О дедуктивной верификации Си программ, работающих с разделяемыми данными[1]

[1]*М.У. Мандрыкин <mandrykin@ispras.ru>*
[1, 2, 3, 4]*А.В. Хорошилов <khoroshilov@ispras.ru>*
[1] *Институт системного программирования РАН, 109004, Россия, г. Москва, ул. А. Солженицына, дом 25*
[2] *Московский государственный университет имени М.В. Ломоносова 119991 ГСП-1 Москва, Ленинские горы, МГУ имени М.В. Ломоносова, 2-й учебный корпус, факультет ВМК*
[3]*Московский физико-технический институт (государственный университет), 141700, Московская область, г. Долгопрудный, Институтский пер., 9*
[4]*НИУ Высшая школа экономики, Россия, Москва, 101000, ул. Мясницкая, д. 20*

Аннотация. В статье рассматривается задача дедуктивной верификации кода ядра ОС Linux, написанного на языке Си и выполняющегося в окружении с высокой степенью параллелизма. Существенной особенностью этого кода является наличие работы с разделяемыми данными, что не позволяет применять классические методы дедуктивной верификации. Для преодоления этих сложностей в работе представлены предложения по формированию подхода к спецификации и верификации кода, работающего с разделяемыми данными, основанные на доказательстве соответствия этого кода заданной спецификации некоторой дисциплины синхронизации. Подход иллюстрируется примерами упрощенной модели спецификации спин-блокировок и внешнего интерфейса механизма синхронизации RCU (Read-copy-update), широко используемого в ядре ОС Linux.

Ключевые слова: Верификация, параллелизм, владение, инварианты, семантика языка С.

DOI: 10.15514/ISPRAS-2015-27(4)-4

Для цитирования: Мандрыкин М.У., Хорошилов А.В. О дедуктивной верификации Си программ, работающих с разделяемыми данными. Труды ИСП РАН, том 27, вып. 4, 2015 г., стр. 49-68. DOI: 10.15514/ISPRAS-2015-27(4)-4.

[1] Исследование проводилось при финансовой поддержке Министерства образования и науки Российской Федерации (уникальный идентификатор проекта – RFMEFI60414X0051).

1. Введение

Дедуктивная верификация – это область статической верификации, в которой изучаются различные подходы к представлению условий корректности программ в соответствии с некоторыми заданными спецификациями в виде множества математических утверждений, называемых *условиями верификации* (УВ), с целью последующего анализа полученных УВ с помощью частично или полностью автоматизированного логического вывода. Одной из основных характеристик большинства методов дедуктивной верификации является их *корректность*, которая означает возможность гарантировать соответствие программы заданной спецификации в некоторых известных предположениях, полагаясь на корректность реализации применяемых инструментов. Инструменты логического вывода, наиболее широко используемые в области дедуктивной верификации, включают интерактивные и автоматические доказатели теорем, а также решатели формул в теориях. Помимо различных решателей и доказателей теорем реализации методов дедуктивной верификации обычно снабжаются соответствующим автоматическим инструментарием для перевода исходной программы в конечный результирующий набор УВ.

В контексте задачи наиболее полной и корректной верификации фрагментов кода ядра Linux применение инструментов дедуктивной верификации имеет как свои преимущества, так и недостатки. С одной стороны, инструменты дедуктивной верификации потенциально могут позволить осуществлять корректную модульную верификацию широкого диапазона свойств непосредственно на существующем низкоуровневом коде ядра, практически не прибегая к его модификации, что особенно важно для кода ядра ОС Linux, который изначально не разрабатывался с целью последующей формальной верификации. Это становится возможным благодаря большой универсальности используемого общего подхода, который в свою очередь полагается на значительную выразительность используемых логических систем. С другой стороны, соответствующие используемым логическим системам инструменты логического вывода всегда имеют ограниченные возможности в силу алгоритмической неразрешимости получаемых задач выполнимости в общем случае и большой алгоритмической сложности используемых алгоритмов логического вывода (для алгоритмически разрешимых фрагментов). На практике инструменты дедуктивной верификации обычно идут по пути частичного решения этих проблем с помощью взаимодействия с пользователем, а также с помощью применения различных сложных техник редукции, декомпозиции и упрощения получаемых УВ. Корме этого, часто инструменты верификации часто предоставляют различные вспомогательные инструменты для облегчения управления множеством используемых инструментов логического вывода и результатов взаимодействия с пользователем (например, полученных полуавтоматических доказательств теорем), получаемых в ходе процесса верификации. Возможность применения редукции обычно является следствием модульности применяемого метода дедуктивной верификации; декомпозиция

достигается за счет разбиения, применяемого к результирующим УВ на нескольких уровнях, таких как, например, разбиение по путям выполнения в исходном коде, разделение памяти на непересекающимся области, раздельная проверка различных аспектов поведения фрагмента программы (безопасности (условий корректности для всех используемых операций), явно специфицированного поведения (функциональной корректности), контекстных условий для ограничения эффектов выполнения императивного кода и др.), разбиение получаемых логических формул в соответствии с их пропозициональной структурой; упрощение формул также обычно возможно благодаря некоторым дополнительным легко проверяемым предположениям, наиболее важные из которых – это системы типов и модели памяти, которые позволяют предварительно использовать высокоэффективные алгоритмы статического анализа для разрешения многих условий корректности и быстрого извлечения дополнительных полезных знаний о решаемой задаче, которые затем могут быть использованы инструментами логического вывода для ускорения решения задачи выполнимости.

В применении дедуктивной верификации к коду ядра Linux основные сложности, таким образом, состоят в достижении эффективности получаемого представления семантики исходного кода в виде логических формул, несмотря на нестрогость типизации, разнообразную прагматику и сложную модель памяти, характерные для языка С, с учетом используемых в коде ядра Linux языковых расширений и низкоуровневых операций при работе с памятью; а также, в еще большей степени, в сохранении модульности применяемого метода дедуктивной верификации в контексте высокой степени параллелизма, характерной для ядра Linux.

