ПРОГРАММИРОВАНИЕ, 2022, № 5, с. 37–53

ПРОГРАММНАЯ ИНЖЕНЕРИЯ, ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ ПРОГРАММ

УДК 004.052.42

Работа над статьей была завершена после кончины Валерия Александровича Непомнящего. Статья посвящается моему учителю, коллеге и соавтору Валерию Александровичу Непомнящему.

АВТОМАТИЗАЦИЯ ДЕДУКТИВНОЙ ВЕРИФИКАЦИИ С-ПРОГРАММ БЕЗ ИСПОЛЬЗОВАНИЯ ИНВАРИАНТОВ ЦИКЛОВ

© 2022 г. Д. А. Кондратьев[a,*] (http://orcid.org/0000-0002-9387-6735),
В. А. Непомнящий[a] (http://orcid.org/0000-0003-1364-5281)

[a] *Институт систем информатики им. А.П. Ершова СО РАН*
630090 Новосибирск, пр. Ак. Лаврентьева, 6, Россия

*****E-mail: apple-66@mail.ru*

Поступила в редакцию 01.12.2021 г.
После доработки 22.03.2022 г.
Принята к публикации 28.03.2022 г.

Автоматизация верификации С-программ – актуальная проблема современного программирования. Для ее решения необходимо автоматизировать решение проблемы инвариантов циклов, доказательство условий корректности и локализацию ошибок в случае ложных условий корректности. В Институте систем информатики СО РАН разрабатывается система C-lightVer, использующая комплексный подход к автоматизированной дедуктивной верификации С-программ. Данный подход включает символический метод верификации финитных итераций для элиминации инвариантов циклов, стратегии доказательства условий корректности и метод локализации ошибок. Символический метод верификации финитных итераций основан на замене действий циклов определенного вида применением специальной рекурсивной функции rep. Метод локализации ошибок основан на сопоставлении условий корректности с исходным кодом и генерации текста о соответствии условий корректности и фрагментов программы. Естественно возникает задача автоматизации верификации С-программ с вложенными циклами. Применение символического метода верификации финитных итераций для таких программ приводит к композиции функций rep для внешнего и вложенного циклов. Новым результатом этой статьи является стратегия автоматизации доказательства таких условий корректности. Данная стратегия основана на индукции по номеру итерации внешнего цикла. Для доказательства шага индукции мы используем другой результат этой статьи, которым является стратегия для программ, спецификации которых содержат функции со свойством конкатенации. В данной статье также представлены стратегии локализации ошибок и модификация метода локализации ошибок для случая вложенных циклов. Эти стратегии используются для проверки выполнения свойств циклов, которые могут означать наличие ошибок. В качестве примера применения наших результатов рассматривается автоматическая верификация сортировки простыми вставками без инвариантов циклов.

DOI: 10.31857/S0132347422050053

1. ВВЕДЕНИЕ

Актуальной проблемой современного программирования является автоматизация дедуктивной верификации программ [1–3]. Для решения этой проблемы необходимо автоматизировать задание инвариантов циклов [4, 5], доказательство условий корректности (УК) [6] и локализацию ошибок [7] в случае ложных УК.

В Институте систем информатики СО РАН разрабатывается система C-lightVer [8–11], основанная на комплексном подходе к автоматизации дедуктивной верификации С-программ. Данный подход включает символический метод верификации финитных итераций для элиминации инвариантов циклов [12], стратегии доказательства [9, 10] для проверки УК на истинность и метод локализации ошибок [7, 11].

Метод локализации ошибок основан на добавлении в правила вывода УК семантической разметки для объяснения результатов применения правил. Генератор УК в процессе вывода добавляет к различным подформулам соответствующие метки, которые извлекаются из УК и переводятся в текст о соответствии УК и фрагментов программы.

Для автоматизации дедуктивной верификации в системе C-lightVer используется ориентация на класс программ, осуществляющих финитные итерации над последовательностями данных [12]. Тело цикла финитной итерации исполняется один раз для каждого элемента последовательности данных. В символическом методе верификации финитных итераций используется специальное правило вывода УК для таких итераций, которое основано на применении операции замены (функции *rep*), выражающей действие цикла в символической форме. Функция *rep* определяется рекурсивно по номеру итерации [9]. Таким образом, УК программ с финитными итерациями могут содержать *rep*.

Но системы доказательства не всегда справляются в автоматическом режиме с доказательством УК, содержащих *rep*. Поэтому, были разработаны стратегии автоматизации доказательства таких УК [9, 10], реализованные для системы ACL2 [13].

Естественно возникает задача автоматизации верификации С-программ с вложенными циклами, например, программ линейной алгебры, программ сортировки массивов и программ, реализующих интерфейс BLAS. Применение символического метода верификации финитных итераций для таких программ приводит к композиции функций *rep* для внешнего и вложенного циклов. В данной статье описана стратегия автоматизации доказательства таких УК, которая основана на индукции по номеру итерации внешнего цикла. Для доказательства шага индукции разработана стратегия для программ, спецификации которых содержат функции со свойством конкатенации. Эта стратегия описана в разделе 4.1.

Также в данной статье описаны стратегии локализации ошибок и модификация метода локализации ошибок для случая вложенных циклов. Эти стратегии используются для проверки выполнения свойств циклов, которые могут означать наличие ошибок. Наш подход к локализации ошибок позволяет генерировать объяснения для таких случаев.

В качестве примера применения наших результатов в данной статье была использована сортировка массивов простыми вставками.

Данная статья состоит из шести разделов. Комплексный подход к автоматизации дедуктивной верификации описан во втором разделе. В третьем разделе описан метод автоматизации локализации ошибок в программах с финитными итерациями. Метод автоматизации доказательства УК программ с финитными итерациями описан в четвертом разделе. В пятом разделе описаны эксперименты по автоматизированной локализации ошибок и автоматической верификации программы сортировки простыми вставками.

1.1. Обзор литературы

Рассмотрим работы в области автоматизации решения проблемы инвариантов циклов. Подход, похожий на символический метод верификации финитных итераций, был предложен в работе [14], где предлагается заменять действие цикла на результат применения рекурсивной функции. Но в этой работе рассматривается модельный язык, более простой, чем язык C-light. Подобный подход [15] также предложен для Scala-программ. Но в работе [15] предлагается использовать инварианты, которые транслируются в аннотации соответствующих циклам рекурсивных функций. В отличие от символического метода верификации финитных итераций, большинство работ в этой области основано на генерации инвариантов циклов. Отметим работу [16], в которой предложен метод генерации инвариантов для класса циклов, называемых P-разрешимыми. Но, в отличие от финитных итераций, правые части присваиваний в теле P-разрешимых циклов должны иметь вид полиномов. В работе [17] предлагается генерировать инварианты специального вида для циклов над массивами, но циклы с инструкцией `break` не рассматриваются. Рассмотрим, как решается проблема инвариантов для программ сортировки. В работе [18] предложен динамический метод, основанный на известной идее модификации постусловия, но свойство перестановочности результирующего массива по отношению к исходному массиву не было доказано. В работе [19] предложено использовать шаблоны инвариантов, задаваемые пользователями, однако, доказано более слабое свойство, чем свойство перестановочности. В работе [20] предложено задавать инварианты специального вида, которые могут быть проще полных инвариантов и позволяют автоматизировать доказательство некоторых свойств. Этот подход применен для автоматизации верификации программы, реализующей обратную перестановку. Но в работе [20] для этой программы доказано выполнение более слабого свойства, чем свойство перестановочности.

Рассмотрим работы в области автоматизации доказательства УК. Подход [21] состоит в генерации лемм, которые могут помочь доказать целевую теорему. Такой подход для системы доказательства ACL2 предложен в работе [22]. В отличие от предложенных нами стратегий доказательства для системы ACL2, стратегии [22] основаны на машинном обучении. Но машинное обучение не подходит для автоматизации доказательства УК, так как предметные области могут отличаться для разных программ. Рассмотрим проблему автоматизации доказательства УК для программ сортировки вставками. Работа [23] была первой работой по использованию систем автоматизированного доказательства для верификации сортировки

вставками, но не все УК были доказаны автоматически. В работе [24] были предложены стратегии-доказательства, которые позволили автоматизировать доказательство некоторых УК. В работе [25] были предложены стратегии-автоматизации верификации свойства перестановочности. Но решение проблемы инвариантов циклов не было автоматизировано в работах [23–25].

Рассмотрим работы со специализированными стратегиями, ориентированными на упрощение формальной верификации. В работе [26] предложено задавать для циклов предусловия и постусловия специального вида вместо инвариантов. В работе [27] рассматривается метод леммафункций, реализованный в системе AstraVer. Этот метод основан на использовании спецификаций специального вида вместо инвариантов циклов. В работе [28] описан похожий метод, реализованный в системе Frama-C [29]. Но эти методы [26–28] основаны на задании спецификаций пользователем. Рассмотрим задание специализированных стратегий, ориентированных на упрощение верификации программ сортировки. В работе [30] на базе модельного функционального языка предложена стратегия для автоматизации верификации сортировки с помощью дизъюнктов Хорна.

