Комбинация методов статической верификации композиции требований

В.О. Мордань <mordan@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, 25

Аннотация. Статическая верификация программного обеспечения доказывает выполнение требований в программах, однако для этого необходимо большое количество вычислительных ресурсов, кроме того, соответствующая задача не всегда может быть решена. На данный момент не существует универсальный метод статической верификации, который мог бы эффективно проверять произвольные программы, поэтому на практике необходимо выбирать более подходящий метод и настраивать его под конкретную задачу. В данной статье предлагается комбинировать различные методы для повышения производительности и улучшения результата верификации, что можно рассматривать как первый шаг в создании универсального метода статической верификации. Предложенные методы были реализованы для комбинации активно развивающихся в настоящее время методов верификации композиции требований. Апробация реализованных методов на модулях ядра операционной системы Linux продемонстрировала их преимущества относительно отдельного применения методов верификации композиции требований.

Ключевые слова: статическая верификация программного обеспечения; уточнение абстракции по контрпримерам; задача достижимости; композиция требований.

DOI: 10.15514/ISPRAS-2017-29(3)-9

Для цитирования: Мордань В.О. Комбинация методов статической верификации композиции требований. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 151-170. DOI: 10.15514/ISPRAS-2017-29(3)-9

1. Введение

Статическая верификация программного обеспечения предназначена для проверки исходного кода программы без его выполнения, при этом рассматриваются все возможные пути выполнения программы. Главное достоинство статической верификации заключается в том, что она нацелена на доказательство отсутствия ошибок, являющихся нарушениями проверяемого требования. Однако в общем случае статическая верификация является

неразрешимой задачей, поскольку она сводится к проблеме останова [1]. Кроме того, на практике статическая верификация требует большого количества вычислительных ресурсов (таких, как процессорное время и оперативная память), что существенно затрудняет ее применение на практике. Поэтому статическую верификацию имеет смысл использовать только для проверки программного обеспечения с высокими требованиями надежности.

В качестве примера подобного программного обеспечения можно рассмотреть ядро операционной системы Linux, которое состоит из более 20 миллионов строк кода на языке программирования C, новая версия которого выходит каждые 2-3 месяца [2]. При этом существуют сотни требований на корректное использование интерфейсов сердцевины ядра [3], которым должен удовлетворять код каждого модуля ядра Linux, число которых составляет порядка 5 тысяч. Если в модуле нарушается одно из подобных требований, то это может привести к отказу всей операционной системы. Статическая верификация способна доказывать, что в модулях нет нарушений проверяемых требований. При проведении статической верификации необходимо выполнить проверку каждого подобного требования в каждом модуле каждой новой версии ядра, поэтому задачи эффективной проверки выполнения композиции требований и нахождения большего числа нарушений требований являются важными и актуальными.

Для решения данных задач были предложены различные методы статической верификации композиции требований [4-7], каждый из которых имеет как преимущества, так и недостатки, а также свою область применимости. Поэтому на практике необходимо правильно выбрать наиболее подходящий метод и множество параметров для него с учетом решаемых задач, что позволит как снизить потребляемые ресурсы, так и успешно решить большее количество задач. В то же самое время известны примеры, в которых комбинация различных методов позволяла улучшить те или иные характеристики верификации [8]. В данной статье будут рассмотрены основные идеи комбинации методов на примере верификации композиции требований для повышения двух основных характеристик – производительности верификации и числа успешно решенных задач.

Следующий раздел содержит краткое описание существующих методов статической верификации композиции требований.

2. Методы верификации композиции требований

С помощью статической верификации проверяются требования к программам, которые сводятся к проблеме достижимости. Процесс верификации состоит из двух этапов – подготовки и решения задачи достижимости. Задача достижимости формулируется следующим образом – требуется доказать недостижимость некоторых точек программы, которые соответствуют нарушениям проверок выполнения требований, из точки входа в программу. Поскольку в общем случае задача достижимости является неразрешимой, то на

практике она решается с ограниченными ресурсами. Результатом верификации является либо доказательство отсутствия нарушений требования (вердикт *safe*), либо пример нарушения требования в виде трассы ошибки (вердикт *unsafe*), либо невозможность решить задачу по тем или иным причинам (вердикт *unknown*), например, из-за исчерпания выделенных на решение задачи вычислительных ресурсов.

Для постановки задач достижимости существует два подхода – это инструментирование исходного кода программы [9] (то есть добавление проверок выполнения требования непосредственно в код) и наблюдательные автоматы [10] (то есть передача модели требования верификатору в его внутреннем представлении независимо от кода). Метод инструментирования предоставляет более широкие возможности по формализации требований, поскольку он использует возможности языка программирования исходной программы. Однако при этом усложняется исходный код программы за счет добавленных проверок, что особенно негативно сказывается при проверке композиции требований.

Для решения задач достижимости наиболее масштабируемым на данный момент является подход уточнения абстракции по контрпримерам (counterexample guided abstraction refinement, или CEGAR [11]). Инструменты, реализующие данный подход, являются одними из лидеров в международных мероприятиях по верификации Competition on Software Verification [12]. Помимо этого, подход CEGAR успешно используется и на практике: в инструментах BLAST [13] и CPAchecker [14] для верификации модулей ядра операционной системы Linux (в рамках системы верификации Linux Driver Verification Tools, или LDV Tools [15]) и в инструменте SLAM [16] для верификации драйверов операционной системы Windows (в рамках проекта Static Driver Verification). Именно поэтому в данной статье в качестве базового подхода решения задач достижимости будет рассматриваться CEGAR.

Для верификации композиции требований необходимо разрешить две проблемы – это чрезмерный рост числа состояний, вызванный тем, что хотя бы одно из проверяемых требований не может быть верифицировано с выделенными ограничениями на ресурсы, и остановка верификации после нахождения первого нарушения требования.

