Моделирование и анализ информационных систем. Т. 26, № 4 (2019), с. 502–519
Modeling and Analysis of Information Systems. Vol. 26, No 4 (2019), pp. 502–519

DOI: 10.18255/1818-1015-2019-4-502-519

УДК 004.052.42

Комплексный подход системы C-lightVer к автоматизированной локализации ошибок в C-программах

Кондратьев Д. А., Промский А. В.

Поступила в редакцию 23 сентября 2019
После доработки 25 ноября 2019
Принята к публикации 27 ноября 2019

Аннотация. В ИСИ СО РАН разрабатывается система C-lightVer для дедуктивной верификации C-программ. Исходя из двухуровневой архитектуры системы, входной язык C-light транслируется в промежуточный язык C-kernel. Метагенератор условий корректности принимает на вход C-kernel программу и логику Хоара для C-kernel. Для решения известной проблемы задания инвариантов циклов выбран подход финитных итераций. Тело цикла финитной итерации исполняется один раз для каждого элемента структуры данных конечной размерности, а правило вывода для них использует операцию замены rep, выражающую действие цикла в символической форме. Также в нашем метагенераторе внедрен и расширен метод семантической разметки условий корректности. Он позволяет порождать пояснения для недоказанных условий и упрощает локализацию ошибок. Наконец, если система ACL2 не справляется с установлением истинности условия, можно сосредоточиться на доказательстве его ложности. Ранее нами был разработан способ доказательства ложности условий корректности для системы ACL2. Необходимость в более подробных объяснениях условий корректности, содержащих операцию замены rep, привела к изменению алгоритмов генерации операции замены, извлечения семантических меток и генерации объяснений недоказанных условий корректности. В статье представлены модификации данных алгоритмов. Эти изменения позволяют пометить исходный код функции rep семантическими метками, извлекать семантические метки из определения rep, а также генерировать описание условия исполнения инструкции break.

Ключевые слова: дедуктивная верификация, семантическая метка, локализация ошибок, C-lightVer, ACL2, метагенератор условий корректности, финитная итерация, стратегия доказательства

Для цитирования: Кондратьев Д. А., Промский А. В., "Комплексный подход системы C-lightVer к автоматизированной локализации ошибок в C-программах", *Моделирование и анализ информационных систем*, **26**:4 (2019), 502–519.

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

Промский Алексей Владимирович, orcid.org/0000-0002-5963-2390, канд. физ.-мат. наук,
Институт систем информатики им. А. П. Ершова СО РАН,
пр. Академика Лаврентьева, 6, г. Новосибирск, 630090 Россия, e-mail: promsky@iis.nsk.su

Благодарности:
Исследование выполнено при частичной финансовой поддержке РФФИ в рамках научного проекта № 17-01-00789.

Введение

Автоматизация верификации С-программ – актуальная проблема современного программирования. В ИСИ СО РАН разрабатывается система C-lightVer [17] для автоматизированной дедуктивной верификации С-программ [16]. Входной язык C-light – значительное подмножество стандарта С99. В языке C-light было выделено ограниченное ядро – язык C-kernel, обладающий непротиворечивой аксиоматической семантикой. Процесс верификации разбивается на три этапа. На первом этапе аннотированная C-light программа транслируется в эквивалентную программу на языке C-kernel. Далее, с помощью аксиоматической семантики выводятся условия корректности (УК) [2]. Затем происходит доказательство полученных УК с помощью системы автоматизированного доказательства теорем. При разработке системы C-light возникла задача расширения генератора УК. Для решения задачи была использована концепция метагенерации УК [26]. Правила вывода аксиоматической семантики поступают на вход метагенератору и сопоставляются с программными конструкциями для вывода УК [16].

Правило вывода условий корректности для циклов языка C-kernel основано на использовании инварианта [24]. Инвариант цикла – это утверждение, которое истинно как перед исполнением цикла, так и для каждой итерации цикла, и обеспечивает корректность на выходе из цикла. Но в общем случае генерация инварианта цикла – алгоритмически неразрешимая задача. В проекте C-light мы реализовали обработку циклов специального вида – финитных итераций над структурами данных [27]. Тело цикла финитной итерации исполняется один раз для каждого элемента структуры данных конечной размерности. Чтобы избежать задания инвариантов для таких циклов, в системе C-lightVer реализован символический метод верификации финитных итераций [14]. В данном методе для таких итераций используется специальное правило вывода УК, основанное на функции *rep*, называемой операцией замены и выражающей действие цикла в символической форме.

Функция *rep* определяется рекурсивно по номеру итерации. В ходе нашей работы был разработан алгоритм автоматической генерации операции замены [15]. Метод метагенерации позволил дополнить систему C-lightVer правилом вывода для финитной итерации [14]. Если C-kernel программа содержит финитную итерацию, то метагенератор порождает для нее условия корректности, основанные на операции замены. В системе C-lightVer используется система автоматизированного доказательства теорем ACL2 [25]. Но использование классической индукции в системе ACL2 недостаточно для автоматического доказательства содержащих *rep* УК. Ранее нами были разработаны стратегии доказательства [15, 18], основанные и на индукции, и на структуре УК, и на структуре финитной итерации.

Для решения проблемы автоматизации дедуктивной верификации необходимо автоматизировать аннотирование циклов инвариантами, доказательство УК и локализацию ошибок в случае ложных УК [9]. Система C-lightVer использует комплексный подход к автоматизированной дедуктивной верификации С-программ. Данный подход включает символический метод верификации финитных итераций [27], который позволяет избежать задания инвариантов циклов, реализующих финитные итерации, стратегии доказательства для проверки УК на истинность [15, 18] и метод семантической разметки Денни и Фишера для локализации ошибок [5].

Денни и Фишер предложили добавить в правила вывода УК семантическую разметку для объяснения результата применения правила. Система C-lightVer была дополнена такими правилами с помощью метода метагенерации [14]. Ключевая особенность подхода Денни и Фишера состоит том, что различные подформулы располагаются на специальных позициях в правилах вывода, и, исходя из этого, генератор УК добавляет соответствующие метки к ним. Для генерации объяснений УК метки извлекаются из них, сортируются по номерам строк и переводятся в текст на естественном языке. Чтобы в таком тексте номера строк были указаны именно для C-light программы, был создан протокол, позволяющий от конструкций C-kernel программы вернуться к конструкциям исходной C-light программы. Протокол реализован с помощью транслятора из C-light в C-kernel и обратного транслятора [24]. Транслятор добавляет в код метаинформацию, которая используется обратным транслятором. Он устанавливает соответствие между диапазонами номеров строк обеих программ. Такие диапазоны являются результатом анализа семантических меток из УК.

