Модел. и анализ информ. систем. Т. **20**, № **6** (2013) 22–35

УДК 004.052.42+004.4'23

Обнаружение дефектов в программном обеспечении путем объединения ограниченной проверки моделей и аппроксимации функций[1]

Ахин М.Х., Беляев М.А., Ицыксон В.М.

Санкт-Петербургский государственный политехнический университет
194021, Россия, г. Санкт-Петербург, Политехническая ул., 21

e-mail: {akhin,belyaev}@kspt.icc.spbstu.ru, vlad@icc.spbstu.ru

получена 15 сентября 2013

Ключевые слова: ограниченная проверка моделей, контракт кода, интерполяция Крейга, SMT, обнаружение ошибок

В последние годы такой метод обеспечения качества программного обеспечения (ПО), как ограниченная проверка моделей (Bounded Model Checking, BMC), исследуется все более и более активно, поскольку он позволяет успешно обнаруживать как функциональные, так и нефункциональные дефекты в реальном ПО. В данной статье предлагается оригинальный подход к реализации BMC, основанный на объединении результатов нескольких последних исследований в этой области: использовании системы компиляции LLVM для разбора и трансформации исходного кода, применении SMT-решателя Z3 для проверки свойств корректности и повышения эффективности анализа за счет аппроксимаций функций. Проведенные экспериментальные исследования показывают применимость подхода к реальным проектам.

Введение

В настоящее время программное обеспечение (ПО) все шире используется в самых разных областях человеческой деятельности, начиная с атомных электростанций и самолетов и заканчивая «умными домами» и бытовой техникой. Ошибки в таком ПО могут привести к различного рода потерям и ущербу, поэтому задача верификации ПО с целью обнаружения подобных ошибок является крайне актуальной.

Одним из способов решения задачи верификации ПО является метод проверки моделей, в частности, метод ограниченной проверки моделей (Bounded Model Checking, BMC), которому уделяется все больше внимания в последние годы [1, 2, 3, 4]. Для поиска ошибок в методе проверки моделей используется рассмотрение

[1]Исследование выполнено при финансовой поддержке Министерства образования и науки Российской Федерации (№8.8491.2013) в рамках государственного заказа высшим учебным заведениям.

полного пространства состояний заданной программы с целью поиска состояний, нарушающих те или иные свойства корректности (например, утечки памяти или нарушения формальной спецификации). При всей полноте проведенных теоретических исследований в области метода проверки моделей его практическое применение серьезно ограничено рядом принципиальных сложностей. В данной работе рассматривается оригинальный подход к решению двух задач в рамках BMC для языков C/C++.

Построение модели ПО для BMC В данной работе для построения модели ПО используется схожий с LLBMC [4] подход, основывающийся не на исходном коде на C/C++, а на внутреннем представлении системы компиляции LLVM (LLVM Internal Representation, LLVM IR). Это значительно упрощает построение модели за счет большей структурированности LLVM IR и возможности использования уже реализованных в LLVM анализов и оптимизаций.

Межпроцедурный анализ В классическом BMC выполняется полная подстановка тел функций во все места вызовов, что приводит к значительному повышению сложности анализа для большинства реальных программ. Одним из способов решения данной проблемы является использование аппроксимаций функций — кратких описаний поведения функций, заменяющих собой полное тело функции в местах вызовов. В данной работе рассматриваются три типа аппроксимаций функций:

- полные спецификации поведения функции на специальном предметно-ориентированном языке (Domain-Specific Language, DSL) PanLang [5, 6];
- частичные контракты функций на ACSL-подобном языке [7];
- аппроксимации функций на основе интерполянтов Крейга [8].

На основе предложенных подходов был разработан прототип[2], опробованный на наборе тестовых BMC примеров. Результаты свидетельствуют о возможности успешного применения подхода на практике для анализа реального ПО.

Оставшаяся часть статьи организована следующим образом. В разделах 1 и 2 кратко рассматриваются используемые фреймворки и подходы. Наиболее интересные особенности практической реализации прототипа описываются в разделе 3. Результаты экспериментов и планы на дальнейшие исследования приведены в разделе 4.