Рассмотрим работы об автоматизации локализации ошибок при дедуктивной верификации программ. В работе [31] описано использование контрпримера, сгенерированного SMT-решателем, для локализации ошибок. Но анализ контрпримера может оказаться достаточно сложным, что продемонстрировано в работе [32]. В работе [7] предложен способ установления соответствия между УК и исходным кодом. В работе [33] описан новый подход в системе Frama-C [29] к автоматизации локализации ошибок при дедуктивной верификации, основанный на изменении выражений программы. В работе [34] предложена логика, названная некорректной логикой разделения. Истинность специальных формул в этой логике означает наличие ошибок в программе. Но авторы [34] предложили сложную модель памяти, приводящую к генерации сложных формул в отличие от используемого нами метода смешанной аксиоматической семантики [8]. Отметим работу [35] в области автоматизации локализации ошибок при дедуктивной верификации программ сортировки. В этой работе описана локализация ошибки при верификации реализации сортировки в Java-машине OpenJDK. Ошибка была локализована с помощью доказательства невыполнения инварианта в определенных случаях. Но решение проблемы инвариантов циклов не было автоматизировано в работах [33–35].

Рассмотрим работы про обоснование актуальности автоматизации дедуктивной верификации программ. В работе [36] предложен единый подход к доказательству корректности программ и к доказательству наличия ошибок в программах. В данной статье мы также используем общий подход к этим проблемам. Но наш подход основан на доказательстве выполнения свойств о функциях замены для циклов и мы, в отличие от работы [36], рассматриваем программы с вложенными циклами. В работах [37, 38] описана актуальность автоматизации дедуктивной верификации и путь к этой цели при разработке систем верификации Dafny-программ и Ada-программ. Из работ [39, 40] о верификации ядра Linux следует актуальность автоматизации дедуктивной верификации для таких задач. В работах [41, 42] содержатся рекомендации по добавлению автоматизированной локализации ошибок в систему AutoProof для верификации Eiffel-программ. В работе [43] описана новая система дедуктивной верификации C-программ, в которой используется автоматизация верификации с помощью доказателя теорем Coq. Но в этой работе не рассматривается решение проблемы инвариантов циклов. Отметим работу [44], в которой содержится обзор, подтверждающий актуальность автоматической верификации программ сортировки простыми вставками.

2. КОМПЛЕКСНЫЙ ПОДХОД К АВТОМАТИЗАЦИИ ДЕДУКТИВНОЙ ВЕРИФИКАЦИИ ПРОГРАММ

В Институте систем информатики СО РАН разрабатывается комплексный подход к автоматизации формальной верификации C-программ. Данный подход реализован в системе C-lightVer.

2.1. Система C-lightVer

Входным языком системы C-lightVer является язык C-light [45]. Данный язык является представительным подмножеством языка C. Для языка C-light была разработана операционная семантика [45]. C-kernel [46] – промежуточный язык верификации в системе C-lightVer. Данный язык является ограниченным подмножеством языка C-light. Для языка C-kernel определена аксиоматическая семантика [46].

Первым этапом исполнения программной системы C-lightVer является трансляция из C-light в C-kernel. Эта трансляция основана на наборе правил трансляции из конструкций языка C-light в эквивалентные конструкции C-kernel. На втором этапе генерируются УК полученной C-kernel программы. Родственный подход применяется в системе верификации Reflex-программ, где в качестве промежуточного языка рассматривается расширенный язык C-kernel [47].

В работе [8] описан метод смешанной аксиоматической семантики, который позволяет использовать специальные версии правил вывода

Рис. 1. Система C-lightVer.

для частных версий программных конструкций. Отметим, что в C-программах есть переменные, к которым не применяются операции взятия адреса и разыменования указателя. Для таких переменных можно использовать более простую модель памяти, чем модель, основанную на конструкциях взятия адреса и разыменования указателей. Поэтому, для конструкций над такими переменными можно применять более простые правила вывода. Это позволяет упростить УК.

Система верификации C-lightVer базируется на классическом дедуктивном подходе Хоара [5]. Схема системы C-lightVer изображена на рис. 1. Система состоит из семи основных модулей:

1. Модуль трансляции из C-light в C-kernel принимает на вход аннотированную C-light программу и производит

• трансляцию в эквивалентную аннотированную C-kernel программу;

• добавление информации [8], позволяющей транслировать конструкции C-kernel программы в конструкции C-light программы.

Эта информация добавляется к конструкциям, измененным в результате трансляции. Это позволяет снабдить конструкции названиями примененных правил трансляции.

2. Модуль генерации УК порождает формулы, из истинности которых следует корректность программы. Если программа содержит финитные итерации [12], то данный модуль генерирует операции замены для них.

3. Модуль управления доказательством УК применяет стратегии доказательства УК и запускает генерацию лемм, соответствующих данным стратегиям.

4. Модуль генерации лемм принимает на вход либо стратегию доказательства УК, либо стратегию локализации ошибок. Данный модуль генерирует леммы, соответствующие входным стратегиям. Леммы передаются на вход модулю доказательства теорем.

5. Модуль доказательства теорем проверяет истинность УК и лемм. В качестве данного модуля мы используем систему ACL2 [13].

6. Модуль локализации ошибок запускает исполнение стратегий локализации ошибок в случае недоказанного УК. Данный модуль генерирует объяснение недоказанных УК, используя семантические метки с информацией о соответствии между подформулами УК и конструкциями программы.

7. Модуль трансляции конструкций C-kernel в конструкции C-light основан на использовании ин-

формации, добавленной транслятором из C-light в C-kernel. Эта информация позволяет применить специальные правила трансляции [8] из C-kernel в C-light. Отчет о результатах верификации, который генерирует модуль локализации ошибок, основан на информации, хранящейся в семантических метках. Но метки содержат информацию не о конструкциях C-light программы, а о конструкциях C-kernel программы. Поэтому, используется данный модуль.

Рассмотрим этапы исполнения программной системы C-lightVer:

1. Аннотированная C-light программа транслируется в аннотированную C-kernel программу.

2. Генерируются УК полученной C-kernel программы.

3. Система ACL2 проверяет истинность УК. Доказанные УК передаются пользователю. Если все УК доказаны, то C-lightVer завершает исполнение.

4. Модуль управления доказательством последовательно применяет все подходящие стратегии для доказательства УК. Их применение состоит в генерации и доказательстве лемм. Доказанные леммы добавляются в теорию предметной области, которая используется в ACL2. Если применение стратегий привело к доказательству всех УК, то C-lightVer завершает исполнение.

5. Модуль локализации ошибок последовательно применяет все подходящие стратегии локализации ошибок. Их применение состоит в генерации и доказательстве лемм. Данный модуль также генерирует отчет о локализации ошибок. Базовой частью отчета является текст о соответствии УК и фрагментов программы. Если ACL2 доказала лемму, порожденную стратегией локализации ошибок, то в отчет добавляется текст, представляющий эту стратегию.

6. Конструкции C-kernel программы заменяются на конструкции C-light программы в отчете о локализации ошибок. Также номера строк C-kernel программы заменяются на номера строк C-light программы. Полученный отчет о локализации ошибок в C-light программе передается пользователю.

2.2. Символический метод верификации финитных итераций

Рассмотрим следующий вид цикла: *for x in S do* $v := body(v, x)$ *end*, где S – последовательность данных, x – переменная типа “элемент S”, v – вектор переменных цикла, который не содержит x, и *body* представляет тело цикла, которое не изменяет x и которое завершается для каждого $x \in S$. Тело цикла может содержать только инструкции присваивания, инструкции *if* (возможно вложенные) и инструкции *break*. Такие циклы *for* называются финитными итерациями [12]. Пусть v_0 – вектор значений переменных v до исполнения цикла. Чтобы выразить эффект финитной итерации, определим операцию замены $rep(n, v, S, body)$, где $rep(0, v, S, body) = v_0$, $rep(i, v, S, body) = = body(rep(i-1, v, S, body), s_i)$ для каждого $i = 1, 2, \ldots, n$. Если оператор выхода из цикла сработал на итерации i $(1 \le i \le n)$, то финитная итерация продолжает свое исполнение, но вектор v не изменяется: $\forall j (i \le j \le n)\ rep(i, v, S, body) = rep(j, v, S, body)$.

2.3. Генерация операции замены

Рассмотрим генерацию операции замены на языке системы ACL2 [13] в случае отсутствия вложенных итераций.

