Обзор инструментов статической верификации Си программ в применении к драйверам устройств операционной системы Linux[1]

М.У. Мандрыкин, В.С. Мутилин, Е.М. Новиков, А.В. Хорошилов,
mandrykin@ispras.ru, mutilin@ispras.ru, joker@ispras.ru, khoroshilov@ispras.ru

Аннотация. В работе рассмотрены методы и техники, используемые в современных инструментах статической верификации Си программ. Представлен обзор текущих возможностей инструментов, в том числе по таким направлениям, как поддержка конструкций языка Си, масштабируемость, виды проверяемых свойств и достоверность результатов анализа. Рассмотрены особенности применения инструментов статической верификации для анализа исходного кода драйверов устройств операционных систем и описаны существующие системы верификации драйверов, построенные на основе таких инструментов.

Ключевые слова: верификация, статический анализ, драйвер устройства, операционная система Linux.

1. Введение

Статическая верификация программ заключается в анализе кода программы без её реального выполнения, что в отличие от динамического анализа не требует настройки тестового окружения и предоставляет возможность проанализировать все возможные выполнения программы, даже те, для воспроизведения которых требуется одновременное выполнение сразу нескольких редко встречающихся условий.

В настоящее время наибольшее распространение получили две группы методов статической верификации: методы дедуктивного анализа программ и методы проверки моделей. Методы дедуктивного анализа используются для доказательства соответствия программы своей спецификации, обычно задаваемой в виде пред- и постусловий. На текущем уровне развития инструменты, реализующие эти методы, требуют ручной аннотации

[1] Работа поддержана ФЦП "Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы" (контракт N 07.514.11.4104)

анализируемых функций и циклов в программе, что не позволяет их применять для верификации больших программ. Тем не менее, эти методы являются эффективным средством для доказательства корректности наиболее важных компонентов программных систем.

Методы проверки моделей формируют по коду программы её модель в явном или неявном виде и проводят анализ этой модели на предмет выполнения заданных ограничений. Как правило, проверяемые ограничения не нацелены на полное описание требований к функциональности программы, а представляют собой проверку выполнения частных требований или проверку отсутствия в коде программы определённого класса ошибок.

Методы проверки моделей обычно работают с исходным текстом программы на языке программирования. Перед анализом происходит разбор текста программы во внутреннее представление, в ходе которого для каждой функции в программе строится граф потока управления (ГПУ). Вершины графа соответствуют местам в тексте программы. Рёбра содержат операторы из текста программы, возможно, преобразованные в последовательность инструкций. Инструкции обращаются к локальным и глобальным переменным программы, к значениям в динамической памяти или к окружению системы.

Провести детальный анализ всех возможных выполнений программы за разумное время не представляется возможным, поэтому при статической верификации выполнения программы моделируются лишь с некоторой точностью. От уровня точности зависит количество ложных сообщений об ошибках, количество упущенных ошибок, а также время, необходимое для выполнения анализа.

В зависимости от того, каким характеристикам отдается предпочтение выделяют легковесные и тяжеловесные подходы. В легковесных подходах упор делается на скорости анализа, при этом точность анализа может снижаться, что приводит к увеличению числа упущенных ошибок и ложных срабатываний. Как правило, высокая скорость достигается за счёт активного использования различных эвристик и игнорирования значительной части контекста точки выполнения программы, то есть пути попадания в эту точку. В тяжеловесных подходах предпочтение отдается точности, что при определённых условиях дает возможность доказать даже полное отсутствие искомых видов ошибок в программе. Для тяжеловесных подходов характерен аккуратный анализ контекста точки выполнения программы, а также более детальный анализ взаимодействия программы с окружением.

В настоящей работе рассмотрение ограничивается тяжеловесными подходами, которые демонстрируют значительный прогресс за последние годы. Одним из первых примеров успешного применения таких подходов в индустрии является использование инструмента Slam [1] для верификации драйверов устройств операционной системы (ОС) Microsoft Windows. Драйверы устройств являются подходящей площадкой для применения тяжеловесных инструментов статической верификации по целому ряду причин:

- массовость — драйверы разрабатываются в большом количестве по единым принципам, что позволяет проверять выполнимость одних и тех же свойств сразу у многих программ;
- размер кода — суммарный размер исходного кода драйверов, как правило, не превышает нескольких десятков тысяч строк кода;
- относительная простота кода — в драйверах обычно не используются операции с плавающей точкой, сложные структуры данных и т.п.

Поэтому в дальнейшем инструменты тяжеловесной верификации неоднократно пытались применить для верификации драйверов устройств, но уже ОС Linux, так как в отличии от разработчиков Slam из Microsoft Research, доступ других исследователей к исходному коду драйверов ОС Microsoft Windows ограничен. В рамках настоящего обзора представлено описание этих попыток, интересное в первую очередь с точки зрения применения тяжеловесных подходов на практике.

Основная часть работы построена следующим образом. В разделе 2 рассматриваются методы и техники, используемые в современных инструментах тяжеловесной статической верификации Си программ. Раздел 3 содержит описание характеристик инструментов верификации, важных для их практического применения. В разделе 4 представлен обзор инструментов тяжеловесной статической верификации Си программ. В разделе 5 обсуждаются особенности применения инструментов статической верификации для анализа исходного кода драйверов устройств операционных систем и описываются существующие системы верификации драйверов, построенные на основе таких инструментов. В заключение подводятся итоги работы и делаются выводы о перспективах развития инструментов тяжеловесной статической верификации в контексте их применения для верификации драйверов устройств.

2. Методы и техники статической верификации программ

2.1. CEGAR (Counter Example Guided Abstraction Refinement)

CEGAR – метод статического анализа кода, основанный на абстракции и уточнении на основе контрпримеров [2].

Свойства программы, которые проверяются при помощи CEGAR, формулируются в виде недостижимости заданного множества ошибочных состояний в программе. Идея CEGAR состоит в итеративном уточнении модели программы (абстракции) до тех пор, пока не будет доказана её корректность или не будет найден ошибочный путь, то есть путь, начинающийся в точке входа в программу и ведущий в ошибочное состояние. Для этого CEGAR предлагает строить такую модель программы, в которой сохраняются ошибочные пути, и осуществлять поиск ошибочных путей в этой

модели. Если в модели ошибочных путей не найдено, то в программе нет искомых ошибок. В противном случае, потребуется проверить реализуемость модельного ошибочного пути в программе, так как возможно, что ошибочный путь в модели появился из-за абстрагирования от существенных деталей в коде программы. Если проверка покажет, что путь реализуем, то это означает, что найдена ошибка в программе. Иначе этот путь в качестве контрпримера будет использован для уточнения модели на следующем шаге, так чтобы в уточнённой модели этого пути уже не было.

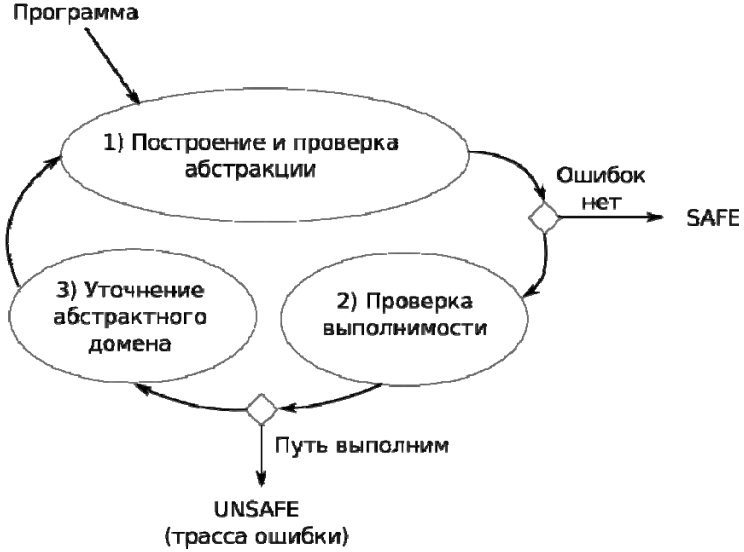

Рис. 1. Подход CEGAR (Counter Example Guided Abstraction Refinement)

В CEGAR выделяют следующие составляющие этапы:

1. Построение и проверка абстракции.
2. Проверка выполнимости.
3. Уточнение абстрактного домена.

На первом этапе строится абстракция программы, включающая в себя все ошибочные пути выполнения в программе. Абстракция строится по начальному абстрактному домену, задающему правила соответствия между абстрактными состояниями и переходами и состояниями и переходами в программе. Построение абстракции — это затратная операция, состоящая в вычислении абстрактных переходов между абстрактными состояниями. Точность абстракции зависит от абстрактного домена, определяющего точность приближения состояний программы абстрактными состояниями, а также от точности моделирования конструкций входной программы при вычислении абстрактных переходов.

Вторая часть первого этапа — проверка абстракции на наличие ошибки. Она может происходить как строго после построения абстракции, так и совместно. В первом случае построенная абстракция может передаваться в том числе и сторонним инструментам, например, классическим инструментам проверки моделей (model checker tools), принимающим программу на модельном входном языке.

