УДК 004.3+519.6 DOI: 10.18698/0236-3933-2016-4-49-64

РАЗРАБОТКА И ИССЛЕДОВАНИЕ СИНТЕТИЧЕСКОГО МЕТОДА ВЕРИФИКАЦИИ ПРОГРАММЫ С ПОМОЩЬЮ SMT-РЕШАТЕЛЕЙ

И.В. Рудаков irudakov@yandex.ru
Р.Е. Гурин rg.bmstu@gmail.com

МГТУ им. Н.Э. Баумана, Москва, Российская Федерация

Аннотация

Рассмотрена проблема разработки и исследования методов поиска недетерминированного поведения программного обеспечения. Методы верификации программного обеспечения предназначены для подтверждения фактов соответствия конечного программного продукта заявленным требованиям, целью верификации программного обеспечения является обнаружение ошибок, уязвимостей, некорректно реализованных свойств и требований. Существующие методы верификации имеют ряд проблем и недостатков. Предложен синтетический метод верификации программного обеспечения на основе SMT-решателя, позволяющий решить проблему комбинаторного взрыва, охватывающий все заданные свойства проверяемой программы, не требующий построения сложной модели программы. Получен алгоритм работы метода верификации и его реализации на языке SMT–LIB

Ключевые слова

Верификация, анализ кода, статический анализ, динамический анализ, интерпретация, символьное выполнение, проверка модели

Поступила в редакцию 09.09.2015
© МГТУ им. Н.Э. Баумана, 2016

Основным методом верификации программного обеспечения (ПО) в настоящее время является тестирование. При тестировании ПО на вход тестовой программы подается подготовленный пользователем набор данных, далее проверяется, соответствует ли выданный программой результат ожидаемому [1]. Одним из важных свойств тестирования является возможность выполнять помимо статической проверки кода еще и динамическую.

Исторически первым методом проверки ПО на наличие ошибок и уязвимостей стала экспертиза (review). Экспертиза может быть применима к любым свойствам ПО на любом этапе проекта и позволяет выявить практически любые виды ошибок и уязвимостей, это минимизирует число ошибок и их последствия в конечном программном продукте. Одним из минусов этого метода является невозможность его автоматизации и активное участие людей при верификации ПО. В связи с этим наряду с существующими методами верификации, особое внимание уделяется автоматизированным методам верификации ПО.

Существующие методы верификации программ, использующие доказательство теорем (теоретико-доказательный подход), основаны на методах доказательства теорем, например, группа средств «theoremprover», где описание и спецификация программы являются ее представлением в виде множества логических утверждений. Несмотря на значительный прогресс в области верификации ПО методами теоретико-доказательного подхода, верификация большинства программ требует участия эксперта.

Методы проверки моделей (ModelChecking (MC)) позволяют осуществить проверку свойств ПО с помощью построения и анализа формальной модели программы. Как правило, в качестве модели используется модель Крипке [2], которая формально задается следующим образом: $M=(S,S_0,R,L)$, где S — множество состояний; S_0 — множество начальных состояний; $R\subseteq S\cdot S$ — отношение переходов; $L:D\rightarrow 2^{AP}$ — функция разметки. Модель программы представляет собой систему переходов, включающую в себя множество состояний модели, множество ее начальных состояний и отношения переходов на множестве состояний. Каждое состояние модели определяется значениями булевых переменных модели. Спецификация программы задается с помощью формул темпоральной логики, которые в отличие от формул классической логики, где истинность не зависит от времени, учитывают зависимость истинности формулы от момента времени. Это позволяет описать свойства поведения программы, реакции процессов на события, связь между процессами и событиями и т. д. Задача проверки выполнимости темпоральных формул является разрешимой, так как состояние модели конечно.

Методы MC имеют ряд недостатков, что ограничивает их применение:

- невозможно определить, охватывает ли заданная спецификация все свойства, которым должна удовлетворять система;
- сложность построения наиболее полной модели программы;
- наличие эффекта комбинаторного взрыва;
- требование конечного числа состояния модели.

Модель программы строится для проверки соответствия программы заданному набору спецификаций. Поэтому модель должна отражать все свойства программы, которые необходимы для проверки спецификации. Модель должна соответствовать поведению исходной программы. Адекватность модели проверяемому набору спецификаций зависит от того, какие свойства программы были перенесены в модель. Существуют методы автоматического создания моделей [3] по заданной программе. Подход Counter Example Guided Abstraction Refinement осуществляет построение модели за счет комбинирования методов булевой абстракции с итеративной верификацией моделей [4–6] .

Эффект комбинаторного взрыва появляется в результате экспоненциального роста пространства состояний при линейном росте числа взаимодействующих процессов. Алгоритмы MC не являются полиноминальными, время решения задачи верификации не ограничено никаким многочленом от длины выхода. Задачи верификации MC имеют экспоненциальную сложность, что при

возрастающей сложности алгоритма и экспоненциальный рост пространства состояний приводят к более высоким требованиям к объему памяти и значительно увеличивают время проверки программы.

Алгоритмы МС эффективно работают и гарантированно завершаются только на моделях с конечным числом состояний. При верификации моделей с бесконечным числом состояний используют два направления: верификация моделей с бесконечным числом состояний и верификация параметризированных моделей.

