Обзор подходов к моделированию памяти в инструментах статической верификации[1]

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

Аннотация. В статье приведен обзор существующих подходов к моделированию памяти Си-программ в инструментах статической верификации. Обозначены основные проблемы, возникающие при разработке моделей памяти для языка Си. В обзоре рассматриваются две основные группы моделей памяти в зависимости от полноты поддержки областей памяти наперед не ограниченного размера. Среди моделей для ограниченных областей памяти рассматриваются модель, использующая результаты предварительного анализа алиасов, и модель на основе слабейших предусловий, использующая теорию неинтерпретируемых функций и логику первого порядка. Среди моделей для областей памяти наперед не ограниченного размера рассматривается типизированная модель, модель Бурсталла-Борната, модель с регионами и полная модель памяти для теории интерпретируемых множеств элементов списков, использованная ранее в инструменте дедуктивной верификации HAVOC.

Ключевые слова: статическая верификация; модели памяти; SMT-решатели.

DOI: 10.15514/ISPRAS-2017-29(1)-12

Для цитирования: Мандрыкин М.У., Мутилин В.С.. Обзор подходов к моделированию памяти в инструментах статической верификации. Труды ИСП РАН, том 29, вып. 1, 2017 г., стр. 195-230. DOI: 10.15514/ISPRAS-2017-29(1)-12

1. Введение

Язык программирования Си продолжает оставаться одним из наиболее широко используемых языков программирования в области системного

[1] Исследования проводились при финансовой поддержке РФФИ в рамках проекта №15-01-03934.

программирования, в частности при написании операционных систем, драйверов, сред окружения и средств поддержки времени выполнения для различных языков программирования, а также при написании других систем, предъявляющих высокие требования к производительности или объему исполняемых программ. При этом одним из основных преимуществ использования языка Си для реализации высокопроизводительных систем является слабая статическая типизация и присутствие широкого набора доступных операций с указателями, позволяющими эффективно управлять использованием памяти, в том числе явно манипулируя адресами размещаемых в ней данных. Ко многим высокопроизводительным системам предъявляются среди прочих требования по высокой надежности, а в некоторых случаях – и по безопасности. Поэтому продолжает оставаться актуальной задача верификации Си-программ.

Эта задача может решаться с применением методов динамической, статико-динамической и статической верификации. В силу того, что каждый из подходов обладает своими преимуществами и ограничениями необходимо вести развитие всех этих направлений. Вместе с тем, только статическая верификация может дать доказательство отсутствия ошибок, по крайней мере, некоторых классов ошибок, или дать доказательство полного соответствия программ заданным формальным спецификациям.

Методы статической верификации ранее показали свою применимость, в частности, для верификации модулей и отдельных подсистем в гипервизорах и ядрах операционных систем, что в свою очередь является важным аргументом, подтверждающим актуальность развития этих методов.

Во многих современных инструментах статической верификации используются SMT-решатели, и, таким образом, семантика языка программирования, на котором написана верифицируемая программа, частично или полностью моделируется с помощью логических формул в соответствующих теориях (SMT-формул, от англ. Satisfiability Modulo Theories) [1]. Подходы к соответствующему эффективному моделированию семантики в виде SMT-формул могут очень существенно различаться друг от друга. При этом даже небольшие изменения в одном методе моделирования семантики могут приводить к изменению, к примеру, времени работы SMT-решателей на результирующих формулах (и, как следствие, времени работы всего инструмента верификации) в несколько десятков раз [2]. В таком контексте разработка соответствующих методов эффективного моделирования семантики используемых языков программирования и спецификации с помощью SMT-формул становится важной и актуальной задачей.

Для языка Си основной проблемой при моделировании семантики в виде логических формул является моделирование семантики операций с указателями, в частности, указателями на динамически выделяемые области

памяти наперед не ограниченного размера, а также моделирование различных приведений типов указателей и случаев использования объединений. Поэтому задача разработки соответствующих методов эффективного и точного моделирования памяти Си-программ для используемых на практике инструментов статической верификации является одной из ключевых.

Помимо развития собственно методов моделирования памяти Си-программ важной задачей является также оценка эффективности этих методов при использовании для верификации реальных промышленных программных систем.

2. Использование решателей логических формул в теориях в инструментах статической верификации

Решатели логических формул в теориях широко используются во всех основных группах инструментов статической верификации:

- проверки моделей программ (software model checking),
- анализа программ,
- дедуктивной верификации.

Среди разновидностей инструментов проверки моделей программ наиболее распространенными являются инструменты, основанные на методах итеративного построения абстракции для программы с помощью уточнения по неосуществимым в исходной программе (фиктивным) контрпримерам (CEGAR, от англ. Counter-Example Guided Abstraction Refinement) [3] и методах ограничиваемой верификации (BMC, англ. Bounded model checking) [4]. Данные инструменты используют SMT-решатели для:

- поиска контрпримеров во всем пространстве состояний модели, представленном в виде логической формулы [4];
- проверки осуществимости трасс выполнения, ведущих к ошибочным состояниям [5];
- проверки k-индуктивности заданного свойства программы [6];
- автоматического построения предикатных абстракций [7, 5];
- автоматического уточнения предикатных абстракций по неосуществимым контрпримерам с помощью построения интерполянтов Крейга [8];
- автоматического поиска индуктивных инвариантов с помощью процедуры индуктивного обобщения в рамках метода проверки моделей программ IC3 [9];

а также других, в том числе вспомогательных, целей.

В инструментах анализа программ (англ. program analysis), которые отличаются от проверки моделей модульностью верификации и применением менее точных и менее ресурсоемких способов абстракции, SMT-решатели используются для [10, 11]:

- абстрактной интерпретации с использованием различных абстрактных доменов, в том числе в предикатной абстракции;
- символьного выполнения с целью поиска входных данных, при выполнении на которых программа достигает заданный целевой оператор;
- дополнительной проверки осуществимости путей выполнения для уменьшения числа ложных предупреждений.

В основе методов дедуктивной верификации лежит логика Хоара [12], методы индуктивных утверждений и оценочных функций Флойда [13], преобразователи предикатов, такие как слабейшее предусловие [14], а также аксиоматическая семантика. С их помощью корректность программы относительно проверяемых свойств сводится к проверке набора формальных математических утверждений — условий верификации (УВ, англ. verification condition, VC), представляющих собой формулы в различных формальных логических системах, таких как пропозициональная или сепарационная [15] логика. В инструментах дедуктивной верификации решатели используются для [16, 17]:

- разрешения (проверки выполнимости) условий верификации;
- поиска контрпримеров для реализаций или спецификаций, не соответствующих друг другу;
- проверки используемых аксиоматических формализаций на непротиворечивость;
- выделения существенно используемых утверждений в контексте условия верификации, например, для поиска причин возникающих противоречий или для упрощения полученного условия верификации с целью повторной проверки с помощью другого решателя.

Кроме того, ведутся исследования в направлении интеграции SMT-решателей в инструменты интерактивного доказательства теорем [18, 19], используемых в подходах к верификации программ, основанных на использовании денотационной семантики.

2.1 Общая схема использования SMT-решателей

Несмотря на существенные различия между разнообразными задачами, для решения которых применяются решатели логических формул в теориях, можно выделить некоторый общий подход, который описывает все

перечисленные случаи их использования в инструментах статической верификации. Этот подход можно условно изобразить в виде схемы, приведенной на рис. 1.

Для генерации запросов к решателю используется некоторое внутреннее представление программы или ее фрагмента, использующее сущности соответствующего входного или промежуточного языка программирования (и, возможно, спецификации), такие как «переменная типа `int`», «указатель на структуру **`struct`** `mutex`», «оператор разыменования `*`», «конструктор `None`» алгебраического типа `Option` (из языка OCaml) и т.д. Так как современные решатели применяются не только для проверок выполнимости логических формул, но также, например, для поиска моделей выполнимых формул, поиска интерполянтов Крейга для невыполнимых формул, упрощения формул и других целей, на процесс генерации запросов к решателю влияют типы решаемых задач. Суть же генерации запроса к решателю, как правило, состоит в преобразовании внутреннего представления программы, использующего сущности языка программирования, в некоторое представление логической формулы в теориях, использующее сущности, определяемые этими теориями, например, «неинтерпретируемая целочисленная константа», «неинтерпретируемая функция», «логический массив», «битовый вектор» и т.д.

Таким образом, имеет смысл ввести условное (неформальное) понятие *модельной семантики языка программирования*, обозначающее способ представления семантики сущностей языка программирования и операций над ними в виде логических формул в теориях. Это понятие с одной стороны отличается от общего понятия *семантики языка программирования*, которое никак не ограничивает используемый способ описания семантики, позволяя выражать её, например, с помощью произвольных определений отношения вычисления [20], произвольных наборов логических утверждений или произвольных денотаций [21]. С другой стороны, оно не ограничивает возможностей использования как произвольного исходного языка программирования, так и произвольного целевого набора логических теорий, что не позволяет определить это понятие достаточно формально. В дальнейшем будем подразумевать, что для используемых в модельной семантике логических теорий существуют автоматические решающие процедуры, возможно, не обладающие полнотой (то есть допускающие незавершение решателя или вердикт «`unknown`» – «неизвестно»). Также для уменьшения возможных неоднозначностей будем в дальнейшем называть исходно заданную в произвольном виде семантику рассматриваемого языка программирования его *исходной семантикой*. Процесс преобразования внутреннего представления программы в запрос к решателю, содержащий логическую формулу в теориях, в соответствии с используемой модельной

семантикой и типом решаемой задачи будем называть *кодированием* (англ. *encoding*) запроса.

Очевидно, используемая модельная семантика существенно зависит от используемых языков программирования и спецификации. В частности, для императивных языков программирования (то есть основанных на операторах, изменяющих состояние программы), допускающих использование синонимичных (англ. aliasing) указателей или ссылок, например, таких как Си, Си++, Java или C#, модельная семантика должна определять способ кодирования операций с указателями или ссылками. Соответствующую часть модельной семантики, содержащую формальные определения понятий указателя или ссылки и формализацию операций над ними с использованием логических формул в теориях, будем называть *моделью памяти*. В данной работе рассматриваются только модели памяти для языка программирования Си в сочетании с различными языками спецификации или другими способами спецификации проверяемых свойств.

Рис. 1. Общая схема использования SMT-решателей в инструментах верификации.
Fig. 1. General diagram illustrating SMT-solver usage in static verification tools.

