Моделирование памяти с использованием неинтерпретируемых функций в предикатных абстракциях[1]

М.У. Мандрыкин <mandrykin@ispras.ru>
В.С. Мутилин <mutilin@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. Одна из фундаментальных проблем в современных методах статической верификации программ состоит в точном учете семантики выражений с указателями. От точности анализа данных выражений зависит достоверность вердикта верификации. В данной работе описывается метод верификации с моделями памяти на основе неинтерпретируемых функций, который позволяет анализировать программы, содержащие выражения с указателями, в том числе указателями на структуры, массивы и выражения с адресной арифметикой. Ограничениями метода является конечность размера массивов и конечная глубина рекурсии для динамических структур данных. Предложенный метод был реализован в инструменте CPAchecker, основанном на подходе CEGAR с использованием булевой предикатной абстракции и интерполяции Крейга для получения новых предикатов при уточнении абстракции. Для решения задач проверки выполнимости формулы пути и интерполяции Крейга в CPAchecker используются интерполирующие решатели, поддерживающие бесквантворные теории вещественной или целочисленной линейной арифметики и равенства с неинтерпретируемыми функциями. В разработанном подходе состояние памяти программы представляется в виде неинтерпретируемой функции, отображающей некоторые условные адреса переменных в памяти в их значения. При записи значения по какому-либо адресу происходит смена версии неинтерпретируемой функции, представляющей состояние памяти. Эксперименты проводились на наборах международных соревнований по верификации программ SV-COMP'2016, содержащих практически значимый набор из драйверов устройств ОС Linux. На этих наборах метод показывает приемлемые результаты по времени, сокращая при этом количество упущенных ошибок и ложных предупреждений. Среди возможных дальнейших направлений исследований отметим возможность более точного разбиения на регионы памяти, так что для этих регионов выделяются независимые неинтерпретируемые функции.

1 Исследование проводилось при финансовой поддержке РФФИ в рамках проекта №15-01-03934

Ключевые слова: модель памяти, предикатная абстракция, метод уточнения абстракций по контрпримеру.

DOI: 10.15514/ISPRAS-2015-27(5)-7

Для цитирования: Мандрыкин М.У., Мутилин В.С. Моделирование памяти с использованием неинтерпретируемых функций в предикатных абстракциях. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 117-142. DOI: 10.15514/ISPRAS-2015-27(5)-7.

1. Введение

Одна из фундаментальных проблем в современных методах статической верификации программ состоит в точном учете семантики выражений с указателями. От точности анализа данных выражений зависит достоверность вердикта верификации. В данной работе описывается метод верификации с моделями памяти, учитывающими семантику выражений с указателями.

Для анализа выражений с указателями существует несколько известных подходов, применяемых в инструментах статической верификации.

Подход ограничиваемой проверки моделей (от англ. Bounded Model Checking, BMC) [1, 2] основывается на разворачивании циклов программы на конечное фиксированное число шагов k и последующей проверке возможности нарушения проверяемого свойства на глубине, не большей, чем эти k шагов. Процесс верификации повторяется для все больших значений числа k, пока не будут обнаружены все возможные нарушения проверяемого свойства или при помощи специальных проверок не будет доказано, что выполнение циклов более чем k раз в данной программе невозможно. Принудительное ограничение сверху числа k является основным ограничением на применимость этого метода для формального доказательства корректности.

В методе BMC ограничение сверху числа k существенно используется при моделировании памяти. Так как программа представляется конечным множеством конечных путей, то на каждом из них можно раскрыть выражения с разыменованием указателей, подставив вместо переменных-указателей выражения, определяющие их значения на этом пути [3]. Так как путь конечен, то можно с помощью таких подстановок прийти к выражению, в котором отсутствуют указатели. Исключения составляют лишь массивы, для моделирования которых вводятся неинтерпретируемые функции.

Метод BMC позволяет поддерживать для языка программирования широчайший набор конструкций, включая выражения с указателями, в том числе указатели на структуры, массивы, адресную арифметику, но так как рассматриваются только конечные пути, то рекурсивные структуры данных поддерживаются на конечную глубину (см. Табл. 1). На сегодняшний день метод BMC является наилучшим методом для поиска неглубоких (в смысле длины пути выполнения) ошибок. Инструменты, реализующие этот метод,

такие как CBMC[1] и F-SOFT[4], с успехом используются для поиска ошибок при разработке полупроводниковых устройств, для которых схемы часто моделируются программой на языке высокого уровня (например, C), а также в системном программном обеспечении, например, в автомобильной промышленности [5].

Вторая группа методов основана на подходе уточнения абстракции по контрпримерам (от англ. CounterExample-Guided Abstraction Refinement, CEGAR) [6, 7]. Инструменты, основанные на данном подходе, пытаются доказать некоторые свойства исходно заданной системы, предварительно ее упростив. Упрощенная система при этом, как правило, не обладает всеми свойствами исходной и поэтому в дальнейшем может потребоваться процесс ее уточнения. В общем случае к упрощенной системе — абстракции — предъявляется требование корректности (англ. soundness). Оно заключается в том, что свойства, доказанные для абстракции, должны быть верны и на исходной системе. Однако не все свойства, верные для исходной системы, верны для абстракции. В абстракции может быть обнаружено нарушение одного из проверяемых свойств, то есть контрпример, который не является осуществимым в исходной системе. Для исключения таких контрпримеров используется уточнение абстракции.

Процесс верификации с использованием уточнения абстракции по контрпримерам начинается с построения неточной (грубой) абстракции исходной системы и продолжается одним или несколькими уточнениями. Когда в абстракции обнаруживается контрпример, инструмент проверяет его на осуществимость в исходной системе (то есть проверяет, является ли данное нарушение подлинным или возникает вследствие неточности построенной абстракции). Если нарушение осуществимо, то верификация завершается и выдается сообщение об ошибке, если нет, то доказательство неосуществимости контрпримера используется для уточнения абстракции, и ее проверка снова повторяется.

В качестве абстракции программы будем рассматривать *предикатные абстракции* [8]. Предикатная абстракция программы основывается на разбиении всего множества состояний программы на подмножества с одинаковыми наборами значений выбранных предикатов. При этом предикаты в абстракции могут комбинироваться либо только с использованием конъюнкций (в таком случае предикатная абстракция называется *декартовой*), либо в том числе с использованием дизъюнкций (в таком случае абстракция называется *булевой*). Проверка контрпримера и пересчет абстракции осуществляется на основе представления последовательностей инструкций исходной программы в виде некоторых логических формул. Для решения формул используются различные решатели формул в теориях (англ. Satisfiability Modulo Theories, SMT). Для уточнения абстракции новые предикаты извлекаются из невыполнимых формул. Среди способов их извлечения имеются *синтаксические* [9] методы, а также *интерполяция*

Крейга [10, 11]. Для поиска интерполянтов Крейга на практике применяются интерполирующие решатели формул в теориях, такие как CSISAT [12] и MATHSAT [13, 14].

Рассмотрим методы реализации поддержки указателей в CEGAR-инструментах. Первый метод основан на анализе *алиасов*. В программировании словом алиас (от англ. alias — имя, прозвище) описывается ситуация, при которой какая-либо ячейка с данными в памяти программы оказывается доступна в исходном тексте (коде) под различными обозначениями (именами). Таким образом, изменение данных с использованием одного из таких обозначений неявно ведет к изменению значений, доступных по всем остальным обозначениям той же ячейки памяти.

Информация об алиасах используется для генерации ограничений, накладываемых на состояние памяти операцией присваивания, в формулах контрпримера и при пересчете абстракции. Для каждого возможного алиаса цели операции присваивания генерируется проверка, является ли его адрес равным адресу цели. В зависимости от результата этой проверки с помощью логических формул выражается либо обновление соответствующего выражения-алиаса, либо сохранение прежнего значения этого выражения.