2.1. Метод обнаружения всех однотипных нарушений

Для решения проблемы остановки верификации после нахождения первого нарушения требования был предложен метод *обнаружения всех однотипных нарушений* (ОВН) [4], который продолжает верификацию после нахождения нарушения требования и производит анализ результата, то есть найденных трасс ошибок. Основная задача анализа результата состоит в исключении так называемых «наведенных» трасс ошибок (то есть соответствующих одному и тому же нарушению требования). Например, имеется утечка памяти в функции, которая может вызываться произвольное количество раз, и для каждого случая

выдается отдельная трасса ошибки. Для того чтобы не пропустить новые нарушения требования, предлагается автоматически фильтровать трассы ошибок по относительно простым критериям (например, на полное совпадение). Далее трассы анализирует человек, помечая и исправляя причину нарушения требования. При этом все трассы ошибок, которые содержат помеченный человеком фрагмент, исключаются из дальнейшего анализа, то есть человек рассматривает ровно столько трасс, сколько уникальных нарушений требований найдено. По сравнению с базовым методом статической верификации метод ОВН способен находить примерно в 1.5 раза больше нарушений требований, при этом время проведения ручного анализа результата возрастает пропорционально числу нарушений требований (то есть также примерно в 1.5 раза) [6, 17].

2.2. Метод условной многоаспектной верификации

Метод *условной многоаспектной верификации* (УМАВ) был предложен для верификации композиции требований [5]. Для предотвращения чрезмерного роста числа состояний время проверки каждого требования ограничивается при проверке композиции требований, то есть выделенные на решение задачи ресурсы распределяются равномерным образом между проверкой каждого требования. Для этого используется предположение о том, что единовременно проверяется только одно требование, которое достаточно точно может быть определено из найденного контрпримера в подходе CEGAR. После нахождения нарушения некоторого требования верификация продолжается, но без нарушенного требования, что позволит получить результат для остальных требований без необходимости проведения ручной фильтрации. За счет переиспользования знаний о верификации между проверкой тех требований, которые не ведут к чрезмерному росту числа состояний, данный метод позволяет повысить производительность верификации композиции требований в несколько раз при незначительных потерях результата (порядка 1-2%) [5, 17], которые вызваны тем, что используемое в методе предположение не всегда работает.

2.3. Метод автоматных спецификаций

Предложенный метод условной многоаспектной верификации обладает следующим недостатком – инструментирование исходного кода усложняет задачи достижимости, поскольку дополнительные проверки добавляются в исходный код программы, и чем больше проверяемых требований, тем больше будут усложняться задачи. Кроме того, при инструментировании проверки требований находятся в исходном коде и нет возможности их удалить во время верификации, например, если было найдено нарушение требования и больше его не нужно проверять. Для решения данной проблемы был предложен метод *автоматных спецификаций* (АС) [7], который формализует требование

независимо от исходного кода во внутреннем представлении верификатора, расширяя наблюдательные автоматы за счет добавления произвольных конструкций языка С во внутреннее представление программы во время верификации, что делает возможности данного метода по представлению требований эквивалентными инструментированию. Метод АС демонстрирует сопоставимые результаты с методом инструментирования [17], при этом больше подходит для верификации композиции требований.

2.4. Метод декомпозиции автоматной спецификации

Поставленную в данной работе цель можно переформулировать следующим образом – дана спецификация, состоящая из набора требований, и необходимо найти ее разбиение на группы требований для совместной верификации. Так, в методе УМАВ все требования помещаются в одну группу, а при последовательной верификации – в каждую группу помещается ровно одно требование. Метод *декомпозиции автоматной спецификации* (ДАС) [7] проверяет отдельно (то есть в разных группах) те требования, которые могут приводить к чрезмерному росту числа состояний, и совместно (в одной группе) все остальные требования. Верификация каждой группы требований производится с помощью метода АС. Все идеи метода УМАВ, которые не относятся к подходу CEGAR, например, продолжение верификации после нахождения нарушения требования, но без нарушенного требования, также используются и в методе ДАС. На практике данный метод требует несколько больше ресурсов на решение задач достижимости, чем метод УМАВ, но при этом демонстрирует меньший процент потерь результата за счет того, что в нем не используются эвристики [7, 17].

2.5. Простейшие комбинации методов

Классификация всех приведенных методов верификации композиции требований приведена в табл. 1. Заметим, что для решения определенных задач можно применять комбинацию методов. Так, например, для проверки выполнения композиции требований с возможностью нахождения нескольких нарушений требований можно использовать методы УМАВ и ОВН совместно [6], так как анализ результата в методе ОВН производится уже после решения задач достижимости методом УМАВ. Комбинация методов УМАВ и ОВН позволяет получить результат метода ОВН с незначительными потерями, при этом требуется практически то же количество ресурсов, что и в методе УМАВ [6, 17].

В следующем разделе будет дана основная идея комбинации методов верификации композиции требований в общем виде.

Табл. 1. Методы верификации композиции требований
Table 1. Verification methods for checking requirements composition

Метод	Постановка задач достижимости	Верификация композиции требований	Остановка после нахождения первого нарушения
Базовый	Инструментирование	Нет	Нет
ОВН	Инструментирование	Нет	Да
УМАВ	Инструментирование	Да	Нет
УМАВ с ОВН	Инструментирование	Да	Да
АС	Автоматы	Нет	Нет
АС с ОВН	Автоматы	Нет	Да
ДАС	Автоматы	Да	Нет
ДАС с ОВН	Автоматы	Да	Да

3. Основная идея комбинации методов

Пусть для верификации дана спецификация $\Omega = \{\omega_1, ..., \omega_n\}$, состоящая из n различных требований, и выделено T единиц ресурсов (например, процессорных секунд). Под композицией требований будем понимать непустое подмножество всех требований (группу): $\xi \subseteq \Omega, \xi \neq \varnothing$. Формально метод верификации композиции требований представляет собой оператор вида $\psi(\xi, T)$: $\xi \to V$, который для каждого требования возвращает вердикт $v \in V = \{safe, unsafe, unknown\}$. Тогда *комбинацией* методов $\Psi = \{\psi_1(\xi_1, t_1), ..., \psi_k(\xi_k, t_k)\}$ верификации композиции требований Ω называется следующий оператор:

$$C(\Psi, \Omega): \quad \Omega \to V.$$

Иными словами, комбинация методов должна получить вердикт для каждого из проверяемых требований, используя для этого заданные методы верификации. Основная цель комбинации методов заключается в том, чтобы либо получить результат быстрее, либо успешно решить больше задач достижимости.