Поскольку большинство используемых методов и программных средств для верификаций ПО требуют участия экспертов, которые сталкиваются с проблемой комбинаторного взрыва, задача реализации метода верификации, минимизирующего эффект комбинаторного взрыва, а также оценка его параметров и генерация соответствующих тестовых случаев являются актуальными.

Цель настоящей работы — разработка и исследование алгоритма верификации ПО, решающего проблему комбинаторного взрыва.

Существующие методы и алгоритмы, которые позволяют минимизировать последствия комбинаторного взрыва при верификации ПО. Эффект комбинаторного взрыва связан с экспоненциальным ростом пространства состояний при линейном росте числа взаимодействующих процессов. Последствиями экспоненциального роста состояний являются значительно возросшие требования к объему памяти, используемой верификатором, а также возрастает время проверки модели. Для сжатия верифицируемого множества состояний используют символьные способы представления модели программы, такие как двоичные решающие диаграммы BDD (Binary Decision Diagrams) [7]. В этом случае описываемые объекты (множества, функции, матрицы) кодируются графовой структурой в соответствии с определенными правилами. Часто такое представление позволяет добиться того, чтобы размер графа рос гораздо медленнее, чем размер соответствующего описания. Кроме того, операции над представленными объектами, выполняются за полиномиальное время от размера соответствующих графов. Алгоритмы BDD могут быть значительно эффективнее простого перебора состояний модели программы в том случае, когда бинарное решающее дерево остается компактным.

Для борьбы с комбинаторным взрывом при верификации моделей программ используются методы редукции частичных порядков [8 – 10]. В этих методах проверяется достаточное подмножество множества всех трасс. Такое подмножество вычисляется на основании зависимостей между переходами процессов модели. Типы проверяемых зависимостей выявляются эмпирически. Верный выбор правил редукции может существенно ускорить алгоритм верификации моделей.

Бинарные решающие диаграммы. Диаграммы BDD являются экономной формулой представления булевых функций в виде ацикличного ориентированного графа.

Бинарная решающая диаграмма представляет собой функцию вида *f*: {0, 1} $n \to S$, которая является ориентированным корневым ациклическим графом с множеством вершин $V = T \cup N$, $T \cap N = \varnothing$. Вершины множества *N* называются

нетерминалами и для каждой такой вершины $v \in N$ определены значение порядка $index(v) \in \{1, \ldots, n\}$ и ровно две дочерние вершины $low(v)$, $high(v) \in V$. Индексы нетерминальных вершин *i* соответствуют аргументам определяемой функции.

Вершины множества T называются терминалами и не имеют дочерних вершин. Для каждой терминальной вершины $v \in T$ определено значение $value(v) \in S$ и выполнено условие: для любого нетерминала $v \in N$ и любой его дочерней вершины $v`$ либо $v` \in T$, либо $index(v) < index(v`)$. Теперь каждой вершине $v \in V$ можно сопоставить функцию $f_v: \{0, 1\{0,1\}^n \to S$:

$$f_v\left(x_1, \ldots, x_n\right) = \begin{cases} value\ (v),\ v \in T; \\ f_{high\,(v)}\left(x_1, \ldots, x_n\right),\ x_{index(v)} = 1,\ v \in N; \\ f_{low\,(v)}\left(x_1, \ldots, x_n\right),\ x_{index(v)} = 0,\ v \in N. \end{cases}$$

Существует несколько разновидностей решающих диаграмм. Для задач, которые связаны с использованием булевых функций вида $f_v: \{0, 1\}^n \to \{0, 1\}$ и конечных множеств, широко используются BDD [10]. Для задач, связанных с использованием функций вида $f_v: \{0, 1\}^n \to S$ (где S — некоторое конечное непустое множество) и для нечетких множеств применяются многотерминальные бинарные решающие диаграммы (MTBDD), а также их модификации. Многокор-невые бинарные решающие диаграммы (MRBDD) являются альтернативой MTBDD, которые работают с конечнозначимыми функциями, как с векторами из булевых функций. Основным преимуществом MRBDD перед MTBDD является меньший размер диаграммы. Это свойство достигается за счет более эффективного повторного использования фрагментов одинаковой структуры. Рассмотрим пример. Далее приведена таблица истинности функции *f* (табл. 1), а на рис. 1 — ее представление в виде бинарного дерева решений.

Таблица 1

Таблица истинности функции *f*

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Проведем сжатие бинарного представления системы в соответствии с тремя правилами:

- слияние дубликатов терминалов с соответствующим перенаправлением дуг;
- слияние дубликатов нетерминалов, т. е. если нетерминальные вершины $u, v \in N$ такие, что $index(u) = index(v)$, $low(u) = low(v)$, $high\,(u) = high\,(v)$, то вершины u и v совмещаются с соответствующим перенаправлением дуг;
- удаление нетерминалов с одной дочерней вершиной с перенаправлением в нее входящих дуг.