Описанные методы анализа и верификации программ были реализованы в системе C-light ранее. Но появляется задача разбора случаев, если системе ACL2 не удалось доказать истинность УК. Отметим, что доказательство ложности УК гарантирует наличие ошибки в программе или в ее спецификациях. В таком случае модуль локализации ошибок может сообщать о несоответствии программы и спецификации. Поэтому для более точного анализа УК необходимо разработать стратегии проверки их ложности. Так как переменные ACL2 находятся под неявным квантором всеобщности, то доказательство отрицания УК приводит к появлению квантора существования. Поэтому наши стратегии [15, 18] не подходят для доказательства отрицания УК. Проблема состоит также в том, чтобы стратегия проверки ложности работала в случае цикла с оператором `break`. Ранее нами был разработан способ доказательства ложности условий корректности для системы ACL2 [17]. Другой проблемой является использование семантических меток для объяснения УК, содержащего функцию *rep*. Согласно методу Денни и Фишера меткой помечается вызов функции *rep*, который не содержит информации о структуре цикла. Для генерации более подробных объяснений условий корректности, содержащих операцию замены, в статье представлен разработанный нами способ помечать семантическими метками код функции *rep*. Использование семантических меток в определении функции *rep* привело к изменению алгоритмов генерации операции замены, извлечения семантических меток и генерации объяснений недоказанных условий корректности. В статье представлены разработанные нами модификации данных алгоритмов.

Данная статья имеет следующую структуру. В разделе 1. описаны методы, реализованные ранее в системе C-lightVer и составляющие основу комплексного подхода к дедуктивной верификации C-программ. В разделах 2. и 3. приведены разработанные нами ранее стратегии доказательства. В разделе 2. описана вспомогательная стратегия проверки формулы на истинность, а в разделе 3. – стратегия проверки ложности УК. В разделе 4. представлены наши новые результаты, расширяющие комплексный подход системы C-lightVer. В разделе 4. приведены модифицированные алгоритмы генерации операции замены, извлечения семантических меток и генерации объяснений недоказанных УК. В разделе 5. показан пример, де-

монстрирующий применение наших новых результатов. В заключении дан список наших новых результатов и описаны наши планы.

Обзор литературы. Существуют различные методы автоматизации дедуктивной верификации программ с циклами. Но в отличие от символического метода верификации финитных итераций, данные методы основаны на генерации инвариантов циклов в определенных случаях. Ли и др. [22] разработали обучающийся алгоритм генерации инвариантов циклов, но их метод не допускает наличие оператора `break` в теле цикла, а также операций над массивами. Туэрк [30] предложил использовать пред- и постусловия для циклов типа `while`, однако их должны задавать пользователи. Галеотти и др. [8] улучшили известный метод изменения постусловия комбинацией генерации тестов и динамическим нахождением инварианта. Тем не менее с помощью этого подхода им не удалось вывести полный инвариант в программах сортировки. Кроме того, Галеотти и др. [8] не доказывали свойство, что отсортированный массив является перестановкой исходного. Сривастава и др. [29] предложили метод, основанный на шаблонах инвариантов, предоставляемых пользователями. Но этот метод не позволил использовать в постусловии свойство перестановочности, необходимое для задания свойства отсортированности. В экспериментах по верификации программ сортировки Сривастава и др. задавали постусловие с использованием более слабого свойства, чем свойство перестановочности. Ковач [20] разработала метод автоматической генерации инвариантов цикла для P-разрешимых циклов. Главные отличия данного метода от символического метода верификации финитных итераций состоят в ограничениях на тело цикла: 1) правые операнды инструкций присваивания должны иметь вид полиномов, 2) в методе [20] не рассматривается инструкция `break`.

Для сравнения с комплексным подходом, применяемым в системе C-lightVer, рассмотрим два класса исследований, основанных на стратегиях автоматизации доказательства по индукции.

В обзорной работе [11] рассматривается класс исследований, основанный на стратегиях генерации таких вспомогательных лемм, которые могут помочь доказать исходную теорему. Свойство полноты для таких методов невозможно гарантировать, а вопрос корректности сводится только к вопросу корректности работы применяемых систем автоматизированного доказательства. Примером реализации такого подхода является система ACL2(ml) [10]. Генерация вспомогательных лемм в данной системе основана на комбинации двух методов. Первым методом является распознавание шаблонов доказательств с помощью статистического машинного обучения. Вторым методом служит символический метод нахождения аналогичной леммы. Но теории предметной области могут серьезно отличаться для разных программ. Поэтому методы машинного обучения плохо подходят для доказательства УК. Отметим, что Регеру и др. для проекта по расширению возможностей системы Vampire [28] пришлось разрабатывать новые стратегии автоматизации доказательства по индукции.

Другой класс исследований основан на узкоспециализированных стратегиях, ориентированных на упрощение формальной верификации [32]. Отметим, что наличие циклов приводит к необходимости автоматизации доказательства по индукции [23]. Так как для решения этой проблемы нами был разработан комплексный подход, основанный на стратегиях доказательства, то применяемый в системе

C-lightVer подход можно отнести к данному классу. Кроме него в качестве примера можно рассмотреть метод формальной верификации CLP [1]. Он основан на моделировании программы логическими конструкциями. Для обработки такой модели необходимо использовать специальные стратегии. Но стратегии проекта CLPT [1] ориентированы только на обработку предикатов. Также в качестве примера можно рассмотреть систему AstraVer [6] и используемый в ней метод лемма-функций [31]. Данная стратегия основана на задании спецификаций определенного вида, что облегчает задачу по сравнению с заданием инвариантов, но не позволяет достигнуть полной автоматизации. Подобная стратегия применена и в проекте по расширению возможностей системы Frama-C [3].

