇔
    (AccessIsPermitted(la)(lo)(m) = TRUE))
```

Исходя из определения типа функции @AccessIsPermitted_type можно увидеть, что она принимает на вход две метки безопасности и множество запрашиваемых доступов, а возвращает TRUE или FALSE в зависимости от того, возможно ли предоставить доступ или нет.

В самом предикате оставлено описание только доступа на запись. В нашем примере существует только доступ на запись либо пустой доступ.

В спецификации Even-B AccessIsPermitted при доступе на запись проверяется эквивалентность уровней конфиденциальности в метках безопасности, соответствие категорий и наличие не меньшего уровня целостности у запрашивающего доступ субъекта. Уровни конфиденциальности и категории у метки безопасности связаны с мандатным управлением доступом (Mandatory Access Control, MAC), а уровень целостности — с мандатным контролем целостности (Mandatory Integrity Control, MIC).

Определение AccessIsPermitted на ACSL выглядит следующим образом:

```
/*@ axiomatic AccessIsPermitted {
    predicate write_bit(int mode) = (mode & MAY_WRITE);
    predicate valid_mode(int mode) = write_bit(mode) ∨ mode ≡ 0;
    predicate AccessIsPermitted{L} (Label s, Label o, int mode);
    axiom AccessIsPermitted_definition:
      ∀ int mode, Label s, Label o;
      valid_mode(mode) ∧
      write_bit(mode) ∧
```

```
      LabelCnfLevel(s) ≡ LabelCnfLevel(o) ∧
      LabelCnfCategorySet(s) ≡ LabelCnfCategorySet(o) ∧
      IntLevelGte(LabelIntLevel(s), LabelIntLevel(o))
        ⇔
      AccessIsPermitted(s, o, mode);
  }
*/
```

Аксиома @AccessIsPermitted_definition повторяет соответствующее определение из Event-B спецификации с точностью до требования корректности параметра mode. Требование корректности mode в Event-B спецификации выражается аксиомой:

```
@Accesses_partition Accesses = {WriteA}
```

В инструментах, работающих с ACSL, не поддерживается трансляция инвариантов типов, поэтому предикат valid_mode непосредственно используется в данной аксиоме. В ACSL на уровне языка не поддерживается перегрузка операторов сравнения для модельных типов, поэтому для сравнения уровней целостности используется предикат IntLevelGte.

Предикат AccessIsPermitted определен на модельном типе метки безопасности Label. Для того чтобы преобразовать реальную метку безопасности типа seclabel_t в модельную, используются логические функции i2m, в частности i2m_Label (использование которой мы уже видели ранее). Определения i2m будут рассмотрены чуть позже.

LabelCnfLevel как в Event-B спецификации, так и в ACSL-спецификации используется для того, чтобы получить из метки безопасности уровень конфиденциальности. LabelCnfCategories и LabelCnfCategorySet — для множества категорий. LabelIntLevel — для уровня целостности.

Прежде чем перейти к рассмотрению функции access_is_permitted, нужно рассмотреть определение метки безопасности в исходном коде.

```
typedef uint8_t ilev_t;
typedef uint8_t lev_t;
typedef uint64_t cat_t;
typedef struct {
  lev_t lev;
  ilev_t ilev;
  cat_t cat;
  atomic_t ucnt;
} seclabel_t;
```

Метка безопасности состоит из уровня конфиденциальности (lev), целостности (ilev) и категорий (cat). Дополнительно, в метке содержится счетчик использований ucnt.

```
predicate valid_seclabel(seclabel_t *l) =
  \valid(l) ∧
  \offset_min(l) ≡ 0 ∧
  0 < l-> ucnt.counter ∧
  (l-> ilev ≡ 0 ∨ l-> ilev ≡ 1);
```

В предикате корректности указателя метки безопасности требуется, чтобы счетчик использований не был равен нулю, уровни целостности могли иметь значение только 0 — низкий либо 1 — высокий. Требование offset_min говорит о том, что указатель содержит адрес начала выделенного блока памяти. Это необходимо для соблюдения предусловий функции освобождения памяти kfree.

Перейдем к рассмотрению функции access_is_permitted и ACSL-спецификации для нее. Функция реализует сравнение меток безопасности субъекта (s), запрашивающего доступ к субъекту (o). В нашем примере в иллюстративных целях рассматривается только доступ на запись (MAY_WRITE). Функция access_is_permitted должна вернуть 0 — если доступ разрешен, код ошибки — если запрещен.

```
/*@ requires valid_seclabel(s);
    requires valid_seclabel(o);
    requires valid_mode(mode);
    assigns \nothing;
    behavior System:
      ensures \result ≡ 0 ∨ \result ≡ -EPERM;
    behavior Model:
      ensures \result ≡ 0 ⇔ AccessIsPermitted(i2m_Label(s),
                                              i2m_Label(o),
                                              mode);
*/
int access_is_permitted(const seclabel_t *s, const seclabel_t *o,
int mode)
{
  if (mode & MAY_WRITE) {
    if (s-> lev == o-> lev
    && s-> cat == o-> cat
    && s-> ilev > = o-> ilev) {
      return 0;
    }
  }
```

```
  return -EPERM;
}
```

В ACSL-спецификации функции в предусловиях требуется корректность рассматриваемых меток безопасности и вида доступа. В рамочном условии обозначается, что функция не имеет побочных эффектов. В реализационных постусловиях обозначаются возможные возвращаемые значения. В модельных постусловиях обозначается, что функция разрешает доступ тогда и только тогда, когда предикат AccessIsPermitted выполнен. Из этих двух постусловий следует, что возвращаемый результат может быть равен -EPERM только тогда, когда AccessIsPermitted не соблюдается.

Для того чтобы использовать предикат AccessIsPermitted, определенный на модельном типе метки безопасности Label, в постусловии данной функции используется логическая функция i2m_Label. Она принимает на вход метку безопасности типа seclabel_t, а возвращает модельную метку типа Label.

Перейдем к определению модельной метки безопасности Label:

```
/*@ axiomatic ModelTypesTheory {
    type Label;
    type CnfLevel = unsigned int;
    type Integrity = LowInt | HighInt;
    predicate IntLevelLt(Integrity i1, Integrity i2) =
      i1 ≡ LowInt ∧ i2 ≡ HighInt;
    predicate IntLevelGt(Integrity i1, Integrity i2) =
      i1 ≡ HighInt ∧ i2 ≡ LowInt;
    predicate IntLevelLte(Integrity i1, Integrity i2) =
      IntLevelLt(i1,i2) ∨ i1 ≡ i2;
    predicate IntLevelGte(Integrity i1, Integrity i2) =
      IntLevelGt(i1,i2) ∨ i1 ≡ i2;
    logic Label newLabel(
        CnfLevel l,
        Integrity il,
        CnfCategorySet c
      );
    logic CnfLevel LabelCnfLevel(Label l);
    logic Integrity LabelIntLevel(Label l);
    logic CnfCategorySet LabelCnfCategorySet(Label l);
    axiom LabelFieldAccessCnfLevel:
      ∀ CnfLevel l,
      Integrity il,
      CnfCategorySet c;
      l ≡ LabelCnfLevel(newLabel(l, il, c));
```

```
    axiom LabelFieldAccessIntLevel:
      ∀ CnfLevel l,
       Integrity il,
       CnfCategorySet c;
      il ≡ LabelIntLevel(newLabel(l, il, c));
    axiom LabelFieldAccessCnfCategorySet:
      ∀ CnfLevel l,
       Integrity il,
       CnfCategorySet c;
      c ≡ LabelCnfCategorySet(newLabel(l, il, c));
}
*/
```

В аксиоматике ModelTypesTheory вводится неявное определение типа Label, без конкретизации его внутреннего устройства. Модельный тип уровня конфиденциальности CnfLevel представляет собой тип данных языка Си unsigned int. Модельный тип уровня целостности представляет собой перечисление. Множество его возможных значений — LowInt и HighInt, в полном соответствии с Event-B спецификацией.

```
@Integrity_partition partition(Integrity, {LowInt}, {HighInt})
```

Операции сравнения уровней целостности реализуются через предикаты IntLevelLt*, IntLevelGt*: в языке ACSL нет перегрузки операторов.

В Event-B спецификации тип CnfLevel определяется как непустое подмножество натуральных чисел, включая 0 (для Event-B $\mathbb{N}$ включает ноль).

```
@CnfLevel_type CnfLevel ⊆ ℕ ∧ CnfLevel ≠ ∅
```

В реальности нет необходимости моделировать уровни конфиденциальности большими числами, не помещающимися в стандартные целочисленные типы. Диапазона значений типа unsigned вполне достаточно.

Определение типа CnfCategorySet вводится отдельно и будет рассмотрено в следующем листинге.

В аксиоматике ModelTypesTheory вводится логическая функция newLabel — конструктор модельной метки Label. Ее семантика не конкретизируется, а определяется неявно через аксиомы.

Следующие три логические функции LabelCnfLevel, LabelIntLevel, LabelCnfCategorySet — функции для доступа к сущностям модельной метки безопасности. Взаимодействие newLabel и этих функций определяется в последующих аксиомах, которые имеют

одинаковое строение и не заслуживают детального рассмотрения ввиду их очевидности.

```
@LabelCnfLevl_type LabelCnfLevel ∈ Labels → CnfLevel
@LabelIntLevel_type LabelIntLevel ∈ Labels → Integrity
@LabelCnfCategories_type LabelCnfCategories ∈ Labels →
    ℙ (CnfCategories)
```

В Event-B данные функции имеют схожее определение тотального отображения из метки безопасности в соответствующий атрибут метки безопасности. LabelCnfCategories возвращает множество категорий метки безопасности. В инструментах, работающих с ACSL-спецификациями, нет поддержки моделирования множеств, поэтому в ACSL-спецификации множество меток безопасности вводится через тип CnfCategoriesSet через аксиоматику CnfCategorySetTheory. Давайте ее рассмотрим:

```
/*@ axiomatic CnfCategorySetTheory {
    type CnfCategory = int;
    type CnfCategorySet;
    logic CnfCategorySet EmptyCnfCategorySet;
    logic CnfCategorySet addCnfCategory(CnfCategorySet s,
                                        CnfCategory c);
    logic CnfCategorySet intersectCnfCategorySet
                                        (CnfCategorySet s1,
                                        CnfCategorySet s2);
    logic boolean isinCnfCategorySet(CnfCategorySet s,
                                        CnfCategory c);
    logic int sizeCnfCategorySet(CnfCategorySet s);
  axiom add2Set:
    ∀ CnfCategorySet s, CnfCategory c;
    addCnfCategory(addCnfCategory(s,c),c) ≡ addCnfCategory(s,c);
  axiom isinEmptySet:
    ∀ CnfCategory c;
    isinCnfCategorySet(EmptyCnfCategorySet,c) ≡ \false;
  axiom isinAddCnfCategory1:
    ∀ CnfCategorySet s, CnfCategory c;
    isinCnfCategorySet(addCnfCategory(s,c),c);
  axiom intersect_def0:
    ∀ CnfCategorySet s;
    intersectCnfCategorySet(EmptyCnfCategorySet,s) ≡
      EmptyCnfCategorySet;
  axiom intersect_def1:
    ∀ CnfCategorySet s1, CnfCategorySet s2, CnfCategory c;
    isinCnfCategorySet(s1,c) ∧ isinCnfCategorySet(s2,c)
      ⇔
    isinCnfCategorySet(intersectCnfCategorySet(s1,s2),c);
```

```
  axiom sizeCnfCategorySet_def0:
    sizeCnfCategorySet(EmptyCnfCategorySet) ≡ 0;
}
*/
```

В данной спецификации вводится модельный тип отдельной категории CnfCategory, который соответствует реальному типу int для удобства работы с ним. Вводится модельный тип множества меток CnfCategorySet и функции addCnfCategory (добавление категории в множество), intersectCategorySet (пересечение множеств), isinCnfCategorySet (проверки принадлежности категории множеству), sizeCnfCategorySet (мощность множества), EmptyCnfCategorySet (логическая функция, которая всегда возвращает пустое множество). Данные функции задаются аксиоматически. В целях наглядности число аксиом и лемм в листинге было существенно сокращено: в нашем примере эти логические функции никак не задействованы в формализации модельных требований.

К настоящему моменту мы рассмотрели как выглядят спецификации Event-B к интерфейсу task_kill, как они выглядят в ACSL-нотации, рассмотрели, как типы Event-B моделируются в ACSL. Необходимо рассмотреть, как типы языка Си преобразуются в модельные типы ACSL, т. е. перейти к рассмотрению implementation-to-model функций.