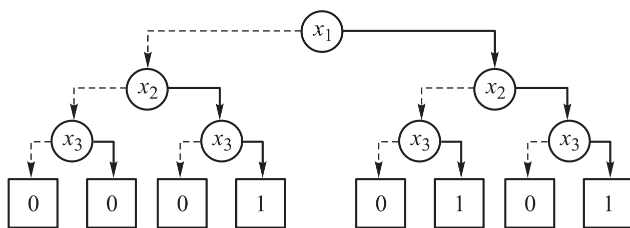

Рис. 1. Задание функции и ее бинарное решающее дерево

Третий вариант диаграммы является конечным для функции f. Такое представление является более компактным, чем таблица истинности и ее бинарное дерево (рис. 2).

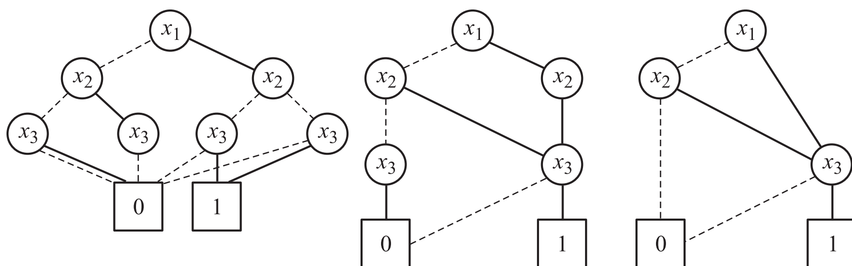

Рис. 2. BDD после первого (слева), второго (в центре), третьего (справа) правил сжатия

При применении бинарных решающих диаграмм становится возможным решать проблемы, которые при традиционном представлении структур неразрешимы.

Такие логические операции (конъюнкция, дизъюнкция, отрицание) могут быть проведены непосредственно над BDD с помощью алгоритмов, выполняющих манипуляции над графами за полиномиальное время. Однако повторение этих операций множество раз, например, при формировании конъюнкций или дизъюнкций набора, могут привести к экспоненциально большому BDD в худшем случае. Это происходит из-за того, что результатом любых предшествующих операций над двумя BDD в общем случае может быть BDD с размером, пропорциональным произведению предшествующих размеров, поэтому для нескольких BDD размер может увеличиваться экспоненциально.

Методы оптимизации процесса верификации моделей большого размера. Для верификации моделей большего размера реализован ряд алгоритмов оптимизации, которые соответственно их методике подразделяются на две различные группы:

- оптимизации, позволяющие сократить число достижимых состояний модели за счет редукции частичных порядков и объединения операторов;
- оптимизации, позволяющие уменьшить объем памяти, занимаемой векторами состояний.

Метод редукции частичных порядков. Редукция частичных порядков позволяет вместо всех полученных на основе интерливинга вариантов выпол-

нения операторов, которые эквиваленты с точки зрения результата, анализировать только один. Варианты выполнения статически выбираются до начала исполнения программы.

Техника редукции частичных порядков позволяет сократить на 10…90 % объем используемой памяти и время выполнения программы верификации [11].

Метод collapse. Глобальное состояние модели задается определенной комбинацией состояний отдельных ее процессов и содержимого ее каналов. Каждое глобальное состояние повторяет полное описание комбинации состояний процессов. Данный вид представления является неэффективным с точки зрения траты машинных ресурсов.

Метод collapse [12] позволяет разделить информацию о состояниях на две составляющие:

- глобальные данные модели, включая содержимое всех каналов;
- управляющую информацию и состояние локальных переменных, исключая содержимое каналов, задаваемых для каждого процесса.

Составляющие хранятся отдельно, а в вектор состояния включаются индексы, один из которых назначен для хранения глобальных данных и содержимого каналов, а остальные — для процессов.

Метод collapse позволяет достичь уменьшения на 60…80 % объема используемой памяти при небольшом увеличении времени выполнения программы верификации (10…20 %) [12].

Разработка синтетического метода верификации ПО. В научной литературе встречается множество описаний различных методов верификации ПО, отметим, что характерной проблемой методов верификации является проблема комбинаторного взрыва. Главная идея разрабатываемого метода — отход от привычного бинарного представления пространства состояний к логическому. Инструменты на основе Satisfiability Modulo Theories (SMT) решателей позволяют сделать это (рис. 3), поскольку в основе SMT-решателя лежат формальные методы такие, как доказательство теорем и дедуктивный анализ.

Рис. 3. Упрощенное представление алгоритма верификации

Разрабатываемый метод верификации ПО обладает следующими характеристиками:

- осуществляет верификацию программ, написанных на языке Си;
- использует синтетический метод верификации — осуществляет верификацию кода программы с помощью методов статического анализа и методов формальной верификации;
- допускает автоматическое и ручное создание тестовых случаев;

- позволяет отказаться от бинарного представления состояний программы;
- выполняет построение и верификацию инварианта программы;
- осуществляет поиск синтаксических и семантических ошибок (ошибки компоновки, ошибки выполнения, логические и динамические ошибки);
- осуществляет покрытие условий, путей и функций программы;
- формирует контрпримеры, тем самым выполняя поиск модели данных, на которых отдельная функция или программа становятся невыполнимыми.

Лексический транслятор. Лексический транслятор (рис. 4) представляет собой средство, объединяющее трансляцию во внутреннее представление компилятора и программу, которая осуществляет построение суммы предикатов по всем состояниям программы.

