# Расширение системы C-light символическим методом верификации финитных итераций

Д. А. Кондратьев

Институт систем информатики им. А.П. Ершова СО РАН, Новосибирск, Россия

Контактный e-mail: apple-66@mail.ru

При разработке в ИСИ СО РАН дедуктивной системы C-light встала задача создания специализированных версий генераторов условий корректности (УК). Для ее решения использована концепция метагенерации УК.

Концепция семантических меток позволяет применять саму процедуру генерации УК для построения их объяснений. Система C-light простым образом дополняется таким расширением с помощью метода метагенерации.

Проблема элиминации инвариантов циклов является алгоритмически разрешимой для циклов специального вида с помощью символического метода верификации финитных итераций. Для таких циклов применяются специальные правила вывода, позволяющие отказаться от использования инварианта цикла. Метод метагенерации позволил удобным образом расширить систему C-light такими правилами.

*Ключевые слова*: верификация, метод метагенерации, символический метод верификации финитных итераций, условие корректности, метод семантической разметки.

## Введение

Верификация программ в дедуктивных системах традиционно основана на генерации условий корректности (УК). Она заключается в формальном доказательстве корректности программ в соответствии с описанием их свойств, задаваемых в виде спецификаций программ. В лаборатории теоретического программирования ИСИ СО РАН развивается проект по верификации программ на языке C-light [1], который является представительным подмножеством языка Си. Но поскольку аксиоматический поход для такого языка, как C-light, может быть слишком громоздким, предложен двухуровневый метод верификации программ. На первом этапе C-light транслируется в промежуточный язык C-kernel [1] с целью элиминации некоторых конструкций, сложных для аксиоматической семантики. Для трансляции используется набор формальных правил. На втором этапе для промежуточной C-kernel-программы генерируются УК, которые в дальнейшем передаются на блок доказательства. Генератор условий корректности (ГУК) создается по аксиоматической семантике языка C-kernel. Сама аксиоматическая семантика является набором аксиом и правил вывода для языковых конструкций.

При разработке системы C-light встала задача создания расширенных или специализированных версий ГУК. Для решения данной задачи использована концепция метагенерации УК [2]. Получая на вход логику Хоара, метагенератор автоматически порождает рекурсивный алгоритм генерации УК. Ограничения нормальной формы на правила вывода, подаваемые на вход метагенератора, гарантируют непротиворечивость и полноту порождаемого генератора как системы вывода относительно исходной аксиоматической семантики.

Правила вывода, поступающие на вход метагенератора, задают шаблоны для сопоставления с программными конструкциями. В результате проделанной работы разработан язык шаблонов, который позволяет записывать аксиомы и правила логики Хоара в привычном виде. Разработан алгоритм сопоставления деревьев, представляющих шаблоны вывода и программы. Воплощение этого алгоритма на языке C-light позволяет провести частичную самоверификацию системы. Аксиоматической семантики языка C-kernel в общем виде, поданной на вход системы C-light [1], достаточно для проведения верификации фрагментов метагенератора. Эксперименты по верификации фрагментов метагенератора были успешно проведены [3].

Проблема автоматической генерации инвариантов циклов является алгоритмически неразрешимой. Но для циклов специального вида она разрешима с помощью символического метода верификации финитных итераций [4], который заключается в том, что для таких циклов применяются специальные правила вывода, позволяющие отказаться от использования инварианта цикла. Они основываются на специальной функции замены, представляющей в символической форме действие цикла. Метод метагенерации позволил удобным образом расширить систему C-light такими правилами.

## 1. Метод метагенерации условий корректности

### 1.1. Основы дедуктивной верификации программ

В 1969 г. Ч. Хоар [5] ввел способ задания аксиоматической семантики, ставший основой метода дедуктивной верификации программ. Подход Хоара заключается в том, что текст программы представляется как особое отношение между утверждениями. Базовыми формулами в рассматриваемом подходе являются тройки Хоара $\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}$, где $\mathcal{P}$ — предусловие (логическая формула), $S$ — программа, $\mathcal{Q}$ — постусловие (логическая формула). Истинность формулы $\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}$ означает, что “если предусловие $\mathcal{P}$ истинно перед исполнением фрагмента программы $S$ и если исполнение $S$ завершилось, то постусловие $\mathcal{Q}$ выполняется после его завершения” [2]. Правила вывода задаются в виде

$$\frac{\psi_1, ..., \psi_n}{\varphi}, \tag{1}$$

где $\psi_1$,...,$\psi_n$ — посылки правила вывода (набор троек Хоара и логических формул) и $\varphi$ — заключение правила вывода (тройка Хоара). Данная нотация означает, что $\varphi$ выводимо при гипотезе $\psi_1$,...,$\psi_n$. Семантика простых операторов (например, присваивания) задается, как правило, с помощью набора аксиом, а любого сложного оператора (например, оператора последовательного исполнения) — с помощью правила вывода. Логическая система, содержащая аксиомы и правила вывода для всех синтаксических форм языка программирования, называется логикой Хоара или аксиоматической семантикой языка.

Генератор условий корректности осуществляет вывод в автоматическом режиме по аксиоматической семантике и сводит истинность тройки Хоара $\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}$ к корректности некоторого числа лемм, называемых условиями корректности, в предметной области. Доказуемости этих лемм достаточно для корректности исходной аннотированной программы.

Одним из методов такого вывода является метод слабейшего предусловия [2]. Для тройки Хоара $\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}$ слабейшее предусловие обозначается как $wp(S, Q)$. Слабейшее предусловие для тройки Хоара — это такое предусловие, которое обеспечивает истинность данной тройки Хоара, и оно выводимо как логическая формула из всех остальных предусловий, обеспечивающих истинность тройки Хоара. Для доказуемости тройки Хоара $\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}$ необходимо и достаточно [5], чтобы $P$ было эквивалентно $wp(S, Q)$. Таким образом, ГУК может быть реализован как программа, вычисляющая $wp(S, Q)$.