Дедуктивная верификация программ основывается на построении и выводе УК. Но для локализации ошибок построения и вывода УК недостаточно, необходимо произвести анализ УК, объясняющий структуру УК. Актуальность проблемы автоматизации локализации ошибок при использовании дедуктивной верификации была продемонстрирована на примере обучения студентов применению системы AutoProof для верификации Eiffel-программ [4, 12]. Кенигхофер и др. [19] реализовали в системе Frama-C автоматизированную локализацию ошибок в C-программах с помощью дедуктивной верификации. Их подход основан на внесении изменений в выражения верифицируемой программы. Но в их подходе не поддерживается полная автоматизация, так как инварианты циклов задает пользователь системы верификации. Рассмотрим два других проекта, основанных на формальном подходе к локализации ошибок. В проекте Centaur [7] УК анализируются для поиска условных выражений из исходных условных операторов и циклов. В проекте Лейно [21] базовая логика расширена метками, предоставляющими пригодную для объяснения семантическую информацию. В проекте Centaur [7] используются некоторые алгоритмы из области отладки программ. В проекте Лейно [21] метки вводятся на этапе трансляции в промежуточный язык, к которому применяется стандартный генератор. Также в этих двух проектах используются более простые, чем C, входные языки.

1. Методы комплексного подхода C-lightVer

Для описания основы комплексного подхода к дедуктивной верификации C-программ рассмотрим методы, реализованные ранее в системе C-lightVer.

Метод метагенерации условий корректности. Так как аксиоматическая семантика представляет собой набор правил вывода и аксиом для всех конструкций языка программирования, то для вывода УК метагенератор принимает в качестве аргумента правила вывода и аксиомы аксиоматической семантики. Они задают собой шаблоны для сопоставления с программными конструкциями [16]. Основой языка шаблонов является логика первого порядка и грамматика языка C. В выражениях этого языка сохраняются нетерминальные символы (неинтерпретированные предикатные символы и “фрагментные переменные” [26] для обозначения фрагментов кода). Принадлежность метаданных определенному классу в языке шаблонов

задается в явном виде [16]. Например, конструкция $any\_code(S)$ может соответствовать любой последовательности (включая пустую) конструкций языка программирования. Для описания конструкции $vector\_substitution$ введем обозначения. Пусть вектор $v = \langle v_1 \dots v_k \rangle$ и для каждого $j (1 \le j \le k)$ выражение $expr_j$ является результатом замены всех вхождений терма $vector\_element$ в выражении $expr$ на v_j. Тогда $vector\_substitution(T,\ vec,\ expr)$ обозначает одновременную замену для каждого $j (1 \le j \le k)$ всех вхождений v_j в формуле T на выражение $expr_j$.

Метод семантической разметки. Денни и Фишер [5] предложили добавить в правила Хоара семантическую разметку для объяснения результата применения правила. Будем использовать для помеченных термов $\lceil t \rceil^l$, означающее, что терму t сопоставляется метка l. Метки имеют вид $c(o, n)$, где c – концепция (тип метки), o – диапазон строк, n – дополнительная информация.

Для каждого типа метки пользователь системы верификации задает шаблон текста. Такие текстовые шаблоны подаются на вход метагенератору. Они подобны форматным строкам в языке Си, так как при их задании можно использовать специальные символы для обозначения диапазона строк.

Для поддержки произвольных концепций меток в проекте C-light язык описания правил вывода был расширен специальной конструкцией $label$ [14], используемой для описания меток. Конструкция $label$ имеет вид $(label\ t\ c)$, где t — терм, к которому приписана метка, а c – строка (тип метки).

Денни и Фишер предложили извлекать метки из УК в порядке увеличения номеров соответствующих метке строк [5]. Так формируется список меток, используемый для генерации объяснения УК. В системе C-lightVer реализован алгоритм извлечения меток из УК [14], отличающийся от подхода Денни и Фишера. Этот алгоритм основан на представлении УК в виде дерева, которое обходится в глубину. Метки добавляются в список, используемый для генерации объяснения УК, в порядке такого обхода.

Таким образом, результатом работы алгоритма извлечения меток из УК является список меток, используемый для генерации объяснения УК. После извлечения меток из УК происходит генерация объяснения УК. Алгоритм генерации объяснения УК основан на последовательном обходе полученного списка меток. Для каждой посещенной при обходе метки текст ее заполненного номерами строк шаблона добавляется к тексту, объясняющему УК.

Символический метод верификации финитных итераций. Рассмотрим инструкцию вида $for\ x\ in\ S\ do\ v := body(v, x)\ end$, где S – структура 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 $x \in S$. Как описано ниже структура S может быть изменена. Тело цикла может содержать только инструкции присваивания, инструкции if (возможно вложенные) и инструкции $break$. Такие инструкции называются допустимыми конструкциями тела цикла [15]. Такие циклы for называются финитными итерациями. 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: $\forall j (i \le j \le n)\ rep(i, v, S, body) = rep(j, v, S, body)$.

Рассмотрим правило вывода, предложенное для финитной итерации и расширенное семантическими метками, на языке описания шаблонов:

```
{P} prog {(vector_substitution(Q, v,
                  (label rep_iter rep(n, v, S, body).vector_element))}
|- {any_predicate(P)} any_code(prog)
for(int_var(i) = 0; int_var(i) < int_var(n); int_var(i)++)
admissible_construct(i, n, v, S, body)
{any_predicate(Q)}
```

где конструкция $admissible\_construct(i, n, v, S, body)$ соответствует допустимой конструкции тела цикла, int_var соответствует целочисленной переменной. Конструкция $vector\_substitution(Q,\ v,\ rep(n,\ v,\ S,\ body).vector\_element)$ обозначает одновременную заменую для каждого $t (1 \le t \le length(v))$ всех вхождений v_t в формуле Q на $rep(n,\ v,\ S,\ body).v_t$. Рекурсивное определение допустимой конструкции описано в [15]. Алгоритмы сопоставления этих шаблонов и программных конструкций были реализованы в системе C-lightVer [14–16]. Правило вывода для обратной итерации определено подобным образом.

Автоматическая генерация операции замены. Генерация операции замены основана на трансляции [15] допустимых конструкций тела цикла в конструкции ACL2 [25]. Рассмотрим конструкцию $(b* \ (\ldots (var\ expr) \ldots)\ result)$.

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 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 fr. 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.

2. Стратегия доказательства формул с операцией замены

Для описания автоматизации локализации ошибок в системе C-lightVer рассмотрим разработанную нами ранее и приведенную в предыдущей работе [17] стратегию доказательства формул с операцией замены.