Рис. 4. Лексический транслятор

LLVMPass позволяет построить более упрощенное представление программы, разбивая сложные условия или состояния программы на их более простые составляющие, тем самым упрощая построение предикатов состояний системы и формирование проверяемого инварианта программы.

Построение инварианта по внутреннему представлению программы. Преобразование внутреннего представления кода в формулы логики первого порядка. Инвариантом программы является сумма предикатов состояний программы. Предикат задает множество переходов программы при произвольных значениях переменных. Состояние программы — это мгновенный снимок, задающий множество значений всех переменных программы и значения счетчика команд pc.

Состояние программы выражается следующим образом:

$$s \in (PC, V), \quad s = pc \leftarrow m_4; \quad v_1 \leftarrow 1; \quad v_2 \leftarrow 4; \quad v_3 \leftarrow 2;$$

здесь PC – множество переменных счетчика команд; $V = \{v_1,\ v_2,\ v_3\}$ — множество переменных программы, придающих значения в D_i, $v_i = D_i$; m_n — метка состояния программы.

Также его можно представить следующим предикатом:

$$(pc = m_4) \wedge (v_1 = 1) \wedge (v_2 = 4) \wedge (v_3 = 2).$$

Предикатом называется функция из некоторого множества (значений векторов программных переменных) во множестве $\{true,\ false\}$.

Другими словами, состояние программы — это предикат, истинный на том конкретном программном состоянии, на котором выполняются заданные соотношения. Предикатом можно задать множество состояний программы:

$P = (v_1 < v_2) \wedge \neg(v_3 = 5)$ — данный предикат определят множество состояний, на котором переменные v_1, v_2, v_3 удовлетворяют соотношению P;

$s = \langle pc \leftarrow m_4;\ v_1 \leftarrow 2;\ v_2 \leftarrow 5;\ v_3 \leftarrow 4 \rangle$ — соотношение принадлежит множеству состояний, которое удовлетворяет предикату P. В свою очередь, состояние $s = \langle pc \leftarrow m_4;\ v_1 \leftarrow 5;\ v_2 \leftarrow 2; v_3 \leftarrow 4 \rangle$ не принадлежит этому множеству.

Рассмотрим оператор присваивания $v_1 = v_1 + v_2$ (метка состояния программы m_4). Новое значение переменной v_1, которое мы обозначим как v', будет равно $v_1 + v_2$ при любых начальных значениях. Следовательно, можно записать предикат, который описывает все переходы, определяемые оператором, $v_1 = v_1 + v_2$ с меткой m_4, если после него идет оператор с меткой m_5, то предикат имеет следующий вид:

$$(pc = m_4) \wedge (pc' = m_5) \wedge (v_1' = v_1 + v_2) \wedge same(\{v_2, v_3\}).$$

Предикат, приведенный ранее, описывает множество переходов программы при произвольных начальных значениях переменных v_1, v_2, v_3. Формула $same\ (V)$ определяется следующим образом: $same\ (V) \equiv \&_{v \in V}(v' = v)$.

Из приведенного ранее следует, что предикатом можно задать не только множество состояний, но и множество переходов любой программы.

Если множество D_i конечно, то каждый предикат может быть представлен булевой функцией.

Операторы программы изменяют ее состояния [13]; (s, s') — упорядоченная пара состояний в структуре Крипке. Нештрихованные переменные задают состояния, из которых направлен переход, штрихованные переменные задают состояние, в которые направлен переход. Поэтому программу можно задать предикатами, каждый из которых соответствует своему оператору.

Лексический транслятор осуществляет преобразование внутреннего представления программного кода на языке Си в формулы логики первого порядка (рис. 5).

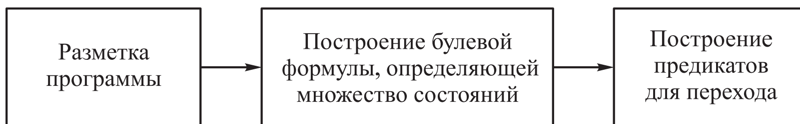

Рис. 5. Алгоритм преобразования программы в формулу логики первого порядка

Для этого лексический транслятор строит следующие формулы:

- $S_0(V, PC)$ — для множества начальных состояний;
- $\mathfrak{R} = \mathfrak{R}(V, PC, V', PC')$ — для всех возможных переходов в программе.

Рассмотрим подробнее алгоритм преобразования кода в формулы логики первого порядка.

Предположим, что программа состоит из операторов, каждый имеет один вход и один выход:

$$P ::= func \,|\, v = E \,|\, P_1;\ P_2 \,|\, if\ b\ then\ P_1\ else\ P_2 \,|\, while\ b\ do\ P$$

Шаг 1. Разметка программы и трансляция ее в идентичный помеченный текст. Каждый оператор помечен меткой m. Метки начала и конца программы считаем заданными. При трансляции любого оператора вводим метку начала, меткой конца является начало следующего оператора (табл. 2).

Таблица 2

Трансляция программы в помеченный текст

Оператор программы	Помеченный оператор программы
$func$	$m : func$
$v = E$	$m : v = E$
$P_1 ; P_2$	$m : P_1 ; m_1 : P_2$
$if\ b\ then\ P_1\ else\ P_2$	$m : if\ b\ then\ m_1 : P_1\ else\ m_2 : P_2$
$while\ b\ do\ P_1$	$m : while\ b\ do\ m_1 : P$

Шаг 2. Построение булевой формулы, определяющей множество начальных состояний программы:

$S_0(V, PC) = Prev(V) \& (pc = m_0)$, где $Prev(V)$ — предусловие, определяющее возможные начальные значения переменных (m_0 − метка начального состояния программы, $Prev(V) = true$, в случае если ограничений на начальные состояния не наложено).