Во втором случае проверка происходит по мере построения частей абстракции. Например, как только достигнуто абстрактное состояние, включающее ошибочное, построение абстракции приостанавливается, и начинается проверка выполнимости найденного ошибочного пути. Одной из распространённых структур данных в этих подходах является *абстрактное дерево достижимости*. Метод *ленивой абстракции* позволяет не перестраивать абстрактное дерево достижимости целиком при уточнении абстрактного домена, например, при нахождении новых предикатов для *предикатной абстракции*.

Метод адаптивного анализа (CPA) используется для того, чтобы при построении абстракции можно было комбинировать построение предикатной абстракции с анализом потоков данных, абстрактные элементы решётки которого хранятся в абстракции. Данный метод позволяет, например, одновременно с анализом на основе предикатной абстракции осуществлять *анализ с явными значениями*.

В результате первого этапа либо доказывается, что ошибок в абстрактной модели нет, либо находится ошибочный путь в абстракции, называемый контрпримером. Так как абстракция включает все ошибочные пути, то отсутствие ошибок в абстрактной модели говорит о том, что ошибки нет в исходной программе, инструмент выдает вердикт SAFE. Данный вердикт может оказаться неправильным, т. е. может быть упущена ошибка, например, если программа содержит не поддерживаемые конструкции языка, которые моделируются неточно, но инструмент не завершается аварийно при их наличии.

Если обнаруживается контрпример, то он передается на проверку его выполнимости в исходной программе. Для этого, как правило, строится формула пути, кодирующая выполнение программы по пути к ошибке. Эта формула затем передается решателю. Семантика выполнения кодируется в терминах теорий, поддерживаемых решателями. Таким образом, точность проверки выполнимости зависит как от точности кодирования конструкций языка, так и от возможностей решателя. Неточности проверки выполнимости ведут к ложным срабатываниям, когда инструмент выдает вердикт UNSAFE в случае отсутствия реальной ошибки.

Когда путь оказывается невыполнимым, это означает, что абстракция программы нуждается в уточнении. На этапе уточнения абстрактного домена, невыполнимый путь используется для конструирования нового домена, такого, чтобы можно было исключить данный путь к ошибке в уточнённой

абстракции, а вместе с ним, возможно, и другие невыполнимые пути. Например, в предикатной абстракции невыполнимый путь к ошибке используется для поиска новых предикатов как с помощью синтаксических методов, основанных на эвристиках, так и с помощью методов *интерполяции*.

2.2. Предикатная абстракция (Predicate Abstraction)

В качестве модели программы может выступать предикатная абстракция [3,4], состояниями которой являются наборы предикатов, описывающих множества конкретных состояний программы. Как правило, строится абстракция существования (existential abstraction), в которой при существовании перехода между двумя конкретными состояниями, всегда существует переход между любыми двумя абстрактными состояниями, включающими данные конкретные состояния. Абстракция существования гарантирует сохранение в абстрактной модели достижимых путей в ошибочное состояние. Таким образом, если абстрактная модель не содержит ошибку, то в программе её тоже нет. Предикатная абстракция может строиться явно или непосредственно при поиске ошибки, например, при построении дерева достижимости.

Точность приближения состояний программы зависит от используемых предикатов. Предикаты состояния могут быть выражены в логике высказываний (пропозициональной логике), с высказываниями относительно пропозициональных переменных, логике первого порядка, расширяющей логику высказываний фиксированными функциональными и предикатными символами, кванторами всеобщности и существования или логике более высоких порядков. Функциональные символы и предикаты могут иметь дополнительную интерпретацию в виде теорий. Примерами теорий являются теории целых и вещественных чисел, теории списков, массивов, битовых векторов и т. д. Использование тех или иных предикатов в состояниях накладывает ограничения на возможности решателей. Чем сложнее предикаты, тем больше возможностей требуется от решателей.

Вычисление переходов между состояниями требует представления конструкций языка программирования в виде логических формул, и здесь, также как и для состояний, могут использоваться различные логики и теории. Кроме того, может использоваться дополнительная аксиоматизация, например, при моделировании памяти в виде аксиом для функций получения размещения, взятия адреса и разыменования, которое было предложено в работе [5]. При вычислении переходов могут использоваться также дополнительные данные, полученные другими видами анализа, например могут использоваться данные анализа указателей для подстановки потенциально равных значений указателей в конструируемую формулу.

В случае использования подхода CEGAR предикаты в состояние добавляются на этапе уточнения предикатов посредством анализа формулы пути контрпримера, например, с помощью интерполирующих решателей или синтаксических методов. Извлекаемые на данном этапе предикаты

существенно влияют на возможность получения более точной абстракции состояний программы.

Наиболее распространенными вариантами предикатных абстракций являются декартова предикатная абстракция и булева предикатная абстракция. В декартовой абстракции предикаты могут быть соединены лишь конъюнкцией. Абстракцией является наиболее строгая конъюнкция предикатов, покрывающая состояния программы.

Для данной абстракции вычисление состояния в окончании перехода сводится к проверке истинности каждого предиката в отдельности. Результирующее состояние является конъюнкцией из истинных предикатов. Таким образом, количество вызовов решателя при вычислении декартовой предикатной абстракции равно количеству предикатов в состоянии.

В булевой предикатной абстракции допускаются произвольные комбинации предикатов в логике высказываний. Абстракцией является наиболее строгая формула, покрывающая состояния программы. Вычисление такой абстракции требует получения набора векторов истинных предикатов, которых может быть экспоненциально много. Поэтому вычисление булевой абстракции требует большей эффективности от решателей.

2.3. Абстрактное дерево достижимости (Abstract Reachability Tree)

Абстрактное дерево достижимости (ART) [6], содержит пути, достижимые из начального состояния. Вершинами дерева являются описания состояний программы. Например, содержащие вершину ГПУ и состояние предикатной абстракции. Ребра дерева достижимости помечены переходами, состоящими из одной или нескольких инструкций ГПУ. В случае, если используется кодирование последовательностей инструкций в ГПУ в один переход, так называемое представление программы в виде больших блоков (large block encoding) [7], одному ребру в ART может соответствовать последовательность инструкций, включающая в себя инструкции ветвления, точки слияния и вызовы функций.

Построение дерева состоит в вычислении конечного состояния по состоянию в текущей вершине и переходу, которым помечено ребро. Процесс вычисления зависит от используемого состояния. Например, в случае предикатной абстракции требуется вычисление следующего состояния по начальному абстрактному состоянию и набору инструкций, как правило, с использованием решателей.

Большие блоки увеличивают нагрузку на решатели, так как запросы к решателю при построении абстракции становятся сложнее. Например, если переход включает точки слияния, то конструируемая формула включает в себя дизъюнкцию для путей входящих в эту точку, что усложняет решение формулы. С другой стороны, становится меньше количество запросов к решателям. В связи с тем, что решатели постоянно совершенствуются и

способны решать все более сложные формулы за приемлемое время, уменьшение запросов в ряде случаев может ускорить анализ.

При применении адаптивного статического анализа каждый алгоритм CPA хранит часть своего состояния дерева и вычисляет следующее состояние для передаваемого перехода. Состояние алгоритма CPA может включать в себя состояние других CPA, вложенных в него. В этом случае при выполнении перехода этот CPA вызывает вложенные CPA для изменения своих элементов состояния. Неосуществимость перехода в одном из алгоритмов влечет неосуществимость перехода в целом.

Построение дерева достижимости происходит одновременно с проверкой заданного свойства программы. В случае проверки свойства достижимости, построение дерева прекращается при обнаружении ошибочного состояния, которое может задаваться как инструкция, помеченная специальной меткой. При использовании адаптивного анализа ошибочное состояние может определяться алгоритмом CPA.

Конечность дерева достижимости обеспечивается конечностью множества абстрактных состояний и тем, что пути из абстрактных состояний, покрываемых другими абстрактными состояниями, не перебираются. Покрытие состояний означает вложенность соответствующих множеств состояний программы. Для предикатной абстракции покрытие проверяется как импликация.

Дерево достижимости может строиться без вычисления следующего абстрактного состояния [8]. В этом случае вначале предполагается, что состояние может быть любым. На этапе уточнения предикатов, вместо выявления новых предикатов для уточнения всей абстракции, выделяются предикаты, которые уточняют состояния дерева, входящие в путь контрпримера. Таким образом, абстракция уточняется только для путей, ведущих к ошибке.

2.4. Ленивая абстракция (Lazy Abstraction)

В подходе CEGAR предикаты, для которых строится абстрактное дерево достижимости, постоянно изменяются, в результате дерево нужно перестраивать каждый раз заново. Для решения данной проблемы используется метод, описанный в [9]. Данный метод получил название «ленивая абстракция» (от англ. «Lazy Abstraction»). После осуществления анализа контрпримера (потенциального ошибочного пути) не осуществляется построение всего абстрактного дерева для всей программы «с нуля». Вместо этого, дерево пересматривается частично, только для тех путей, в которых его пересмотр мог бы повлиять на наличие ошибочных состояний.