Входными аргументами данной стратегии являются импликация ψ, содержащая применение функции $rep(n, \ldots)$, и ее посылка ϕ. Попытаемся доказать формулу

$$\phi \to rep(n, \ldots).loop\text{-}break \qquad (\psi\text{-}lemma\text{-}1)$$

индукцией по n. Если система ACL2 доказывает (ψ-*lemma-1*), то добавляем ее в теорию предметной области. Эта лемма означает, что посылка ϕ импликации ψ описывает один из случаев, когда при исполнении цикла происходит исполнение *break*. Попытаемся доказать ψ, используя (ψ-*lemma-1*) и индукцию по n.

Если система ACL2 не доказала (ψ-*lemma-1*), то попытаемся доказать формулу

$$\phi \to \neg rep(n, \ldots).loop\text{-}break \qquad (\psi\text{-}lemma\text{-}2)$$

индукцией по n. Если система ACL2 доказывает (ψ-*lemma-2*), то добавляем ее в теорию предметной области. Эта лемма означает, что посылка ϕ импликации ψ описывает один из случаев, когда при исполнении цикла не происходит исполнение *break*. Попытаемся доказать ψ, используя (ψ-*lemma-2*) и индукцию по n.

Отметим, что стратегия доказательства формул с операцией замены похожа на описанную в предыдущей работе [15] стратегию доказательства УК для программ с инструкцией выхода из цикла. Во-первых, обе стратегии основаны на использовании значения поля *loop-break*. Во-вторых, обе стратегии полностью автоматизированы, так как при применении данных стратегий от пользователя системы верификации не требуется предоставлять дополнительные данные. В-третьих, обе стратегии являются эвристическими методами доказательства, т.е. применение данных стратегий к истинной формуле может не привести к успешному доказательству.

Поэтому опишем различия между стратегиями. Во-первых, стратегия доказательства формул с операцией замены принимает в качестве аргумента любую импликацию, содержащую *rep*. Стратегия доказательства УК для программ с инструкцией выхода из цикла [15] анализирует только постусловие программы. Во-вторых, стратегия доказательства УК для программ с инструкцией выхода из цикла [15] генерирует лемму в виде конъюнкции. Первый элемент такой конъюнкции – это УК, второй – равенство посылки импликации из постусловия и значения поля *loop-break*. Такая структура генерируемой леммы обусловлена тем, что стратегия доказательства УК для программ с инструкцией выхода из цикла [15] применяется только тогда, когда постусловие программы представляет собой конъюнкцию импликаций, т.е. представляет собой разбор случаев, где каждый случай описывается посылкой импликации. Поэтому стратегия доказательства УК для программ с инструкцией выхода из цикла [15] генерирует лемму более сложной структуры, чем стратегия доказательства формул с операцией замены.

3. Стратегия проверки ложности недоказанного УК

Для описания автоматизации локализации ошибок в системе C-lightVer рассмотрим разработанную нами ранее и описанную в предыдущей работе [17] стратегию доказательства ложности недоказанного УК.

Входным аргументом данной стратегии является УК (формула ω), содержащая применение функции $rep(n, \ldots)$ и имеющая следующую структуру:

$$\forall x_1 \ldots x_n \ (\phi_1(x_1 \ldots x_n) \to \psi_1(x_1 \ldots x_n)) \wedge \ldots \wedge (\phi_m(x_1 \ldots x_n) \to \psi_m(x_1 \ldots x_n)).$$

Все переменные формулы находятся под квантором всеобщности из-за ориентированности данной стратегии на систему ACL2. Отметим, что в определении любой

из формул ϕ_i или ψ_i могут использоваться не все переменные из набора $x_1 \dots x_n$. Но мы обозначили в качестве параметров весь набор переменных, так как в определении любой формулы можно ввести использование любой новой переменной, записав формулу в конъюнкции с этой переменной. Такое преобразование возможно, так как весь набор переменных находится под квантором всеобщности. Поэтому без ограничения общности можно обозначать зависимость от всего набора переменных. Известно, что формула ложна тогда и только тогда, когда ее отрицание истинно. Поэтому докажем формулу $\neg\omega$:

$$(\exists x_1 \dots x_n\ (\phi_1(x_1 \dots x_n) \wedge \neg\psi_1(x_1 \dots x_n))) \vee \dots \vee (\exists x_1 \dots x_n\ (\phi_m(x_1 \dots x_n) \wedge \neg\psi_m(x_1 \dots x_n))).$$

Для каждого $i(1 \le i \le m)$ подформулу $\exists x_1 \dots x_n\ (\phi_i(x_1 \dots x_n) \wedge \neg\psi_i(x_1 \dots x_n))$ обозначим как T_i. Для доказательства формулы $\neg\omega$ достаточно доказать любую формулу $T_i(1 \le i \le m)$.

Поэтому наша стратегия проверки ложности УК состоит в том, чтобы применять специальную процедуру доказательства к формулам T_i. Если удалось доказать любую формулу T_i, то результатом стратегии будет сообщение о ложности УК `Verification condition is false`. Если не удалось доказать ни одну формулу T_i, то результатом стратегии будет сообщение о неизвестном результате `Unknown`.

Опишем специальную процедуру доказательства. Рассмотрим применение данной процедуры к формуле $T_i(1 \le i \le m)$. Для доказательства формулы $\exists x_1 \dots x_n$ $(\phi_i(x_1 \dots x_n) \wedge \neg\psi_i(x_1 \dots x_n))$ достаточно доказать формулу $(\exists x_1 \dots x_n\ \phi_i(x_1 \dots x_n))$ $\wedge(\forall x_1 \dots x_n(\phi_i(x_1 \dots x_n) \to \neg\psi_i(x_1 \dots x_n)))$ (формула T_i'). Обозначим как U_i формулу $\exists x_1 \dots x_n\ \phi_i(x_1 \dots x_n)$. Обозначим как V_i $\forall x_1 \dots x_n(\phi_i(x_1 \dots x_n) \to \neg\psi_i(x_1 \dots x_n))$. Специальная процедура доказательства сводится к проверке истинности U_i и к доказательству V_i.