Целью проекта *Astraver* [1] является развитие набора инструментов для высокоавтоматизированной дедуктивной верификации модулей ядра Linux. Набор инструментов разрабатывается на основе двух платформ анализа исходного кода и дедуктивной верификации. Первая, внешняя платформа – *Frama-C* [2] – это пакет инструментов для анализа исходного кода ПО, написанного на языке С. Frama-c – расширяемая и модульная платформа с архитектурой на основе динамически подключаемых модулей, которая включает свой собственный подключаемый модуль дедуктивной верификации *WP* [3], а также позволяет создавать и использовать сторонние подключаемые модули. В частности, модуль *Jessie* [4], который изначально был частью платформы дедуктивной верификации *Why*, был отщеплен и в настоящее время разрабатывается в рамках проекта Astraver. Являясь полноценным практическим средством анализа исходного кода на языке С, платформа Frama-C обеспечивает хорошую совместимость с диалектом GNU C, включая его различные специфичные возможности и расширения, используемые в коде ядра Linux. Frama-C также предоставляет свою собственную адаптированную модификацию инфраструктуры CIL [5] для анализа и трансформации исходного кода, которая значительно облегчает реализацию серии преобразований и нормализации исходного кода,

применяемых модулем Jessie с целью предварительного упрощения сложной модели памяти и семантики исходной С-программы. Модуль *Jessie* реализует дедуктивную верификацию путем перевода исходной программы на языке С вместе с соответствующими функциональными спецификациями, написанными на специальном языке спецификаций *ACSL* [6], в набор модулей на языке программирования и спецификации *Why3ML* [7], являющемся входным языком платформы дедуктивной верификации *Why3* [8]. Таким образом, платформа *Why3* служит второй, внутренней платформой дедуктивной верификации, предоставляющей выразительный входной язык и реализующей управление УВ: их генерацию, преобразование, разрешение с использованием внешних инструментов проверки выполнимости и управление ручными доказательствами. *Why3* реализует расширяемый механизм поддержки внешних инструментов проверки выполнимости (решателей и доказателей теорем), с использованием которого в инструменте реализована встроенная поддержка нескольких решателей формул в теориях ([9], [10]), доказателей теорем на основе расширения резолютивного вывода ([11], [12]) и интерактивных доказателей теорем ([13], [14]). Эффективность кодирования УВ достигается модулями WP и Jessie по-разному. В то время как модуль WP обеспечивает гибкость и эффективность, реализую генерацию УВ напрямую, реализуя высокоуровневые преобразования УВ (с помощью модуля *Qed* [3]) и несколько различных моделей памяти (“модель Хоара”, типизированную и побитовую модели), которые могут быть заданны отдельно для каждого УВ, модуль Jessie реализует одну оптимизированную гибридную модель памяти на основе регионов [15] и эффектов [16] с частичной поддержкой явной низкоуровневой переинтерпретации типов указателей [17], оставляя непосредственно генерацию и управление (в том числе различные преобразования) УВ платформе Why3. В ходе начальной стадии исследований в рамках проекта Astraver, модель памяти Jessie (со значительными модификациями [17]) показала достаточную выразительность для представления последовательной части семантики фрагментов кода ядра Linux (включая различные виды приведений типа указателей и побитовые операции). В то же время в к настоящему времени ни в платформе Frama-C, на в платформе Why3 не была реализована поддержка параллельной части семантики для какого-либо класса параллельных программ на языке С.

Как и большая часть современного системного программного обеспечения, модули ядра Linux работают параллельно. Более того, в коде ядра Linux широко используются неблокирующие механизмы синхронизации, такие как атомарные операции, барьеры и особенно RCU [18].

Наиболее существенной сложностью при верификации параллельных программ одидаемо является отсутствие непрерывного потока управления. В то время как общие методы дедуктивной верификации последовательных программ, такие как исчисление слабейших предусловий полагаются на заданные пользователем контракты, представленные в виде предусловий, постусловий и инвариантов, соответствующих точкам в потоке управления программы, па-

раллельная среда на первый взгляд делает применение аналогичных методов верификации невозможным, разрешая недетерминированные наложения эффектов параллельного выполнения различных участков других потоков управления на большинстве путей между точками рассматриваемого потока управления. Такая возможность прерывания потока управления нарушает свойства локальности и композируемости методов дедуктивной верификации и приводит к комбинаторному взрыву числа возможных путей выполнения. Даже введение в рассмотрение примитивов синхронизации и атомарных операций само по себе практически не помогает сократить возникающее огромное пространство поиска среди всевозможных чередований. К счастью, существуют техники, позволяющие избежать явного полного перебора в возникающем пространстве чередований с помощью переноса инвариантов, ранее сопоставленных точкам в потоке управления последовательной программы, на типы данных, с которыми работает параллельная программа. С учетом такого изменения методологии (при определенных дополнительных ограничениях на инварианты используемых типов данных) введение дисциплины синхронизации позволяет свести полный перебор всевозможных чередований к так называемому *крупнозернистому параллелизму*, в рамках которого вмешательство параллельных потоков выполнения может происходить только в относительно небольшом числе явно обозначенных точек в потоке управления программы, а заданные инварианты типов данных требуется проверять лишь *локально*, то есть на заранее известном конечном множестве отрезков путей выполнения. Это позволяет полностью восстановить локальность метода верификации ценой потери гарантии завершимости параллельной программы. Методология *владения с локально проверяемыми двухметочными инвариантами* (англ. *locally-checked two-state invariants*, сокр. *LCI*) [22] является одним из таких локальных методов верификации параллельных программ. Основные понятия, водимые в методологии владения, – *двухметочный инвариант*, *взаимная допустимость* инвариантов и *динамическое утверждение*. Вместе они составляют основу локального метода дедуктивной верификации параллельных программ с использованием инвариантов типов данных. Общая методология владения может быть приспособлена для рассуждений в классической логике первого порядка, поддерживаемой большинством современных автоматических средств логического вывода, и ранее была успешно применена в инструменте дедуктивной верификации VCC [20].