Такой метод был реализован в инструменте BLAST 2.5 [15-17]. Анализ алиасов в нем осуществляется в терминах анализа Андерсена [18]. Для хранения и выполнения запросов к информации об алиасах используется представление BDD и алгоритм, похожий (но не совпадающий) с алгоритмом в [19]. Для того чтобы выписывать ограничения в формуле на состояния памяти, которые доступны через несколько разыменований, рассматривается *замыкание алиасов* по операции разыменования. Для каждой переменной программы выписываются алиасы на не более чем заданное число ее разыменований. Такой метод ограничивает доступы к структурам на конечную глубину (см. Табл. 1).

В инструменте BLAST 2.7 [20] метод улучшен так, что в формулу для оператора присваивания добавляются только алиасы, влияющие на ее выполнимость, которые определяются по выражениям, использованным в последующих операторах на пути. Таким образом стало возможно поддерживать произвольную глубину разыменований для указателей и структур. Однако в инструменте все еще не поддерживаются массивы, адресная арифметика и рекурсивные структуры данных.

Анализ рекурсивных структур данных (shape analysis) — это метод статического анализа программ, который позволяет находить и проверять свойства динамически выделяемых структур данных. Он обладает достаточно большой точностью при анализе различных соотношений между указателями в программе и динамически размещаемыми в памяти структурами данных, в том числе при анализе выражений с указателями. Однако, имея высокую точность, этот подход является достаточно плохо масштабируемым. Так, например, представленный на соревновании SV-COMP 2014 [21] инструмент

PREDATOR [22], который реализует данный вид анализа на основе символьных графов памяти (англ. Symbolic Memory Graphs, SMG), набрал 50 баллов из 2766 в категории, соответствующей реальным драйверам ОС Linux. Разработка другого инструмента, BLAST, реализующего *ленивый анализ рекурсивных структур данных* [23] на основе предикатов троичной логики (three-valued logic) [24], к моменту написания данной статьи не велась уже в течение нескольких лет.

Таблица 1. Сравнение инструментов статической верификации.

Инструмент/подход	Указатели	Структуры	Массивы	Рек. структуры данных	Адресная арифметика	Масштабир.
Ограничиваемая проверка моделей (BMC)	+	+	+	± (конечная глубина)	+	-
BLAST 2.5, алиасы с замыканием	± (конечная глубина)	± (конечная глубина)	-	-	-	+
BLAST 2.7, алиасы с бесконечным замыканием	+	+	-	± (конечная глубина)	-	+
BLAST, ленивый анализ рекурсивных структур данных	+	+	-	+	-	?
CPAchecker, модель памяти на основе неинтерпретируемых функций	+	+	± (огранич. размер)	± (конечная глубина)	+	+

Третий метод, который предложен в данной работе, основан на использовании теории неинтерпретируемых функций без кванторов. По определению неинтерпретируемая функция — это функция, для которой задано только имя и количество аргументов. Соответственно, для одних и тех же значений аргументов функция выдает один и тот же результат. Неинтерпретируемые функции используются для представления состояния памяти программы как отображения некоторых условных адресов переменных в памяти в их значения. В зависимости от конкретной реализации данный подход позволяет достигать различной точности анализа выражений с указателями, при этом использование теории неинтерпретируемых функций дает возможность его применения только для объектов с наперед заданными размерами. Данный подход обладает ограничениями по сравнению с анализом рекурсивных структур данных для анализа динамических структур данных, так как в нем отсутствуют средства сворачивания длинных последовательностей элементов, однако для структур данных относительно небольшого фиксированного размера его использование может быть вполне оправдано.

2. Обзор метода

Для реализации метода был выбран инструмент верификации CPAchecker, основанный на подходе CEGAR с использованием булевой предикатной абстракции и интерполяции Крейга для получения новых предикатов при уточнении абстракции. Для решения задач проверки выполнимости формулы пути и интерполяции Крейга в CPAchecker используются интерполирующие решатели MATHSAT 5, SMTINTERPOL, Z3, PRINCESS. Все эти решатели поддерживают бескванторные теории вещественной или целочисленной линейной арифметики и равенства с неинтерпретируемыми функциями, но не поддерживают (либо не полностью поддерживают, как MATHSAT 5) теорию массивов. Поэтому в данной работе разрабатывался и исследовался способ построения формулы пути с использованием только неинтерпретируемых функций.

В разработанном подходе состояние памяти программы представляется в виде неинтерпретируемой функции *f*, отображающей некоторые условные адреса переменных в памяти в их значения. Для неинтерпретируемых функций выполнена аксиома конгруэнтного замыкания отношения эквивалентности $\forall a. \forall b. ((a = b) \rightarrow f(a) = f(b))$. Эта аксиома моделирует равенство значений, полученных после разыменования равных указателей при одном и том же состоянии памяти программы. То есть если есть два равных указателя *p1* и *p2*, то значения **p1* и **p2* также равны.

При записи значения по какому-либо адресу (**e = expr*) происходит смена версии неинтерпретируемой функции, представляющей состояние памяти. При этом в получаемую логическую формулу пути необходимо явно добавлять равенства между значениями соседних версий неинтерпретируемых функций для не участвовавших в присваивании адресов. Поскольку адрес в таком присваивании часто может вычисляться динамически и быть в общем случае неизвестен в точке присваивания, данные равенства в логической формуле будут представлены в виде дизъюнкций следующего вида:

$$e = a \vee f_i(a) = f_{i-1}(a),$$

где e — адресное выражение, a — адрес, для которого выписывается равенство, f_{i-1} и f_i — соответственно старая и новая версия неинтерпретируемой функции, обновляемой в результате присваивания.

Для представления адресов в памяти предлагается использовать суммы вида $b + n$, где b — переменная (неинтерпретируемая константа), соответствующая адресу некоторой сущности (переменной, структуры, объединения или массива) верхнего уровня, называемая базовым адресом, n — смещение рассматриваемой переменной относительно сущности верхнего уровня. При таком представлении для адресов сущностей верхнего уровня необходимо выполнение условий положительности и непересечения внутренних адресов. Эти условия предлагается записывать в виде двух аксиом модели памяти:

$$b>0 \qquad (A1)$$

$$B(b+n) = k, \qquad (A2)$$

где b — переменная, представляющая базовый адрес сущности, k — целое число, уникальное для каждой такой переменной, n — смещение относительно начала сущности, принимающее значения от 0 до $s-1$ включительно, где s — размер сущности.

Экземпляры (A2) позволяют задать всевозможные попарные неравенства внутренних адресов. Докажем, что из $B(b_1 + n_1) = k_1, B(b_2 + n_2) = k_2$ и $k_1 \neq k_2$ следует, что $b_1 + n_1 \neq b_2 + n_2$. Пусть $b_1 + n_1 = b_2 + n_2$, тогда из (A2) получаем $B(b_1 + n_1) = B(b_2 + n_2)$, $k_1 = k_2$ – противоречие с $k_1 \neq k_2$.

2.1 Полнота предлагаемого метода

Одним из важнейших требований, предъявляемых к подходу уточнения и построения абстракции в инструменте, основанном на использовании метода CEGAR, является требование полноты этого подхода, или иначе говоря, требование надежного уточнения абстракции. Оно означает, что на каждой следующей итерации метода CEGAR, после уточнения абстракции на основе какого-либо ложного контрпримера, этот контрпример обязательно оказывается исключенным из уточненной абстракции. Таким образом достигается постоянное уточнение абстракции с точки зрения уменьшения числа порождаемых ею ложных контрпримеров. Для выполнения данного требования в той реализации CEGAR, которая используется инструментом CPAchecker, достаточно выполнения требования индуктивности получаемых интерполянтов и независимого построения различных частей формулы пути.

