Моделирование и анализ информационных систем. Т. 25, № 5 (2018), с. 491–505
Modeling and Analysis of Information Systems. Vol. 25, No 5 (2018), pp. 491–505

DOI: 10.18255/1818-1015-2018-5-491-505

УДК 004.052.42

Автоматизация верификации С-программ с использованием символического метода элиминации инвариантов циклов

Кондратьев Д. А.[1], **Марьясов И. В.**, **Непомнящий В. А.**

получена 10 сентября 2018

Аннотация. При дедуктивной верификации программ, написанных на императивных языках программирования, особую сложность вызывает порождение и доказательство условий корректности, соответствующих циклам, поскольку каждый из них должен быть снабжён инвариантом, построение которого часто является нетривиальной задачей. Методы синтеза инвариантов циклов, как правило, носят эвристический характер, что затрудняет их применение. Альтернативой является символический метод элиминации инвариантов циклов, предложенный В.А. Непомнящим в 2005 году. Его идея состоит в представлении тела цикла в виде специальной операции замены при выполнении определённых ограничений. Такая операция в символической форме выражает действие цикла, что позволяет ввести в аксиоматическую семантику правило вывода для циклов, не использующее инварианты. В данной работе представлено дальнейшее развитие этого метода. Он расширяет метод смешанной аксиоматической семантики, предложенный для верификации C-light программ. Данное расширение включает в себя метод верификации итераций над изменяемыми массивами с возможным выходом из тела цикла в C-light программах. Метод содержит правило вывода для итерации без инвариантов циклов. Данное правило было реализовано в генераторе условий корректности, являющемся частью системы автоматизированной верификации C-light программ. Для проведения автоматического доказательства в используемой системе ACL2 были разработаны и реализованы два алгоритма: первый порождает операцию замены на языке системы ACL2, а второй генерирует вспомогательные леммы, позволяющие системе ACL2 успешно доказать получаемые условия корректности в автоматическом режиме. Применение вышеуказанных методов и алгоритмов проиллюстрировано примером.

Ключевые слова: Си-лайт, инварианты циклов, смешанная аксиоматическая семантика, финитная итерация, массивы, ACL2, спецификация, верификация, логика Хоара

Для цитирования: Кондратьев Д. А., Марьясов И. В., Непомнящий В. А., "Автоматизация верификации С-программ с использованием символического метода элиминации инвариантов циклов", *Моделирование и анализ информационных систем*, **25**:5 (2018), 491–505.

Об авторах: Кондратьев Дмитрий Александрович, orcid.org/0000-0002-9387-6735, аспирант,
Институт систем информатики им. А.П. Ершова СО РАН,
пр. Академика Лаврентьева, 6, г. Новосибирск, 630090 Россия, e-mail: apple-66@mail.ru

Марьясов Илья Владимирович, orcid.org/0000-0002-2497-6484, канд. физ.-мат. наук,
Институт систем информатики им. А.П. Ершова СО РАН,
пр. Академика Лаврентьева, 6, г. Новосибирск, 630090 Россия, e-mail: ivm@iis.nsk.su

Непомнящий Валерий Александрович, orcid.org/0000-0003-1364-5281, канд. физ.-мат. наук,
Институт систем информатики им. А.П. Ершова СО РАН,
пр. Академика Лаврентьева, 6, г. Новосибирск, 630090 Россия, e-mail: vnep@iis.nsk.su

Благодарности:
[1] Автор частично поддержан грантом РФФИ № 17-01-00789.

Введение

В настоящее время верификация С-программ является актуальной проблемой программирования. Некоторые проекты (например [2, 4]) предлагают различные подходы. Но ни один из них не содержит каких-либо методов автоматической верификации программ с циклами без инвариантов. Как известно, для того чтобы верифицировать программы с циклами, пользователь должен предоставить инварианты, построение которых часто является сложной задачей. Туэрк [15] предложил использовать пред- и постусловия для циклов типа **while**, однако их построение также целиком лежит на пользователе. Ли и другие [10] разработали обучающийся алгоритм генерации инвариантов циклов, тем не менее, их метод не допускает наличие оператора **break** в теле цикла, а также операций над массивами.

Мы рассматриваем циклы с определёнными ограничениями [14]. Мы расширяем наш метод смешанной аксиоматической семантики языка C-light [1] правилом для верификации таких циклов, основанным на операции замены [9, 14]. С помощью этого правила порождаются условия корректности специального вида.

В нашей статье [12] мы представили:

1. Метод верификации программ над неизменяемыми массивами с выходом из цикла.
2. Рекурсивный алгоритм построения операции замены.
3. Стратегию интерактивного доказательства условий корректности.

В процессе наших исследований появились две цели: рассмотреть случай изменяемых структур данных и разработать автоматизированные методы доказательства условий корректности. Данная статья представляет следующие результаты:

1. Метод верификации программ над изменяемыми массивами с выходом из цикла.
2. Рекурсивный алгоритм построения операции замены для таких программ.
3. Стратегия автоматизации доказательства условий корректности.

На стадии доказательства мы используем систему ACL2 [6] вместо SMT-решателей. Поэтому был разработан новый алгоритм построения операции замены, который транслирует инструкции тела цикла на языке C-light в конструкции входного языка системы ACL2. Также был разработан и реализован эвристический метод генерации вспомогательных утверждений, позволяющих системе ACL2 успешно доказывать условия корректности в автоматическом режиме.