В качестве примеров правил вывода рассмотрим правило из классической работы [5]. Правило для цикла с предусловием имеет следующий вид:

$$\frac{\{P \wedge B\}\ S\ \{P\}, \quad P \wedge \neg B \supset Q}{\{P\}\ \texttt{while}\ (B)\ \texttt{assert}\ P\ \texttt{do}\ S\ \{Q\}}. \tag{2}$$

Чтобы вывести тройку Хоара для цикла `while`, необходимо использовать индукцию. Ввести индукцию позволяет специальная логическая формула, приписываемая циклу и называющаяся инвариантом цикла. Инвариант цикла — это утверждение, которое истинно перед исполнением цикла, истинно для каждой итерации цикла и обеспечивает корректность на выходе из цикла. Таким образом, посылками правила вывода для цикла с предусловием являются тройка Хоара, соответствующая итерации цикла без выхода из него, и логическая формула, соответствующая выходу из цикла.

### 1.2. Необходимость внесения изменений в ГУК

Когда верификацию изучают на примере простого модельного языка, в такой период обычно не требуется вносить изменения в ГУК. Но на практике при использовании дедуктивной верификации появляется необходимость вносить такие изменения. Что может вызывать необходимость их внесения? Во-первых, язык программирования может быть расширен новыми конструкциями [3]. Например, рассматриваются планы по дальнейшему расширению языка C-light. Большой интерес представляют языковые конструкции, появившиеся в новом стандарте языка Си, именуемом C11. Также C-light может быть расширен языковыми конструкциями родственных языков, таких как Objective C, C++. Во-вторых, представляет интерес создание узкоспециализированных версий ГУК, ориентированных на конкретные классы программ. Такая специализация позволяет упростить верификацию. Приведем пример для демонстрации такого упрощения.

Данный пример относится к программам вычислительной математики. Аксиоматическая семантика для этого класса программ была разработана в лаборатории теоретического программирования нашего института ранее [6]. Такие программы основаны на обработке массивов и матриц с использованием циклов. Способы задания аксиом и правил вывода для таких программ приведены в разд. 6, одна из аксиом является рассматриваемым примером. Она позволяет выразить действие двух вложенных циклов без использования инвариантов с помощью замены всех вхождений матрицы в предусловии на результат специальной операции замены *rep*. При условии, что `M` — это

двумерная матрица и $e(k,i)$ — выражение, зависящее от индексов $k$ и $i$, рассмотрим следующий шаблон:

$\{Q(M \leftarrow rep(M, mat(e_1, e_2, e_3, e_4), e(s,t)))\}$
```
   for(k = e1; k <= e2; k++)
      for(i = e3; i <= e4; i++)
         M[k][i] = e(k,i);
```
{Q}

Здесь матрица $rep(M, mat(e_1, e_2, e_3, e_4), e(s,t)))$ получается путем замены всех элементов из подматрицы $mat(e_1, e_2, e_3, e_4)$ на выражение $e$. Заметим, что проблема автоматической генерации инварианта цикла является алгоритмически неразрешимой. Но для цикла, рассматриваемого в нашем примере, задание двух инвариантов для двух вложенных циклов заменено использованием специальных логических конструкций ($rep$ и $mat$). Полная аксиоматизация этих конструкций приведена в [7].

Главное следствие из данного примера состоит в том, что при применении ГУК на практике важна его расширяемость. Трудно использовать большой ГУК, покрывающий все классы программ. Коллекция специализированных ГУК — лучшее решение данной проблемы. Таким образом, нас интересуют методы, позволяющие упростить разработку нового ГУК. Для решения данных задач подходит метод метагенерации УК [2].

Несмотря на то что известно много проектов по дедуктивной верификации программ, случаи применения метода метагенерации УК или разработки самоприменимой системы неизвестны. В своих проектах по дедуктивной верификации другие исследователи использовали отличающиеся друг от друга входные языки системы и языки реализации системы (примером может служить платформа WHY [8]) или концентрировались на верификации конкретных программ (примером может служить верификация гипервизора Hyper-V в проекте VCC [9]).

### 1.3. Нормальная форма правил вывода

Метод метагенерации УК был предложен Морикони и Шварцем в 1981 г. [2]. Метагенератор принимает на входе правила вывода аксиоматической семантики в нормальной форме и автоматически порождает ГУК, основанный на вычислении слабейшего предусловия. Подробное определение нормальной формы приведено в [2]. Морикони и Шварц определяют нормальную форму с помощью накладываемых на правило вывода пяти ограничений. Кратко опишем результат введения этих ограничений.

Для определения нормальной формы введена специальная терминология. В частности, метаидентификаторы, использующиеся в программных фрагментах троек Хоара, названы “фрагментными переменными”. Используя введенные обозначения, Морикони и Шварц переходят к определению ограничений.

Согласно первому ограничению, все предусловия посылок и постусловие заключения представляют собой предикатные символы, свободно входящие в правило вывода. Согласно второму ограничению, все свободные предикатные символы каждой формулы из посылки должны входить в объединение множества свободных предикатных символов всех предусловий и постусловий троек Хоара из правила вывода и множества “фрагментных переменных” программного фрагмента из заключения. Согласно третьему ограничению, все “фрагментные переменные” программных фрагментов из посылки

должны входить во множество “фрагментных переменных” программного фрагмента из заключения.

Четвертое ограничение обеспечивает то, что ГУК сможет вычислить вхождения всех свободных, неинтерпретированных предикатных символов в правилах. В частности, ограничение (4a) устанавливает следующий порядок свободных предикатных символов:

$$\frac{\begin{array}{c}\{P_1\}\ S_1\ \{\mathcal{Q}_1(P_2,...,P_n)\},...,\{P_i\}\ S_i\ \{\mathcal{Q}_i(P_{i+1},...,P_n)\},\\ ...,\ \{P_n\}\ S_n\ \{\mathcal{Q}_n\},\ \Gamma\end{array}}{\{\mathcal{P}(P_1,...,Q)\}\ S\ \{Q\}}.$$

В результате этого устраняются циклические зависимости, такие как посылки вида $\{P\}$ ... $\{P\}$ или пары посылок вида $\{P\}$ ... $\{R\}$ и $\{R\}$ ... $\{P\}$. При таком упорядочивании ограничение (4b) обеспечивает то, что конец каждой цепочки зависимостей (бинарного отношения между предикатными символами $A$ и $B$, указывающего на то, что связывание $A$ зависит от связывания $B$) либо представляет собой функцию от постусловия $Q$, либо связан с программным фрагментом.