```
/*@ axiomatic i2m_ModelTypes {
    logic Integrity i2m_IntLevel(ilev_t il) = (il ≡ 0) ?
      LowInt : HighInt;
    logic CnfCategorySet i2m_CnfCategorySet64(cat_t c, int n) =
      n ≡ 0 ?
        EmptyCnfCategorySet
      :
        \let cnfset = i2m_CnfCategorySet64(
          (cat_t) (c ≪ (cat_t) 1),
            (int) (n - (int) 1));
        ((c & (cat_t) 0x1) ≡ 1) ?
          addCnfCategory(cnfset, (int) ((int) 64 - n))
       :
         cnfset;
    logic CnfCategorySet i2m_CnfCategorySet(cat_t c) =
                         i2m_CnfCategorySet64(c, (int) 64);
    logic Label i2m_Label{L} (seclabel_t *l) =
      newLabel(
          l-> lev,
          i2m_IntLevel(l-> ilev),
```

```
            i2m_CnfCategorySet(l-> cat)
        );
}
*/
```

Для начала рассмотрим последнее определение логической функции i2m_Label. Она задается явным образом. Принимает на вход Си-метку безопасности seclabel_t и возвращает модельную метку типа Label. Внутри себя она вызывает конструктор модельного типа newLabel. В качестве значений последний принимает аргументы уровня конфиденциальности (модельный тип LabelCnfLevel), уровня целостности (LabelIntLevel), множества категорий (LabelCnfCategorySet). Модельный уровень конфиденциальности один-в-один отображается в модельный тип LabelCnfLevel и не требует специального преобразования, потому что LabelCnfLevel определяется как alias Си-типа unsigned int. Для отображения уровня целостности и множества категорий в модельные используются функции i2m_IntLevel и i2m_CnfCategorySet.

i2m_IntLevel ставит в соответствие нулю — низкий уровень целостности, ненулевому значению — высокий уровень. В предикате корректности метки безопасности (valid_seclabel) есть требование что целостность может иметь значения только 0 и 1.

Множество категорий в коде представляется битовыми полями. В примере возможно 64 различных категорий — вместимость типа cat_t. Если в переменной типа cat_t выставлен определенный бит — это значит что метка безопасности содержит в себе эту категорию. Логическая функций i2m_CnfCategorySet (через вспомогательную функцию i2m_CnfCategorySet64) реализует рекурсивную проверку битов в cat_t и пополняет множество категорий через addCnfCategory, если бит выставлен.

Для доказательства определенных требований иногда нужны дополнительные утверждения о взаимодействии уровня релизации и модельного представления. В нашем примере таким утверждением является лемма о том, что неравенство множеств категорий cat_t влечет за собой неравенство модельных множеств CnfCategorySet:

```
/*@ axiomatic EqualCategories {
    lemma EqualCategories:
      ∀ cat_t scat, cat_t ocat;
      (scat ≠ ocat) ⇒
      (i2m_CnfCategorySet(scat) ≠ i2m_CnfCategorySet(ocat));
}
*/
```

Таким образом, нами был рассмотрен пример реализации функции LSM-интерфейса task_kill. В примере демонстрируется, как спецификации Event-B отображаются в спецификации ACSL, рассматриваются модельные типы на ACSL, показывается их связь с реализацией через implementation-to-model функции. Соответствие ACSL-спецификаций спецификациям Event-B формальным образом не обосновывается.

7 Динамический мониторинг

В этой главе рассматривается вопрос интегральной проверки соответствия поведения ядра Linux требованиям модели политики безопасности управления доступом (например, МРОСЛ ДП-модели). Соответствие проверяется посредством сверки результатов реальных операций по запросу доступа при помощи системных вызовов ядра ОС с правилами предоставления доступа (или отказа в доступе) модели политики безопасности. Для этого во время выполнения приложений (программ пользовательского пространства) в динамике трассируются системные вызовы, фиксируются их параметры и результаты. Далее на основе полученных данных воспроизводится обработка запросов доступа в ядре на формальной функциональной спецификации на языке Event-B (ФСП). На функциональной спецификации проверяется допустимость результатов обработки системного вызова.

Данный подход наиболее целесообразно применять в рамках выполнения требований класса доверия «Тестирование» ГОСТ Р ИСО/МЭК 15408-3, который требует проведения тестирования ОО на соответствие функциональной спецификации (см. семейство доверия ATE_COV «Покрытие»). Автоматический сбор трассы выполнения и проведение проверки её соответствия формальной функциональной спецификации на языке Event-B позволяет избавиться от необходимости вручную вычислять ожидаемый результат каждой операции, что, в свою очередь, даёт возможность значительно повысить число и разнообразие проводимых тестов. Сбор информации о покрытии различных ситуаций в формальной функциональной спецификации также позволяет автоматизировать оценку покрытия интерфейсов функций безопасности, что является еще одним требованием семейства доверия ATE_COV «Покрытие».

7.1. Управление доступом в ядре ОС Linux

Предварительно рассмотрим несколько подробнее схему работы ядра ОС в случае, когда необходимо провести анализ возможности

предоставления того или иного вида доступа к некоторому ресурсу. Это позволит нам обосновать утверждение, что динамический мониторинг является необходимым этапом процесса верификации механизма управления доступом в ОССН.

Программы пользовательского пространства осуществляют работу с внешними, по отношению к ним, ресурсами через ядро ОССН. Запросы на доступы выполняются посредством системных вызовов. Когда программа делает системный вызов, например, на открытие файла или сокета, ядро ОС выделяет соответствующий ресурс (например файловый дескриптор) и прикрепляет его к дескриптору процесса. Перед тем как ресурс будет выделен, делается большое число проверок на правильность аргументов в системном вызове, на существование запрашиваемых ресурсов, на возможность выделения этих ресурсов (например, хватает ли памяти), на возможность доступа к этим ресурсам (см. рис. 6.1). Если все предварительные проверки проходят и модуль LSM также разрешает доступ, то ресурс выделяется. В этой схеме важно еще раз отметить то, что управление до LSM-интерфейса может не дойти по очень большому числу причин. Соответственно, даже если системный вызов предполагает проверку LSM-модулем на какой-то стадии, то в общем случае осуществление системного вызова (запрос на доступ к ресурсам) не означает, что этот запрос попадает на обработку в LSM-модуль.

ФСП не моделирует таких проверок из ядра, как возможность выделения ресурса: хватает ли памяти, не превышено ли ограничение на число открытых файлов и др. Соответственно, в реальности доступ к ресурсу может быть не предоставлен по причине нехватки ресурсов машины, хотя функциональная спецификация его разрешает. В ФСП ресурсы полагаются неограниченными. В ОС, где, например, ситуация нехватки ресурсов вполне может сложиться, отказ на выполнение той или иной операции доступа по этой причине может считаться нормой, т. е. не всегда считается ошибочным. В контексте проверки корректности реализации интерфейсов ОС по отношению к ФСП (а следовательно, и к модели политики управления доступом) как ошибка рассматривается ситуация, когда модель политики безопасности доступ запрещает, а в реальности он по какой-то причине был предоставлен.

Подобная ошибка может произойти по разным причинам. Первая причина — ошибка обусловлена некорректной функциональностью модуля LSM. Это означает, что модуль неправильно реализует модель политики безопасности. Стадия дедуктивной верификации исходного кода модуля LSM, описанная в главе 6, нужна для то-

го, чтобы исключить подобные ошибки. Вторая причина ошибки — ядро в какой-то момент не вызывает интерфейс LSM [88–90], хотя это необходимо сделать. Либо LSM-интерфейс может быть не достаточно полным, в нем может не хватать некоторых функций для проверки определенных ситуаций [91, 92].

Помимо этого, стоит отметить, что дедуктивная верификация — это статический анализ кода. Его точность зависит, в том числе, от самих инструментов анализа и от того, насколько модели, лежащие в основе анализа, полны и адекватны. Так, на текущий момент инструменты дедуктивной верификации (Frama-C и AstraVer) не моделируют стек (см. главу 6). В предусловиях к функциям нет требования на размер доступной памяти в стеке, необходимой для корректной работы функции. Также не моделируются еще некоторые ошибки, например не различается память, доступная только на чтение или на запись и чтение. Дедуктивная верификация анализирует тексты исходных кодов, в то время как на процессоре выполняется бинарный код, который был получен из исходного посредством компиляции. То есть мы не имеем гарантий того, что бинарный код семантически эквивалентен исходному [93]. Все перечисленные моменты являются ограничениями дедуктивной верификации и являются аргументами в пользу необходимости интегральной проверки соответствия поведения ядра ОС требованиям модели политики безопасности управления доступом в динамике, на основе анализа трасс выполнения системных вызовов.

Для динамической проверки необходим тестовый набор, который воспроизводит разнообразные ситуации, связанные с запросами на доступ. Помимо этого, должен быть реализован «тестовый оракул» для проверки соответствия результатов работы системы ее функциональной спецификации как минимум в той части, которая определяется моделью политики безопасности управления доступом. В данной работе мы не рассматриваем вопрос построения такого тестового набора и вопрос критериев его полноты. Далее описывается сам метод динамической проверки, т. е. рассматриваются два вопроса: как получить данные о поведении ОО (трассу) и как выполнить динамический анализ трассы, т. е. как должен быть построен тестовый оракул.

7.2. Схема работы анализа

Схема работы мониторинга разбивается на две последовательных стадии — сбора информации о поведении ОО и ее анализ.

На первой стадии осуществляется сбор трассы выполнения ядра Linux с модулем защиты в отдельную базу событий в человекочитаемом формате. В ОС в этот момент либо запускается специальный тестовый набор, либо система работает в стандартном режиме (работают некоторые пользовательские приложения). Важно уточнить, что ядро ОС в этот момент работает в одноядерном режиме, без поддержки мультипроцессорности и без режима вытеснения.

Если мониторинг запускается без тестового набора, то в момент старта анализа в трассу записывается информация о глобальном состоянии ядра (модель состояния). Сюда попадает, например, информация о числе работающих процессов в системе, открытых ими файлах и соединениях. Для режима анализа с тестовым набором этого делать не требуется, так как влияние остальных процессов системы на тесты полагается минимальным и вся необходимая информация о начальном состоянии содержится в тестовом наборе.

Далее в трассу собирается информация об обработке системных вызовов ядром. Для работы с тестовым набором собирается информация только о системных вызовах от тестов, в ином случае рассматриваются все процессы. В трассу попадает информация об аргументах системного вызова, о результатах его обработки ядром с кодом возврата. Дополнительно в трассу может записываться информация, которая помогает отобразить аргументы системных вызовов в вид, пригодный для использования при сравнении с ФСП.

В момент завершения анализа в трассу записывается информация о глобальном состоянии ядра аналогично тому, как это делается при старте анализа.

На второй стадии осуществляется анализ собранной трассы с целью выявления в ней ошибок предоставления доступа. Для этого обработка системных вызовов из собранной трассы перепроверяется на ФСП, т. е. обработка системных вызовов воспроизводится на ФСП. Результаты обработки системных вызовов в ядре и в спецификации сверяются. Здесь необходимо помнить, что в ФСП (см. главу 5) задаются события старта и конца обработки для каждого системного вызова, вводятся промежуточные события между событиями Event-B спецификации МРОСЛ ДП-модели и порядок их выполнения. Событие начала обработки системного вызова принимает на вход аргументы системного вызова. ФСП разрабатывается так, что при заданных аргументах системного вызова и состоянии системы из каждого события существует не более одного возможного следующего события обработки системного вызова, не являющегося событием возврата кода ошибки. Это означает, что существует не

более одного пути в графовом представлении цепочки событий обработки системного вызова из начального события системного вызова в его конечное событие. В момент начала анализа трассы состояние формальной функциональной спецификации инициализируется в соответствии с записанной информацией о состоянии ядра ОС на момент старта мониторинга. В случае, если такая информация не собиралась, используется стандартная инициализация состояния спецификации.

Далее последовательно анализируются системные вызовы из собранной трассы. Воспроизведение обработки системного вызова на ФСП выглядит следующим образом:

1) в событие начала обработки данного системного вызова поступают аргументы с реальной системы (берутся из трассы). В соответствии со спецификацией вычисляется следующее событие, обновляется состояние ФСП;

2) если обработка следующего события оказалась невозможной, так как его охранные условия не выполнены, то считается, что с точки зрения ФСП доступ к ресурсу в результате обработки системного вызова не мог быть предоставлен;

3) если охранные условия для следующего события выполнены и оно не является уточняющим для вышележащего слоя спецификации, то его обработка происходит так же, как в пункте 1;

4) если охранные условия для следующего события выполнены и оно является уточняющим для вышележащего слоя спецификации, то происходит проверка охранных условий всех вышележащих уровней уточнений. В случае успешного прохождения последней состояние спецификации обновляется в соответствии с правилами перехода системы из состояния в состояние. Если же условия не выполнены, то считается, что доступ к запрашиваемому ресурсу не мог быть предоставлен в соответствии с моделью политики безопасности;

5) в случае, если обработка системного вызова на ФСП дошла до события завершения системного вызова и охранные условия для него были выполнены, то считается, что с точки зрения ФСП и модели политики безопасности доступ к ресурсу может быть предоставлен (системный вызов может быть успешно обработан).

Следующим шагом сверяются результаты (возвращаемые коды) обработки системного вызова из трассы и из ФСП:

1) когда оба результата положительны (нет ошибки, доступ запрещен), осуществляется переход к анализу следующего системного вызова из трассы;

2) когда результат в трассе отрицательный (доступ запрещен), а на ФСП положительный (доступ разрешен), то рассматривается код ошибки системного вызова. В коде ошибки содержится информация о причине неудачи обработки системного вызова. После рассмотрения «расхождения» на ФСП происходит возврат к модельному состоянию «до начала обработки» данного системного вызова (на момент вызова события его обработки), и начинается обработка следующего системного вызова из трассы. Если причина неудачи обработки системного вызова в ядре Linux заключается в:

- нехватке ресурсов физической машины, то не производится никаких дополнительных действий;
- в «некорректных» аргументах системного вызова, то с большой вероятностью это сигнализирует о том, что ФСП недостаточно полна. Данное расхождение записывается в журнал «аномалий»;
- нехватке разрешений (для доступа), то с большой вероятностью это сигнализирует о наличии ошибки в ядре и модуле безопасности ядра (слишком сильное ограничение). Данное расхождение записывается в журнал «аномалий»;

3) когда результат в трассе положительный, а на ФСП отрицательный, данная ситуация свидетельствует об ошибке в ядре и модуле безопасности ядра. При этом расхождении дальнейший анализ трассы теряет смысл. Анализ останавливается и сигнализирует о найденной ошибке предоставления доступа;

4) когда оба результата отрицательны (ошибка, доступ разрешен), осуществляется переход к анализу следующего системного вызова из трассы;

5) если анализ трассы дошел до ее конца, то производится сверка состояний спецификации и ядра. Расхождения записываются в журнал «аномалий».

Результатом работы анализа является журнал «аномалий», в котором содержатся пункты расхождения поведения реальной системы с ФСП. Данные журнала анализируются ручным образом. По результатам выявляются ошибки либо в исходном коде ядра и модуля безопасности, либо в ФСП.

В этом разделе была рассмотрена общая схема работы анализатора, состоящая из двух последовательных этапов: сбор трасс и их анализ на ФСП. Далее будут рассмотрены механизмы осуществления данных этапов.

7.3. Сбор трасс ядра

Для мониторинга ядра Linux существует большое число инструментов [94–100]. Одним из наиболее удобных является SystemTap. Он позволяет относительно легко логировать, как обработку системных вызовов, так и иные события на работающем ядре Linux.

SystemTap поддерживает специальный язык (похожий на скриптовый) для описания точек мониторинга и сбора информации о работающей системе. При этом инструментом предоставляется возможность обращения к практически любой информации в точке мониторинга: глобальные и локальные переменные, стек вызовов и др.

Описание скрипта мониторинга преобразуется инструментом в код загружаемого модуля ядра Linux на языке Си. Для мониторинга данный код компилируется и динамическим образом загружается в ядро, что не требует перезагрузки системы. Инструмент предоставляет наиболее богатые возможности описания точек мониторинга, если ядро Linux собрано с отладочной информацией, что ведет к дополнительным накладным расходам и частично объясняет разбиение анализа на два последовательных этапа.

SystemTap имеет много уже готовых описаний точек мониторинга (probe points). Например, для мониторинга большинства системных вызовов, их аргументов и результатов их обработки на одноядерной машине возможно использовать простой скрипт:

Листинг 7.1. Скрипт мониторинга системных вызовов SystemTap