Независимость построения различных частей формулы пути означает, что каждой из этих частей будет соответствовать одна и та же логическая формула вне зависимости от контекста ее построения (при проверке контрпримера, интерполяции или пересчете абстракции). Это требование могло бы быть нарушено, например, при использовании эвристик, учитывающих наличие либо отсутствие в контрпримере каких-либо неиспользуемых переменных. В предлагаемом подходе такие эвристики не используются. В случае же выполнения обоих требований полноту подхода можно показать, воспользовавшись определением интерполянта. Предлагаемый подход удовлетворяет этим требованиям, так как предполагает последовательное получение индуктивных интерполянтов и для одинаковых блоков операторов строит формулы пути, совпадающие с точностью до имен индексированных символов.

3. Определения

3.1 Программы и поток управления

Мы ограничимся рассмотрением простого императивного языка программирования (аналогично [25, 26]), в котором все переменные имеют типы *int* и *int**, а все операции — это либо присваивания, либо предположения *assume*, представленные в Таблице 2. Мы рассмотрим программы без вызовов функций, хотя описанный подход может быть расширен на программы с несколькими функциями[2].

Программа представляется *автоматом потока управления* (АПУ) (от англ. control-flow automaton, CFA). АПУ *A=(L,G)* состоит из множества точек программы L, моделирующих счетчик команд, и множества дуг потока управления $G \subseteq L \times Ops \times L$, которые моделируют действия, выполняемые при переходе из одной точки программы в другую. Множество переменных, встречающихся в операциях *Ops*, обозначим как *X*. Программа $P = (A, l_0, l_E)$ состоит из АПУ *A=(L,G)* (моделирующего поток управления программы), начальной точки программы $l_0 \in L$ (моделирует точку входа), и целевой точки программы l_E (моделирует ошибочное состояние).

Конкретное состояние данных программы – это состояние памяти программы, как динамической выделяемой, так памяти под переменные *X*, выделенные на стеке и в статической памяти. Далее мы не будем разделять разные виды пямяти, а будем считать, что для каждой переменной из множества X выделена память, адрес которой обозначается именем переменной. Состояние памяти задается функцией $f : Z \to Z$, отображающей адрес ячейки памяти в значение, содержащееся в ней. Так как переменные $x \in X$ обозначают адреса ячеек памяти, то значения для переменной *x* будут представляться как *f(x)*. Обозначим множество конкретных состояний программы как $\mathcal{E}$.

Множества $r \subseteq \mathcal{E}$ назовем регионами, которые будем представлять с помощью формул алгебры логики φ. Формулы будут содержать переменные из множества *X*, а также неинтерпретируемые функции из множества *F*, заданные над целыми числами ($Z \to Z$), которые служат для моделирования состояния памяти. Формула φ представляет множество $[[\varphi]]$ конкретных состояний данных *c*, для которых выполнено φ (т. е. $[[\varphi]] = \{c \in \mathcal{E} \mid c \vDash \varphi\}$).

Конкретное состояние программы это пара *(l,c)*, где $l \in L$ точка программы, а *c* — это конкретное состояние данных. Пара *(l, φ)* представляет множество $\{(l, c) \mid c \vDash \varphi\}$ конкретных состояний. Конкретная семантика операции $op \in Ops$ определяется оператором сильнейшего постусловия $SP_{op}(\cdot)$: для

[2] Реализация в инструменте CPAchecker работает с программами на языке C и поддерживает вызовы функций

формулы φ оператор $SP_{op}(\varphi)$ представляет наименьшее по включению множество состояний, содержащее все состояния, получаемые хотя бы из одного состояния региона, представленного φ, после выполнения оператора op.

Путь σ — это последовательность $\langle(op_1, l_1), \ldots, (op_n, l_n)\rangle$ пар из операции и точки программы. Путь σ называется *путем программы*, если он начинается в l_0 (см. определение P) и для каждого i, такого что $0 < i \leq n$, существует дуга АПУ $g = (l_{i-1}, op_i, l_i)$. Таким образом, σ представляет синтаксический путь в АПУ.

Конкретная семантика для пути программы σ = $\langle(op_1, l_1), \ldots, (op_n, l_n)\rangle$ определяется как последовательное применение оператора сильнейшего постусловия для каждой операции: $SP_\sigma(\varphi) = SP_{op_n}(\ldots SP_{op_1}(\varphi)\ldots)$. Формула $SP_\sigma(\varphi)$ называется *формулой пути*.

Множество конкретных состояний, являющихся результатом выполнения пути программы σ представляется парой $(l_n, SP_\sigma(true))$. Путь программы называется *достижимым*, если формула $SP_\sigma(true)$ выполнима. Конкретное состояние программы (l_n, c_n) называется *достижимым*, если существует достижимый путь σ, заканчивающийся в точке l_n, и такой, что $c_n \vDash SP_\sigma(true)$. Точка программы l достижима, если существует конкретное состояние c, такое что *(l, c)* достижимо. Программа корректна (safe), если l_E недостижимо.

3.2 Булевы предикатные абстракции

Пусть F — множество неитерпретируемых функций. Пусть $\wp$ — множество предикатов из теории без кванторов над переменными программы X и неинтерпретируемыми функциями F. Формула φ — это булева комбинация предикатов из $\wp$. *Точность для формул* — это конечное подмножество $\pi \in \wp$. *Точность для программы* — это функция $\Pi : L \to 2^{\wp}$, которая задает точность для формул в каждой точке программы.

Булева предикатная абстракция $(\varphi)^\pi$ для формулы φ — это сильнейшая булева комбинация предикатов из точности π, которая следует из φ. Данная предикатная абстракция формулы φ, которая представляет регион конкретных состояний программы, используется как *абстрактное состояние данных* (абстрактное представление региона) в верификации программ. Для формулы φ и точности π булева предикатная абстракция $(\varphi)^\pi$ может быть вычислена с помощью запросов к SMT решателю с поддержкой ALL-SAT следующим образом. Каждому предикату $p_i \in \pi$ сопоставим булеву переменную v_i. Затем сделаем запрос к решателю для выдачи всех векторов решений $v_1, \ldots, v_{|\pi|}$ формулы $\varphi \wedge_{p_i \in \pi}(p_i \Leftrightarrow v_i)$. Для каждого вектора решений мы строим конъюнкцию всех предикатов из π, которые входят в вектор решения

как истина. Дизъюнкция всех таких конъюнкций будет булевой предикатной абстракцией для формулы φ.

Абстрактный оператор сильнейшего постусловия с точностью π и операцией *op*, который преобразует абстрактное состояние φ в следующее состояние φ', может быть определен с помощью применения оператора сильнейшего постусловия и последующего вычисления предикатной абстракции, т. е. $\varphi' = \left(SP_{op}(\varphi)\right)^{\pi}$. Более детально предикатные абстракции описаны в работах [27, 28, 7].

3.3 Кодирование с настраиваемым размером блока

В кодировании с настраиваемым размером блока (Adjustable-Block Encoding, ABE) предикатные абстракции не вычисляются при каждом переходе по дуге из АПУ, а напротив, вычисляются только в некоторых абстрактных состояниях, которые будем называть *состояниями абстракции* (другие абстрактные состояния будем называть состояниями *без абстракции*). На пути между двумя состояниями вычисления абстракции сильнейшее постусловие пути хранится во втором компоненте состояния, который мы назовем *дизъюнктивной формулой пути*. Таким образом, абстрактное состояние ABE содержит два компонента-формулы $\langle \psi, \varphi \rangle$, где формула абстракции ψ – это результат вычисления абстракции, а дизъюнктивная формула φ представляет сильнейшее постусловие с момента вычисления последнего состояния абстракции. Для заданной дуги АПУ $g = (l, op, l')$ и абстрактного состояния $\langle \psi, \varphi \rangle$, следующее состояние либо только расширяет формулу пути φ, либо вычисляет новую формулу абстракции ψ и сбрасывает φ. Точки вычисления абстракции (и, таким образом, размер блока) определяется так называемым оператором настройки блока *blk*. Если оператор $blk(e, g)$ возвращает *false* (нет вычисления абстракции, т. е. абстрактное состояние e без абстракции), то следующее состояние $\langle \psi', \varphi' \rangle$ содержит $\psi' = \psi$ (неизмененное) и $\varphi' = SP_{op}(\varphi)$. Если оператор $blk(e, g)$ возвращает *true* (e — состояние абстракции), то следующее состояние $\langle \psi', \varphi' \rangle$ содержит результат вычисления абстракции по формуле $\psi \wedge \varphi$, $\psi' = \left(SP_{op}(\varphi \wedge \psi)\right)^{\pi}$ и $\varphi' = true$ как новую дизъюнктивную формулу. Если $\varphi \wedge \psi$ невыполнимо для состояния e, то e недостижимо.

