Расширение метагенерации условий корректности концепцией семантической разметки

Кондратьев Д.А.
Лаборатория Теоретического Программирования
Институт Систем Информатики им. А.П. Ершова
г. Новосибирск, Россия
apple-66@mail.ru

Верификация программ традиционно основана на генерации условий корректности (УК). В проекте C-light по созданию системы дедуктивной верификации Си-программ возникла задача разработки расширенных или специализированных версий генераторов УК. Для решения данной задачи было решено использовать концепцию метагенерации УК. Получая на вход логику Хоара, метагенератор автоматически порождает рекурсивный алгоритм генерации УК. Идея метагенерации позволила органично дополнить систему C-light и концепцией семантической разметки, ориентированной на такую важную задачу, как анализ, трассировку и объяснение самих УК. Основываясь на предложенным Денни и Фишере методе, опишем такое расширение правил Хоара семантическими метками, что само по себе исчисление может использоваться для построения объяснений УК. Объяснения могут быть построены для различных аспектов УК, в этой статье рассмотрим построение объяснений для их структуры и целей. Также опишем отличие нашего метода от метода Денни и Фишера, заключающееся во введении иерархии на метках.

Ключевые слова — верификация; семантическая метка; условие корректности; язык Си; язык C-light; метод метагенерации

I. Введение

Обеспечение надежности программ является одним из важнейших направлений программирования. Наиболее часто используемым методом обеспечения надежности является тестирование, однако это дорогостоящий и сложный процесс, к тому же с его помощью невозможно обнаружить все ошибки и уязвимости программных продуктов. Совершенно иной подход в решении этой проблемы – это формальная верификация программы. Он является актуальным направлением современного программирования и заключается в формальном доказательстве корректности программ в соответствии с описанием их свойств, задаваемых в виде спецификаций программ.

Работа частично поддержана грантом РФФИ № 15-01-05974 «Онтологический подход к формальной семантике языков программирования».

Наличие формальной семантики для языка программирования является необходимым условием разработки метода верификации программ. В лаборатории теоретического программирования ИСИ СО РАН развивается проект по верификации программ на языке C-light [6], который является представительным подмножеством языка Си. Традиционно, наиболее естественным подходом при конструировании семантики является операционный подход. Операционный подход, фактически, задает некоторую абстрактную машину, исполняющую программу. Однако низкий уровень подобного формализма неудобен для верификации, и, как правило, применяется метод дедуктивной верификации, основанный на аксиоматической семантике.

С другой стороны, аксиоматический поход для такого языка, как C-light, может быть слишком громоздким, поэтому был предложен двухуровневый метод [11] верификации программ. На первом этапе C-light транслируется в промежуточный язык C-kernel с целью элиминации некоторых конструкций, сложных для аксиоматической семантики [15]. Для трансляции используется набор формальных правил [13]. На втором этапе для промежуточной C-kernel программы генерируются условия корректности (УК), которые в дальнейшем передаются на блок доказательства [12]. Генератор условий корректности (ГУК) создается по аксиоматической семантике языка C-kernel. Сама аксиоматическая семантика является набором аксиом и правил вывода для языковых конструкций.

В ходе развития проекта важными стали две задачи. Во-первых, необходим способ простого добавления новых языковых конструкций, не требующий значительной переработки генератора. Во-вторых, интерес представляют узкоспециализированные версии генератора, ориентированные на конкретные методы вывода и классы приложений и позволяющие упростить верификацию. Для решения этих задач было решено использовать предложенную Морикони и Шварцем [7] концепцию метагенерации условий корректности.

Метагенератор принимает на входе правила вывода аксиоматической семантики в специальном формате и автоматически порождает генератор условий

корректности. Заметим, что порождаемый генератор сам задает некоторую семантику языка программирования. Поэтому, встает вопрос о его непротиворечивости относительно исходной аксиоматической семантики. Эта непротиворечивость обеспечивается тем, что на вход метагенератору подаются только ограниченные правила вывода, находящиеся в нормальной форме [7], а сам алгоритм построения генератора строго формализован.

При практическом применении дедуктивной верификации могут возникнуть следующие проблемы: программа может быть не корректна, ее спецификации могут быть не корректны, автоматический доказатель теорем может не обладать достаточной мощностью, теория предметной области может быть не полна. В этих случаях пользователь системы верификации получает набор недоказанных условий корректности, но не получает дополнительной информации о причинах неудачи. Он должен проанализировать такие условия корректности, изучить их составные части, определить, какие именно правила вывода были применены, и соотнести их с соответствующими фрагментами программы.

Опишем предложенный Денни и Фишером метод семантической разметки, ориентированный на такую важную задачу, как анализ, трассировку и объяснение самих УК. Идея метагенерации позволила органично дополнить систему C-light данной концепцией. Основываясь на предложенным Денни и Фишере методе, опишем такое расширение правил Хоара семантическими метками, что само по себе исчисление может использоваться для построения объяснений УК. Метки проходят различные стадии обработки и переводятся в объяснения на естественном языке. Генерация объяснений основывается только на анализе меток, а не на анализе логических значении УК или их доказательств. Объяснения могут быть построены для различных аспектов УК, в этой статье рассмотрим построение объяснений для их структуры и целей. Также опишем отличие нашего метода от метода Денни и Фишера, заключающееся во введении иерархии на метках.