1. Основы

В данном разделе рассматриваются методы и фреймворки, составляющие основу предлагаемого подхода: метод ограниченной проверки моделей, задача Satisfiability Modulo Theories и система компиляции LLVM.

[2]В настоящее время прототип поддерживает только аппроксимацию функций на основе частичных контрактов.

1.1. Метод ограниченной проверки моделей

Метод проверки моделей является одним из хорошо изученных способов верификации корректности различных свойств систем с конечным числом состояний. В его основе лежит рассмотрение полного пространства состояний системы с дальнейшим поиском нарушений заданных свойств корректности.

Основная проблема такого подхода заключается в том, что для любой реальной системы с большим числом возможных состояний рассмотрение полного пространства состояний является практически невозможным. Существуют несколько способов решения этой проблемы, такие как слайсинг, редукция частичных порядков, абстракция и другие. В данной работе рассматривается метод ограниченной проверки моделей (Bounded Model Checking, BMC, [9]).

BMC решает проблему взрыва пространства состояний путем ограничения длины рассматриваемых путей в пространстве состояний; для ПО это ограничение соответствует, например, анализу циклов на конечном числе итераций. После получения ограниченного пространства состояний оно представляется в виде формулы пропозициональной логики, которая затем комбинируется с условиями корректности программы[3] и проверяется на непротиворечивость при помощи SMT.

1.2. Satisfiability Modulo Theories

Задача SMT (Satisfiability Modulo Theories) — одна из классических задач математической логики [10], представляющая собой задачу разрешимости логической формулы в логике первого порядка при наличии в ней предикатов, использующих дополнительные теории. Широкое использование SMT в различных областях информатики связано с тем, что она является одной из наиболее изученных NP-полных задач[4] и, в то же время, достаточно удобной для формализации и сведения к ней других вычислительных задач.

Это привело к активной разработке различных средств решения задачи SMT — SMT решателей, — предоставляющих эффективные способы ее решения. Очевидно, что в общем случае решение задачи SMT за приемлемое время является невозможным, но результаты исследований последних лет говорят о возможности эффективного решения достаточно широкого класса практически интересных подмножеств SMT.

1.3. Система компиляции LLVM

Система компиляции LLVM [11] позиционируется как современная замена системы компиляции GCC. В основе LLVM лежит промежуточное представление кода (LLVM Intermediate Representation, LLVM IR), которое используется в качестве связующего звена между всеми анализами и оптимизациями внутри LLVM.

LLVM IR представляет собой типизированный мета-ассемблер, естественным образом обеспечивающий выполнение свойства статического однократного присваивания (Static Single Assignment, SSA, [12]) и поддерживающий как эффективную

[3]Например, «в программе не разыменовывается нулевой указатель».

[4]В действительности вычислительная сложность задачи SMT зависит от используемых теорий и может быть NP-трудной или неразрешимой для некоторых классов теорий.

трансляцию исходного кода в IR, так и компиляцию IR в машинный код. За счет того, что все преобразования в LLVM выполняются над LLVM IR, любой язык программирования, который может быть преобразован в LLVM IR, потенциально поддерживается LLVM.

Кроме IR, LLVM предоставляет удобный фреймворк для разработки различных анализов и оптимизаций кода, а также использования уже существующих средств. Он основывается на «проходах», оперирующих строго над LLVM IR, без доступа к исходному или машинному коду.

При реализации реального средства BMC для C/C++ желательно использовать максимально простую модель программы со свойством SSA, работающую с памятью при помощи компактного ядра инструкций и, в то же время, способную выразить полную семантику C/C++. LLVM IR полностью удовлетворяет данным требованиям, поэтому в настоящее время значительное количество средств анализа и верификации основываются на LLVM.

2. Метод ограниченной проверки моделей

Предлагаемый в данной статье подход основывается на объединении результатов двух недавних исследований в области BMC: анализа C/C++ программ на базе LLVM [4] и использования интерполяции Крейга в методе проверки моделей [8, 13]. Рассмотрим основные положения предлагаемого подхода и его сходства и различия с существующими работами.