При анализе пути контрпримера алгоритм ленивой абстракции находит в нем самую первую с конца пути вершину, из которой ошибка уже недостижима. Например, начиная с конца пути последовательно перебираются вершины, в которых ошибка достижима, до тех пор пока не дойдем до вершины, в

которой ошибка станет недостижима. Все пути в дереве, которые начинаются с этой вершины, необходимо пересмотреть. При пересмотре абстракции для таких путей используются предикаты, как предоставленные на вход, так и полученные при анализе ошибочного пути. Отметим, что остальная часть абстракции, с учетом новых предикатов, тоже может быть уточнена, и это будет сделано, если в ней находятся вершины, ранее покрытые другими пересматриваемыми вершинами. Не покрытые же ничем вершины в оставшейся части абстракции пересматривать не обязательно, поскольку, хотя пересмотр и приведет эту часть абстракции в более соответствующий реализуемым состояниям памяти вид, это не повлияет на решение о (не)достижимости ошибочного состояния, поскольку даже более грубая абстракция доказывала его недостижимость по путям в этой части абстракции.

В [9] авторами предложен расширяемый алгоритм осуществления ленивой абстракции, который при соблюдении определенных условий на расширения, всегда корректен, если он завершается.

Тем не менее, этот подход обладает тем недостатком, что если ранее рассмотренная вершина тем или иным образом попадала в очередь пересмотра абстракции, абстракцию в ней и в ее потомках необходимо пересматривать с учетом всех предикатов, когда-либо полученных при анализе контрпримеров или на вход.

В статье [10] предложено развитие подхода ленивой абстракции, преодолевающее эту трудность. В качестве решения предлагается сделать предикаты локальными для вершин дерева. На этапе уточнения предикатов новые предикаты приписываются только к вершинам на ошибочном пути. Уточнение абстракции в других вершинах не производится, что делает абстракцию еще более ленивой.

2.5. Адаптивный статический анализ (Configurable Program Analysis)

В теории известно, что анализ потоков данных широко используемый в легковесных анализаторах и методы проверки моделей, используемые в тяжеловесных анализаторах, могут быть сведены друг к другу [11-15]. Однако, данные теоретические работы мало влияют на практику верификации. Легковесные анализаторы проверяют некоторые простые свойства в больших программах. Тяжеловесные инструменты верификации нацелены на исключение ложных срабатываний для сравнительно небольших программ.

Будучи направленными на эффективность, легковесные статические анализаторы обычно нечувствительны к путям, так как даже с наиболее эффективными абстрактными множествами состояний анализаторы теряют точность в точках соединения. С другой стороны, тяжеловесные анализаторы, будучи направленными на точность, не соединяют пути выполнения вовсе, рассматривая каждый путь отдельно.

Главная идея адаптивного статического анализа состоит в том, чтобы объединить преимущества анализа потоков данных и методов проверки моделей и разработать подход, в котором можно конфигурировать их взаимосвязь. В анализе потоков данных конфигурируемость заключается в выборе соответствующего интерпретатора: абстрактного множества, функции перехода, оператора расширения. Недавние работы позволяют конфигурировать несколько интерпретаторов [13-15].

С одной стороны, в анализе потоков данных алгоритм работает с ГПУ, распространяя информацию вдоль ребер, с другой стороны, алгоритмы проверки моделей в тяжеловесном анализе разворачивают ГПУ в дерево достижимых состояний до тех пор, пока одно из состояний не будет покрыто другим состоянием.

В адаптивном статическом анализе, предложенном в [16], допускается конфигурировать алгоритм анализа CPA, выбирая оператор слияния (merge) и проверки завершения анализа пути (stop). Кроме того, в адаптивном статическом анализе можно конфигурировать сразу несколько алгоритмов CPA, осуществляющих разные виды анализа.

Оператор слияния (merge) определяет, когда две вершины дерева достижимости сливаются в одну, а когда они обходятся по отдельности. В анализе потоков данных слияние происходит всегда, когда они относятся к одной и той же точке программы. В классических методах проверки моделей вершины никогда не объединяются.

Проверка завершения (stop) определяет, когда рассмотрение данного пути останавливается в данной вершине и не рассматриваются дальнейшие продолжения выполнений. В анализе потоков данных останов происходит, когда не находится состояний включающих новые конкретные состояния, т. е. достигается неподвижная точка. В методах проверки моделей останов происходит, когда одно множество конкретных состояний, соответствующих абстрактному состоянию, является подмножеством множества конкретных состояний, соответствующих какому-то другому абстрактному состоянию.

Как было сказано выше теоретически возможно переопределить анализ, состоящий из операторов слияния и останова, как единый интерпретатор в анализе потоков данных. Однако цель адаптивного статического анализа — создать практически пригодный метод конфигурируемого анализа, на основе использования готового набора уже существующих реализаций алгоритмов анализа. Это особенно важно, когда мы хотим комбинировать несколько алгоритмов анализа одновременно. Для этого определяются составной оператор слияния и составной оператор останова, которые составлены из компонент, являющихся соответствующими операторами слияния и останова алгоритмов CPA [17,18].

Использование адаптивного статического анализа в ряде случаев позволяет увеличить скорость анализа. Например, алгоритм CPA, реализующий анализ с явными значениями совместно с предикатным анализом, позволяет

показывать недостижимость ветвей условных операторов с проверкой значений переменных, которым в программе присвоили константы, за счет отслеживания присваивания переменным константных значений. Выигрыш по скорости происходит за счет того, что не требуется тратить время на уточнение предикатной абстракции. С другой стороны, анализ с явными значениями требует дополнительную память для хранения состояний.

Адаптивный статический анализ может влиять на точность, например, подключение алгоритма CPA, реализующего анализ рекурсивных структур данных, позволяет учитывать семантику потенциально бесконечных структур данных, таких как списки, множества и т. д.

2.6. Анализ с явными значениями (Explicit Analysis)

В анализе с явными значениями для каждой переменной абстрактным состоянием является множество всех ее возможных значений. Если число хранимых возможных значений для переменной превышает некоторое установленное пороговое число, то с этого момента считается, что переменная может принимать любое значение. Подробная формализация этого вида анализа описана в статье [18] в разделе 3.1 «CPA+ for Explicit Analysis».

При конструировании переходов учитывается семантика инструкций, присваивающих переменным из абстрактного состояния явные значения, например, константы или выражения, которые могут быть вычислены в константы в данном абстрактном состоянии. Также учитывается семантика выражений в условных операторах, для которых можно вывести явное значение переменной. Анализ с явными значениями достаточно прост в реализации и не требует предварительной настройки точности (такой как выведение предикатов для абстракции).

Анализ с явными значениями достаточно эффективен в случае, когда число переменных и принимаемых ими возможных значений относительно невелико. Но при увеличении этого числа он либо теряет точность (при малом установленном пороговом значении), либо требует слишком много ресурсов (при большом). Предикатная абстракция гораздо лучше справляется с большим числом переменных, но требует предварительного и довольно сложного анализа контрпримера для вывода нужных предикатов. Адаптивный статический анализ позволяет совмещать анализ с явными значениями переменных и предикатную абстракцию.

2.7. SMT/SAT решатели

В широком смысле решающие процедуры (от англ. «decision procedures», или решатели, от англ. «solvers») – это алгоритмы, которые, приняв на вход задачу разрешимости, то есть вопрос, сформулированный в рамках какой-либо формальной системы (аксиоматической теории), и требующий ответа «да» или «нет», выдают на выходе соответствующий корректный результат. В узком смысле мы рассматриваем решатели для алгоритмически разрешимых задач,

сформулированных в рамках теорий первого порядка, и используемых на практике в областях верификации, доказательства корректности, оптимизации и др. Поскольку возможности многих технологий в этих областях ограничены возможностями используемых решателей, последние остаются объектом многих активных исследований во всем мире, как в академической среде, так и в промышленности.

Решающие процедуры можно разделять по поддерживаемым ими логикам и теориям. Решатели для логики высказываний, т.е. классической логики нулевого порядка, которые по сути являются инструментами для решения задачи выполнимости булевых формул (ВЫП), называются SAT-решателями. Решатели для формул классической логики первого порядка с равенством, заданных в рамках комбинации некоторых аксиоматических теорий, называются SMT-решателями. Решатели могут ограничиваться поддержкой только некоторых видов формул, например только формул без кванторов, в том числе решатель может поддерживать различные теории. Среди наиболее часто используемых на практике теорий – вещественная и целочисленная линейная арифметика, неинтерпретируемые функции, массивы и битовые векторы.

Практически все современные решатели для пропозициональной (логической) части формулы используют либо схему DPLL (Davis-Putnam-Logemann-Loveland) [19], основанную на поиске с возвратом, для решения задачи в виде КНФ, либо основаны на суперпозиционном исчислении (от англ. «superposition calculus»), расширении резолютивного вывода. Для взаимодействия с решающими процедурами для подформул, заданных в рамках какой-либо теории, используются в основном техники комбинирования Нельсона-Оппена (Nelson-Oppen Combination Procedure), пропозиционального кодирования (от англ. «propositional encodings») и отложенного комбинирования теорий (DTC, Delayed Theory Combination) [19]. Отложенное комбинирование теорий предложено в статье [20] и используется в решателе MathSAT.