Работа имеет следующую структуру. В разделе 1 даются предварительные сведения, необходимые для понимания дальнейшего материала. В разделе 2 рассмотрены теоретические основы метода метагенерации условий корректности и метода Денни и Фишера. В разделе 3 описан язык представления правил вывода. В разделе 4 описана реализация метагенератора и поддержки семантических меток в проекте C-light. В заключении сформулированы результаты проделанной работы

II. Теоретические основы

Рассмотрим в этом разделе основные понятия аксиоматической семантики, метод метагенерации и метод семантической разметки.

A. Аксиоматическая семантика

В 1964 году Ч. Хоар [1] ввел способ задания аксиоматической семантики, ставший основой метода дедуктивной верификации программ. Подход Хоара заключается в том, чтобы представлять текст программы как особое отношение между утверждениями. Базовыми формулами в рассматриваемом подходе являются тройки Хоара {P} S {Q}, где P — предусловие (логическая формула), S — программа, Q — постусловие (логическая формула). Истинность формулы {P} S {Q} означает, что «если предусловие P истинно перед исполнением фрагмента программы S, и, если исполнение S завершилось, тогда постусловие Q выполняется после его завершения» [7]. Правила вывода задаются в виде

$$\frac{\varphi_1, \ldots, \varphi_n}{\psi},$$

где $\varphi_1, \ldots, \varphi_n$ — посылки правила вывода (набор троек Хоара и логических формул) и ψ — заключение правила вывода (тройка Хоара). Данная нотация означает, что ψ выводимо при гипотезе $\varphi_1, \ldots, \varphi_n$. 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 схемы вывода для всех синтаксических форм языка программирования, называется логикой Хоара или аксиоматической семантикой языка.

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

Одним из способов реализации генератора условий корректности является рекурсивно определенный преобразователь предиката pre(S, Q), отображающий фрагмент программы S и постусловие Q в предусловие. Функция pre вычисляет утверждение, достаточное, чтобы гарантировать, что Q будет являться постусловием (т. е., что тройка Хоара {pre(S, Q)} S {Q} доказуема). Доказуемости условия корректности $P \supset pre(S, Q)$ в теории, описывающей предметную область, достаточно, чтобы показать, что {P} S {Q} доказуема в любой системе Хоара, содержащей правило для композиции. Функция wdp(S, Q) вычисляет слабейшее предусловие, т.е. такое предусловие, что для любого другого предусловия R должно быть верно $R \supset wdp(S, Q)$ для программы S и постусловия Q. Для доказуемости тройки Хоара {P} S {Q} необходимо и достаточно, чтобы P было эквивалентно wdp(S, Q) [7].

B. Метод метагенерации

Метод метагенерации, предложенный Морикони и Шварцем [7], состоит из двух шагов. На первом из них

обычная аксиоматическая семантика переводится в нормальную форму, по которой затем создается рекурсивно определенный генератор условий корректности. Что касается генератора условий корректности, то для его построения используется метод слабейшего предусловия [1].

Следуя Морикони и Шварцу, мы будем использовать метапеременные P, Q, Γ, ... для обозначения частично интерпретированных формул первого порядка. Рассмотрим бинарное отношение $\Leftarrow$ на неинтерпретированных предикатных символах. Для тройки Хоара вида

$$\{P(P_1, ...,P_m)\}\ S\ \{Q(Q_1, ...,Q_n)\}$$

где предикатные символы $P_1,...,P_m$ и $Q_1,..., Q_n$ свободно входят в P и Q соответственно, имеем

$$P_i \Leftarrow Q_j, \text{ для } i \in \{1, ..., m\} \text{ и } j \in \{1, ..., n\}$$

Нотация $\overset{+}{\Leftarrow}$ обозначает транзитивное замыкание $\Leftarrow$.

Аналогично, для правила в форме

$$\frac{\{P_1\}\ S_1\ \{Q_1\}, ..., \{P_n\}\ S_n\ \{Q_n\}, \Gamma}{\{P\}\ S\ \{Q\}},$$

отношение $<<$ определяет зависимость выводимости S от выводимости $S_1, ..., S_n$. В частности, верно $S << S_i$ для $i \in \{1,...,n\}$. Для системы аксиом логики Хоара очевидным образом определим транзитивное замыкание <<+.

Мы используем функцию *FreePreds*, чтобы обозначить множество свободных предикатных символов. Мы будем использовать функцию *FragVars* для обозначения множества "фрагментных переменных" (в терминах Морикони и Шварца) в фрагменте программы S из тройки Хоара $\{P\}\ S\ \{Q\}$.