2.1. LLBMC

LLBMC является относительно недавним дополнением в семействе средств BMC, таких как CBMC [1], SMT-CBMC [2] и ESBMC [3]. Как и прочие средства BMC, для борьбы с проблемой взрыва числа состояний он анализирует программу с ограничениями на число итераций циклов и/или глубину рекурсивных вызовов.

Основными особенностями LLBMC по сравнению с другими средствами BMC являются:

- LLBMC анализирует LLVM IR, что позволяет использовать все возможности системы LLVM для проведения предварительных оптимизаций и упрощений анализируемых программ. Кроме того, за счет использования компактного LLVM IR достигается значительное сокращение поверхности анализа[5].
- LLBMC моделирует память как линейный массив байт, то есть практически не отличается в этом от семантики модели памяти C/C++. Благодаря этому LLBMC способен успешно анализировать программы с большим числом низкоуровневых операций над памятью.

Предлагаемый подход, так же как и LLBMC, базируется на LLVM и аппроксимирует память в виде линейной последовательности байт.

Основным отличием является то, что вместо подстановки тел функций в места вызовов предлагается для межпроцедурного анализа использовать аппроксимации

[5] Число семантически различных анализируемых операций.

функций. Результаты недавних исследований свидетельствуют о том, что использование аппроксимаций может значительно повысить эффективность анализа без потери полноты и точности [8].

2.2. Аппроксимация функций на основе интерполяции Крейга

В случае если пользователь не предоставил никакой информации о поведении функции, для аппроксимации функции используется интерполяция Крейга[6] [14]. Полученный интерполянт представляет собой аппроксимацию поведения функции с точки зрения ее входов и выходов, что позволяет анализировать каждую функцию только один раз, тем самым значительно сокращая вычислительную сложность анализа.

Основная проблема при использовании интерполяции Крейга для получения аппроксимаций функций заключается в правильном выборе целевой формулы B для проведения интерполяции. В случае, если анализируется такое простое свойство, как «функция возвращает положительное число», очевидно, что $B = \backslash return > 0$; однако для более сложных случаев построение B уже не является столь тривиальным (как, например, для свойства «вызов функции не приводит к утечке памяти» при наличии глобальных переменных).

2.3. Аппроксимация функций на основе пользовательских спецификаций

Еще одним отличием предлагаемого подхода от LLBMC является поддержка пользовательских спецификаций — аппроксимаций функций, задаваемых непосредственно пользователем. В зависимости от полноты спецификаций, они могут как полностью заменять собой аппроксимации на основе интерполяции, так и использоваться для их усиления.

Для задания полных спецификаций поведения функций используется специально спроектированный язык PanLang [5, 6]; пример спецификации на PanLang приведен ниже:

```
function void* malloc(unsigned size): {
    if (size < 0) {
        action defect;
        Result = 0;
    } else if (size == 0) {
        Result = 0;
    } else {
        Result = new Heap(size);
    }
};
```

[6]Интерполянтом Крейга для формулы $A \to B$ является формула I, такая что $A \to I, I \to B$ и I содержит только общие между A and B символы.