```
probe syscall.* {
  printf("%s: %s(%s)",
    execname(), name, argstr)
}
probe syscall.*.return {
  printf("res: %s\n", retstr)
}
```

Пример выводимого лога приведен в табл. 7.1.

Для воспроизведения поведения ядра на ФСП в трассу необходимо собирать информацию об аргументах системных вызовов, результатах их обработки, дополнительную информацию, которая позволила бы отображать состояние структур данных ядра на состояние структур данных спецификации, глобальное состояние структур данных ядра на момент начала и конца мониторинга.

В ядре Linux из модулей могут напрямую вызываться функции системных вызовов (то есть, условно говоря, системный вызов при-

Таблица 7.1
Пример выводимого лога скрипта мониторинга системных вызовов

Процесс	Системный вызов	Аргументы	Код возврата
konsole	write	95, «\0», 1	1
ksmserver	ioctl	28, 21531, 0x7fffab1c9ec4	0
ksmserver	read	28, 0x5626d6785938, 40	40
konsole	lseek	94, -2147482516, SEEK_SET	-22 (EINVAL)
konsole	write	2, «HistoryFile::add.seek: Invalid argument\n», 40	40
konsole	lseek	93, 22447548, SEEK_SET	22447548
konsole	write	93, «l\004\0\200», 4	4

ходит не от приложения, а от самого ядра) и таким образом происходит вмешательство в пользовательское окружение со стороны ядра, которое в данном случае может рассматриваться как действие привилегированного процесса. Сама по себе такая ситуация происходит достаточно редко и по вполне определенным причинам (например, coredump, OOM Killer). В том случае, когда мы контролируем тестовый набор, подобного рода события могут не обрабатываться специальным образом, так как они заранее известны.

7.4. Механизм воспроизведения трасс на Event-B спецификации

Инструменты для работы с Event-B спецификациями [44] не предоставляют возможности проверки допустимости определенной последовательности событий. Для этого требуется интерпретатор спецификации. Добиться аналогичного результата возможно, если осуществить трансляцию Event-B спецификации в нотацию одного из интерпретируемых или исполняемых языков.

Существующие трансляторы Event-B в исполняемый код [101–103] достаточно узкоспециализированы и в полной мере возможнос-

тей, необходимых для воспроизведения трасс, не предоставляют.

Для анализа трассы системных вызовов на воспроизводимость на ФСП используется разработанный авторами транслятор Event-B спецификации в язык Python. Правила трансляции операций аналогичны тем, которые используются в вышеприведенных работах. При трансляции задаются две модели управления: когда на вход подается трасса, где уже полностью записаны переходы между событиями спецификации, и вторая модель — когда осуществляется подбор следующего события из текущего (для ФСП).

Статическая часть спецификации Event-B транслируется в систему типов (отображения, множества, перечисления), константы, а динамическая часть в функции. Переменные состояния спецификации в соответствии с их инвариантами транслируются в глобальные переменные Python:

Листинг 7.2. Переменные состояния спецификации на Event-B

```
variables
  CurrUnion
  SubjectUser
invariants
  @CurrUnionType
    CurrUnion ⊆ Union
  @SubjectUserType
    SubjectUser ∈ Subjects → UserAccs
```

Листинг 7.3. Трансляция переменных состояния в Python

```
CurrUnion = set()
SubjectUser = dict()
```

Инварианты спецификации транслируются в функции проверки глобальных переменных, которые вызываются после каждого события спецификации:

Листинг 7.4. Инварианты Event-B спецификации

```
invariants
  @CurrUnionPartition
    partition(CurrUnion, UserAccs, Subjects, Entities, Roles)
  @UserAccsAreNotEmpty
    UserAccs ≠ ∅
  @SubjectsAreNotEmpty
    Subjects ≠ ∅
  @SubjectAccessesType
    SubjectAccesses ∈ Subjects → (Entities ⇔ Accesses)
```

Листинг 7.5. Трансляция инвариантов в Python

```
@invariant
def CurrUnionPartiton() -> bool:
  return (CurrUnion == UserAccs | Subjects | Entities) and \
       not (UserAccs & Subjects) and \
       not (UserAccs & Entities) and \
       not (Subjects & Entities)
@invariant
def UserAccsAreNotEmpty() -> bool:
  return bool(UserAccs)
@invariant
def SubjectsAreNotEmpty() -> bool:
  return bool(Subjects)
@invariant
def SubjectAccessesType() -> bool:
  return all(map(lambda s: s in Subjects and all(map(lambda j: j[0]
  in Entities and type(j[1]) is Accesses, SubjectAccesses[s])),
  SubjectAccesses))
```

События транслируются в соответствующие функции. Охранные условия события транслируются в проверки внутри функции, а действия в событии преобразуются в операции обновления глобальных переменных. Уровни уточнений реализуются через механизм наследования в языке Python.

Листинг 7.6. Пример транслируемых Event-B событий

```
event delete_user
  any user
  where
    @grd1 user ∈ UserAccs
    @grd2 ∀ s · s ∈ Subjects ⇒ SubjectUser(s) ≠ user
  then
    @act1 CurrUnion := CurrUnion \ {user}
    @act2 UserAccs := UserAccs \ {user}
end
event create_object
  any subject object parent name dLabel mountPoint depth
  where
    @grd1 object ∈ Union \ CurrUnion
    @grd2 subject ∈ Subjects
    @grd3 parent ∈ Containers
    @grd4 parent ↦ WriteA ∈ SubjectAccesses(subject)
    @grd5 name ∈ Names
    @grd6 ∀ e · e ∈ dom(EntityNames) ⇒ parent ↦ name ∉
      EntityNames(e)
```

```
    @grd7 mountPoint ∈ Containers
    @grd8 dLabel ∈ BOOL
    @grd9 ∀ e · e ∈ dom(EntityNames) ∧ parent ∈ dom(EntityNames(e))
      ⇒ Direct(e) = dLabel
    @grd10 dLabel = TRUE ⇒ Direct(parent) = TRUE
    @grd11 dLabel = TRUE ⇒ mountPoint = Root
    @grd12 dLabel = FALSE∧ Direct(parent) = FALSE
      ⇒ mountPoint = EntityMP(parent)
    @grd13 dLabel = FALSE ∧ Direct(parent) = TRUE ⇒ mountPoint =
parent
  then
    @act1 CurrUnion := CurrUnion ∪ {object}
    @act2 Entities := Entities ∪ {object}
    @act3 Objects := Objects ∪ {object}
    @act4 EntityNames(object) := {parent ↦ name}
    @act5 Direct(object) := dLabel
    @act6 EntityMP(object) := mountPoint
end
```

Листинг 7.7. Пример результата трансляции Event-B событий в Python

```
@event
def delete_user(user):
  global CurrUnion
  global UserAccs
  assert user in UserAccs, "grd1"
  assert all(map(lambda s: SubjectUser[s] != user, Subjects)),
      "grd2"
  CurrUnion.discard(user)
  UserAccs.discard(user)
@event
def create_object(subject, object: Union, parent, name:
Names, dLabel: bool, mountPoint):
  global CurrUnion
  global Entities
  global Objects
  global EntityNames
  global Direct
  global EntityMP
  assert object not in CurrUnion, "grd1"
  assert subject in Subjects, "grd2"
  assert parent in Containers, "grd3"
  assert (parent, WriteA) in SubjectAccesses[subject], "grd4"
  assert all(map(lambda e: (parent, name) not in EntityNames[e],
```