Определим нотацию для связанного вхождения неинтерпретированного предикатного символа в правило. Для правила R предикатный символ из *FreePreds*(R) является связанным в R, если он входит в *FragVars*(R). Иначе, вхождение будет являться свободным в R.

Определение. Правило в нормальной форме — это любое правило N вида:

$$\frac{\{P_1\}\ S_1\ \{Q_1\}, ..., \{P_n\}\ S_n\ \{Q_n\}, \Gamma}{\{P\}\ S\ \{Q\}},$$

которое удовлетворяет следующим ограничениям:

1) $P_1,..., P_n$ и Q являются предикатными символами, свободно входящими в N.

2) Γ является предложением в предметной области, чьи свободно входящие предикатные символы могут входить только в *FreePreds(N)* или во *FragVars(S)*.

3) Фрагментные переменные из каждого S_i должны быть связанными в S. То есть, это должно означать, что $\cup_{1\le j\le n} FragVars(S_j) \subseteq FragVars(S)$.

4) Порядок зависимостей. Тройки Хоара из посылок N должны удовлетворять двум ограничениям:

a) $P_i \overset{+}{\Leftarrow} P_j \supset i < j$.

b) $T \overset{+}{\Leftarrow} U \wedge \neg(\exists R)U \overset{+}{\Leftarrow} R \supset U \equiv Q \vee U$ *связан в* N

5) Монотонность. Пусть P$[P \leftarrow false, P \in s]$ обозначает P прямое с заменой на false всех предикатов P из множества s. Тогда, следующее ограничение на P должно быть удовлетворено:

$$P[P_1,..., P_n, Q \leftarrow true] \vee \forall s \subseteq \{P_1,..., P_n, Q\}\ \neg P[P \leftarrow false, P \in s]$$

Это ограничение должно выполняться для Γ и для всех Q_i.

Два ограничения применяются к набору правил вывода, находящихся в нормальной форме: (i) Любая терминальная строка σ в языке программирования может быть представлена не более чем одним фрагментом S, входящем либо в аксиому, либо в правило вывода. (ii) Отношение <<+ должно быть иррефлексивным.

Необходимо отметить, что ограничения нормальной формы приводят для правил вывода к появлению порядка на посылках и к нетипичному виду предусловий в тройках Хоара.

Для находящегося в нормальной форме правила вывода вида

$$\frac{\{P_1\}\ S_1\ \{Q_1\}, ..., \{P_n\}\ S_n\ \{Q_n\}, \Gamma}{\{P\}\ S\ \{Q\}}$$

слабейшее предусловие (wdp) определяется следующим образом:

$$wdp(S, Q) = P[P_1 \leftarrow wdp(S_1, Q_1), ..., P_n \leftarrow wdp(S_n, Q_n)] \wedge$$
$$(\forall \bar{v})\Gamma[P_1 \leftarrow wdp(S_1, Q_1), ..., P_n \leftarrow wdp(S_n, Q_n)],$$

где $[P_1 \leftarrow t_1, ..., P_n \leftarrow t_n]$ обозначают n подстановок, выполненных последовательно слева направо, и $\bar{v}$ является множеством всех свободных логических переменных в Γ.

Напомним, что ограничения нормальной формы приводят к появлению порядка на посылках и к непривычному виду предусловий в тройках Хоара. Для решения данной проблемы Морикони и Шварц ввели менее ограниченную общую форму.

Заметим, что алгоритм построения генератора условий корректности по правилам вывода применяется к нормальной форме. Чтобы избавить пользователя от необходимости перевода правил вывода из общей формы в нормальную, можно позволить ему задавать правила вывода в общей форме. А сам перевод доверить алгоритму, формализованному Морикони и Шварцем [7].

Условия, при которых порождаемый генератор условий корректности является непротиворечивым и полным как система вывода относительно поданной на вход метагенератору аксиоматической семантики были найдены Морикони и Шварцем. Чтобы обеспечить

корректность и полноту порождаемого генератора необходимо и достаточно подавать на вход метагенератору только ограниченные правила вывода, находящиеся в нормальной форме.

C. Метод семантической разметки

Структура УК. После упрощения, УК обычно имеют вид Хорновских дизъюнктов (т.е. $H_1 \wedge ... \wedge H_n \supset C$). В данном представлении единственное заключение C можно рассматривать как *цель*. Однако, для более осмысленного описания структуры, мы должны дать более детальную характеристику подформул. Такую информацию нельзя восстановить, исходя только из условий корректности или кода программы, поэтому, ее необходимо задавать явно. Ключевая особенность подхода Денни и Фишера [2] состоит том, что различные подформулы располагаются на специальных позициях в правилах Хоара, и, исходя из этого, ГУК добавляет соответствующие метки к УК.

Концепции. Базовой информацией для генерации объяснений является множество основных концепций, которое зависит от специфических аспектов объясняемых УК. Для объяснения структуры УК большинство концепций отражают *позицию* подформулы в нем. Другие концепции передают информацию о первичной и вторичной *цели* подформулы.