2.8. Интерполяция

Интерполяция используется на этапе уточнения абстрактного домена для последующего уточнения абстракции, в первую очередь для уточнения предикатов в предикатной абстракции. Исходными данными для уточнения предикатов является невыполнимая формула пути. В результате должны быть найдены новые предикаты, таким образом, чтобы при повторном построении абстракции исключить из нее невыполнимый путь. Метод итерполяции состоит в том, чтобы использовать интерполянты Крейга (Craig), позволяющие находить неявные зависимости [10] (см. определение интерполянта в [21]). Интерполянт существует всегда, когда формулы заданы в логике предикатов первого порядка. В статье [22] показано, что интерполянт может быть построен за линейное время по размеру резолютивного

доказательства для формул в теориях равенства с неитерпретируемыми функциями и линейных неравенств.

Интерполяционные методы в дополнение к предикатам позволяют определять точки в программе, для которых эти предикаты полезны. Это позволяет рассматривать не глобальное множество предикатов, а разные множества предикатов для разных точек программы и соответственно перестраивать абстракцию только для частей программы, связанных с изменившимся набором предикатов.

2.9. Ограничиваемая проверка моделей (Bounded Model Checking)

Метод ограничиваемой проверки моделей (от англ. BMC, Bounded Model Checking) [23] — это одна из наиболее часто применяемых формальных техник при разработке полупроводниковых устройств. Метод обязан своим успехом большим возможностям SAT решателей. Он был предложен в 1999 как вспомогательный для неограниченного метода проверки моделей на основе BDD.

Этот метод основан на разворачивании графа состояний программы на конечное число шагов. На каждом этапе проверяется, может ли спецификация быть нарушена не более, чем за текущее выбранное число шагов. Если нарушение не обнаружено, выполняется проверка условий разворачивания (от англ. «unwinding assertions»), таких как условия в заголовках циклов, для того, чтобы исключить дальнейшее разворачивание графа по неосуществимым путям. Такая проверка может показать, что дальнейшее увеличение числа шагов бессмысленно, т.к. ни одно из условий разворачивания не выполнено. Это доказывает корректность исходной программы. В противном случае, число шагов увеличивают и проверку повторяют до тех пор, пока не обнаружат нарушение спецификации, либо не выйдут за пределы указанного максимального числа шагов. Проверка корректности развёрнутого фрагмента графа состояний обычно подразумевает кодирование условия нарушения спецификации в виде задачи для решателя.

В качестве оптимизаций применяются отдельное раскрытие циклов и вычисление порога полноты (от англ. «completeness threshold»).

В BMC, как правило, используются решатели, поддерживающие битовую арифметику.

BMC – это наилучший метод для нахождения поверхностных ошибок, предоставляющий полную трассу контрпримера в случае их обнаружения. Инструментам, реализующим этот метод, как правило, свойственна высокая точность анализа. Они поддерживают широчайший набор конструкций в программе, включая динамически выделяемые структуры в памяти, операции с массивами, использование арифметики указателей, битовые операции над числами, ограниченность размеров целочисленных типов данных (например, перечислимых) и другие, не требуя при этом встроенного знания о структурах

данных в программе. С другой стороны, доказательство отсутствия ошибок гарантируется только для простых программ, т.е. программ без глубоких циклов.

2.10. Анализ рекурсивных структур данных (Shape Analysis)

Анализ рекурсивных структур данных [24-26] — это анализ, который моделирует потенциально бесконечные рекурсивные структуры данных в куче, используя абстракции, основанные на графах связей. Примерами таких структур данных являются списки, множества, отображения, деревья, и т.д.. Для построения графов необходимо задание классов связей, описывающих связи между элементами структуры данных, которые необходимо отслеживать. Класс связей в том числе задает предикаты на данные, хранимые в элементах, например на значения данных, хранящиеся в списке. Граф может включать суммарные вершины, описывающие одно или несколько состояний, что дает возможность представлять бесконечные части структуры данных, неразличимые с точки зрения заданного класса связей.

Анализ рекурсивных структур данных позволяет использовать предикаты, которые сложно найти другими методами. Например, при использовании предикатной абстракции этап уточнения может выполняться бесконечное число раз, т. к. каждый раз будет находиться предикат для конечного экземпляра бесконечной структуры данных.

2.11. Модульность анализа

Под модульностью анализа понимается возможность анализировать программу по частям, используя результаты анализа части программы для анализа программы в целом. Наиболее быстрые подходы, применяемые в легковесных инструментах, анализируют каждую функцию программы отдельно, составляя для каждой проанализированной функции аннотацию, хранящую значимые результаты анализа. Для анализа потока данных аннотация может хранить некоторые элементы решетки. В случае использования предикатной абстракции аннотация может хранить абстрактное состояние на выходе из функции при условии её выполнения из начального абстрактного состояния [27]. Модульность может обеспечиваться не только на уровне функций. В подходе [28] запоминаются результаты анализа рёбер в абстрактном дереве достижимости, которые могут содержать как вызовы функций целиком, так и отдельные итерации циклов. Таким образом, в случае повторного прохода по аналогичному ребру, переиспользуются результаты предыдущего анализа. Модульность во многих случаях ускоряет анализ, а кроме того, делает инструмент масштабируемым.

2.12. Генерация тестов

Под генерацией тестов понимаются методы, которые позволяют генерировать входные параметры программы, так чтобы выполнение программы с

заданными параметрами обеспечивало покрытие того или иного участка программы. Нацеливание инструмента генерации тестов на покрытие ошибочного состояния программы, определяемого переходом на ошибочную метку, дает возможность использовать их для проверки свойств достижимости.

Инструмент статической верификации	**CEGAR**	**Предикатная абстракция**	**Абстрактное дерево достижимости**	**Ленивая абстракция**	**Адаптивный анализ**	**Анализ с явными значениями**	**Интерполяция**	**Решатель**	**Ограничиваемая проверка моделей**	**Анализ рекурсивных структур данных**	**Модульность анализа**	**Генерация тестов**
Blast	+	+	+	+	±	+	+	SMT				
CPAchecker	+	+	+	+	+	+	+	SMT			±	
HSF(C)	+	+	+				+	CUST				
Satabs	+	+						SAT				
Slam	+	+						SMT				
Wolverine	+		+	+			+	SAT				
Yogi	+	+						SMT			+	+
Cbmc								SAT	+			
Esbmc								SMT	+			
Llbmc								SMT	+			
FShell								SAT	+			+
Predator								—		+		

Табл. 1. Методы и техники инструментов статической верификации

3. Характеристики инструментов статической верификации

3.1. Поддержка конструкций анализируемой программы

Булевы выражения – это выражения логической конъюнкции, дизъюнкции, отрицания, равенства, неравенства. Не включает побитовые или/и/отрицания и т. д.

Линейные выражения – это выражения сложения, вычитания, умножения на константу, линейные неравенства, равенства.

Умножение/деление могут встречаться как для целых типов, так и для вещественных. Для последних требуется поддержка семантики вещественных чисел.

Битовое представление типов позволяет учитывать особенности представления целых типов в виде ограниченного набора бит, учитывать при этом переполнение для целых и округление для вещественных типов.

Битовые выражения включают битовые операции, такие как и/или/отрицание, исключающее или, сдвиги и т. д.

Выражения с указателями включают присваивание указателей и арифметику указателей, такую как сложение адреса с целым, разность адресов и т. д. Задача инструмента верификации – отслеживать изменения указателей так, что изменение памяти доступной по одному указателю, отражается при чтении этой же памяти по другому указателю на эту же память. Здесь можно выделить два подкласса: инструменты верификации, которые умеют работать только с присваиваниями указателей, и инструменты верификации, которые отслеживают арифметику указателей.

Поддержка рекурсивных структур данных означает возможность отслеживания взаимосвязей указателей в куче и нахождения общих шаблонов связей, дающих возможность распознавать потенциально бесконечные структуры данных и строить только те части из этих структур, которые необходимы для проверки свойства.

Поддержка указателей включает разыменование переменной указателя на функцию так, что будет осуществлен переход выполнения на тело функции, соответствующее значению указателя в данной переменной.

Поддержка рекурсии в наиболее полном случае означает нахождение соответствующих инвариантов для доказательства проверяемого свойства. Подходы на основе составления аннотации вызова функции осуществляют анализ с помощью использования аннотации для анализа самой себя. Также есть подходы с ограничением на глубину рекурсии.

Поддержка многопоточности подразумевает, что учитывается семантика выполнения в нескольких потоках, которая моделируется, например, с помощью чередования выполнения инструкций разных потоков. Имеет

значение, какие примитивы синхронизации поддерживает данный инструмент верификации, поддерживается ли динамическое создание потоков.