1. Финитная итерация над изменяемыми структурами данных и операция замены

Метод элиминации инвариантов циклов для финитной итерации был предложен в [14]. Он состоит из четырёх случаев:

1. Финитная итерация над неизменяемыми структурами данных без выхода из цикла.

2. Финитная итерация над неизменяемыми структурами данных с выходом из цикла.

3. Финитная итерация над изменяемыми структурами данных без / с выходом из цикла.

4. Финитная итерация над иерархическими структурами данных с выходом из цикла.

Первый случай был рассмотрен в [11], второй — в [12]. В этой статье мы рассматриваем третий случай.

Напомним понятие структур данных, которые содержат конечное число элементов. Пусть $memb(S)$ обозначает мультимножество элементов структуры S, а $|memb(S)|$ — мощность мультимножества $memb(S)$. Для структуры S определены следующие операции:

1. $empty(S) = true$ тогда и только тогда, когда $|memb(S)| = 0$.

2. $choo(S)$ возвращает элемент из $memb(S)$, если $\neg empty(S)$.

3. $rest(S) = S'$, где S' — структура типа S и $memb(S') = memb(S) \setminus \{choo(S)\}$, если $\neg empty(S)$.

Множества, последовательности, списки, строки, массивы, файлы и деревья являются типичными примерами структур данных.

Пусть $\neg empty(S)$, тогда положим $vec(S) = [s_1, s_2, \ldots, s_n]$, где $memb(S) = \{s_1, s_2, \ldots, s_n\}$ и $s_i = choo(rest^{i-1}(S))$ для $i = 1, 2, \ldots, n$.

Рассмотрим оператор

$$\textbf{for x in S do v} := \textbf{body}(\mathbf{v}, \mathbf{x})\ \textbf{end},$$

где S — структура, x — переменная типа «элемент S», v — вектор переменных цикла, не содержащий x, а $body$ представляет вычисление, реализуемое телом цикла, которое не изменяет x и завершается для каждого $x \in memb(S)$. Структура S может быть изменена следующим образом. Тело цикла может содержать только операторы присваивания, условные операторы (в том числе вложенные) и операторы выхода из цикла **break**. Такой цикл **for** называется финитной итерацией.

Операционная семантика такого оператора выглядит следующим образом. Пусть v_0 — вектор начальных значений переменных из v. Если $empty(S)$ истинно, то результат итерации $v = v_0$. Иначе $vec(S) = [s_1, s_2, \ldots, s_n]$, тогда тело цикла последовательно итерирует таким образом, что x принимает значения $s_1, s_2, \ldots, s_n$, а $body(v, s_j)$ может изменять $s_1, s_2, \ldots, s_{j-1}$.

Для того чтобы выразить результат действия итерации, определим операцию замены $rep(v, S, body, n)$, где $rep(v, S, body, 0) = v$, $rep(v, S, body, i) = body(rep(v, S, body, i-1), s_i)$ для всех $i = 1, 2, \ldots, n$, если $\neg empty(S)$.

В [14] были доказаны теоремы, выражающие важные свойства операции замены.

Правило вывода для финитной итерации имеет вид:

$$\frac{E, SP \vdash \{P\}\,\mathbf{A};\{Q(v \leftarrow rep(v, S, body))\}}{E, SP \vdash \{P\}\,\mathbf{A};\ \mathbf{for\ x\ in\ S\ do\ v} := \mathbf{body(v, x)}\,\mathbf{end}\{Q\}}$$

где **A** — операторы программы перед циклом. Мы используем обратное прослеживание: мы двигаемся от конца программы до её начала, удаляя самый правый оператор (на верхнем уровне), применяя соответствующее правило вывода смешанной аксиоматической семантики [1] языка C-light. E называется окружением, которое содержит информацию о текущей функции (её идентификатор, тип и тело), которую верифицируют, информацию о текущем блоке и идентификатор метки, если ранее был встречен оператор перехода **goto**. SP — спецификация программы, которая включает все предусловия, постусловия и инварианты циклов и помеченных операторов.

2. Входной язык системы ACL2

Входным языком системы ACL2 [6] является аппликативный диалект языка Common Lisp, в котором поддерживается только функциональная парадигма и отсутствует поддержка императивной. Далее под языком ACL2 будем понимать входной язык системы ACL2.

Удобным способом задания операции замены *rep* является трансляция каждой конструкции цикла в семантически эквивалентную композицию конструкций языка ACL2. Так как конструкции цикла написаны в императивной парадигме, то прямых аналогов им в языке ACL2 нет. Поэтому рассмотрим те конструкции языка ACL2, которые позволяют моделировать используемые в финитных итерациях конструкции языка C-light.

Так как язык Common Lisp ориентирован на обработку списков, то массивы языка C-light мы моделируем с помощью списков. При этом мы используем две операции для обработки индексированных последовательностей. Рассмотрим функции *nth* и *update-nth*, реализующие эти операции. Если i — индекс, l — список, то $(nth\ i\ l)$ — значение i-го элемента списка l. Если *expr* — выражение языка ACL2, то *(update-nth i expr l)* — новый список, который совпадает со списком l за исключением i-го элемента, значением которого является значение *expr*.

Рассмотрим используемые в языке ACL2 операции над списками, позволяющие моделировать соответствующие операции над массивами. Предикат *integer-listp* истинен тогда и только тогда, когда аргументом является список и все его элементы имеют целочисленный тип. Предикат *equal* истинен тогда и только тогда, когда равны и длины списков, и их элементы. Функция *length* вычисляет длину списка.