В то время как многие локальные методы верификации многопоточных программ допускают использование только блокирующих примитивов синхронизации [19], инструмент VCC реализует модель параллелизма, позволяющую выражать многие варианты использования как блокирующих, так и неблокирующих примитивов. Модель параллелизма VCC является расширением общей методологии *LCI* дополнительными операциями атомарных обновлений асинхронно-изменяемых (англ. volatile) данных. По многим характеристикам инструмент VCC является концептуально схожим с набором Frama-c – Jessie – Why3, в частности, он использует в качестве входного языка подмножество

языка C, расширенное собственным языком спецификаций, предполагает использование некоторой модели типизированных объектов поверх языка C, реализует типизированную модель памяти с явной переинтерпретацией [21], а также использует решатель формул Z3 [9] (поддерживаемый Why3) в качестве средства автоматического логического вывода.

В данной статье предлагается предварительное описание расширения языка спецификаций ACSL, поддерживаемого платформой анализа C-программ Frama-C, набором примитивов методологии LCI, аналогичных тем, которые включены в язык спецификаций для инструмента VCC, а также приводится несколько примеров использования предлагаемых расширений для формализации примитивов синхронизации, включая модель для проверки корректности использования базовых примитивов (внешнего интерфейса) механизма синхронизации RCU, используемого в коде модулей ядра Linux.

2. Методология владения и LCI

Методология владения – это объектно-ориентированная дисциплина синхронизации, предполагающая, что каждый поток управления может осуществлять последовательные (неатомарные) операции записи только в объекты, которыми этот поток *владеет* и может осуществлять любые операции чтения только из объектов, которыми он либо владеет, либо имеет возможность доказать, что во время чтения эти объекты не будут изменены (иначе говоря, останутся *закрытыми*). Объекты как вершины, связанные направленными дугами, соответствующими отношению владения, организуются в лес, в котором каждый объект всегда имеет строго одного владельца, отличного от самого объекта, кроме потоков управления, которые также рассматриваются как объекты, всегда владеющие самими собой. Объектам типизированы и каждому типу объектов приписывается некоторый набор *двухметочных инвариантов*, которые ограничивают возможные операции обновления объектов соответствующего типа и могут быть сформулированы с использованием значений из состояния объекта как до, так и после его обновления. Несколько идущих подряд обновлений одного объекта могут быть объединены в серию так, что каждая такая серия обновлений (операция последовательной записи) может быть представлена для системы в целом как одно атомарное обновление состояния объекта. Это соответствует синхронизации с использованием блокирующих примитивов и при определенных ограничениях на инварианты типов объектов, параллельная программа, использующая только такую блокирующую синхронизацию, может быть без потери общности представлена с использованием *крупнозернистого параллелизма*, который гарантированно корректно аппроксимирует реальный, мелкозернистый параллелизм в предположении корректного использования методологии владения [23]. Для поддержки серий обновлений, а также неатомарной инициализации вновь выделяемых в памяти объектов в условиях методологии с двухметочными инвариантами, все объекты расширяются специальным булевым теневым состоянием, которое отражает, нахо-

дится ли соответствующий объект (с точки зрения текущего потока) в состоянии обновления. В фиксированной точке потока управления программы обновляемые объекты называются *открытыми*, в то время как гарантированно неизменяемые объекты называются *закрытыми*. Корректность использования методологии LCI, таким образом, прямо включает требование того, что все потоки управления должны работать только либо с закрытыми объектами, либо с объектами, которыми они владеют (и, как следствие, могут работать с ними эксклюзивно, в последовательном режиме). Из методологии LCI также прямо следует, что открывать объект должен только поток, который им владеет. В то же время если возможно доказать, что объект остается закрытым, он не может изменен и чтение из него корректно. Двухметочные инварианты предполагаются выполненными только для переходов между состояниями, в которых соответствующие объекты являются закрытыми. Наиболее важным свойством, которое обеспечивается методологией владения, является свойство локальности, означающее, что проверки двухметочных инвариантов для каждой сери операций обновления каждого объекта в отдельности при определенных ограничениях оказывается достаточно для гарантии сохранения инвариантов во всей параллельной программе целиком. Ограничение, налагаемое на двухметочные инварианты для достижения этого свойства называется ограничением *взаимной допустимости*. Двухметочный инвариант (взаимно) *допустим*, если он *рефлексивен*, то есть для него выполнено следствие $I(s_1, s_2) \rightarrow I(s_2, s_2)$ для любой пары состояний s_1 и s_2соответствующего объекта, и *стабилен*, то есть сохраняется при любом переходе, сохраняющем инварианты всех измененных объектов. Взаимная допустимость двухметочных инвариантов – это нелокальное свойство, которое в общем случае может зависеть от любого инварианта какого-либо типа объектов, но это свойство монотонно в том смысле, что будучи выполненным для какого-либо набора инвариантов, оно не может быть нарушено только лишь путем добавления в программу новых типов объектов или инвариантов. Проверка взаимной допустимости всех заданных инвариантов, таким образом, позволяет локализовать проверки сохранения этих инвариантов, каждый раз принимая в рассмотрение только инварианты объектов, непосредственно затрагиваемых при данном обновлении состояния.

В инструменте VCC используется расширение методологии владения, в котором вводится понятие *асинхронно-изменяемых* полей, для которых операции чтения и записи выполняются *атомарно* и разрешены вне зависимости от владения объектом, но только при условии, что они не выполняются одновременно с операциями последовательного обновления состояния объекта, то есть объект при обновлении остается закрытым. Любое атомарное обновление асинхронно-изменяемых полей объекта должно сохранять все его инварианты. Расширение методологии владения асинхронно-изменяемыми полями и атомарными операциями не нарушает свойства локальности всей методологии, но позволяет выразить (или специфицировать) с её помощью более широкий класс механизмов синхронизации.