4. Модель памяти на основе неинтерпретируемых функций

Для моделирования памяти будем использовать неинтерпретируемые функции f (f_m) и B. Имя неинтерпретируемой функции f_m определим как конкатенацию имени f и индекса m: *f#m*. Над формулами ψ для

неинтерпретируемых функций f и f_m определим операцию подмены всех таких функций, содержащихся в формуле ψ на новое имя $f_{m'}$: $\psi[f_{m'}]$ — заменяет все вхождения f и f_m на $f_{m'}$.

Для задания модели памяти нам потребуются вспомогательные компоненты, которые будут храниться в абстрактном состоянии и изменяться при переходе к следующему состоянию. Во-первых, $Alloc$ — множество пар $(A, n) \in \mathcal{A} \times int$, где константа $A \in \mathcal{A}$ — представляет базовый адрес выделенной памяти, а число n — смещение относительно базового адреса. Во-вторых, m — индекс функции памяти f. В-третьих, k — последний индекс базовых адресов.

В модели памяти оператор сильнейшего постусловия задается как $SP_{op}(\varphi) = \varphi \wedge \Gamma(op)$, где $\Gamma(op)$ определяется Таблицей 2.

Будем использовать вспомогательную функцию:

- $mem\_update(p, m', m, addrs) = \bigwedge_{(a,i)\in addrs}\Big((p = a + i) \vee \big(f_{m'}(a + i) = f_m(a + i)\big)\Big)$,

где e — это выражение без побочных эффектов и без разыменований указателей.

В выражениях **(s1 + i)*, i имеет тип int.

Размер целого и указателей принимается равным 1 байту.

Иначе нужно в выражении **(s + i)* в i указывать размер как количество элементов, помноженное на соответствующий размер элемента.

4.1 Расширение подхода для структур

Подход к построению ограничений Г может быть легко расширен на случай использования в программе структурных типов. Основная идея состоит в том, что структура рассматривается как участок памяти размера, равного сумме размеров полей. Обращения к полям транслируются как сумма указателя на начало структуры и смещения поля относительно начала. Обозначим как $\omega(A, f)$ смещение поля с именем f в структуре A.

1. При выделении памяти *alloc(size)* для структуры A, размер *size* определяется по размеру структуры — сумма размеров полей, и, возможно, с учетом выравнивания.

Таблица 2. Правила построения ограничений Γ

Операция (op)	*Индекс функции памяти m'*	*Множество адресных переменных Alloc'*	*Индекс базового адреса k'*	*Ограничения Γ*
Выделение переменной на стеке int s; или int *s;	Не меняется	*A'* — новое имя переменной. $Alloc' = Alloc \cup \{(A', 0)\}$	*k'* — новый индекс	*s = A'* ∧ *A' > 0* ∧ *B(A')=k'*
Выделение памяти размера size в куче s = alloc(size)	*m'* — новый индекс	*A'* — новое имя переменной. *Alloc' = Alloc ∪ {(A',0)...(A',size-1)}*	*k'* — новый индекс	$f_{m'}(s) = A'$ ∧ *A' > 0* ∧ *B(A'+i)=k'*, где *i* от *0* до *size-1* ∧ *mem_update(s, m', m, Alloc)*
s = e	*m'* — новый индекс	Не меняется	Не меняется	$f_{m'}(s) = \Gamma(e)$ ∧ *mem_update(s, m', m, Alloc)*, где Γ(e) для выражения e вычисляется по следующим правилам: $\Gamma(const)$: $const$ $\Gamma(s)$: $f_m(s)$ $\Gamma(s1\ op\ s2)$, $op \in \{'+', '-', '*', '/'\}$: $f_m(s1)\ op\ f_m\ (s2)$
*(s1 + i) = s2	*m'* — новый индекс	Не меняется	Не меняется	$f_{m'}(f_m(s1) + f_m(i)) = f_m(s2)$ $\wedge\ mem\_update(f_m(s1) + f_m(i), m', m, Alloc)$
s1 = *(s2 + i)	*m'* — новый индекс	Не меняется	Не меняется	$f_{m'}(s1) = f_m(f_m(s2) + f_m(i))$ $\wedge\ mem\_update(s1, m', m, Alloc)$
assume p	Не меняется	Не меняется	Не меняется	Γ(p) для предиката p вычисляется по следующим правилам: $\Gamma(const)$: $const$ $\Gamma(s)$: $f_m(s)$ $\Gamma(p1 == p2)$: $\Gamma(p1) = \Gamma(p2)$ $\Gamma(p1 < p2)$: $\Gamma(p1) < \Gamma(p2)$ $\Gamma(p1 <= p2)$: $\Gamma(p1) \leq \Gamma(p2)$ $\Gamma(p1 \,||\, p2)$: $\Gamma(p1) \vee \Gamma(p2)$ $\Gamma(p1 \,\&\&\, p2)$: $\Gamma(p1) \wedge \Gamma(p2)$ $\Gamma(!\,p)$: $\neg\Gamma(p)$

2. Выражение доступа к полю $s->f$ рассматривается как $*(s + \omega(A, f))$.
3. В присваивании структур по значению *s1 = s2* необходимо выписывать ограничения для присваивания всех полей структуры.

4.2 Пример построения формулы пути

В примере программы используется макрос *container_of(p, type, field_name)*, который раскрывается как $(type)(p + (0 - \omega(type, field\_name)))$. То есть для указателя на вложенное поле структуры *field_name*, мы получаем указатель на структуру, содержащую это поле. Макрос *container_of* часто используется в коде ядра ОС Linux и составляет большую сложность для инструментов, не поддерживающих адресную арифметику. Из-за этого возникают как ложные срабатывания, так и упущенные ошибки.

Пусть задана следующая программа:

```
struct B { int a; int b; };

struct B *p; struct B *q; int *x;

p =alloc(sizeof(B));

p->b = 1;

x = &(p->a);

q = container_of(x, struct B, a);

assume(p->b != q->b);
```

Программа содержит единственный путь, являющийся недостижимым, так как *p->b == q->b* и условие в последнем *assume* не выполнено. В данном примере при моделировании памяти требуется учитывать семантику арифметики указателей, иначе возможно ложное срабатывание, из-за того что путь может быть признан достижимым.

Для данного пути предварительно мы заменим операции со структурами на операции с указателями и построим для него формулу пути.