Шаг 3. Построение предикатов для переходов:

$\boldsymbol{m : func;\ m_1 :}$

$\Re(m, func, m_1) \equiv (pc = m) \wedge (pc' = m_1) \wedge same(V)$ — в этом предикате ничего не изменилось;

$\boldsymbol{m : v = E;\ m_1 :}$

$\Re(m, v = E, m_1) \equiv (pc = m) \wedge (pc' = m_1) \wedge (v' = E) \wedge same(V\{v\})$ — изменились pc и v;

$\boldsymbol{m : if\ b\ then m_1 :\ P_1\ else\ m_2 :\ P_2 m_3 :}$

$\Re(m,\ if\ b\ \ then\ m_1,\ P_1\ else\ m_2,\ P_2) \equiv b \wedge (pc = m) \wedge (pc' = m_1) \wedge same(V)$ — если b истинно, то выполняем P_1;

$\vee\ \Re(if\ \ b\ then\ m_1 : P_1\ else\ m_2 : P_2) \equiv \neg b \wedge (pc = m) \wedge (pc' = m_2) \wedge same(V)$ — если b ложно, то выполняем P_1;

$\vee \Re(m_1, P_1, m_3) \vee \Re(m_1, P_1, m_3)$ — выполняем P_1 или P_2 и выходим;

$\boldsymbol{m : while\ b\ do\ m_1 :\ P\ m_2 :}$

$\Re(m,\ while\ \ b\ \ do\ m_1 : P, m_2 :) \equiv b \wedge (pc = m) \wedge (pc' = m_1) \wedge same(V)$ — если b истинно, выполняем P;

$\vee \neg b \wedge (pc = m) \wedge (pc' = m_2) \wedge same(V)$

$\vee\ \Re(m_1, P_1, m_2)$ — если b ложно, выходим из цикла.

Предикат, который определяет все возможные переходы для всей программы, строится как дизъюнкция предикатов переходов для всех операторов программы [14]:

$\mathfrak{R}(Programm) = \mathfrak{R}_1 \vee \mathfrak{R}_2 \ldots \vee \mathfrak{R}_n.$

Стоит отметить, что проводится анализ путей выполнения программы. Это наиболее предпочтительно, так как анализ условий программы ведет к комбинаторному взрыву.

Пример. Рассмотрим простую программу и построим для нее сумму предикатов переходов программы.

```
int main()
{
int k = 3;
int n = 1024;
for(inti = 1; i< n; i++)
{
if(i> 4) k = k + 1;
else
k = k*2;
printf("k > i: %i>%i\n", k,i);
}
return 0;
}
```

Осуществим разметку программы и трансляцию ее в идентичный текст (табл. 3).

Таблица 3

Разметка и трансляция программы

Оператор программы	Помеченный оператор программы
$k = 3$	$m : k = 3$
$n = 1024$	$m : n = 1024$
$for(int\, i = 1; i < n; i++)$	$m : for(int\, i = 1; i < n; m_1 : i++)$
$if\, i > 4\ then\ k = k + 1 else k = k \cdot 2$	$m : if\, i > 4\ then\ m_1 : k = k + 1\ else\ m_2 : k = k \cdot 2$

Далее построим предикат для переходов программы:

$\boldsymbol{m} : \boldsymbol{k} = 3\ \ \boldsymbol{m}_1 :$

$$\mathfrak{R}(m, k = 3, m_1) \equiv (pc = m) \wedge (pc' = m_1) \wedge (k' = 3) \wedge same(V\{k\});$$

$\boldsymbol{m} : \boldsymbol{n}1024\ \boldsymbol{m}_1 :$

$$\mathfrak{R}(m, n = 1024, m_1) \equiv (pc = m) \wedge (pc' = m_1) \wedge (n' = 1024) \wedge same(V\{n\});$$

$\boldsymbol{m} : \boldsymbol{for\ int\ i} = 1; \boldsymbol{i} < n; \boldsymbol{m}_1 : \boldsymbol{i}++)\ \boldsymbol{m}_2 :$

$$\mathfrak{R}(m : for(int\, i = 1; i < n; m_1 : i++) m_2 :) \equiv (i < n) \wedge (pc = m)(pc' = m_1) \wedge$$
$$\wedge (i' = i + 1) \wedge same(V \setminus \{i\}); \vee \neg(i < n) \wedge (pc = m) \wedge (pc' = m_2) \wedge same(V);$$