Пусть S – одномерный массив из n элементов примитивных типов языка C-light и $S \in v$, где v – вектор изменяемых переменных финитной итерации. Рассмотрим специальный случай финитной итерации над одномерным массивом S: *for* $(i = 0; i < n; i++)$ $v := body(v, i)$, где тело цикла является допустимой конструкцией [9].

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 **a = 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 $\{\mathbf{a_1\ a_2 \ldots a_{k-1}\ a_k}\}$, где a_r – допустимая конструкция для каждого $r: 1 \le r \le k$.

В вектор v входит S и переменные простого типа, которые могут изменяться в теле цикла.

Генерация операции замены основана на трансляции допустимых конструкций тела цикла в конструкции языка системы ACL2. Рассмотрим конструкцию (b^* (…(*var expr*)…) *result*), где конструкция вида (*var expr*) означает связывание переменной *var* со значением выражения *expr*. Выражение *expr* может зависеть от связанных ранее переменных. Значением b^* является значение *result*, которое может зависеть от связанных переменных. Значения переменных вектора изменяемых переменных v соответствуют значениям полей структуры *fr* типа *frame*. Поэтому для моделирования изменения значения переменной вектора

v мы связываем объект *fr* с новым объектом, который отличается от старого значением соответствующего поля. Для моделирования выхода из цикла мы используем булевское поле *loop–break* объекта *frame*. Это поле истинно только после срабатывания `break`. Для моделирования `break` мы используем связывание вида ((*when t*) *fr*). Так как в этом случае условием *when* является *t*, т.е. "истина", то такое связывание прекращает исполнение текущего блока *b** и возвращает *fr*.

Определим генератор операции замены как рекурсивную функцию *gen_rep* [11]:

- *gen_rep*(**empty statement**) = (*fr fr*)
- *gen_rep*(**break;**) = ((*when t*) *fr*)
- *gen_rep*(**c = b;**) = (*fr* (*change-frame fr* :*c b*))
- *gen_rep*(**a[i] = b;**) = (*fr* (*change-frame fr* :*a* (*update-nth i b fr.a*)))
- *gen_rep*(**if (c) b else d**) = (*fr* (*if c* (*b** (*gen_rep*(*b*)) *fr*) (*b** (*gen_rep*(*d*)) *fr*))) ((*when fr.loop-break*) *fr*)
- *gen_rep*($\{\mathbf{a}_1\ \mathbf{a}_2\ \ldots\ \mathbf{a}_{k-1}\ \mathbf{a}_k\}$) = (*fr* (*b** (*gen_rep*(a_1) … *gen_rep*(a_k)) *fr*)) ((*when fr.loop-break*) *fr*)

Результатом работы данного алгоритма для заданной финитной итерации является представление функции *rep* на языке системы ACL2, которая является операцией замены для этой финитной итерации.

2.4. Метод Денни и Фишера

Денни и Фишер предложили добавить в правила Хоара [6] семантическую разметку [7] для объяснения результата применения правила. Будем использовать обозначение $\lceil t \rceil_l$, означающее, что терму *t* сопоставляется метка *l*. Метки имеют вид *c*(*o*), где *c* – тип метки, *o* – диапазон строк. Денни и Фишер предложили несколько типов меток для разных видов подформул из спецификаций и программ. Каждому типу метки соответствует текстовый шаблон. Метод Денни и Фишера [7] состоит из следующих шагов:

1. Генерация УК с использованием правил вывода с семантической разметкой. При применении такого правила вывода в метках сохраняются номера строк кода исходной программы. В полученных УК подформулы помечены семантическими метками.

2. Задание порядка на семантических метках из УК с помощью создания специального списка меток. Это называется извлечением меток из УК. Денни и Фишер предложили извлекать метки из УК в порядке увеличения номеров соответствующих метке строк.

3. Генерация объяснения УК с помощью последовательного обхода полученного списка меток. Для каждой посещенной при обходе метки текст ее заполненного номерами строк шаблона добавляется к тексту, объясняющему УК.

2.5. Метод локализации ошибок

Рассмотрим работу модуля локализации ошибок системы C-lightVer. Данный модуль работает в том случае, если не удалось доказать истинность УК. В таком случае модуль локализации ошибок анализирует УК и применяет наш подход к локализации ошибок. Рассмотрим отличия нашего подхода [11] от подхода Денни и Фишера.

В нашем подходе используется не ограниченный, а произвольный набор концепций семантических меток. Эта возможность реализована как задание пользователем новых концепций семантических меток и правил вывода с этими метками. Для каждого типа метки пользователь системы верификации задает шаблон текста. Такие шаблоны подаются на вход генератору УК.

Для поддержки произвольных типов меток в системе C-lightVer язык описания правил вывода был расширен специальной конструкцией *label* [11], используемой для описания меток. Конструкция *label* имеет вид (*label t c*), где *t* – терм, снабженный меткой, а *c* – строка (тип метки).

В нашем подходе используется добавление семантических меток не только в подформулу УК, но и в определение *rep*. Во-первых, добавление семантических меток в тело функции *rep* позволяет автоматизировать верификацию циклов с помощью символического метода верификации финитных итераций [12]. Во-вторых, добавление семантических меток в тело функции *rep* позволяет сопоставлять финитные операции и определение операции замены для генерации подробных объяснений УК с применениями *rep*.

Так как определение операции замены представляет собой последовательность связываний в блоке *b** вектора изменяемых переменных с новыми значениями компонент, то для реализации семантической разметки была использована такая конструкция языка системы ACL2, как одновременное связывание. Эта конструкция позволяет связать с новым значением не только вектор изменяемых переменных, но и специальную переменную *label*, соответствующую семантической метке. Каждое такое связывание представляет собой присваивание новых значений метке

label и переменной *fr*, соответствующей вектору изменяемых переменных *v*. Метка *label* и структура *fr* связываются с конструкцией *cons*, которая создает пару значений. Первым компонентом этой пары является список, который содержит все атрибуты метки. Вторым компонентом этой пары являются новые значения компонент структуры *fr*. Это одновременное связывание позволяет снабдить семантической меткой присваивание новых значений вектору изменяемых переменных.

В нашем подходе используется извлечение семантических меток из УК не в порядке номеров строк, а порядке обхода дерева УК в глубину. Это позволяет улучшить генерацию текста в случае вложенных меток. Семантические метки из определения операции замены извлекаются с помощью обхода в глубину дерева кода функции *rep*. В случае применения функции *rep* в УК, дерево кода *rep* рассматривается как поддерево УК и включается в общий обход в глубину для извлечения меток.

В нашем подходе используются заданные пользователем шаблоны объяснений меток для генерации объяснения всего УК. При их задании можно использовать специальные символы для обозначения диапазона строк.

В нашем подходе используются стратегии локализации ошибок, которые проверяют, удовлетворяют ли конструкции программы определенным свойствам. Выполнение этих свойств может означать наличие ошибок в программе. Если применение стратегии приводит к доказательству выполнения какого-либо свойства, в объяснение УК добавляется текст о наличии соответствующей проблемы. Самым общим видом такой стратегии является проверка ложности УК. Ложность УК означает наличие ошибки в программе, либо в ее спецификациях. Но в системе ACL2 доказательство ложности формулы вызывает трудности. Так как переменные языка системы ACL2 находятся под квантором общности, то доказательство ложности УК приводит к появлению квантора существования. Поэтому, описанные стратегии не подходят для доказательства отрицания УК. Также необходимо, чтобы стратегия проверки ложности работала в случае цикла с инструкцией break. Поэтому, была разработана стратегия [11] проверки ложности недоказанного УК, содержащего операцию замены. Данная стратегия генерирует формулу, истинность которой означает ложность УК. Такая формула основана на импликации, где из посылок доказывается отрицание заключения. Актуальна задача создания набора автоматических стратегий проверки программ на наличие ошибок.

3. АВТОМАТИЗАЦИЯ ЛОКАЛИЗАЦИИ ОШИБОК ПРИ ДЕДУКТИВНОЙ ВЕРИФИКАЦИИ ПРОГРАММ

Для автоматизации локализации ошибок в программах с финитными итерациями мы расширили комплексный подход алгоритмом генерации операции замены для программ с вложенными циклами, стратегией поиска циклов с неиспользуемыми присваиваниями элементам массива и стратегией проверки исполнения инструкции `break` на первой итерации цикла.

3.1. Алгоритм генерации операции замены с семантическими метками для программ с вложенными циклами

Введем нумерацию финитных итераций в программе. Для первой финитной итерации генерируется функция rep_1, для второй генерируется rep_2 и т.д. Расположение финитных итераций в программе можно представить как последовательность деревьев, корнями которых являются финитные итерации на самом верхнем уровне вложенности программы. Потомками любой вершины этих деревьев являются циклы, вложенные в цикл, соответствующий вершине. Сначала нумеруются итерации из первого дерева вложенности, потом нумеруются итерации из второго дерева вложенности и т.д. Итерации внутри каждого дерева вложенности нумеруются путем обхода в ширину. При переходе к следующему дереву вложенности нумерация продолжается.