3. Методология владения и ACSL

Язык спецификации *ACSL* не ориентирован на поддержку методологии владения, поэтому, ожидаемо, существует ряд трудностей, неизбежных как при непосредственной интеграции поддержки методологии владения в этот язык, так и при применении методологии владения поверх этого языка. Наиболее явные и значительные проблемы при этом включают следующие:

- В отличие от VCC, ACSL не обеспечивает и не предполагает использования какой-либо объектно-ориентированной парадигмы поверх языка C и даже не имеет достаточно высокоуровневой семантики. Однако, модуль Jessie в текущей реализации фактически уже осуществляет трансляцию исходных специфицированных C/ACSL-программ в объектно-ориентированный промежуточный язык (также называемый *Jessie* [24]). Таким образом, одним из первых шагов в направлении поддержки методологии владения в языке ACSL является предоставление с помощью него доступа к некоторым возможностям, доступным в промежуточном языке Jessie (или аналогичном объектно-ориентированном языке), в частности, к дополнительным предопределенным теневым полям каждого объекта, требуемым для поддержки методологии владения (например, *closed* и *owns*).
- В VCC (и в методологии владения в целом) отсутствует прямой аналог понятия валидности указателя, принятого в языке ACSL, и доступного через конструкции *valid* и *valid_read*. Наиболее близкими аналогами понятия валидного указателя в методологии владения являются понятия указателя на открытый объект (которым владеет текущий поток) для неатомарных операций и указателя на закрытый объект для атомарных операций. Объединение этих случаев наиболее близко соответствует семантике предиката *valid* (точнее говоря, это корректное нижнее приближение этого предиката). Понятие указателя на закрытый объект или на объект, которым владеет текущий поток, аналогично корректно приближает предикат *valid_read*. В целях упрощения интеграции существующих верифицированных последовательных фрагментов кода в контекст рассматриваемой методологии для параллельных программ предлагается объединить указанные близко соответствующие понятия, выразив их друг через друга в конечных специфицированных программах на языке Why3ML. При этом понятия методологии владения предлагается рассматривать как более примитивные, а предикаты *valid* и *valid_read* – как составные, выразимые через понятия методологии владения. В большинстве случаев это должно позволить части ранее верифицированного последовательного кода в параллельном контексте непосредственно, без каких-либо изменений в соответствующих спецификациях. Типичный пример такого использования – вызов функции, требующей валидно-

сти какого-либо указателя в контексте, где объект, адресуемый этим указателем открыт и его владельцем является текущий процесс.

- Подход к спецификации контрактов функций, принятый в языке ACSL, в частности конструкции *assigns* и *allocates*/*frees*, не имеет прямого соответствия в методологии владения (и в языке спецификаций для инструмента VCC). Семантика конструкции *assigns* не соответствует понятию разрешения на запись объекта в языке спецификаций для инструмента VCC, а конструкции для спецификации контекстных ограничений на состояние памяти (аналоги *allocates*/*frees*) полностью отсутствуют в этом языке. Однако по аналогии с предложенным решением для понятия валидности, рассматриваемые конструкции могут быть относительно легко сопоставлены контекстным ограничениям на запись в открытые объекты (текущего процесса) и на изменение множества объектов, которыми владеет текущий процесс соответственно. Таким образом, если объект остается открытым и принадлежащим текущему процессу (фактически это соответствует последовательному доступу) на протяжении всего выполнения функции (в пре- и пост- состояниях, а также в любом состоянии, в котором управление находится в этой функции), такой объект должен быть специфицирован с помощью конструкции *assigns*. Когда же в результате выполнения функции объект изменяет своего владельца на текущий процесс или с текущего процесса на какой-либо другой объект (возможно, процесс), такой объект должен быть специфицирован с помощью конструкций *allocates*/*frees* соответственно. Такое соответствие может показаться неожиданным для пользователей, поэтому для конструкций *allocates*/*frees* могут быть введены более очевидные синтаксически синонимичные обозначения (например *acquires*/*releases*).
- В последней версии языка ACSL (в версии, соответствующей Frama-C Sodium-20150201 [6]) определяется два вида инвариантов типов данных – сильные и слабые инварианты. Двухметочные инварианты, лежащие в основе методологии владения, не являются ни теми, ни другими, поэтому их предлагается непосредственно интегрировать в язык ACSL. Далее в данной статье для обозначения двухметочных инвариантов типов используется ключевое слово 2state.
- Еще одним не обязательным, но весьма желательным в контексте методологии владения, расширением языка ACSL является возможность объявлять теневые структурные типы с заданными инвариантами из любой точки программного кода (а не только в глобальном контексте). Это расширение наиболее важно для упрощения использования динамических утверждений (см. секцию 3.2).