3.2. Масштабируемость

Под масштабируемостью понимается возможность применения инструмента верификации к большим и очень большим размерам исходного кода программы. Наиболее серьезных успехов в плане масштабируемости добиваются легковесные подходы к верификации, позволяя анализировать программы, состоящие из миллионов строк исходного кода, за счет использования модульных подходов к анализу. Тяжеловесные подходы способны анализировать десятки тысяч строк кода. Они могут использовать модульные подходы, а также подходы с абстракцией и уточнением.

3.3. Доказательство свойств

Инструменты, осуществляющие доказательство заданного пользователем свойства, как правило, предоставляют некоторые гарантии правдивости ответа, т. е. того, что свойство действительно не нарушено. Например, в CEGAR подходах абстракция программы может выбираться таким образом, что все выполнения, нарушающие заданное свойство в программе, присутствуют и в абстрактной модели, с точностью до поддерживаемых инструментом конструкций языка программирования при построении абстракции. Таким образом, отсутствие нарушения в абстрактной модели гарантирует ее отсутствие в программе. В подходах без доказательства гарантии отсутствия ошибок могут быть либо для очень простых программ, либо их может не быть вовсе. Например, в подходе ограничиваемой проверки моделей доказательство не осуществляется и утверждать об отсутствии нарушения свойства можно только вплоть до заданного ограничения на количество итераций цикла.

3.4. Проверяемые свойства

Свойства достижимости говорят о том, что программа не достигает состояний, в которых она делает что-то плохое. Например, программа не должна достигать состояния, в котором не выполнено выражение, передаваемое функции assert или состояния, помеченного специальной меткой ошибки.

Свойство завершаемости означает, что программа когда то завершится. Для проверки этого свойства необходимо задать точку входа в программу и точки выхода, которых должна достичь программа.

4. Инструменты статической верификации

4.1. Blast

Инструмент Blast [6,29] реализует подход CEGAR с использованием декартовой предикатной абстракции. В инструменте реализован метод ленивой абстракции на основе абстрактного дерева достижимости.

Реализован вариант адаптивного анализа, позволяющий параллельно с построением элементов предикатной абстракции строить элементы анализа потоков данных, основанного на задании решетки (lattice) в качестве алгоритма CPA, с различными вариантами функций слияния (merge) и функций покрытия (stop). В инструменте реализована решетка SymbolicStore, осуществляющая анализ с явными значениями. Для получения новых предикатов и уточнения абстракции инструмент использует интерполирующие решатели. Анализ контрпримера на выполнимость осуществляется решателями, работающими в формате SMT-LIB.

Инструмент Blast поддерживает программы, содержащие булевы и линейные выражения. Поддерживаются конструкции вызова функций по имени, с помощью подстановки тела функции. Поддерживает семантику присваивания указателей на базовые типы, обращения к полям структуры, присваивание полей и структуры целиком. Не поддерживаются операции сложения и вычитания с адресами, например, не поддерживаются смещения для элементов массива, получение внешней структуры с помощью вычитания смещения поля вложенной структуры относительно внешней. Поддерживаются указатели на функции. Анализ рекурсивных функций осуществляется с помощью ограничения на глубину раскрытия. От неподдерживаемых конструкций Blast абстрагируется до тех пор пока они не потребуются для доказательства свойства. В этом случае может быть выдан ложный вердикт.

Масштабируемость в Blast обеспечивается за счет применения абстракции. В ядре ОС Linux встречаются драйверы размером порядка 40 тыс. строк, на которых Blast успешно завершается [30]. Абстракция в Blast строится таким образом, чтобы включать все ошибочные пути программы, что позволяет доказывать выполнение заданного свойства. Blast поддерживает проверку свойств достижимости, задаваемых в виде метки, приписанной к точке в программе.

4.2. CPAchecker

Инструмент CPAchecker [31] во многом аналогичен инструменту Blast. В отличии от инструмента Blast, адаптивный статический анализ реализован в нем полностью, т. е. инструмент полностью компонуется из алгоритмов анализа CPA, задаваемых в конфигурации. Это позволяет добавлять новые виды анализа или изменять анализатор без его перекомпиляции и делает его удобным для проведения экспериментов с различными вариантами анализа.

В инструменте CPAchecker реализована поддержка длинных переходов в абстрактной модели (больших и адаптивных блоков), которые включают не только базовые блоки, содержащие выражения без вызова функций и ветвлений, но и слияния путей по выходу из цикла и вложенные вызовы функций [32].

Инструмент отчасти обеспечивает модульность анализа, реализуя технику сохранения абстрактных переходов, также называемых абстрактными блоками (Abstract Block Memorization) [28], для того чтобы не перевычислять их при последующем прохождении. Для этого из вычисленного абстрактного блока удаляется информация о зависимости с другими блоками, так что абстрактный блок можно использовать при выполнении в другом абстрактном состоянии. Например, для вложенного цикла из абстрактного состояния удаляется информация о переменных внешнего цикла. На следующей итерации внешнего цикла для анализа итерации внутреннего цикла используется сохраненный абстрактный блок.

В части поддержки конструкций языка программирования CPAchecker также аналогичен инструменту Blast. В настоящий момент CPAchecker поддерживает выражения с присваиваниями указателей на примитивные типы. Присваивание указателей на структуры и арифметические операции не поддерживаются.

По умолчанию CPAchecker считает, что все функции без тела являются неинтерпретируемыми, т. е. для одинаковых аргументов возвращают одинаковые значения, и что они не имеют побочных эффектов. Существует возможность задания неопределенных функций, которые каждый раз выдают произвольное значение вне зависимости от параметров.

4.3. HSF(C)

HSF(C) [33] — это инструмент статического анализа, основанный на использовании подхода CEGAR и булевой абстракции программ.

Инструмент поддерживает булевы и линейные выражения, умеет игнорировать неподдерживаемые конструкции. Не учитывает семантику выражений с указателями.

Отличительной особенностью инструмента является то, что он позволяет проверять не только свойства достижимости, но и завершаемости [34]. Доказательство завершаемости состоит в том, чтобы предложить функции ранжирования и проверить, что соответствующее им отношение фундировано. Долгое время в предлагаемых решениях данной задачи старались построить единственную функцию, обладающую свойством ранжирования. Однако, такой подход не давал возможности полностью автоматизировать доказательство и был непригоден для практически значимых программ.

Основным отличием от предыдущих инструментов является смещение основной вычислительной нагрузки от задачи нахождения кандидатов

ранжирующей функции к задаче проверки того, что предложенная функция является функцией ранжирования. В классическим методе основная нагрузка состоит в том, чтобы построить такую функцию, чтобы она сразу же обладала свойством ранжирования. Более того, в предлагаемом подходе строится не одна функция, а набор догадок о возможных функциях ранжирования, некоторые из которых могут быть неудачными догадками. Таким образом, множество догадок как объединение ранжирующих функций не обязано само быть функцией ранжирования, а может являться только надмножеством над ней. Это происходит в силу того, что объединение ранжирующих функций в общем случае не является ранжирующей функцией.

Инструмент также реализует методы многопоточного анализа, опирающиеся на автоматическое обнаружение переходов, выполняемых окружением, используя предикатную абстракцию переходов. Схема абстракции и уточнения основана на хорновских дизъюктах, т.е. ограничения на переходы записываются в виде хорновских дизъюнкций (англ. «Horn clause»), затем эти ограничения решаются с помощью общего алгоритма для хорновских дизъюнкций без рекурсии.

4.4. Satabs

Satabs [35] — это инструмент автоматической верификации, основанный на использовании подхода CEGAR и булевой абстракции программ. Отличительной особенностью инструмента является использование SAT решателей вместо SMT решателей, что позволяет работать с битовыми представлениями чисел и, таким образом, более точно моделировать семантику языка Си, включая семантику переполнения, арифметику указателей, массивы и объединения.

Для получения новых предикатов и уточнения абстракции Satabs использует синтаксические методы [36], основанные на вычислении слабейшего предусловия для трассы контрпримера. Эти методы не гарантирует получение предикатов, делающих заданный путь невыполнимым, поэтому несмотря на поддержку большинства конструкций языка при составлении абстракции, верификация может завершиться неудачно.

В своей работе инструмент опирается на сторонние инструменты проверки моделей, такие как Cadence SMV [37], Booро [38], Boom [39]. Инструмент строит предикатную абстракцию существования и передает ее стороннему инструменту проверки моделей, а затем использует полученный от него результат для дальнейшего уточнения абстракции или для выдачи итогового результата. Инструмент Satabs умеет строить абстракции для многопоточных программ [40], опираясь при этом на возможности по проверке параллельных программ сторонними инструментами проверки моделей.

4.5. Slam (Slam2)

Slam (Slam2) – это инструмент автоматической верификации, основанный на использовании подхода CEGAR и булевой абстракции программ. Во многом благодаря успехам этого инструмента предикатная абстракция в настоящий момент является доминирующей техникой статической верификации программного обеспечения [41].

В инструменте Slam используется построение абстракции существования. Абстрактная программа, соответствующая данному построению, является булевой программой, в которой используются переменные только булева типа и те же графы потоков управления, что и в исходной программе, включая вызовы процедур [42].