Таблица 3. Пример построения формулы пути

Путь	Построение формулы пути			
	m'	Alloc'	k'	Γ
struct B *p;	*0*	*{(A0,0)}*	*1*	*p = A0* *∧ A0 > 0 ∧ B(A0) = 1*
struct B *q;	*0*	*{(A0,0), (A1,0)}*	*2*	*q = A1* *∧ A1 > 0 ∧ B(A1) = 2*

int *x;	*0*	*{(A0,0), (A1,0), (A2,0)}*	*3*	$x = A2$ $\wedge\ A2 > 0 \wedge B(A2) = 3$
p =alloc(sizeof(struct B));	*1*	*{(A0,0), (A1,0), (A2,0), (A3,0),...,(A3, sizeof(struct B)-1)}*	*4*	$f_1(p) = A3$ $\wedge\ A3>0$ $\wedge\ B(A3+i) = 4, 0 \le i < sizeof(struct\ B)$ $\wedge\ ((p = A_i + o_i) \vee (f_1(A_i + o_i) = f_0(A_i + o_i)))$, *где* $(A_i, o_i) \in Alloc$
*(p + $\omega(B,b)$) = 1;	*2*	*{(A0,0), (A1,0), (A2,0), (A3,0),...,(A3, sizeof(struct B)-1)}*	*4*	$f_2(f_1(p) + \omega(B,b)) = 1$ $\wedge$ $((f_1(p) + \omega(B,b) = A_i + o_i) \vee (f_2(A_i + o_i) = f_1(A_i + o_i)))$, *где* $(A_i, o_i) \in Alloc$
x = p + $\omega(B,a)$;	*3*	*{(A0,0), (A1,0), (A2,0), (A3,0),...,(A3, sizeof(struct B)-1)}*	*4*	$f_3(x) = f_2(p) + \omega(B,a)$ $\wedge\ ((x = A_i + o_i) \vee (f_3(A_i + o_i) = f_2(A_i + o_i)))$, *где* $(A_i, o_i) \in Alloc$
q = x + (0 - $\omega(B,a)$);	*4*	*{(A0,0), (A1,0), (A2,0), (A3,0),...,(A3, sizeof(struct B)-1)}*	*4*	$f_4(q) = f_3(x) - \omega(B,a)$ $\wedge\ ((q = A_i + o_i) \vee (f_4(A_i + o_i) = f_3(A_i + o_i)))$, *где* $(A_i, o_i) \in Alloc$
assume *(p + $\omega(B,b)$) != *(q + $\omega(B,b)$)	*4*	*{(A0,0), (A1,0), (A2,0), (A3,0),...,(A3, sizeof(struct B)-1)}*	*4*	$\neg(f_4(f_4(p) + \omega(B,b)) = f_4(f_4(q) + \omega(B,b)))$

Таким образом, мы получим формулу пути $SP_\sigma(true)$, являющуюся конъюнкцией формул в столбце Г. Эта формула является невыполнимой, что подтверждает недостижимость данного пути, что и требовалось показать в приведенном примере.

4.3 Конфигурируемый анализ (CPA) с моделью памяти на основе неинтерпретируемых функций

Формализуем анализ с моделью памяти на основе неинтерпретируемых функций в виде *конфигурируемого анализа* (Configurable program analysis, CPA) [29]. Это позволяет использовать гибкость операторов CPA для описания анализа без изменения основного алгоритма (см. Алгоритм CPA на Рис. 1).

Конфигурируемый анализ $\mathbb{D} = (D, \rightsquigarrow, merge, stop)$ состоит из абстрактного домена D, отношения перехода $\rightsquigarrow$, оператора *merge*, оператора *stop*, определяемых следующим образом.

Пусть задана программа $P = (A, l_0, l_E)$, где X — обозначает множество переменных используемых в программе P, F — множество неинтерпретируемых функций, используемых для моделирования памяти, $\wp$ — множество предикатов без кванторов над переменными X и функциями F, и $\Pi: L \to 2^{\wp}$ — точность предикатной абстракции.

1. **Абстрактный домен** $D = (C, R, [[\cdot]])$ — это тройка, состоящая из множества конкретных состояний C, полурешетки $R = (E, \top, \sqsubseteq, \sqcup)$ и функции конкретизации $[[\cdot]] : E \to C$.

Элементы решетки также называются абстрактными состояниями и являются семерками $(l, \psi, l^{\psi}, \varphi, Alloc, k, m)$, где первые четыре компонента являются стандартными компонентами анализа АВЕ, $l, l^{\psi} \in (L \cup \{l_{\top}\})$; $\psi, \varphi \in \wp$. Компонент l моделирует счетчик команд, формула абстракции ψ — булева комбинация предикатов, заданных в Π, l^{ψ} — точка в программе, в которой была вычислена абстракция ψ, а φ — это дизъюнктивная формула, представляющая некоторые или все пути из точки l^{ψ} в l. Заметим, что в состоянии абстракции всегда $l = l^{\psi}$ и $\varphi = true$.

Для уточненной модели памяти мы ввели три новых компонента. Во-первых, $Alloc$ — множество пар $(A, n) \in \mathcal{A} \times int$, где $A \in \mathcal{A}$ — представляет базовый адрес выделенной памяти, а число n — смещение относительно базового адреса. Во-вторых, m — индекс функции памяти f. В-третьих, k — последний индекс базовых адресов.

Верхний элемент решетки — это абстрактное состояние $\top = (l_{\top}, true, l_{\top}, true, \{\}, 0, 0)$. Частичный порядок $\sqsubseteq \subseteq E\ x\ E$ определяется так, что для любых двух состояний $e_1 = (l_1, \psi_1, l^{\psi_1}, \varphi_1, Alloc_1, k_1, m_1)$ и $e_2 = (l_2, \psi_2, l^{\psi_2}, \varphi_2, Alloc_2, k_2, m_2)$ из E выполнено:

$$e_1 \sqsubseteq e_2 \equiv (e_2 = \top) \vee ((l_1 = l_2) \wedge (\varphi_1 = \varphi_2 = true) \wedge (\psi_1[f] \Rightarrow \psi_2[f]))$$

Заметим, что мы подменяем все вхождения версий памяти в формулах абстракции $\psi_1[f]$ и $\psi_2[f]$ на одинаковую версию f.

Оператор соединения $\sqcup: E\ x\ E \to E$ выдает наименьшую верхнюю грань двух операндов в соответствии с частичным порядком $\sqsubseteq$.

Алгоритм CPA($\mathbb{D}, e_0$) (взят из работы [29])
Вход: CPA $\mathbb{D} = (D, \rightsquigarrow, merge, stop)$, начальное состояние $e_0 \in E$, где E обозначает множество элементов решетки D
Выход: множество достижимых абстрактных состояний
Переменные: множество $reached$ достигнутых элементов из E, множество $waitlist$ элементов из E
1: $waitlist$:= $\{e_0\}$
2: $reached$:= $\{e_0\}$
3: **пока** $waitlist \neq 0$ **делать**
4: выбрать элемент e из $waitlist$

```
5:  waitlist := waitlist \ {e}
6:  для каждого e' такого что e ⇝ e' делать
7:    для каждого e'' ∈ reached делать
8:      //слияние с существующими абстрактными состояниями
9:      e_new := merge(e', e'')
10:     если e_new ≠ e'' то
11:       waitlist := (waitlist ∪ {e_new}) \ {e''}
12:       reached := (reached ∪ {e_new}) \ {e''}
13:   если ¬stop(e', reached) то
14:     waitlist := waitlist ∪ {e'}
15:     reached := reached ∪ {e'}
16: вернуть reached
```

Рис. 1. Алгоритм конфигурируемого анализа (CPA)

2. **Отношение перехода** $\rightsquigarrow \subseteq E \times G \times E$ содержит все дуги (e, g, e'), где $e = (l, \psi, l^{\psi}, \varphi, Alloc, k, m)$, $e' = (l', \psi', l^{\psi'}, \varphi', Alloc', k', m')$ и $g = (l, op, l')$ для которых выполнено:

$$\begin{cases} (\varphi' = \text{true}) \wedge \left(\psi' = \left(\mathrm{SP}_{\mathrm{op}}(\varphi \wedge \psi)\right)^{\Pi(l')}[\mathrm{f}_{\mathrm{m}'}]\right) \wedge (l^{\psi'} = l'), \text{если } \mathrm{blk}(e, g) \vee (l' = \mathrm{l}_{\mathrm{E}}) \\ (\varphi' = \mathrm{SP}_{\mathrm{op}}(\varphi)) \wedge (\psi' = \psi) \wedge (l^{\psi'} = l^{\psi}) \text{ иначе} \end{cases}$$

В представленной модели памяти оператор сильнейшего постусловия задается как $SP_{op}(\varphi) = \varphi \wedge \Gamma(op)$, где $\Gamma(op)$ определяется таблицей 2. При этом значения $Alloc', k', m'$ в следующем состоянии определяются также по таблице 2.