4. Методология владения на примерах

4.1 Спин-блокитровки

Рассмотрим теперь пример достаточно простого механизма синхронизации, формализованного с использованием методологии владения. Намеренно упрощенная спецификация для механизма спин-блокировок, представленная на рис. 1, иллюстрирует предлагаемый подход к интеграции методологии владения в язык ACSL. Представленный пример в основном повторяет аналогичный пример спецификации из документации VCC [25], перенесенный на язык ACSL, однако он также демонстрирует некоторые особенности, специфичные для языка ACSL. Наиболее заметная из них – использование специально вводимого предиката *\new* для различения указателей на открытые и закрытые вновь выделенные объекты. Используемый в ACSL предикат *\fresh*, следуя предлагаемой семантике для валидности указателей, должен быть выполнен в обоих этих случаях. Семантики операций закрытия вновь выделенных и ранее открытых объектов отличаются, так как требуют выполнения различных проверок инвариантов (например, инвариант counter != *\old*(counter) → counter = *\old*(counter) + 1 невозможно доказать для операции закрытия после инициализации объекта). ACSL также требует явной спецификации контекстных ограничений на множество объектов, которыми владеет текущий процесс. В примере также используются вводимые встроенные примитивы *\acquire* и *\release* для изменения значений предопределенных теневых полей *\owner* и *\owns*.

```
volatile_owns struct spinlock {
  volatile unsigned int slock;
  //@ ghost void *resource;
};

/*@ 2state type invariant
  @   same_resource(struct spinlock l) = \old(l.resource) ==
l.resource;
  @*/

/*@ 2state type invariant
  @   ownership(struct spinlock l) =
  @     !l.slock ==> \subset(resource, \owns(&l));
  @
  @*/

/*@ requires \owned(obj) && \closed(obj);
  @ requires \valid(l) && \new(l);
  @ requires \owns(l) == \empty;
  @ frees obj;
  @ ensures \closed(l);
  @ ensures l->resource == obj;
  @*/
void spin_lock_init(struct spinlock *l /*@ ghost void *obj */)
```

```
{
  l->slock = 0;
  /*@ ghost {
    @   l->resource = obj;
    @   \release(obj, l);
    @   \close(l);
    @ }
    @*/
}

/*@ requires \closed(l);
  @ allocates l->resource;
  @ ensures \owned(l->resource) && \closed(l->resource);
  @*/
void spin_lock(struct spinlock *l)
{
  int stop = 0;
  do {
    /*@ atomic (l) */ {
      stop = !cmpxchg(&l->slock, 1, 0);
      /*@ ghost
        @   if (stop)
        @     \acquire(l->resource, l);
        @*/
    }
  } while (!stop);
}

/*@ requires \closed(l);
  @ requires \owned(l->resource) && \closed(l->resource);
  @ frees l->resource;
  @*/
void spin_unlock(struct lock *l)
{
  /*@ atomic (l) */ {
    l->slock = 0;
    //@ ghost \release(l->resource, l);
  }
}
```

Рис. 1. Пример спецификации спин-блокировок.

4.2 Динамические утверждения

Так же как и пример упрощенной спецификации спин-блокировок из документации VCC, его рассмотренная ACSL-версия изначально делает не приемлемые в реальности предположения, закоючающиеся в требовании в предусловиях примитивов синхронизации выполненности предикатов *closed*(l). Проблема такого предусловия заключается в том, что неясно, как оно может

быть непосредственно доказано для потока, не владеющего объектом *l*. Для решения этой проблемы в инструменте VCC вводится специальное понятие *динамического утверждения*, которое представляет собой объект определенного типа (с соответствующим набором инвариантов), специально предназначенный для того, чтобы им мог владеть некоторый поток управления и, выполняя над этим объектом операции открытия/закрытия, использовать инварианты его типа для доказательства закрытости (и, возможно, некоторых других свойств) других объектов. Для упрощения использования механизма динамических утверждений в VCC вводятся специальное предопределенное теневое поле *\claim_counter*, которое проверяется в операциях открытия объектов и может быть изменено только с помощью специальных конструкций выделения/освобождения динамических утверждений. Пример объявления типа структуры для динамического утверждения приведен на рис. 2. В этом определении используется специальный атрибут *claim*, который позволяет добавить к типу структуры неявное поле *claimed* и двухметочный инвариант, который не является взаимно допустимым без учета дополнительного предусловия вида *\claim_counter* = 0 для операций открытия объектов. Таким образом, поддержка динамических утверждений требует введения дополнительных предусловий для операции открытия объектов. Эти предусловия могут быть либо фиксированными, либо задаваемыми пользователем для каждого типа структур аналогично двухметочным инвариантам. Введение фиксированных предусловий и встроенной поддержки динамических утверждений может быть обосновано большой частотой их использования в спецификациях параллельных программ. В варианте методологии владения, реализованной в инструменте VCC, роль динамических утверждений не ограничивается использованием их для доказательства закрытости объектов, принадлежащих множеству *claimed*. За счет формулирования дополнительных двухметочных инвариантов динамические утверждения в инструменте VCC превращаются в аналоги проверочных утверждений, а также пред- и постусловий в условиях параллельной среды. Поэтому конструкция выделения экземпляров динамических утверждений в VCC позволяет непосредственно указывать произвольный инвариант (предикат над состоянием совокупности закрытых объектов) при создании динамического утверждения. Предполагаемый синтаксис соответствующей конструкции в языке ACSL – *\claim(тег_структуры, указ._на_объект_1, .., указ._на_объект_n, предикат)*. В предикате предполагается возможность добавления полей к типу структуры динамического утверждения с помощью конструкции *выражение \as имя_поля*. Так как создание динамических утверждений выполняется для предикатов над совокупностью закрытых объектов, поля *\claim_counter* являются асинхронно-изменяемыми у всех типов структур. Поддержка конструкции *\claim* требует добавления возможности неявного объявления типов структур динамических утверждений вместе с соответствующими инвариантами, указываемыми при выделении экземпляров этих утверждений. Операции выделения динамических утверждений могут осуществляться только внутри атомарных секций, что в принципе позволяет син-

хронизировать операции изменения теневых счетчиков *\claim_counter* и реальных счетчиков ссылок с помощью соответствующих инвариантов и осуществлять соответствующие операции внутри одних тех же атомарных секций.

```
/*@ ghost claim struct claim {
  @  set<void*> claimed;
  @ };
  @
  @ // 2state type invariant
  @ //  claim(struct claim c) =
  @ //    \forall void *o;
  @ //      \subset(o, c.claimed) ==>
  @ //      \closed(o);
  @*/
```

Рис. 2. Пример спецификации типа структуры динамического утверждения.