Определим трансляцию вложенных итераций на язык системы ACL2 с помощью функции *gen_rep*:

gen_rep(
for(i = 0; i < n; i + +) v := body(v, i) end)) =
((*cons* ?!*label fr*)
(*cons* (*list 'inner_iter begin end 'break_path*)
(rep_i *n* env_i fr_i))),

где

• '*inner_iter* – тип семантической метки, предложенный нами для вложенной финитной итерации;

• *begin*/(*end*) – специальная конструкция. На место этой конструкции алгоритм генерации операции замены вставляет номер первой (последней) строки вложенной итерации. Такая конструкция позволяет записывать начало (конец) диапазона строк итерации в список атрибутов метки. Хранение номеров диапазона строк итерации в семантической метке позволяет сопоставить применение функции rep_i и фрагмент программы;

• '*break_path* – опциональный атрибут метки. Алгоритм генерации операции замены добавляет

этот атрибут в список, когда вложенная итерация лежит на пути к одному из операторов выхода из цикла. Этот атрибут позволяет генерировать более подробные объяснения в случае, когда стратегия локализации ошибок позволяет доказать, что исполнение внешней итерации завершилось в результате исполнения `break`.

Конструкция, которую генерирует *gen_rep* для вложенной итерации, представляет собой одновременное связывание метки *label* со своим значением и вектора изменяемых переменных (структуры *fr*) со своим значением. Обе эти переменные связываются с парой, созданной конструкцией *cons*. Метка *label* связывается с первым элементом этой пары. Первым элементом этой пары является список атрибутов метки. Структура *fr* связывается со вторым элементом этой пары. Вторым элементом этой пары является применение функции rep_i к аргументам n, env_i, fr_i, где

- n – обозначение количества итераций,
- env_i – обозначение вектора неизменяемых переменных итерации i,
- fr_i – обозначение вектора изменяемых переменных итерации i.

Это одновременное связывание позволяет снабдить применение операции замены для вложенной итерации семантической меткой с соответствующими атрибутами.

Отметим, что определение операции замены для внешней итерации содержит применение функции *rep* для внутренней итерации. Эта зависимость усложняет автоматизацию доказательства и приводит к разработке новых стратегий доказательства УК, содержащих применение вложенных операций замены.

3.2. Стратегия поиска циклов с неиспользуемыми присваиваниями элементам массива

Пусть в цикле, реализующем финитную итерацию над массивом, содержатся присваивания элементам этого массива и значения элементов массива после исполнения цикла равны значениям элементов массива до исполнения цикла. Значит, эти присваивания могут быть неиспользуемыми операциями. Такая ситуация может означать наличие ошибки. Поэтому, можно предупредить пользователей системы верификации о таких присваиваниях и о содержащем такие присваивания цикле.

Стратегия поиска таких циклов проверяет каждый цикл над массивом с присваиваниями элементам массива. Пусть такой цикл встречается i-тым в коде программы. Стратегия основана на генерации и проверке истинности леммы $P \to (a = rep_i(a, args).a)$, где

- P – предусловие,
- a – массив, над которым исполняется финитная итерация,
- rep_i – операция замены для финитной итерации,
- $args$ – аргументы rep_i, вычисленные с помощью обратного прослеживания,
- $rep_i(a, args).a$ – массив a после исполнения цикла.

Если такую лемму удалось доказать, то с помощью метода локализации ошибок генерируется текст с соответствующим предупреждением.

3.3. Стратегия проверки исполнения инструкции `break` *на первой итерации цикла*

Пусть в цикле, реализующем финитную итерацию над массивом, присутствует операция `break` и эта операция всегда исполняется на первой итерации цикла. Значит, вместо этого цикла можно задать последовательность операций, соответствующих инициализирующему выражению и телу цикла. Такая ситуация может означает наличие ошибки. Поэтому, можно предупредить пользователей системы верификации о такой операции `break`, об условии, при котором исполняется такая операция `break`, и о содержащем такую операцию `break` цикле.

Стратегия проверяет каждый цикл над массивом с инструкцией `break`. Пусть такой цикл встречается i-тым в коде программы. Стратегия основана на генерации и проверке истинности следующей леммы:

$$P \to ((j_0 = rep_i(a, args).j) \wedge \\ \wedge (rep_i(a, args).loop\text{-}break)),$$

где

- P – предусловие,
- a – массив, над которым исполняется финитная итерация,
- j – счетчик цикла *for*, реализующего финитную итерацию,
- $args$ – аргументы rep_i, вычисленные с помощью обратного прослеживания. Также вычисляется j_0 – значение счетчика j до исполнения цикла,
- $rep_i(a, args).j$ – значение j после исполнения цикла,
- $loop\text{-}break$ – специальное поле в возвращаемой структуре данных. Его значение истинно тогда и только тогда, когда при исполнении цикла исполнилась инструкция `break`.

Таким образом, первый конъюнкт заключения леммы является утверждением, что исполнение цикла не изменило значения счетчика цикла. Второй конъюнкт из заключения является утвер-

ждением, что при исполнении цикла исполнилась инструкция `break`.

Если лемму удалось доказать, то вычислим условие исполнения `break` с помощью обратного прослеживания. Истинность леммы гарантирует, что итерации, следующие за первой, не исполняются. Отметим, что невозможно вычислить символически условие исполнения `break` на произвольной итерации цикла, так как неизвестно, сколько итераций исполнилось до этой итерации. Но, в случае истинности леммы, возможно символически вычислить условие исполнения break с помощью подстановки значения переменных цикла до итерации. Если лемма доказана, то с помощью метода локализации ошибок генерируется текст с предупреждением о такой инструкции `break`, об условии исполнения этой инструкции `break`, и о цикле с этой инструкцией `break`.

3.4. Автоматизация локализации ошибок для программ с вложенными циклами

Расширение комплексного подхода для локализации ошибок в случае вложенных циклов привело к модификации системы C-lightVer. Новые алгоритмы и стратегии были реализованы в соответствующих модулях данной системы.

Алгоритм генерации операции замены для программ с вложенными циклами был реализован в модуле генерации УК. Отметим, что этот алгоритм позволяет снабдить вложенные итерации специальными семантическими метками. Для такого типа меток мы предложили специальный шаблон текста с описанием структуры вложенной итерации. В случае программ с вложенными финитными итерациями данный текст добавляется к объяснению УК.

Стратегии локализации ошибок реализованы в модуле локализации ошибок и в генераторе лемм. Ранее модуль локализации ошибок только генерировал объяснения недоказанных УК [11]. Модифицированный модуль также запускает последовательное исполнение всех стратегий локализации ошибок в случае недоказанного УК. Если УК не удалось доказать, то к циклам, соответствующим операциям замены из этого УК, последовательно применяются стратегии:

• Стратегия поиска циклов с неиспользуемыми присваиваниями элементам массива;

• Стратегия проверки исполнения инструкции `break` на первой итерации цикла.

Если применение стратегии локализации ошибок позволяет доказать выполнение соответствующего свойства для цикла, то, можно предположить, что программа содержит ошибку. В таком случае модуль локализации ошибок запускает генерацию текста с предупреждением о возможной ошибке, используя шаблон для стратегии. Номера строк и выражения из программы подставляются в специальные места шаблона. Полученный текст добавляется к объяснению УК. Шаблоны для обеих рассмотренных стратегий – это вторая и третья часть отчета из раздела 5.2 без номеров строк и выражений программы.

4. АВТОМАТИЗАЦИЯ ДЕДУКТИВНОЙ ВЕРИФИКАЦИИ ПРОГРАММ БЕЗ ИНВАРИАНТОВ ЦИКЛОВ

Для автоматизации доказательства УК программ с финитными итерациями мы расширили комплексный подход стратегией для программ, спецификации которых содержат функции со свойством конкатенации, стратегией для программ с финитными итерациями над массивами и стратегией для программ с вложенными циклами.

4.1. Стратегия для программ, спецификации которых содержат функции со свойством конкатенации

Важным этапом автоматизации верификации является этап автоматизации доказательства УК. Доказательство УК, содержащих операцию замены для финитной итерации, основано на индукции по длине массива, над которым осуществляется данная итерация. При этом, использования индукции по длине массива не достаточно для доказательства УК в системе ACL2 [13], если итерация изменяет элементы массива или содержит инструкции `break`. Рассмотрим разработанную для таких случаев стратегию автоматизации доказательства.