Гипотезы подразделяются на *утверждения* и на *предикаты, управляющие потоком исполнения*. Утверждениями называются вошедшие в УК аннотации из программы. Они включают в себя предусловия и постусловия (метки *ass_pre* и *ass_post*), предусловия и постусловия функций (метки *ass_fpre* и *ass_fpost*) и инварианты циклов. Так как правило вывода для циклов использует инварианты как гипотезы в двух разных позициях и целях, то для подчеркивания различий были введены метки *ass_inv* и *ass_inv_exit*. Управляющими потоком исполнения предикатами называются подформулы в УК, отображающие ход потока исполнения программы. Для оператора *if* и цикла *while* предикаты, управляющие потоком исполнения, используются в формах с отрицанием и без него, приводя к появлению четырех разных меток: *if_ff*, *if_tt*, *while_ff* и *while_tt*.

Заключения передают информацию об основной цели УК. Как и в случае гипотез, инварианты используются в заключениях в двух разных формах: для обозначения входа в цикл (метка *est_inv*) и для обозначения итерации цикла (метка *est_inv_iter*). Заметим, что одна и та же подформула может являться и гипотезой, и заключением, как в разных УК, так и в одном УК.

Уточнители характеризуют и гипотезы, и заключения, передавая информацию о том, как подформула появилась в УК и как она была преобразована. Например, разные концепции *подстановки* отражают подстановки в используемом исчислении Хоара. Концепции *sub* и *upd* передают то, как сказывается эффект от присваивания и обновления массива на УК.

Индуктивные уточнители передают второстепенную цель УК. Они вводятся, когда рекурсивный вызов ГУК для вложенной языковой конструкции порождает УК, цель которых концептуально связана с целью УК для объемлющей конструкции. Например, такая ситуация характерна для цикла, вложенного в другой цикл.

Структура метки. 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.

Модифицированные правила Хоара. Обычно, недостаточно просто генерировать объяснение того, как УК было получено. Вместо этого, ГУК должен добавлять метки в тех местах, где это нужно. Также он должен предоставлять удобный интерфейс для доступа к разметке. Это нужно для работы с метками, в том числе, и для генерации по ним объяснений. В модифицированные правила Хоара добавляется семантическая разметка (тип меток и их позиции), нужная для объяснения результата применения правила. Метки добавляются к следующим местам: к постусловию посылки, получившемуся при рекурсивном вызове ГУК, к слабейшему предусловию, к сгенерированному УК или к тройке Хоара. Далее, для ясности изложения, опустим в формулировке правила относящуюся к меткам информацию о позиции, но, полагаем, что ГУК получает ее от программных конструкций и аннотаций и добавляет в метки. Рассмотрим в качества примера находящее в нормальной форме правило вывода для while в модели памяти языка C-kernel:

$$\frac{\begin{array}{l}\ulcorner\{P_1\}\ S\ \{\ulcorner INV\urcorner^{est\_inv\_iter}\}\urcorner^{pres\_inv},\\ (\ulcorner INV\urcorner^{ass\_inv\_exit} \wedge \\ \quad \ulcorner cast(val(val(e, MeM..STD)),\\ \quad\quad type(e, MeM, TP), int) = 0)\urcorner^{while\_ff} \supset Q,\\ \ulcorner(\ulcorner INV\urcorner^{ass\_inv} \wedge \\ \quad \ulcorner cast(val(val(e, MeM..STD)),\\ \quad\quad type(e, MeM, TP), int) \neq 0)\urcorner^{while\_tt} \supset P_1\urcorner^{pres\_inv}\end{array}}{\{\ulcorner INV\urcorner^{est\_inv}\}\ while(e)\ S\ \{Q\}}$$

Рассмотрим структуру данного правила вывода. Чтобы вывести тройку Хоара для цикла while, необходимо использовать индукцию. Ввести индукцию позволяет специальная 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, связанная с ней формула, обозначающая итерацию цикла, и формула, соответствующая выходу из цикла. Слабейшее предусловие включает в себя инвариант, отвечающий за вход в цикл, и, поэтому, имеющий метку est_inv. В посылках обособленные подформулы обеих формул, обозначающих выход из цикла и итерацию цикла, имеют соответствующие метки. Также, вся формула в посылке, обозначающая итерацию цикла, имеет метку для обозначения ее второстепенной цели – способствовать сохранению инварианта. В тройке Хоара в посылке постусловие INV должно иметь метку, обозначающую ее цель (т.е. гарантирование выполнения инварианта после каждой итерации цикла) для передачи информации при рекурсивном вызове. Более того, все УК, выведенные от данной тройки, должны иметь метку pres_inv, обозначающую второстепенную цель такого УК. Это обеспечивается тем, что вся рассматриваемая тройка имеет такую метку. Заметим, как одна и та же формула INV имеет четыре различные роли и, поэтому, имеет четыре различные метки. Эта информация следует из контекста, доступного только при применении правила вывода, и не может быть восстановлена простым способом при анализе полученных УК без использования семантической разметки.