4.3 Механизм синхронизации RCU

Read-copy-update (сокращенно RCU [18]) – это механизм синхронизации с поддержкой неблокирующих операций чтения и частично неблокирующей операции записи. В терминах механизма синхронизации RCU запись называется обновлением и осуществляется в две отдельных фазы. Первая фаза, фаза удаления, сводится к защищенному барьером синхронизации памяти атомарному обновлению значения защищенной механизмом синхронизации RCU указательной переменной на значение NULL (или другое предопределенное значение невалидного указателя), либо на значение адреса полностью и корректно инициализированного объекта (и, таким образом, готового к выполнению над ним произвольных операций чтения). Вторая фаза, фаза освобождения, состоит в эксклюзивном (обычно защищаемом блокировками) освобождении памяти, занимаемой ранее удаленным по завершению соответствующей предшествующей первой фазы устаревшим объектом. Две стадии обновления разделяются с помощью специальной блокирующей операции синхронизации, которая гарантирует, что соответствующий устаревший объект (уже удаленный, но еще не освобожденный) останется валидным и доступным для чтения до тех пор, пока все операции чтения этого объекта во всех параллельно выполняющихся потоках не будут гарантированно завершены. Для обеспечения этой гарантии операции чтения из всех защищаемых механизмом RCU указательных переменных должны быть обрамлены парой специальных ограничительных операций начала/конца критической сессии. Кроме этого, когерентность (свежесть) значений защищаемых переменных, из которых происходит чтение внутри критической секции, должна быть явно обеспечена хотя бы однажды внутри любой критической секции, причем строго до первого доступа к этим переменным внутри секции. Выполнение этого условия обеспечивается с помощью специальной операции защищенного разыменования. При

выполнении описанных условий блокирующая операция, осуществляемая в ходе обновления, может просто ожидать окончания всех критических секций, оставшихся открытыми по завершении фазы удаления, что автоматически гарантирует полное завершение всех операций чтения устаревшего объекта к моменту завершения блокирующей операции ожидания. Таким образом, сущность реализации механизма синхронизации RCU заключается в предоставлении следующих пяти основных примитивов: 1) защищенная барьером операция атомарного обновления значения для первой фазы обновления (в ядре Linux соответствующий примитив называется *rcu_assign_pointer*); 2) блокирующая операция, разделяющая две фазы обновления (в Linux она называется *synchronize_rcu*); 3) операция входа в критическую секцию чтения (*rcu_read_lock*); 4) операция выхода из критической секции чтения (*rcu_read_unlock*); 5) операция защищенного разыменования (*rcu_dereference*). Реализация механизма RCU в ядре Linux позволяет для большинства процессорных архитектур сделать реализацию операций 3–5 пустой, переложив таким образом все накладные расходы на синхронизацию на операции 1 и 2. Кроме этого, фаза освобождения вместе с предшествующей ей блокирующей операцией ожидания может быть выполнена в параллельном процессе. Это обеспечивает высокую эффективность и частоту использования механизма RCU в ядре Linux [26].

На рис. 3 показан пример упрощенной спецификации примитивов синхронизации механизма RCU, предназначенной для проверки корректности использования этого механизма в модулях ядра Linux. Пример служит для предварительной иллюстрации применимости расширенной (атомарными операциями) методологии владения для верификации модулей ядра Linux, активно [26] использующих предоставляемый основной частью ядра интерфейс механизма синхронизации RCU. Внутренняя (для основной части ядра) реализация механизма RCU, вполне вероятно, не может быть верифицирована в рамках существующих расширений методологии владения, так как в ней существенную роль играет слабая консистентность, не моделируемая в рамках методологии владения. Наибольшую сложность при спецификации внешнего интерфейса механизма RCU представляет формализация понятия защищенной указательной переменной, доступной для разыменования только внутри соответствующей критической секции, а также обеспечение эксклюзивности при выполнении операции освобождения.

```
/*@ axiomatic rcu {
  @   type rcu_section = integer;
  @
  @   2state type invariant
  @     rcu_secion(rcu_section s) = \owned(&s);
  @
  @   logic struct lock *rcu_lock(void **loc, void *obj);
  @   predicate rcu_reclaimable(void *obj);
  @ }
  @
  @ ghost rcu_section current_section = 0;
```

```
  @ ghost set<void *> rcu_dereferenced = \empty;
  @*/

/*@ requires current_section == 0;
  @ assigns current_section;
  @ ensures current_section != 0;
  @*/
void rcu_read_lock(void);

/*@ requires current_section != 0;
  @ assigns current_section;
  @ assigns rcu_dereferenced;
  @ frees rcu_dereferenced;
  @ ensures current_section == 0;
  @ ensures rcu_dereferenced == \empty;
  @*/
void rcu_read_unlock(void);

/*@ requires rcu_lock(loc, obj) == l;
  @ requires rcu_reclaimable(obj);
  @ allocates l;
  @ ensures \closed(l);
  @*/
void synchronize_rcu(void /*@ ghost void **loc, void *obj, struct lock
*l */);

/*@ requires \owned(l) && \closed(l);
  @ requires l->resource == obj;
  @ requires \claim_count(l) == 0;
  @ frees l;
  @ ensures rcu_lock(loc, obj) == l;
  @ ensures rcu_reclaimable( *loc);
  @*/
void rcu_protect(void **loc, void *obj, struct lock *l);

#define rcu_assign_pointer(p, v, l) \
  {\
    rcu_protect(&p, v, l);\
    p = v;\
    v;\
  }

/*@ requires current_section != 0;
  @ requires rcu_lock(loc, obj) != NULL;
  @ assigns rcu_dereferenced;
  @ allocates obj;
  @ ensures obj != NULL ==> \closed(obj);
  @ ensures rcu_dereferenced =
  @         \union(\old(rcu_dereferenced), obj);
  @*/
void *rcu_deref(void **loc, void *obj)

#define rcu_dereference(p, c) \
  ((typeof(p)) rcu_deref(&p, p));
```

Рис. 3. Упрощенная спецификация интерфейса механизма RCU.

5. Заключение

В статье предложено расширение языка спецификаций ACSL, поддержка которого реализована в наборе инструментов Frama-C–Jessie–Why3, с целью поддержки верификации параллельного кода с использованием методологии владения. В статье также рассмотрены примеры применения предложенного расширения для спецификации спин-блокировок и внешнего интерфейса механизма синхронизации RCU, широко используемого модулями ядра Linux. Предложенная формализация, однако, не была формально верифицирована.