Комбинация методов может выполняться как параллельно, так и последовательно.

3.1. Параллельная комбинация

Наиболее простой является *параллельная* (или *статическая*) комбинация, схема которой приведена на рис. 1.

Основная идея параллельной комбинации состоит в том, что до начала верификации с помощью оператора *разбиения* создаются группы требований:

$$\{\xi_1,...,\xi_k\},\quad \xi_i \subset \Omega,\quad \xi_i \cap \xi_j = \varnothing,\quad \forall i \neq j,\quad i,j \in \overline{1,k},\quad \cup_{i=1}^{k} \xi_i = \Omega,$$

каждой из которых также назначается один из методов для верификации. Далее методы производят верификацию соответствующих групп (возможно, параллельно), после чего результат их работы объединяется. При этом предполагается, что ресурсы, выделенные на верификацию всех требований, распределяются для каждого метода пропорционально числу верифицируемых требований, то есть на верификацию группы ξ_i будет выделено $t_i = T\frac{|\xi_i|}{n}$ единиц ресурсов.

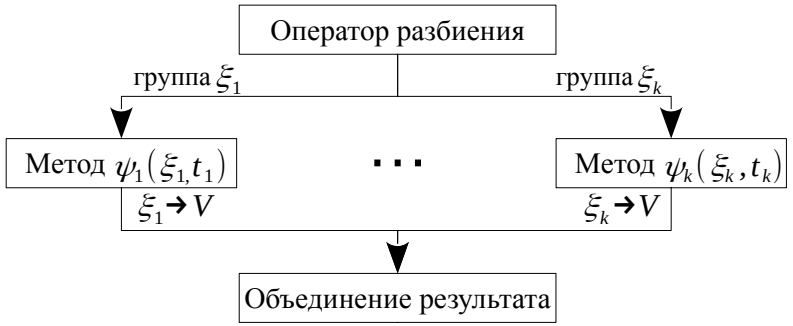

Рис. 1. Параллельная комбинация
Fig 1. Parallel combination

Параллельная комбинация во многом схожа с методами параллельной верификации, которые за счет параллельного решения различных задач достижимости на нескольких компьютерах или ядрах процессора позволяют сократить астрономическое время верификации. Одним из примеров параллельной верификации является проверка выполнения каждого требования на отдельном компьютере с помощью базового метода. Основное отличие параллельной комбинации от обычных методов параллельной верификации состоит в том, что главной целью является снижение требуемых ресурсов и улучшение результата, а не только сокращение астрономического времени верификации.

Для достижения поставленной цели оператор разбиения должен помещать в одну группу требования, совместная верификация которых более эффективна. Например, разные требования к одним и тем же программным интерфейсам

имеет смысл верифицировать вместе, а требование, которое ведет к чрезмерному росту числа состояний значительно чаще остальных, скорее, всего, следует верифицировать отдельно. Для выполнения подобного разбиения может использоваться информация о проверяемом коде.

Стоит заметить, что подобную технику можно реализовать внутри рассмотренного метода ДАС, поскольку он также автоматически разбивает спецификацию на группы требований. Однако комбинация методов имеет ряд преимуществ. Во-первых, решаемая задача достижимости не будет усложняться за счет требований, которые не нужно проверять. Например, в методе УМАВ в код не будут добавлены проверки подобных требований, что сократит накладные расходы на верификацию. Во-вторых, чрезмерный рост числа состояний, вызванный хотя бы одним из требований, потенциально может привести к исчерпанию всей доступной оперативной памяти и невозможности получить результат для остальных требований. Если же данное требование будет верифицироваться отдельно, то результат не будет получен только для данного требования. В-третьих, комбинация методов более универсальна, поскольку может использовать не только метод ДАС, а потенциально те же идеи могут быть применимы и не только для верификации композиции требований. Поэтому параллельная комбинация может как предоставить более точный результат, так и затратить на его получение меньше ресурсов.

Однако параллельная комбинация имеет достаточно ограниченные возможности, поскольку методы верификации не взаимодействуют между собой. Например, один метод не может справиться с решение задачи и тратит все выделенные ресурсы впустую, в то время как она относительно просто решается с помощью другого. Проблема заключается в том, что группы требований получаются статически и не могут меняться во время верификации, что может негативно сказываться на результате. Кроме того, чрезмерное увеличение верифицируемых групп требований будет вести к увеличению числа однотипных действий, что приведет к снижению производительности. Следствием данных проблем является отсутствие уверенности в том, что в общем случае параллельная комбинация методов будет более эффективной или более точной, чем применение каждого из методов в отдельности.

Таким образом, параллельная комбинация простейшим способом объединяет методы верификации с целью повышения производительности и улучшения результата, однако в общем случае не предоставляет гарантий достижения поставленной цели.

3.2. Последовательная комбинация

Последовательная (или *динамическая*) комбинация предполагает итеративное применение методов верификации в некоторой последовательности, при этом каждый последующий метод будет верифицировать требования из числа тех, с

которыми не справились предыдущие методы. Схема последовательной комбинации приведена на рис. 2.