Упрощение имеющих метки УК. УК (имеющие метки или нет) обычно являются довольно сложными и, поэтому, они нуждаются в упрощении перед их проверкой на автоматическом доказателе теорем. Правила для упрощения не имеющих меток УК не могут быть просто переиспользованы для имеющих метки УК по следующим причинам:

1) Семантическая разметка изменяет структуру терма, и, соответственно, применимость правила.

2) Метки нужно аккуратно обрабатывать – с одной строны они не могут быть просто распространены на все операторы без нарушения их области действия. С другой стороны, они не могут быть просто подняты на самый верхний уровень УК, так как это может привести к генерации слишком подробных и неточных объяснений.

Правила упрощения имеющих метки УК должны достигать следующих целей:

1) Удаление лишних меток.

2) Минимизация области действия оставшихся меток.

3) Сохранение достаточного числа меток для объяснения любых неожиданных неудач, основываясь на допущении, что большая часть УК может считаться доказанными.

Такие правила подразделяются на пять различных групп. Будем использовать вспомогательную функцию $|\cdot|$ для удаления меток из термов и оператор композиции меток $\otimes$ для добавления внутреннего списка меток к списку меток, вложенному во внешнюю метку, т.е. $c(o, l) \otimes m = c(o, l \bullet m)$, где $\bullet$ – конкатенация списков. Первая группа содержит правила, такие как $\ulcorner true \urcorner^l \to true$ или $P \supset P' \to true$ (если $|P| = |P'|$), удаляющие метки из тривиальных тождественно истинных (под-)формул, потому что их не нужно объяснять. Следующая группа содержит правила, такие как $\ulcorner false \urcorner^l \vee P \to P$, которые выборочно удаляют имеющие метки тождественно ложные подформулы. В такой ситуации информацию для генерации объяснения предоставляет оставшийся контекст. Правила из следующей группы, заменяющие всю формулу на false, например $\ulcorner false \urcorner^l \wedge P \to \ulcorner false \urcorner^l$, должны сохранять метки, так как иначе не остается контекста для объяснения неудачного доказательства. Правила $\ulcorner P \wedge Q \urcorner^l \to \ulcorner P \urcorner^l \wedge \ulcorner Q \urcorner^l$ и $P \supset \ulcorner Q \supset R \urcorner^l \to P \wedge \ulcorner Q \urcorner^l \supset \ulcorner R \urcorner^l$ входят в состав четвертой группы. Они осуществляют «просачивание» меток через конъюнкцию и (вложенную) импликацию, соответственно, область действия меток минимизирована в полученных после упрощения УК. Последняя группа основывается на знаниях, как интерпретировать метки в предметной области. Эта группа также содержит устраняющее вложенность правило $\ulcorner\ulcorner t \urcorner^n \urcorner^m \to \ulcorner t \urcorner^{n \otimes m}$, которое «просачивает» вложенные метки, и, тем самым, позволяет применять другие имеющие или не имеющие метки правила, сохраняя при этом вложенную структуру самих меток. Это обеспечивает правильную вложенность уточнителей и сохранение их первоначальной принадлежности к определенному терму.

Трансляция. Последней стадией работы с (имеющими метки) УК является генерация объяснений для них, т.е. превращение их в понятный для пользователя текст, называемое *трансляцией*. Структуру и текстовое представление объяснений можно задать как грамматику, в которой правая часть каждого правила представляет собой шаблон для их генерации, похожий на форматную строку из языка Си. Эти шаблоны объяснений позволяют удобно и детально задавать текст, который будет содержаться в генерируемых объяснениях. Для генерации по УК текстового представления транслятор выделяет из данного УК метки и сортирует их по номерам строк, получая при этом список меток. Затем, транслятор сопоставляет этому набору шаблон объяснения, используя правила для генерации текстовых представлений. В трансляторе по умолчанию заданы правила генерации текстовых представлений для часто встречающихся концепций, таких как sub и upd. Трансляция включает в себя четыре шага: (i) Нормализация УК с использованием правил их упрощения; (ii) Извлечение меток после применения правила устранения вложенности; (iii) Нормализация меток для сопоставления меткам шаблонов объяснений; (iv) Генерация текста с использованием шаблонов объяснений. На третьем шаге происходит устранение вложенности в параметрах метки, например метка sub(p, sub(q, sub(r))) превращается в список меток $\langle$sub(p), sub(q), sub(r)$\rangle$.

III. Язык шаблонов

Напомним, что правила в нормальной форме, поступающие на вход метагенератору, задают собой шаблоны для сопоставления с программными

конструкциями. Классическое графическое представление правил вывода удобно для чтения, но мало подходит для ввода их с клавиатуры. Поэтому, 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. В данном разделе рассмотрим случай применения этой схемы для языка C-light.