2.2 Теории, поддерживаемые современными решателями

Набор поддерживаемых решателем теорий в конечном счете определяет набор сущностей, с помощью которых может быть формализована модельная семантика используемого языка программирования, а алгоритмическая разрешимость или сложность решения задач выполнимости, соответствующих используемым комбинациям теорий, определяет вероятность успешного разрешения (с вердиктом «выполнимо»/«невыполнимо») генерируемых логических формул и требуемое для этого количество ресурсов – времени и памяти. Обзор основных теорий, поддерживаемых современными SMT-решателями, дан в диссертации [22].

В данной работе будут описаны методы моделирования семантики Си-программ с использованием комбинаций теории неинтерпретируемых функций с теориями целочисленной и вещественной линейной арифметики с кванторами и без кванторов (то есть с использованием логик QF_UFLRA, UFLRA, UFLIA и QF_UFLIA).

3. Классификация моделей памяти

В дальнейшем для каждого рассматриваемого подхода к моделированию семантики Си-программ и в частности, соответствующей модели памяти, будем рассматривать следующие характеристики:

1. Теории, используемые при моделировании. Выбор используемых теорий непосредственно влияет на эффективность метода моделирования, а также в некоторых случаях и на область его применимости (например, при необходимости поддержки интерполяции Крейга).
2. Поддерживаемые возможности языка Си. Многие распространенные методы моделирования семантики языка Си являются в той или иной степени неполными, поэтому имеет смысл говорить о полноте или, иначе говоря, о выразительности рассматриваемых методов. Выразительность метода особенно важна для инструментов дедуктивной верификации, так как они предъявляют более высокие требования к корректности работы инструмента верификации. В то время как при использовании неподдерживаемой возможности инструмент автоматической статической верификации, к примеру, может продолжить работу с выдачей предупреждения о возможности пропуска ошибки (заменив неподдерживаемый фрагмент кода на некоторое приемлемое приближение), в надежде обнаружить одну или несколько ошибок, не связанных с неподдерживаемой возможностью, инструмент дедуктивной верификации должен давать как можно больше гарантий именно для случая отсутствия

обнаруженных ошибок (так как чаще всего применяется к коду, уже прошедшему множество проверок различными инструментами верификации и тестирования).

3. Поддержка областей памяти наперед не ограниченного размера. Будем далее называть такие области памяти неограниченными. Как и выразительность метода, поддержка неограниченных областей памяти наиболее важна для инструментов дедуктивной верификации.
4. Масштабируемость метода моделирования. Под масштабируемостью будем понимать примерное число операторов языка Си, для которых соответствующая формула пути (для свойства достижимости соответствующего состояния программы), полученная с использованием рассматриваемого метода, может быть разрешена хотя бы одним из современных решателей за некоторое фиксированное время. Масштабируемость становится важной для применения метода моделирования семантики в инструментах автоматической статической верификации, использующих встраивание функций, и таким образом анализирующих длинные последовательности операторов на путях от точки входа до предполагаемого ошибочного состояния.

Будем классифицировать рассматриваемые методы моделирования памяти Си-программ по поддержке в них неограниченных областей памяти.

4. Модели для ограниченных областей памяти

В методах, поддерживающих моделирование памяти не более чем наперед заданного размера, возможно использование теорий, для которых существуют более эффективные алгоритмы разрешения формул, в частности, отказ от использования теорий массивов и логики первого порядка в пользу неинтерпретируемых функций или констант и пропозициональной логики. Самым простым с точки зрения используемых теорий является моделирование памяти Си-программ с использованием неинтерпретируемых констант, которое, как правило, опирается на результаты работы некоторого алгоритма анализа синонимичных указателей — *алиасов.*

4.1 Использование анализа алиасов

Рассматриваемый метод применялся в инструменте автоматической статической верификации BLAST [5], использующем предикатную абстракцию с уточнением по невыполнимым контрпримерам. Метод описан в статьях [23, 8], а также в [24, 25] и диссертации [26] (глава 4).

Суть данного метода можно описать следующим образом. Пусть имеется некоторый алгоритм межпроцедурного нечувствительного к контексту и

потоку управления анализа алиасов, например, [27]. Такой анализ алиасов может определять возможную синонимичность указательных выражений глобально для всей анализируемой программы. Пусть для каждого указательного выражения e из множества всех выражений $\mathcal{E}$ в программе в результате работы анализа алиасов определено множество $\mathcal{A}(e)$ указательных выражений, значение которых может совпадать со значением выражения e. Для хранения отображения $e \to \mathcal{A}(e)$ можно использовать, к примеру, его представление в виде отношения с помощью BDD [28]. Для представления изменяющихся значений всех выражений в исходной программе в рассматриваемом методе предлагается использовать последовательности индексированных неинтерпретируемых констант. Обозначим каждую индексированную неинтерпретируемую константу из конечной последовательности, представляющей изменяющееся в зависимости от состояния программы значение синтаксического выражения e (рассматриваемого в контексте некоторой функции) через $[\![e]\!]_\theta$, где θ — изменяющееся отображение синтаксических выражений в текущие индексы в соответствующих последовательностях. Рассмотрим оператор присваивания по указателю на простой тип данных, то есть не структуру, не объединение и не массив. Пусть правая часть оператора присваивания v может быть представлена в виде формулы $[\![v]\!]_\theta$. Тогда сильнейшее постусловие [29] оператора присваивания по указателю на простой тип будет выглядеть следующим образом:

$$SP(*\,e = v) \equiv Update_\theta(e, v) \wedge \bigwedge_{e' \in \mathcal{A}(e)} \mathbf{ite}([\![e]\!]_\theta = [\![e']\!]_\theta, Update_\theta(e', v), Retain_\theta(e'))$$

$$Update_\theta(e, v) \equiv [\![*\,e]\!]_{\theta'} = [\![v]\!]_\theta \wedge closure_\theta(*\,e, v)$$

$$Retain_\theta(e) \equiv [\![*\,e]\!]_{\theta'} = [\![*\,e]\!]_\theta \wedge closure_\theta(*\,e, *\,e)$$

$$\theta' = \theta \ddagger Incr_\theta(\{*\,e\} \cup \bigcup_{\substack{k=1 \\ *^k e \in \mathcal{E}}}^{\infty} \{*^k\,e\} \cup \bigcup_{e' \in \mathcal{A}(e)} \bigcup_{\substack{k=1 \\ *^k e' \in \mathcal{E}}}^{\infty} \{*^k\,e'\})$$

$$Incr_\theta(S) \equiv \bigcup_{e \in S} \{e \to \theta(e) + 1\}$$

$$closure_\theta(e, v) \equiv \bigwedge_{\substack{k=1 \\ \{*^k e, *^k v\} \subset \mathcal{E}}}^{\infty} [\![*^k\,e]\!]_{\theta'} = [\![*^k\,v]\!]_\theta$$

$$\mathbf{ite}(C, A, B) \equiv (C \wedge A) \vee (\neg C \wedge B)$$

Здесь при обновлении значения выражения по указателю происходит одновременное обновление всех его возможных алиасов (разыменований

указателей с тем же значением), а также всех присутствующих в программе разыменований обновляемого значения и его алиасов, так как записываемое по указателю значение может также являться указателем. При этом происходит увеличение на единицу индексов во всех соответствующих последовательностях.

Части формулы, соответствующие обновлению разыменований левой части оператора присваивания по указателю или синонимичных ей выражений, назовем замыканиями (англ. closure) операции присваивания по указателю. Этим частям соответствует обозначение $closure_\theta(e, v)$. Здесь $*^k\, e$ обозначает $\underbrace{* \ldots *}_{k} e$. В случае введения дополнительного ограничения наличия в исходном тексте программы соответствующих выражений ($\{*^k\, e, *^k\, v\} \subset \mathcal{E}$), глубину разыменования можно не ограничивать ($k = \infty$), получая при этом формулы ограниченного размера.

Обновления всех возможных синонимичных выражений — условные, значение разыменования синонимичного указательного выражения обновляется (обозначение $Update_\theta(e, v)$) в случае, если значение самого выражения совпадает со значением выражения в левой части оператора присваивания и остается прежним (обозначение $Retain_\theta(e)$) в противном случае.

Отображение θ' соответствует обновленному отображению θ после увеличения на единицу индексов всех потенциально обновленных выражений. Для формализации этого отображения использована операция перезаписывающего объединения отображений $m_1 \ddagger m_2$, которая обозначает отображение m_2, дополненное парами $k \to m_1(k)$, где $k \notin \operatorname{dom} m_2, k \in \operatorname{dom} m_1$.

Чтение значения по указателю e представляется в виде формулы непосредственно как $[\![e]\!]_\theta$. Благодаря использованию дополнительного ограничения $\{*^k\, e, *^k\, v\} \subset \mathcal{E}$ (рассматриваются только выражения, присутствующие в исходном коде программы) данный метод позволяет полностью поддерживать указатели на любые простые типы данных, включая другие указатели (поддерживаются типы вида $t \underbrace{* \ldots *}_{k}$ для любого $k \geq 0$). Используя эквивалентность $* \&a \equiv a$, можно также поддерживать взятие адреса у переменной. С помощью ряда приемов, описанных в [8], данный метод может быть расширен на структуры, в том числе вложенные.

При моделировании семантики операций с указателями с использованием неинтерпретируемых констант и анализа алиасов непосредственно используется только теория равенства. Поэтому данный метод может использовать различные теории, в зависимости от используемой модельной семантики операций над значениями. В инструменте BLAST использовалась

теория вещественной линейной арифметики и неинтерпретируемых функций (для приближения нелинейных операций). В принципе возможно также использование теорий нелинейной вещественной арифметики, целочисленной арифметики (линейной и нелинейной), теории битовых векторов конечной длины. Поддерживаемые возможности языка Си в данном методе ограничены простыми и структурными типами данных, не поддерживаются массивы, адресная арифметика, объединения, произвольные приведения типов указателей. Неограниченные наперед области памяти в общем случае (например, массивы переменной длины) не поддерживаются, однако может поддерживаться проверка некоторых простых свойств (не выходящих за рамки используемой логики) для произвольных рекурсивных структур данных, в том числе неограниченного размера. Для этого вместо множества $\mathcal{E}$ выражений в программе необходимо рассмотреть множество всех выражений, значение которых прямо или косвенно доступно на текущем рассматриваемом пути выполнения. Например, для пути, в котором имеются операторы $q = p;$ и $p \to f = q;$ и поле f имеет тот же тип, что и указатель p, доступно не только значение выражения $p \to f$ (прямо), но и $p \to f \to f$ (косвенно). В силу конечности длины пути, соответствующее множество доступных выражений также конечно. Благодаря использованию теорий, эффективно поддерживаемых современными решателями, данный метод хорошо масштабируем. Из данных, приведенных в статье [30], можно сделать вывод, что реализация инструмента BLAST, использующая данный метод, успешно работала для трасс длиной в несколько сотен операторов.