Литература

[1]. http://linuxtesting.org/astraver.

[2]. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski, “FRAMA-C, A Software Analysis Perspective”, Proceedings of International Conference on Software Engineering and Formal Methods 2012 (SEFM’12), October 2012.

[3]. L. Correnson, Z. Dargaye, A. Pacalet, “WP (Draft) Manual”',

[4]. http://frama-c.com/download/frama-c-wp-manual.pdf.

[5]. Y. Moy. Automatic Modular Static Safety Checking for C Programs: Ph.D. Thesis. Université Paris-Sud. – 2009. – January. http://www.lri.fr/~marche/moy09phd.pdf.

[6]. http://kerneis.github.io/cil/doc/html/cil/.

[7]. P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, V. Prevosto, “ACSL: ANSI/ISO C Specification Language. Version 1.7”, http://frama-c.com/download/acsl.pdf.

[8]. F. Bobot, J.-C. Filliâtre, C. Marché, A. Paskevich, “Why3: Shepherd your herd of provers”, Boogie 2011: First International Workshop on Intermediate Verification Languages, 2011.

[9]. J.-C. Filliâtre, A. Paskevich, “Why3 – where programs meet provers", In Programming Languages and Systems, pp. 125–128, Springer Berlin Heidelberg, 2013.

[10]. L. De Moura, N. Bjørner, “Z3: An efficient SMT solver”', In Tools and Algorithms for the Construction and Analysis of Systems, pp. 337—340, Springer Berlin Heidelberg, 2008.

[11]. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, C. Tinelli, “CVC4”, In Computer aided verification, pp. 171—177, Springer Berlin Heidelberg, January, 2011.

[12]. A. Riazanov, A. Voronkov, “The design and implementation of Vampire”, In AI communications, vol. 15(2, 3), pp. 91—110, 2002.

[13]. S. Schulz. “System Description: E 1.8”', In Proceedings of the 19th LPAR, Stellenbosch, pp. 477—483, LNCS 8312, Springer Verlag, 2013.

[14]. Y. Bertot, P. Castéran, “Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions”', Springer Science & Business Media, 2013.

[15]. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, M. Srivas, “PVS: Combining Specification, Proof Checking, and Model Checking”', In proceedings of Computer-Aided Verification '96, pp. 411—414, 1996.

[16]. T. Hubert, C. Marché, “Separation analysis for deductive verification”', In Heap Analysis and Verification, Braga, Portugal, March, 2007.

[17]. J.-P. Talpin, P. Jouvelot, “Polymorphic type region and effect inference”, Technical Report EMP-CRI E/150, 1991.

[18]. M. Mandrykin, A. Khoroshilov, “High level memory model with low level pointer cast support for Jessie intermediate language”, In Programming and Computer Software, Vol. 41, No. 4, pp. 197—208, 2015.
[19]. P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, M. Soni, “Read-copy update”, In AUUG Conference Proceedings, p. 175, 2001.
[20]. C. Flanagan, S. N. Freund, S. Qadeer, “Thread-modular verification for shared-memory programs”, In ESOP 2002, Number 2305 in LNCS, Springer, pp. 262—277, 2002.
[21]. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, S. Tobies, “VCC: A practical system for verifying concurrent C”, In Theorem Proving in Higher Order Logics, pp. 23—42, Springer Berlin Heidelberg, 2009.
[22]. E. Cohen, M. Moskal, S. Tobies, W. Schulte, “A precise yet efficient memory model for C”, In Electronic Notes in Theoretical Computer Science, vol. 254, pp. 85—103. 2009.
[23]. E. Cohen, M. Moskal, W. Schulte, S. Tobies, “Local Verification of Global Invariants in Concurrent Programs”, In Computer Aided Verification, Springer Berlin Heidelberg, pp. 480—494, January, 2010.
[24]. E. Cohen, M. Moskal, W. Schulte, S. Tobies. “A practical verification methodology for concurrent programs”, Tech. Rep. MSR-TR-2009-15, Microsoft Research, 2009. (http://research.microsoft.com/pub)
[25]. J.-C. Filliâtre, C. Marché, “The Why/Krakatoa/Caduceus platform for deductive program verification”, In Proceedings of the 19th International Conference on Computer Aided Verification, Lecture Notes in Computer Science, Springer, 2007.
[26]. M. Moskal, W. Schulte, E. Cohen, M. A. Hillebrand, S. Tobies, “Verifying C programs: a VCC tutorial”', MSR Redmond, EMIC Aachen, 2012.
[27]. P.E. McKenney, S. Boyd-Wickizer, J. Walpole, “RCU usage in the Linux kernel: one decade later”, Technical report, 2013.

Towards Deductive Verification of C Programs with Shared Data

[1, 2, 3, 4]A.V. Khoroshilov <khoroshilov@ispras.ru>
[1]M.U. Mandrykin <mandrykin@ispras.ru>
[1] Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., 109004, Moscow, Russia
[2]Lomonosov Moscow State University, 2nd Education Building, Faculty CMC, GSP-1, Leninskie Gory, Moscow, 119991, Russian Federation
[3]Moscow Institute of Physics and Technology, 9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia
[4]Higher School of Economics, National Research University, 20 Myasnitskaya Ulitsa, Moscow 101000, Russia