Спецификация для проверки в этом инструменте, так же, как и в Blast, может быть сформулирована в виде условия достижимости. При этом проблема проверка достижимости для булевых программ разрешима, несмотря на неограниченность стека вызовов функций. Интуитивно это основано на том, что последующее состояние определяется полностью по данным на вершине стека и значениям глобальных переменных, которые принимают конечное множество значений [43].

Для поиска набора новых предикатов для уточнения абстракции по контрпримеру в Slam вначале используется одно из простых решений – это поиск ядра недостижимости (от англ. «unsatisfiable core»), подмножества предикатов, встречающихся в формуле пути, конъюнкция которых невыполнима. Есть несколько способов нахождения данного множества. В инструменте Slam для этого используется жадный алгоритм [44]. Если предикатов, полученных в результате поиска ядра недостижимости, оказывается недостаточно для отсечения неосуществимого контрпримера, используется синтаксический метод поиска новых предикатов, описанный в [45]. Обоснование его полноты можно найти в [46].

Инструмент Slam используется в Microsoft Static Driver Verifier (SDV), подробно рассмотренном в соответствующем подразделе, для проверки корректности использования драйверами интерфейсов, предоставляемых ядром ОС Microsoft Windows. О некоторых результатах его применения с этой целью написано в статье [47].

4.6. Wolverine

Инструмент Wolverine [48] основан на подходе CEGAR. Он использует абстрактное дерево достижимости при построении переходов которого не используется вычисление постусловия. Абстрактные состояния в вершинах дерева уточняются посредством интерполяции по контрпримерам.

Встроенный интерполятор Wolverine версии 0.5c не предоставляет поддержки линейной арифметики, инвариантов с кванторами, арифметики с указателями.

4.7. Yogi

Инструмент Yogi [49] объединяет возможности статического и динамического анализа.

Динамический анализ позволяет получить аппроксимацию множества состояний снизу, т. е. состояния и пути, перебираемые в динамическом анализе, реально достижимы. Данный анализ традиционно хорошо подходит для того, чтобы показать, что ошибки есть. Тогда как сильная сторона статического анализа заключается в том, чтобы показать, что ошибок нет.

В работе [27] предлагается метод, позволяющий одновременно выполнять статический и динамический анализы. Статический анализ строит «может» (от англ. «may») абстракцию, включая в нее все осуществимые и, возможно, неосуществимые пути. Динамический анализ строит «должен» (от англ. «must») абстракцию, включая только осуществимые пути, но, возможно, не все.

Объединение статического и динамического анализа позволяет статическому анализу не доказывать неосуществимость путей, для которых динамический анализ показал осуществимость, а динамическому анализу позволяет не искать ошибку там, где статический анализ показал, что ее быть не может. Этот метод реализован в инструменте Yogi [49]. Инструмент встраивается в Microsoft SDV, используемый для верификации драйверов ОС Windows. По результатам, приведенным в статье [49], инструмент сравним с основным используемым в настоящее время инструментом Slam.

4.8. Cbmc

Инструмент Cbmc [50] — это инструмент ограничиваемой проверки моделей для верификации программ на ANSI-C и C++.

Поддерживает семантику большинства выражений, опираясь на решатели, поддерживающие решение с представлением целых чисел в виде битовых векторов. Для поддержки операций умножения/деления используется библиотека, вычисляющая результат через операции сложения и вычитания. Таким образом, Cbmc заменяет операции умножения/деления на вызов функций библиотеки, которые затем анализируются. Рекурсия анализируется с помощью ограничения на глубину раскрытия.

4.9. Esbmc

Инструмент Esbmc [51] во многом аналогичен Cbmc, и использует его компоненты в процессе верификации. В отличие от Cbmc он использует SMT решатели. Кроме того, в Esbmc реализована возможность проверки параллельных программ посредством чередования инструкций различных потоков с ограничением на количество переключений между потоками.

4.10. Llbmc

Инструмент Llbmc [52] реализует BMC подход, используя внутреннее представление LLVM. Он направлен на представление типов с битовой точностью и нахождение ошибок, связанных с неправильной работой с памятью. Инструмент использует SMT решатель Stp, который поддерживает битовые векторы и массивы.

4.11. FShell

FShell [53] — это инструмент генерации тестов для C программ. Пользователем задается критерий тестового покрытия, который необходимо достигнуть. Исходя из этого критерия методом BMC генерируются формулы, решение которых дает входные данные, необходимые для покрытия того или иного пути. В качестве критерия покрытия может также выступать достижимость метки в C программе. Инструмент использует SAT решатель MiniSAT.

4.12. Predator

Predator [54] — это анализатор программ, построенный на основе логики разделения (separation logic) и нацеленный на анализ программ, содержащих динамические списки. Формулы логики разделения, описывающие потенциально бесконечные структуры связей в куче, представляются в виде графов. Инструмент не использует решатели и реализован как встраиваемый модуль к GCC.

Характеристики инструментов статической верификации представлены в таблице 2.

Инструмент статической верификации		**Поддержка конструкций входной программы**														
		Булевы выражения	**Линейные выражения**	**Умножение/деление**	**Вещественные числа**	**Битовое представление типов**	**Битовые выражения**	**Выражения с указателями**	**Рекурсивные структуры данных**	**Указатели на функции**	**Рекурсия**	**Многопоточность**	**Масштабируемость**	**Доказательство свойства**	**Свойства достижимости**	**Свойства завершаемости**
CEGAR	Blast	+	+	−	−	−	−	±	−	+	±	−	±	+	+	−
	CPAchecker	+	+	−	−	−	−	±	−	+	±	−	±	+	+	−
	HSF(C)	+	+	−	−	−	−	−	−	−	−	+	±	+	+	+
	Satabs	+	+	+	+	±	±	±	−	+	±	+	±	+	+	−
	Slam	+	+	−	−	−	−	+	−	−	−	−	±	+	+	−
	Wolverine	+	±	−	−	−	−	±	−	−	−	−	±	+	+	−
	Yogi	+	+	−	−	−	−	+	−	−	±	−	+	+	+	−
BMC	Cbmc	+	+	+	+	+	+	+	−	+	±	−	−	−	+	−
	Esbmc	+	+	+	+	+	+	+	−	+	±	+	−	−	+	−
	Llbmc	+	+	+	−	+	+	+	−	−	±	−	−	−	+	−
	FShell	+	+	+	−	+	+	+	−	+	±	−	−	−	+	−
	Predator	+	±	−	−	−	−	+	+	+	±	−	+	+	+	−

Табл. 2. Характеристики инструментов статической верификации

4.13. Результаты соревнований

В таблице 3 представлены результаты международных соревнований по верификации SV-COMP-2012 [55] в категории DeviceDrivers64. Данная категория состоит из драйверов, подготовленных к верификации.

Категория DeviceDrivers64		**Корректные результаты (всего 41 файл)**	**Упущенные ошибки**	**Ложные предупреждения**	**Баллов (макс. 66)**	**Среднее время работы (сек.)**	**Среднее время для корректных результатов (сек.)**
CEGAR	Blast	33			55	76	42
	CPAchecker	33		2	49	117	15
	HSF(C)	–					
	Satabs	17			32	634	188
	Slam	–					
	Wolverine	12			16	585	108
	Yogi	–					
BMC	Cbmc	–					
	Esbmc	7	1		10	176	124
	Llbmc	3		2	1	293	37
	FShell	0			0	0,5	0
	Predator	0			0	44	0

Табл. 3. Результаты соревнований SV-COMP 2012

Каждый подготовленный драйвер включает необходимое окружение и проверки свойств достижимости, встроенные непосредственно в код. Таблица 3 показывает результаты запуска инструментов, участвовавших в соревнованиях, на наборе из 41 входного файла. Прочерком обозначены инструменты, не принимавшие участия в соревнованиях или в данной категории.

Инструмент выдает вердикт SAFE, если он не обнаружил ошибок в программе. Если он обнаружил ошибку, то выдает UNSAFE. Если же по каким то причинам инструмент не может определенно сказать имеется ли в программе ошибка, то выдается вердикт UNKNOWN.

Корректным считается результат, когда либо при вердикте инструмента UNSAFE в программе имеется ошибка, либо для программы без ошибок выдается результат SAFE. При подсчете очков в соревновании поощрялись правильные вердикты SAFE (+2 балла), правильные UNSAFE (+1 балл). За UNKNOWN баллы не начислялись. С другой стороны, за неправильные вердикты снимали баллы в двойном количестве. За каждый неправильный UNSAFE (ложное срабатывание) -2 балла, за неправильный SAFE (упущенная ошибка) -4.

Среднее время считалось от общего времени на всех задачах, в том числе и тех, на которых был получен вердикт UNKNOWN, например по причине превышения лимита по времени, который составлял 15 минут. Среднее время для корректных результатов включает только времена запусков на задачах, для которых был получен правильный результат. Отметим, что в соревнованиях учитывалось только время на корректных результатах, так как при некорректных результатах снимаются или не начисляются баллы. С точки зрения пользователя инструмента важно знать, сколько в среднем составит время ожидания ответа инструмента верификации, и в этом случае необходимо учитывать время всех запусков, в том числе неуспешных.