Пятое ограничение необходимо для полноты ГУК, т. е. оно гарантирует, что ГУК сможет вычислить слабейшее предусловие $wp(S, Q)$ для данных $S$ и $Q$.

Необходимо отметить, что ограничения нормальной формы приводят для правил вывода к появлению порядка на посылках и нетипичному виду предусловий в тройках Хоара, если сравнить с аналогичными правилами из подразд. 1.1.

### 1.4. Общая форма правил вывода

Наложение ограничений нормальной формы позволяет доказать [2], что ГУК (как система вывода) является корректной и полной относительно исходной аксиоматической семантики в нормальной форме. С другой стороны, ограничения нормальной формы значительно сужают класс аксиоматических семантик, которые могут быть поданы на вход метагенератору.

Заметим, что аксиоматическая семантика языка C-kernel, который является промежуточным языком в нашем проекте, не удовлетворяет данным ограничениям. Более того, правила в нормальной форме имеют нетипичный вид, и поэтому Морикони и Шварц [2] ввели менее ограниченную общую форму правил вывода, а также алгоритм их перевода в нормальную форму. Для дальнейшего понимания материала приведем их краткое описание.

Общая форма сохраняет ограничения (1–3) и (4b) нормальной формы (вместе с модифицированным ограничением монотонности). При этом общая форма позволяет избежать нетипичного порядка на посылках. Согласно ограничениям общей формы, предусловия в посылках могут принимать более разнообразный вид и представлять собой не только одиночные предикатные символы, но и формулы предметной области, а также конъюнкцию данных вариантов.

Главная идея алгоритма перевода из общей формы в нормальную состоит в следующем: отделяют те предусловия, которые не представляют собой одиночные предикатные символы. Вместо них будут использоваться “очищенные” предикатные символы. Связь между этими новыми предикатными символами и исходными формулами устанавливается с помощью импликаций, в которых исходные формулы могут быть собраны в конъюнкцию (с удалением дубликатов при необходимости). На конечном шаге алгорит-

ма посылки получившегося правила должны быть переупорядочены для выполнения ограничения (4a).

Отметим, что аксиоматическая семантика языка C-kernel соответствует ограничениям общей формы [3]. По предложенному Морикони и Шварцем [2] алгоритму правила вывода для C-kernel были переведены в нормальную форму. По получившейся аксиоматической семантике может быть построен полный и корректный как система вывода ГУК. Таким образом, метод метагенерации УК можно применить в нашем проекте.

## 2. Метод семантической разметки

При практическом применении дедуктивной верификации могут возникнуть следующие проблемы: программа может быть некорректна, ее спецификации могут быть некорректны, автоматический доказатель теорем может не обладать достаточной мощностью, теория предметной области может быть неполна. В этих случаях пользователь системы верификации получает набор недоказанных условий корректности, но не получает дополнительной информации о причинах неудачи. Опишем предложенный Денни и Фишером метод семантической разметки [10], ориентированный на такую важную задачу, как анализ, трассировку и объяснение самих УК.

После упрощения УК обычно имеют вид хорновских дизъюнктов (т. е. $H_1 \wedge H_2 \wedge ... \wedge H_n \supset C$). В данном представлении единственное заключение $C$ можно рассматривать как цель. Однако для более осмысленного описания структуры необходимо дать более детальную характеристику подформул. Ключевая особенность подхода Денни и Фишера [10] состоит том, что различные подформулы располагаются на специальных позициях в правилах Хоара, и исходя из этого ГУК добавляет соответствующие метки к УК. Поэтому в модифицированные правила Хоара добавляется семантическая разметка, нужная для объяснения результата применения правила. Метки добавляются к следующим местам: к постусловию посылки, получившемуся при рекурсивном вызове ГУК, к слабейшему предусловию, сгенерированному УК или тройке Хоара [11].

Будем использовать для помеченных термов нотацию $\ulcorner t \urcorner^l$, означающую, что терму $t$ сопоставляется метка $l$ или список меток $l$. Метки имеют вид $c(o, n)$. Здесь $c$ — одна из концепций, описывающая предназначение данного терма и то, как обрабатывать такую метку. Позиция $o$ передает местоположение в программе связанных с данной меткой конструкций и представляет собой диапазон строк. Список меток $n$ содержит дополнительную уточняющую информацию для рассматриваемой метки. Сначала для данного вложенного терма список $n$ пуст, но после нормализации он содержит метки, “просочившиеся” из объемлющих термов.

## 3. Языки для задания шаблонов и семантических меток

Напомним, что правила в нормальной форме, поступающие на вход метагенератора, задают собой шаблоны для сопоставления с программными конструкциями [3]. Важной задачей является разработка для них некоторого языка. Очевидно, что в основе этого языка лежит язык логики первого порядка и целевой язык программирования. Вместе с тем в этот язык необходимо добавить некоторые метаобозначения, поскольку правила вывода задают семантику не конкретных программ, а схем программ. Заметим, что в соответствии с идеей метагенерации в общем случае разрабатывается схема языка шаблонов. Рассмотрим случай применения этой схемы для языка C-light.

### 3.1. Язык для задания шаблонов

Как было сказано выше, основой языка шаблонов для C-light является логика первого порядка и грамматика языка Си. Введение метаобозначений в этот язык фактически означает, что в выражениях этого языка сохраняются некоторые нетерминальные символы (например, неинтерпретированные предикатные символы и "фрагментные переменные" из подразд. 1.3 для обозначения фрагментов кода). Принадлежность метаданных тому или иному классу в языке шаблонов задается в явном виде [3]. В языке шаблонов не устанавливается жестких ограничений на то, что символы $P$, $Q$ и $R$ должны обозначать предикаты, а $S_i$ — программные фрагменты. Для обозначения данных конструкций в языке шаблонов могут быть использованы любые символы, а принадлежность к определенному классу конструкций в случае необходимости указывается явно. Например, конструкция `any_code(S)` может соответствовать любой последовательности (включая пустую) конструкций языка программирования, а `exists_code(S)` соответствует одиночной конструкции. Конструкция `simple_expression` обозначает выражение, которое не содержит вызовов функций и приведений типов.