$\boldsymbol{m} : \boldsymbol{if\ i} > 4 then \boldsymbol{m}_1 : \boldsymbol{k} = \boldsymbol{k} + 1\ \boldsymbol{else}\ \boldsymbol{m}_2 : \boldsymbol{k} = \boldsymbol{k} * 2\ \boldsymbol{m}_3 :$

$$\mathfrak{R}\left(m: if\ i>4\ then\ m_1: k=k+1\ else\ m_2: k=k*2 m_3\right)\equiv$$
$$\equiv(i>4)\wedge(pc=m)\wedge(pc'=m_1)\wedge same(V\setminus\{k,i\}).$$
$$\vee b\wedge(pc=m)\wedge(pc'=m_2)\wedge same(V\setminus\{k,i\}).$$
$$\vee\mathfrak{R}(m_1,k=k+1,m_3)\vee\mathfrak{R}(m_1,k=k*2,m_3).$$

Поиск контрпримера осуществляется в соответствии со следующими правилами:

– с помощью конъюнкции к сумме предикатов переходов программы добавляем отрицание проверяемого условия

$$\mathfrak{R}(Programm)=\mathfrak{R}_1\vee\mathfrak{R}_2\ldots\mathfrak{R}_n\vee\neg(Condition);$$

– анализ программы с помощью SMT-решателя;

SMT-решатель строит модель данных, на которых проверяемое условие невыполнимо.

Анализ инварианта программы с помощью SMT-решателя. SMT — это задача разрешимости для логических формул с учетом лежащих в их основе теорий: SMT-формула — это формула в логике первого порядка, где функции и предикатные символы имеют дополнительную интерпретацию, основной ее задачей является возможность определения выполнимости или невыполнимости этой формулы. Формула содержит произвольные переменные, а предикатами являются булевы функции от этих переменных. SMT включает в себя теорию массивов и списков и теорию битовых секторов. Также решатель обеспечивает поиск уязвимостей, генерацию эксплойтов, анализ механизмов защиты и генерацию полезной нагрузки.

SMT-решатели — это алгоритмы, которые принимают на выход задачу разрешимости (вопрос, сформулированный в рамках какой-либо формальной системы и требующий ответа «Да» или «Нет»), выдают на выходе соответствующий корректный результат. Задача или вопрос представляют собой некую логическую формулу, в нашем случае инвариант программы, выраженную на языке SMT–LIB; SMT-решатели обладают высокой математической точностью и выразительной силой.

Сравнение предложенного синтетического метода верификации программы с уже существующими методами. В настоящее время существует ряд инструментов, которые используют внутреннее представление кода для реализации задач верификации ПО: LLBMC [15]; Pagai [16].

Инструмент LLBMC осуществляет полностью автоматический статический анализ кода. Для преобразования кода в SMT-формулу инструмент использует представление LLVMIR. Такой подход позволяет осуществить следующие проверки: целочисленные переполнения; деление на ноль; недопустимые битовые сдвиги; недопустимый доступ к памяти (выход за границы массива, неверный доступ с помощью указателя и т. д.); двойное освобождение области памяти (doublefree); пользовательские проверки с помощью команд assume и assert.

Инструмент PAGAI для преобразования программы на языке Си в SMT-формулу также использует возможности LLVM, но в отличие от LLBMC преобразование осуществляет в байт-код, затем осуществляется апробирование программы в виде байт-кода в программу на языке SMT–LIB.

В отличие от инструмента LLBMC, который обладает встроенными проверками кода, PAGAI имеет значительно меньшие функциональные возможности. Все проверки осуществляются с помощью пользовательских инструкций assert и assume. Пользователь добавляет в исходный код программы инструкции вида *assert* ($x > 10$) для проверки интересующего его свойства, далее в автоматическом режиме происходит проверка заданного пользователем свойства. В результате пользователь получает информацию о выполнимости свойства, в случае успешного выполнения */*assertproved*/* или */*assertnotproved*/*, в случае, если свойство не удается доказать.

Приведенные инструменты осуществляют исключительно статический анализ кода, в свою очередь, с помощью приведенного метода можно выполнить динамический анализ кода, проверку моделей с помощью SMT-решателей, а также осуществить анализ графа потока управления программы.

Предложенный метод построения контрпримеров можно применить не только к самим инструкциям программного кода, но и к состояниям программы, тем самым можно осуществить проверку выполнимости и достижимости состояний программы. Под состоянием программы следует понимать блок, который объединяет несколько программных инструкций.

Существует большое число методов динамического анализа [14]. Одним из методов является проверка поведения программы в зависимости от данных, которые подаются на ее вход.

С помощью предложенного алгоритма можно накладывать проверки и ограничения на генерируемый поток данных для проверки поведения программы.

Предложенный и существующие алгоритмы покрывают все классы операторов языка Си.

В табл. 4. приведено сравнение синтетического метода с существующими алгоритмами. Главным преимуществом предлагаемого метода перед существующими является возможность осуществлять не только статический анализ, но и динамический, а также проверку моделей. Это позволяет значительно повысить точность по сравнению с ныне существующими методами.