Из таблицы 3 можно видеть, что количество корректных результатов для CEGAR подходов значительно выше чем для BMC. Это объясняется лучшей масштабируемостью на драйверах, которые являются большими программами (несколько тысяч строк кода), относительно программ, представленных в других категориях. В тоже время видно, что инструменты, использующие точную семантику для целых чисел и битовых операций, например, Satabs и Wolverine, требуют больше времени. Ещё одной особенностью, выявленной в результате соревнований стало то, что некоторые инструменты существенно опираются на предположение о доступности определений всех функций, в том числе библиотечных, и что вся используемая память выделяется и освобождается в коде анализируемой программы. Для драйверов устройств, представленных в категории DeviceDrivers64, данное предположение принципиально не выполнено, так как драйвер использует различные структуры данных, инициализируемые в сердцевине ядра, а также использует программный интерфейс сердцевины ядра.

5. Системы тяжеловесной статической верификации драйверов устройств

Для широкомасштабного применения инструментов тяжеловесной статической верификации к драйверам устройств требуется разрешить целый ряд вопросов. При этом часть из них являются специфичными для верификации драйверов устройств. Например, инструменты верификации ожидают, что у анализируемой программы есть точка входа, иными словами, функции *main* или её аналог. У драйверов такая точка входа есть — функция инициализации драйвера. Но в функции инициализации обычно происходит только инициализация структур драйвера и регистрация в ядре ОС функций-обработчиков событий. А основная функциональность драйвера находится в функциях-обработчиках, которые вызываются ядром ОС для обработки событий, таких как обращение к устройству из прикладных приложений, обработка прерываний и т.д. В результате, если инструменты верификации начнут анализ с функции инициализации, то основная функциональность драйвера останется для них недостижимой, то есть непроверенной. Поэтому для проведения верификации драйверов требуется создание модельного окружения драйвера в виде искусственной точки входа, в которой сначала вызывается функция инициализации, а потом выполняются обращения к функциям-обработчикам аналогично тому, как это происходит при реальной работе драйвера. Для проведения единичных экспериментов такое окружение может быть создано вручную, но для широкомасштабного применения инструментов верификации модельное окружение должно генерироваться автоматически.

Вторая потребность заключается в необходимости автоматизировать извлечение информации о составе драйвера и настройках его компиляции из уже имеющихся данных, предназначенных для сборки драйверов. При этом важно учитывать, что с одной стороны у драйверов есть много различных зависимостей от функций из основной части ядра ОС, а с другой — что для эффективного применения статической верификации количество строк кода должно быть не очень большим.

Ещё один вопрос возникает с формулировкой проверяемых свойств, например, таких как корректное использование драйвером интерфейсов ядра ОС. Для постановки задачи инструменту верификации требуется описать, какая ситуация является ошибочной, причём это описание в случае широкомасштабного применения должно находиться вне кода драйверов. Некоторые инструменты верификации предоставляют встроенные возможности для формулировки проверяемых свойств, для других это должно быть реализовано отдельно.

Кроме того, для широкомасштабного применения требуется минимизации участия человека в настройке инструментов и максимально удобный интерфейс для их использования, что включает в себя удобство запуска верификации и удобство анализа её результатов.

Для разрешения всех этих вопросов фактически требуется разработка специальной системы верификации драйверов, автоматизирующей применение инструментов статической верификации. Такая система сама по себе является нетривиальным программным продуктом, поэтому желательно, чтобы система верификации поддерживала несколько инструментов верификации, в особенности с учетом динамичного развития в этой области.

Рассмотрим основные возможности систем верификации драйверов, разработанных в рамках существующих проектов по верификации драйверов с использованием тяжеловесных методов статического анализа кода.

Наиболее долгую историю имеет система верификации Microsoft SDV [56]. Данная система предоставляет широкие возможности по верификации с помощью статического анализа драйверов ОС Microsoft Windows, как входящих в состав самой ОС, так и поступающих от сторонних разработчиков в существующей программе сертификации драйверов. К особенностям системы относятся следующие:

- Для создания окружения от пользователя требуется вручную аннотировать исходный код драйверов, указав в нем роли каждой из функций обработчиков, после этого окружение генерируется автоматически.
- Проверяемые правила корректности формализуются с помощью языка SLIC [57], в котором связь проверок с исходным кодом драйвера задается с помощью аспектно-ориентированных конструкций, перехватывающих вызовы функций ядра. В настоящее время уже выделен набор из более чем 210 правил, а в исследовательской версии была реализована возможность добавления новых правил.
- Известно, что в собственных исследовательских целях разработчики Microsoft SDV могут подключать два инструмента статической верификации: Slam и Yogi [49]. Подключение инструментов верификации сторонними разработчиками не предусмотрено.
- Имеется возможность просмотреть сводную статистику по всем проверяемым правилам для анализируемого драйвера. Для найденных ошибок можно просмотреть представленный удобным и наглядным образом путь из точки входа в ошибочное состояние (трассу ошибки), который связан с исходным кодом драйверов и ядра ОС.

Существует также несколько систем, использующих тяжеловесные методы статического анализа кода для верификации драйверов ядра ОС Linux: DDVerify [58], разработка университета Карнеги-Меллон (США), Avinux [59], разработка университета города Тюбинген (Германия) и система верификации Linux Driver Verification [30], разработанная в Институте системного программирования РАН (Россия).

Особенности системы DDVerify таковы:

- Информацию о составе и настройках сборки драйверов DDVerify получает, используя собственные файлы сборки без учёта файлов сборки ядра. Поэтому система не учитывает специфику компиляции ядра в полной мере.
- Для создания окружения используется модель ядра для некоторых типов драйверов. Разработчиками были написаны модели ядра для трёх типов драйверов, а всего их несколько десятков.
- Правила корректности задаются как часть модели ядра. Код ограничений, накладываемых правилом, задается вместе с кодом, описывающим семантику функции. Поэтому для использования DDVerify требуется существенно изменять заголовочные файлы ядра, что осложняет поддержку применимости системы для различных версий ядра ОС Linux.
- Система позволяет подключать два инструмента статической верификации: Cbmc [50] и Satabs [35].
- Для анализа результатов анализа имеется специальный плагин для интегрированной среды разработки Eclipse. В частности, имеется возможность анализа трассы ошибки с одновременным просмотром соответствующего исходного кода драйверов и ядра ОС Linux, что существенно облегчает анализ трасс ошибок.

Система Avinux, которая также предназначена для верификации драйверов ядра ОС Linux, обладает следующими характерными особенностями:

- Получение исходного кода драйвера для последующей верификации происходит на основе встраивания в процесс сборки ядра путём модификации файлов, описывающих сборку. Однако, Avinux предоставляет возможность автоматической работы только с единичными препроцессированными файлами. Поэтому, например, верификация драйверов, состоящих из нескольких файлов возможна только вручную, так как информация о зависимостях теряется.
- Для создания окружения драйвера требуется вручную написать функцию *main*, моделирующую взаимодействие драйвера с ядром операционной системы и, кроме того, выбрать файлы, входящие в драйвер. На основе этого код инициализации параметров генерируется автоматически [60].
- Для задания правил используются аспектно-ориентированные конструкции, похожие на конструкции SLIC.
- Система интегрирована с единственным инструментом статической верификации Cbmc [50].
- Для упрощения запуска Avinux предоставляет плагин к среде Eclipse, но средства для визуализации трассы ошибок отсутствуют.

Система верификации Linux Driver Verification [30]:

- Система интегрирована с процессом сборки ядра, поэтому вся необходимая информация о составе и настройках сборки драйверов извлекается автоматически.
- Генерация окружения осуществляется полностью автоматически на основе иерархии шаблонов, покрывающей все типы драйверов.
- Система позволяет добавлять новые правила корректности с помощью аспектно-ориентированного расширения языка программирования Си.
- Система поддерживает встраивание внешних инструментов статической верификации с помощью написания адаптеров.
- Для удобного анализа трасс ошибок и сравнительного анализа результатов система предоставляет специальные компоненты с Веб-интерфейсом.

6. Заключение

В обзоре рассмотрены инструменты тяжеловесной статической верификации для программ на языке Си, которые позволяют проверять выполнимость определённых свойств программ или проверять отсутствие в коде программы определённого класса ошибок. Основными подходами, реализуемыми в современных инструментах, являются CEGAR и BMC. Подходы CEGAR обладают большей масштабируемостью по сравнению с BMC, что наглядно продемонстрировали результаты соревнований SV-COMP-2012 в категории DeviceDrivers64. Точность работы инструментов во многом определяется возможностями используемых решателей. Чем точнее происходит моделирование семантики языка, тем больше ресурсов требуется инструменту верификации для анализа, но тем меньше возможностей для появления ложных сообщений об ошибках. Инструменты верификации представляют различные гарантии отсутствия ошибок. Если инструменты на основе CEGAR показывают отсутствие ошибок с точностью до поддерживаемых конструкций языка Си, то инструменты на основе BMC лишь проверяют отсутствие ошибок вплоть до задаваемого в качестве параметра ограничения на количество итераций циклов.