Как мы описали ранее в предыдущей работе [17], истинность формулы U_i проверяется пользователем. Таким образом, наша стратегия проверки ложности УК является не автоматической, а автоматизированной. Мы отказываемся от автоматизации процесса доказательства U_i по следующим причинам. Во-первых, для генерации УК мы вычисляем слабейшее предусловие [15]. Поэтому формула U_i не может содержать операцию замены. Значит, формула U_i может содержать только известные пользователю функции из спецификации. Во-вторых, практически все известные нам системы доказательства имеют проблемы при автоматическом доказательстве формулы с квантором существования. В-третьих, наша эвристика основана на гипотезе о простой посылке и сложном заключении импликаций, сгенерированных в результате вычисления слабейшего предусловия. По нашей гипотезе мы автоматизируем более сложную задачу доказательства V_i. Если пользователь не сообщил об истинности U_i, то мы полагаем, что формулу T_i не удалось доказать.

Если пользователь сообщил об истинности U_i, то мы пытаемся доказать V_i [17]. Сначала рассмотрим случай, когда формула V_i не содержит применения функции *rep*. Так как система ACL2 ориентирована на доказательство формул, где все переменные находятся под квантором всеобщности, то попытаемся автоматически доказать формулу V_i в системе ACL2. Теперь рассмотрим случай, когда формула V_i содержит применение функции $rep(n, \dots)$. Так как пользователь не знает кода

функции rep, то в этом случае возможно только автоматическое доказательство. Также проблемой является возможный выход из цикла, усложняющий код функции rep. Поэтому попытаемся доказать V_i, используя стратегию автоматического доказательства содержащей операцию замены формулы из раздела 2.

4. Дополнение семантическими метками определения операции замены

В данном разделе представим наши новые результаты, которыми являются модифицированный алгоритм генерации операции замены, модифицированный алгоритм извлечения семантических меток и модифицированный алгоритм генерации объяснений недоказанных условий корректности.

Описанное в разделе 1. расширение метода семантической разметки позволило пользователю системы верификации задавать свои концепции семантических меток и снабжать ими правила вывода [14]. Но возникла необходимость еще в одном расширении метода семантической разметки.

Рассмотрим случай, когда УК содержит применение операции замены. Рассмотренное правило вывода для операции замены помечает применение функции rep в УК специальной семантической меткой. Но такая метка не может предоставить информацию о структуре финитной итерации, которую она помечает. Объяснение УК в таком случае будет содержать текст о том, что применение функции rep выражает действие цикла. Но такой текст не содержит информацию о структуре цикла.

Денни и Фишер предложили способ снабжать семантическими метками формулы, но они не предложили способа снабжать семантическими метками определения применяемых в формулах функций. В случае применения операции замены возникла необходимость исправить этот недостаток метода семантической разметки. Поэтому нами было предложено решение расширить метод семантической разметки, снабжая семантическими метками определение операции замены. Для реализации этого подхода нами был предложен способ генерации снабженного метками определения функции rep, способ извлечения меток из определения функции rep и способ генерации текста для списка извлеченных меток. Рассмотрим эти способы.

Модифицированный алгоритм генерации операции замены. Для дополнения метками определения функции нужно найти такой способ, который не нарушает семантику языка задания этого определения, и который позволяет упростить извлечение меток из этого определения. Поэтому предложенный нами способ генерации снабженного метками определения функции rep основан на возможностях, предоставляемых языком системы ACL2.

Наш способ представляет собой модификацию алгоритма автоматической генерации операции замены [15] из раздела 1. Тот алгоритм основан на моделировании последовательности конструкций цикла с помощью последовательности связываний переменной fr с новыми значениями. Но конструкция $b*$ позволяет задать не только последовательное связывание переменных со значением, но и одновременное связы-

вание переменных со значениями. Для одновременного связывания двух переменных с двумя значениями удобно использовать конструкцию *cons*.

Введем переменную *label*, соответствующую семантической метке. Каждое связывание переменной *label* со значением соответствует своей конструкции цикла и происходит одновременно со связыванием переменной fr с новым значением. Когда мы генерируем связывание, моделирующее очередную конструкцию цикла, мы связываем с новым значением не только переменную fr, но и переменную *label*.

Рассмотрим, что представляет собой значение, связываемое с переменной *label*. Оно представляет собой список следующего вида

$$(list\ 'concept\ begin\ end\ 'break\_path),$$

где

- concept – одна из концепций меток. Для помечающих код меток заранее задан определенный набор концепций (типов) меток. Рассмотрим эти концепции:
  - $empty\_stmt$ – концепция метки, соответствующей пустому оператору;
  - $break\_stmt$ – концепция метки, соответствующей инструкции `break`;
  - $assign\_stmt$ – концепция метки, соответствующей присваиванию;
  - if_stmt – концепция метки, соответствующей сокращенному `if`;
  - $full\_if\_stmt$ – концепция метки, соответствующей инструкции `if`;
  - $block\_stmt$ – концепция метки, соответствующей блоку.

  Отметим, что концепции меток однозначно соответствуют видам допустимых конструкций тела финитной итерации;
- begin – начало диапазона строк кода соответствующей метке конструкции;
- end – конец диапазона строк кода соответствующей метке конструкции;
- $'break\_path$ – опциональный элемент списка, который присутствует в списке тогда и только тогда, когда соответствующая метке конструкция лежит на пути к одному из операторов выхода из цикла.

Такая конструкция представляет собой запись семантической метки.

Также нами был предложен способ помечать специальной семантической меткой условие инструкции `if`. Для этого выражение e, соответствующее такому условию, записывается как результат конструкции $b*$, но переменная fr связывается только своим значением, так как при вычислении условия инструкции `if` значения переменных не изменяются. В такой конструкции происходит связывание только переменной *label*, а значением этой конструкции является значение e. Переменная *label* связывается в такой конструкции со списком, реализующим семантическую метку. Такой список соответствует введенному ранее формату, но в качестве задающего концепцию первого элемента списка используется if_cond.

Модифицированный алгоритм генерации операции замены – это алгоритм из предыдущей работы [15], в котором использование функции $generate\_rep$ заменено

на использование функции *gen_rep*. Эти функции принимают в качестве аргумента допустимую конструкцию и транслируют ее на язык системы ACL2. Функция *c_kernel_translator* [15] осуществляет трансляцию C-kernel выражений на язык системы ACL2.