Пусть `construction` — обозначение одной из специальных синтаксических конструкций и пусть `construction_identifier` обозначает уникальный идентификатор. Запись `construction(construction_identifier)` в шаблоне означает, что у данного вхождения `construction` вводится идентификатор `construction_identifier`. Программная конструкция, сопоставленная такому вхождению `construction`, будет обозначаться как `construction_identifier`. Введение такого обозначения необходимо для того, чтобы описать некоторые правила вывода. Например, такие, у которых программная конструкция, сопоставленная такому вхождению `construction`, используется в их посылках. Проиллюстрируем это находящимся в нормальной форме правилом для цикла `while`:

```
{P1} S {INV},
(INV /\ cast(val(val(e, MeM..STD)),  type(e, MeM, TP), int) = 0)  => Q,
(INV /\ cast(val(val(e, MeM..STD)),  type(e, MeM, TP), int) != 0) => P1
|- {any_predicate(INV)} while(simple_expression(e)) any_code(S)
{any_predicate(Q)}
```

Обозначения `MD`, `MeM` и `STD` относятся к модели памяти языка C-kernel [1] и не влияют коренным образом на структуру правила вывода.

### 3.2. Расширение языка шаблонов семантическими метками

В связи с введением меток в проект C-light язык описания правил вывода был расширен специальной конструкцией `label` [11], используемой для описания меток. Конструкция `label` имеет вид `(label t c)`, где `t` — терм, к которому приписана метка, а `c` — строка (тип метки). Конструкция `label` обеспечила переход в проекте C-light от обычных правил Хоара к размеченным.

## 4. Реализация метагенератора

При рассмотрении реализации метагенератора необходимо обратить внимание на разницу между ней и идеями Морикони и Шварца [2]. Наш метагенератор представляет

собой функцию с двумя параметрам [3], т. е. при его работе совершается каррирование. Таким образом, если $H$ — аксиоматическая семантика и $AP$ — аннотированная программа, которую нужно верифицировать, то

$$\mathrm{MetaVCG}(H, AP) = \mathrm{VCG}_H(AP),$$

где $\mathrm{VCG}_H$ — ГУК, построенный по $H$. Это не лучшее решение с точки зрения эффективности, так как каждый эксперимент по верификации (аргумент $AP$) приводит к порождению генератора, даже когда аксиоматическая семантика остается той же самой (например, чаще всего в экспериментах используется аксиоматическая семантика языка C-kernel). С другой стороны, это позволяет нам верифицировать одну программу вместо двух, одна из которых порождалась бы без спецификаций.

### 4.1. Сопоставление программных конструкций и шаблонов

Аргументы метагенератора — аксиомы и правила вывода вместе с аннотированной программой — анализируются и транслируются в соответствующее внутреннее представление. Отметим, что для этого используется в том числе API, предоставляемое компилятором Clang [3].

Так как API Clang имеет встроенный лексический и синтаксический анализатор Си-программ, то использование API Clang позволяет облегчить реализацию этой функциональности. На первом этапе анализатор с помощью инструментария, предоставляемого API Clang, строит внутреннее представление шаблона. Данное представление ориентировано на язык C++. Так как важной задачей проекта C-light является частичная самоприменимость, внутреннее представление на следующем шаге переводится в структуру `pattern_node`, совместимую с C-light. В результате работы анализатора шаблонов получается внутреннее представление шаблонов — множество деревьев, узлами которых являются структуры `pattern_node`.

Тип данных `pattern_node` представляет аксиомы и заключения правил вывода. Такая структура задает дерево, первый и последний узлы которого являются соответственно предусловием и постусловием. Каждый узел имеет атрибуты (категорию, идентификатор, тип), которые активно используются в процессе сопоставления данному шаблону программных конструкций. Также одним из атрибутов является таблица соответствия между идентификаторами программы и шаблона, которая заполняется в процессе сопоставления. Внутренним представлением программы служит древовидная структура `program_node`, подобная структуре `pattern_node`.

Таким образом, метагенератор строит деревья как для аннотированной программы, так и для набора аксиом и правил вывода. В соответствии с направлением вывода для правейшей/левейшей программной конструкции происходит вывод соответствующего ей заключения правила вывода. Для выбранного шаблона процедура рекурсивно применяется к его посылкам.

Необходимо отметить, что задача сопоставления программной конструкции и шаблона правила вывода решается с помощью алгоритма сопоставления деревьев [3]. На текущий момент он реализован как “жадный” алгоритм. Корректность его применения гарантируется простотой аксиоматической семантики языка C-kernel [1] и ограничениями языка C-light (например, запретом на передачу управления в составные операторы извне) [1]. Аксиоматическая семантика правил вывода из разд. 6 также позволяет применять данный алгоритм. Поэтому на данном этапе развития проекта C-light текущая

реализация сопоставления нас устраивает. При использовании в проекте C-light новых наборов аксиом и правил вывода может возникнуть необходимость перехода к более сложной реализации сопоставления.

### 4.2. Реализация системы локализации ошибок

При генерации объяснений для недоказанных УК происходит их упрощение по правилам нормализации [10]. При этом в нашем подходе, в отличие от подхода Денни и Фишера [10], не применяется группа правил, осуществляющих "просачивание" меток через конъюнкцию и (вложенную) импликацию, а также правило устранения вложенности [11].

Таким образом, метки в дереве УК продолжают быть структурой, также представляющей собой дерево. Следовательно, в нашем подходе, в отличие от подхода Денни и Фишера [10], метки в условиях корректности образуют некоторую иерархию, что используется при генерации текста на естественном языке. Для этого дерево меток обходится в глубину и для каждой рассмотренной метки к общему тексту добавляется текст ее заполненного номерами строк шаблона. Текстовые шаблоны для каждой концепции метки, подобные форматной строке в языке Си, также подаются на вход метагенератора. Таким образом в проекте C-light работает система локализации ошибок.