Наибольшие проблемы в инструментах на основе CEGAR вызывает анализ указателей и поддержка операций с указателями. Потенциально перспективным направлением является сочетание различных методов, позволяющих увеличить точность и скорость верификации. Для обеспечения масштабируемости перспективным направлением является разработка модульных подходов к верификации.

Широкомасштабное применение инструментов тяжеловесной статической верификации для драйверов устройств требует разработки специализированных систем верификации драйверов. Рассмотренные системы верификации драйверов демонстрируют разнообразие подходов к построению таких систем, но в то же время демонстрируют разрешимость этой задачи.

Литература

[1] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In Integrated Formal Methods (IFM), volume 2999 of LNCS. Springer, 2004.

[2] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. J. ACM, 50(5), 752–794, 2003.

[3] S. Graf and H. Saïdi. Construction of abstract state graphs with Pvs. In Proc. CAV, LNCS 1254, pages 72–83. Springer, 1997.

[4] T. Ball and S. K. Rajamani. The Slam project. Debugging system software via static analysis. In Proc. POPL, pages 1–3. ACM, 2002.

[5] T. Ball, E. Bounimova, V. Levin, and L. De Moura. Efficient evaluation of pointer predicates with Z3 SMT Solver in SLAM2. 2010.

[6] Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R. The Software Model Checker Blast. Int. J. Softw. Tools Technol. Transfer 9 (5-6), 505–525, 2007.

[7] D. Beyer, A. Cimatti, A. Griggio, M.E. Keremoglu, R. Sebastiani. Software Model Checking via Large-Block Encoding. In Proc. FMCAD, pp. 25–32. IEEE, 2009.

[8] K.L. McMillan. Lazy abstraction with interpolants. In CAV. Volume 4144 of LNCS. Springer, pp. 123–136, 2006.

[9] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. Proceedings of the 29th Annual Symposium on Principles of Programming Languages (POPL), ACM Press, pp. 58-70, 2002.

[10] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. POPL '04 Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM New York, NY, USA, 2004.

[11] P. Cousot and R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 293–308. Springer, Heidelberg, 1995.

[12] D.A. Schmidt. Data-flow analysis is model checking of abstract interpretations. In Proc. POPL, pp. 38–48. ACM Press, New York, 1998.

[13] B. Steffen. Data-flow analysis as model checking. In Proc. TACS, pp. 346–365, 1991.

[14] S. Gulwani and A. Tiwari. Combining abstract interpreters. In Proc. PLDI, pp. 376–386. ACM Press, New York, 2006.

[15] S. Lerner, D. Grove, and C. Chambers. Composing data-flow analyses and transformations. In Proc. POPL, pp. 270–282. ACM Press, New York, 2002.

[16] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007), pp. 504-518, 2007.

[17] J. Fischer, R. Jhala, and R. Majumdar. Joining data flow with predicates. In: Proc. ESEC/FSE, pp. 227–236. ACM Press, New York, 2005.

[18] D. Beyer, T. A. Henzinger, and G. Theoduloz. Program Analysis with Dynamic Precision Adjustment. ASE '08 Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society Washington, DC, USA, 2008.

[19] D. Kroening, O. Strichman. Decision Procedures. An Algorithmic Point of View. Springer, 2008.

[20] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient Satisfiability Modulo Theories via Delayed Theory Combination. CAV'05, 2005.

[21] W. Craig. Linear reasoning. J. Symbolic Logic 22, 250–268, 1957.

[22] R. Jhala and R. Majumdar. Path Slicing. Proceedings of the 27th Annual ACM Conference on Programming Language Design and Implementation, 2005. (PLDI), 2005.

[23] A. Biere, A. Cimatti, E. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. In Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg, 1999.

[24] M. Sagiv, T. Reps, R. Wilhelm. Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems (TOPLAS), 24 (3), pp. 217–298, 2002.

[25] N.D. Jones, S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. POPL '82 Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1982.

[26] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Lazy Shape Analysis. In T. Ball and R.B. Jones, editors, Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006), LNCS 4144, pages 532-546, Springer-Verlag, Berlin, 2006.

[27] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. Compositional may-must program analysis: unleashing the power of alternation. POPL '10 Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM New York, NY, USA, 2010.

[28] D. Wonisch. Block Abstraction Memoization for CPAchecker. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 532–534. Springer, Heidelberg, 2012.

[29] P. Shved, M. Mandrykin, V. Mutilin. Predicate Analysis with Blast 2.7. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 525–527. Springer, Heidelberg, 2012.

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

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

[32] S. Löwe, P. Wendler. CPAchecker with Adjustable Predicate Analysis. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 528–530. Springer, Heidelberg, 2012.

[33] S. Grebenshchikov, A. Gupta, N.P. Lopes, C. Popeea, A. Rybalchenko. HSF(C): A Software Verifier Based on Horn Clauses. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 549–551. Springer, Heidelberg, 2012.

[34] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. PLDI, 2006.

[35] G. Basler, A. Donaldson, A. Kaiser, D. Kröning, M. Tautschnig, T. Wahl. SATabs: A Bit-Precise Verifier for C Programs. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 552–555. Springer, Heidelberg, 2012.

[36] R. Jhala and R. Majumdar. Software model checking. ACM Computing Surveys, 2009.

[37] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[38] B. Cook, D. Kroening, and N. Sharygina. Symbolic model checking for asynchronous Boolean programs. In SPIN Workshop on Model Checking of Software, volume 3639 of LNCS. Springer, 2005.
[39] G. Basler, D. Kroening, and G. Weissenbacher. SAT-based summarisation for Boolean programs. In SPIN Workshop on Model Checking of Software, volume 4595 of LNCS, 2007.
[40] A. Donaldson, A. Kaiser, D. Kroening, T. Wahl. Symmetry-aware predicate abstraction for shared variable concurrent programs. Proceedings of the 23rd international conference on Computer aided verification, p.356-371, July 14-20, 2011.
[41] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In Model Checking and Software Verification (SPIN), volume 1885 of LNCS, pages 113–130. Springer, 2000.
[42] J. R. Buchi. Regular canonical systems. Archive for Mathematical Logic, 6(3-4):91, April 1964.
[43] T. Ball, and S. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Tech. Rep. MSR-TR-2002-09, Microsoft Research, 2002.
[44] T. Ball, B. Cook, S. Das, S. K. Rajamani. Refining Approximations in Software Predicate Abstraction. TACAS 2004, 2004.
[45] T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In TACAS 02: Tools and Algorithms for Construction and Analysis of Systems. Lecture Notes in Computer Science 2280. Springer-Verlag, 158–172, 2002.
[46] T. Ball, E. Bounimova, R. Kumar, V. Levin. SLAM2: Static Driver Verification with Under 4% False Alarms. FMCAD, 2010.
[47] Weissenbacher, G., Kröning, D., Malik, S.: Wolverine: Battling Bugs with Interpolants. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 556–558. Springer, Heidelberg (2012)
[48] A. V. Nori, S. K. Rajamani, S. Tetali, A. V. Thakur. The Yogi Project: Software Property Checking via Static Analysis and Testing. In TACAS '09: Tools and Algorithms for the Construction and Analysis of Systems, March, 2009.
[49] E. Clarke, D. Kroening, F. Lerda. A Tool for Checking ANSI-C Programs. In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, LNCS, Vol. 2988/168-176, 2004.
[50] L. Cordeiro, J. Morse, D. Nicole, B. Fischer. Context-Bounded Model Checking with Esbmc. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 535–538. Springer, Heidelberg, 2012.
[51] C. Sinz, F. Merz, S. Falke. Llbmc: A Bounded Model Checker for Llvms Intermediate Representation. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 542–544. Springer, Heidelberg, 2012.
[52] A. Holzer, D. Kröning, C. Schallhart, M. Tautschnig, H. Veith. Proving Reachability Using FShell. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 538–541. Springer, Heidelberg, 2012.
[53] K. Dudka, P. Müller, P. Peringer, T. Vojnar. Predator: A Verification Tool for Programs with Dynamic Linked Data Structures. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 545–548. Springer, Heidelberg, 2012.
[54] D. Beyer. Competition on Software Verification. In Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg, 2012.
[55] T. Ball, E. Bounimova, V. Levin et al. The static driver verifier research platform. Computer Aided Verification, pp. 119–122, 2010.

[56] T. Ball, S.K. Rajamani. SLIC: A specification language for interface checking. Tech. rep. Microsoft Research, 2001.
[57] T. Witkowski, N. Blanc, D. Kroening, G. Weissenbacher. Model checking concurrent linux device drivers. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE '07), pages 501-504, 2007.
[58] H. Post, W. Küchlin. Integration of static analysis for linux device driver verification. The 6th Intl. Conf. on Integrated Formal Methods, IFM 2007, 2007.
[59] H. Post, W. Kuchlin. Automatic data environment construction for static device drivers analysis. Proceedings of the 2006 conference on Specification and verification of component-based systems (SAVCBS ’06), ACM, pp. 89–92, 2006.