Моделировать новые типы данных позволяет библиотека *fty* языка ACL2, в которой работа с типизированными переменными сводится к использованию специальных функций. Каждому типу соответствует предикат, определяющий принадлежность к данному типу, функция приведения к данному типу и отношение эквивалентности. Такие типы можно задать с помощью специальных конструкций библиотеки *fty*. Макрос *fty::defprod* задаёт тип, аналогичный инструкции **struct** языка C-light. Если *st* — такой тип, то будут сгенерированы связанные с типом *st* функции с названиями *st-p*, *st-fix* и *st-equiv* соответственно. Также с помощью *fty::defprod* будет сгенерирован конструктор, макросы *make-st, change-st* и функции

доступа к значениям полей. Пусть fd — поле структуры s, имеющей тип st. Тогда *(change-st s :fd expr)* — новая структура типа st, совпадающая со структурой s за исключением поля fd, значением которого является значение $expr$. Если fd — единственное поле структуры st, то *(make-st :fd expr)* — новая структура типа st, у которой значением поля fd является значение $expr$. Тогда *(st->fd s)* значение поля fd структуры s. Другим способом доступа к данному значению является $s.fd$.

Макрос $b*$ позволяет промоделировать последовательное исполнение инструкций. Он является расширением макроса $let*$ языка ACL2, позволяющего удобным образом задать вложенный let. Рассмотрим общий вид конструкции $let*$:

$$(let^* \ ((var_1 \ term_1) \ ... \ (var_n \ term_n)) \ body),$$

где все var_i являются переменными (не обязательно различными), $body$ и все $term_i$ являются выражениями языка ACL2. Эта конструкция эквивалентна следующей:

$$(let \ ((var_1 \ term_1)) \ldots (let \ ((var_n \ term_n)) \ body) \ ...).$$

Таким образом, связывание переменных val_i со значениями соответствующих выражений $term_i$ происходит последовательно. Значением такой конструкции является значение выражения $body$. Отметим, что каждая пара $(var_i \ term_i)$ называется связыванием переменной var_i и значения выражения $term_i$, а переменная var_i называется локальной переменной в блоке $let*$.

Рассмотрим общий вид конструкции $b*$:

(b <list-of-bindings>.<list-of-result-forms>)* .

где *<list-of-bindings>* — список конструкций *binding*, *<list-of-result-forms>* — список выражений языка ACL2. Значением конструкции $b*$ является последнее выражение списка *<list-of-result-forms>*. По аналогии с конструкцией $let*$ операции *binding* выполняются последовательно. При этом, общим видом конструкции *binding* является

(<binder-form> [<expression>]) ,

где *<binder-form>* — конструкция *b*-binder*, *<expression>* — выражение языка ACL2. Так как $b*$ является расширением $let*$, то частным случаем конструкции *binding* является связывание переменной и значения выражения. При этом переменная становится локальной переменной блока $b*$.

Таким образом, одним из возможных видов конструкции *<binder-form>* является переменная. Другим возможным видом конструкции *<binder-form>* является (*when condition*), где *condition* — логическое выражение языка ACL2. Пусть блок $b*$ содержит *binding*-конструкцию ((*when condition*) *expression*). Если выражение *condition* истинно, то следующие за данной конструкцией операции *binding* не исполняются, а значением блока $b*$ является *expression*. Такая конструкция позволяет моделировать изменение потока управления C-light программы.

Язык ACL2 поддерживает многозначные функции. Конструкция mv позволяет функции возвратить более одного значения. Для связывания многих переменных с результатом исполнения многозначных функций используется следующая конструкция *binding*:

((mv x_1 x_2 ... x_{n-1} x_n*) (form-returning-n-values)) ,*

где x_i — переменная для каждого i $(1 \leq i \leq n)$, *(form-returning-n-values)* — вызов функции, возвращающий n значений. Таким образом i-я переменная x связывется с i-м значением *(form-returning-n-values)*.

Язык ACL2 обеспечивает поддержку пользовательских теорий. Формулы задаются с помощью логических связок *not* (логическое «не»), *or* (логическое «или»), *and* (логическое «и»), *implies* (импликация). В таких логических выражениях удобно использовать предикаты для проверки принадлежности переменной определённому типу. Например, предикат *integerp* проверяет принадлежность аргумента целочисленному типу. Конструкция $define$ задаёт функцию на языке ACL2, а также позволяет вводить леммы о ней. С помощью конструкции $defrule$ задаются леммы, теоремы и специальные секции, содержащие информацию, которая помогает ACL2 доказать теорему. Например, в секции *prep-lemmas* можно задать леммы, позволяющие доказать определяемую конструкцией $defrule$ теорему. Секция *enable* позволяет при доказательстве использовать леммы о функциях, определённых в $define$. Секция *induct* разрешает проведение доказательства по индукции. Аргументом данной секции служит применение специальной функции к переменным теоремы. Рекурсивное определение такой функции задаёт схему индукции. Например, рекурсивное определение библиотечной функции *dec-induct* определяет классическую схему индукции по натуральному параметру с шагом 1.

3. Генерация операции замены для одномерных массивов на входном языке системы ACL2

Пусть S — одномерный массив из n элементов простого типа языка C-light и $S \in v$. Рассмотрим специальный случай финитной итерации над массивом S:

$$\mathbf{for}\,(\mathbf{i} = \mathbf{0}; \mathbf{i} < \mathbf{n}; \mathbf{i}++)\ \mathbf{v} := \mathbf{body}(\mathbf{v}, \mathbf{i})\ \mathbf{end},$$

где тело цикла является допустимой конструкцией.