## 5. Эксперимент по локализации ошибки

В ходе развития проекта C-light проведен эксперимент по верификации функции `max`, которая ищет индекс максимального элемента массива. Ее спецификации написаны на языке автоматического доказателя теорем Simplify и представляют собой обычные Си-комментарии. Такой подход позволяет сохранить семантику программы. Конструкции `AND`, `OR`, `IMPLIES` и `FORALL` языка спецификаций представляют собой логическое и, или, импликацию и квантор всеобщности соответственно. Спецификации, расположенные перед циклом, а также до и после определения функции, являются инвариантом цикла, а также предусловием и постусловием функции соответственно. Отметим, что в ходе эксперимента каждая аннотация располагалась только на одной строке. Рассмотрим C-kernel-вариант функции `max`, снабженный номерами строк для понимания работы системы локализации ошибок и объяснения недоказанных УК:

```
1.  // (AND (NEQ a |@NULL|)(> length 0))
2.  int max(int* a, int length)
3.  {
4.      auto int x = 0;
5.      auto int y = length - 1
6.      /* (AND (<= 0 x)(< x length)(<= 0 y)
                (< y length)(>= y x)
                (FORALL (i) (IMPLIES (AND (<= 0 i) (<= i x))
                                     (OR (<= a[i] a[x]) (<= a[i] a[y]))))
                (FORALL (i) (IMPLIES (AND (<= y i)
                                          (<= i (- length 1)))
                                     (OR (<= a[i] a[x])
                                     (<= a[i] a[y])))))) */
7.      while (x == y)
```

```
8.      {
9.          if (a[x] <= a[y]) {x = x + 1; } else {y = y - 1;}
10.     }
11.     return x;
12. }
13. /* (AND (<= 0 x)(< x length)
            (FORALL (i) (IMPLIES (AND (<= 0 i) (< i length))
                                      (>= a[x] a[i])))) */
```

Ошибка в программе заключается в использовании оператора `==` в условии цикла `while` вместо оператора `!=`. Поэтому было получено недоказанное УК:

$$\left(\begin{array}{l}\left[\begin{array}{l}0 \le MD(MeM(x)) \wedge MD(MeM(x)) < MD(MeM(length)) \wedge \\ 0 \le MD(MeM(y)) \wedge \\ MD(MeM(y)) < MD(MeM(length)) \wedge \\ MD(MeM(y)) \ge MD(MeM(x)) \wedge \\ \forall i \left(\begin{array}{l} 0 \le MD(MeM(i)) \wedge MD(MeM(i)) \le MD(MeM(x)) \\ \Rightarrow \left(\begin{array}{l} MD(MeM(a), MD(MeM(i))) \le MD(MeM(a), MD(MeM(x))) \vee \\ MD(MeM(a), MD(MeM(i))) \le MD(MeM(a), MD(MeM(y))) \end{array}\right) \end{array}\right) \\ \forall i \left(\begin{array}{l} MD(MeM(y)) \le MD(MeM(i)) \wedge \\ MD(MeM(i)) \le MD(MeM(length)) - 1 \\ \Rightarrow \left(\begin{array}{l} MD(MeM(a), MD(MeM(i))) \le MD(MeM(a), MD(MeM(x))) \vee \\ MD(MeM(a), MD(MeM(i))) \le MD(MeM(a), MD(MeM(y))) \end{array}\right) \end{array}\right) \end{array}\right]^{ass\_inv\_exit(6)} \wedge \\ \left[cast\left(\begin{array}{l} val(val(MD(MeM(x)) = MD(MeM(y))), MeM...STD)), \\ type(MD(MeM(x)) = MD(MeM(y))), MeM, TP), \mathsf{int} \end{array}\right) = 0\right]^{while\_ff(7)} \end{array}\right)$$
$$\Rightarrow \left(\begin{array}{l} 0 \le MD(MeM(x)) \wedge MD(MeM(x)) < MD(MeM(length)) \wedge \\ \forall i \left(\begin{array}{l} 0 \le MD(MeM(i)) \wedge MD(MeM(i)) < MD(MeM(length)) \\ \Rightarrow MD(MeM(a), MD(MeM(x))) \ge MD(MeM(a), MD(MeM(i))) \end{array}\right) \end{array}\right)$$

Продемонстрированное УК является слишком сложным для анализа “вручную”. В результате генерации объяснения по данному УК получается следующий текст:

```
This VC corresponds to function ‘‘max".
Hence, given
- assumption that loop invariant holds without loop entry at line 6,
- assumption that the loop condition doesn’t hold at line 7.
```

Рассматриваемое УК означает, что из выполнения инварианта и невыполнения условия цикла следует постусловие. Таким образом, данная формула соответствует выходу из цикла. Соответственно, рассматриваемый текст явно указывает на то, что необходимо обратить внимание на допущения о выполнении инварианта цикла (строка 6) и невыполнении условия цикла (строка 7). При детальном рассмотрении допущения о невыполнении условия цикла можно прийти к пониманию того, что есть проблема с осуществлением входа в цикл (строка 7). Следовательно, для обнаружения ошибки необходимо рассмотреть условие цикла. Заметим, что именно неправильное условие цикла представляет собой искомую ошибку. Таким образом, данный текст содержит информацию, которая помогает локализовать ошибку. Этот пример демонстрирует работу системы локализации ошибок и объяснения недоказанных УК.

## 6. Подходы к элиминации циклов в проекте C-light

Проблема автоматической генерации инвариантов циклов является алгоритмически неразрешимой. Но для циклов специального вида она разрешима с помощью символического метода верификации финитных итераций [4].

### 6.1. Символический метод верификации финитных итераций

Приведем описание данного метода из [4]. Предположим, что тело финитной итерации состоит из последовательности операторов присваивания и условных операторов. Представим его в виде оператора векторного (т. е. одновременного) присваивания $v = body(v, x)$, где $x$ — параметр итерации; $v$ — вектор остальных переменных; $body(v, x)$ — вектор условных выражений, построенных с помощью операции $if - then - else$. Такое представление получается посредством последовательности подходящих подстановок, которые осуществляют замену условных операторов условными выражениями и последовательности операторов присваивания одним оператором присваивания.

Финитная итерация над структурами данных имеет вид

$$\texttt{for}\ \ x\ \ \texttt{in}\ \ S\ \ \texttt{do}\ \ v\ =\ body(v,\ x)\ \ \texttt{end}, \tag{3}$$

где $S$ — структура, возможно, иерархическая. Пусть $empty(S)$ — предикат, соответствующий пустоте мультимножества элементов структуры $S$, а $v_0$ — начальное значение вектора $v$. При $empty(S)$ результатом считается $v_0$. Если $\neg emptyS$ и $vec(S) = [s_1, ..., s_n]$, то тело цикла исполняется последовательно для $x$, принимающего значение $s_1, ..., s_n$.