Определим такое свойство предикатов, как свойство конкатенации. Будем говорить, что предикат R обладает свойством конкатенации, если для R выполнено свойство вида

$$R(i,k,u_1,\ldots,u_n) \wedge R(k+1,j,u_1,\ldots,u_n) \rightarrow \\ \rightarrow R(i,j,u_1,\ldots,u_n)$$

Будем говорить, что предикат R обладает свойством конкатенации со склейкой на границе по предикату f, если для R выполнено свойство следующего вида:

$$(R(i,k,u_1,\ldots,u_n) \wedge R(k+1,j,u_1,\ldots,u_n) \wedge \\ \wedge f(u_1[k],u_1[k+1]) \wedge \ldots f(u_m[k],u_m[k+1])) \rightarrow \\ \rightarrow R(i,j,u_1,\ldots,u_n)$$

Для каждого $m(1 \le m \le n)$ будем называть выражение $f(u_m[k],u_m[k+1])$ условием склейки на границе для свойства конкатенации.

Пусть A – исходный массив, B – массив, полученный в результате применения функции *rep* к массиву A. Рассматриваемая стратегия генериру-

ет утверждения о равенстве подмассивов массива A и подмассивов массива B. Эти подмассивы выбираются с помощью следующих эвристик для цикла *for*:

• если счетчик цикла с инструкцией `break` возрастает (убывает), то генерируется утверждение, что равны подмассивы массивов B и A, начинающиеся с итогового значения счетчика (с нуля) до длины массива минус 1 (до итогового значения счетчика);

• если i – счетчик цикла с выражениями вида $A[i + expr] = A[i]$, то генерируется утверждение, что подмассив массива B является результатом сдвига подмассива массива A;

• если счетчик цикла возрастает (убывает) на произвольной итерации цикла, то генерируется утверждение, что равны подмассивы массивов B и A, начинающиеся с текущего значения счетчика (с нуля) до длины массива минус 1 (до текущего значения счетчика).

Утверждения о равенстве этих подмассивов преобразуются в леммы, которые поступают на вход системе доказательства. В результате образуется множество D пар подмассивов, равенство которых удалось доказать. Первым элементом каждой пары является подмассив массива A, а вторым элементом является подмассив массива B, для которого удалось доказать равенство первому элементу пары.

Каждый предикат из постусловия проверяется на выполнение свойства конкатенации или свойства конкатенации со склейкой на границе по предикату, найденному с помощью синтаксического анализа теории предметной области. Для каждого предиката S, удовлетворяющего любому из этих свойств, выполним следующие действия. Рассматриваемая стратегия генерирует леммы о выполнении свойства S для вторых элементов пар D, исходя из выполнения свойства S для первых элементов пар D. Для предиката со свойством конкатенации со склейкой на границе стратегия генерирует леммы о выполнении условия склейки на границах вторых элементов пар D. Такие леммы помогают системе ACL2 доказать выполнение свойства S для подмассивов, полученных в результате объединения вторых элементов пар D. Следовательно, такие леммы заданы для автоматизации доказательства выполнения свойства конкатенации для результирующего массива.

4.2. Стратегия для программ с финитными итерациями над массивами

Пусть $a[i : j]$ – обозначение подмассива a от элемента с индексом i до элемента с индексом j (включая элементы $a[i]$ и $a[j]$). Пусть n – длина массива a. Пусть j-я итерация цикла является финитной итерацией над $a[0 : j - 1]$ и $j + 1$-я итерация является финитной итерацией над $a[0 : j]$. Такая ситуация может означать, что подмассив $a[j + 1 : n - 1]$ после исполнения j-й итерации равен подмассиву $a[j + 1 : n - 1]$ после исполнения $j + 1$-й итерации. В этом случае будем называть подмассив $a[j + 1 : n - 1]$ необработанной частью массива $a[0 \ldots n - 1]$. Аналогично будем называть подмассив $a[0 :]$ обработанной частью массива $a[0 : n - 1]$. Проверим истинность леммы о равенстве необработанной части массива после двух смежных итераций. В случае успеха добавим эту лемму в теорию предметной области.

Стратегия генерирует лемму для каждой финитной итерации. Пусть такой цикл встречается i-тым в коде программы. Сгенерируем лемму:

$$(P \wedge j \in N \wedge 0 < j < length(a)) \rightarrow$$
$$\rightarrow (rep_i(j, a, args).a)[j + 1 : length(a) - 1] =$$
$$= (rep_i(j + 1, a, args).a)[j + 1 : length(a) - 1],$$

где

• P – предусловие,

• a – массив, над которым исполняется финитная итерация,

• j – счетчик цикла *for*, реализующего финитную итерацию,

• N – множество натуральных чисел,

• $length$ – функция, вычисляющая длину массива,

• $args$ – аргументы rep_i, вычисленные с обратного прослеживания,

• $rep_i(j, a, args).a$ – массив a после j-той итерации,

• $rep_i(j + 1, a, args).a$ – массив a после $j + 1$-й итерации.

Такая лемма может быть полезна если предикаты из спецификации программы удовлетворяют свойству конкатенации. Так как свойство конкатенации определено с помощью разбиения массива на части, то доказательство равенства необработанных частей массива может облегчить доказательство выполнения свойства конкатенации.

4.3. Стратегия автоматизации доказательства условий корректности для программ с вложенными циклами

Рассмотрим случай, когда возрастание количества итераций внешнего цикла приводит к возрастанию части последовательности, обрабатываемой внутренним циклом. Тогда попытаемся доказать УК, используя индукцию по количеству итераций внешнего цикла.

Пусть УК имеет вид $P \rightarrow Q$, где

• Q – заключение УК, зависящее от rep_i,

• P – посылка УК,

• a – массив, над которым исполняется финитная итерация,

• n – длина массива,

• $args$ – аргументы rep_i, вычисленные с помощью обратного прослеживания,

• rep_i – функция замены для i-того цикла, тело которой содержит применение rep_{i+1}.

Стратегия заключается в доказательстве такого УК индукцией по n. Такая стратегия может быть полезна, если внутренний цикл определяет ту часть последовательности, над которой исполняется каждая итерация внешнего цикла. Тогда подстановка вместо операции замены для внешнего цикла ее определения приводит к формуле, где доказываемое свойство сформулировано относительно внутреннего цикла. Доказывать это свойство позволяет индукционная гипотеза о части последовательности, обрабатываемой внутренним циклом.

5. ЭКСПЕРИМЕНТЫ ПО АВТОМАТИЧЕСКОЙ ВЕРИФИКАЦИИ ПРОГРАММЫ СОРТИРОВКИ

Опишем эксперимент по автоматизированной локализации ошибки в сортировке простыми вставками и эксперимент по автоматической верификации данной сортировки.

5.1. Спецификации программы сортировки простыми вставками

Рассмотрим программу на языке C-light без инвариантов циклов, реализующую сортировку простыми вставками:

```
1. /* P */
2. void insertion_sort(int a[], int n){
3.     int k, i, j;
4.     for (i = 1; i < n; i++){
5.       k = a[i];
6.       for (j = i - 1; j >= 0; j-){
7.          if (a[j] <= k) break;
8.          a[j + 1] = a[j];}
9.       a[j + 1] = k;}}
10. /* Q */,
```

где спецификации записаны в конструкциях, задающих на языке C комментарии. Задание спецификаций в комментариях позволяет избежать влияния спецификаций на семантику исходной программы. P и Q обозначают предусловие и постусловие соответственно.

Отметим, что данная программа не содержит явных операций взятия адреса и разыменования указателей. Поэтому, метод смешанной аксиоматической семантики [8] позволяет нам использовать в данном случае простую модель памяти без конструкций взятия адреса и разыменования указателей. Это приводит к упрощению УК.

Также отметим, что данная программа относится к классу программ с вложенными циклами. Мы не задаем инварианты циклов для этой программы.

Спецификации и формулы в системе C-light-Ver задаются на языке системы ACL2, но для записи формул в этой статье мы используем классическую инфиксную нотацию.

Предусловие P является следующей формулой: $0 < n \le length(a_0) \wedge a = a_0$, где

• $length$ – функция, принимающая на вход массив и вычисляющая его длину,

• a_0 – вспомогательный массив, который хранит исходное значение массива a.

Вспомогательный массив используется для задания в постусловии исходного состояния массива a.

Постусловие Q является следующей формулой: $perm(0, n-1, a_0, a) \wedge ord(0, n-1, a)$, где

• $perm$ – предикат, задающий свойство перестановочности,

• ord – предикат, задающий свойство упорядоченности.

Конъюнкция свойств перестановочности и упорядоченности является свойством отсортированности массива.

Первым аргументом предиката *perm* является нижняя граница диапазона индексов. Вторым аргументом *perm* является верхняя граница диапазона индексов. Третьим и четвертым аргументами предиката *perm* являются массивы. Предикат *perm* проверяет, являются ли эти массивы перестановкой друг друга в диапазоне от нижней границы (включая нижнюю границу) до верхней границы (включая верхнюю границу).