A. Язык для написания шаблонов

Основой языка шаблонов для C-light является логика первого порядка и грамматика языка Си [8]. Введение метаобозначений в этот язык фактически означает, что в его выражениях сохраняются некоторые нетерминальные символы (например, неинтерпретированные предикатные символы P, Q, R и фрагментные переменные для обозначения фрагментов кода). Принадлежность метаданных тому или иному классу в языке шаблонов задается в явном виде. Опишем далее способы задания этой информации в языке шаблонов для случая языка C-light вместе с ограничениями на них.

1) Любой идентификатор в шаблоне является метаидентификатором. Таким образом, на место любого идентификатора `ID1` в шаблоне может быть подставлен любой идентификатор `ID2` из программной конструкции на языке Си, которая проверяется на соответствие шаблону. При этом идентификатор `ID2` должен удовлетворять тем же самым семантическим ограничениям, что и идентификатор `ID1`. Например, если из контекста шаблона можно сделать однозначный вывод, что идентификатор `ID1` – это идентификатор функции с двумя аргументами, то и идентификатор `ID2` должен быть идентификатором функции с двумя аргументами. Пусть `pattern_identifier` – любой идентификатор шаблона. Кроме того, пусть `program_identifier` – любой идентификатор программной конструкции. Тогда, если при сопоставлении шаблона, содержащего более одного вхождения `pattern_identifier`, этой программной конструкции одному из вхождений `pattern_identifier` был сопоставлен `program_identifier`, то и всем остальным вхождениям `pattern_identifier` должен быть сопоставлен `program_identifier`. Например, если в шаблоне одному из вхождений идентификатора `i` была сопоставлена переменная `j` из программной конструкции, то и всем остальным вхождениям `i` должна быть сопоставлена только переменная `j`.

2) Помимо идентификаторов к фрагментным переменным в шаблонах относятся и более «крупные» конструкции. Для их задания используются ключевые слова `any_code`, `exist_code` и `simple_expression`. Синтаксическая конструкция `any_code` может быть сопоставлена любому, в том числе и пустому, набору программных конструкций, последовательно стоящих в программе. Пусть `construction` – одна 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. Синтаксическая конструкция `exist_code` должна быть сопоставлена одной программной конструкции. Синтаксическая конструкция `simple_expression` должна быть сопоставлена выражению, не включающему вызовы функций и операторов приведения.

3) Для метаданных в спецификациях по аналогии используется явный способ задания.

В качестве примера рассмотрим синтаксическую конструкцию `any_predicate`. Ей должно быть сопоставлено одно утверждение из спецификаций. Запись `any_predicate(construction_identifier)` означает, что у данного вхождения конструкции имеется уникальный идентификатор `construction_identifier`. Когда конструкция `any_predicate` имеет уникальный идентификатор, то этот идентификатор можно использовать в спецификациях из посылки правила вывода.

Таким образом, наличие идентификатора у специальных синтаксических конструкций позволяет расширить класс правил вывода, которые можно написать на рассматриваемом языке.

Важное ограничение на язык шаблонов состоит в следующем: на одном и том же уровне вложенности для шаблона и программного фрагмента, у шаблона должна быть либо одна конструкция `any_code`, либо конструкции `any_code` должны быть только последними на данном уровне. Правила вывода для C-kernel в нормальной форме удовлетворяют этому ограничению, что следует из жестких ограничений, наложенных на данный язык. Это ограничение необходимо для применимости «жадного» алгоритма сопоставления программных конструкций и шаблонов.

B. Пример задания аксиомы и правила вывода

Рассмотрим пример аксиомы для C-kernel из [16]. Пусть выражение e' не включает вызовы функций и операторов приведения. Аксиома для операции присваивания имеет вид:

```
{Q(MD ← upd(MD,
            loc(val(e, MeM..STD)),
            cast(val(val(e', MeM..STD)),
                 type(e', MeM, TP),
                 type(e, MeM, TP)))))}
e = e';
{Q}
```

Данная аксиома находится в нормальной форме [7]. Имена MeM, MD отражают специфику модели памяти языка C-kernel. На рассматриваемом языке для написания аксиом и правил вывода данная аксиома записывается так:

```
{(any_predicate(Q))(MD <- upd(MD,
            loc(val(e, MeM..STD)),
            cast(val(val(e', MeM..STD)),
                 type(e', MeM, TP),
                 type(e, MeM, TP))))}
e = simple_expression(e');
{any_predicate(Q)}
```

Шаблоном в этой аксиоме является конструкция:

```
e = simple_expression(e');
```

Такая запись означает, что программная конструкция, соответствующая данному шаблону, должна представлять собой операцию присваивания, операндами которой служат `e` и `e'`, и `e'` не включает вызовы функций и операторов приведения.