Допустимая конструкция — это один из следующих операторов языка C-kernel:

1. Пустой оператор, в том числе пустой блок.
2. Оператор выхода из цикла **break**;.
3. Оператор присваивания $\mathbf{a} = \mathbf{b}$;, где a — переменная простого типа, либо имеет вид $S[i]$, b — выражение языка C-kernel.
4. Условный оператор **if** (**a**) **b**, где a — выражение языка C-kernel, b — допустимая конструкция.
5. Условный оператор **if** (**a**) **b else c**, где a — выражение языка C-kernel, b и c — допустимые конструкции.
6. Блок $\{\mathbf{a_1}\ \mathbf{a_2}\ ...\ \mathbf{a_{k-1}}\ \mathbf{a_k}\}$, где a_r — допустимая конструкция для каждого r: $1 \leq r \leq k$.

В вектор v входит массив S и переменные простого типа, которые могут изменяться в теле цикла. Введём функцию w, её аргументом является допустимая конструкция, и она возвращает множество элементов вектора v для данной конструкции. Определим такую функцию для каждого вида допустимой конструкции *op*:

1. Если *op* — это пустой оператор (в том числе пустой блок), то $w(op) = \emptyset$.
2. Если *op* — это **break**;, то $w(op) = \emptyset$.
3. Если *op* – это **a = b**;, где a — переменная простого типа либо имеет вид $S[i]$, b — выражение языка C-kernel, то возможны два варианта:
   (a) Если a — переменная простого типа, то $w(op) = \{a\}$.
   (b) Если a имеет вид $S[i]$, то $w(op) = \{S\}$.
4. Если *op* — это **if (a) b**, то $w(op) = w(b)$.
5. Если *op* — это **if (a) b else c**, то $w(op) = w(b) \cup w(c)$.
6. Если *op* — это $\{\mathbf{a_1\ a_2\ \ldots\ a_{k-1}\ a_k}\}$, то $w(op) = w(a_1) \cup w(a_2) \cup \ldots \cup w(a_{k-1}) \cup w(a_k)$.

Пусть T — результат применения функции w к телу цикла. Рассмотрим произвольное упорядочивание элементов множества T. Обозначим его как вектор v.

Генерация функции *rep* состоит из трёх этапов. На первом шаге осуществляется генерация типа данных $frame$ с помощью конструкции *fty::defprod*. Этот тип данных представляет собой структуру, полями которой являются элементы вектора v, а также булевское поле **loop-break**. Также конструкция *fty::defprod* генерирует макросы *make-frame* и *change-frame*, функции *frame-p*, *frame-fix*, *frame-equiv* и функции доступа к значениям полей.

Значения полей переменной типа $frame$ являются значениями соответствующих переменных в ходе исполнения цикла. Поле **loop-break** истинно тогда и только тогда, когда сработал оператор **break**. Таким образом, переменная типа $frame$ хранит значения вектора v в ходе исполнения цикла. Поэтому необходим способ задания начальных значений элементов вектора v, то есть значений перед исполнением цикла. Для этого на втором этапе с помощью конструкции *make-frame* происходит генерация функции *frame-init*. Эта функция возвращает структуру типа $frame$ с заданными начальными значениями полей. Значением поля **loop-break** у такой структуры является nil. Это означает, что перед исполнением цикла оператор **break** не срабатывает.

На третьем этапе происходит генерация функции *rep* по телу цикла следующим образом. Первым аргументом функции *rep* является номер итерации. При использовании функции *rep* в условии корректности в качестве значения первого аргумента используется n, так как n — номер последней итерации. Вторым аргументом функции *rep* является переменная типа $frame$, поля которой соответствуют начальным значениям элементов вектора v. Таким образом, при использовании функции *rep* в условии корректности значением второго аргумента является *frame-init*(v_0), где v_0 — вектор начальных значений элементов v.

Пусть значением первого аргумента функции rep является m. Тогда функция rep должна возвращать такую структуру типа $frame$, что значения ее полей будут значениям вектора v после m итераций цикла.

Будем считать, что нулевая итерация цикла соответствует значениям элементов вектора v перед исполнением цикла. Поэтому

$$rep(0,\ frame\text{-}init\ (v_0)) = frame\text{-}init\ (v_0)\ .$$

Это ограничивает глубину рекурсии.

В случае выхода из цикла на итерации l $(0 < l \leq n)$ при исполнении оператора **break** будем считать, что итерации цикла продолжаются, но значения v не изменяются на таких итерациях j, что $l \leq j \leq n$. Зададим rep так, что

$$rep(l,\ frame\text{-}init\ (v_0)) = rep(j,\ frame\text{-}init\ (v_0))\ .$$

Поэтому необходимо проверять на истинность значение поля **loop-break** у возвращённой на предыдущей итерации структуры типа $frame$. В случае, когда выход из цикла происходит на текущей итерации, необходимо прекратить исполнение и вернуть такую структуру типа $frame$, что её поле **loop-break** будет истинным.

Каждую переменную функции rep и ее аргумент типа $frame$ обозначим через fr. При необходимости изменить значения полей переменной fr создаётся новая переменная fr с новыми значениями полей. Это позволяет удобным образом обрабатывать элементы вектора v.