Первым аргументом предиката *ord* является нижняя граница диапазона индексов. Вторым аргументом предиката *ord* является верхняя граница диапазона индексов. Третьим аргументом предиката *ord* является массив. Предикат *ord* проверяет, является ли этот массив упорядоченным по возрастанию в диапазоне от нижней границы (включая нижнюю границу) до верхней границы (включая верхнюю границу).

Предикаты *perm* и *ord* являются предикатами предметной области. Мы задали модуль для системы ACL2 с определением предиката *perm* и с теоремами о предикате *perm*. Этот модуль задан в файле “permut.lisp”. Данный файл доступен в репозитории [48]. Также мы задали модуль для системы ACL2 с определением предиката *ord* и с теоремами о предикате *ord*. Этот модуль задан в

файле “ordered.lisp”. Данный файл также доступен в репозитории [48].

5.2. Автоматизированная локализация ошибки в программе сортировки

Рассмотрим программу на языке C-light без инвариантов циклов, реализующую сортировку простыми вставками с внесенной ошибкой:

```
1. /* P */
2. void insertion_sort(int a[], int n){
3.     int k, i, j;
4.     for (i = 1; i < n; i++){
5.        k = a[i];
6.        for(j=i;j>=0;j-){
7.           if (a[j] <= k) break;
8.           a[j + 1] = a[j];}
9.        a[j + 1] = k;}}
10. /* Q */
```

Ошибка состоит в том, что инициализирующей инструкцией вложенного цикла `for` является инструкция `j=i` вместо инструкции `j=i-1`.

Следующее УК генерируется с помощью алгоритма генерации операции замены для программ с вложенными циклами:

$$0 < n \le length(a_0) \wedge a = a_0 \rightarrow$$
$$\rightarrow perm(0, n-1, a_0, rep_1(n, a, 1).a) \wedge$$
$$\wedge ord(0, n-1, rep_1(n, a, 1).a),$$

где

- rep_1 – операция замены для внешнего цикла,
- 1 – начальное значение счетчика внешнего цикла,
- $rep_1(n, a, 1).a$ – массив a после исполнения внешнего цикла.

Определения функций rep_1 и rep_2 снабжены семантическими метками в результате применения алгоритма генерации операции замены для программ с вложенными циклами. Определение функции rep_1 основано на применении операции замены rep_2 для внутреннего цикла. Ошибка в программе приводит к тому, что функция rep_2 применяется в теле функции rep_1 с неправильными аргументами. Вместо начального значения счетчика цикла $i-1$ в качестве аргумента функции rep_2 используется i.

Рассмотрим влияние ошибки в программе на исполнение i-той итерации внешнего цикла. Ошибка приводит к тому, что внутренний цикл всегда завершает исполнение на первой итерации из-за истинности условия инструкции *if*, содержащей `break` в положительной ветви. После завершения внутреннего цикла происходит присваивание элементу массива с индексом $i+1$ элемента массива с индексом i. В итоге, данное УК является истинным, когда элемент массива с индексом 0 имеет произвольное значение, а все остальные элементы совпадают и их значение не меньше значения нулевого элемента. Элемент с индексом 0 может отличаться от всех остальных, так как начальным значением счетчика внешнего цикла является 1, а не 0. Во всех остальных случаях это УК является ложным.

Система ACL2 не доказала истинность данного УК. Также система ACL2 не доказала ложность этого УК во всех случаях, используя проверку на истинность отрицания данного УК. Эта стратегия локализации ошибок не приводит к успеху из-за истинности УК в некоторых случаях. Модуль локализации ошибок в системе C-lightVer последовательно применяет стратегию поиска циклов с неиспользуемыми присваиваниями элементам массива и стратегию проверки исполнения инструкции `break` на первой итерации цикла.

Ошибка в программе приводит к тому, что вложенный цикл не изменяет массив a. Отметим, что вложенный цикл содержит присваивание элементам массива. Применение стратегии поиска циклов с неиспользуемыми присваиваниями элементам массива приводит к успешному доказательству соответствующей леммы. В репозитории [48] в файле “warnings.lisp” эта лемма задана как `warnings-lemma-11`.

Также ошибка в программе приводит к тому, что исполнение вложенного цикла всегда завершается на первой итерации. Инструкция `break` вложенного цикла всегда исполняется на первой итерации и вложенный цикл не изменяет значение счетчика цикла (переменной j). Поэтому, применение стратегии проверки исполнения инструкции `break` на первой итерации цикла приводит к успешному доказательству соответствующей леммы. В репозитории [48] в файле “warnings.lisp” эта лемма задана как `warnings-lemma-9`. Условие `a[i] <= a[i]` было символически вычислено, используя подстановку i вместо j и постановку $a[i]$ вместо k в условие инструкции *if*. Это условие исполнения инструкции `break` во вложенном цикле. Это условие вычислено для отчета о локализации ошибки. Отметим, что данное условие всегда истинно и является условием инструкции `if`. Эта информация является еще одним признаком наличия ошибки в программе.

В итоге, алгоритм генерации объяснений недоказанных УК для программ с вложенными циклами генерирует отчет, состоящий из трех частей. Рассмотрим первую часть отчета, которая является результатом применения метода локализации ошибок:

```
This formula corresponds to lines 1–10 in
function “insertion_sort”. This formula has
not been proven by C-ligthVer system using
ACL2 prover. Hence,
– the following explanation of this formula
    precondition from line 1
       implies
    postcondition goal from line 10
      with substitution loop effect
      from lines 4–9 by rep1
       that corresponds to the loop with
       initialization expression “i = 1”
       from line 4
       condition expression “i < n”
       from line 4
       iteration expression “i++”
       from line 4
       and the following loop body
       sequence of the following statements
       from line 5 to line 9
         assignment statement “k = a[i]”
         from line 5
         substitution of inner loop effect
         from lines 6–8 by rep2
         that corresponds to the inner loop
         with
           initialization expression “j=i”
           from line 6
           condition expression “j>=0”
           from line 6
           iteration expression “j--”
           from line 6 and
           the following loop body sequence of the following statements
           from line 7 to line 8
             if statement from line 7
               with condition “a[j] <= k”
               from line 7
               and with
               the following positive branch
               sequence of
               the following statements
               from line 7 to line 7
                 break statement
                 from line 7
           array update “a[j + 1] = a[j]”
           from line 8
       array update “a[j + 1] = k”
       from line 9
has been generated
```

Эта часть отчета позволяет сопоставить подформулы УК и исходную программу, а также сопоставить определение операции замены для внешнего цикла и исходную программу. Снабжение семантическими метками определения операции замены для внутренних циклов позволило сопоставить определение операции замены для внутреннего цикла и исходную программу.

Рассмотрим вторую часть отчета, которая является предупреждением о возможной ошибке, генерируемым стратегией поиска циклов с неиспользуемыми присваиваниями элементам массива:

```
- the following warning about inner loop
  assumption that precondition
  from line 1 holds,
  ensures that
  array update "a[j + 1] = a[j]"
  from line 8
  is always unused
has been generated
```

Эта часть отчета позволяет обратить внимание пользователя на вложенный цикл.

Рассмотрим третью часть отчета, которая состоит из двух предупреждений о возможной ошибке, генерируемых стратегией проверки исполнения инструкции `break` на первой итерации цикла:

```
- the following warning about inner loop
    assumption that precondition
    from line 1 holds,
    ensures that
    break statement
    from line 7 always executes
    at first iteration
  has been generated
- the following warning about inner loop
    assumption that precondition
    from line 1 holds,
    ensures that
    break condition "a[j] <= k"
    from line 7
    is always true at first iteration as
    "a[i] <= a[i]"
  has been generated
```

Эта часть отчета также позволяет пользователю обратить внимание на вложенный цикл. Кроме того, эта часть отчета позволяет обратить внимание пользователя на условие исполнения `break`.

Данный отчет позволяет обратить внимание пользователя на вложенный цикл и локализовать ошибку. Отметим, что пользователю не нужно просматривать всю программу, чтобы локализовать ошибку. Вместо изучения сложной структуры УК пользователю предоставляется текст о сопоставлении подформул УК и исходной программы. Комплексный подход к автоматизации дедуктивной верификации С-программ, реализованный в системе C-lightVer, позволяет упростить и автоматизировать локализацию ошибки в данном эксперименте.

5.3. Автоматическая верификации программы сортировки

Рассмотрим программу на языке C-light из раздела 5.1, реализующую сортировку простыми вставками.