Рассмотрим пример правила вывода для C-kernel. Правило для while в модели памяти языка C-kernel имеет следующий вид:

$$\frac{\{INV \wedge cast(val(val(e, MeM..STD)), type(e, MeM, TP), int) \neq 0\}\ S\ \{INV\}}{\{INV\}\ while(e)\ S\ \{INV \wedge cast(val(val(e, MeM..STD)), type(e, MeM, TP), int) = 0\}}$$

В нормальной форме эквивалент этого правила выглядит так:

```
{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)}
```

Посылки отделяются от заключения знаком "`|-`", который представляет собой вертикальную черту "`|`" и следующий за ней дефис "`-`". Два косые черты (прямая и обратная) "`/\`" обозначают конъюнкцию.

Шаблоном в этом правиле вывода является конструкция:

```
while(simple_expression(e)) any_code(S)
```

Такая запись означает, что программная конструкция, соответствующая данному шаблону, должна представлять собой цикл, у которого есть условие `e` и может присутствовать тело `S`. Отметим, что `e` и `S` являются фрагментными переменными [7].

C. Расширение языка шаблонов семантическими метками

Для локализации ошибок в проекте C-light, как и у Денни и Фишера [2], используются семантические метки. Семантическая метка приписывается терму в условии корректности и имеет вид c(o), где c – тип метки (характеризующий предназначение метки), o – местоположение относящегося к данной метке программного кода (диапазон строк). В связи с введением меток в проект C-light, в качестве аксиоматической семантики языка стали использоваться размеченные правила вывода, отличающиеся от прежних тем, что в них были добавлены метки. Соответственно, язык описания правил вывода был расширен специальной конструкцией `label`, используемой для описания меток. Конструкция `label` имеет вид `(label t c)`, где `t` – терм, к которому приписана метка, а `c` – строка (тип метки). В качестве примера приведем описание на рассматриваемом языке, расширенном метками, находящегося в нормальной форме правила для while:

```
(label {P1} S {(label INV est_inv_iter)}
 pres_inv),
((label INV ass_inv_exit) /\
 (label
   cast(val(val(e, MeM..STD)),
        type(e, MeM, TP), int) = 0
                          while_ff)) => Q,
(label
 ((label INV ass_inv) /\
  (label
    cast(val(val(e, MeM..STD)),
         type(e, MeM, TP), int) != 0
                          while_tt)) => P1
                                 pres_inv)
|-
{(label any_predicate(INV) est_inv}
 while(simple_expression(e)) any_code(S)
{any_predicate(Q)}
```

Конструкция `label` обеспечила переход в проекте C-light от обычных правил Хоара к размеченным.

D. Задание шаблонов объяснений

Для генерации текста объяснения УК в нашем подходе, как и в подходе Денни и Фишера, используются шаблоны объяснений. Они задаются для каждой концепции метки с помощью специальной конструкции `label_pattern`, имеющей вид `(label_pattern label format_text)`, где `label` – концепция метки, а `format_text` – форматная строка, задающая текстовый шаблон. Он представляет собой описывающий концепцию метки текст на естественном языке, дополненный специальными конструкциями `%begin` и `%end`, на место которых будут подставлены соответственно начало и конец диапазона строк, в котором записан относящийся к данной метке программный код.

IV. Реализация метагенератора

Схема работы метагенерации состоит в переработке правил вывода в рекурсивную процедуру генерации условий корректности. В этой процедуре происходит сопоставление программных конструкций и шаблонов. Приведем примеры типов данных, которые используются при реализации этой схемы. У нас есть два ключевых типа данных – программа и шаблон. Входными данными являются шаблоны и программные конструкции, которые описываются с помощью соответствующих программных типов данных. В контексте задачи самоверификации отметим, что все эти типы данных вкладываются в язык Си. Реализация состоит в погружении входных данных в программные типы данных и из функций для работы с ними. Рассмотрим некоторые из них.

A. Структура program_node

Программа, поступающая на вход метагенератору, представляется в виде дерева. Рассмотрим структуру `program_node` [14], которая задает этот тип данных:

```
struct program_node
{
   char* category;
   int has_identifier;
   char identifier[64];
   int has_type;
   char* type;
   int has_value;
   char* value;
   int children_count;
   struct program_node* children[1000];
};
```

Поскольку для нас важной является задача самоверификации метагенератора, в текущей его реализации есть ряд ограничений. В частности, предполагается, что длина идентификаторов в программе не может быть более 64 символов, и у одного узла не может быть более 1000 потомков, при этом используются статические массивы. В противном случае, необходимо использовать функции для работы с динамической памятью, что усложняет верификацию из-за относительно громоздких спецификаций данных функций [9]. Заметим, что принцип модульной верификации и производительность автоматических доказателей теорем позволяют говорить о разумности этих ограничений.

В поле `category` содержится информация о том, какая именно синтаксическая конструкция хранится в данном узле. Например, для условного оператора поле `category` примет значение "`if_statement`", а для цикла while поле `category` примет значение "`while_statement`". Поле `category` заполнено у каждого узла рассматриваемого дерева.