Алгоритм генерации функции rep состоит из двух шагов. На первом шаге используется функция $generate\_rep\_one$, которая строит сигнатуру функции rep и конструкцию $b*$. Эта конструкция $b*$ является самым верхним по уровню вложенности блоком кода функции rep на языке ACL2. Этот блок соответствует составному оператору, являющемуся телом цикла. Такая конструкция позволяет промоделировать последовательное исполнение следующих инструкций: проверку условия выхода из рекурсии с помощью конструкции $when$, рекурсивный вызов rep для предыдущей итерации, результат которого сохраняется в переменной fr, проверку с помощью конструкции $when$ выхода из цикла на предыдущей итерации.

Каждая такая инструкция блока $b*$ является конструкцией *binding*. Следующие за ними в этом блоке $b*$ инструкции *binding* моделируют последовательное исполнение тела цикла, совершая операции над структурой fr. Они будут сгенерированы функцией $generate\_rep$ на втором шаге алгоритма. Кроме того, на первом шаге алгоритма функция $generate\_rep\_one$ генерирует выражение fr, являющееся значением данного блока $b*$. Таким образом, если не было выхода из цикла, то значением функции rep на текущей итерации будет переменная fr, полученная при применении к результату предыдущей итерации конструкций, соответствующих телу цикла.

На втором шаге алгоритма используется функция $generate\_rep$, генерирующая конструкции, соответствующие телу цикла. Отметим, что для любой финитной итерации данного вида на первом шаге алгоритма генерируется один и тот же код на языке ACL2. То есть различия между разными финитными итерациями учитываются не на первом, а на втором шаге алгоритма.

Функция $generate\_rep$ осуществляет трансляцию тела цикла на язык ACL2. Она принимает в качестве аргумента допустимую конструкцию и транслирует её на

язык ACL2. Далее под объемлющим блоком для конструкции *binding* будем понимать блок $b*$, в котором она находится. Функция $c\_kernel\_translator$ осуществляет трансляцию C-kernel выражения на язык ACL2. Определим функцию $generate\_rep$ для каждого вида допустимой конструкции op:

1. Если op – это пустой оператор (в том числе пустой блок), то результатом $generate\_rep(op)$ будет $(fr\ fr)$. Данная запись является конструкцией *binding* в объемлющем блоке $b*$. Она означает, что внутри блока создается новая переменная fr со значением прежней и с новой областью видимости. То есть пустой оператор (в том числе пустой блок) не меняет значения элементов вектора v.

2. Если op — это **break**;, то результатом $generate\_rep(op)$ будет

$$\textit{(fr (change-frame fr :loop-break t)) ((when t) fr)} .$$

   Данная запись представляет собой две последовательно идущие конструкции *binding* в объемлющем блоке $b*$. Первая конструкция *binding* означает, что внутри блока создается новая переменная fr со значением *(change-frame fr :loop-break t)* и с новой областью видимости. То есть у новой переменной fr поле *loop-break* стало истинным.

   Вторая конструкция *binding* означает, что при соблюдении условия *when* происходит выход из объемлющего блока $b*$. Так как таким условием *when* является t, то всегда происходит выход из объемлющего блока $b*$ при срабатывании данной второй конструкции. То есть при срабатывании оператора **break** происходит выход из цикла. Возвращаемым значением такого объемлющего блока $b*$ будет являться переменная fr, поле *loop-break* которой будет указывать на то, что необходимо также произвести выход из блоков, в которые вложен данный объемлющий блок (если они существуют).

3. Если op — это $\mathbf{a} = \mathbf{b}$;, где a — переменная простого типа либо имеет вид $S[i]$, b — выражение языка C-kernel, то возможны два варианта:

   (a) Если a — переменная простого типа, то результатом $generate\_rep(op)$ будет

$$\textit{(fr (change-frame fr :a c\_kernel\_translator(b)))} .$$

   Данная запись является конструкцией *binding* в объемлющем блоке $b*$. Она означает, что внутри блока создаётся новая переменная fr со значением *(change-frame fr :a c_kernel_translator(b))* и с новой областью видимости. То есть у новой переменной fr поле a принимает значение выражения b. Таким образом, оператор присваивания меняет значение переменной a вектора v.

   (b) Если a имеет вид $S[i]$, то результатом $generate\_rep(op)$ будет

$$\textit{(fr (change-frame fr :S (update-nth i c\_kernel\_translator(b) (frame->S fr))))} .$$

Данная запись является конструкцией *binding* в объемлющем блоке $b*$. Она означает, что внутри блока создается новая переменная fr со значением

(change-frame fr :S (update-nth i c_kernel_translator(b) (frame->S fr)))

и с новой областью видимости. То есть у новой переменной fr поле S принимает значение последовательности S, в которой на месте элемента с индексом i стоит значение выражения b. Таким образом оператор присваивания изменяет массив S из вектора v.

4. Если *op* — это **if** (**a**) **b**, то результатом *generate_rep*(*op*) будет

(fr (if c_kernel_translator(a) (b (generate_rep(b)) fr) fr)) ((when (frame->loop-break fr)) fr) .*

Данная запись представляет собой две последовательно идущие конструкции *binding* в объемлющем блоке $b*$. Первая конструкция *binding* означает, что внутри блока создается новая переменная fr с новой областью видимости и со значением

(if c_kernel_translator(a) (b(generate_rep(b)) fr) fr) .*

То есть при выполнении условия a значением переменной fr станет

(b (generate_rep(b)) fr) ,*

где *generate_rep*(b) — результат трансляции b на входной язык системы ACL2. Отметим, что любой оператор b (в том числе составной) транслируется в одну или две *binding* конструкции. Блок $b*$ в ACL2-коде для оператора if является объемлющим для этих *binding* конструкций. Таким образом, при выполнении условия a значение вектора v будет определяться положительной ветвью условного оператора.