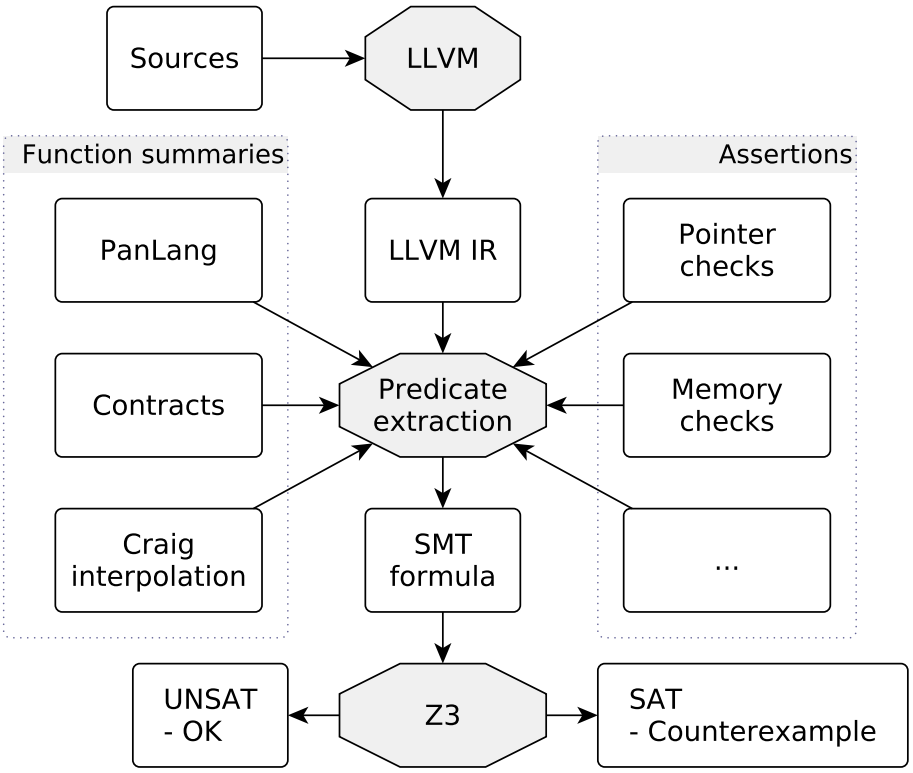

Рис. 1. Общая архитектура прототипа

Частичные спецификации или контракты функций описываются на ACSL-подобном языке [7] в виде специальных комментариев:

```
\\ @requires size > 0
\\ @ensures \result != \nullptr
void* safe_malloc(size_t size) ...
```

Данные спецификации в процессе анализа добавляются в построенную модель, за счет чего достигается увеличение точности и полноты анализа. Кроме того, это позволяет анализировать проекты, использующие сторонние библиотеки без исходного кода, за счет их аннотирования при помощи языка PanLang.

3. Особенности реализации

На основе предложенного подхода была разработана программа-прототип. Для разбора исходного кода используется компилятор Clang, для проведения анализа и трансформации кода — инфраструктура системы LLVM, а в качестве средства логического вывода — SMT-решатель Z3[7] [15]. Общая архитектура прототипа приведена на рис. 1.

[7]Z3 был выбран на основе результатов ежегодного соревнования SMT-COMP [16, 17].

3.1. Извлечение логических предикатов

Анализируемая программа преобразуется в промежуточное представление LLVM IR. Для сохранения дополнительной семантики, необходимой для анализа, в данном представлении используются специальные псевдо-функции. Например, функция `anno` используется для сохранения в коде информации о контрактах кода из исходных файлов. Вызовы данных функций обрабатываются отдельно от прочих вызовов в процессе анализа. Такое представление позволяет гарантировать, что уже присутствующие средства в системе LLVM не удалят необходимую для анализа информацию в ходе оптимизаций кода.

После описанного преобразования для каждой функции производится извлечение логических предикатов, описывающих ее поведение, которые затем комбинируются с уже имеющимися аппроксимациями для этой функции. Дополнительно во множество предикатов вставляется набор проверок для поиска дефектов[8]. Результат в виде формулы SMT подается на вход SMT-решателю Z3 для верификации.

```
// @requires size > 0
// @ensures \result != \nullptr
void* safe_malloc(unsigned size) {
    void* res = malloc(size);
    if (res) return res;
    else exit(-1);
}
```

Рис. 2. Пример на языке C для функции `safe_malloc`

```
define i8* @safe_malloc(i32 %size) {
entry:
  call void @"anno"(
    c"@requires (size > 0)")
  %0 = call i8* @"malloc"(i64 2048)
  %c = icmp eq i8* %0, null
  br i1 %c, label %if.e, label %if.t

if.t:
  call void @"anno"(
    c"@ensures (result != 0)")
  ret i8* %0

if.e:
  call void @exit(i32 -1)
  unreachable
}
```

Рис. 3. Пример в LLVM IR для функции `safe_malloc`

Проиллюстрируем подход на примере простой функции `safe_malloc` (см. рис. 2 и 3). Каждая инструкция LLVM IR преобразуется в отдельный предикат, терминирующие инструкции[9] преобразуются в набор предикатов, соответствующий определенным путям выполнения программы. В текущем варианте прототипа каждый путь анализируется отдельно, то есть к окончанию извлечения предикатов каждой инструкции соответствует свой набор возможных состояний предикатов. Состояния предикатов для инструкций `ret %0` и `call @exit` для примера с функцией `safe_malloc` приведены на рис. 4 и 5 соответственно.