Таким образом, мы имеем отношения перехода, которое работает в двух режимах, определяемых оператором $blk: E \times G \to B$, который отображает абстрактное состояние e и дугу g АПУ в *true* или *false*. Оператор *blk* задается как параметр анализу. В первом режиме строится абстракция, а во втором вычисляется только сильнейшее постусловие.

3. **Оператор слияния** $merge: E \times E \to E$ для двух абстрактных состояний $e_1 = (l_1, \psi_1, l^{\psi_1}, \varphi_1, Alloc_1, k_1, m_1)$ и $e_2 = (l_2, \psi_2, l^{\psi_2}, \varphi_2, Alloc_2, k_2, m_2)$ определяется следующим образом: $merge(e_1, e_2) =$

$$\begin{cases} \text{если } (\mathrm{l}_1 = \mathrm{l}_2) \wedge (\psi_1[\mathrm{f}] = \psi_2[\mathrm{f}]) \wedge (l^{\psi_1} = l^{\psi_2}), \text{то } (\mathrm{l}_2, \psi_2, l^{\psi_2}, (\varphi_1 \wedge \mathrm{mem\_unchanged}(\mathrm{m}', \mathrm{m}_1, \mathrm{Alloc}_1)) \\ \quad \vee (\varphi_2 \wedge \mathrm{mem\_unchanged}(\mathrm{m}', \mathrm{m}_2, \mathrm{Alloc}_2)), \mathrm{Alloc}_1 \cup \mathrm{Alloc}_2, \max(\mathrm{k}_1, \mathrm{k}_2), \mathrm{m}') \\ \quad e_2, \text{иначе} \end{cases}$$

где m' — новый индекс функции памяти, а функция *mem_unchanged* определяется как:

- $mem\_unchanged(m', m, addrs) = \wedge_{(a,i) \in addrs}(f_{m'}(a + i) = f_m(a + i))$

Таким образом, множества адресных переменных объединяются, а при построении новой дизъюнктивной формулы вводится новый индекс функции памяти, к которой приравниваются функции в каждом из состояний.

4. **Оператор останова** $stop\colon E \times 2^E \to B$ проверяет, покрывается ли состояние e другим состоянием из пройденных состояний R (множество *reached*):

$$\forall e \in E, R \subseteq E\colon stop(e,R) \;=\; \exists e' \in R : (e \sqsubseteq e')$$

5. Оптимизации

Эффективность модели памяти сильно зависит от количества дизъюнкций, выписываемых в *mem_update* из Таблицы 2. В предлагаемом подходе используются следующие оптимизации для сокращения числа этих дизъюнкций:

1. Разделение области памяти по типам, так что каждая неинтерпретируемая функция задает отображение адресов переменных в их значения для одного соответствующего ей примитивного типа данных. Например, вводятся функции *f_long_int, f_char*, f_struct_B**. Примитивными, то есть не составными типами данных в данном подходе, считаются символьный и целочисленный типы, а также любой тип указателя. Таким образом, неявно предполагается, что значение, записанное по какому-либо адресу в качестве значения какого-то типа данных не может быть впоследствии считано по этому же адресу как значение другого типа. Это предположение является одним из ограничений подхода.
2. Выделение среди множества всех переменных программы подмножества «чистых» переменных, к которым возможен доступ лишь по именам (то есть переменных, не имеющих алиасов). Для таких переменных нет необходимости в использовании неинтерпретируемых функций для представления значений. Поэтому для них используется сходное с используемым инструментами BLAST и CPAchecker SSA-представление (для них не выписывается разыменование и имя переменной представляет значение переменной, а не ее адрес). Определение «чистых» переменных возможно проводить не для всей программы, а на заданном пути, до тех пор пока не встретиться взятие адреса для этой переменной на этом пути или на другом пути при выполнении оператора *merge*.
3. Использование эвристики для полей структур. В предлагаемом подходе предполагается, что указатель на поле структуры, получаемый сложением адреса структуры с соответствующим смещением поля, может принимать только значения адресов того же самого поля в других структурах того же самого типа, что и структура, которой это поле принадлежит. Такой указатель не может,

в частности, быть равным адресу элемента массива или отдельной переменной, не являющейся полем структуры. Например, обновление поля *skb1->next* не может повлиять ни на какой *skb2->prev*, даже если *next* и *prev* одного типа. В этом случае в качестве оптимизации мы опускаем посыл импликации в *mem_update*, если смещения заранее не равны. Такое предположение также является одним из ограничений метода.

4. Инициализация константами. Оптимизация заключается в том, что мы выписываем одно обновление mem_update для нескольких присваиваний. Например, при выделении памяти, заполненной нулями под структуру *kzalloc(sizeof(*info), GFP_KERNEL)*, *mem_update* выписывается только один раз после инициализации всех полей нулями.
5. Сокращение множества *Alloc*, за счет сохранения смещений только для тех полей, которые были использованы в пути. Соответственно для неиспользованных полей не будут выписываться дизъюнкции в *mem_update*.

6. Результаты

Предложенный метод был реализован в инструменте CPAchecker версии 1.4 (для экспериментов была взята ревизия *18237*). За включение разработанной модели памяти на основе неинтерпретируемых функций отвечает опция *cpa.predicate.handlePointerAliasing*. Эксперименты проводились на наборах международных соревнований по верификации программ SV-COMP'2016 (http://sv-comp.sosy-lab.org/2016).

В первую очередь была рассмотрена категория по работе со структурами данных в куче *Heap Data Structures*. Задачи в этой категории требуют поддержки анализа указателей.

Запуски проводились в двух конфигурациях предикатного анализа *predicateAnalysis* с поддержкой модели памяти на основе неинтерпретируемых функций (с НФ) и без нее (без НФ). Как можно видеть в Таблице 4, на представленном наборе анализ с использованием предложенного метода не дает некорректных результатов на представленном наборе по сравнению с не использующей его версией, которая давала 24 некорректных результата.

Отметим, что вердикт *unknown* возникает, когда верификатор не может найти ошибку или доказать ее отсутствие, в силу превышения лимитов по времени, памяти, обнаружению неподдерживаемых конструкций (например, рекурсии) или в силу ограничений самого CEGAR.

Количество вердиктов *unknown* составило 14 шт. для НФ, большая часть из них из-за превышения лимита по времени 10 шт. В результате общее время работы анализа с НФ составило 9514 секунд, что почти в десять раз больше чем без НФ (см. Табл. 5). Однако если рассматривать только время на

корректные результаты (не включающее время на вердикты *unknown*), то время работы оказывается сравнимо.

Таблица 4. Результаты запуска на наборе HeapDataStructures в конфигурации предикатного анализа, с лимитом времени 15 минут и 15 Gb памяти.

Модели памяти	без НФ	с НФ
Общее количество	81	81
Корректные результаты	53	67
Доказано отсутствие ошибки	34	44
Ошибка найдена	19	23
Некорректные результаты	24	0
Упущенная ошибка	5	0
Ложное предупреждение	19	0

Таблица 5. Время работы CPU (в секундах) запуска на наборе HeapDataStructures в конфигурации предикатного анализа, с лимитом времени 15 минут и 15 Gb памяти.

Модели памяти	без НФ	с НФ
Общее время	680	9514
Время для корректных результатов	476	487

В качестве второго набора был выбран *DeviceDriversLinux64*, состоящий из драйверов устройств ядра операционной системы Linux, большая часть которого подготовлена в рамках проекта LDV [30, 31]. Для этого набора была использована конфигурация *ldv*, использующаяся в проекте LDV. Видно, что время работы для этого набора сравнимо как на всех тестах, так и на тестах, для которых получен корректный результат (см. Табл. 7). Причем корректных результатов без НФ получено на 11 шт. больше (см. Табл. 6).