Рассмотрим пример построения формулы в модели памяти на основе анализа алиасов. Формулу будем строить для следующего фрагмента Си-программы с заданным проверочным условием:

```
struct range {
      int start, end;
} default = {0, 0};
…
struct range *r0 = &default, *r1;
r1 = malloc(sizeof (struct range));
r1->start = r0->start;
r0 = r1;
assert (r0->start == 0);
```

Формулу построим для случая, когда вызов malloc успешно выделяет память, возвращая ненулевой указатель. Здесь для присваиваний r0 = &default; и r0 = r1; в указатель r0 на структуру range необходимо генерировать формулы замыкания на глубину 1. То есть в формуле необходимо представлять тот

факт, что после присваивания в указатель r0 значения r0->start и r0->end также изменятся соответствующим образом. Для определения новых значений r0->start и r0->end достаточно взять соответствующие поля от правых частей соответствующих операторов присваивания. Подформулы замыкания для присваиваний в указатели на структуры далее отмечены серым цветом. Для присваивания r1->start = r0->start;, соответствующего изменению значения поля start структуры range необходимо генерировать формулы изменения значения возможных алиасов указателя r1. Будем полагать, что возможными алиасами указателя r1 являются указатели r0 и &default (в общем случае возможные алиасы зависят от используемого алгоритма анализа алиасов). В таком случае для каждого из них нужно сгенерировать формулу возможного обновления значения при условии равенства указателю r1 и формулу возможного сохранения прежнего значения при условии неравенства указателю r1. Соответствующие подформулы также отмечены далее серым цветом. Будем использовать для последовательностей значений переменных, соответствующих изменяющимся значениям полей структур, имена вида имя_структуры$имя_поля. Все последовательности будем индексировать, начиная с нуля, соответствующего начальному значению, заданному при инициализации соответствующей переменной (или поля структуры). Для проверки выполненности целевого свойства объединим полученную формулу пути с помощью конъюнкции с отрицанием формулы, выражающей целевое свойство. Если полученная формула окажется невыполнимой, то можно будет утверждать, что целевое свойство выполнено для любого выполнения проверяемого фрагмента Си-программы.

В целом построенная формула будет иметь вид:

$$
\begin{gathered}
default\$start_0 = 0 \wedge default\$end_0 = 0 \wedge \\
r0_1 = default \wedge (r0\$start_1 = default\$start_0 \wedge r0\$end_1 = default\$end_0) \wedge \\
r1_1 > 0 \wedge r1_1 \neq default \wedge \\
r1\$start_1 = r0\$start_1 \\
\wedge ((r1_1 = r0_1 \wedge r0\$start_2 = r0\$start_1 \wedge r0\$end_2 \\
= r0\$end_1) \\
\vee (r1_1 \neq r0_1 \wedge \ r0\$start_2 = r0\$start_1 \wedge r0\$end_2 \\
= r0\$end_1)) \\
\wedge ((r1_1 = defualt \wedge default\$start_1 \\
= r0\$start_1 \wedge default\$end_1 = r0\$end_1) \vee (r1_1 \\
\neq default \wedge default\$start_1 \\
= default\$start_0 \wedge default\$end_1 = default\$end_0))) \wedge \\
r0_2 = r1_2 \wedge (r0\$start_3 = r1\$start_1 \wedge r0\$end_3 = r1\$end_1) \wedge \\
\neg(r0\$start_3 = 0)
\end{gathered}
$$

Для того чтобы убедиться в ее невыполнимости, оставим в формуле только существенные конъюнкты:

$$default\$start_0 = 0 \wedge r0\$start_1 = default\$start_0 \wedge r1\$start_1 \\ = r0\$start_1 \wedge r0\$start_3 = r1\$start_1 \wedge \neg(r0\$start_3 = 0)$$

В полученной формуле легко видеть, что $r0\$start_3 = r1\$start_1 = default\$start_0 = 0$ противоречит условию $\neg(r0\$start_3 = 0)$. Таким образом, полученная формула невыполнима, а целевое свойство выполнено.

4.2 Использование неинтерпретируемых функций и логики первого порядка

Модель памяти на основе анализа алиасов неприменима для верификации программ, в которых на выполненность целевых свойств существенное влияние оказывают различные операции с массивами. Для массивов наперед не заданного размера, а также для массивов достаточно большого размера соответствующие формулы для обновлений значений переменных после присваивания в указатели на элементы массивов, а также после присваиваний в сами элементы массивов либо в принципе не могут быть получены, либо слишком быстро растут пропорционально размерам используемых массивов. Метод моделирования памяти, описанный в этом разделе, также основан на использовании анализа возможных алиасов, но позволяет поддерживать проверку некоторых свойств для программ с массивами.

В статье [31] описан метод моделирования семантики предикатов с указателями, используемый в инструменте верификации SLAM2 [32]. В методе предлагается использование теории неинтерпретируемых функций в сочетании с логикой первого порядка. При кодировании используются три неинтерпретируемые функции: A, V и S, соответствующие трем возможным операциям над *размещениями* (англ. *locations*) – произвольными объектами в памяти программы. А именно: $A(X)$ соответствует взятию адреса размещения X, $V(X)$ — взятию значения размещения X, $S(X, f)$ — получению вложенного размещения, соответствующего полю или индексу f составного размещения X (структуры или массива), а специальный случай $S(X, -1)$ (предполагается, что любое другое $f \neq -1$) — разыменованию указателя, являющегося значением размещения X. Семантика неинтерпретируемых функций формализуется с помощью набора аксиом, например, $\forall X, Y.\ A(X) = A(Y) \Longrightarrow X = Y$. Метод применим для переменных простых типов, в том числе указателей, а также структур, в том числе вложенных, и массивов. Не поддерживаются объединения и произвольные приведения типов указателей. В статье [31] не описывается моделирование семантики оператора присваивания (например, в виде соответствующего слабейшего предусловия или сильнейшего постусловия), что не позволяет однозначно сделать вывод о наборе

поддерживаемых конструкций языка Си, однако в статьях [32, 33] упоминается использование слабейших предусловий и анализа алиасов. Таким образом, вероятно, поддерживается проверка некоторых простых (неиндуктивных) свойств для рекурсивных структур данных и массивов наперед не ограниченного размера. Исходя из данных, приведенных в статье [34], масштабируемость модели памяти, используемой SLAM, сравнима с масштабируемостью модели памяти, используемой BLAST (длина трасс до нескольких сот операторов).

Рассмотрим пример фрагмента программы, использующего массивы, и проверим заданное для него целевое свойство с помощью модели памяти на основе неинтерпретируемых функций с кванторами, используемой в инструменте SLAM2.

```
struct array {
      int len, *data;
} a = {2, (int []){0, 0}};
…
struct array *b = &a;
b->data[0] = 1;
assert (b == &a && a.data[0] == 1 && b->len == 2);
```

Несмотря на то, что рассматриваемая модель памяти использует логику первого порядка, для построения формул в ней также требуется использование анализа алиасов. Для целевого пути в программе строится слабейшее предусловие по рекурсивным правилам преобразования предикатов, которые требуют знания о возможных алиасах всех указательных выражений в пути для правильной подстановки значений переменных в формуле при обработке операторов присваивания. Более точно, соответствующее преобразование слабейшего предусловия для присваивания вида $* x = e$:

$(x = \&y_1 \wedge Q[e/y_1]) \vee \ldots \vee (x = \&y_k \wedge Q[e/y_k])$

Здесь Q — текущее слабейшее предусловие для нижней части пути, $y_1, \ldots, y_k$ — переменные, которые может адресовать указательное выражение x, $[x\ /\ y]$ означает подстановку выражения x вместо переменной y (правила построения формулы гарантируют корректность такой подстановки). Таким образом, каждое присваивание по указателю дублирует нижнюю часть формулы отдельно для каждого возможного рассматриваемого значения указательного выражения. В приведенном примере значение указателя `b->data[0]` в операторе присваивания `b->data[0] = 1;` может быть разрешено однозначно. Поэтому в результирующей формуле рассматривается

только один случай `b->data[0]==&a.data[0]`. Формула пути так же, как и в примере для модели памяти на основе анализа алиасов, объединяется с отрицанием проверяемого свойства. Поля структуры array имеют индексы: len — 0, data — 1. Для получения размещения поля структуры используется функция $S(l,f)$, где l — размещение структуры, f — индекс поля. Для моделирования операции разыменования используется частный случай применения функции S при $f=-1$. Для доказательства невыполнимости формулы в данном примере требуется только одна аксиома, задающая свойство функций S,V и A: $\forall X,Y.V(X)=A(Y)\Longrightarrow S(X,-1)=Y$. Аксиома утверждает, что если значение указателя X равно адресу переменной Y, то результатом разыменования указателя X является переменная Y (соответствующее ей размещение). Полученная формула имеет вид:

$$\begin{aligned}(\forall X,Y.V(X)=A(Y)\Longrightarrow S(X,-1)=Y)\wedge V\big(S(a,0)\big)=2\wedge V(b)\\ =A(a)\wedge V\big(S(S(S(S(b,-1),1),-1),0)\big)\\ =1\\ \wedge\neg\big(V(b)=A(a)\wedge V\big(S(S(S(a,1),-1),0)\big)\\ =1\wedge V\big(S(S(b,-1),0)\big)=2\big)\end{aligned}$$

Аксиома позволяет заменить выражения $S(b,-1)$ внутри аргументов функции V на a, показав тем самым невыполнимость построенной формулы.

Модель памяти на основе теории неинтерпретируемых функций с кванторами, однако, плохо справляется с некоторыми случаями вложенных массивов (массивов, содержащих внутри элементов указатели на элементы других массивов). Рассмотрим следующий пример и соответствующую формулу:

```
struct array *b[] = {&a, &a};
int i = nondet_int() ? 0 : 1;
b[i]->data[0] = 1;
assert (a.data[0] == 1);
```

$$\begin{aligned}(\forall X\,Y.X=A(Y)\Longrightarrow S(X,-1)=Y)\wedge((V\big(S(S(b,0),-1)\big)=A(a)\wedge V(i)\\ =0\wedge V\big(S(S(S(S(b,0)),-1),1)-1),0)\big)\\ =1\wedge\neg\big(V\big(S(S(S(a,0),-1),0)\big)=1\big)\vee(V\big(S(S(b,1),-1)\big)\\ =A(a)\wedge V(i)=1\wedge\\ V\big(S(S(S(S(b,1)),-1),1)-1),0)\big)=1\wedge\neg\big(V\big(S(S(S(a,0),-1),0)\big)=1\big)))\end{aligned}$$

Здесь внешняя функция nondet_int()моделирует недетерминированный выбор целочисленного значения и отдельно рассматриваются два случая для различных возможных значений переменной i. Нетрудно видеть, что размер таких формул для фиксированной длины рассматриваемого пути пропорционален длине

используемых в программе массивов указателей и так же, как и в случае с моделью памяти на основе анализа алиасов, легко может оказаться неприемлемо большим для достаточно длинных рассматриваемых путей или достаточно больших массивов указателей. Данную проблему позволяют решить модели памяти для областей наперед не ограниченного размера.