Следующее УК генерируется с помощью алгоритма генерации операции замены для программ с вложенными циклами:

$$0 < n \le length(a_0) \wedge a = a_0 \rightarrow$$
$$\rightarrow perm(0, n-1, a_0, rep_1(n, a, 1).a) \wedge$$
$$\wedge ord(0, n-1, rep_1(n, a, 1).a),$$

где

- rep_1 – операция замены для внешнего цикла,
- 1 – начальное значение счетчика внешнего цикла,
- $rep_1(n, a, 1).a$ – массив a после исполнения внешнего цикла.

В репозитории [48] в файле “vc-1.lisp” данное УК задано как `vc-1-lemma-103`. Определение rep_1 задано в репозитории [48] в файле “rep1.lisp” как `rep1`. Определение rep_2 задано в репозитории [48] в файле “rep2.lisp” как rep2. Определение функции rep_1 основано на определении функции rep_2. Следовательно, УК содержит неявную композицию функций rep_1 и rep_2.

Система ACL2 не доказала истинность данного УК без применения стратегий доказательства. Так как спецификации УК содержат предикаты, которые могут удовлетворять свойству конкатенации, то модуль управления доказательством применил стратегию для программ, спецификации которых содержат функции со свойством конкатенации. Так как данное УК содержит вложенные циклы, то модуль управления доказательством применил стратегию для программ с финитными итерациями над массивами и стратегию для программ с вложенными циклами.

Спецификации рассматриваемой программы содержат функции со свойством конкатенации, так как для предиката *perm* выполнено свойство конкатенации и для предиката *ord* выполнено свойство конкатенации со склейкой на границе.

Предикат *perm* удовлетворяет свойству конкатенации, так как в теории предметной области содержится следующая лемма о свойствах *perm*:

$$(i \in N \wedge j \in N \wedge k \in N \wedge i \le k \wedge k < j \wedge$$
$$\wedge j < length(u) \wedge j < length(v) \wedge$$
$$perm(i,k,u,v) \wedge perm(k+1,j,u,v)) \rightarrow$$
$$\rightarrow perm(i,j,u,v),$$

где N – множество натуральных чисел. Выполнение свойства конкатенации автоматически доказывается благодаря этой теореме. В репозитории [48] в файле “permut.lisp” эта теорема задана как `permutation-7`.

Предикат *ord* удовлетворяет свойству конкатенации со склейкой на границе, так как в теории предметной области содержится следующая лемма о свойствах *ord*:

$$(i \in N \wedge j \in N \wedge k \in N \wedge i \le k < j \wedge$$
$$\wedge j < length(u) \wedge ord(i,k,u) \wedge ord(k+1,j,u) \wedge$$
$$\wedge u[k] \le u[k+1]) \rightarrow ord(i,j,u),$$

где N – множество натуральных чисел, $u[k] \le u[k+1]$ – условие склейки на границе для свойства конкатенации.

Эвристический анализ обнаружил в теле циклов инструкции присваивания элементам массива элементов того же самого массива. В результате, были сгенерированы леммы о равенстве следующих пар подмассивов:

1. $(rep_2(i,a,args).a)[0:j]$ и $a[0:j]$
2. $(rep_2(i,a,args).a)[j+2:i]$ и $a[j+1:i-1]$,

где

- a – массив, над которым исполняется финитная итерация,
- i – количество итераций внутреннего цикла в случае, если не исполнится break,
- *args* – аргументы rep_2, вычисленные с помощью обратного прослеживания,
- $rep_2(i,a,args).a$ – массив a после i-той итерации.

Равенство подмассивов $(rep_2(i,a,args).a)[0:j]$ и $a[0:j]$ означает, что вложенный цикл не изменил начало массива до этой позиции j. В репозитории [48] в файле “rep2.lisp” эта теорема задана как `rep2-lemma-76`.

Равенство подмассивов $(rep_2(i,a,args).a)[j+2:i]$ и $a[j+1:i-1]$ означает, что вложенный цикл привел к сдвигу на единицу последовательности $a[j+1:i-1]$. В репозитории [48] в файле “rep2.lisp” эта теорема задана как `rep2-lemma-55`.

Данные леммы о равенстве подмассивов были автоматически доказаны и добавлены в теорию предметной области.

Сравним результаты двух смежных итераций внешнего цикла. Результат следующей итерации отличается от результата предыдущей возрастанием отсортированной части массива на один элемент и убыванием неотсортированной части массива на один элемент. Отметим, что на смежных итерациях необработанные части последовательностей за вставляемым элементом равны. В данном случае необработанная часть последовательности внешнего цикла – это необработанная часть последовательности внутреннего цикла. Поэтому, применение стратегии для программ с финитными итерациями над массивами приводит к успешному доказательству соответствующей леммы и добавлению этой леммы в теорию предметной области. В репозитории [48] в файле “rep2.lisp” эта лемма задана как `rep2-lemma-53`.

Рассмотрим результат i-той итерации внешнего цикла. В результате такой итерации массив делится на две части: отсортированная часть до элемента с индексом i (включающая элемент с индексом i), и неотсортированная часть (не включающая элемент с индексом i). Значение i является границей между отсортированной частью массива и неотсортированной частью массива. Поэтому, в данном случае, стратегия автоматизации доказательства УК для программ с вложенными циклами является индукцией по границе между отсортированной и неотсортированной частью массива. Система ACL2 автоматически доказывает базу индукции.

Индукционная гипотеза является утверждением о выполнении свойств перестановочности, и упорядоченности от начала массива до i-го элемента (не включая i-тый) элемент. Леммы, порожденные примененными стратегиями, являются утверждениями о структуре массива после исполнения i-той итерации. Система ACL2 автоматически доказывает индукционный переход, используя данные леммы. Автоматическое доказательство базы индукции и индукционного перехода приводит к автоматическому доказательству данного УК.

Символический метод верификации финитных итераций позволяет избежать задания инвариантов для сортировки простыми вставками. Применение описанных стратегий доказательства позволяет автоматически доказать полученное УК. Итого, комплексный подход к автоматизации дедуктивной верификации С-программ, реализованный в системе C-lightVer, позволяет автоматически верифицировать программу сортировки простыми вставками на языке С.

6. ЗАКЛЮЧЕНИЕ

В данной статье представлены следующие основные результаты:

1. Описаны стратегии автоматизации доказательства УК:

• для программ с вложенными циклами;

• для программ, спецификации которых содержат функции со свойством конкатенации;

• для программ с финитными итерациями над массивами.

2. Описаны стратегии автоматизации локализации ошибок при дедуктивной верификации:

• поиска циклов с неиспользуемыми присваиваниями элементам массива;

• проверки исполнения инструкции `break` на первой итерации цикла.

Также представлены следующие алгоритмы:

1. Генерации операции замены для программ с вложенными циклами;

2. Генерации объяснений недоказанных УК программ с вложенными циклами для локализации ошибок.

Впервые проведен успешный эксперимент по автоматической верификации программы сортировки простыми вставками без инвариантов циклов. Также проведен эксперимент по автоматизированной локализации ошибки в этой программе.

Представленные результаты описывают расширение комплексного подхода к автоматизации дедуктивной верификации С-программ, реализованного в системе C-lightVer.

В будущем планируется расширить этот подход новыми стратегиями автоматизации доказательства УК и новыми стратегиями автоматизации локализации ошибок. Также планируется проведение экспериментов по автоматической верификации других классов программ с финитными итерациями и автоматизированной локализации ошибок в этих программах.