При невыполнении условия a значением переменной fr станет предыдущее значение fr. Таким образом, при невыполнении условия a значение вектора v не изменится.

Вторая конструкция *binding* означает, что при соблюдении условия $when$ происходит выход из объемлющего блока $b*$. Так как таким условием $when$ является *(frame->loop-break fr)*, то выход из объемлющего блока $b*$ происходит при истинности поля *loop-break* переменной fr. То есть если при исполнении оператора сработала инструкция $break$, то происходит выход из объемлющего блока $b*$. Возвращаемым значением такого объемлющего блока $b*$ будет являться переменная fr, поле $loop - break$ которой будет указывать на то, что необходимо также произвести выход из блоков, в которые вложен данный объемлющий блок (если они существуют).

5. Если *op* — это **if** (**a**) **b else c**, то результатом *generate_rep*(*op*) будет

$$\begin{array}{c}\textit{(fr (if c\_kernel\_translator(a) (b*(generate\_rep(b)) fr)}\\ \textit{(b*(generate\_rep(c)) fr))) ((when (frame->loop-break fr)) fr)}\ .\end{array}$$

Данная запись представляет собой две последовательно идущие конструкции *binding* в объемлющем блоке $b*$. Первая конструкция *binding* определяется по аналогии с пунктом 4.

Определение второй конструкции *binding* совпадает с определением из пункта 4.

6. Если *op* — это $\{\mathbf{a_1}\ \mathbf{a_2}\ \ldots\ \mathbf{a_{k-1}}\ \mathbf{a_k}\}$, то результатом *generate_rep*(*op*) будет

$$\begin{array}{c}(fr\ (b^*(generate\_rep(a_1)\ generate\_rep(a_2)\ldots generate\_rep(a_{k-1})\\ generate\_rep(a_k))fr))((when\ (frame\text{->}loop\text{-}break\ \ fr))\ \ fr)\ .\end{array}$$

Данная запись представляет собой две последовательно идущие конструкции *binding* в объемлющем блоке $b*$. Первая конструкция *binding* означает, что внутри блока создается новая переменная fr с новой областью видимости и со значением

$$\begin{array}{c}(b^*(generate\_rep(a_1)\ generate\_rep(a_2)\ldots\\ generate\_rep(a_{k-1})\ generate\_rep(a_k))fr)\ ,\end{array}$$

где *generate_rep*(a_r) для каждого r $(1 \le r \le k)$ — результат трансляции a_r на входной язык системы ACL2. Отметим, что любой оператор **c** (в том числе составной) транслируется в одну или две *binding* конструкции. Блок $b*$ в ACL2-коде для составного оператора является объемлющим для этих *binding* конструкций. Таким образом, значение переменной fr при исполнении конструкции $b*$ является значением вектора v при исполнении составного оператора.

Вторая конструкция *binding* определяется как в пункте 4.

Вопрос существования объемлющего блока решается построением дерева вложенности блоков для кода, сгенерированного на втором шаге алгоритма. В качестве корня такого дерева возьмём блок $b*$, сгенерированный функцией *generate_rep_one* на первом шаге алгоритма. Этот блок соответствует телу цикла. Отметим, что каждому оператору тела цикла можно сопоставить блок в данном дереве вложенности блоков. Например, если if-ветвь представлена единственным оператором, то объемлющим для него будет блок $b*$, сгенерированный при трансляции данного if. А если оператор входит в последовательность инструкций составного оператора ветви if, то объемлющим для него будет блок $b*$, полученный при трансляции данного составного оператора. Поиск объемлющего блока для определённого оператора в таком дереве вложенности можно проводить в глубину, пока не будет найден блок $b*$, одна из конструкций *binding* которого будет соответствовать данному оператору. Так как самый объемлющий блок — тело цикла, то такой поиск будет всегда завершаться.

4. Стратегия автоматического доказательства условий корректности в ACL2

При доказательстве условий корректности нашим методом возникает необходимость использования индукции из-за рекурсивного определения операции замены. Хотя ACL2 поддерживает доказательства по индукции, мы столкнулись с определёнными трудностями при проведении экспериментов. Мы предлагаем эвристический метод, позволяющий успешно доказать частичную корректность ряда примеров с финитной итерацией над изменяемыми массивами с выходом из цикла.

Идея состоит в проверке предположения, что постусловие программы описывает случаи в форме конъюнкции импликаций, случился или нет выход из тела цикла. ACL2 способна проверить истинность такого предположения. Если оно истинно, то это помогает более точно описать случаи в постусловии. Доказательство всех уточнённых случаев помогает доказать условие корректности.

На вход алгоритма поступает условие корректности ϕ, переменная n, определение функции rep, теория предметной области и постусловие.

В результате работы алгоритм выдаёт либо «ϕ истина», либо «неизвестно».

Данный алгоритм выглядит следующим образом:

1. Пусть M обозначает кортеж импликаций, являющихся конъюнктами постусловия. Рассмотрим кортеж N, такой что i-й элемент N является посылкой i-го элемента M. Переход на шаг 2.