5. Модели для неограниченных областей памяти

Модели памяти, поддерживающие произвольные области памяти наперед не ограниченного размера (в частности, массивы переменной длины), опираются на использование логики первого порядка или теории массивов. При использовании этих моделей памяти размер формулы пути никогда не зависит от размеров упоминаемых на этом пути областей памяти программы (например, размеров массивов или числа элементов в связном списке).

5.1 Типизированная модель памяти

Типизированная модель памяти реализована, в частности, в инструменте дедуктивной верификации VCC2 [35] и описана в статье [36]. Основная особенность этой модели памяти — в том, что в контексте инструмента дедуктивной верификации в этой модели памяти делаются дополнительные предположения о семантике моделируемых операторов исходной программы без потери корректности и полноты моделирования. Это возможно благодаря наличию задаваемых пользователем аннотаций, а также возможности генерации нескольких условий верификации вместо одной общей формулы пути. В частности, можно потребовать от пользователя аннотировать некоторые дополнительные аспекты семантики верифицируемой программы, осуществить моделирование семантики в заданных пользователем предположениях, а затем помимо обычных условий верификации сгенерировать дополнительные проверки для заданных пользователем предположений. При этом в контексте дедуктивной верификации при моделировании памяти помимо собственно состояния памяти программы можно использовать также и дополнительное модельное (англ. model или ghost) состояние, изменяемое как в результате выполнения операторов исходной программы, так и в результате выполнения специальных модельных операций, задаваемых в виде аннотаций.

В модели памяти VCC2 вводится модельное отображение пар вида (адрес, тип) в специальный булевый флаг валидности, который определяет, является ли данный адрес адресом начала объекта соответствующего типа. Так как Си является слабо типизированным языком, модельное отображение валидности типизированных указателей в VCC2 имеет состояние, которое может быть изменено с помощью специальных модельных операций. Таким образом,

модельное отображение валидности остается постоянным при моделировании семантики одного оператора, но может изменяться в ходе выполнения программы. При этом для обеспечения корректности моделирования памяти поддерживается инвариант о том, что в каждом состоянии для каждого адреса может быть не более одного соответствующего валидного типа данных. В силу отсутствия непосредственного моделирования защиты памяти в модельной семантике VCC2 можно считать, что для каждого адреса этот тип всегда ровно один. Моделирование семантики программы происходит в предположении о том, что каждый указатель содержит адрес, валидный тип которого совпадает с типом указателя. Такой указатель будем называть *валидным*. Таким образом, моделируемая программа должна обращаться к памяти только через валидные указатели. Это свойство проверяется с помощью генерации соответствующих условий верификации. В VCC2 используется две модельные операции для изменения состояния модельного отображения валидности: одна для разбиения произвольного объекта в памяти в массив байт (char), а другая – для соединения массива байт в объект произвольного типа соответствующего размера. Семантика этих двух операций включает преобразование значений в моделируемой памяти в/из типа char в соответствии с побайтовой структурой преобразуемого объекта. При этом для кодирования значений различных типов могут в принципе использоваться различные логические теории. Кроме этого, модель памяти VCC2 предполагает использование логики первого порядка для задания дополнительных аксиом, ограничивающих возможные пересечения адресуемых объектов в памяти программы случаем вложения. Вместе использование типизации памяти и дополнительных аксиом позволяет генерировать более простые условия верификации, увеличивая масштабируемость модели памяти. Кроме этого, в большинстве случаев уменьшается размер задаваемых пользователем спецификаций, так как в них не требуется явно указывать отсутствие пересечений (за исключением вложения) между объектами различающихся типов в памяти программы.

В статье [36] модель памяти описывается с помощью введения базового («игрушечного», «toy language») языка описания трасс для дедуктивной верификации и формализации исходной (нетипизированной) и модельной (типизированной) семантик этого базового языка. При этом трассы на базовом языке представляют собой последовательности операторов, а обе семантики определятся как мелкошаговые [37] с помощью правил вывода отношений вычисления и рекуррентных соотношений для функций означивания чистых выражений. Множество значений (результатов вычисления) последовательностей инструкций принимается состоящим из двух элементов – $\top$ и $\bot$, соответствующих результату проверки всех условий верификации. $\top$ означает выполненность всех условий верификации, $\bot$ – невыполненность хотя бы одного из них. Промежуточное состояние вычисления определяется

как объединение множества $\{\top, \bot\}$ и множества промежуточных состояний, содержащих оставшуюся часть последовательности операторов и некоторый набор отображений, определяющих состояние памяти программы и модельного отображения валидности после выполнения обработанной части трассы. Таким образом, моделирование памяти программы с помощью SMT-формул описывается неявно, однако достаточно компактно и удобно для доказательства свойств рассматриваемой модельной семантики (в частности, её корректности относительно исходной семантики).

Рассмотрим описанный способ формализации модельной семантики подробнее на примере оператора присваивания значения по указателю $* e_1 = e_2$. Предполагается, что сами выражения e_1и e_2 не содержат операции с указателями.

В исходной семантике присваивание значения по указателю на значение типа t будет моделироваться как одновременное обновление значения (sizeof t) последовательных байт в нетипизированной куче, начиная с адреса, определяемого значением выражения e_1. При этом после вычисления значение выражения e_2 обязательно должно быть разбито на (sizeof t) байт и каждый из них должен быть записан в нетипизированную кучу по соответствующему адресу, так как без привлечения каких-либо дополнительных предположений невозможно гарантировать, что ни один из этих изменяемых байт не попадает в область памяти, адресуемую каким-либо другим указательным выражением e_3 (возможно, имеющим тип, отличный от $t*$), по которому может происходить обращение в память далее на рассматриваемом пути выполнения. Таким образом, каждый оператор присваивания по указателю в нетипизированной семантике требует моделирования значений используемых выражений на побайтовом уровне, возможного моделирования обновления значений сразу по нескольким адресам в куче, а также явной спецификации дополнительных предположений о непересечении адресов между объектами различных типов. Все эти факторы приводят к генерации трудноразрешимых условий верификации.

Поэтому при построении формул в инструментах дедуктивной верификации вместо исходной семантики языка программирования обычно используется его модельная семантика.

В модельной семантике оператору присваивания $* e_1 = e_2$ соответствует правило вывода для соответствующего отношения вычисления :

$$e_1 : t* \Longrightarrow \langle * e_1 = e_2; R | \mathcal{E}, \mathcal{T}, \mathcal{M} \rangle \blacktriangleright \langle R | \mathcal{E}, \mathcal{T}, \mathcal{M} \ddagger \{ [\![e_1]\!]^d_{\mathcal{E}} \to [\![e_2]\!]^{8*\text{sizeof } t}_{\mathcal{E}} \} \rangle$$

Это правило вывода по сути означает, что если выражение e_1 имеет тип $t*$, то промежуточное состояние вычисления, в котором первым оператором является $* e_1 = e_2$, а оставшаяся часть пути — R, связано (несимметричным) отношением с промежуточным состоянием, в котором присутствует только

оставшаяся часть пути R (то есть обработан один оператор $*e_1 = e_2$). При этом эти два связанных отношением промежуточных состояния вычисления отличаются состоянием памяти программы. Память описывается тремя отображениями: $\mathcal{E}, \mathcal{T}$ и $\mathcal{M}$, где $\mathcal{E}$ — означивание переменных (в базовом языке они все находятся в одной области видимости), $\mathcal{T}$ — упомянутое ранее отображение валидности, отображающее адреса в однозначно соответствующие им типы данных размещенных по этим адресам значений, а $\mathcal{M}$ — отображение пар вида (адрес, тип) в значения соответствующих типов. После выполнения оператора присваивания в память по адресу $[\![e_1]\!]_{\mathcal{E}}^{d}$ (результат вычисления выражения e_1 без побочных эффектов и без операций с указателями в контексте $\mathcal{E}$, представленный в виде битового вектора длины d) будет записано значение $[\![e_2]\!]_{\mathcal{E}}^{8*\mathrm{sizeof}\,t}$ длиной (**sizeof** t) байт.

В формуле сильнейшего постусловия для представления значения отображения $\mathcal{M}$ может быть использовано несколько последовательностей логических массивов различного типа — по одной последовательности для каждого типа данных. При этом для целочисленных типов данных можно использовать, к примеру, теорию линейной арифметики, так как в модельной семантике в силу неявного предположения о разделении памяти по типам формализация операций на побитовом уровне не обязательна. Таким образом, рассмотрение семантики на побитовом уровне в каждом присваивании не происходит. Кроме этого, эффективно моделируется неявное предположение о непересечении значений различных простых типов данных в памяти программы.

Рассмотренное правило вывода, однако, нельзя непосредственно применять для каждого оператора присваивания, так как используемые в нем неявные предположения о валидности указателя в левой части оператора присваивания могут быть не выполнены. Поэтому каждый раз при использовании данного правила вывода инструмент дедуктивной верификации дополнительно генерирует проверку соответствующего предусловия $\mathcal{T}\left[[\![e_1]\!]_{\mathcal{E}}^{d}\right] = t$.

Если предусловие окажется невыполненным, то соответствующее промежуточное состояние $\langle *e_1 = e_2; R|\mathcal{E}, \mathcal{T}, \mathcal{M}\rangle$ окажется связано отношением с ошибочным состоянием $\perp$.

Более полный пример формализации модельной семантики описанным способом можно найти в статье [38].

Итак, модель памяти VCC2 предполагает использование теории массивов *и/или* логики первого порядка, а также других теорий для моделирования операций над данными, при этом для каждого типа данных может быть использован отдельный набор теорий. При отсутствии обращений к одной и той же области памяти через указатели различных типов в рамках одного оператора (это ограничение можно считать чисто синтаксическим и легко

устранимым с помощью использования дополнительных переменных), модель памяти VCC2 поддерживает все конструкции языка Си и применима для областей памяти произвольного наперед не заданного размера. Однако использование в контексте инструмента дедуктивной верификации говорит о применении модели памяти к трассам лишь небольшого размера, как правило, ограниченных рамками одной функции, то есть длиной порядка нескольких десятков операторов.