```
dom(EntityNames))), "grd6"
  assert mountPoint in Containers, "grd7"
  assert all(map(lambda e: not (parent in dom(EntityNames[e])) or
Direct[e] == dLabel, dom(EntityNames))), "grd9"
  assert not(dLabel == True) or Direct[parent] == True, "grd10"
  assert not(dLabel == True) or mountPoint == Root, "grd11"
  assert not(dLabel == False and Direct[parent] == False) or
      (mountPoint == EntityMP[parent]), "grd12"
  assert not(dLabel == False and Direct[parent] == True) or
      (mountPoint == parent), "grd13"
  CurrUnion.add(object)
  Entities.add(object)
  Objects.add(object)
  EntityNames[object] = {(parent, name)}
  Direct[object] = dLabel
  EntityMP[object] = mountPoint
```

7.5. Интеграция дедуктивной верификации и динамического мониторинга при верификации модуля безопасности

За рамками данной главы остается вопрос трансляции ACSL-спецификаций в проверки времени выполнения кода (assertions).

Дедуктивная верификация основывается на следующих предположений.

1. Предположение о корректной работе компилятора. Процесс дедуктивной верификации доказывает соответствие исходного кода требованиям. В дальнейшем исходный код преобразуется транслятором/компилятором в бинарный. Дедуктивная верификация опирается на то предположение, что трансляция кода в бинарный осуществляется корректным образом. Отталкиваясь от этого предположения, можно сказать, что требования по безопасности соблюдаются и на бинарном коде, т. е. том коде, что непосредственно выполняется на процессоре.

2. Предположение о корректной работе инструментов дедуктивной верификации. Ошибки в инструментах дедуктивной верификации не являются чем-то исключительным и могут приводить к разнообразным последствиям. Большинство из них удается выявить на этапе работы с инструментами, так как они имеют видимые проявления. Например, инструменты не запускаются на коде, порождаемые логические формулы не доказываются. К сожалению, нельзя исключать возможности ошибок в инструментах, которые

приводили бы к тому, что порождаемые условия верификации доказываются, хотя на самом этого происходить не должно (по причине ошибки в коде/требованиях).

3. Предположение о полноте спецификаций внешних интерфейсов модуля (интерфейсов ядра). Как правило, размер модуля безопасности ядра ограничивается несколькими тысячами строк кода. Размеры сердцевины ядра, как и остальных модулей ядра, используемых при работе, в сумме могут давать несколько миллионов строк кода [104]. Модуль безопасности ядра взаимодействует с сердцевиной посредством интерфейса LSM (сердцевина $\rightarrow$ модуль) и через экспортируемые функции ядра (модуль $\rightarrow$ сердцевина). Доказательство корректности экспортируемых ядром функций не представляется возможным, так как для этого, по меньшей мере, потребовалось бы доказательство корректности сердцевины ядра. Соответственно, для задействованных модулем функций ядра составляются спецификации без их доказательства, для функций интерфейса LSM составляются спецификации, описывающие контекст их вызова, которые также остаются без доказательства. В данных спецификациях могут быть ошибки, отсутствие полноты описания возможных особенностей контекста вызова. В случае подобной ошибки, например, в предусловиях к интерфейсу LSM в коде модуля также могут обнаружится ситуации, обработка которых не предусматривалась. В таком случае дедуктивная верификация покажет, отталкиваясь от неполных/ошибочных предположений, полную корректность кода.

Учитывая отсутствие гарантий выполнения перечисленных выше предположений и принимая во внимание ограничения подхода, можно в дополнение к дедуктивной верификации осуществлять проверки в момент выполнения кода. Так, например, если оттранслировать предусловия к интерфейсу LSM в проверки времени выполнения (assertions), то это позволит в динамике проверить спецификации, при отсутствии возможности их доказать. Таким образом повышается общий уровень доверия к тем исходным предположениям, на основе которых осуществлялась дедуктивная верификация.

Для реализации этой идеи можно воспользоваться расширением E-ACSL [105] для платформы Frama-C, где на основе исходных кодов и спецификаций к ним на языке ACSL порождает новый исходный код с проверками времени выполнения. Расширение E-ACSL работает лишь с подмножеством языка ACSL [106]. Так, например, невозможно отобразить леммы и аксиомы в исполняемый код по очевидным причинам. Для моделирования работы со спецификаци-

онными типами чисел (неограниченными) используется библиотека libgmp, а также собственная библиотека для отслеживания состояний памяти [107].

К сожалению, существующие инструменты для подобного рода трансформации спецификаций недостаточно зрелы для использования на коде ядра Linux и поддерживают в полной мере лишь некоторое подмножество E-ACSL. В инструментах реализована пока далеко не полная функциональность из заявленной в E-ACSL.

Данный вид анализа важен и должен быть интегрирован в систему тестирования модуля безопасности по мере развития средств верификации. Одним из открытых вопросов его интеграции остается трансляция логических утверждений, описывающих состояние памяти. Для ядра Linux это требует гораздо больших усилий, чем при работе с приложением пользовательского уровня. Одним из решений могло бы быть использование уже существующих динамических средств проверок доступа к памяти kasan, kmsan. Однако их использование не позволяет в полной мере транслировать все допустимые языком E-ACSL логические утверждения на память в проверки времени выполнения и не позволяет обращаться к состояниям памяти по временным меткам.

Заключение

В монографии описан процесс, позволяющий выполнить требования действующих стандартов и регламентов при получении сертификатов по профилям защиты ОС типа «А» третьего и второго классов защиты.

Представленный процесс не является единственно возможным, отдельные работы могут выполняться при помощи других средств и методов. Исследования в различных направлениях развития таких методов и средств — это тема будущих работ.

Опыт проведенных исследований и опыт внедрения результатов исследований показал, что ГОСТ Р ИСО/МЭК 15408, хотя и кажется перегруженным специальными терминами, несет в себе много важных идей и при содержательной интерпретации позволяет существенно продвинуться в обеспечении информационной безопасности программных систем. При этом ограничиваться формальной трактовой требований стандарта нельзя, поверхностное, формальное выполнение требований стандарта не дает гарантий защищенности и может пойти во вред общему уровню информационной безопасности оцениваемых систем. Это означает, что необходимо развивать как сам стандарт, так и конкретные методики и инструменты разработки и анализа программных систем, поддерживающие процессы жизненного цикла, включая инструменты управления требованиями, моделирования, статического и динамического анализа, формальной верификации, конфигурационного управления и др.

Работа посвящена задачам верификации политики безопасности управления доступом в операционных системах, вместе с тем очевидно, что аналогичные исследования необходимо вести и для других видов программных систем. В первую очередь это касается СУБД, затем на повестке дня встают вопросы верификации распределенных систем, в частности распределенных файловых систем. Отдельной проработки требуют вопросы сертификации систем реального времени, встроенных систем, «интернета вещей» и др.

Таким образом, авторы надеются, что это не последняя редакция данной книги. В следующих выпусках должны появиться результаты развития методов моделирования и верификации программных и программно-аппаратных систем для обеспечения информационной безопасности широкого класса объектов оценки.

Используемые сокращения и обозначения

ОССН	Операционная система специального назначения
ACSL	ANSI/ISO C Specification Language
ADV	Класс доверия «Разработка»
ADV_ARC	Семейство требований доверия «Архитектура безопасности»
ADV_FSP	Семейство требований доверия «Функциональная спецификация»
ADV_IMP	Семейство требований доверия «Представление реализации»
ADV_INT	Семейство требований доверия «Внутренняя структура ФБО»
ADV_SPM	Семейство требований доверия «Моделирование политики безопасности»
ADV_TDS	Семейство требований доверия «Проект ОО»
API	Application programing interface
AVA_CCA_EXT	Семейство требований доверия «Анализ скрытых каналов»
ATE_COV	Семейство требований доверия «Покрытие»
DAC	Discretionary access control
LSM	Linux Security Modules
RBAC	Role-based access control
SELinux	Security-Enhanced Linux
SMT	Satisfiability modulo theories
ДП-модель	Модель управления доступом и информационными потоками
МРОСЛ ДП-модель	Мандатная сущностно-ролевая модель управления доступом и информационными потоками в ОС семейства Linux
ОО	Объект оценки
ОС	Операционная система
ОУД	Оценочные уровни доверия
ПБО	Политика безопасности объекта оценки
ПО	Программное обеспечение
СЗИ	Средства защиты информации
ФБО	Функции безопасности ОО
ФСП	Функциональная спецификация
ФСТЭК	Федеральная служба по техническому и экспортному контролю
ФТБ	Функциональные требований безопасности

Литература

1. Операционные системы Astra Linux. URL: http://www.astralinux.ru

2. Astra Linux сертифицирована по требованиям ФСТЭК России к операционным системам (2017). URL: http://old.astra-linux.su/home/novosti/437-rbt-fstec.html

3. Буренин П.В., Девянин П.Н., Лебеденко Е.В. и др. Безопасность операционной системы специального назначения Astra Linux Special Edition. Учебное пособие для вузов / Под ред. д-ра техн. наук П.Н. Девянина. 2-е издание, стереотипное. — М.: Горячая линия — Телеком, 2016. — 312 с.

4. ГОСТ Р ИСО/МЭК 15408-1-2012. Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий. Часть 1. Введение и общая модель.

5. ГОСТ Р ИСО/МЭК 15408-2-2013. Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий. Часть 2. Функциональные компоненты безопасности.

6: ГОСТ Р ИСО/МЭК 15408-3-2013. Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий. Часть 3. Компоненты доверия к безопасности.

7. Bell D.E., LaPadula L.J. Secure Computer Systems: Unified Exposition and Multics Interpretation. — Bedford, Mass.: MITRE Corp., 1976. MTR-2997 Rev. 1.

8. Bishop M. Computer Security: art and science. ISBN 0-201-44099-7, 2002. — 1084 p.

9. Девянин П.Н. Модели безопасности компьютерных систем. Управление доступом и информационными потоками. Учебное пособие для вузов. 2-е изд., испр. и доп. — М.: Горячая линия — Телеком, 2013. — 338 с.

10. Thumser W. Formal Specification of Security Policy Models // Proc. of The 7th International Common Criteria Conference, Spain, 2006.

11. Trusted Computer System Evaluation Criteria. US Department Of Defense, 1985. CSC-STD-001-83.

12. Гостехкомиссия России. Руководящий документ. Средства вычислительной техники защита от несанкционированного доступа к информации показатели защищенности от несанкционированного доступа к информации. — М.: Военное издательство, 1992.

13. Профиль защиты операционных систем типа «А» третьего класса защиты. Методический документ. — ФСТЭК России, 2017.

14. Профиль защиты операционных систем типа «А» второго класса защиты. Методический документ. — ФСТЭК России, 2017.

15. Информационное сообщение об утверждении Требований безопасности информации к операционным системам от 18 октября 2016 г. № 240/24/4893 / ФСТЭК России. URL: http://fstec.ru/component/attachments/download/1051

16. Security-Enhanced Linux. URL: http://www.nsa.gov/research/selinux.index.shtml

17. Документы по сертификации средств защиты информации и аттестации объектов информатизации по требованиям безопасности информации. URL: http://fstec.ru/tekhnicheskaya-zashchita-informatsii/dokumenty-po-sertifikatsii/120-normativnye-dokumenty

18: ГОСТ Р ИСО/МЭК 18045-2013. Информационная технология. Методы и средства обеспечения безопасности. Методология оценки безопасности информационных технологий.

19. ГОСТ Р 53113.1-2008. Информационная технология. Защита информационных технологий и автоматизированных систем от угроз информационной безопасности, реализуемых с использованием скрытых каналов. Часть 1. Общие положения.

20. Девянин П.Н. О запрещенных информационных потоках по времени через параметры сущностей в операционных системах семейства Linux с мандатным управлением доступом // Методы и технические средства обеспечения безопасности информации: Материалы 21-й научно-технической конференции 24 июня — 29 июня 2012 года. СПб.: Изд-во Политехн. ун-та, 2012. С. 88–90.

21. Abrial J.-R. Modeling in Event-B: System and Software Engineering. — Cambridge University Press, 2010.

22. Biba K. Integrity considerations for secure computer systems. Technical Report MTR-3153, The MITRE Corporation. 1975.

23. Девянин П.Н. Анализ безопасности управления доступом и информационными потоками в компьютерных системах. — М.: Радио и связь, 2006. — 176 с.

24. Назаров И.О. Анализ безопасности веб-систем, в условиях реализации уязвимости класса межсайтового скриптинга // Проблемы информационной безопасности. Компьютерные системы / Под ред. П.Д. Зегжды. Выпуск 2, 2007. С. 105–117.

25. Назаров И.О. Обеспечение безопасности управления доступом и информационными потоками в веб-системе на основе СУБД // Вестник Казанского государственного технического университета им. А.Н. Туполева. Выпуск 2, 2008. С. 56–59.

26. Колегов Д.Н. ДП-модель компьютерной системы с функционально и параметрически ассоциированными с субъектами сущностями // Вестник Сибирского государственного аэрокосмического университета им. академика М.Ф. Решетнева. Вып. 1(22), 2009. Часть 1. С. 49–54.

27. Буренин П.В. Подходы к построению ДП-модели файловых систем // Прикладная дискретная математика. 2009. № 1(3). С. 93–112.

28. Девянин П.Н. Базовая ролевая ДП-модель // Прикладная дискретная математика. 2008. № 1(1). С. 64–70.

29. Девянин П.Н. Ролевая ДП-модель управления доступом и информационными потоками в операционных системах семейства Linux // Прикладная дискретная математика. 2012. № 1(15). С. 69–90.

30. Девянин П.Н. О результатах формирования иерархического представления МРОСЛ ДП-модели // Прикладная дискретная математика. 2016. Приложение № 9. С. 83–87.

31. Тележников В.Ю. Правила преобразования состояний системы в рамках ДП-модели управления доступом в компьютерных сетях, построенных на основе ОС семейства Linux // Прикладная дискретная математика. 2016. № 1(31). С. 67–85.

32. Программный комплекс «Виртуализации и управления». URL: http:// old.astra-linux.su/doc-pkviu.html

33. Девянин П.Н. Реализация невырожденной решетки уровней целостности в рамках иерархического представления МРОСЛ ДП-модели // Прикладная дискретная математика. 2017. Приложение № 10. С. 111–114.

34. Способ обеспечения безопасности информационных потоков в защищенных информационных системах с мандатным и ролевым управлением доступом: пат. 2525481 Рос. Федерация : МПК G 06 F 21/62 / Девянин П.Н.; патентообладатели Девянин П.Н. и ОАО «НПО РусБИТех». № 2012146550/08; заявл. 01.11.2012; опубл. 10.05.2014, Бюл. № 13. 12 с.

35. Sandhu R. Rationale for the RBAC96 family of access control models // Proceeding of the 1st ACM Workshop on Role-Based Access Control. ACM, 1997.

36. Sandhu R. Role-Based Access Control // Advanced in Computers. — Academic Press, 1998. Vol. 46.

37. Rodin Handbook. http://handbook.event-b.org

38. Gurevich Y. Evolving algebras 1993: Lipari guide. In: Börger, E. (ed.) Specification and Validation Methods, pp. 9-36. Oxford University Press, Oxford, 1995.

39. Jackson D. Software Abstractions: Logic, Language, and Analysis. — MIT Press, Cambridge, 2006.

40. Abrial J.-R. The B-Book: Assigning Programs to Meanings. — Cambridge University Press, Cambridge, 1996.

41. Lamport L. Specifying Concurrent Systems with TLA+. NATO Science Series, III: Computer and Systems Sciences. IOS Press, Amsterdam. 173 (Calculational System Design): 183–247, 2000.

42. Bjorner D., Jones C.B. The Vienna Development Method: The Meta-Language. Lecture Notes in Computer Science 61. — Berlin, Heidelberg, New York: Springer, 1978.

43. Abrial J.-R. Data Semantics // Klimbie J.W., Koffeman K.L. Proceedings of the IFIP Working Conference on Data Base Management, North-Holland, pp. 1–59, 1974.

44. Abrial J.-R., Butler M., Hallerstede S. et al. Rodin: An Open Toolset for Modelling and Reasoning in Event-B // International Journal on Software Tools for Technology Transfer. 2010. Vol. 12, no. 6. P. 447–466.

45. AstraVer Toolset URL: https://forge.ispras.ru/projects/astraver-toolset

46. Wirth N. Program development by stepwise refinement // CACM: Communications of the ACM 14 (1971).

47. Abrial J.-R., Hallerstede S. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. Fundamentae // Informatica. 2007. 77(1,2). 1–28.