[8]На данный момент поддерживаются только проверки указателей и контрактов кода.

[9]Инструкции, определяющие, какой базовый блок следует выполнять следующим.

```
(
  @R (size > 0) is true,
  %0 is malloc(2048),
  c is (%0 == <null>),
  @P c is false,
  \result_safe_malloc is %0
)
```

Рис. 4. Состояние предикатов для инструкции `ret %0`

```
(
  @R (size > 0) is true,
  %0 is malloc(2048),
  c is (%0 == <null>),
  @P c is true
)
```

Рис. 5. Состояние предикатов для инструкции `call @exit`

После выделения всех состояний предикатов над ними выполняется набор проверок с целью верификации корректности программы. Например, в инструкции `ret %0` необходимо проверить, выполняется ли контракт $@ensures\ \backslash result \neq \backslash nullptr$. Данный контракт выполняется, если $\backslash result \neq \backslash nullptr$ всегда верно в соответствующем состоянии (состояниях) предикатов. Для этого используется классический способ представления подобных задач в виде задачи SMT: формула всегда верна, если не существует таких значений переменных, для которых она не верна. Таким образом, необходимо проверить свойство SAT выражения $\backslash result = \backslash nullptr$ в заданном состоянии (состояниях) предикатов.

Если выражение невыполнимо (обладает свойством UNSAT), исходная формула всегда истинна, то есть соответствующий контракт $@ensures$ всегда выполняется и ошибки нет. Если же Z3 удалось найти такой набор значений переменных, для которого выражение истинно (обладает свойством SAT), возможно нарушение контракта, которое приводится к изначальной формуле и возвращается пользователю в виде ошибки.

3.2. Моделирование C/C++ в SMT

В рамках SMT существует несколько принципиальных проблем с моделированием кода на C/C++ даже в случае использования только внутрипроцедурного анализа. Наиболее важными из них являются моделирование памяти и обработка циклов. Рассмотрим то, как данные проблемы решаются в рамках предложенного подхода и прототипа.

3.2.1. Моделирование памяти в SMT

Наиболее сложной частью реализации анализа языков C и C++ традиционно являются особенности этих языков, связанные с наличием указателей и прямым доступом к памяти программы. Кроме того, с данными особенностями связано наибольшее количество реальных дефектов в этих языках [18], поэтому при анализе реального ПО им следует уделять особое внимание.

Можно выделить две основные проблемы работы с указателями: анализ указателей, который является чрезвычайно сложным (NP-полным, если быть точным) и арифметика над указателями, которая еще больше усложняет ситуацию. Так как BMC занимается анализом ограниченных выполнений программы, первая проблема

становится разрешимой[10], в то время как арифметика указателей требует правильного моделирования памяти.

В описываемом подходе память представляется как последовательный набор состояний для каждого пути выполнения программы. Каждое состояние памяти представляет собой функциональный массив[11]; нулевой индекс в данном массиве используется для представления нулевых указателей, а последний — для представления некорректных указателей. Функциональные массивы в рамках теорий SMT можно представить тремя различными способами: с использованием теории неинтерпретированных функций (UF), теории массивов и лямбда-выражений (анонимных функций) в рамках самого средства анализа. Описываемый прототип реализует все три подхода; проведенные эксперименты показывают, что наилучшие и наиболее стабильные результаты показывает подход на основе теорий массивов.

Другим аспектом моделирования памяти в виде массивов является то, как представляются отдельные элементы памяти через элементы массива. Можно использовать два основных подхода: унифицированный (каждому значению в памяти соответствует один элемент массива) и байтовый (массив представляет собой массив байт; представление значений в памяти соответствует представлению значений в памяти в машинном коде). Подход на основе байт более точен, но, в то же время, значительно увеличивает сложность итоговой задачи SMT и соответственно время ее решения. Унифицированный подход более производителен и приводит к неточностям в анализе только в случаях, когда машинное представление данных действительно важно (например, при бинарной сериализации структур данных). Представленный прототип поддерживает оба способа кодирования, используя по умолчанию более производительное унифицированное представление.