В следующем разделе на примере будет рассмотрена модель памяти, являющаяся уточнением типизированной модели памяти, использованной в инструменте VCC2.

5.2 Модель Бурстала-Борната

Модель Бурсталла-Борната, также известная как модель «поле как массив» (англ. Burstall-Bornat model, component-as-array model) была предложена и использована в работах [39, 40] в контексте частично автоматизированной дедуктивной верификации. Основная идея этой модели памяти заключается в том, чтобы раздельно представлять состояния отдельных компонентов составных объектов, что для языка Си по сути означает разделение представления состояния памяти программы по полям структур. Представление состояния каждого отдельного поля при описании модели может быть не конкретизировано, так же, как и при описании типизированной модели памяти (на практике может использоваться теория массивов или аксиоматизация в логике первого порядка). Основную проблему при описании этой модели памяти для языка Си составляют переменные и поля структур, доступные по указателям, то есть потенциально все переменные и поля, для которых в исходном коде Си-программы присутствует операция взятия адреса &.

Одним из решений этой проблемы является применение к исходному коду программы нормализующих преобразований, устраняющих адресацию переменных и полей структур с помощью переписывания их в указатели и введения специальных фиктивных структур с одним полем соответствующего типа. Такие преобразования описаны, например, в статьях [41, 42, 43]. Другим возможным решением является введение отдельных отображений для представления состояния адресуемых указателями полей и переменных, например, с использованием разделения по типам аналогично типизированной модели памяти. В рамках самой модели Бурсталла-Борната выбор одного из этих решений не существенен, так как в итоге приводит к построению одинаковых логических формул (адресуемые переменные и поля в любом случае разделяются только по типам). Однако при расширении данной модели, например, с помощью дальнейшего разделения представления состояния памяти, перетипирующие преобразования исходного кода оказываются проще, по крайней мере, при теоретическом описании полученных моделей. Такие

преобразования применяются, например, в инструменте дедуктивной верификации Jessie. Решение без дополнительных преобразований используется, например, в инструменте дедуктивной верификации WP [44], реализующем модель Бурсталла-Борната. Один из вариантов этой модели памяти реализован также в инструменте дедуктивной верификации VCC3 [2]. Для достижения полноты поддержки возможностей языка Си в инструментах дедуктивной верификации модель Бурсталла-Борната может быть расширена модельным состоянием и специальными модельными операциями разбиения и соединения аналогично типизированной модели памяти.

Модель Бурсталла-Борната в целом обладает примерно теми же характеристиками, что и типизированная модель памяти, однако обеспечивает в общем случае меньшее количество индексов, приходящихся на одно отображение, представляющее часть состояния памяти программы и, как показывают результаты сравнения моделей памяти, в частности, [2], может быть более эффективна на практике по крайней мере в контексте дедуктивной верификации.

Рассмотрим модель Бурсталла-Борната на приведенном ранее примере, в котором происходит потенциальное ухудшение масштабируемости модели памяти, используемой в инструменте SLAM:

```
struct array *b[] = {&a, &a};
int i = nondet_int() ? 0 : 1;
b[i]->data[0] = 1;
assert (a.data[0] == 1);
```

В данном примере будем представлять значения поля `data` структуры `array` с помощью последовательности индексированных неинтерпретируемых функций $array\_data_i$, значения массивов типа **`int`** — с помощью последовательности int_i, а значения массивов типа **`struct`** `array *` — с помощью последовательности $struct\_array\$_i$. Для каждого присваивания по указателю в этой модели памяти добавляется аксиома, утверждающая сохранение значений соответствующей неинтерпретируемой функции для всех индексов (адресов), отличных от изменяемого в операторе присваивания. Результирующая формула имеет вид:

$$\begin{aligned} struct\_array\$_1(b) = a \wedge struct\_array\$_1(b+1) = a \wedge i_1 \\ = \boldsymbol{ite}(x = 0{,}0{,}1) \\ \wedge\, int_1\big(array\_data_0\big(struct\_array\$_1(b+i)\big) + 0\big) \\ = 1 \\ \wedge\, \big(\forall X.\, X \neq array\_data_0\big(struct\_array\$_1(b+i)\big) + 0 \\ \Longrightarrow int_1(X) = int_0(X)\big) \wedge \neg\big(int_1\big(array\_data_0(a)\big) = 1\big) \end{aligned}$$

Здесь неинтерпретируемая константа x используется для моделирования недетерминированного выбора значения переменной i из множества $\{0, 1\}$ (соответствует оператору i = nondet_int() ? 0 : 1;). По сравнению с формулой, построенной ранее с использование модели памяти инструмента SLAM, описанной в разделе 4.2, размер данной формулы не зависит от размера используемых на рассматриваемом пути массивов (в том числе массивов указателей). Для установления ее невыполнимости необходимо рассмотрение случаев $i_1 = 0$ и $i_1 = 1$, однако оно может происходить непосредственно во время разрешения формулы, не влияет существенно на ее размер и позволяет полнее использовать возможности алгоритмов разрешения, реализованные в современных SMT-решателях — для длинных путей с большой вероятностью будут рассмотрены различные случаи только для переменных, существенно влияющих на выполнимость формулы.

В случае использования в программе большого числа структур одинакового типа, модель Бурсталла-Борната, предполагающее разбиение памяти программы по полям структур, может становиться менее эффективной. Ее можно оптимизировать с использованием результатов некоторого алгоритма анализа алиасов. В случае использования простого алгоритма на основе системы непересекающихся множеств, который обеспечивает корректность модели памяти для существенного подмножества Си-программ, получим модель памяти с регионами.

5.3 Модель с регионами

Это расширение модели памяти Бурсталла-Борната предложено в статье [41] и используется в инструменте дедуктивной верификации Jessie [45] (а также в его предшественнике, инструменте Caduceus [46]). Основная идея этой модели памяти заключается в том, что память программы в логическом представлении может разделяться не только по компонентам составных объектов (для языка Си – полям структур), как в модели Бурстала-Борната, но и по регионам – непересекающимся множествам указательных выражений, таких, что выражения из разных множеств не могут адресовать пересекающиеся области памяти. При этом эти два критерия разделения состояния памяти могут применяться независимо друг от друга.

Формально предложенный в статье [41] подход к моделированию памяти Си-программ включает в себя три составляющие.

Первая из них – это набор нормализующих преобразований, применяемых к исходной Си-программе, в результате которого получается программа на базовом языке. В программе на базовом языке в качестве типов переменных и полей структур допускаются только целые числа и указатели на структуры. В частности, структуры не могут непосредственно (без использования указателей) содержать в себе другие структуры или массивы, не допускается использование

объединений и приведение типов указателей. Все структуры и массивы в программах на базовом языке размещаются в динамической памяти.

Вторая составляющая – это специальная система типов для базового языка, в которой типы указателей на структуры параметризованы регионами. При этом регионы указательных типов параметров функций сами являются параметрами функций, что порождает систему типов с параметрическим полиморфизмом, аналогичную классической системе Хендли-Милнера [47]. Основной целью введения системы типов для базового языка является существенное упрощение доказательства теоремы об относительной корректности модели памяти с регионами. Теорема утверждает, что корректно типизированная программа на базовом языке имеет одинаковую семантику в модели памяти с регионами и в модели Бурсталла-Борната.

Наконец, третьей составляющей подхода является алгоритм вывода типов (регионов) для базового языка. В статье [41] этот алгоритм приводится без формального доказательства корректности, в предположении, что после вывода типов программа на базовом языке будет транслирована на язык WhyML [48] (имеющий систему типов с параметрическим полиморфизмом) так, что все типы, выведенные для базового языка будут непосредственно выражены с помощью типов языка WhyML в результирующей программе. Таким образом, будет осуществлена апостериорная проверка корректности вывода типов для программы на базовом языке.

Для моделирования семантики языка Си с использованием модели памяти с регионами, как и для двух ранее рассмотренных моделей, может использоваться теория массивов или комбинация теории неинтерпретируемых функций с логикой первого порядка, а также другие теории для моделирования операций над данными. Возможности языка Си, поддерживаемые моделью памяти с регионами, в том виде, в котором она описана в статье [41], ограничены возможностями используемого в статье базового языка. Таким образом, модель не поддерживает по крайней мере вложенность объектов, объединения и произвольные приведения типов указателей. В остальном, модель с регионами применима для областей памяти в том числе наперед не ограниченного размера. Масштабируемость модели памяти с регионами незначительно отличается от масштабируемости типизированной модели памяти и модели Бурсталла-Борната, однако модель с регионами обеспечивает в общем случае еще меньшее число индексов на отображение, чем в этих моделях, и может быть эффективнее на практике.

Приведем пример Си-программы и соответствующей формулы для проверки целевого свойства с использованием модели памяти с регионами:

```
struct cords {
      int len, *x, *y;
} v0 = {2, (int []){0, 0}, (int []){0, 0}};
…
```

```
struct coords *v[] = {&v0, &v0};
int i = nondet_int() ? 0 : 1;
v[i]->x[0] = 1;
assert (v0.y[0] == 0);
```

$$
\begin{aligned}
& v0\_len_0(v0) = 2 \wedge v0\_x_0(v0) > 0 \wedge v0\$x_0(v0\_x_0(v0) + 0) \\
& \quad = 0 \wedge v0\$x_0(v0\_x_0(v0) + 1) = 0 \wedge v0\_y_0(v0) \\
& \quad > 0 \wedge v0\$y_0(v0\_y_0(v0) + 0) = 0 \wedge v0\$y_0(v0\_y_0(v0) + 1) \\
& \quad = 0 \wedge v_1(v + 0) = v0 \wedge v_1(v + 1) = v0 \wedge i_1 \\
& \quad = \boldsymbol{ite}(x, 0, 1) \wedge v0\$x_1\big(v0\_x_0\big(v_1(v + i)\big) + 0\big) \\
& \quad = 1 \\
& \quad \wedge \Big(\forall X. X \neq v0\_x_0\big(v_1(v + i)\big) + 0 \Longrightarrow v0\$x_1(X) = v0\$x_0(X)\Big) \\
& \quad \wedge \neg(v0\$y_0(v0\_y_0(v0) + 0) = 0)
\end{aligned}
$$

Здесь для именования индексированных последовательностей функций используются имена вида имярегиона_имяполя[необязательное]. Имя поля отсутствует для регионов, соответствующих простым типам данных. Регионы, соответствующие разыменованиям указательных полей, обозначаются как имяуказателянаструктуру$имяполя.