СПИСОК ЛИТЕРАТУРЫ

1. *Apt K.R., Olderog E.-R.* Assessing the Success and Impact of Hoare's Logic // Theories of Programming: The Life and Works of Tony Hoare. 2021. P. 41–76.
2. *Hähnle R., Huisman M.* Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools // Computing and Software Science. Lecture Notes in Computer Science. 2019. V. 10000. P. 345–373.
3. *Müller P., Shankar N.* The First Fifteen Years of the Verified Software Project // Theories of Programming: The Life and Works of Tony Hoare. 2021. P. 93–124.
4. *Furia C.A., Meyer B., Velder S.* Loop invariants: Analysis, classification, and examples // ACM Computing Surveys. 2014. V. 46. Is. 3. Article 34. P. 1–51.
5. *Apt K.R., Olderog E.-R.* Fifty years of Hoare's logic // Formal Aspects of Computing. 2019. V. 31. Is. 6. P. 751–807.
6. *Hoare C.A.R.* An axiomatic basis for computer programming // Communications of the ACM. 1969. V. 12. Is. 10. P. 576–580.
7. *Denney E., Fischer B.* Explaining verification conditions // Proc. AMAST 2008. Lecture Notes in Computer Science. 2008. V. 5140. P. 145–159.
8. *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. 2014. V. 48. Is. 7. P. 407–414.
9. *Kondratyev D.A., Maryasov I.V., Nepomniaschy V.A.* The Automation of C Program Verification by the Symbolic Method of Loop Invariant Elimination // Automatic Control and Computer Sciences. 2019. V. 53. Is. 7. P. 653–662.
10. *Kondratyev D., Maryasov I., Nepomniaschy V.* Towards Automatic Deductive Verification of C Programs over Linear Arrays // Proc. PSI 2019. Lecture Notes in Computer Science. 2019. V. 11964. P. 232–242.
11. *Kondratyev D.A., Promsky A.V.* The Complex Approach of the C-lightVer System to the Automated Error Localization in C-Programs // Automatic Control and Computer Sciences. 2020. V. 54. Is. 7. P. 728–739.
12. *Nepomniaschy V.A.* Symbolic method of verification of definite iterations over altered data structures // Programming and Computing Software. 2005. V. 31. Is. 1. P. 1–9.
13. *Moore J.S.* Milestones from the pure lisp theorem prover to ACL2 // Formal Aspects of Computing. 2019. V. 31. Is. 6. P. 699–732.
14. *Myreen M.O., Gordon M.J.C.* Transforming programs into recursive functions // Electronic Notes in Theoretical Computer Science. 2009. V. 240. P. 185–200.
15. *Blanc R., Kuncak V., Kneuss E., Suter P.* An overview of the Leon verification system: verification by translation to recursive functions // Proc. 4th Workshop on Scala. 2013. Article 1. P. 1–10.
16. *Humenberger A., Jaroschek M., Kovás L.* Invariant Generation for Multi-Path Loops with Polynomial Assignments // Proc. VMCAI 2018. Lecture Notes in Computer Science. 2018. V. 10747. P. 226–246.
17. *Chakraborty S., Gupta A., Unadkat D.* Inductive Reasoning of Array Programs Using Difference Invariants // Proc. CAV 2021. Lecture Notes in Computer Science. 2021. V. 12760. P. 911–935.
18. *Galeotti J.P., Furia C.A., May E., Fraser G., Zeller A.* Inferring loop invariants by mutation, dynamic analysis, and static checking // IEEE Transactions on Software Engineering. 2015. V. 41. Is. 10. P. 1019–1037.
19. *Srivastava S., Gulwani S., Foster J.S.* Template-based program verification and program synthesis // International Journal on Software Tools for Technology Transfer. 2013. V. 15. Is. 5–6. P. 497–518.
20. *Filliâtre J.-C.* Simpler proofs with decentralized invariants // Journal of Logical and Algebraic Methods in Programming. 2021. V. 121. Article ID: 100645.
21. *Johansson M.* Lemma Discovery for Induction // Proc. CICM 2019. Lecture Notes in Computer Science. 2019. V. 11617. P. 125–139.
22. *Heras J., Komendantskaya E., Johansson M., Maclean E.* Proof-pattern recognition and lemma discovery in ACL2 // Proc. LPAR 2013. Lecture Notes in Computer Science. 2013. V. 8312. P. 389–406.
23. *Filliâtre J.-C., Magaud N.* Certification of Sorting Algorithms in the Coq System // Proc. conf. on "Theorem

Proving in Higher Order Logics: Emerging Trends". Nice. 1999.

24. *Imine A., Ranise S.* Building Satisfiability Procedures for Verification: The Case Study of Sorting Algorithms // Proc. LOPSTR'03. 2003.

25. *Safari M., Huisman M.* A Generic Approach to the Verification of the Permutation Property of Sequential and Parallel Swap-Based Sorting Algorithms // Proc. IFM 2020. Lecture Notes in Computer Science. 2020. V. 12546. P. 257–275.

26. *Tuerk T.* Local reasoning about while-loops // Proc. Theory Workshop at VSTTE 2010. 2010. P. 29–39.

27. *Volkov G., Mandrykin M., Efremov D.* Lemma functions for Frama-C: C programs as proofs // Proc. 2018 Ivannikov ISP RAS Open Conference. 2018. P. 31–38.

28. *Blanchard A., Loulergue F., Kosmatov N.* Towards full proof automation in Frama-C using auto-active verification // Proc. NFM 2019. Lecture Notes in Computer Science. 2019. V. 11460. P. 88–105.

29. *Baudin P., Bobot F., Bühler D., Correnson L., Kirchner F., Kosmatov N., Maroneze A., Perrelle V., Prevosto V., Signoles J., Williams N.* The dogged pursuit of bug-free C programs: the Frama-C software analysis platform // Communications of the ACM. 2021. V. 64. Is. 8. P. 56–68.

30. *de Angelis E., Fioravanti F., Pettorossi A., Proietti M.* Proving Properties of Sorting Programs: A Case Study in Horn Clause Verification // Proc. HCVS/PERR 2019. Electronic Proceedings in Theoretical Computer Science. 2019. V. 296. P. 48–75.

31. *Dailler S., Hauzar D., Marché C., Moy Y.* Instrumenting a weakest precondition calculus for counterexample generation // Journal of Logical and Algebraic Methods in Programming. 2018. V. 99. P. 97–113.

32. *Becker B., Lourenço C.B., Marché C.* Explaining Counterexamples with Giant-Step Assertion Checking // Proc. F-IDE 2021. Electronic Proceedings in Theoretical Computer Science. 2021. V. 338. P. 82–88.

33. *Könighofer R., Toegl R., Bloem R.* Automatic error localization for software using deductive verification // Proc. HVC 2014. Lecture Notes in Computer Science. 2014. V. 8855. P. 92–98.

34. *Raad A., Berdine J., Dang H.H., Dreyer D., O'Hearn P., Villard J.* Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic // Proc. CAV 2020. Lecture Notes in Computer Science. 2020. V. 12225. P. 225–252.

35. *de Gouw S., de Boer F.S., Bubel R., Hähnle R., Rot J., Steinhöfel D.* Verifying OpenJDK's Sort Method for Generic Collections // Journal of Automated Reasoning. 2019. V. 62. Is. 1. P. 93–126.

36. *Möller B., O'Hearn P., Hoare T.* On Algebra of Program Correctness and Incorrectness // Proc. RAMiCS 2021. Lecture Notes in Computer Science. 2021. V. 13027. P. 325–343.

37. *Grebing S., Klamroth J., Ulbrich M.* Seamless Interactive Program Verification // Proc. VSTTE 2019. Lecture Notes in Computer Science. 2020. V. 12031. P. 68–86.

38. *Dailler S., Marché C., Moy Y.* Lightweight Interactive Proving inside an Automatic Program Verifier // Proc. F-IDE 2018. Electronic Proceedings in Theoretical Computer Science. 2018. V. 284. P. 1–15.

39. *Mandrykin M.U., Khoroshilov A.V.* Towards Deductive Verification of C Programs with Shared Data // Programming and Computing Software. 2016. V. 42. Is. 5. P. 324–332.

40. *Efremov D., Mandrykin M., Khoroshilov A.* Deductive verification of unmodified Linux kernel library functions // Proc. ISoLA 2018. Lecture Notes in Computer Science. 2018. V. 11245. P. 216–234.

41. *de Carvalho D., Hussain R., Khan A., Khazeev M., Lee JY., Masiagin S., Mazzara M., Mustafin R., Naumchev A., Rivera V.* Teaching programming and design-by-contract // Proc. ICL 2018. Advances in Intelligent Systems and Computing. 2020. V. 916. P. 68–76.

42. *Khazeev M., Mazzara M., Aslam H., de Carvalho D.* Towards a broader acceptance of formal verification tools // Proc. ICL 2019. Advances in Intelligent Systems and Computing. 2020. V. 1135. P. 188–200.

43. *Sammler M., Lepigre R., Krebbers R., Memarian K., Dreyer D., Garg D.* RefinedC: automating the foundational verification of C code with refined ownership types // Proc. 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 2021. P. 158–174.

44. *Jiang D., Zhou M.* A comparative study of insertion sorting algorithm verification // Proc. 2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference. 2017. P. 321–325.

45. *Nepomniaschy V.A., Anureev I.S., Mikhailov I.N., Promskii A.V.* Towards verification of C programs. C-light language and its formal semantics // Programming and Computing Software. 2002. V. 28. Is. 6. P. 314–323.

46. *Nepomniaschy V.A., Anureev I.S., Promskii A.V.* Towards Verification of C Programs: Axiomatic Semantics of the C-kernel Language // Programming and Computing Software. 2003. V. 29. Is. 6. P. 338–350.

47. *Anureev I.S., Garanina N.O., Lyakh T.V., Rozov A.S., Zyubin V.E., Gorlatch S.P.* Dedicative Verification of Reflex Programs // Programming and Computing Software. 2020. V. 46. Is. 4. P. 261–272.

48. *Kondratyev D.A.* Automatic verification of insertion sorting. https://bitbucket.org/Kondratyev/verify-loops (Accessed 11 Nov 2021)