Пусть $R(y \leftarrow exp)$ обозначает результат подстановки выражения $exp$ вместо всех вхождений переменной $y$ в формулу $R$. Пусть $R(vec \leftarrow vexp)$ обозначает результат одновременной подстановки в формулу $R$ компонент вектора выражений $vexp$ вместо всех вхождений соответствующих компонент вектора $vec$. Правило вывода условий корректности для итерации (3) использует операцию замены $rep(v, s, body)$, где $body$ обозначает функцию, ассоциированную с правой частью тела итерации (3). Операция замены $rep(v, s, body)$ представляет действие итерации (3) и определяется следующим образом. Пусть $rep(v, s, body) = v_n$, где $v_0 = v$, $n = 0$ при $empty(S)$, а $v_i = body(v_{i-1}, s_i)$ для каждого $i = 1, ..., n$ при $\neg emptyS$ и $vec(S) = [s_1, ..., s_n]$. Следующая теорема [4] описывает полезные свойства операции замены.

**Теорема.** *Итерация (3) эквивалентна оператору векторного присваивания.*

Из данной теоремы вытекает следующее правило вывода условий корректности:

$$\frac{\{P\}\ prog\ \{Q(v \leftarrow rep(v, s, body))\}}{\{P\}\ prog;\ \ \texttt{for}\ \ x\ \ \texttt{in}\ \ S\ \ \texttt{do}\ \ v\ =\ body(v,\ x)\ \ \texttt{end}\ \{Q\}}, \tag{4}$$

где $P$ — предусловие; $Q$ — постусловие, не зависящее от параметра итерации $x$; $prog$ — фрагмент программы, а $\{P\}\ prog\ \{Q\}$ обозначает частичную корректность программы $prog$ относительно $P$ и $Q$.

В результате применения правила порождаются УК, содержащие операцию замены. Для доказательства таких УК применяется как универсальная техника, базирующаяся на принципах индукции, так и проблемно-ориентированная техника.

### 6.2. Реализация символического метода верификации финитных итераций

Переход к использованию метода метагенерации УК в проекте C-light позволил удобным образом расширять ГУК новыми правилами вывода. Соответственно, внедрение символического метода метагенерации в проект C-light основано на идее метагенерации УК. Рассмотрим это подробнее на примере правила вывода (4).

Так как метод метагенерации УК позволил расширить проект C-light методом семантической разметки, на вход метагенератора стало возможно подавать размеченные

семантическими метками правила вывода. Поэтому правило вывода (4) также было снабжено семантическими метками:

$$\frac{\{P\}\ prog\ \{\ulcorner Q(v \leftarrow \ulcorner rep(v,s,body)\urcorner^{rep\_iter})\urcorner^{for\_iter}\}}{\{P\}\ prog;\quad \texttt{for}\quad x\quad \texttt{in}\quad S\quad \texttt{do}\quad v\ =\ body(v,\ x)\quad \texttt{end}\ \{Q\}}.$$

Метод метагенерации УК позволяет не переписывать ГУК "вручную" для добавления новых концепций меток. Это позволило удобным образом задать для рассматриваемого правила две новых концепции меток — $rep\_iter$ и $for\_iter$. Меткой с концепцией $rep\_iter$ снабжается подформула, образованная операцией замены. Меткой с концепцией $for\_iter$ обозначается постусловие, измененное подстановкой операции замены.

Данное правило вывода соответствует ограничениям нормальной формы и поэтому может быть подано на вход метагенератора УК. Рассмотрим данное правило на языке написания аксиом и правил вывода, описанном в разд. 3 и расширенном конструкцией для написания семантических меток:

```
{P} prog {(label for_iter substitution(Q, v,
                          (label rep_iter rep(v, S, body))))}
|- {any_predicate(P)} any_code(prog) for_iteration(x, v, S, body)
{any_predicate(Q)}
```

Здесь `substitution(Q, v, rep(v, S, body))` — обозначение операции подстановки в `Q` вместо `v` конструкции `rep(v, S, body)`, а `for_iteration(x, v, S, body)` — обозначение финитной итерации.

Отметим, что `for_iteration(x, v, S, body)` представляет собой шаблон, которому будут сопоставляться программные конструкции в ходе генерации УК. Реализованный в метагенераторе алгоритм позволяет успешно сопоставить такому шаблону программную конструкцию, представляющую собой финитную итерацию.

Заметим, что благодаря методу метагенерации УК реализация символического метода верификации в проекте C-light не вызвала затруднений. Это подтвердило полезность поддерживаемого в нашем проекте принципа расширяемости.

### 6.3. Верификация программ линейной алгебры

В символическом методе верификации финитных итераций проблема элиминации инвариантов циклов рассматривается для широкого класса структур данных. Исследуем данную проблему для структур данных, используемых в программах линейной алгебры [7]. Для таких программ характерны такие циклы `for`, в которых осуществляются перестановка строк матрицы, деление строки на число, вычитание из элементов одной строки, соответствующих другой, умноженных на число. Результаты таких циклов, а значит и их инварианты, можно выразить с помощью операции замены $rep$. Это позволяет сформулировать специальные правила вывода УК, не использующие инварианты циклов. Но для применения таких правил необходимо выполнить специальные условия.

Рассмотрим пример такого правила из [7]. Пусть для матрицы $M$, множества индексов матрицы $S$ и выражения $e(s,t)$ через $rep(M,S,e(s,t))$ обозначим матрицу, полученную из $M$, если для каждого элемента $(s,t) \in S$ в качестве $(s,t)$-го компонента взять значение выражения $e(s,t)$. Так как в проекте C-light поддерживается метод семантической разметки, правила вывода для линейной алгебры снабжены семантическими

метками. Тогда правило для замены части строки матрицы выражением, зависящим только от этой строки и текущего столбца матрицы, примет следующий вид:

$$\frac{\{P\}\ prog\ \{\ulcorner Q(M \leftarrow \ulcorner rep(M, row(i, e_1, e_2), e(s,t))\urcorner^{rep\_row})\urcorner^{for\_row}\}}{\{P\}\ prog;\quad \texttt{for}(k = e_1; k <= e_2; k++)\quad M[i][k] = e(i,k)\quad \{Q\}}.$$

Пусть $row(l, m, n)$ — множество, определяющее $l$-ю строку матрицы от $m$-го до $n$-го столбца. Следовательно, $(u, v) \in row(l, m, n) \iff (u = l \wedge m \leq v \leq n)$ . Тогда данное правило имеет следующее условие применения: *если* $e(i, k)$ *зависит от* $M[m][n]$, *то* $(m, n) \notin row(i, e_1, k-1)$. Заметим, что для правил, относящихся к программам линейной алгебры и приведенным в [7], условия применимости обеспечивают главным образом независимость операции $rep(M, S, e(s, t))$ от порядка выбора элементов множества $S$.