Таблица 8 показывает изменения вердиктов при переходе от запуска без НФ к запуску с НФ. Видно, что модель памяти с НФ теряет 31 корректный результат, из них 29 из-за вердикта *unknown*, одно ложное срабатывание получено из-за того, что для точной работы модели памяти с НФ в тесте не хватает явного выделения памяти, кроме того, еще в одном тесте анализ без НФ находит ложную трассу ошибки, так как считает возможным равенство нулю адреса переменной на стеке. С другой стороны, получается 20 новых корректных результатов, а 5 некорректных результатов становятся *unkown*.

Таблица 6. Результаты запуска на наборе DeviceDriversLinux64 в конфигурации ldv, с лимитом по времени 15 минут и 15 Gb по памяти.

Модели памяти	без НФ	с НФ
Общее количество	2121	2121
Корректные результаты	1654	1643
Доказано отсутствие ошибки	1450	1450
Ошибка найдена	204	193
Некорректные результаты	17	6
Упущенная ошибка	5	3
Ложное предупреждение	12	3

Таблица 7. Время работы CPU (в часах) запуска на наборе DeviceDriversLinux64 в конфигурации ldv, с лимитом по времени 15 минут и 15 Gb по памяти.

Модели памяти	без НФ	с НФ
Общее время	140 часов	144 часа
Время для корректных результатов	22,2 часов	22,7 часов

Таблица 8. Результаты изменения вердиктов на наборе DeviceDriversLinux64 в конфигурации ldv.

Переходы из модели памяти без НФ в модель памяти с НФ	Количество
Корректный результат → Некорректный результат	2
Корректный результат → unknown	29
Некорректный результат → Корректный результат	8
Некорректный результат → unknown	5
Unknown → Корректный результат	12
Unknown → Некорректный результат	0

7. Заключение

Метод моделирования памяти на основе неинтерпретируемых функций позволяет анализировать программы, содержащие выражения с указателями, в том числе указателями на структуры, массивы, и выражения, содержащие адресную арифметику. Ограничениями метода является конечность размера массивов и конечная глубина рекурсии для динамических структур данных. Метод является масштабируемым, так как показывает приемлемые результаты по скорости на практически значимом наборе из драйверов устройств ОС Linux. Вместе с тем, ряд корректных результатов не может быть получен, по причине замедления работы анализа. Это требует дальнейшего развития метода.

Среди возможных направлений отметим, возможность более точного разбиения на регионы памяти, так что для этих регионов выделяются независимые неинтерпретируемые функции. Например, отдельные функции могут быть использованы для каждого поля структуры, для которого не берется адрес. Регионы также могут быть выделены на основе предварительного анализа кода программы. Кроме того, в качестве направления исследований может быть рассмотрено объединение инструкций программы, например, объединение последовательности присваиваний, меняющих независимые участки памяти, и их рассмотрение как одного обновления памяти.

Литература

[1]. Edmund Clarke, Daniel Kroening, Flavio Lerda. A Tool for Checking ANSI-C Programs. In Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, volume 2988, pp. 168-176, 2004.

[2]. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Yunshan Yhu. Symbolic model checking without BDDs. In Tools and Algorithms for Construction and Analysis of Systems, pp. 193–207, 1999.

[3]. Daniel Kroening, Edmund Clarke, Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In Proceedings of DAC 2003, pp. 368–371, ACM Press, 2003.

[4]. F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, P. Ashar. F-Soft: Software Verification Platform. In CAV, LNCS, volume 3576, pp. 301-306, 2005.

[5]. H. Post, C. Sinz, F. Merz, T. Gorges, T. Kropf. Linking Functional Requirements and Software Verification. In 17th IEEE International Requirements Engineering Conference, pp.295-302, 2009.

[6]. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith. Counterexample-Guided Abstraction Refinement. In CAV, LNCS, volume 1855, pp. 154-169, 2000.

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

[8]. S. Graf, H. Saidi. Construction of abstract state graphs with PVS. In CAV, LNCS, volume 1254, pp. 72-83, 1997.

[9]. Ranjit Jhala, Rupak Majumdar. Software model checking. ACM Computing Surveys, volume 41, issue 4, article 21, pp.1-54, 2009.
[10]. William Craig. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. The Journal of Symbolic Logic, vol. 22, no. 3, pp. 269-285, 1957.
[11]. Roger Lyndon. An interpolation theorem in the predicate calculus. Pacific Journal of Mathematics, vol. 9, no. 1, pp. 129-142, 1959.
[12]. Dirk Beyer, Damien Zufferey, Rupak Majumdar. CSIsat: Interpolation for LA+EUF. In CAV, LNCS, volume 5123, pp. 304-308, 2008.
[13]. Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Roberto Sebastiani. The MathSAT 4 SMT Solver. In CAV, LNCS, volume 5123, pp. 299-303, 2008.
[14]. Alberto Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. Journal on Satisfiability, Boolean Modeling and Computation, vol. 8 pp. 1-27, 2012.
[15]. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar. The software model checker BLAST: Applications to software engineering. International Journal on Software Tools for Technology Transfer, volume 9, issue 5, pp. 505-525, 2007.
[16]. Швед П. Е., Мутилин В. С., Мандрыкин М. У. Опыт развития инструмента статической верификации BLAST. Программирование, том 3, стр. 24–35, 2012.
[17]. P. E. Shved, V. S. Mutilin, M. U. Mandrykin. Experience of improving the BLAST static verification tool. Programming and Computer Software, volume 38, issue 3, pp. 134-142, 2012.
[18]. Andersen L. O. Program Analysis and Specialization for the C Programming Language. Københavns Universitet, Datalogisk Institut, DIKU, 1994.
[19]. Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, Navindra Umanee. Points-to analysis using BDDs. In Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, ACM, volume 38, issue 5, pp. 103-114. 2003.
[20]. Pavel Shved, Mikhail Mandrykin, Vadim Mutilin. Predicate Analysis with BLAST 2.7. In Tools and Algorithms for the Construction and Analysis of Systems, LNCS, volume 7214, pp. 525–527, 2012.
[21]. Dirk Beyer. Status Report on Software Verification (Competition Summary SV-COMP 2014). In Tools and Algorithms for the Construction and Analysis of Systems, LNCS, volume 8413, pp. 373-388, 2014.
[22]. K. Dudka, P. Peringer, T. Vojnar. Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In CAV, LNCS, volume 6806, pp. 372-378, 2011.
[23]. Dirk Beyer, Thomas A. Henzinger, Grégory Théoduloz. Lazy Shape Analysis. In CAV, LNCS, volume 4144, pp. 532-546, 2006.
[24]. Tal Lev-Ami, Mooly Sagiv. TVLA: A System for Implementing Static Analyses. In Static Analysis, LNCS, volume 1824, pp. 280-301, 2000.
[25]. Dirk Beyer, M. Erkan Keremoglu, Philipp Wendler. Predicate abstraction with adjustable-block encoding. In Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design, pp. 189-198, 2010.
[26]. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, Kenneth L. McMillan. Abstractions from proofs. In Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, volume 39, issue 1, pp. 232-244, 2004.

[27]. T. Ball, A. Podelski, S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. In Tools and Algorithms for the Construction and Analysis of Systems, LNCS, volume 2031, pp. 268-283, 2006.
[28]. S. K. Lahiri, R. Nieuwenhuis, A. Oliveras. SMT techniques for fast predicate abstraction. In CAV, LNCS, volume 4144, pp. 424-437, 2006.
[29]. Dirk Beyer, Thomas A. Henzinger, Grégory Théoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In CAV, LNCS, 4590, pp. 504-518, 2007.
[30]. M. U. Mandrykin, V. S. Mutilin, E. M. Novikov, A. V. Khoroshilov, P. E. Shved. Using Linux Device Drivers for Static Verification Tools Benchmarking. Programming and Computer Software, volume 38, issue 5, pp. 245-256, 2012.
[31]. Alexey Khoroshilov, Mikhail Mandrykin, Vadim Mutilin, Eugene Novikov, Alexander Petrenko, Ilya Zakharov. Configurable toolset for static verification of operating systems kernel modules. Programming and Computer Software, vol. 41, n.1, pp. 49-64, 2015.