Последним важным аспектом моделирования памяти является представление ее начального состояния. Для этой цели можно использовать либо неинициализированный (пустой) массив данных, либо массив, заполненный определенными значениями (например, некорректными указателями). Первый позволяет более точно моделировать память в функциях, которые ожидают определенные значения в определенных участках памяти (как, например, функции, принимающие указатели в качестве аргументов), в то время как второй на таких функциях показывает худшие результаты, но более прост и производителен в общем случае.

3.2.2. Обработка циклов

Работа с циклами является одной из сложнейших проблем для любого анализа программ. Существует несколько известных путей решения этой проблемы. Во-первых, можно заниматься поиском вариантов и инвариантов цикла и использовать их для вывода числа итераций. Во-вторых, подход классического BMC [9] состоит в том, чтобы итеративно анализировать программу с циклами, увеличивая число итераций до тех пор, пока очередное такое увеличение не станет невозможным. Наконец, циклы могут быть развернуты частично или полностью. Последний путь обладает тем недостатком, что может в части случаев менять семантику программы неопреде-

[10]Это справедливо только для внутрипроцедурного анализа, указатели на глобальную память остаются проблемой и для BMC.

[11]Объект, над которым можно проводить операции «load» (чтение) и «store» (запись).

ленным образом, но аккуратный подбор коэффициента развертки цикла позволяет уменьшить данный негативный эффект.

В данной работе используется развертка циклов как самый простой и, в то же время, эффективный способ работы с циклами. Для осуществления развертки циклов был создан отдельный проход LLVM, поскольку стандартный проход для развертки циклов работает для весьма ограниченного набора видов циклов. Кроме того, это позволяет вручную задавать коэффициент развертки там, где это необходимо, посредством специальной разновидности аннотаций кода.

3.3. Ограничения прототипа

Текущая реализация прототипа не имеет полной поддержки межпроцедурного анализа. Из трех разновидностей аппроксимаций функций в данный момент поддерживаются только контракты кода. Спецификации на языке PanLang и на основе интерполянтов Крейга в данный момент разрабатываются в рамках отдельных проектов и еще не интегрированы в прототип.

Прототип также не поддерживает такие возможности языков C/C++, как указатели на функции, полноценная арифметика чисел с плавающей точкой и объединения. В дальнейшем планируется внимательно изучить то, каким образом данные возможности и их поддержка в разрабатываемом средстве влияют на характеристики анализа на основе BMC.

4. Экспериментальные исследования

В данном разделе представлены результаты предварительных экспериментов. Приводится сравнение производительности различных аспектов представления памяти на наборе тестовых программ.

4.1. Набор тестов

Тестовый набор состоит из 78 примеров различной сложности на языке C. Для каждого примера также имеется описание присутствующих в программах дефектов и соответствующих им фрагментов кода. Данные примеры созданы с целью тестирования прототипа с поправкой на его особенности и содержат только те дефекты, обнаружение которых входит в его спектр возможностей.

Набор тестовых примеров может быть разделен на две большие части. Первая из них содержит примеры, созданные в процессе разработки прототипа и представляет собой набор тех случаев, которые вызывали у прототипа различные проблемы на разных этапах его разработки (некорректные результаты анализа или некорректное поведение прототипа в целом). Вторая часть основана на стандартном наборе тестовых примеров NECLA [19], созданном в лабораториях компании NEC.

Наш прототип не способен анализировать все тесты NECLA в связи с тем, что набор обнаруживаемых дефектов и возможности межпроцедурного анализа в нем ограничены. Тем не менее, после введения в соответствующие примеры аппроксимаций функций на основе контрактов кода большая часть примеров была успешно верифицирована.

4.2. Сравнение моделей памяти

Как уже было упомянуто выше, производительность описываемого подхода принципиальным образом зависит от используемого представления памяти. Тестирование прототипа производилось в нескольких режимах, результаты сравнения приведены в табл. 1. Строки таблицы соответствуют различным вариантам представления самой памяти (UF, AT, Lambda-based) и ее начального состояния (Fixed, Uninit). Кроме статистики для полного набора из 78 тестов (столбец *Полный*), в таблице приведена отдельная статистика для 15 примеров, отличающихся активной работой с памятью и/или межпроцедурностью (столбец *Специальный*).