48. Bourdonov I., Kossatchev A., Petrenko A., Galter D. KVEST: Automated Generation of Test Suites from Formal Specifications // FM'99: Formal Methods. LNCS, vol. 1708, pp. 608–621, Springer-Verlag, 1999.

49. Microsoft Interoperability Initiative. URL: http://www.microsoft.com/openspecifications

50. OLVER project. URL: http://linuxtesting.org

51. George C., Haxthausen A.E., Hughes S., Milne R., Prehn S., Pedersen J.S. The Raise Development Method. — London: Prentice-Hall International, 1995.

52. Baudin P., Filliâtre J. C., Marché C., Monate B., Moy Y., Prevosto V. ACSL: ANSI C Specification Language. ver 1.12, 2017.

53. Shutemov K. Kernel: use the gnu89 standard explicitly // Журнал разработки ядра ОС Linux, Октябрь 2014. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=51b97e354ba9fce1890cf38ecc754aa49677fc89

54. Edge J. LSM: loadable or static? LWN.net, Октябрь 2007. URL: https://lwn.net/Articles/255650/

55. Wright C., Cowan C., Smalley S., Morris J., Kroah-Hartman G. Linux Security Module Framework // Ottawa Linux Symposium, 2002. URL: http://www.kroah.com/linux/talks/ols_2002_lsm_paper/lsm.pdf

56. Wright C., Cowan C., Morris J., Smalley S. and Kroah-Hartman G.: Linux security modules: general security support for the linux kernel. Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems], 2003, pp. 213-226. doi: 10.1109/FITS.2003.1264934. URL: https://www.usenix.org/legacy/event/sec02/full_papers/wright/wright.pdf

57. Edge J. Progress in security module stacking // LWN.net, Март 2015. URL: https://lwn.net/Articles/635771

58. Zhang X., Edwards A., Jaeger T. Using CQUAL for Static Analysis of Authorization Hook Placement // Proceedings of the 11th USENIX Security Symposium, D. Boneh (Ed.), 2002, USENIX Association, Berkeley, CA, USA, 33-48. URL: https://www.usenix.org/legacy/event/sec02/full_papers/zhang/zhang.pdf

59. Edwards A., Jaeger T., Zhang X. Runtime verification of authorization hook placement for the linux security modules framework // Proceed-

ings of the 9th ACM conference on Computer and communications security (CCS '02), 2002, V. Atluri (Ed.). ACM, New York, NY, USA, 225–234. doi: 10.1145/586110.58614

60. Jaeger T., Edwards A., Zhang, X. Consistency analysis of authorization hook placement in the Linux security modules framework // ACM Trans. Inf. Syst. Secur. 7, 2 (May 2004), 175–205. doi: 10.1145/996943.99694

61. Ganapathy V., Jaeger T., Jha S. Automatic placement of authorization hooks in the linux security modules framework // Proceedings of the 12th ACM conference on Computer and communications security (CCS '05), 2005, ACM, New York, NY, USA, 330–339. doi: 10.1145/1102120.110216

62. Zheng, Y.: ceph: fix security xattr deadlock. Лог разработки ядра, Март 2016. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=315f24088048a51eed341c53be66ea477a3c7d1

63. Floyd R. Assigning Meanings to Programs // Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics, American Mathematical Society, Providence, Rhode Island, 1967, pp. 19–32.

64. Hoare C.A.R. An Axiomatic Basis for Computer Programming // Comm. ACM 12, 10 (1969), 576–583.

65. King J.C. A Program Verifier. Ph.D. Th., Carnegie-Mellon University, 1969.

66. Boyer R.S., Moore J.S. A Verification Condition Generator for FORTRAN // The Correctness Problem in Computer Science, R.S. Boyer and J.S. Moore, Eds. — London: Academic Press, 1981.

67. Tschannen J., Carlo F., Nordio M., Polikarpova N. AutoProof: Auto-active Functional Verification of Object-oriented Programs // Proceedings of 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 15, 2015.

68. Chapman R., Schanda F. Are we there yet? 20 years of industrial theorem proving with SPARK // International Conference on Interactive Theorem Proving, Springer, Cham, 2014, pp. 17–26.

69. Pearce D.J., Groves L. Whiley: a platform for research in software verification // International Conference on Software Language Engineering, Springer, Cham, 2013, pp. 238–248.

70. Leino, K.R.M. Dafny: An automatic program verifier for functional correctness // International Conference on Logic for Programming Artificial Intelligence and Reasoning, Springer, Berlin, Heidelberg, 2010, pp. 348–370.

71. Filliâtre J.C., Paskevich A. Why3 — where programs meet provers // European Symposium on Programming, Springer, Berlin, Heidelberg, 2013, pp. 125–128.

72. Leroy X. et al. The CompCert verified compiler. Documentation and user's manual. — INRIA Paris-Rocquencourt, 2012.

73. Klein G. et al. seL4: Formal verification of an OS kernel // Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ACM, 2009, pp. 207–220.

74. Cok D.R., Blissard I., Robbins J. C Library annotations in ACSL for

Frama-C: experience report. GrammaTech Inc., 2017. URL: http://annotationsforall.org/resources/links/GT-libc-experience-report.pdf

75. Pariente D., Ledinot E. Formal verification of industrial C code using Frama-C: a case study. Formal Verification of Object-Oriented Software, 2010, 205 pp.

76. Мандрыкин М., Хорошилов А. Высокоуровневая модель памяти промежуточного языка Jessie с поддержкой произвольного приведения типов указателей // Программирование. 2015. № 4. С. 23–39.

77. Мандрыкин М. Моделирование памяти Си-программ для инструментов статической верификации на основе SMT-решателей. Диссертация. — М., 2016.

78. Barnes J. Rationale for Ada 2012: Contracts and aspects. January 2013, DOI: 10.1007/978-3-642-45210-9

79. Мейер Б. Объектно-ориентированное конструирование программных систем. — М.: Русская редакция, 2004. — 1204 с.

80. AstraVer Project. ИСП РАН им. В.П. Иванникова. URL: http://linuxtesting.ru/astraver

81. Kirchner F., Kosmatov N., Prevosto V., Signoles J., & Yakobowski B. Frama-C: a software analysis perspective // Formal Aspects of Computing. 2015. 27(3). P. 573–609.

82. Marché C., Moy Y. The Jessie plugin for deductive verification in Frama-C. — INRIA Saclay Île-de-France and LRI, CNRS UMR, 2012.

83. Gouw S., Boer F., Bubel R., Hähnle R., Rot J., Steinhöfel D. Verifying OpenJDK's Sort Method for Generic Collections // Journal of Automated Reasoning. 2017. URL: https://doi.org/10.1007/s10817-017-9426-4 URL: https://github.com/abstools/java-timsort-bug/blob/master/KeY/src/java/util/TimSort.java

84. AstraVer Toolset. ИСП РАН им. В.П. Иванникова. URL: http://linuxtesting.ru/astraver

85. MISRA C: 2012. Guidelines for the Use of the C Language in Critical Systems, ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF), March 2013.

86. Watson A., Wallace D., McCabe,T. Structured testing: A testing methodology using the cyclomatic complexity metric. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1996, 500-235, 140 p.

87. Howells D. Credentials in Linux. URL: https://www.kernel.org/doc/Documentation/security/credentials.txt

88. Wright C. LSM update, another missing hook. Linux kernel log, lwn.net, 2005. URL: https://lwn.net/Articles/155496/

89. Georget L. Add missing LSM hooks in mq_timed {send,receive} and splice. Linux kernel log, 2016. URL: http://thread.gmane.org/gmane.linux.kernel.lsm/28737

90. Belousov K., Viro A.: Linux Kernel LSM file_permission Hook Restriction Bypass. 2005. URL: https://vulners.com/osvdb/OSVDB:25747

91. Jurgens D. SELinux support for Infiniband RDMA. 2016. URL: https://lwn.net/Articles/684431/

92. Goyal V. Overlayfs SELinux Support. 2016. URL: https://lwn.net/Articles/693663/

93. Corbet J. ACCESS_ONCE() and compiler bugs. 2014. URL: https://lwn.net/Articles/624126/

94. Draios, Sysdig 2018 URL: http://www.sysdig.org/

95. Desnoyers M., Dagenais M. LTTng: Tracing across execution layers, from the hypervisor to user-space // Linux symposium, 2008.

96. Gregg B., Mauro J. DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD. — Prentice Hall Professional, 2011.

97. Ktap: A lightweight script-based dynamic tracing tool for Linux URL: https://github.com/ktap/ktap URL: http://www.ktap.org

98. perf: Linux profiling with performance counters URL: https://perf.wiki.kernel.org/index.php/Main_Page

99. ftrace — Function Tracer URL: https://www.kernel.org/doc/Documentation/trace/ftrace.txt

100. Eigler F.C. et al.: Architecture of systemtap: a Linux trace/probe tool. 2005.

101. Méry D., Singh N.K. Automatic code generation from Event-B models // Proceedings of the second symposium on information and communication technology, ACM, 2011. pp. 179-188.

102. Wright S. Automatic generation of C from Event-B // Workshop on integration of model-based formal methods and tools, 2009, 14 pp.

103. Dalvandi M., Butler M., Rezazadeh A. From Event-B Models to Dafny Code Contracts // Fundamentals of Software Engineering, 2015, Springer International Publishing, LNCS volume 9392, ISBN: 978-3-319-24644-4 DOI: https://doi.org/10.1007/978-3-319-24644-4_21

104. Новиков Е.М.: Развитие ядра операционной системы Linux // Труды Института системного программирования РАН. 2017. Т. 29, вып. 2. С. 77–96.

105. E-ACSL Executable ANSI/ISO C Specification Language Version 1.12. URL: https://frama-c.com/download/e-acsl/e-acsl.pdf

106. Delahaye M., Kosmatov N., Signoles J. Common specification language for static and dynamic analysis of C programs // Proceedings of the 28th Annual ACM Symposium on Applied Computing, ACM, 2013, С. 1230–1235.

107. Kosmatov N., Petiot G., Signoles J. An optimized memory monitoring for runtime assertion checking of C programs // International Conference on Runtime Verification. Springer, Berlin, Heidelberg, 2013, С. 167–182.

108. A Concise Summary of the Event-B mathematical toolkit. http://wiki.event-b.org/images/EventB-Summary.pdf

109. AstraVer Toolset URL: https://forge.ispras.ru/projects/astraver-toolset

110. Back R.-J. On the correctness of refinement steps in program development. Ph.D. thesis, University of Helsinki (1978).

Приложение А. Язык спецификаций Event-B

Данное приложение содержит краткое введение в Event-B. Оно адресовано тем читателям, которые не знакомы с данной нотацией и с методами строгого доказательства корректности формальных спецификаций.

Event-B — это формальный метод, предназначенный для специфицирования сложных систем. Event-B является дальнейшим развитием метода B. Он обладает более простой нотацией, которую легче изучать и использовать. Некоторые языковые конструкции B, как, например, поддержка транзитивного замыкания, были удалены. Данные изменения сделаны с целью упрощения процесса разработки и доказательства спецификаций, а также для защиты разработчика-верификатора от излишних деталей. Кроме этого, также претерпела изменения структура спецификаций.

А.1. Математическая нотация

Математическая нотация Event-B основана на теории множеств и логики первого порядка. В расположенных далее таблицах описаны основные конструкции языка. Их следует использовать в качестве справочной информации при изучении текста спецификации базового уровня МРОСЛ ДП-модели. Данное описание не является полным, оно затрагивает только те конструкции, которые были

Таблица B.1

Числа

Множество целых чисел	$\mathbb{Z}$
Множество натуральных чисел	$\mathbb{N}$
Множество положительных натуральных чисел	$\mathbb{N}_1 = \mathbb{N} \setminus \{0\}$
Сумма	m + n (Здесь и далее m и n это числа)
Разность	m — n
Произведение	m * n
Частное	m / n
Остаток	m mod n
Интервал	$m..n = \{i \mid m \leqslant i \wedge i \leqslant n\}$ (Множество таких i, что m меньше или равно i и i меньше или равно n)

Таблица В.2

Предикаты над числами

Больше	$m > n$
Меньше	$m < n$
Больше или равно	$m \geqslant n$
Меньше или равно	$m \leqslant n$

Таблица В.3

Предикаты

Ложь, false	FALSE
Истина, true	TRUE
Множество BOOL	BOOL = {FALSE, TRUE}
Конъюнкция	$P \land Q$
	(Здесь и далее P и Q это предикаты)
Дизъюнкция	$P \lor Q$
Импликация	$P \Rightarrow Q$
Эквивалентность	$P \leftrightarrow Q = P \Rightarrow Q \land Q \Rightarrow P$
Отрицание	$\neg P$
Квантор всеобщности	$\forall z \cdot P \Rightarrow Q$
	(Для всех значений переменной z, удовлетворяющих предикату P, выполняется предикат Q)
Квантор существования	$\exists z \cdot P \land Q$
	(Существует такое значение переменной z, удовлетворяющее предикату P, что выполняется предикат Q)
Равенство	$P = Q$
Неравенство	$P \neq Q$

Таблица В.4

Отношения

Отношение (множество упорядоченных пар)	$r \in S \Leftrightarrow T = \mathbb{P}(S \times T)$
	(Здесь и далее r это отношение)
Область определения отношения	dom(r)
	$\forall r \cdot r \in S \leftrightarrow T \Rightarrow dom(r) =$
	$\{x \cdot (\exists y \cdot x \mapsto y \in r)\}$
Область значения отношения	ran(r)
	$\forall r \cdot r \in S \leftrightarrow T \Rightarrow ran(r) = \{y \cdot (\exists x \cdot x \mapsto$
	$y \in r)\}$
Ограничение области определения	$S \lhd r = \{x \mapsto y \mid x \mapsto y \in r \land x \in S\}$
Вычитание области определения	$S \lhd\!\!\!- r = \{x \mapsto y \mid x \mapsto y \in r \land x \notin S\}$
Ограничение области значения	$r \rhd T = \{x \mapsto y \mid x \mapsto y \in r \land y \in T\}$
Вычитание области значения	$r \rhd\!\!\!- T = \{x \mapsto y \mid x \mapsto y \in r \land y \notin T\}$
Образ отношения	$r[S] = \{y \mid \exists x \cdot x \in S \land x \mapsto y \in r\}$
Отношение тождества	id
	$S \lhd id = \{x \mapsto x \mid x \in S\}$
	(Множество S определяется автоматически из контекста)

использованы при разработке спецификации МРОСЛ ДП-модели. Полную версию, послужившую источником информации для данного раздела, можно найти на сайте Event-B [108].

Таблица В.5

Множества

Пустое множество	$\varnothing$
Задание множества с помощью перечисления его элементов	$\{E, F\}$ (Здесь и далее E и F это выражения)
Задание множества с помощью описания его элементов	$\{x \mid P\}$ (Множество всех значений переменной x, которые удовлетворяют предикату P. Также возможно задание множества, состоящего из упорядоченных пар, если вместо x написать $x \mapsto y$)
Объединение	$S \cup T = \{x \mid x \in S \vee x \in T\}$ (Здесь и далее S и T это множества)
Пересечение	$S \cap T = \{x \mid x \in S \wedge x \in T\}$
Разность	$S \setminus T = \{x \mid x \in S \wedge x \notin T\}$
Упорядоченная пара	$E \mapsto F$
Декартово произведение (множество всех упорядоченных пар между элементами двух множеств)	$S \times T = \{x \mapsto y \mid x \in S \wedge x \in T\}$
Булеан — множество всех подмножеств множества	$\mathbb{P}(S) = \{s \mid s \subseteq S\}$
	$\mathbb{P}_1(S) = \mathbb{P}(S) \setminus \{\varnothing\}$
Мощность множества (определено только для конечных множеств)	$\mathrm{card}(S)$
Отношение принадлежности	$E \in S$, $E \notin S$
Отношение включения	$S \subseteq T$, $S \not\subseteq T$
Отношение строгого включения	$\mathrm{finite}(S)$
Партиция	$\mathrm{partition}(S, x, y)$ (Означает, что $S = x \cup y \wedge x \cap y = \varnothing$)

Таблица В.6

Функции

Частичная функция (функция — это отношение, каждому элементу области определения которого поставлен в соответствие только один элемент области значения)	$f \in S \nrightarrow T$ (Здесь и далее f это функция)
Тотальная функция (областью определения тотальной функции является все множество S, в отличие от частичной функции)	$f \in S \to T = \{x \cdot x \in S \nrightarrow T \wedge \mathrm{dom}(x) = S\}$
Применение функции к аргументу	$f(E)$

A.2. Контекст

Каждая спецификация на Event-B состоит из компонентов двух типов: контекстов и машин. Контексты содержат статическую, неизменяемую часть спецификации: определения множеств и констант, а также аксиомы, которые являются предикатами, в виде которых описываются типы и свойства констант и множеств. Контексты могут быть расширены другими контекстами.

Аксиомы принимаются истинными без требования доказательства. Аксиомы также могут быть помечены как требующие доказательства теоремы. Такие теоремы по сути являются следствиями из определенных ранее аксиом и обычно используются в процессе верификации спецификации.

Это означает, что их выполнимость требуется доказывать отдельно, используя аксиомы, которые были определены ранее.

А.3. Пример контекста

Рассмотрим простую спецификацию, состоящую всего из одного контекста. Данная спецификация описывает задачу «Кто убил тетушку Агату?», которую часто рассматривают на занятиях по формальной логике. Далее следует ее краткое описание*:

Кто-то в особняке Дредсбери убил тетушку Агату. В особняке живет всего три человека: Агата, дворецкий и Чарльз. Известно, что убийца ненавидел свою жертву, а также не был богаче ее. Чарльз не ненавидит тех людей, которых ненавидит Агата. Агата ненавидела всех, за исключением дворецкого. Дворецкий ненавидит всех, кто не богаче тетушки Агаты. Также дворецкий ненавидит всех, кого ненавидела Агата. В особняке нет человека, который бы ненавидел всех остальных. Вопрос: кто убил тетушку Агату?

В данной задаче идет речь о трех людях: самой Агате, дворецком, и Чарльзе. Опишем этих людей в виде констант (по одной константе на человека) в соответствующей секции контекста под названием constants:

```
constants
  Agatha
  butler
  Charles
```

Данные константы должны являться частью общего множества. Объявим его в секции контекста sets:

```
sets
  persons
```

Теперь требуется задать тип объявленных констант. Как было отмечено выше, константы должны быть элементами множества

* Все последующие примеры являются переводом материала из Rodin Handbook [37], распространяемого по лицензии Creative Commons Attribution-ShareAlike 3.0 Unported.

persons. Опишем данную связь в виде аксиомы с использованием конструкции partition в секции контекста axioms. Данная аксиома означает, что множество persons состоит ровно из трех попарно различных констант:

```
axioms
  @persons_partition
    partition(persons, {Agatha}, {butler}, {Charles})
```

где @persons_partition — метка, или название данной аксиомы.

Создадим две дополнительные константы hates и richer для моделирования отношений между людьми, описывающих их ненависть друг к другу и степень их богатства. Типом этих констант будет отношение между элементами множества persons. Отношение между двумя множествами — это множество упорядоченных пар из элементов этих множеств, пример: $\{\text{Agatha} \mapsto \text{Charles}, \text{Charles} \mapsto \text{butler}, \text{Charles} \mapsto \text{Agatha}\}$.

```
constants
  hates
  richer
axioms
  @hate_relation
    hates ∈ persons ↔ persons
  @richer_relation1
    richer ∈ persons ↔ persons
```

Данные отношения пока довольно абстрактны: в них ничего не говорится о конкретных людях и их отношениях друг с другом. Подробнее данные отношения будут описаны в последующих аксиомах.

Отношение hates должно быть иррефлексивным, т. е. никто не должен быть богаче себя самого. Опишем данное свойство с помощью специального отношения id (см. Таблицу В.5), которое в контексте с отношением richer будет описывать множество пар следующего вида: $\{\text{Agatha} \mapsto \text{Agatha}, \text{butler} \mapsto \text{butler}, \text{Charles} \mapsto \text{Charles}\}$. Чтобы убедиться, что в отношении richer нет упорядоченных пар упомянутого выше вида, нужно добавить в контекст следующую аксиому:

```
axioms
  @richer_relation2
    richer ∩ id = ∅
```

Также известно, что отношение richer является транзитивным:

```
axioms
  @richer_relation3
    ∀x, y, z · x ↦ y ∈ richer ∧ y ↦ z ∈ richer ⇒ x ↦ z ∈ richer
```

Всегда выполнено следующее условие, называемое антисимметричным: один человек либо богаче другого, либо беднее; оба этих условия не могут быть выполнены одновременно:

axioms
  @richer_relation4
    $\forall x, y \cdot x \mapsto y \in \text{richer} \Rightarrow y \mapsto x \notin \text{richer}$

Так как целью задачи является нахождение убийцы, нужно добавить еще одну константу, назовем ее killer, которая будет являться элементом множества persons. В аксиоме с меткой @persons_partition задано, что множество persons состоит ровно из трех попарно различных констант, следовательно, константа killer должна совпадать с одной из них (иначе множество persons будет состоять из четырех элементов, что противоречит упомянутой аксиоме).

constants
  killer
axioms
  @killer_type
    killer $\in$ persons

В условии задачи описываются дополнительные отношения между людьми, которые мы также оформим в виде аксиом. Мы знаем, что убийца ненавидит свою жертву, и не является богаче ее:

axioms
  @killer_hates
    killer $\mapsto$ Agatha $\in$ hates
  @killer_not_richer
    killer $\mapsto$ Agatha $\notin$ richer

Чарльз не ненавидит тех людей, которых ненавидит Агата, и Агата ненавидит всех, за исключением дворецкого:

axioms
  @charles_hates
    hates[{Agatha}] $\cap$ hates[{Charles}4] $= \varnothing$
  @agatha_hates
    hates[{Agatha}] = persons $\setminus$ {butler}

Дворецкий ненавидит всех, кто не богаче тетушки Агаты. Также дворецкий ненавидит всех, кого ненавидит Агата. Но нет человека, который бы ненавидел всех остальных:

axioms
  @butler_hates1
    $\forall x \cdot x \mapsto$ Agatha $\notin$ richer $\Rightarrow$ butler $\mapsto x \in$ hates
  @butler_hates2
    hates[{Agatha}] $\subseteq$ hates[{butler}]

```
  @noone_hates_everyone
    ∀x · x ∈ persons ⇒ hates[{x}] ≠ persons
```

Осталось смоделировать решение задачи. Предположим, что убийцей является сама Агата, и опишем это в виде помеченной как теорема аксиомы:

```
axioms
  theorem @solution
    killer = Agatha
```

Доказать данную теорему можно с помощью средств автоматического доказательства теорем, которые работают с Event-B. При желании используя средства интерактивного доказательства можно показать, что ни Чарльз, ни дворецкий, не могут являться убийцами, так как при этом будут нарушены определенные в контексте аксиомы.

На этом пример контекста завершен. Целиком спецификация выглядит следующим образом:

```
context AgathaContext
  sets
  persons
  constants
    Agatha
    butler
    Charles
    hates
    richer
    killer
axioms
  @persons_partition
    partition(persons, {Agatha}, {butler}, {Charles})
  @hate_relation
    hates ∈ persons ↔ persons
  @richer_relation1
    richer ∈ persons ↔ persons
  @richer_relation2
    richer ∩ id = ∅
  @richer_relation3
    ∀ x, y, z · x ↦ y ∈ richer ∧ y ↦ z ∈ richer ⇒ x ↦ z ∈ richer
  @richer_relation4
    ∀ x, y · x ↦ y ∈ richer ⇒ y ↦ x ∉ richer
  @killer_type
    killer ∈ persons
  @killer_hates
```

```
    killer ↦ Agatha ∈ hates
  @killer_not_richer
    killer ↦ Agatha ∉ richer
  @charles_hates
    hates[{Agatha}] ∩ hates[{Charles}] = ∅
  @agatha_hates
    hates[{Agatha}] = persons \{butler}
  @butler_hates1
    ∀ x · x ↦ Agatha ∉ richer ⇒ butler ↦ x ∈ hates
  @butler_hates2
    hates[{Agatha}] ⊆ hates[{butler}]
  @noone_hates_everyone
    ∀ x · x ∈ persons ⇒ hates[{x}] ≠ persons
  theorem @solution
    killer = Agatha
end
```

А.4. Машина

В отличие от контекстов машины содержат динамическую часть спецификации: переменные, инварианты, события. Переменные, как и константы, соответствуют простым математическим объектам: они могут быть множествами, бинарными отношениями, функциями, числами, принимать значения логического типа и т. д. Значение переменных формируют текущее состояние спецификации, а инварианты — предикаты, определенные на множестве переменных спецификации — ограничивают его.

Текущее состояние спецификации может быть изменено событием. Каждое событие обычно состоит из названия, параметров, охранных условий и действий. Охранные условия являются обязательными условиями в виде набора предикатов, ограничивающих множество возможных состояний спецификации, в которых данное событие может случиться. Типы параметров события также задаются в блоке охранных условий. Действия изменяют текущее состояние спецификации за счет модификации значения переменных спецификации, причем модификация может быть как детерминированной, так и недетерминированной. Определенные в машине инварианты должны сохраняться в результате любых модификаций значений переменных, поэтому корректность каждого изменения состояния необходимо доказывать.

Все события атомарны и могут произойти, только когда выполняются их охранные условия. Если одновременно выполняются охранные условия нескольких событий, то только одно из них может

произойти в данный момент, причем какое именно событие произойдет выбирается недетерминированным образом.

Машины имеют доступ к элементам контекста через механизм «видения», что позволяет использовать определенные там константы и множества. Машина «M» может видеть контекст «C» косвенным образом, если машина видит явным образом контекст, который является расширением контекста «C». Кроме того, машины могут уточнять друг друга с помощью техники пошагового уточнения. Взаимосвязь контекстов и машин была показана рис. 4.1.

Существует два основных способа уточнения событий в Event-B. В первом способе все определенные в уточняемом событии охранные условия и действия наследуются и, кроме того, могут быть дополнены дополнительными охранными условиями и действиями, причем новые действия могут изменять только переменные уточненной машины.

Второй способ подразумевает полное переписывание охранных условий и действий уточняемого события в уточненном. При этом требуется доказать, что из предусловий уточненного события следуют предусловия уточняемого, а уточненные действия не противоречат уточняемым. Если у уточняемого события был параметр, которые требуется заменить на другой, то в уточненном событии может быть добавлено специальное поле, называемое свидетельством (witness), которая связывает значения старого и нового параметров.

Как и аксиомы, инварианты и охранные условия событий также могут быть помечены как требующие доказательства теоремы. В случае инварианта метка теоремы означает, что данный инвариант является следствиями из определенных ранее инвариантов и аксиом и, следовательно, его выполнимость не нужно доказывать в результате каждого изменения состояния. Вместо этого, корректность данной теоремы-инварианта доказывается единожды, после чего она может использоваться в процессе верификации спецификации. Аналогичным образом, помеченное как теорема охранное условие события перестает быть охранным условием и становится следствием прочих охранных условий данного события, а также определенных в спецификации инвариантов и аксиом.

Также в Event-B имеется возможность доказательства отсутствия состояний взаимной блокировки, а также доказательства завершимости с помощью инвариантов.

А.4.1. Пример машины

В качестве примера рассмотрим простую спецификацию контроллера двух светофоров на пешеходном переходе (рис. В.1): светофора для автомобилистов и светофора для пешеходов.

Начнем с описания состояния спецификации. Необходимо моделировать два светофора: один для пешеходов и один для автомобилистов, так что создадим две переменные — cars_go и peds_go:

```
variables
  cars_go
  peds_go
```

Для описания типов переменных нужно добавить в спецификацию соответствующие инварианты. Для упрощения определим их как переменные логического типа, где значение TRUE будет означать, что светофор горит зеленым, а FALSE — красным:

```
invariants
  @inv1
    cars_go ∈ BOOL
  @inv2
    peds_go ∈ BOOL
```

Также необходимо проинициализировать переменные начальными значениями. В нашем случае присвоим им значение по умолчанию FALSE:

```
event INITIALISATION
  then
    @act1 cars_go := FALSE
    @act2 peds_go := FALSE
end
```

где блок then события — блок, в котором описываются его действия.

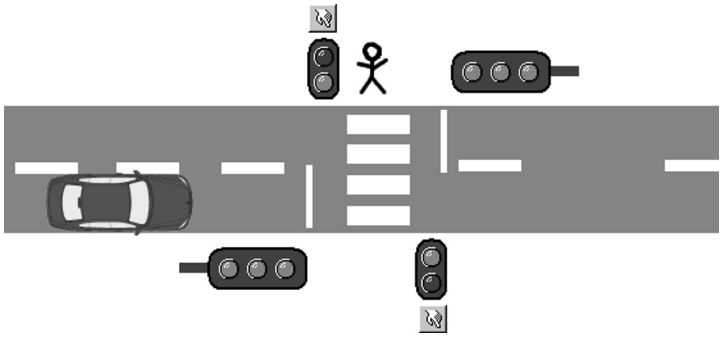

Рис. В.1. Светофоры на пешеходном переходе

Теперь нужно добавить в спецификацию события, которые смогут изменять ее состояние: события, отвечающие за изменения цвета светофоров. Сначала создадим два события, отвечающих за светофор для пешеходов, действия которых будут изменять значение переменной peds_go на TRUE и FALSE:

```
event set_peds_go
  then
    @act1 peds_go := TRUE
end
event set_peds_stop
  then
    @act1 peds_go := FALSE
end
```

Для моделирования события изменения цвета светофора для автомобилистов set_cars мы воспользуемся другим подходом. Событие будет включать в себя новое состояние светофора неявным образом в виде параметра, и благодаря этому нам будет достаточно всего одного события. Назовем параметр new_value и опишем его тип в блоке охранных условий как логический. Больше данный параметр никак не ограничивается, т. е. он может иметь любое значение, как TRUE, так и FALSE:

```
event set_cars
  any new_value
  where
    @grd1 new_value ∈ BOOL
  then
    @act1 cars_go := new_value
end
```

any — блок с объявлением параметров события, where — блок с охранными условиями.

Теперь осталось отразить в спецификации требование безопасности к контроллеру светофоров: нельзя допустить ситуацию, в которой оба светофора показывают зеленый свет. Оформим это требование в виде инварианта:

```
invariants
  @inv3
    ¬ (cars_go = TRUE ∧ peds_go = TRUE)
```

В текущем виде спецификации данный инвариант легко может быть нарушен событиями set_peds_go и set_cars. Чтобы исправить это, необходимо усилить предусловия этих событий следующим образом:

```
event set_peds_go
  where
    @grd1 cars_go = FALSE
  then
    @act1 peds_go := TRUE
end
event set_cars
  any new_value
  where
    @grd1 new_value ∈ BOOL
    @grd2 new_value = TRUE ⇒ peds_go = FALSE
  then
    @act1 cars_go := new_value
end
```

Данные дополнительные охранные условия обеспечивают сохранность инварианта в результате выполнения этих событий, что может быть подтверждено формальным доказательством с использованием автоматических инструментов. Целиком машина выглядит следующим образом:

```
machine M0
variables
  cars_go
  peds_go
invariants
  @inv1
    cars_go ∈ BOOL
  @inv2
    peds_go ∈ BOOL
  @inv3
    ¬ (cars_go = TRUE ∧ peds_go = TRUE)
events
  event INITIALISATION
    then
      @act1 cars_go := FALSE
      @act2 peds_go := FALSE
  end
  event set_peds_go
    where
      @grd1 cars_go = FALSE
    then
      @act1 peds_go := TRUE
  end
  event set_peds_stop
```

```
    then
      @act1 peds_go := FALSE
  end
  event set_cars
    any new_value
    where
      @grd1 new_value ∈ BOOL
      @grd2 new_value = TRUE ⇒ peds_go = FALSE
    then
      @act1 cars_go := new_value
  end
end
```

А.4.2. Уточнение

Продолжим пример из прошлого раздела, в котором была разработана упрощенная спецификация контроллера светофоров на пешеходном переходе. Спецификация была упрощенной, так как цвета светофоров были описаны простыми переменными логического типа.

В данном примере воспользуемся уточнением для замены абстрактных переменных логического типа переменными, которые будут явно описывать текущий цвет светофоров. Разобьем задачу на две части: сначала создадим контекст, в котором опишем структуры данных, отражающие цвета светофоров, а затем проведем уточнение существующей машины из предыдущего примера, которое будет иметь доступ к созданному контексту, а также заменит переменные логического типа на новые переменные.

Предлагаемый контекст достаточно прост. Он называется C1, и в нем определяется три константы, описывающие цвета светофоров: red, yellow, green. Кроме того, в нем имеются множество COLOURS, которые будет нашим новым типом данных, а также аксиома, которая описывает, что множество COLOURS состоит из трех различных цветов. Данная аксиома задает типы определенных констант:

```
context C1
sets
  COLOURS
constants
  red
  yellow
  green
axioms
  @colours_type
```

```
    partition(COLOURS, {red}, {yellow}, {green})
end
```

Создадим уточнение машины из прошлого примера, которое при этом будет иметь доступ к элементам только что созданного контекста:

```
machine M1 refines M0 sees C1
```

У светофора для пешеходов имеется только два цвета (красный и зеленый), и в каждый момент времени показывается только один их них. Введем новую переменную peds_colour, значение которой будет представлять текущий цвет светофора. Также добавим соответствующий инвариант, описывающий тип переменной. Светофор для автомобилистов может показывать более чем один цвет в каждый момент времени, так что переменная, которая будет его описывать — cars_colours, должна иметь тип подмножества множества COLOURS:

```
variables peds_colour cars_colours
invariants
  @inv4
    peds_colour ∈ COLOURS \ {yellow}
  @inv5
    cars_colours ⊆ COLOURS
```

Затем добавим инвариант, которые свяжет значения старой переменной peds_go и новой переменной peds_colour. Зеленый цвет должен быть тогда и только тогда, когда значением переменной peds_go является TRUE. В противном случае цвет должен быть красным. Наличие данного инварианта позволит далее заменить переменную peds_go во всех местах, где она встречается, на переменную peds_colour. Также добавим связующий инвариант для переменной cars_colours:

```
invariants
  @inv6
    peds_go = TRUE ↔ peds_colour = green
  @inv7
    cars_go = TRUE ↔ green ∈ cars_colours
```

Для замены переменных peds_go и cars_go на новые необходимо уточнить имеющиеся в машине события, в коде которых встречается их использование. Начнем с модифицирования события инициализации. Необходимо переписать действия с метками @act1 и @act2 следующим образом:

```
event INITIALISATION
```

```
  then
    @act1 peds_colour := red
    @act2 cars_colours := {red}
end
```

Уточним событие set_peds_go событием set_peds_green. Для этого заменим охранное условие с меткой @grd1 (cars_go = FALSE) на green $\notin$ cars_colours, а действие @act1 peds_go :=TRUE на peds_colour := green. Благодаря связующим инвариантам старое охранное условие следует из нового, т. е. правила использования уточнения выполнены:

```
event set_peds_green refines set_peds_go
  where
    @grd1 green ∉ cars_colours
  then
    @act1 peds_colour := green
end
```

Аналогичным образом уточняем событие set_peds_stop:

```
event set_peds_red refines set_peds_stop
  then
    @act1 peds_colour := red
end
```

Осталось разобраться с последним событием — set_cars, которое мы после уточнения переименуем в set_cars_colours. Так как данное событие использует параметр логического типа new_value, нам потребуется его заменить на новый параметр, который назовем new_value_colours, а также связать старый и новый параметры с помощью свидетельства (блок with события). Также, чтобы цвета загорались в нужном порядке, необходимо добавить несколько дополнительных охранных условий, которые свяжут цвета светофора, которые он показывает сейчас, с новыми, которые загорятся после выполнения события:

```
event set_cars_colours refines set_cars
  any new_value_colours
  where
    @grd1 new_value_colours ⊆ COLOURS
    @grd2 green ∈ new_value_colours ⇒ peds_colour = red
    @grd3 cars_colours = {yellow} ⇒ new_value_colours = {red}
    @grd4 cars_colours = {red} ⇒ new_value_colours = {red, yellow}
    @grd5 cars_colours = {red, yellow} ⇒ new_value_colours = {green}
    @grd6 cars_colours = {green} ⇒ new_value_colours = {yellow}
  with
    @new_value new_value = TRUE ↔ green ∈ new_value_colours
```

```
  then
    @act1 cars_colours := new_value_colours
end
```

Целиком уточненная машина выглядит следующим образом:

```
machine M1 refines M0 sees C1
variables
  peds_colour
  cars_colours
invariants
  @inv4
    peds_colour ∈ COLOURS \{yellow}
  @inv5
    cars_colours ⊆ COLOURS
  @inv6
    peds_go = TRUE ↔ peds_colour = green
  @inv7
    cars_go = TRUE ↔ green ∈ cars_colours
events
  event INITIALISATION
    then
      @act1 peds_colour := red
      @act2 cars_colours := {red}
  end
  event set_peds_green refines set_peds_go
    where
      @grd1 green ∉ cars_colours
    then
      @act1 peds_colour := green
  end
  event set_peds_red refines set_peds_stop
    then
      @act1 peds_colour := red
  end
  event set_cars_colours refines set_cars
    any new_value_colours
    where
      @grd1 new_value_colours ⊆ COLOURS
      @grd2 green ∈ new_value_colours ⇒ peds_colour = red
      @grd3 cars_colours = {yellow} ⇒ new_value_colours = {red}
      @grd4 cars_colours = {red} ⇒ new_value_colours = {red, yellow}
      @grd5 cars_colours = {red, yellow} ⇒ new_value_colours =
        {green}
      @grd6 cars_colours = {green} ⇒ new_value_colours = {yellow}
    with
```

```
      @new_value new_value = TRUE ↔ green ∈ new_value_colours
    then
      @act1 cars_colours := new_value_colours
  end
end
```

Вместе с уточняемой машиной M0 и контекстом C1 она составляет спецификацию контроллера светофоров на пешеходном переходе.

Приложение В. Среда разработки и верификации на языке Event-B

Комплекс инструментальных средств AstraVer Toolset [109] включает в себя среду для разработки и верификации спецификаций на Event-B, основанную на на платформе Rodin*. Rodin содержит в себе текстовый редактор спецификаций, который обладает возможностью обнаружения в спецификациях синтаксических ошибок. Более сложный анализ корректности проводится с помощью автоматических и интерактивных средств, которые также включены в состав Rodin. Интерактивные средства позволяют проводить доказательства вручную, причем их корректность затем проверяется одним из компонентов платформы. Автоматическое же доказательство осуществляется средствами встроенных инструментов, а также SMT-решателями, которые можно добавить в Rodin с помощью плагина. Возможна и комбинация интерактивного и ручного доказательств, при которой доказательство сначала упрощается и разбивается на составные части вручную, каждая из которых затем подается на вход автоматическим инструментам.

Для каждого требующего доказательства случая — неоднозначность выражений, сохранность инвариантов, корректность проведенного пошагового уточнения (если данная техника была использована) — Rodin генерирует соответствующие утверждения для доказательства, причем платформа решает проблему поддержки актуальности сгенерированных утверждений и выполненных доказательств в случае изменений в спецификации. Полное доказательство спецификации означает, что доказаны все сгенерированные утверждения.

Далее приводятся основные типы генерируемых утверждений для доказательства:

- WD (well-definedness) — аксиомы, инварианты, охранные условия и действия событий должны быть определены правильным образом. Пример: если в инварианте есть деление, то будет сгенерировано утверждение для доказательства того, что делимое не является нулем;
- INV (invariant preservation) — для каждого события, изменяющего значение переменной, используемой в инварианте, требу-

* Платформа включает в себя базовый набор инструментов и средства для расширения функциональности.

ется доказать, инвариант остается выполненным и при новых значениях переменной;

- THM (proving theorems) — генерируется для доказательства справедливости аксиом, инвариантов, охранных условий, которые помечены как теоремы;
- FIS (action feasibility) — каждое действие события должно быть выполнимо. Данное условие тривиально в случае детерминированных присваиваний;
- GRD (guard strengthening) — охранное условие уточненного события должно являться усилением охранных условий соответствующего уточняемого события и не должно им противоречить;
- SIM (action simulation) — действие уточненного события не должно противоречить действию уточняемого события;
- EQL (equality of a preserved variable) — уточненное событие не должно изменять переменные уточняемой спецификации;
- MRG (guard strengthening (merge)) — аналогично типу GRD, но для слияния двух уточняемых событий в одно уточненное (для слияния требуется, чтобы действия этих событий были идентичны, а одноименные параметры обладали одинаковым типом);
- VWD (well definedness of a witness) — аналогично WD, но для свидетельств;
- WFIS (feasibility of a witness) — аналогично FIS, но для свидетельств.

Rodin разработан на базе интегрированной среды разработки Eclipse и обладает набором расширяющих функциональность плагинов. В работе по спецификации МРОСЛ ДП-модели использовались плагины SMT Solvers и Atelier B Provers для расширения возможностей средств автоматического доказательства. На ранних этапах разработки также использовались альтернативный текстовый редактор спецификаций Camille и инструмент проверки и анимации моделей (model checking) ProB, однако от них пришлось отказаться из-за возникших проблем с производительностью в следствии роста размера спецификации.

Отдельного упоминания заслуживает Theory Plugin, который позволяет пользователям расширять математическую нотацию Event-B. Вместе с плагином поставляется набор готовых к использованию теорий, среди которых есть теории двоичных деревьев, списков, последовательностей, действительных чисел.

В.1. Пошаговое уточнение в Event-B

Поскольку поведение сложных систем описывается сложными же спецификациями, проведение их верификации также может требовать значительных усилий. Для облегчения этого процесса часто применяется техника пошагового уточнения (далее просто уточнение). Уточнение было разработано для декомпозиции сложных систем на составные части, которые легче понять и поддерживать.

Уточнение — это широко известная техника декомпозиции спецификаций, одно из первых формальных описаний которой можно встретить в работе Р.-Й. Бака 1978 года [110]. Данная техника поддерживается многими формальными методами. Вместо создания единой монолитной спецификации, которая будет содержать в себе все детали моделируемой системы, уточнение предлагает разрабатывать серию связанных между собой спецификаций. В такой серии первая спецификация представляет собой некоторую базовую версию системы, которая содержит в себе только основные детали. Дополнительные детали системы шаг за шагом добавляются в остальных спецификациях серии таким образом, что каждая последующая спецификация в серии является уточнением предыдущих.

Уточнение может быть двух типов. Горизонтальное уточнение используется для добавления новых свойств и событий, тогда как вертикальное уточнение позволяет заменять имеющиеся абстрактные структуры данных спецификации на более конкретные, как, например, неупорядоченное множество можно заменить на упорядоченный массив. Данный способ также иногда называют уточнением данных.

Отношение между двумя спецификациями, которые можно назвать уточняемой и уточненной, считается отношением уточнения, если выполнены следующие условия:

- переменные уточненной спецификации включают все переменные уточняемой и могут содержать новые переменные. Заимствованные из уточняемой спецификации переменные называются далее старыми переменными, все остальные переменные уточненной спецификации — новыми;
- если взять значения только старых переменных в любом начальном состоянии уточненной спецификации, должно получиться начальное состояние уточняемой спецификации (множество начальных состояний, рассматриваемое с точки зрения значений заимствованных переменных, может только сузиться в уточненной спецификации);

- события уточненной спецификации содержат все события уточняемой и могут включать новые события;
- правила, описывающие изменения переменных при наступлении событий, либо остаются неизменными, либо расширяются следующим образом:
- предусловия событий, имеющихся в уточняемой спецификации, в уточненной могут сохраняться или усиливаться (т. е. из уточненного предусловия должно следовать уточняемое);
- постусловия событий, имеющихся в уточняемой спецификации, в уточненной могут быть пополнены правилами изменения новых переменных. Старые переменные должны меняться в точности по тем же правилам, что и в уточняемой спецификации;
- постусловия новых событий уточненной спецификации могут описывать только изменения новых переменных.

Корректность каждого шага уточнения должна быть доказана, так как новые детали, добавляемые на определенном уровне уточнения, могут противоречить деталям, определенным ранее. Однако если проводить уточнение с использованием специальных правил трансформации, то оно может быть корректно по построению, что позволит избежать дополнительных доказательств.

Если уточнение построено корректно (выполняются указанные ограничения на отношение между двумя спецификациями), то инварианты, верные для уточняемой спецификации, будут верны и для уточненной спецификации. Соответственно, их можно не проверять, если доказана корректность уточнения, сэкономив тем самым усилия на проверку корректности уточненной спецификации.

Крайне важно понимать возможности и ограничения уточнения, особенности выбранного формального метода и, разумеется, самой формализуемой системы. Каждое неверное решение может в итоге привести к необходимости полной переделки спецификации. В некоторых случаях неправильно проведенное уточнение может затруднить формализацию и верификацию.

Оглавление

Адрес издательства в Интернет WWW.TECHBOOK.RU

Научное издание

АО «НПО РусБИТех»
Девянин Петр Николаевич

Институт системного программирования им. В.П. Иванникова РАН

Ефремов Денис Валентинович, **Кулямин** Виктор Вячеславович, **Петренко** Александр Константинович, **Хорошилов** Алексей Владимирович, **Щепетков** Илья Викторович

Моделирование и верификация политик безопасности управления доступом в операционных системах

Монография

Редактор Ю. Н. Чернышов
Компьютерная верстка Ю. Н. Чернышова
Обложка художника В. Г. Ситникова

Подписано в печать 15.01.2019. Печать цифровая. Формат 60×88/16. Уч. изд. л. 13,38.
Тираж 500 экз. (1-й з-д 150 экз.) Изд.№ 190787
ООО «Научно-техническое издательство «Горячая линия – Телеком»