В данном примере предполагается, что анализ регионов разделил указательные выражения в программе на три региона: $v0$, $v0\$x$ и $v0\$y$. При этом регион $v0$ соответствует структурному типу struct cords, который имеет три поля (len, x и y), что дает три индексированные последовательности функций: $v0\_len_i, v0\_x_i$ и $v0\_y_i$ для представления состояния этого региона. Регионы $v0\$x$ и $v0\$y$ соответствуют простому типу данных int и поэтому их состояние кодируются с помощью индексированных последовательностей $v0\$x_i$ и $v0\$y_i$.

По сравнению с формулой, построенной с использованием модели памяти Бурсталла-Борната, элементы массивов `v0.x` и `v0.y` в данной формуле представляются независимо с помощью различных последовательностей неинтерпретируемых функций ($v0\$x_i$ и $v0\$y_i$). Таким образом, состояние этих массивов изменяется независимо. Поэтому невыполнимость построенной формулы легко устанавливается по наличию двух несовместных условий: $v0\$y_0(v0\_y_0(v0) + 0) = 0$ и $\neg(v0\$y_0(v0\_y_0(v0) + 0) = 0)$ (в формуле выделены серым цветом).

5.4 Полное моделирование семантики относительно логики множеств элементов списков

Все рассмотренные ранее модели памяти обладают тем ограничением, что не позволяют проверять индуктивно определяемые свойства рекурсивных структур данных, такие, например, как “все элементы односвязного списка неотрицательны” или “данный элемент списка адресуется его головным

элементом и только им". Для одного из классов свойств рекурсивных структур данных, а именно – класса LISBQ – Logic of Interpreted Sets and Bounded Quantification – логики интерпретируемых множеств и ограниченной квантификации, существует метод построения полных относительно этой логики слабейших предусловий для операторов присваивания значения в поля структур по указателю. Этот метод был реализован в инструменте дедуктивной верификации Си-программ HAVOC [40]. Метод детально описан в статье [50] и в данной статье приведем лишь пример формулы для проверки свойства "все элементы односвязного списка отличны от нуля" для одного оператора присваивания, осуществляющего объединение (склеивание или конкатенацию) двух таких списков:

```
struct list {
      struct list *next;
      int v;
};
…
struct list *l1, *l2, *l;

assume(
```

$$(\forall X \in Btwn(list.next, l1, NULL).list.next(X) \neq NULL \wedge list.v(X) \neq 0) \wedge$$
$$(\forall X \in Btwn(list.next, l2, NULL).list.next(X) \neq NULL \wedge list.v(X) \neq 0) \wedge$$
$$l \neq NULL \wedge$$
$$Reach(list.next, l1, l, NULL) \wedge Reach(list.next, l2, NULL, NULL) \wedge$$
$$\neg Reach(list.next, l2, l, l));$$

```
l->next = l2;
assert(
```

$\forall X \in Btwn(list.next, l1, NULL).list.v(X) \neq 0);$

Здесь предикат вида $\forall X \in Btwn(s.f, e1, e2).P(X)$ обозначает выполненность свойства $P(X)$ для всех элементов, достижимых из элемента, адресуемого указателем $e1$, включительно, с помощью прохождения по полям f структур с тегом s, за исключением поля f структуры, адресуемой указателем $e2$. Предикат $Reach(s.f, e1, e2, e3)$ обозначает достижимость по полям f структур с тегом s элемента, адресуемого $e2$, из элемента, адресуемого $e1$, и элемента, адресуемого $e3$ из элемента, адресуемого $e2$.

В генерируемых формулах для каждого поля структуры $s.f$ используется отдельный предикат $Reach_{s.f}$. Так как в данном примере по существу используется только поле $list.next$, будем для удобства обозначать предикат $Reach_{s.f}(e_1, e_2, e_3)$ как $e_1 \longrightarrow e_2 \longrightarrow e_3$. С использованием дополнительно вводимого обозначения $u \underset{\nrightarrow w}{\longrightarrow} v$ формулу для данного примера можно представить следующим образом:

$$u \underset{\nrightarrow w}{\longrightarrow} v \equiv u \longrightarrow v \longrightarrow w \vee (u \longrightarrow v \longrightarrow v \wedge \neg(u \longrightarrow w \longrightarrow w))$$

$$WP\left(\begin{matrix} Pre; l \to next = l2; \\ \neg(\forall X \in Btwn(list.next, l1, NULL).\, list.v(X) \neq 0) \end{matrix}\right)$$
$$\equiv Pre$$
$$\wedge \neg\Big(\Big(l_1 \underset{\nrightarrow l}{\to} NULL \Longrightarrow \forall X.\, l_1 \longrightarrow X \longrightarrow NULL \Longrightarrow list.v(X) \neq 0\Big)$$
$$\wedge \Big(l \neq NULL \wedge l_1 \underset{\nrightarrow NULL}{\longrightarrow} l \wedge l_2 \underset{\nrightarrow l}{\to} NULL$$
$$\Longrightarrow (\forall X.\, l_1 \longrightarrow X \longrightarrow l \Longrightarrow list.v(X) \neq 0)$$
$$\wedge (\forall X.\, l_2 \longrightarrow X \longrightarrow NULL \Longrightarrow list.v(X) \neq 0)\Big)\Big)$$

Здесь Pre обозначает формулу для предусловия, заданного с помощью конструкции assume, $\mathrm{W}P(op, \varphi)$ — преобразователь предикатов для слабейшего предусловия.

Метод полного моделирования семантики относительно логики LISBQ предполагает использование логики первого порядка и теории неинтерпретируемых функций. При этом существующие реализации этого метода в инструментах дедуктивной верификации генерируют формулы, для которых выполнено дополнительное ограничение стратификации по сортам [51]. В условиях этого ограничения получаемые формулы в логике первого порядка алгоритмически разрешимы. Кроме этого, для соответствующего фрагмента логики первого порядка были предложены эффективные на практике разрешающие алгоритмы [52], реализованные, в частности, в решателе Z3. В результате масштабируемость метода (при использовании соответствующих эффективных решателей) позволила применить его в инструменте дедуктивной верификации HAVOC. Метод позволяет моделировать семантику программ, использующих рекурсивные структуры данных наперед не ограниченного размера, для проверки целевых свойств корректности, заданных в логике LISBQ. Эта логика помимо рассмотренного предиката $Reach$ и множества $Btwn$, а также соответствующего предиката $\in$, вводит еще и класс множеств вида $s.f^{-1}(o)$, каждое из которых содержит все указатели на структуры с тегом s, которые в своем поле f содержат адрес структуры, адресуемой указателем o.

Следует отметить, что поддерживаемый данным методом класс свойств не включает в себя, к примеру, такие индуктивно определяемые функции, как сумма или число элементов списка.

6. Выводы

Подведем общие итоги обзора известных существующих методов моделирования семантики операций над указателями для Си-программ с помощью логических формул в теориях, то есть моделей памяти.

1. Модели памяти могут предполагать использование различных комбинаций логических теорий (логик), причем модель памяти может как ограничивать набор возможных комбинаций теорий, так и поддерживать несколько различных комбинаций теорий, возможно с привлечением различных дополнительных предположений.
2. В различных моделях памяти делаются различные предположения о верифицируемых программах и проверяемых свойствах. Проверка этих предположений может как предусматриваться в рамках самой модели памяти или соответствующего инструмента верификации, так и не предусматриваться. Это может влиять на применимость модели памяти в различных классах инструментов верификации. Например, инструменты дедуктивной верификации должны делать как можно меньше непроверенных предположений о проверяемой программе.
3. Модели памяти различаются по полноте в смысле поддержки конструкций языка Си, а также поддержки неограниченных областей памяти. Соответствующие ограничения также влияют как на применимость модели памяти в различных инструментах верификации, так и на ее применимость для решения различных практических задач. Например, инструменты дедуктивной верификации должны как можно более полно поддерживать определенный набор конструкций языка программирования. Однако, с другой стороны, при постановке задачи реализации новой верифицированной программы в отличии от задачи верификации существующего кода, набор поддерживаемых конструкций может быть ограничен. В инструментах автоматической статической верификации, верифицирующих практически исключительно существующий код, ограничения на входные программы могут существенно сужать область применимости инструмента, однако на практике часто допускается пропуск ошибок (для небольшого числа проверяемых программ) при использовании в исходном коде программы неподдерживаемых возможностей языка.
4. Так как различные наборы логических теорий (логики) обладают разными характеристиками сложности (как правило, от NP-полной до алгоритмически неразрешимой), при использовании логик с большой алгоритмической сложностью, что практически неизбежно в некоторых контекстах, таких как дедуктивная верификация свойств функциональной корректности, большое значение может иметь принадлежность генерируемых формул некоторому фрагменту используемой логики с лучшими характеристиками сложности. Также важны некоторые другие свойства генерируемых формул, существенно влияющие на производительность разрешающих

алгоритмов, такие как количество индексов, приходящихся на логический массив или последовательность неинтерпретируемых функций.

В табл. 1 представлены основные характеристики рассмотренных в данной главе моделей памяти.

Для основных характеристик моделей памяти в таблице приведено одно из трех значений: «да», «нет» и «частично». Значение «частично» в графе «рекурсивные структуры» для всех моделей памяти, кроме LISBQ, соответствует отсутствию возможностей представления в модели памяти соответствующих индуктивных предикатов. Значение «частично» в графе «Неограниченные области памяти» для модели памяти на основе анализа алиасов соответствует отсутствию возможности задания предикатов для неограниченных областей памяти. Значение «частично» в графе «Неограниченные области памяти» для модели памяти SLAM соответствует ограниченности длин вложенных массивов.

Табл. 1. Основные характеристики моделей памяти.

Table 1. Key features of the memory models.

Модель памяти	**Логики**	**Возможности языка Си**			
		рекурсивные структуры	массивы и адресная арифметика	объединения и приведения типов ук-лей	неограниченные области памяти
На основе анализа алиасов	QF_LIA, QF_LRA	**част.**	**нет**	**нет**	**част.**
SLAM	UFLIA, UFLRA	**част.**	**да**	**нет**	**част.**
Типизированная, Бурсталла-Борната	QF_ALIA, QF_ALRA, UFLIA, UFLRA	**част.**	**да**	**да**	**да**
С регионами	QF_ALIA, QF_ALRA, UFLIA, UFLRA	**част.**	**да**	**да**	**да**
LISBQ	UFLIA, UFLRA	**да**	**нет**	**нет**	**да**

Список литературы

[1]. Leonardo de Moura, Nikolaj Bjørner. Satisfiability modulo theories: An appetizer. In Formal Methods: Foundations and Applications, 12th Brazilian Symposium on Formal Methods (SBMF 2009). Gramado, Brazil, August 19–21, 2009. Revised Selected Papers, Lecture Notes in Computer Science, vol. 5902, pp. 23–36. Springer, 2009.