Таблица 4

Сравнение существующих методов

Заданные свойства ПО	Алгоритмы		Синтетический метод
	LLBMC	PAGAI	
Использование LLVMIR	+	–	+
SMT-решатель	Boolector или STP	Microsoft Z3 Solver	Microsoft Z3 Solver
Автоматизация	+	+/–	+

Окончание табл. 4

Заданные свойства ПО	Алгоритмы		Синтетический метод
	LLBMC		
Встроенные проверки кода	+	–	+
Пользовательские проверки (assert и assume)	+	+	+
Статический анализ	+	+	+
Динамический анализ	–	–	+
Проверка моделей	–	–	+

Выводы. Приведен синтетический метод верификации ПО. Метод позволяет решить проблему комбинаторного взрыва, охватывает все заданные свойства проверяемой программы, не требует построения наиболее полной сложной модели программы. В результате работы был получен алгоритм работы метода верификации и его реализация на языке SMT–LIB. Предложенный метод и алгоритмы являются новыми. Автоматически осуществляется компиляция программы в ее внутреннее представление, построение инварианта программы по ее внутреннему представлению и построение контрпримера к проверяемому свойству ПО. Построение контрпримеров допускает участие экспертов.

ЛИТЕРАТУРА

1. *Бурякова Н.А., Чернов А.В.* Классификация частично формализованных и формальных моделей и методов верификации программного обеспечения // Электрон. науч. жур. «Инженерный вестник Дона». 2010. Т. 14. № 4. URL: http://www.ivdon.ru/ru/magazine/archive/n4y2010/259

2. *Вельдер С.Э., Шалыто А.А.* Верификация простых автоматных программ на основе метода ModelChecking // Материалы XV Междунар. науч.-методич. конф. «Высокие интеллектуальные технологии и инновации в образовании и науке». 2008. С. 286.

3. *Holzmann G., Smith M.* Software model checking: Extracting verification models from source code // Formal methods for protocol engineering and distributed systems. Kluwer Academic Publ., 1999. P. 481–497.

4. *Clarke E., Kroening D., Lerda F.* A tool for checking ANSI-C programs // Tools and algorithms for the construction and analysis of systems (TACAS 2004) / Ed. by K. Jensen, A. Podelski. Vol. 2988 of Lecture Notes in Computer Science. Springer, 2004. P. 168–176.

5. *Counterexample-guided* abstraction refinement / E. Clarke, O. Grumberg, S. Jha et al. // Computer Aided Verification. 2000. P. 154–169.

6. *Software* verification with blast / T.A. Henzinger, R. Jhala, R. Majumdar, G. Sutre // 10th SPIN Workshop on Model Checking Software (SPIN). LNCS 2648. Springer-Verlag, 2003. P. 235–239.

7. *Cousot P.* Abstract interpretation. ACM Computing Surveys (CSUR) // ACM. New York. 1996. Vol. 28. Iss. 2. P. 324–328.

8. *Глухих М.И., Ицыксон В.М., Цесько В.А.* Использование зависимостей для повышения точности статического анализа программ // Модел. и анализ информ. систем. 2011. № 4. С. 72–73.

9. *Holzmann G., Peled D.* An improvement in formal verification // FORTE 1994 Conference. 1994. P. 197–211.
10. *A partial* order approach to branching time logic model checking / R. Gerth, R. Kuiper, D. Peled, W. Penczek // Inf. Comput. 1999. Vol. 150. No. 2. P. 132–152.
11. *Partial* order reductions preserving simulations / W. Penczek, M. Szreter, R. Gerth, R. Kuiper // Proc. of Concurrency, Specification and Programming (CSP'99). Warsaw, 1999. P. 153–172.
12. *Holzmann G.J.* An analysis of bitstate hashing // Formal methods in system design, 1998. Vol. 13. No. 3. P. 287–307.
13. *Holzmann G.J.* The spin model checker: Primer and Reference Manual. Addison-Wesley, 2003. 608 p.
14. *Вартанов С.П., Герасимов А.Ю.* Динамический анализ программ с целью поиска ошибок и уязвимостей при помощи целенаправленной генерации входных данных // Сб. трудов ИСП РАН. 2014. Т. 26. Вып. 1. С. 375–394.
DOI: 10.15514/ISPRAS-2014-26(1)-15
URL: http://www.ispras.ru/proceedings/isp_26_2014_1/isp_26_2014_1_375/
15. *Julien H., Monniaux D., Moy M.* PAGAI: a path sensitive static analyzer. Grenoble-INP, VERIMAG Grenoble France, 2012. 14 p.
16. *Merz F., Falke S., Sinz C.* LLBMC: Bounded model checking of C and C++ programs using a compiler IR. Institute for Theoretical Computer Science Karlsruhe Institute of Technology (KIT). Germany, 2012. 16 p.

Рудаков Игорь Владимирович — канд. техн. наук, доцент, заведующий кафедрой «Программное обеспечение ЭВМ и информационные технологии» МГТУ им. Н.Э. Баумана (Российская Федерация, 105005, Москва, 2-я Бауманская ул., д. 5).

Гурин Ростислав Евгеньевич — инженер МГТУ им. Н.Э. Баумана (Российская Федерация, 105005, Москва, 2-я Бауманская ул., д. 5).