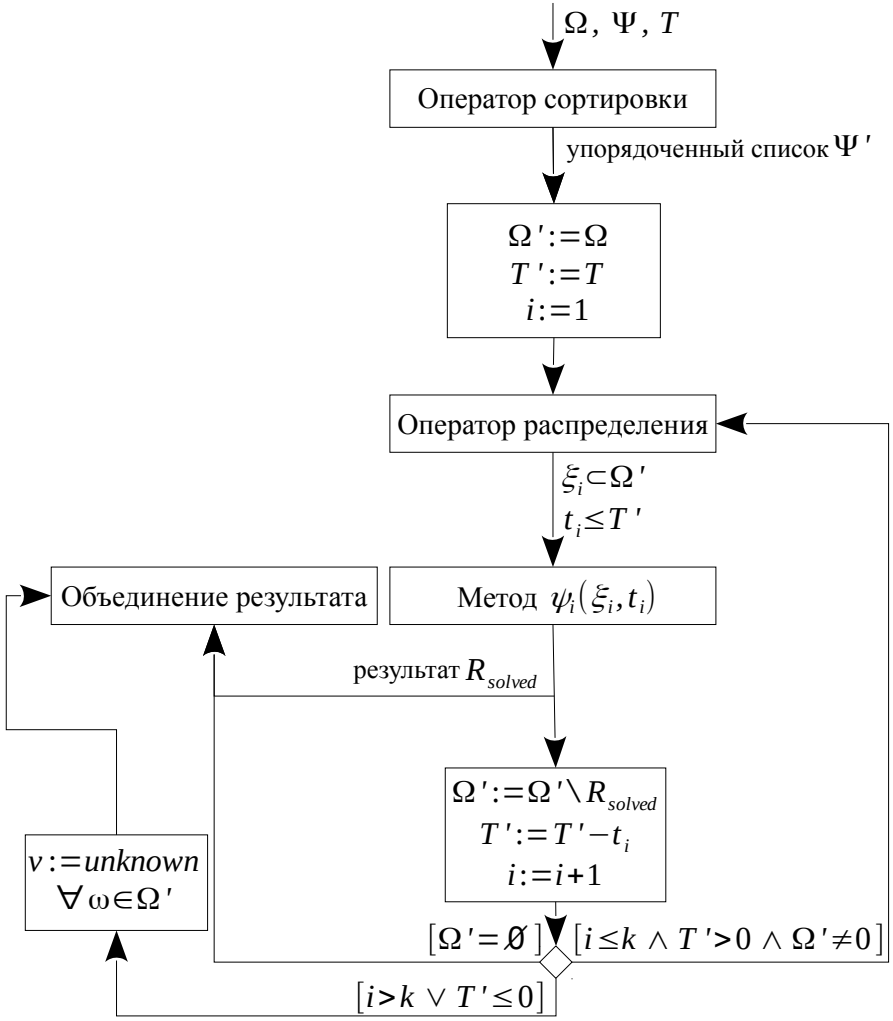

Рис. 2. Последовательная комбинация
Fig 2. Sequential combination

Последовательная комбинация вначале упорядочивает все методы с помощью оператора *сортировки*, а затем по очереди их применяет. Для текущего используемого метода $\psi_i(\xi_i, t_i)$ оператор *распределения* выделяет группу требований $\xi_i \subseteq \Omega'$ и вычислительные ресурсы $t_i \le T'$. Результат работы

каждого метода разбивается на два множества: успешно верифицированные требования $\left(R_{solved} = \{\xi_i \to v \mid v \in \{safe, unsafe\}\}\right)$ и остальные. Результат R_{solved} запоминается, и соответствующие требования больше не проверяются (то есть удаляются из множества проверяемых требований Ω'). Данная процедура повторяется до тех пор, пока множество проверяемых требований не пусто $(\Omega' \neq \varnothing)$, список методов верификации не исчерпан $(i \le k)$ и ресурсы не исчерпаны $(T' > 0)$. Если же после завершения работы комбинации методов не все требования получили вердикт (то есть $\Omega' \neq \varnothing$), то они искусственно получают вердикт *unknown*.

Основное преимущество последовательной комбинации перед параллельной состоит в том, что имеется возможность учитывать результат работы предыдущих методов для формирования групп требований и выделения ресурсов для их верификации (с помощью оператора распределения). Кроме того, можно отметить, что последовательная комбинация является более общим случаем параллельной комбинации. Действительно, если определить оператор распределения таким образом, что он для каждого метода выдает заранее определенный набор групп, среди которых нет пересечений, и равномерным образом распределяет ресурсы, то он будет работать аналогично оператору разбиения параллельной комбинации.

Последовательная комбинация базируется на идее условной проверки моделей [8]: имеется несколько методов верификации, которые последовательно применяются для решения поставленной задачи, если она не была решена одним из методов, то выдается условие, описывающее причину неудачного завершения (например, какое именно требование ведет к исчерпанию выделенной оперативной памяти), которое поможет разбить требования на группы и правильно распределить ресурсы для оставшихся методов. Однако в отличие от условной проверки моделей последовательная комбинация решает разные задачи достижимости, в которых проверяются различные требования. Помимо этого, последовательная комбинация предполагает, что в общем случае методы могут быть реализованы разными инструментами, то есть для каждого метода будет производиться отдельный запуск инструмента верификации. Поэтому последовательную комбинацию можно рассматривать как обобщение метода условной проверки моделей, поскольку она также может быть использована не только для верификации композиции требований.

Последовательная комбинация предоставляет универсальный способ для улучшения выбранной характеристики. Например, если требуется повысить производительность, то имеет смысл сначала использовать более быстрые методы с небольшими ограничениями на ресурсы, что позволит успешно решить большую часть задач. Если же требуется максимально снизить число

потерь, то стоит перепроверять известные ситуации, ведущие к потерям результата в используемых методах. В данном случае преимуществом последовательной комбинации является универсальность, поскольку требуемая конфигурация для достижения поставленной цели будет автоматически подобрана во время верификации для каждой из решаемых задач.