[2]. Sascha Böhme, Michał Moskal. Heaps and Data Structures: A Challenge for Automated Provers. In Nikolaj Børner, Viorica Sofronie-Stokkermans, eds. Automated Deduction. Lecture Notes in Computer Science, vol. 6803, pp. 177–191. Springer, 2011.

[3]. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith. Counterexample-Guided Abstraction Refinement. In Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00), E. Allen Emerson, A. Prasad Sistla (Eds.). Springer-Verlag, London, UK, pp. 154–169, 2000.

[4]. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In Proc. of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), LNCS, vol. 1579, pp. 193–207. Springer-Verlag, 1999.

[5]. 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, October 2007, vol. 9, issue 5, pp. 505–525.

[6]. Mary Sheeran, Satnam Singh, Gunnar Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. Formal Methods in Computer-Aided Design. Lecture Notes in Computer Science, vol. 1954, pp. 127–144.

[7]. Susanne Graf, Hassen Saïdi. Construction of abstract state graphs with PVS. Proceedings of International Conference on Computer Aided Verification, 1997. LNCS, vol. 1254, pp. 72–83. Springer, Berlin Heidelberg, 1997.

[8]. 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 (POPL '04). SIGPLAN Notes, vol. 39, no. 1, pp. 232–244. ACM, New York, NY, USA. 2004.

[9]. Aaron R. Bradley. SAT-based model checking without unrolling. In Proceedings of the 12th International conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'11), Ranjit Jhala, David Schmidt (Eds.), Springer-Verlag, Berlin, Heidelberg, pp. 70–87, 2011.

[10]. Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, Neha Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. Automated Software Engineering, 2013, vol. 20, no. 3, pp. 391–425.

[11]. В. К. Кошелев, В. Н. Игнатьев, А. И. Борзилов. Инфраструктура статического анализа программ на языке C#. Труды ИСП РАН, том 28, вып. 1, стр. 21–40. 2016. DOI: 10.15514/ISPRAS-2016-28(1)-2.

[12]. Hoare C. A. R. An Axiomatic Basis for Computer Programming. Commun. ACM., 1969, vol. 12, no. 10, pp. 576–580.

[13]. Robert W. Floyd. Assigning meanings to programs. In Proceedings of the Symposium in Applied Mathematics, vol. XIX, pp. 19–32. American Mathematical Society, April 1967.

[14]. Edsger W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. Communications of the ACM. 1975, vol. 18, no. 8, pp. 453–457.

[15]. John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science. LICS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 55–74.
[16]. François Bobot, Jean-Christophe Filliâtre, Claude Marché, Andrei Paskevich. Why3: Shepherd Your Herd of Provers. Boogie 2011: First International Workshop on Intermediate Verification Languages. Wrocław, Poland. 2011, pp. 53 – 64.
[17]. K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning (LPAR'10). Edmund M. Clarke, Andrei Voronkov (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 348–370, 2010.
[18]. Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, Alwen Tiu. Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants. International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2006, LNCS, vol. 3920, pp. 167–181. Berlin, Heidelberg: Springer-Verlag, 2006.
[19]. Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Thery, Benjamin Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. Proceedings of the First International Conference on Certified Programs and Proofs. CPP'11. Berlin, Heidelberg: Springer-Verlag, pp. 135–150, 2011.
[20]. Benjamin C. Pierce. Types and Programming Languages. Chapter I. Untyped Systems. The MIT Press, 2002.
[21]. Robert D. Tennent. The denotational semantics of programming languages. Commun. ACM 19, vol. 8 (August 1976), pp. 437–453, 1976.
[22]. Мандрыкин М. У. Моделирование памяти Си-программ для инструментов статической верификации на основе SMT-решателей. Диссертация на соискание ученой степени кандидата физико-математических наук. Институт системного программирования РАН, октябрь, 2016.
[23]. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar. Lazy abstraction. Symposium on Principles of Programming Languages, pp. 58–70, ACM Press, 2002.
[24]. Pavel Shved, Mikhail Mandrykin, Vadim Mutilin. Predicate Analysis with BLAST 2.7. Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems 2012 (TACAS'12), LNCS, vol. 7214,. pp. 525–527.
[25]. Pavel Shved, Vadim S. Mutilin, Mikhail U. Mandrykin. Experience of improving the Blast static verification tool. Programming and Computer Software, vol. 38, issue 3, pp. 134–142, June 2012. DOI: 10.1134/S0361768812030061.
[26]. Мутилин В.С. Верификация драйверов операционной системы Linux при помощи предикатных абстракций. Диссертация на соискание ученой степени кандидата физико-математических наук. Институт системного программирования РАН. 2012. ноябрь.
[27]. Lars O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, University of Copenhagen, DIKU, 1994.
[28]. Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, Navindra Umanee. Points-to Analysis using BDDs. SIGPLAN Notes, vol. 38, no. 5, pp. 103–114. ACM. New York, NY, USA. May 2003.
[29]. Edsger W. Dijkstra, Carel S. Schölten. Predicate Calculus and Program Semantics. The strongest postcondition, pp. 209–215. Springer New York. 1990.

[30]. Alexey Khoroshilov, Vadim Mutilin, Eugene Novikov, Pavel Shved, Alexander Strakh. Towards An Open Framework for C Verification Tools Benchmarking. Proceedings of the 8th international conference on Perspectives of System Informatics (PSI'11), LNCS, vol. 7162, pp. 179–192, 2011.

[31]. Thomas Ball, Ella Bounimova, Vladimir Levin, Leonardo de Moura. Efficient evaluation of pointer predicates with Z3 SMT Solver in SLAM2. March 2010. MSR-TR-2010-24.

[32]. Thomas Ball, Ella Bounimova, Rahul Kumar, Vladimir Levin. SLAM2: Static driver verification with under 4% false alarms. Formal Methods in Computer-Aided Design (FMCAD), 2010, pp. 35–42.

[33]. Thomas Ball, Todd Millstein, Sriram K. Rajamani. Polymorphic Predicate Abstraction. ACM Trans. Program. Lang. Syst., vol. 27, no. 2, March 2005, pp. 314–343. ACM, New York, NY, USA. 2005.

[34]. Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, Abdullah Ustuner. Thorough Static Analysis of Device Drivers. Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006. (EuroSys '06) Leuven, Belgium, pp. 73–85. ACM. New York, NY, USA. 2006.

[35]. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs '09, pp. 23–42. Springer-Verlag. Berlin, Heidelberg. 2009.

[36]. Ernie Cohen, Michał Moskal, Stephan Tobies, Wolfram Schulte. A Precise Yet Efficient Memory Model For C. Electron. Notes Theor. Comput. Sci., October, 2009, vol. 254, pp. 85–103. Elsevier Science Publishers B. V., Amsterdam, The Netherlands. 2009.

[37]. Gordon D. Plotkin. A structural approach to operational semantics. Computer Science Department, Aarhus University. Aarhus, Denmark. Internal Report DAIMI FN-19. 1981.

[38]. Мандрыкин М. У., Хорошилов А. В. Анализ регионов для дедуктивной верификации Си-программ. Программирование, 2016, № 5, стр. 3–29.

[39]. Rodney M. Burstall. Some techniques for proving correctness of programs which alter data structures. Machine intelligence, vol. 7, no. 3, pp. 23–50, 1972.

[40]. Richard Bornat. Proving Pointer Programs in Hoare Logic. In Proceedings of the 5th International Conference on Mathematics of Program Construction (MPC '00), pp. 102–126. Springer-Verlag. London, UK, 2000.

[41]. Thierry Hubert, Claude Marché. Separation Analysis for Deductive Verification. Heap Analysis and Verification (HAV'07), Braga, Portugal, March 2007, pp. 81–93.

[42]. Yannick Moy. Union and Cast in Deductive Verification. Proceedings of the C/C++ Verification Workshop. Technical Report ICIS-R07015, pp. 1–16. Radboud University Nijmegen, July, 2007.

[43]. Мандрыкин М.У., Хорошилов А.В. Высокоуровневая модель памяти промежуточного языка Jessie с поддержкой произвольного приведения типов указателей. Программирование, 2015, т. 41, № 4, стр. 23–29.

[44]. Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, Boris Yakobowski. Frama-C: A Software Analysis Perspective. Proceedings of the 10th International Conference on Software Engineering and Formal Methods. SEFM'12.

Thessaloniki, Greece. 2012. Lecture Notes in Computer Science, vol. 7504, pp. 233–247. Springer-Verlag. Berlin, Heidelberg. 2012.
[45]. Yannick Moy. Automatic Modular Static Safety Checking for C Programs. PhD thesis. Université Paris-Sud. January, 2009.
[46]. Jean-Christophe Filliâtre, Claude Marché. Multi-prover Verification of C Programs. In Proceedings of Formal Methods and Software Engineering: 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8–12, 2004. LNCS, vol. 3308, pp. 15–29. Springer Berlin Heidelberg. 2004.
[47]. Robin Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences, vol. 17, no. 3, pp. 348–375. December, 1978.
[48]. Jean-Christophe Filliâtre, Andrei Paskevich. Why3: Where Programs Meet Provers. Proceedings of the 22Nd European Conference on Programming Languages and Systems. ESOP'13. Rome, Italy. LNCS, vol. 7792, pp. 125–128. Springer-Verlag.
[49]. https://www.microsoft.com/en-us/research/project/havoc/
[50]. Shuvendu Lahiri, Shaz Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '08), pp. 171–182. ACM, New York, NY, USA. 2008.
[51]. Aharon Abadi, Alexander Rabinovich, Mooly Sagiv. Decidable fragments of many-sorted logic. Journal of Symbolic Computation, vol. 45, issue 2, pp. 153–172, February 2010.
[52]. Yeting Ge, Leonardo Moura. Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09). LNCS, vol. 5643, pp. 306–320. Springer-Verlag, Berlin, Heidelberg, 2009.

Survey of memory modeling methods in static verification tools

M.U. Mandrykin <mandrykin@ispras.ru>
V.S. Mutilin <mutilin@ispras.ru>
ISP RAS, 25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation

Abstract. The paper presents a survey of existing approaches to modeling memory states of C programs with SMT-formulas in context of static verification. The paper highlights the essential problems of C memory model development and describes two major groups of C memory models: one comprising of models that support unbounded memory regions and another including the models that don't. Among the models for a priori bounded memory regions the paper discusses a strongest postcondition-based model relying on preliminary alias analysis and a weakest precondition-based model that uses uninterpreted functions and first-order logic to represent pointer predicates. Among the models for unbounded memory areas the paper describes a typed memory model, the Burstall-Bornat model, a region-based model and a model with full support for the Logic of Interpreted Sets and Bounded Quantification (LISBQ) earlier implemented in the HAVOC deductive verification tool.