Просьба ссылаться на эту статью следующим образом:
Рудаков И.В., Гурин Р.Е. Разработка и исследование синтетического метода верификации программы с помощью SMT-решателей // Вестник МГТУ им. Н.Э. Баумана. Сер. Приборостроение. 2016. № 4. С. 49–64. DOI: 10.18698/0236-3933-2016-4-49-64

SYNTHETIC SOFTWARE VERIFICATION METHOD USING SMT-SOLVERS

I.V. Rudakov irudakov@yandex.ru
R.E. Gurin rg.bmstu@gmail.com

Bauman Moscow State Technical University, Moscow, Russian Federation

Abstract

Non-deterministic behavior of the software remains a pressing issue. Software verification aims to detect errors, identify vulnerabilities and check whether the required features are implemented correctly. Verification methods can be divided into groups: the empirical (using review), formal (using mathematics for software verification), static (checking the software implementation without immediate start), dynamic (checking the software

Keywords

Verification, code analysis, static analysis, dynamic analysis, interpretation, symbolic execution, model checking

implementation through direct start), with different levels of automation. Existing verification methods are flawed as there is an exponential increase in the number of states (combinatorial explosion). One way to resolve this problem is to create a synthetic method. We offer a synthetic software verification method based on SMT — solver that does not require a complex model. Its algorithm is implemented using SMT-LIB

REFERENCES

[1] Buryakova N.A., Chernov A.V. Classification of partially formal and formal models and methods for software verification. *Inzhenernyy vestnik Dona* [Electron. sci. journ. Engineering Journal of Don], 2010, vol. 14, no. 4. Available at: http://www.ivdon.ru/ru/magazine/archive/n4y2010/259

[2] Vel'der S.E., Shalyto A.A. Verification of simple automate programs on the basis of the modelchecking method. ModelChecking. *Mat. XV Mezhdunar. nauch.-metodich. konf. “Vysokie intellektual'nye tekhnologii i innovatsii v obrazovanii i nauke”* [Highly intellectual technologies and innovations in education and science], 2008, p. 286 (in Russ.).

[3] Holzmann G., Smith M. Software model checking: Extracting verification models from source code. *Formal methods for protocol engineering and distributed systems*. Kluwer Academic Publ., 1999, pp. 481–497.

[4] Clarke E., Kroening D., Lerda F. A tool for checking ANSI-C programs. *Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS 2004*). Ed. by K. Jensen, A. Podelski. Vol. 2988 of Lecture Notes in Computer Science. Springer, 2004, pp. 168–176.

[5] Clarke E., Grumberg O., Jha S. et al. Counterexample-guided abstraction refinement. *Computer Aided Verification*, 2000, pp. 154–169.

[6] Henzinger T.A., Jhala R., Majumdar R., Sutre G. Software verification with blast. *10th SPIN Workshop on Model Checking Software* (*SPIN*). LNCS 2648. Springer-Verlag, 2003, pp. 235–239.

[7] Cousot P. Abstract interpretation. ACM Computing Surveys (CSUR). *ACM*. N.Y., 1996, vol. 28, iss. 2, pp. 324–328.

[8] Glukhikh M.I., Itsykson V.M., Tsesko V.A. The use of dependencies for improving the precision of program static analysis. *Model. Anal. Inform. Sist*., 2011, vol. 18, no. 4, pp. 68–79.

[9] Holzmann G., Peled D. An improvement in formal verification. *FORTE 1994 Conference*, 1994, pp. 197–211.

[10] Gerth R., Kuiper R., Peled D., Penczek W. A partial order approach to branching time logic model checking. *Inf. Comput*., 1999, vol. 150, no. 2, pp. 132–152.

[11] Penczek W., Szreter M., Gerth R., Kuiper R. Partial order reductions preserving simulations. *Proc. of Concurrency, Specification and Programming* (*CSP’99*). Warsaw, 1999, pp. 153–172.

[12] Holzmann G.J. An analysis of bitstate hashing. *Formal methods in system design*, 1998, vol. 13, no. 3, pp. 287–307.

[13] Holzmann G.J. The spin model checker: Primer and Reference Manual. Addison-Wesley, 2003. 608 p.

[14] Vartanov S.P., Gerasimov A.Yu. Dynamic program analysis for error detection using goal-seeking input data generation. *Sb. tr. ISP RAN* [Proceedings of ISP RAS], 2014, vol. 26, iss. 1, pp. 375–394. DOI: 10.15514/ISPRAS-2014-26(1)-15
Available at: http://www.ispras.ru/en/proceedings/isp_26_2014_1/isp_26_2014_1_375/

[15] Julien H., Monniaux D., Moy M. PAGAI: a path sensitive static analyzer. Grenoble-INP, VERIMAG Grenoble France, 2012. 14 p.

[16] Merz F., Falke S., Sinz C. LLBMC: Bounded model checking of C and C++ Programs using a compiler IR. Institute for Theoretical Computer Science Karlsruhe Institute of Technology (KIT). Germany, 2012. 16 p.

Rudakov I.V. — Cand. Sci. (Eng.), Assoc. Professor, Head of Computer Software and Information Technology Department, Bauman Moscow State Technical University (2-ya Baumanskaya ul. 5, Moscow, 105005 Russian Federation).

Gurin R.E. — engineer, Bauman Moscow State Technical University (2-ya Baumanskaya ul. 5, Moscow, 105005 Russian Federation).

Please cite this article in English as:

Rudakov I.V., Gurin R.E. Synthetic Software Verification Method using SMT-Solvers. *Vestn. Mosk. Gos. Tekh. Univ. im. N.E. Baumana, Priborostr.* [Herald of the Bauman Moscow State Tech. Univ., Instrum. Eng.], 2016, no. 4, pp. 49–64. DOI: 10.18698/0236-3933-2016-4-49-64