С другой стороны, подобная универсальность может приводить к тому, что в частных случаях последовательная комбинация будет уступать применению отдельных методов верификации. Например, задача может быть успешно решена только методом, который помещен в конец списка. В данном случае результат все равно будет получен, однако будут затрачены ресурсы на попытку решить задачу предыдущими методами.

Таким образом, последовательная комбинация методов предназначена для использования преимуществ нескольких методов и минимизацию их недостатков с целью повышения производительности верификации и улучшения результата.

4. Варианты реализации комбинации методов

Различные варианты комбинации методов верификации композиции требований были реализованы в рамках открытой системы верификации Linux Driver Verification Tools [15] (LDV Tools), которая предназначена для верификации модулей ядра операционной системы Linux относительно требований на корректное использование программных интерфейсов сердцевины ядра. Система LDV Tools готовит задачи достижимости на основе модулей ядра, которые решаются с помощью статического верификатора CPAchecker [14]. Данный верификатор реализует все описанные в разделе 2 методы верификации композиции требований.

Помимо предложенных возможны и другие варианты реализации комбинации методов в других системах верификации.

4.1. Регрессионная условная многоаспектная верификация

Каждый из рассмотренных методов верификации композиции требований может терять эффективность и получаемый результат с ростом числа требований: в методе УМАВ происходит усложнение кода из-за инструментирования, а в методе ДАС имеются накладные расходы на использование нескольких автоматов. В методе УМАВ данная проблема проявляется наиболее заметно, что приводит к относительно большим потерям результата, среди которых встречаются и реальные ошибки. Одним из возможных решений этой проблемы является разбиение композиции требований на группы в рамках параллельной комбинации с использованием одного метода УМАВ.

Для выполнения разбиения вводится функция сложности $f(\omega)$, которая задает некоторый критерий сложности проверки каждого из требований. Например, функция сложности может возвращать среднее число нерешенных задач для каждого требования. Будем считать, что все требования в Ψ упорядочены в соответствии с выбранной функцией сложности. Для оператора разбиения дополнительно задается ожидаемое число групп в разбиении $2 \le m \le \lfloor \log_2 n \rfloor$. Операция разбиения выполняется по следующей схеме. Вначале для последовательности значений функции сложности $f(\omega_1), ..., f(\omega_n)$ вычисляются среднее арифметическое и медиана. Максимальное из полученных двух значений *M* разделяет требования на две группы – «простую», для требований из которой значение функции сложности не превосходит *M,* и «сложную», содержащую остальные требования. Данная процедура разбиения повторяется для требований из «сложной» группы, если число требований в ней больше 1, или для требований из «простой» группы в противном случае. После выполнения данной процедуры *(m – 1)* раз будет получено искомое разбиение на *m* групп.

Основная идея данного разбиения состоит в том, чтобы верифицировать отдельно наиболее сложные требования в небольших группах или отдельно, если их сложность значительно выше остальных, и создать одну группу, сложность требований из которой существенно ниже. Для получения функции сложности может использоваться информация о верификации данных требований в предыдущих версиях программного обеспечения (по аналогии с регрессионной верификацией [18]).

4.2. Последовательная комбинация «точность»

Для получения более точного результата была реализована следующая версия последовательной комбинации. На первом шаге все требования верифицируются с помощью метода УМАВ с ограничением по времени t_1, что позволит более эффективно решить большую часть задач за счет эвристик данного метода. На втором шаге все требования, получившие вердикт *unknown* на предыдущем шаге из-за эвристик, проверяются отдельно с помощью метода АС с ограничением по времени *t* на проверку каждого требования. В данном случае предполагается, что подобные требования наиболее сложные для верификации совместно с другими, поэтому для получения более точного результата их следует проверять отдельно. На третьем шаге все оставшиеся требования проверяются более ресурсоемким и более точным методом ДАС. Число оставшихся требований на этом шаге соизмеримо с числом всех требований, при этом более оптимизированным методом УМАВ задачу уже решить не удалось, поэтому и применяется метод ДАС.

Данная реализация объединяет преимущества использования методов верификации композиции требований УМАВ и ДАС и с помощью метода АС устраняет недостатки для предотвращения потерь.

4.3. Последовательная комбинация «производительность»

Для повышения производительности был предложен упрощенный вариант предыдущей реализации. Вначале задача решается методом УМАВ, если же она не была решена, то применяется метод ДАС.

5. Эксперименты

Для сопоставления предложенных комбинаций с существующими методами сравнивалось процессорное время (как решения задач достижимости, так и всего процесса верификации) и потери результата (то есть задачи, не решенные с теми же ограничениями на ресурсы, что и в базовом методе). Эксперименты проводились с помощью системы верификации LDV Tools [15] (ветка *composition_of_reachability_tasks*), задачи достижимости решались с помощью статического верификатора CPAchecker [14]. Проверялись все модули ядра операционной системы Linux версии *4.0-rc1* в конфигурации *allmodconfig*, для которых система LDV Tools успешно готовит задачи достижимости (то есть 4 041 модуль), относительно 30 имеющихся в системе LDV Tools требований. В качестве решения данной задачи от системы верификации ожидались 121 230 вердиктов. В совокупности проверяемые модули содержат около 9 млн. строк кода. Для метода УМАВ использовалась версия 23 131 инструмента CPAchecker ветки *cmav*, для метода ДАС – версия 20 125 инструмента CPAchecker ветки *muauto*. В обоих случаях CPAchecker использовался с конфигурацией, которая включает предикатную абстракцию [19] и анализ явных значений [20]. Для экспериментов использовались компьютеры (узлы кластера) со следующими характеристиками: процессор Intel Xeon E312xx (Sandy Bridge) 2.6 GHz (8 ядер), 64 GB оперативной памяти, операционная система Ubuntu 14.04 (64-bit) с ядром Linux 3.13, Java версии 1.7.0_101 (вычислительный кластер ИСП РАН).