Примечательно, что в отличие от метода Денни и Фишера [10] проект C-light может быть расширен новыми концепциями семантических меток удобным образом, включая концепции $rep\_row$ и $for\_row$. Меткой с концепцией $rep\_row$ снабжается подформула, образованная операцией замены. Меткой с концепцией $for\_row$ обозначается постусловие, измененное подстановкой операции замены.

Данное правило вывода удовлетворяет ограничениям нормальной формы и поэтому может быть подано на вход метагенератора УК. Рассмотрим данное правило, снабженное семантическими метками, на языке написания аксиом и правил вывода:

```
{P} prog {(label for_row substitution(Q, M,
                  (label rep_row rep(M, row(i, e1, e2), e(s, t))))))},
|- {any_predicate(P)} any_code(prog) for_row(M, i, k, e1, e2, e(s, t))
{any_predicate(Q)}
```

Здесь `substitution(Q, M, rep(M, row(i, e1, e2), e(s, t)))` — обозначение операции подстановки в `Q` вместо `M` конструкции `rep(M, row(i, e1, e2), e(s, t))`, при этом `for_row(M, i, k, e1, e2, e(s, t))` — шаблон, обозначающий итерацию над строкой матрицы.

Алгоритм сопоставления программных конструкций и шаблонов, описанный в подразд. 4.1 и реализованный в метагенераторе УК, успешно сопоставит подходящие итерации над строкой матрицы данному шаблону, проверив при этом условие применимости. Такой алгоритм находит подходящие `while`-итерации, так как в языке C-kernel отсутствуют `for`-циклы. Но рассматриваемый набор методов элиминации циклов открывает возможности для расширения языка C-kernel циклом `for`.

Заметим, что элиминация циклов при итерации над матрицей позволяет облегчить доказательство УК относительно символического метода верификации финитных итераций над структурами данных в общем виде. Чтобы доказать условие корректности, полученное символическим методом верификации финитных итераций, нужно отдельно решить задачу задания аксиом для операции замены $rep$ в каждом конкретном случае. Такая аксиоматика необходима для работы автоматического доказателя теорем. В случае работы с программами линейной алгебры задать аксиоматику операции замены $rep$ легче, так как свойства такой структуры данных, как матрица, общеизвестны. При работе с большим количеством программ линейной алгебры можно заранее задать аксиоматику операции замены $rep$ для соответствующего набора правил вывода.

Данный подход к циклам в программах линейной алгебры позволяет облегчить этап доказательства УК относительно использования аксиоматической семантики C-kernel

в общем виде. При выводе в общем случае при обработке цикла произошла бы генерация двух дополнительных УК (при сравнении рассматриваемого правила вывода с правилом вывода из подразд. 3.1). На практике увеличение количества УК часто приводит к увеличению времени, затраченному на их доказательство.

## Заключение

В проекте C-light важнейший целью является создание системы верификации, удобной не только для специалистов-теоретиков, но и для обычных программистов. Для достижения этой цели необходимо решить ряд задач. Во-первых, мы стремимся гарантировать корректность нашей системы. Ситуация, при которой система верификации написана на целевом языке, дает возможность применить ее к самой себе.

Для создания "удобно верифицируемого верификатора" мы применили метод метагенерации УК и реализовали его с некоторыми модификациями, используя в основном язык C-light. Этот программный код был снабжен аннотациями. Большой интерес для верификации представляет фрагмент метагенератора, обрабатывающий таблицу соответствия метаидентификаторов шаблона и идентификаторов программной конструкции. Такая таблица строится в ходе работы процедуры сопоставления шаблона и программной конструкции. Данная таблица реализована как двумерный массив. Реализация обрабатывающего ее фрагмента метагенератора основана на использовании циклов, которые представляют собой итерации над этим массивом.

Следовательно, такие методы элиминации инвариантов циклов, как символический метод верификации финитных итераций, могут упростить верификацию данного фрагмента. Поэтому проведение верификации рассматриваемого фрагмента с помощью данного метода входит в планы по дальнейшей работе над системой C-light. Мы планируем продолжать работу по аннотированию и верификации различных компонентов нашей системы, чтобы добиться одной из наших целей — полной самоверификации.

Во-вторых, необходимо обеспечить расширяемость проекта C-light. Метод метагенерации позволяет это сделать, что продемонстрировало удобство внедрения в наш проект метода символической верификации финитных итераций. По нему вводятся итерация над структурой данных специального вида и правило вывода для нее. Метод Морикони и Шварца [2] позволил подавать такое правило как аргумент метагенератора.

Для определенных классов программ можно предложить более частные случаи итерации над структурами данных. Например, для программ линейной алгебры были предложены итерации специального вида над матрицами и правила вывода для них. Внедрение таких правил вывода в проект C-light с помощью метода метагенерации УК демонстрирует возможность проведения эффективной верификации, ориентированной на определенные классы программ. Такая эффективность выражается в облегчении этапа доказательства УК. В ходе дальнейшей работы планируется рассмотрение возможности проведения эффективной верификации и для других классов программ, например для программ, реализующих телекоммуникационные протоколы.

В-третьих, необходимо предоставить пользователю инструментарий для анализа недоказанных УК. Примером, демонстрирующим преимущество расширяемости, является внедрение в проект C-light такого инструментария. Метод Денни и Фишера [10] позволяет генерировать текстовые объяснения для недоказанных УК с помощью меток, которыми УК снабжаются при выводе. В ходе работы над проектом C-light принято решение сохранять иерархию на метках в отличие от идей Денни и Фишера [10].