Abstract. The paper takes a look at the problem of deductive verification of Linux kernel code that is concurrent and involves accesses to shared data and interactions with highly concurrent environment. The presence of shared data does not allow to apply traditional deductive verification techniques based solely on function contracts and loop invariants, so we

consider verification of such code using type invariants and a particular object-oriented methodology. For Linux kernel modules one of the usual goals of deductive verification is a formal proof of the code's compliance to a specification of a synchronization discipline. We propose formalizing both the synchronization discipline and the required properties of the code in terms of ownership methodology with locally checked invariants (LCI) that was previously successfully applied for verifying the Microsoft Hyper-V Hypervisor with VCC deductive verification tool. However to maintain good compatibility with the various specific C features and extensions used in the Linux kernel code, efficiently handle data type representations with many large nested structure definitions and provide a reacher specification language we propose using Frama-C static analysis platform with its ACSL specification language and Jessie plugin for deductive verification as these tools have shown a good applicability to verification of sequential Linux kernel code fragments in the course of the Astraver project. The paper presents preliminary discussion of issues arising from integration of support for the ownership methodology and LCI into both the ACSL specification language and its underlying toolset. In particular, the issue of reusing existing ACSL specifications in sequential context and the related issue of establishing a correspondence between ACSL notion of validity and LCI notions of owned, wrapped and closed object are discussed. The overall approach to specification of concurrent kernel code using ownership methodology, LCI and ACSL specification language is demonstrated on examples of spinlock specification and a simplified specification of RCU (Read-copy-update) API.

Keywords: Verification, concurrency, invariants, C programming language.

DOI: 10.15514/ISPRAS-2015-27(4)-4

For citation: Khoroshilov A.V., Mandrykin M.U. Towards Deductive Verification of C Programs with Shared Data. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 4, 2015, pp. 49-68 (in Russian). DOI: 10.15514/ISPRAS-2015-27(4)-4.

References

[1]. http://linuxtesting.org/astraver.
[2]. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski, “FRAMA-C, A Software Analysis Perspective”, Proceedings of International Conference on Software Engineering and Formal Methods 2012 (SEFM’12), October 2012.
[3]. L. Correnson, Z. Dargaye, A. Pacalet, “WP (Draft) Manual”',
[4]. http://frama-c.com/download/frama-c-wp-manual.pdf.
[5]. Y. Moy. Automatic Modular Static Safety Checking for C Programs: Ph.D. Thesis. Université Paris-Sud. – 2009. – January. http://www.lri.fr/~marche/moy09phd.pdf.
[6]. http://kerneis.github.io/cil/doc/html/cil/.
[7]. P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, V. Prevosto, “ACSL: ANSI/ISO C Specification Language. Version 1.7”, http://frama-c.com/download/acsl.pdf.
[8]. F. Bobot, J.-C. Filliâtre, C. Marché, A. Paskevich, “Why3: Shepherd your herd of provers”, Boogie 2011: First International Workshop on Intermediate Verification Languages, 2011.
[9]. J.-C. Filliâtre, A. Paskevich, “Why3 – where programs meet provers", In Programming Languages and Systems, pp. 125–128, Springer Berlin Heidelberg, 2013.

[10]. L. De Moura, N. Bjørner, “Z3: An efficient SMT solver”, In Tools and Algorithms for the Construction and Analysis of Systems, pp. 337—340, Springer Berlin Heidelberg, 2008.
[11]. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, C. Tinelli, “CVC4”, In Computer aided verification, pp. 171—177, Springer Berlin Heidelberg, January, 2011.
[12]. A. Riazanov, A. Voronkov, “The design and implementation of Vampire”, In AI communications, vol. 15(2, 3), pp. 91—110, 2002.
[13]. S. Schulz. “System Description: E 1.8”, In Proceedings of the 19th LPAR, Stellenbosch, pp. 477—483, LNCS 8312, Springer Verlag, 2013.
[14]. Y. Bertot, P. Castéran, “Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions”, Springer Science & Business Media, 2013.
[15]. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, M. Srivas, “PVS: Combining Specification, Proof Checking, and Model Checking”, In proceedings of Computer-Aided Verification '96, pp. 411—414, 1996.
[16]. T. Hubert, C. Marché, “Separation analysis for deductive verification”, In Heap Analysis and Verification, Braga, Portugal, March, 2007.
[17]. J.-P. Talpin, P. Jouvelot, “Polymorphic type region and effect inference”, Technical Report EMP-CRI E/150, 1991.
[18]. M. Mandrykin, A. Khoroshilov, “High level memory model with low level pointer cast support for Jessie intermediate language”, In Programming and Computer Software, Vol. 41, No. 4, pp. 197—208, 2015.
[19]. P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, M. Soni, “Read-copy update”, In AUUG Conference Proceedings, p. 175, 2001.
[20]. C. Flanagan, S. N. Freund, S. Qadeer, “Thread-modular verification for shared-memory programs”, In ESOP 2002, Number 2305 in LNCS, Springer, pp. 262—277, 2002.
[21]. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, S. Tobies, “VCC: A practical system for verifying concurrent C”, In Theorem Proving in Higher Order Logics, pp. 23—42, Springer Berlin Heidelberg, 2009.
[22]. E. Cohen, M. Moskal, S. Tobies, W. Schulte, “A precise yet efficient memory model for C”, In Electronic Notes in Theoretical Computer Science, vol. 254, pp. 85—103. 2009.
[23]. E. Cohen, M. Moskal, W. Schulte, S. Tobies, “Local Verification of Global Invariants in Concurrent Programs”, In Computer Aided Verification, Springer Berlin Heidelberg, pp. 480—494, January, 2010.
[24]. E. Cohen, M. Moskal, W. Schulte, S. Tobies. “A practical verification methodology for concurrent programs”, Tech. Rep. MSR-TR-2009-15, Microsoft Research, 2009. (http://research.microsoft.com/pub)
[25]. J.-C. Filliâtre, C. Marché, “The Why/Krakatoa/Caduceus platform for deductive program verification”, In Proceedings of the 19th International Conference on Computer Aided Verification, Lecture Notes in Computer Science, Springer, 2007.
[26]. M. Moskal, W. Schulte, E. Cohen, M. A. Hillebrand, S. Tobies, “Verifying C programs: a VCC tutorial”, MSR Redmond, EMIC Aachen, 2012.
[27]. P.E. McKenney, S. Boyd-Wickizer, J. Walpole, “RCU usage in the Linux kernel: one decade later”, Technical report, 2013.