Базовое ограничение на используемые ресурсы было выбрано в соответствии с международными мероприятиями по верификации Competition on Software Verification [12]: 15 минут процессорного времени и 15 GB оперативной памяти на проверку одного требования в одной задаче достижимости. Дополнительно использовалось ограничение по памяти на размер кучи для виртуальной машины Java в 13 GB (13/15 от базового ограничения на оперативную память), что необходимо для корректной работы верификатора CPAchecker. При этом на проверку нескольких требований в одной задаче достижимости ограничение на процессорное время увеличивалось пропорционально числу требований (то есть проверка 30 требований ограничивалась 27 000 процессорных секунд). Ограничение на оперативную память не изменялось, чтобы подготавливаемые

задачи могли решаться на тех же компьютерах. При этом на каждом узле одновременно могло решаться четыре задачи достижимости (по одному ядру процессора и 15 GB оперативной памяти на каждую), оставшиеся ресурсы (4 ядра процессора и 4 GB оперативной памяти) использовались системой LDV Tools для подготовки задач достижимости, запуска комбинации методов и обработки результата.

5.1. Оценка вариантов параллельной комбинации

В данном эксперименте функция сложности возвращала среднее значение релевантных требованию модулей, полученное при верификации предыдущих версий ядра Linux (то есть число модулей, в которых верификация завершалась успешно и требование потенциально могло быть нарушено). Значения данной функции сложности после сортировки образовали следующую последовательность: 1, 2, 4, 11, 11, 17, 18, 25, 30, 36, 55, 60, 63, 68, 83, 91, 111, 113, 118, 131, 134, 155, 178, 214, 302, 356, 783, 970, 1048, 1048. Разбиение производилось на 2 и на 3 группы. Нетрудно заметить, что при применении алгоритма, описанного в разделе 4.1, данные требования в первом случае разбиваются на 23 и 7, а во втором – на 23, 4 и 3. Помимо этого, было рассмотрено случайное разбиение требований на две равные группы (то есть по 15 требований в каждой группе). Результаты эксперимента представлены в табл. 2. Процессорное время и общее ускорении в таблице представлено отдельно для решения задач достижимости верификатором CPAchecker (столбец «Решение задач») и для всего процесса верификации системой LDV Tools (столбец «Всего»), поскольку нужно учитывать, что при верификации *m* групп требований готовится и решается *m* задач достижимости для каждого модуля.

Параллельной комбинации удалось ускорить решение задач достижимости в сравнении с методом УМАВ и снизить число потерь результата. Параллельная комбинация с разбиением требований на 2 группы согласно предложенному алгоритму требует примерно на 20% меньше ресурсов на решение задач достижимости и примерно на 10% меньше ресурсов на весь процесс верификации в сравнении с методом УМАВ, при этом число потерь снижается до 1%, а количество потерянных реальных ошибок сокращается с 9 до 2. При увеличении числа групп наблюдается возрастание как времени подготовки задач достижимости примерно на 40% (с 317 000 секунд до 443 000) за счет того, что ставится большее число задач, так и времени решения задач достижимости (что, в частности, выражается снижением среднего ускорения решения задач с 5.83 до 4.68), поскольку выполняется большее число однотипных действий. В результате разбиение на 3 группы (и больше) уже не приводит к снижению суммарного времени верификации относительно метода УМАВ, хотя способствует незначительному сокращению процента потерь. Случайное разбиение требований на две группы в общем случае демонстрирует более низкие показатели, так как не учитываются особенности решаемых задач.

Табл. 2. Оценка параллельной комбинации
Table 2. Evaluation of parallel combination

Метод	Результат			Процессорное время (с) / общее ускорение	
	Safe	*Unsafe*	Потери	Решение задач	Всего
Базовый	118 703	667	0 (0.00%)	3 889 000 1.00	6 742 000 1.00
УМАВ(30)	117 162	634	1 648 (1.36%)	1 289 000 3.02	1 514 000 4.45
УМАВ(23) || УМАВ(7)	117 556	654	1 201 (0.99%)	1 080 000 3.60	1 397 000 4.83
УМАВ(23) || УМАВ(4) || УМАВ(3)	117 603	658	1 147 (0.95%)	1 141 000 3.41	1 584 000 4.26
УМАВ(15) || УМАВ(15) (случайное разбиение)	117 310	641	1 465 (1.21%)	1 176 000 3.31	1 526 000 4.42

Таким образом, параллельная комбинация демонстрирует как повышение производительности, так и улучшение результата при разбиении на небольшое число групп. Однако для этого требуется использование вспомогательных данных о решаемых задачах, с помощью которых выполняется разбиение требований на группы.

5.2. Оценка вариантов последовательной комбинации

В последовательной комбинации метод УМАВ в обоих случаях был ограничен 1 200 секундами процессорного времени, для каждого запуска метода АС выделялось 900 секунд (базовое ограничение). Результаты эксперимента представлены в табл. 3.

Последовательная комбинация «точность» позволила максимально сократить количество потерь до 0.26%, что сопоставимо со статистической погрешностью. Помимо этого, не было потеряно ни одной реальной ошибки в отличие от остальных методов (метод УМАВ теряет 9 реальных ошибок, а метод ДАС – 2), что крайне важно на практике. При этом требуется разумное количество ресурсов для верификации – меньше метода ДАС, но больше метода УМАВ. Последовательная комбинация «производительность» сократила ресурсы как на решение задач достижимости, так и на весь процесс верификации по сравнению с методами УМАВ и ДАС, при этом процент потерь оказался незначительно больше метода ДАС.