	Полный, s	**Специальный, s**
UF/Fixed	23.263	11.945
AT/Fixed	19.456	8.965
Lam/Fixed	20.467	10.027
UF/Uninit	33.590	21.486
AT/Uninit	29.286	17.890
Lam/Uninit	30.755	19.168

Таблица 1. Результаты сравнения различных представлений памяти

Эксперименты показали, что производительность различных представлений памяти зависит от того, содержит ли анализируемая программа ошибки (и, как следствие, обрабатывает ли SMT-решатель случаи SAT или UNSAT). Подходы на основе лямбда-выражений (lambda-based) и неинтерпретированных функций (UF) показывают достаточно близкие показатели производительности, несколько превосходящие производительность подхода на основе теории массивов (AT) на случае SAT и отстающие от нее на случае UNSAT. Так как тестовый набор содержит относительно малое количество ошибок, результаты на случае SAT оказали незначительное влияние на интегральные показатели, приведенные в таблице.

Представление начального состояния памяти на основе неинициализированных переменных ухудшает производительность во всех случаях, но необходимо учесть, что такое представление улучшает другие показатели анализа. Например, в случае, когда поведение функции зависит от состояния памяти в момент вызова, такое представление позволяет обнаруживать больше ошибок, чем представление на основе фиксированной начальной памяти. Из 78 тестовых примеров в трех зафиксировано пропадание обнаруживаемого истинного дефекта при представлении памяти с фиксированным начальным состоянием.

К сожалению, результаты сравнения унифицированного представления памяти с представлением на основе полной симуляции байтовых массивов не представлены в таблице, поскольку на некоторых примерах симуляция байтовых массивов показывает крайне низкую производительность (для измерения которой нужно задействовать отдельные технические средства). На подмножестве тестовых примеров, для которых использование байтового представления является возможным, сравнительные результаты не отличаются качественно от соответствующих результатов для унифицированного представления.

Заключение

В данной статье представлен подход к решению задачи BMC, основанный на комбинации трех компонентов: LLVM, SMT и аппроксимации функций. Это позволяет предложенному подходу успешно справляться с такими сложностями BMC, как излишняя сложность семантики C/C++ или взрыв пространства состояний при межпроцедурном анализе.

Реализованный на основе описанного подхода прототип был проверен на наборе тестовых примеров. Результаты свидетельствуют о возможности использования подхода для анализа реальных программ на C/C++ с приемлемыми затратами времени и ресурсов.

В дальнейшем планируется расширить функциональность прототипа: реализовать более полную поддержку всех трех типов аппроксимаций функций, оптимизировать взаимодействие с SMT-решателем Z3, обеспечить поддержку всего спектра возможностей языков C/C++ и провести более широкие экспериментальные исследования.

Список литературы

1. *Clarke E., Kroening D., Lerda F.* A Tool for Checking ANSI-C Programs // TACAS '04. Springer-Verlag, 2004. P. 168–176.

2. *Armando A., Mantovani J., Platania L.* Bounded Model Checking of Software using SMT Solvers instead of SAT Solvers // Int. J. Softw. Tools Technol. Transf. Springer-Verlag, 2009. 11, 1. P. 69–83.

3. *Cordeiro L., Fischer B., Marques-Silva J.* SMT-Based Bounded Model Checking for Embedded ANSI-C Software // ASE '09. IEEE. 2009. P. 137–148.

4. *Merz F., Falke S., Sinz C.* LLBMC: Bounded Model Checking of C and C++ Programs using a Compiler IR // VSTTE '12. Springer-Verlag, 2012. P. 146–161.