Modeling Memory with Uninterpreted Functions for Predicate Abstractions

M.U. Mandrykin <mandrykin@ispras.ru>
V.S. Mutilin <mutilin@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation

Abstract. One of the key problems in modern static verification methods is a precise model for semantics of expressions containing pointers. The trustworthiness of the verification verdict highly depends on the analysis of these expressions. In the paper, we describe the verification methods with memory models based on uninterpreted functions, allowing to analyze programs containing expressions with pointers, including pointers to structures, arrays and pointer arithmetic. The approach is limited finite array size and finite recursion depth for dynamic data structures. The method was implemented in CPAchecker tool, based on CEGAR with boolean predicate abstractions and Craig interpolation for inferring new predicates used in abstraction refinement. For solving safisfability of path formulas and Craig interpolation CPAchecker uses interpolation solvers supporting theories of linear integer and real inequalities and equalities with uninterpreted functions. In the method, the memory state is represented as uninterpreted function mapping some variable addresses in memory to its values. After each write to memory for a pointer the version of uninterpreted function is changed. The experiments were performed on the benchmarks of International Competition on Software Verification (SV-COMP'2016) containing industrial size benchmarks of device drivers of Linux operating system. On these benchmarks, the method demonstrates reasonable verification times, reducing the number of missed defects and false alarms. Among the future work directions we consider deviding memory into region thus having a separate uninterpreted for each region.

Keywords: memory model, predicate abstraction, counterexample-guided abstraction refinement

DOI: 10.15514/ISPRAS-2015-27(5)-7

For citation: Mandrykin M.U., Mutilin V.S. Modeling Memory with Uninterpreted Functions for Predicate Abstractions. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 117-142 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-7.

References

[1]. Edmund Clarke, Daniel Kroening, Flavio Lerda. A Tool for Checking ANSI-C Programs. In Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, volume 2988, pp. 168-176, 2004.

[2]. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Yunshan Yhu. Symbolic model checking without BDDs. In Tools and Algorithms for Construction and Analysis of Systems, pp. 193–207, 1999.

[3]. Daniel Kroening, Edmund Clarke, Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In Proceedings of DAC 2003, pp. 368–371, ACM Press, 2003.

[4]. F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, P. Ashar. F-Soft: Software Verification Platform. In CAV, LNCS, volume 3576, pp. 301-306, 2005.

[5]. H. Post, C. Sinz, F. Merz, T. Gorges, T. Kropf. Linking Functional Requirements and Software Verification. In 17th IEEE International Requirements Engineering Conference, pp.295-302, 2009.

[6]. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith. Counterexample-Guided Abstraction Refinement. In CAV, LNCS, volume 1855, pp. 154-169, 2000.

[7]. Khoroshilov A.V., Mandrykin M. U., MutilinV. S. Vvedenie v metod CEGAR — utochnenie abstrakcii po kontrprimeram [Introduction to CEGAR — Counter-Example Guided Abstraction Refinement]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 24, pp. 219-292, 2013 (in Russian).

[8]. S. Graf, H. Saidi. Construction of abstract state graphs with PVS. In CAV, LNCS, volume 1254, pp. 72-83, 1997.

[9]. Ranjit Jhala, Rupak Majumdar. Software model checking. ACM Computing Surveys, volume 41, issue 4, article 21, pp.1-54, 2009.

[10]. William Craig. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. The Journal of Symbolic Logic, vol. 22, no. 3, pp. 269-285, 1957.

[11]. Roger Lyndon. An interpolation theorem in the predicate calculus. Pacific Journal of Mathematics, vol. 9, no. 1, pp. 129-142, 1959.

[12]. Dirk Beyer, Damien Zufferey, Rupak Majumdar. CSIsat: Interpolation for LA+EUF. In CAV, LNCS, volume 5123, pp. 304-308, 2008.

[13]. Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Roberto Sebastiani. The MathSAT 4 SMT Solver. In CAV, LNCS, volume 5123, pp. 299-303, 2008.

[14]. Alberto Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. Journal on Satisfiability, Boolean Modeling and Computation, vol. 8 pp. 1-27, 2012.

[15]. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar. The software model checker BLAST: Applications to software engineering. International Journal on Software Tools for Technology Transfer, volume 9, issue 5, pp. 505-525, 2007.

[16]. P. E. Shved, V. S. Mutilin, M. U. Mandrykin. Opyt razvitiya instrumenta staticheskoy verifikatsii BLAST. [Experience of improving the BLAST static verification tool]. Programmirovanie [Programming and Computer Software], vol. 38, issue 3, pp. 25-34, 2012 (in Russian).

[17]. P. E. Shved, V. S. Mutilin, M. U. Mandrykin. Experience of improving the BLAST static verification tool. Programming and Computer Software, volume 38, issue 3, pp. 134-142, 2012.

[18]. Andersen L. O. Program Analysis and Specialization for the C Programming Language. København Universitet, Datalogisk Institut, DIKU, 1994.

[19]. Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, Navindra Umanee. Points-to analysis using BDDs. In Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, ACM, volume 38, issue 5, pp. 103-114. 2003.

[20]. Pavel Shved, Mikhail Mandrykin, Vadim Mutilin. Predicate Analysis with BLAST 2.7. In Tools and Algorithms for the Construction and Analysis of Systems, LNCS, volume 7214, pp. 525–527, 2012.

[21]. Dirk Beyer. Status Report on Software Verification (Competition Summary SV-COMP 2014). In Tools and Algorithms for the Construction and Analysis of Systems, LNCS, volume 8413, pp. 373-388, 2014.

[22]. K. Dudka, P. Peringer, T. Vojnar. Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In CAV, LNCS, volume 6806, pp. 372-378, 2011.

[23]. Dirk Beyer, Thomas A. Henzinger, Grégory Théoduloz. Lazy Shape Analysis. In CAV, LNCS, volume 4144, pp. 532-546, 2006.

[24]. Tal Lev-Ami, Mooly Sagiv. TVLA: A System for Implementing Static Analyses. In Static Analysis, LNCS, volume 1824, pp. 280-301, 2000.

[25]. Dirk Beyer, M. Erkan Keremoglu, Philipp Wendler. Predicate abstraction with adjustable-block encoding. In Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design, pp. 189-198, 2010.

[26]. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, Kenneth L. McMillan. Abstractions from proofs. In Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, volume 39, issue 1, pp. 232-244, 2004.

[27]. T. Ball, A. Podelski, S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. In Tools and Algorithms for the Construction and Analysis of Systems, LNCS, volume 2031, pp. 268-283, 2006.

[28]. S. K. Lahiri, R. Nieuwenhuis, A. Oliveras. SMT techniques for fast predicate abstraction. In CAV, LNCS, volume 4144, pp. 424-437, 2006.

[29]. Dirk Beyer, Thomas A. Henzinger, Grégory Théoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In CAV, LNCS, 4590, pp. 504-518, 2007.

[30]. M. U. Mandrykin, V. S. Mutilin, E. M. Novikov, A. V. Khoroshilov, P. E. Shved. Using Linux Device Drivers for Static Verification Tools Benchmarking. Programming and Computer Software, volume 38, issue 5, pp. 245-256, 2012.

[31]. Alexey Khoroshilov, Mikhail Mandrykin, Vadim Mutilin, Eugene Novikov, Alexander Petrenko, Ilya Zakharov. Configurable toolset for static verification of operating systems kernel modules. Programming and Computer Software, vol. 41, n.1, pp. 49-64, 2015.