Определим функцию *gen_rep* для каждого вида допустимой конструкции *op*, обозначив как ... поставляемые номера строк и опциональный элемент *'break_path*:

- если *op* – это пустой оператор, то результатом *gen_rep*(*op*) будет

$$((cons\ ?!label\ fr)\ (cons\ (list\ 'empty\_stmt\ \ldots)\ fr));$$

- если *op* — это **break**;, то результатом *gen_rep*(*op*) будет

$$\begin{gathered}((cons\ \ ?!label\ \ fr)\ \ (cons\\ (list\ \ 'break\_stmt\ \ \ldots)\\ (change\text{-}frame\ \ fr\ \ :loop\text{-}break\ \ t)))\\ ((when\ \ t)\ \ fr);\end{gathered}$$

- если *op* — это **if** (**a**) **b**, то результатом *gen_rep*(*op*) будет

$$\begin{gathered}((cons\ \ ?!label\ \ fr)\ \ (cons\ \ (list\ \ 'if\_stmt\ \ \ldots)\\ (if\ \ (b*\ \ ((cons\ \ ?!label\ \ ?!fr)\ \ (cons\\ (list\ \ 'if\_cond\ \ \ldots)\ \ fr))\ \ c\_kernel\_translator(a))\\ (b*\ (gen\_rep(b))\ fr))))\\ ((when\ \ (frame\text{->}loop\text{-}break\ \ fr))\ \ fr);\end{gathered}$$

- если *op* — это **if** (**a**) **b else c**, то результатом *gen_rep*(*op*) будет

$$\begin{gathered}((cons\ \ ?!label\ \ fr)\ \ (cons\ \ (list\ \ 'full\_if\_stmt\ \ \ldots)\\ (if\ \ (b*\ \ ((cons\ \ ?!label\ \ ?!fr)\ \ (cons\\ (list\ \ 'if\_cond\ \ \ldots)\ \ fr)\ c\_kernel\_translator(a))\\ (b*(gen\_rep(b))\ fr))\ (b*(gen\_rep(c))\ fr))))\\ ((when\ \ (frame\text{->}loop\text{-}break\ \ fr))\ \ fr);\end{gathered}$$

- если *op* — это {$\mathbf{a_1}$ $\mathbf{a_2}$ ... $\mathbf{a_{k-1}}$ $\mathbf{a_k}$}, то результатом *gen_rep*(*op*) будет

$$\begin{gathered}((cons\ \ ?!label\ \ fr)\ \ (cons\ (list\ \ 'block\_stmt\ \ \ldots)\\ (b*(gen\_rep(a\_1)\ gen\_rep(a\_2)\ \ldots gen\_rep(a\_k\text{-}1)\ \ gen\_rep(a\_k))\ fr)))\\ ((when\ \ (frame\text{->}loop\text{-}break\ \ fr))\ \ fr).\end{gathered}$$

Модифицированный алгоритм генерации операции замены отличается от исходного тем, что он снабжает все виды допустимых конструкций семантическими метками.

Модифицированный алгоритм извлечения семантических меток. Модифицированный алгоритм извлечения меток – это алгоритм из раздела 1., дополненный способом извлечения семантических меток из специального представления функции *rep* в виде дерева кода. Опишем способ извлечения семантических меток из специального представления функции *rep* в виде дерева кода.

Представим определение функции *rep* в виде дерева. Построим его по процедуре построения дерева кода функции. Примененная к связываниям, которые соответствуют пустым операторам, инструкциям `break` и присваиваниям тела цикла, данная процедура возвращает вырожденные деревья в виде соответствующих листьев. Примененная к связыванию, которое соответствует инструкции `if`, данная процедура возвращает дерево, корнем которого является это связывание, а потомки (поддеревья) получены в результате применения данной процедуры к блокам, соответствующим ветвям этой инструкции `if`. Примененная к блоку $b*$, которая соответствует C-блоку, данная процедура возвращает дерево, корнем которого является этот блок, а потомки (поддеревья) получены в результате применения этой процедуры к последовательности связываний, составляющих этот блок. Таким образом, процедура построения дерева кода функции определена рекурсивно.

Для извлечения меток из определения функции *rep* для него строится дерево с помощью применения процедуры построения дерева кода функции к блоку $b*$, соответствующему телу цикла. Затем это дерево обходится в глубину, и метки извлекаются из вершин дерева в порядке обхода. Таким образом представление в виде дерева определения функции *rep* используется для реализации обхода в глубину.

Модифицированный алгоритм генерации объяснений недоказанных условий корректности. Модифицированный алгоритм генерации объяснений недоказанных условий корректности – это алгоритм из раздела 1., дополненный способом генерации текста для меток, извлеченных из специального представления функции *rep* в виде дерева кода. Опишем способ генерации текста для меток, извлеченных из определения функции *rep*.

Каждой концепции меток, помечающих определение функции *rep*, соответствует текстовый шаблон. Такие шаблоны похожи на форматные строки языка C со специальными управляющими конструкциями для вставки номера первой строки диапазона строк кода и номера последней строки диапазона строк кода.

При генерации текста список извлеченных меток обрабатывается последовательно. Для каждой метки происходит извлечение диапазона строк, затем с помощью обратного транслятора [24] данный диапазон строк C-kernel программы переводится в диапазон строк C-light программы. Потом для данной метки генерируется текст с помощью подстановки полученных номеров строк C-light программы. Таким образом, общий текст получается из набора текстов для каждой метки.

Если в верифицируемой программе содержится финитная итерация с инструкцией `break`, то нами был предложен способ генерации дополнительного поясняющего текста. УК такой программы представляет собой импликацию, содержащую применение функции *rep*. Рассмотрим посылку такого УК. Проанализируем случай, описываемый рассматриваемой посылкой, используя результат применения стратегии для формул с операцией замены из раздела 2.

Если удалось доказать лемму, что в описываемом рассматриваемой посылкой

случае исполнилась инструкция `break`, то в данном случае рассмотрим связывания, соответствующие инструкциям `if` на пути к данной инструкции. Именно для этой цели список, соответствующий метке, содержит опциональный элемент $'break\_path$. Используя все условия условных операторов, вычислим символически с помощью подстановок общее условие, при выполнении которого исполняется связывание, соответствующее инструкции `break`. Обозначим такое условие как θ. Так как в рассматриваемом случае инструкция `break` исполнилась, то хотя бы для одной итерации условие θ истинно. Значит, пользователю системы верификации имеет смысл проверить условие θ на предмет наличия ошибки. Поэтому об этом генерируется дополнительный поясняющий текст. Этот текст содержит: 1)информацию о исполнении инструкции *break*, заданной на определенной строке программы, 2) информацию о существовании такого номера итерации, что условие θ выполняется на этой итерации, и 3)условие θ.