2. Для каждого элемента из N выполнить шаг 3. Если результат истинен, добавить в теорию лемму, представляющую собой конъюнкцию ϕ и равенства i-го элемента N с $rep(\ldots).$*loop-break*. Иначе перейти на шаг 4. Если результат истинен, добавить в теорию лемму, представляющую собой конъюнкцию ϕ и равенства i-го элемента N с $\neg rep(\ldots).$*loop-break*. Перейти на шаг 5.

3. Пусть θ обозначает i-й элемент N. Пусть ω является конъюнкцией ϕ и равенства θ с $rep(\ldots).$*loop-break*. Проверить истинность ω в ACL2. Если ω была доказана, то вернуть «истина», иначе — «ложь».

4. Пусть θ обозначает i-й элемент N. Пусть ω является конъюнкцией ϕ и равенства θ с $\neg rep(\ldots).$*loop-break*. Проверить истинность ω в ACL2. Если ω была доказана, то вернуть «истина», иначе — «ложь».

5. Проверить истинность ϕ в ACL2 с помощью полученных лемм. Если ϕ была доказана, то вернуть «ϕ истина», иначе — «неизвестно».

Данный алгоритм может быть обобщён для использования с другими интерактивными доказателями и SMT-решателями, например CVC4 и Z3.

5. Эксперимент по автоматической верификации

В данном разделе мы продемонстрируем применение наших методов. Рассмотрим следующую функцию **negate_first** [5], которая в заданном одномерном массиве целых чисел меняет знак первого по порядку индексов отрицательного элемента.

```
void negate_first(int n, int* a) {
     int i;
     for (i = 0; i < n; i++) {
         if (a[i] < 0) {a[i] = -a[i]; break;}}}
```

Предусловие имеет вид:

```
(and (integer-listp a) (integer-listp a_0) (equal a a_0)
     (integerp n) (< 0 n) (<= n (length a_0)))
```

Постусловие выглядит следующим образом:

```
(let (((mv found-spec index-spec) (count_index n a_0)))
     (and (implies (not found-spec) (equal a a_0))
          (implies found-spec (equal a
              (update-nth index-spec (- (nth index-spec a_0)) a_0)))))
```

Определение *count_ index* дано в [8], приложение E. Данная функция возвращает пару: её первый компонент истинен тогда и только тогда, когда массив a содержит отрицательный элемент. В этом случае второй компонент содержит индекс такого элемента.

Структура данных типа $frame$, рекурсивное определение операции замены rep (см. [8], приложение F) и основанное на ней условие корректности были сгенерированы с помощью метода, изложенного в разделе 3. Условие корректности **mytheorem1** является конъюнкцией двух случаев: имеет ли массив отрицательный элемент или нет. Заметим, что постусловие верифицируемой функции также описывает эти два случая. Таким образом мы применили эвристический метод из раздела 4. В итоге была сгенерирована лемма об эквивалентности утверждения о выходе из цикла и утверждения о существовании отрицательного элемента в массиве. Условие корректности (вместе с данной леммой) имеет вид:

```
(defrule my-theorem1 (b* (((mv found-spec index-spec)
  (count_index n a_0)) ((frame fr) (rep n (frame-init a)))) (implies
  (and (integer-listp a) (integer-listp a_0) (equal a a_0) (integerp n)
  (< 0 n) (<= n (length a_0))) (and (implies (not found-spec)
  (equal fr.a a_0)) (implies found-spec (equal fr.a
  (update-nth index-spec (- (nth index-spec a_0)) a_0))))))
  :prep-lemmas ((defrule lemma (b* (((mv found-spec index-spec)
    (count_index n a_0)) ((frame fr) (rep n (frame-init a)))) (implies
    (and (integer-listp a) (integer-listp a_0) (equal a a_0)
    (integerp n) (< 0 n) (<= n (length a_0))) (and (equal fr.loop-break
     found-spec) (implies (not found-spec) (equal fr.a a_0)) (implies
     found-spec (equal fr.a (update-nth index-spec (- (nth index-spec
     a_0)) a_0))))))
  :enable (frame-init count_index rep)
  :induct (dec-induct n))))
```

Система ACL2 успешно доказала условие корректности в полученной теории.

Заключение

Данная статья представляет расширение системы верификации C-light программ [13]. В случае финитной итерации над изменяемыми массивами с выходом из цикла данное расширение позволяет порождать условия корректности без инвариантов циклов.

Эта генерация основана на правиле вывода для оператора **for** языка C-light, использующем операцию замены. Она выражает финитную итерацию в символическом виде. Операция замены порождается автоматически специальным алгоритмом.

Полученные условия корректности автоматически доказываются в ACL2 с помощью предложенного эвристического метода.

Отметим, что верификация функций, реализующих интерфейс BLAS [3], является известной проблемой. Ранее мы успешно выполнили такие эксперименты [7]. Наши методы позволили верифицировать функцию *asum*, реализующую соответствующую функцию из интерфейса BLAS, которая вычисляет сумму абсолютных значений вектора.

В дальнейшем мы планируем рассмотреть случаи более сложных структур данных и верифицировать другие функции интерфейса BLAS.

Список литературы / References

[1] Anureev I. S., Maryasov I. V., Nepomniaschy V. A., “C-programs Verification Based on Mixed Axiomatic Semantics”, *Automatic Control and Computer Sciences*, **45**:7 (2011), 485–500.