В ходе исследования проведены эксперименты по верификации программ с заранее внесенной в них ошибкой. Один из таких экспериментов описан в данной работе. Заметим, что по полученным в ходе экспериментов объяснениям на естественном языке было возможно локализовать местоположение ошибок.

Также отметим, что Денни и Фишер [10] предлагают строго определенный набор концепций меток. Метод Морикони и Шварца [2], в котором входными аргументами являются правила вывода, позволяет подать на вход правила вывода, снабженные любыми семантическими метками. Благодаря этому были введены новые концепции семантических меток для правил вывода, элиминирующих инварианты циклов. В ходе дальнейшей работы планируется рассмотреть другие методы, которыми проект C-light может быть расширен с помощью метода метагенерации.

**Благодарности.** Работа выполнена при частичной поддержке РФФИ (грант № 15-01-05974).

## Список литературы / References

[1] **Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В.** Ориентированный на верификацию язык C-light // Системная информатика: Сб. науч. тр. / РАН. Сиб. отд-ние. Ин-т систем информатики. 2004. Т. 9. С. 51–134.
**Nepomniaschy, V.A., Anureev, I.S., Mikhaylov, I.N., Promskiy, A.V.** Verification oriented language C-light // Sistemnaya Informatika: Sb. Nauch. Tr. / RAN. Sib. Otd-nie. In-t Sistem Informatiki. 2004. Vol. 9. P. 51–134. (In Russ.)

[2] **Moriconi, M., Schwarts, R.L.** Automatic construction of verification condition generators from hoare logics // Lecture Notes in Computer Science. Berlin: Springer-Verlag, 1981. Vol. 115. P. 363–377.

[3] **Кондратьев Д.А., Промский А.В.** Разработка самоприменимой системы верификации. Теория и практика // Моделирование и анализ информационных систем. 2014. Т. 21, № 6. С. 70–81.
**Kondratyev, D.A., Promsky, A.V.** Developing a self applicable verification system. Theory and practices // Automatic Control and Computer Sciences. 2015. Vol. 49, No. 7. P. 445–452.

[4] **Непомнящий В.А.** Символический метод верификации финитных итераций над изменяемыми структурами данных // Программирование. 2005. № 1. С. 3–14.
**Nepomniaschy, V.A.** Symbolic method of verification of definite iterations over altered data structures // Programming and Computer Software. 2005. Vol. 31, No. 1 P. 1–9.

[5] **Apt, K.R., Olderog, E.R.** Verification of sequential and concurrent programs. Berlin: Springer-Verlag, 1991. 450 p.

[6] **Непомнящий В.А., Сулимов А.А.** Верификация программ линейной алгебры в системе СПЕКТР // Кибернетика и системный анализ. 1992. № 5. С. 136–144.
**Nepomniaschy, V.A., Sulimov, A.A.** Verification of linear algebra programs in the SPECTRUM system // Cybernetics and System Analysis. 1992. No. 5. P. 136–144. (In Russ.)

[7] **Непомнящий В.А., Рякин О.М.** Прикладные методы верификации программ. М.: Радио и связь, 1988. 256 с.
**Nepomniaschy, V.A., Ryakin, O.M.** Applied methods of program verification. Moscow: Radio i Svyaz', 1988. 256 p. (In Russ.)

[8] **Filliâtre, J.C., Marché, C.** Multi-prover verification of C programs // Lecture Notes in Computer Science. Berlin: Springer-Verlag, 2004. Vol. 3308. P. 15–29.

[9] **Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.** VCC: A practical system for verifying concurrent C // Lecture Notes in Computer Science. Berlin: Springer-Verlag, 2009. Vol. 5674. P. 23–42.

[10] **Denney E., Fischer B.** Explaining verification conditions // Lecture Notes in Computer Science. Berlin: Springer-Verlag, 2008. Vol. 5140. P. 145–159.

[11] **Кондратьев Д.А.** Расширение метагенерации условий корректности концепцией семантической разметки // Тр. Междунар. науч.-практ. конф. "Инструменты и методы анализа программ". СПб.: Изд-во Политехн. ун-та, 2015. С. 107–118.
**Kondratyev, D.A.** The extension of the MetaVCG approach by semantic mark-up concept // Proc. of the Intern. Workshop Conf. "Tools & Methods of Program Analysis". Sankt-Peterburg: Izd-vo Politekhn. Univ., 2015. P. 107–118. (In Russ.)

*Поступила в редакцию 13 февраля 2017 г.*

**The extension of the C-light project using symbolic verification method of definite iterations**

Kondratyev, Dmitry A.

Institute of Informatics Systems SB RAS, Novosibirsk, 630090, Russia

Corresponding author: Kondratyev, Dmitry A., e-mail: apple-66@mail.ru

The project for deductive verification of C programs is being developed in the Laboratory of theoretical programming, IIS SB RAS. One of the goals of this project is development of an extendable self-applicable verification condition generator for the C language. The MetaVCG approach was chosen to achieve this purpose. The metagenerator utilizes axiomatic semantics in a special format and automatically produces the verification condition generator. The correctness and completeness of MetaVCG are ensured by using only strongly limited axiomatic semantics as the argument of metagenerator.

The metageneration idea allows the C-light verification system to be supplemented with the semantic mark-up method. It focuses on such problems as the analysis, tracing and explanation of verification conditions. This concept contains such definition of Hoare rules extension by semantic labels, that calculus itself can be used to generate explanations of verification conditions. The error localization experiments were successfully performed.

The symbolic method of verifying for-loops that have the statement of assignment to array elements as the loop body is based on the replacement operation. It represents the loop effect in a symbolic form and allows the proof rule to be created for these loops without invariants. The MetaVCG approach allows the C-light system to be easily supplementing with such rules and new concepts of semantic labels.

The application of C-light system to the verification of linear algebra programs is explained in this article. The MetaVCG approach allows special set of inference rules to be used for verifying such programs.

*Keywords*: verification, MetaVCG, symbolic method of verification, definite iterations, verification condition, semantic mark-up method.

**Acknowledgements.** This research was partly supported by RFBR (grant No. 15-01-05974).

*Received 13 February 2017*