5. Пример

В данном разделе описываются результаты эксперимента с примером из работы [17], чтобы проиллюстрировать применение разработанных и добавленных в систему C-lightVer инструментов для автоматизированной локализации ошибок. Эксперимент состоит в применении системы C-lightVer к содержащей ошибку функции `grt_eql_key`:

```
1. /*@ requires (0 < n) && (n <= len(a));
2.     ensures (grt-eql-cnt(n, key, a) == 0 ==> \result == 0) &&
3.             (grt-eql-cnt(n, key, a) > 0 ==> \result == 1)
4. */
5. int grt_eql_key(int n, int key, int a[]){
6.    int i, result = 0;
7.    for (i = 0 ; i < n; i++){
8.        if (a[i] < key){result = 1; break;}}
9.    return result;}
```

Спецификации функции заданы на языке ACSL. Определение функции *grt-eql-cnt* на языке ACL2 приведено в репозитории [13]. В соответствии со спецификацией функция `grt_eql_key` проверяет существование в массиве `a` элемента, который больше ключа `key` или равен ключу `key`. Если такой элемент найден, то по спецификации функция должна возвращать `1`, иначе – `0`. Таким образом, ошибка состоит в использовании оператора `<` вместо оператора `>=` в инструкции `if` тела цикла.

Полученная C-kernel программа, ее УК, описание применения стратегий и вспомогательные леммы приведены в статье [17]. Заданные на языке системы ACL2 УК, определение *rep* и вспомогательные леммы приведены в репозитории [13].

В ходе проведенного ранее эксперимента с этим примером [17] применение стратегии проверки ложности из раздела 3. позволило доказать ложность УК, используя его первый конъюнкт. Поэтому результатом проведенного ранее эксперимента является следующее объяснение [17], сгенерированное для этого конъюнкта:

```
This formula corresponds to lines 1-9 in function "grt_eql_key".
Its purpose is to show unsatisfiable case. Hence, given
    - assumption that precondition from line 1 holds,
    - assumption that postcondition hypothesis from line 2 holds,
ensure that postcondition goal from line 2
    with substitution loop effect from lines 7-8 by rep
does not hold.
```

Отметим, что системой C-lighVer генерируются именно такие объяснения. Единственное отличие объяснений, генерируемых системой C-lightVer, от объяснений, приведенных в данном разделе, состоит в наличие отступов, которые для улучшения читабельности текста добавляются вручную.

В отличие от предыдущей работы [17] для проведения эксперимента мы использовали модифицированный алгоритм генерации операции замены, модифицированный алгоритм извлечения меток и модифицированный алгоритм генерации объяснений недоказанных условий корректности. Поэтому в результате проведения эксперимента было сгенерировано более подробное объяснение:

```
This formula corresponds to lines 1-9 in function "grt_eql_key".
Its purpose is to show unsatisfiable case. Hence, given
    - assumption that precondition from line 1 holds,
    - assumption that postcondition hypothesis from line 2 holds,
ensure that
    - postcondition goal from line 2
      with substitution loop effect from lines 7-8 by rep
        that corresponds to the following loop body
        sequence of the following statements
        from line 8 to line 8
          if statement from line 8
          with condition "a[i] < key" from line 8
          and with the following positive branch
          sequence of the following statements
          from line 8 to line 8
            assignment statement "result = 1" from line 8
            break statement from line 8
      does not hold
    - break statement execution in the loop body from line 8
      holds
    - exists such i (0 <= i < n)
      that condition
      a[i] < key
      holds
```

В отличие от объяснения, полученного в результате проведенного ранее эксперимента [17], данное объяснение содержит текст, соответствующий телу цикла,

и текст, описывающий содержащее ошибку условие. Данный текст позволяет обратить внимание пользователя системы верификации на содержащее ошибку условие инструкции `if`, в ветви которой находится `break`.

6. Заключение

Эта статья посвящена работе подсистемы локализации ошибок системы C-lightVer. Данная подсистема работает в том случае, если не удалось доказать истинность УК. В таком случае подсистема локализации ошибок пытается проверить ложность УК. Если недоказанное УК основано на функции *rep*, то подсистема объяснения УК должна давать подробные объяснения о структуре цикла. Если посылка ложного УК описывает случай исполнения инструкции `break`, то подсистема объяснений должна обратить внимание пользователя на условие исполнения `break`.

В данной статье описаны разработанные авторами модификации алгоритмов:

1. Модифицированный алгоритм генерации операции замены. Новая модификация алгоритма позволяет пометить исходный код функции *rep* семантическими метками.

2. Модифицированный алгоритм извлечения семантических меток. Новая модификация алгоритма позволяет извлекать семантические метки не только из УК, но и из специального представления функции *rep* в виде дерева кода.

3. Модифицированный алгоритм генерации объяснений недоказанных условий корректности. Если посылка ложного УК описывает случай исполнения инструкции `break`, то новая модификация алгоритма позволяет подсистеме локализации ошибок обратить внимание на условие исполнения `break`.

Данные методы позволили нам провести эксперимент по локализации ошибки в функции `grt_eql_key`. Результат данного эксперимента приведен в этой статье.

Ранее мы провели успешные эксперименты [14] по дедуктивной верификации функции `asum` [15] из интерфейса BLAS. Так как функции интерфейса BLAS совершают итерации над векторами и матрицами, то планы нашей дальнейшей работы состоят в применении к данным функциям комплексного подхода системы C-lightVer, основанного на символическом методе верификации финитных итераций.

Список литературы / References

[1] De Angelis E., Fioravanti F., Pettorossi A., Proietti M., “Lemma Generation for Horn Clause Satisfiability: A Preliminary Study”, VPT 2019, EPTCS, **299**, 2019, 4–18.

[2] Apt K. R., Olderog E.-R., “Fifty years of Hoare’s logic”, *Formal Aspects of Computing*, **31**:6 (2019), 751–807.

[3] Blanchard A., Loulergue F., Kosmatov N., “Towards Full Proof Automation in Frama-C Using Auto-active Verification”, NFM 2019, LNCS, **11460**, 2019, 88–105.