[2] Cohen E., Dahlweid M., Hillebrand M., Leinenbach D., Moskal M., Santen T., Schulte W., Tobies S., “VCC: A Practical System for Verifying Concurrent C”, 22nd Int. Conf. TPHOLs, *LNCS*, **5674** (2009), 23–42.

[3] Dongarra J. J., van der Steen A. J., “High-performance computing systems: Status and outlook”, *Acta Numerica*, **21** (2012), 379–474.

[4] Filliâtre J.-C., Marché C., “Multi-prover Verification of C Programs”, 6th ICFEM, *LNCS*, **3308** (2004), 15–29.

[5] Jacobs B., Kiniry J. L., Warnier M., “Java Program Verification Challenges”, FMCO 2002, *LNCS*, **2852** (2003), 202–219.

[6] Kaufmann M., Moore J.S., “An Industrial Strength Theorem Prover for a Logic Based on Common Lisp”, *IEEE Transactions on Software Engineering*, **23**:4 (1997), 203–213.

[7] Kondratyev D., “Implementing the Symbolic Method of Verification in the C-Light Project”, PSI 2017, *LNCS*, **10742** (2018), 227–240.

[8] Kondratyev D.A., “Towards Loop Invariant Elimination for Definite Iterations over Changeable Data Structures in C Programs Verification. Appendices”, https://bitbucket.org/c-light/loop-invariant-elimination.

[9] Kondratyev D.A., Maryasov I.V., Nepomniaschy V.A., “Towards Loop Invariant Elimination for Definite Iterations over Changeable Data Structures in C Programs Verification”, *PSSV 2018*, Workshop Proceedings, Yaroslavl, 2018, 51–57.

[10] Li J., Sun J., Li L., Loc Le Q., Lin S-W., “Automatic Loop Invariant Generation and Refinement through Selective Sampling”, *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, 782–792.

[11] Maryasov I.V., Nepomniaschy V.A., “Loop Invariants Elimination for Definite Iterations over Unchangeable Data Structures in C Programs”, *Modeling and Analysis of Information Systems*, **22**:6 (2015), 773–782.

[12] Maryasov I.V., Nepomniaschy V.A., Kondratyev D.A., “Invariant Elimination of Definite Iterations over Arrays in C Programs Verification”, *Modeling and Analysis of Information Systems*, **24**:6 (2017), 743–754.

[13] Maryasov I.V., Nepomniaschy V.A., Promsky A.V., Kondratyev D.A., “Automatic C Program Verification Based on Mixed Axiomatic Semantics”, *Automatic Control and Computer Sciences*, **48**:7 (2014), 407–414.

[14] Nepomniaschy V.A., “Symbolic Method of Verification of Definite Iterations over Altered Data Structures”, *Programming and Computer Software*, **31**:1 (2005), 1–9.

[15] Tuerk T., “Local Reasoning about While-Loops”, *VSTTE 2010*, Workshop Proceedings, 2010, 29–39.

Kondratyev D. A.[1], **Maryasov I. V., Nepomnyaschy V. A.**, "The Automation of C Program Verification by Symbolic Method of Loop Invariants Elimination", *Modeling and Analysis of Information Systems*, **25**:5 (2018), 491–505.

DOI: 10.18255/1818-1015-2018-5-491-505

Abstract. During deductive verification of programs written in imperative languages, the generation and proof of verification conditions corresponding to loops can cause difficulties, because each one must be provided with an invariant whose construction is often a challenge. As a rule, the methods of invariant synthesis are heuristic ones. This impedes its application. An alternative is the symbolic method of loop invariant elimination suggested by V.A. Nepomniaschy in 2005. Its idea is to represent a loop body in a form of special replacement operation under certain constraints. This operation expresses loop effect in a symbolic form and allows to introduce an inference rule which uses no invariants in axiomatic semantics. This work represents the further development of this method. It extends the mixed axiomatic semantics method suggested for C-light program verification. This extension includes the verification method of iterations over changeable arrays possibly with loop exit in C-light programs. The method contains the inference rule for iterations without loop invariants. This rule was implemented in verification conditions generator which is a part of the automated system of C-light program verification. To prove verification conditions automatically in ACL2, two algorithms were developed and implemented. The first one automatically generates the replacement operation in ACL2 language, the second one automatically generates auxiliary lemmas which allow to prove the obtained verification conditions in ACL2 successfully in automatic mode. An example which illustrates the application of the mentioned methods is described.

Keywords: C-light, loop invariants, mixed axiomatic semantics, definite iteration, arrays, ACL2, specification, verification, Hoare logic

On the authors:
Dmitry A. Kondratyev, orcid.org/0000-0002-9387-6735, postgraduate student,
A.P. Ershov Institute of Informatics Systems SB RAS,
6 Akademik Lavrentyev av., Novosibirsk 630090, Russia, e-mail: apple-66@mail.ru

Ilya V. Maryasov, orcid.org/0000-0002-2497-6484, PhD,
A.P. Ershov Institute of Informatics Systems SB RAS,
6 Akademik Lavrentyev av., Novosibirsk 630090, Russia, e-mail: ivm@iis.nsk.su

Valery A. Nepomniaschy, orcid.org/0000-0003-1364-5281, PhD,
A.P. Ershov Institute of Informatics Systems SB RAS,
6 Akademik Lavrentyev av., Novosibirsk 630090, Russia, e-mail: vnep@iis.nsk.su

Acknowledgments:
[1] This author is partially supported by RFBR, grant 17-01-00789.