5. *Ицыксон В.М., Глухих М.И.* Язык спецификаций поведения программных компонентов // Научно-технические ведомости СПбГПУ. Информатика. Телекоммуникации. Управление. 3. СПб: СПбГПУ, 2010. С. 63–71. (*Itsykson V., Glukhikh M.* Yazyk specifikaciy povedeniya programmnyh komponentov // Nauchno-tehnicheskie vedomosti SPbGPU. Informatika. Telekommunikacii. Upravlenie. 3. SPb: SPbGPU, 2010. P. 63–71 [in Russian]).

6. *Ицыксон В. М., Зозуля А. В.* Автоматизированная трансформация программ при миграции на новые библиотеки // Программная инженерия. 6. М: Новые Технологии, 2012 С. 8–14. (English transl.: *Itsykson V., Zozulya A.* Automated Program Transformation for Migration to New Libraries // SECR '11. IEEE, 2011. P. 1–7).

7. *Baudin P., Filliâtre J.C., Hubert T., Marché C., Monate B., Moy Y., Prevosto V.* ACSL: ANSI/ISO C Specification Language. Preliminary Design, Version 1.4, 2008. `http://www.frama-c.cea.fr/download/acsl_1.4.pdf`

8. *Sery O., Fedyukovich G., Sharygina N.* Interpolation-Based Function Summaries in Bounded Model Checking // HVC '11. Springer-Verlag, 2011. P. 160–175.

9. *Biere A., Cimatti A., Clarke E. M., Zhu Y.* Symbolic Model Checking without BDDs // TACAS '99. Springer-Verlag, 1999. P. 193–207.

10. *Barrett C., Sebastiani R., Seshia S.A., Tinelli C.* Satisfiability Modulo Theories // Handbook of Satisfiability. 185. Frontiers in Artificial Intelligence and Applications, 2009. P. 825–885.

11. *Lattner C., Adve V.* LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation // CGO '04. IEEE, 2004. P. 75–86.

12. *Cytron R., Ferrante J., Rosen B.K., Wegman M.N., Zadeck F.K.* Efficiently Computing Static Single Assignment Form and the Control Dependence Graph // ACM TOPLAS. 1991. 13, 4. P. 451–490.

13. *McMillan K.L.* Applications of Craig Interpolants in Model Checking // TACAS '05. Springer-Verlag, 2005. P. 1–12.

14. *Craig W.* Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory // The Journal of Symbolic Logic. 22, 3. Association for Symbolic Logic, 1957. P. 269–285.

15. *De Moura L., Bjørner N.* Z3: An Efficient SMT Solver // TACAS '08. Springer-Verlag, 2008. P. 337–340.

16. *Barrett C., De Moura L., Stump A.* SMT-COMP: Satisfiability Modulo Theories Competition // CAV '05. Springer-Verlag, 2005. P. 503–516.

17. *Cok D.R., Griggio A., Bruttomesso R.* The 2012 SMT Competition. 2012. `http://smtcomp.sourceforge.net/2012/`

18. *Coverity.* Open Source Report 2011. `http://www.coverity.com/library/pdf/coverity-scan-2011-open-source-integrity-report.pdf`

19. *NEC Laboratories.* NECLA Static Analysis Benchmarks. 2013. `http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php`

Defect Detection: Combining Bounded Model Checking and Code Contracts

Marat Akhin, Mikhail Belyaev, Vladimir Itsykson

Saint-Petersburg State Polytechnical University
Polytechnicheskaya street, 21, Saint-Petersburg 194021 Russia

Keywords: bounded model checking, satisfiability modulo theories, LLVM, function contracts, function summaries

Bounded model checking (BMC) of C/C++ programs is a matter of scientific enquiry that attracts great attention in the last few years. In this paper, we present our approach to this problem. It is based on combining several recent results in BMC, namely, the use of LLVM as a baseline for model generation, employment of high-performance Z3 SMT solver to do the formula heavy-lifting, and the use of various function summaries to improve analysis efficiency and expressive power. We have implemented a basic prototype; experiment results on a set of simple test BMC problems are satisfactory.

Сведения об авторах:
Ахин Марат Халимович,
Санкт-Петербургский государственный политехнический университет, аспирант;
Беляев Михаил Анатольевич,
Санкт-Петербургский государственный политехнический университет, аспирант;
Ицыксон Владимир Михайлович,
Санкт-Петербургский государственный политехнический университет, доцент.