[4] De Carvalho D. et al., “Teaching Programming and Design-by-Contract”, ICL 2018, AISC, **916**, 2019, 68–76.

[5] Denney E., Fischer B., "Explaining Verification Conditions", AMAST 2008, LNCS, **5140**, 2008, 145–159.

[6] Efremov D., Mandrykin M., Khoroshilov A., "Deductive Verification of Unmodified Linux Kernel Library Functions", ISoLA 2018, LNCS, **11245**, 2018, 216–234.

[7] Fraer R., "Tracing the Origins of Verification Conditions", AMAST 1996, LNCS, **1101**, 1996, 241–255.

[8] 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*, **41**:10 (2015), 1019–1037.

[9] Hähnle R., Huisman M., "Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools", *Computing and Software Science*, LNCS, **10000**, 2019, 345–373.

[10] Heras J., Komendantskaya E., Johansson M., Maclean E., "Proof-Pattern Recognition and Lemma Discovery in ACL2", LPAR 2013, LNCS, **8312**, 2013, 389–406.

[11] Johansson M., "Lemma Discovery for Induction", CICM 2019, LNCS, **11617**, 2019, 125–139.

[12] Khazeev M., Mazzara M., De Carvalho D., Aslam H., "Towards A Broader Acceptance of Formal Verification Tools: The Role of Education", 2019, arXiv:abs/1906.01430.

[13] Kondratyev D. A., "Automated Error Localization in C Programs.", bitbucket.org/Kondratyev/verify-c-light.

[14] Kondratyev D., "Implementing the Symbolic Method of Verification in the C-Light Project", PSI 2017, LNCS, **10742**, 2018, 227–240.

[15] 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; [Kondratyev D. A., Maryasov I. V., Nepomniaschy 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, (in Russian).]

[16] Kondratyev D. A., Promsky A. V., "Developing a Self-Applicable Verification System. Theory and Practice", *Automatic Control and Computer Sciences*, **49**:7 (2015), 445–452.

[17] Kondratyev D. A., Promsky A. V., "Towards Automated Error Localization in C Programs with Loops", *System Informatics*, 2019, № 14, 31–44.

[18] Kondratyev D., Promsky A., "Proof Strategy for Automated Sisal Program Verification", TOOLS 2019, LNCS, **11771**, 2019, 113-120.

[19] Könighofer R., Toegl R., Bloem R., "Automatic Error Localization for Software Using Deductive Verification", HVC 2014, LNCS, **8855**, 2014, 92–98.

[20] Kovács L., "Symbolic Computation and Automated Reasoning for Program Analysis", IFM 2016, LNCS, **9681**, 2016, 20–27.

[21] Leino K. R. M., Millstein T., Saxe J. B., "Generating Error Traces from Verification-Condition Counterexamples", *Science of Computer Programming*, **55**:1–3 (2005), 209–226.

[22] Li J., Sun J., Li L., Loc Le Q., Lin S-W., "Automatic Loop Invariant Generation and Refinement through Selective Sampling", ASE 2017, 2017, 782–792.

[23] Lin Y., Bundy A., Grov G., Maclean E., "Automating Event-B invariant proofs by rippling and proof patching", *Formal Aspects of Computing*, **31**:1 (2019), 95–129.

[24] 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.

[25] Moore J. S., "Milestones from the Pure Lisp Theorem Prover to ACL2", *Formal Aspects of Computing*, **31**:6 (2019), 699–732.

[26] Moriconi M., Schwarts R. L., "Automatic Construction of Verification Condition Generators From Hoare Logics", ICALP 1981, LNCS, **115**, 1981, 363–377.

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

[28] Reger G., Voronkov A., “Induction in Saturation-Based Proof Search”, CADE 2019, LNCS, **11716**, 2019, 477–494.

[29] Srivastava S., Gulwani S., Foster J. S., “Template-Based Program Verification and Program Synthesis”, *International Journal on Software Tools for Technology Transfer*, **15**:5–6 (2013), 497–518.

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

[31] Volkov G., Mandrykin M., Efremov D., “Lemma Functions for Frama-C: C Programs as Proofs”, 2018 Ivannikov Ispras Open Conference (ISPRAS), 2018, 31–38.

[32] Yang W., Fedyukovich G., Gupta A., “Lemma Synthesis for Automating Induction over Algebraic Data Types”, CP 2019, LNCS, **11802**, 2019, 600-617.

Kondratyev D. A., Promsky A. V., "The Complex Approach of the C-lightVer System to the Automated Error Localization in C-programs", *Modeling and Analysis of Information Systems*, **26**:4 (2019), 502–519.

DOI: 10.18255/1818-1015-2019-4-502-519

Abstract. The C-lightVer system for the deductive verification of C programs is being developed at the IIS SB RAS. Based on the two-level architecture of the system, the C-light input language is translated into the intermediate C-kernel language. The meta generator of the correctness conditions receives the C-kernel program and Hoare logic for the C-kernel as input. To solve the well-known problem of determining loop invariants, the definite iteration approach was chosen. The body of the definite iteration loop is executed once for each element of the finite dimensional data structure, and the inference rule for them uses the substitution operation rep, which represents the action of the cycle in symbolic form. Also, in our meta generator, the method of semantic markup of correctness conditions has been implemented and expanded. It allows to generate explanations for unproven conditions and simplifies the errors localization. Finally, if the theorem prover fails to determine the truth of the condition, we can focus on proving its falsity. Thus a method of proving the falsity of the correctness conditions in the ACL2 system was developed. The need for more detailed explanations of the correctness conditions containing the replacement operation rep has led to a change of the algorithms for generating the replacement operation, and the generation of explanations for unproven correctness conditions. Modifications of these algorithms are presented in the article. They allow marking rep definition with semantic labels, extracting semantic labels from rep definition and generating description of break execution condition.

Keywords: deductive verification, semantic label, error localization, C-lightVer, ACL2, MetaVCG, definite iteration, proof strategy

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

Alexei V. Promsky, orcid.org/0000-0002-5963-2390, PhD,
A. P. Ershov Institute of Informatics Systems SB RAS,
6 Akademik Lavrentyev av., Novosibirsk 630090, Russia, e-mail: promsky@iis.nsk.su

Acknowledgments:
This work was partially funded by the RFBR according to the research № 17-01-00789.