Keywords: static verification; memory models; SMT-solvers.

DOI: 10.15514/ISPRAS-2017-29(1)-12

For citation: Mandrykin M.U., Mutilin V.S. Survey of memory modeling methods in static verification tools. *Trudy ISP RAN / Proc. ISP RAS*, vol. 29, issue 1, 2017, pp. 195-230 (in Russian). DOI: 10.15514/ISPRAS-2017-29(1)-12

References

[1]. Leonardo de Moura, Nikolaj Bjørner. Satisfiability modulo theories: An appetizer. In Formal Methods: Foundations and Applications, 12th Brazilian Symposium on Formal Methods (SBMF 2009). Gramado, Brazil, August 19–21, 2009. Revised Selected Papers, Lecture Notes in Computer Science, vol. 5902, pp. 23–36. Springer, 2009.

[2]. Sascha Böhme, Michał Moskal. Heaps and Data Structures: A Challenge for Automated Provers. In Nikolaj Børner, Viorica Sofronie-Stokkermans, eds. Automated Deduction. Lecture Notes in Computer Science, vol. 6803, pp. 177–191. Springer, 2011.

[3]. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith. Counterexample-Guided Abstraction Refinement. In Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00), E. Allen Emerson, A. Prasad Sistla (Eds.). Springer-Verlag, London, UK, pp. 154–169, 2000.

[4]. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In Proc. of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), LNCS, vol. 1579, pp. 193–207. Springer-Verlag, 1999.

[5]. 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, October 2007, vol. 9, issue 5, pp. 505–525.

[6]. Mary Sheeran, Satnam Singh, Gunnar Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. Formal Methods in Computer-Aided Design. Lecture Notes in Computer Science, vol. 1954, pp. 127–144.

[7]. Susanne Graf, Hassen Saïdi. Construction of abstract state graphs with PVS. Proceedings of International Conference on Computer Aided Verification, 1997. LNCS, vol. 1254, pp. 72–83. Springer, Berlin Heidelberg, 1997.

[8]. 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 (POPL '04). SIGPLAN Notes, vol. 39, no. 1, pp. 232–244. ACM, New York, NY, USA. 2004.

[9]. Aaron R. Bradley. SAT-based model checking without unrolling. In Proceedings of the 12th International conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'11), Ranjit Jhala, David Schmidt (Eds.), Springer-Verlag, Berlin, Heidelberg, pp. 70–87, 2011.

[10]. Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, Neha Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. Automated Software Engineering, 2013, vol. 20, no. 3, pp. 391–425.

[11]. V. K. Koshelev, V. N. Ignatyev, A. I. Borzilov. C# static analysis framework. Trudy ISP RAN / Proc. ISP RAS. vol. 28, issue 1, 2016, pp. 21-40 (*In Russian*). DOI: 10.15514/ISPRAS-2016-28(1)-2.

[12]. Hoare C. A. R. An Axiomatic Basis for Computer Programming. Commun. ACM., 1969, vol. 12, no. 10, pp. 576–580.

[13]. Robert W. Floyd. Assigning meanings to programs. In Proceedings of the Symposium in Applied Mathematics, vol. XIX, pp. 19–32. American Mathematical Society, April 1967.

[14]. Edsger W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. Communications of the ACM. 1975, vol. 18, no. 8, pp. 453–457.

[15]. John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science. LICS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 55–74.

[16]. François Bobot, Jean-Christophe Filliâtre, Claude Marché, Andrei Paskevich. Why3: Shepherd Your Herd of Provers. Boogie 2011: First International Workshop on Intermediate Verification Languages. Wrocław, Poland. 2011, pp. 53 – 64.

[17]. K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning (LPAR'10). Edmund M. Clarke, Andrei Voronkov (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 348–370, 2010.

[18]. Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, Alwen Tiu. Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants. International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2006, LNCS, vol. 3920, pp. 167–181. Berlin, Heidelberg: Springer-Verlag, 2006.

[19]. Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Thery, Benjamin Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. Proceedings of the First International Conference on Certified Programs and Proofs. CPP'11. Berlin, Heidelberg: Springer-Verlag, pp. 135–150, 2011.

[20]. Benjamin C. Pierce. Types and Programming Languages. Chapter I. Untyped Systems. The MIT Press, 2002.

[21]. Robert D. Tennent. The denotational semantics of programming languages. Commun. ACM 19, vol. 8 (August 1976), pp. 437–453, 1976.

[22]. Mandrykin M. U. Modeling memory of C programs in SMT-based static verification tools. PhD Thesis. ISP RAS, October 2016 (*In Russian*).

[23]. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar. Lazy abstraction. Symposium on Principles of Programming Languages, pp. 58–70, ACM Press, 2002.

[24]. Pavel Shved, Mikhail Mandrykin, Vadim Mutilin. Predicate Analysis with BLAST 2.7. Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems 2012 (TACAS'12), LNCS, vol. 7214,. pp. 525–527.

[25]. Pavel Shved, Vadim S. Mutilin, Mikhail U. Mandrykin. Experience of improving the Blast static verification tool. Programming and Computer Software, vol. 38, issue 3, pp. 134–142, June 2012. DOI: 10.1134/S0361768812030061..

[26]. Mutilin V. S. Linux driver verification using predicate abstraction. PhD Thesis. ISP RAS, November 2012 (*In Russian*).

[27]. Lars O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, University of Copenhagen, DIKU, 1994.
[28]. Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, Navindra Umanee. Points-to Analysis using BDDs. SIGPLAN Notes, vol. 38, no. 5, pp. 103–114. ACM. New York, NY, USA. May 2003.
[29]. Edsger W. Dijkstra, Carel S. Schölten. Predicate Calculus and Program Semantics. The strongest postcondition, pp. 209–215. Springer New York. 1990.
[30]. Alexey Khoroshilov, Vadim Mutilin, Eugene Novikov, Pavel Shved, Alexander Strakh. Towards An Open Framework for C Verification Tools Benchmarking. Proceedings of the 8th international conference on Perspectives of System Informatics (PSI'11), LNCS, vol. 7162, pp. 179–192, 2011.
[31]. Thomas Ball, Ella Bounimova, Vladimir Levin, Leonardo de Moura. Efficient evaluation of pointer predicates with Z3 SMT Solver in SLAM2. March 2010. MSR-TR-2010-24.
[32]. Thomas Ball, Ella Bounimova, Rahul Kumar, Vladimir Levin. SLAM2: Static driver verification with under 4% false alarms. Formal Methods in Computer-Aided Design (FMCAD), 2010, pp. 35–42.
[33]. Thomas Ball, Todd Millstein, Sriram K. Rajamani. Polymorphic Predicate Abstraction. ACM Trans. Program. Lang. Syst., vol. 27, no. 2, March 2005, pp. 314–343. ACM, New York, NY, USA. 2005.
[34]. Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, Abdullah Ustuner. Thorough Static Analysis of Device Drivers. Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006. (EuroSys '06) Leuven, Belgium, pp. 73–85. ACM. New York, NY, USA. 2006.
[35]. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs '09, pp. 23–42. Springer-Verlag. Berlin, Heidelberg. 2009.
[36]. Ernie Cohen, Michał Moskal, Stephan Tobies, Wolfram Schulte. A Precise Yet Efficient Memory Model For C. Electron. Notes Theor. Comput. Sci., October, 2009, vol. 254, pp. 85–103. Elsevier Science Publishers B. V., Amsterdam, The Netherlands. 2009.
[37]. Gordon D. Plotkin. A structural approach to operational semantics. Computer Science Department, Aarhus University. Aarhus, Denmark. Internal Report DAIMI FN-19. 1981.
[38]. M. U. Mandrykin, A. V. Khoroshilov. Region analysis for deductive verification of C programs. Programming and Computer Software, 2016, vol. 42, no. 5, pp. 257–278. DOI: 10.1134/S0361768816050042.
[39]. Rodney M. Burstall. Some techniques for proving correctness of programs which alter data structures. Machine intelligence, vol. 7, no. 3, pp. 23–50, 1972.
[40]. Richard Bornat. Proving Pointer Programs in Hoare Logic. In Proceedings of the 5th International Conference on Mathematics of Program Construction (MPC '00), pp. 102–126. Springer-Verlag. London, UK, 2000.
[41]. Thierry Hubert, Claude Marché. Separation Analysis for Deductive Verification. Heap Analysis and Verification (HAV'07), Braga, Portugal, March 2007, pp. 81–93.

[42]. Yannick Moy. Union and Cast in Deductive Verification. Proceedings of the C/C++ Verification Workshop. Technical Report ICIS-R07015, pp. 1–16. Radboud University Nijmegen, July, 2007.
[43]. M. U. Mandrykin, A. V. Khoroshilov. High-level memory model with low-level pointer cast support for Jessie intermediate language. Programming and Computer Software, 2015, vol. 41,. no. 4,. pp. 197–207. DOI: 10.1134/S0361768815040040.
[44]. Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, Boris Yakobowski. Frama-C: A Software Analysis Perspective. Proceedings of the 10th International Conference on Software Engineering and Formal Methods. SEFM'12. Thessaloniki, Greece. 2012. Lecture Notes in Computer Science, vol. 7504, pp. 233–247. Springer-Verlag. Berlin, Heidelberg. 2012.
[45]. Yannick Moy. Automatic Modular Static Safety Checking for C Programs. PhD thesis. Université Paris-Sud. January, 2009.
[46]. Jean-Christophe Filliâtre, Claude Marché. Multi-prover Verification of C Programs. In Proceedings of Formal Methods and Software Engineering: 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8–12, 2004. LNCS, vol. 3308, pp. 15–29. Springer Berlin Heidelberg. 2004.
[47]. Robin Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences, vol. 17, no. 3, pp. 348–375. December, 1978.
[48]. Jean-Christophe Filliâtre, Andrei Paskevich. Why3: Where Programs Meet Provers. Proceedings of the 22Nd European Conference on Programming Languages and Systems. ESOP'13. Rome, Italy. LNCS, vol. 7792, pp. 125–128. Springer-Verlag.
[49]. https://www.microsoft.com/en-us/research/project/havoc/
[50]. Shuvendu Lahiri, Shaz Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '08), pp. 171–182. ACM, New York, NY, USA. 2008.
[51]. Aharon Abadi, Alexander Rabinovich, Mooly Sagiv. Decidable fragments of many-sorted logic. Journal of Symbolic Computation, vol. 45, issue 2, pp. 153–172, February 2010.
[52]. Yeting Ge, Leonardo Moura. Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09). LNCS, vol. 5643, pp. 306–320. Springer-Verlag, Berlin, Heidelberg, 2009.