Таким образом, последовательные комбинации методов позволяют улучшать характеристики методов верификации без дополнительной информации о решаемых задачах.

Табл. 3. Оценка последовательной комбинации
Table 3. Evaluation of sequential combination.

Метод	Результат			Процессорное время (с) / общее ускорение	
	Safe	*Unsafe*	Потери	Решение задач	Всего
Базовый	118 703	667	0 (0.00%)	3 889 000 1.00	6 742 000 1.00
АС	118 929	680	182 (0.15%)	3 434 000 1.13	6 107 000 1.10
УМАВ	117 162	634	1 648 (1.36%)	1 289 000 3.02	1 514 000 4.45
ДАС	118 386	673	548 (0.45%)	1 373 000 2.83	1 550 000 4.35
УМАВ → АС → ДАС («точность»)	118 679	695	311 (0.26%)	1 367 000 2.84	1 592 000 4.23
УМАВ → ДАС («производительность»)	118 320	637	630 (0.52%)	1 196 000 3.25	1 426 000 4.73

6. Заключение

В данной статье были предложены различные варианты комбинации методов статической верификации для повышения производительности и улучшения результата с использованием существующих методов верификации. Реализация предложенных методов для верификации композиции требований продемонстрировала улучшение характеристик при проверке модулей ядра операционной системы Linux.

Параллельная комбинация методов рекомендуется для использования совместно с регрессионной верификацией, что позволит получать требуемую информацию о верификации для разбиения требований на группы. Помимо этого, параллельная комбинация для верификации одной программы может использовать несколько компьютеров (ядер процессора), что позволит сократить и астрономическое время.

Последовательная комбинация не требует дополнительной информации о решаемых задачах и является более универсальным методом для проверки

произвольного программного обеспечения. Различные ее вариации могут улучшать главным образом ту или иную характеристику верификации.

В будущем планируется использовать комбинацию методов для большего числа методов верификации с целью решения произвольных задач достижимости.

Список литературы

[1]. Turing A. M. On Computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, pp. 230-265, 1936.

[2]. Corbet J., Kroah-Hartman G., McPherson A. Linux kernel development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. https://www.linux.com/publications/linux-kernel-development-how-fast-it-going-who-doing-it-what-they-are-doing-and-who-5, дата обращения 10.06.2017.

[3]. Мутилин В. С., Новиков Е. М., Хорошилов А. В. Анализ типовых ошибок в драйверах операционной системы Linux. Труды ИСП РАН, т. 22, с. 349-374, 2012. DOI: 10.15514/ISPRAS-2012-22-19.

[4]. Mordan V., Novikov E. Minimizing the number of static verifier traces to reduce time for finding bugs in Linux kernel modules. Proceedings of 8th Spring/Summer Young Researchers Colloquium on Software Engineering, vol. 1, 2014. DOI: 10.15514/syrcose-2014-8-5.

[5]. Mordan V., Mutilin V. Checking several requirements at once by CEGAR. LNCS, vol. 9609, pp. 218-232, 2016. DOI: 10.1007/978-3-319-41579-6_17.

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

[7]. Apel S., Beyer D., Mordan V., Mutilin V., Stahlbauer A. On-The-Fly Decomposition of Specifications in Software Model Checking. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 349-361, 2016. DOI: 10.1145/2950290.2950349.

[8]. Beyer D., Henzinger T. A., Keremoglu M. E., Wendler P. Conditional model checking: a technique to pass information between verifiers. Proceedings ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE 2012), article 57, 2012. DOI: 10.1145/2393596.2393664.

[9]. Новиков Е. М. Развитие метода контрактных спецификаций для верификации модулей ядра операционной системы Linux. Диссертация на соискание ученой степени кандидата физико-математических наук. ИСП РАН, 2013.

[10]. Beyer D., Chlipala A., Henzinger T. A., Jhala R., Majumdar R. The BLAST query language for software verification. Proceedings of SAS. LNCS, vol. 3148, pp. 2-18, 2004. DOI: 10.1007/978-3-540-27864-1_2.

[11]. Clarke E., Grumberg O., Jha S., Lu Y., Veith H. Counterexample-guided abstraction refinement. Proceedings of CAV, LNCS, vol. 1855, pp. 154-169, 2000. DOI: 10.1007/10722167_15.

[12]. Beyer D. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). Proceedings of TACAS. LNCS, vol. 9636, pp. 887-904, 2016. DOI: 10.1007/978-3-662-49674-9_55.

[13]. Beyer D., Henzinger T., Jhala R., Majumdar R. The software model checker BLAST. International Journal on Software Tools for Technology Transfer, vol. 9, issue 5, pp. 505-525, 2007. DOI: 10.1007/s10009-007-0044-z.

[14]. Beyer D., Keremoglu M. CPAchecker: A tool for configurable software verification. Proceedings of Computer Aided Verification. LNCS, vol. 6806, pp. 184–190, 2011. DOI: 10.1007/978-3-642-22110-1_16.

[15]. Khoroshilov A., Mutilin V., Novikov E., Shved P., Strakh A. Towards an open framework for C verification tools benchmarking. Perspectives of Systems Informatics. LNCS, vol. 7162, pp. 179-192, 2012. DOI: 10.1007/978-3-642-29709-0_17.

[16]. Ball T., Rajamani S. K. The SLAM project. Debugging system software via static analysis. Proceedings of Symposium on Principles of Programming Languages (POPL), pp. 1–3, 2002. DOI: 10.1145/565816.503274.

[17]. Мордань В. О. Методы верификации программ на основе композиции задач достижимости. Диссертация на соискание ученой степени кандидата физико-математических наук. ИСП РАН, 2017.

[18]. Beyer D., Wendler P. Reuse of verification results. Proceedings of the 20th International Workshop on Model Checking Software (SPIN 2013). LNCS, vol. 7976, pp. 1-17, 2013. DOI: 10.1007/978-3-642-39176-7_1.

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

[20]. Beyer D., Löwe S. Explicit-state software model checking based on CEGAR and interpolation. Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013). LNCS, vol. 7793, pp. 146-162, 2013. DOI: 10.1007/978-3-642-37057-1_11.

Combination of static verification methods for checking requirements composition

V.O. Mordan <mordan@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. Static verification proves correctness of the software against checked requirements, but it requires a lot of resources for that and its task is undecidable in general case. At present there is no universal static verification method, which could efficiently check any software. That is why one should choose more appropriate method and set its parameters for checking correctness of the given requirements in a given program. This paper suggests to combine different static verification methods in order to increase efficiency and effectiveness of verification, which is the first step in creating universal method for static verification. The suggested methods were implemented as combination of actively developing static verification methods for checking requirements composition. Implementation of the suggested methods showed their advantages on Linux kernel modules in comparison with using of each verification method separately.

Keywords: software model checking; counterexample guided abstraction refinement; reachability task; requirements composition.

DOI: 10.15514/ISPRAS-2017-29(3)-9

For citation: Mordan V.O. Combination of static verification methods for checking requirements composition. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 3, pp. 151-170 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-9

References

[1]. Turing A. M. On Computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, 1936, pp. 230-265.
[2]. Corbet J., Kroah-Hartman G., McPherson A. Linux kernel development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. https://www.linux.com/publications/linux-kernel-development-how-fast-it-going-who-doing-it-what-they-are-doing-and-who-5, свободный, accessed 29.06.2017.
[3]. Mutilin V.S., Novikov E.M., Khoroshilov A.V. Analysis of typical faults in Linux operating system drivers. Trudy ISP RAN / Proc. ISP RAS, vol. 22, 2012, pp. 349–374 (in Russian). DOI: 10.15514/ISPRAS-2012-22-19.
[4]. Mordan V., Novikov E. Minimizing the number of static verifier traces to reduce time for finding bugs in Linux kernel modules. Proceedings of 8th Spring/Summer Young Researchers Colloquium on Software Engineering, vol. 1, 2014. DOI: 10.15514/syrcose-2014-8-5.
[5]. Mordan V., Mutilin V. Checking several requirements at once by CEGAR. LNCS, vol. 9609, pp. 218-232, 2016. DOI: 10.1007/978-3-319-41579-6_17.

[6]. Mordan V. O., Mutilin V. S. Checking several requirements at once by CEGAR. Programming and Computer Software, vol. 42, no. 4, pp. 225–238, 2016. DOI: 10.1134/S0361768816040058.
[7]. Apel S., Beyer D., Mordan V., Mutilin V., Stahlbauer A. On-The-Fly Decomposition of Specifications in Software Model Checking. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 349-361, 2016. DOI: 10.1145/2950290.2950349.
[8]. Beyer D., Henzinger T. A., Keremoglu M. E., Wendler P. Conditional model checking: a technique to pass information between verifiers. Proceedings ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE 2012), article 57, 2012. DOI: 10.1145/2393596.2393664.
[9]. Novikov E. M. Development of a contract specification method for verification of Linux kernel modules. PhD thesis. ISP RAS, 2013 (in Russian).
[10]. Beyer D., Chlipala A., Henzinger T. A., Jhala R., Majumdar R. The BLAST query language for software verification. Proceedings of SAS. LNCS, vol. 3148, pp. 2-18, 2004. DOI: 10.1007/978-3-540-27864-1_2.
[11]. Clarke E., Grumberg O., Jha S., Lu Y., Veith H. Counterexample-guided abstraction refinement. Proceedings of CAV, LNCS, vol. 1855, pp. 154-169, 2000. DOI: 10.1007/10722167_15.
[12]. Beyer D. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). Proceedings of TACAS. LNCS, vol. 9636, pp. 887-904, 2016. DOI: 10.1007/978-3-662-49674-9_55.
[13]. Beyer D., Henzinger T., Jhala R., Majumdar R. The software model checker BLAST. International Journal on Software Tools for Technology Transfer, vol. 9, issue 5, pp. 505-525, 2007. DOI: 10.1007/s10009-007-0044-z.
[14]. Beyer D., Keremoglu M. CPAchecker: A tool for configurable software verification. Proceedings of Computer Aided Verification. LNCS, vol. 6806, pp. 184–190, 2011. DOI: 10.1007/978-3-642-22110-1_16.
[15]. Khoroshilov A., Mutilin V., Novikov E., Shved P., Strakh A. Towards an open framework for C verification tools benchmarking. Perspectives of Systems Informatics. LNCS, vol. 7162, pp. 179-192, 2012. DOI: 10.1007/978-3-642-29709-0_17.
[16]. Ball T., Rajamani S. K. The SLAM project. Debugging system software via static analysis. Proceedings of Symposium on Principles of Programming Languages (POPL), pp. 1–3, 2002. DOI: 10.1145/565816.503274.
[17]. Mordan V. O. Methods of software verification based on composition of reachability tasks. PhD thesis. ISP RAS, 2017 (in Russian).
[18]. Beyer D., Wendler P. Reuse of verification results. Proceedings of the 20th International Workshop on Model Checking Software (SPIN 2013). LNCS, vol. 7976, pp. 1-17, 2013. DOI: 10.1007/978-3-642-39176-7_1.
[19]. Beyer D., Keremoglu M., Wendler P. Predicate abstraction with adjustable-block encoding. Proceedings of Formal Methods in Computer-Aided Design, pp. 189-198, 2010.
[20]. Beyer D., Löwe S. Explicit-state software model checking based on CEGAR and interpolation. Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013). LNCS, vol. 7793, pp. 146-162, 2013. DOI: 10.1007/978-3-642-37057-1_11.