У некоторых синтаксических конструкций, например, у переменных, имеется идентификатор. Поле `has_identifier` принимает значение `1`, если у синтаксической конструкции, представленной в рассматриваемом узле дерева имеется идентификатор, и принимает значение `0` в обратном случае. Соответственно, поле `identifier` содержит идентификатор в случае его наличия.

По аналогии с предыдущим, синтаксическая конструкция может иметь тип и значение. Поэтому, мы используем поля `has_type` и `type`, `has_value` и `value`. Например, у синтаксической конструкции, отображающей вызов функции, типом будет являться тип возвращаемого значения.

Поле `children_count` содержит количество потомков данного узла. А поле `children` является массивом указателей на потомков данного узла.

B. Структура label

В ходе работы метагенератора термам сопоставляются характеризующие их метки. Рассмотрим структуру, задающую этот тип данных:

```
struct label
{
   char* concept;
   int location_begin;
   int location_end;
}
```

В данной структуре поле concept содержит концепцию метки, описывающую предназначение сопоставленного метки терма. Это поле заполняется при синтаксическом анализе аксиом и правил вывода УК. Поля `location_begin` и `location_end` содержат начало и конец диапазона строк, в котором записан относящийся к данной метке программный код. Эти поля заполняет ГУК, так как именно при применении аксиом и правил вывода известна информация о позиции программного кода.

Информация о метках должна содержаться во внутреннем представлении спецификаций. Оно представляет собой дерево, образованное структурой term_node (для краткости изложения не будем приводить определения данной структуры). В ходе работы ГУК терму, имеющему метку, может быть сопоставлена другая метка и т.д. Эти метки хранятся в соответствующем узле дерева спецификаций в виде списка, в котором последним элементом является метка самого верхнего уровня. Данный список представлен в структуре term_node атрибутом labels.

C. Структура pattern_node

Метагенератор производит анализ правил вывода, подаваемых ему на вход, и в результате получает множество шаблонов. Шаблоны представляется в виде деревьев. Рассмотрим структуру pattern_node [14], представляющую такое дерево (для краткости изложения не будем приводить определения данной структуры).

В данной структуре поля category, has_identifier, has_type, has_value, value, children_count, children имеют ту же самую семантику, что и соответствующие поля в структуре program_node.

Пусть для языка написания шаблонов construction – одна из специальных синтаксических конструкций и пусть construction_identifier обозначает уникальный идентификатор. У узла дерева шаблона, соответствующего записи construction(construction_identifier), атрибут has_identifier имеет значение 1, а атрибут identifier имеет значение construction_identifier.

Но у узла дерева шаблона, в отличие от узла дерева программы, поле category может быть не заполнено. То есть, узел шаблона может не хранить информацию о том, какому классу синтаксических конструкций он может быть сопоставлен. Поэтому, в структуре pattern_node присутствует поле has_category, которое принимает значение 1, когда поле category заполнено, и принимает значение 0 в обратном случае.

Также в рассматриваемой структуре присутствует поле is_omitted, принимающее либо значение 1, либо значение 0. Поле is_omitted принимает значение 1 тогда и только тогда, когда узел шаблона может быть сопоставлен любому, в том числе и пустому, набору программных конструкций, последовательно стоящих в программе.

Конструкция any_code представляется в дереве шаблона с помощью узла, у которого атрибут is_omitted имеет значение 1. Также у такого узла атрибуты has_category, has_type, has_value имеют значение 0. Кроме того, у такого узла отсутствуют дочерние узлы и атрибут children_count имеет значение 0.

Конструкция exist_code представляется в дереве шаблонов также, как и другие синтаксические конструкции из шаблона. То есть, конструкции exist_code в дереве шаблонов соответствует узел, у которого могут быть заполнены некоторые атрибуты.

Любая структура pattern_node является корнем дерева, хранящем часть шаблона. Поле is_matched содержит информацию о том, удалось ли сопоставить это дерево какой-либо программной конструкции. Если сопоставить удалось, то поле is_matched принимает значение 1, иначе – значение 0.

Шаблон может быть сопоставлен с программной конструкцией только когда идентификаторы, содержащиеся в нем, сопоставлены с идентификаторами, содержащимися в ней. То есть, если в некоторой части шаблона идентификатор ID1 сопоставлен идентификатору ID2 из программной конструкции, то в других частях шаблона идентификатору ID1 может быть сопоставлен только идентификатор ID2.

Для установления такого соответствия между идентификаторами в структуре pattern_node добавлено поле match_identifiers. Фактически это поле представляет собой два массива строк, расположенных по нулевому и первому индексу первого измерения массива match_identifiers. Массив строк, расположенный по нулевому индексу первого измерения массива match_identifiers, – это массив сопоставленных идентификаторов шаблона. А массив строк, расположенный по первому индексу первого измерения массива match_identifiers, – это массив сопоставленных идентификаторов программной конструкции.

Для удобства назовем массив сопоставленных идентификаторов шаблона нулевым массивом, а массив сопоставленных идентификаторов программной конструкции – первым массивом. Эти два массива имеют одинаковую длину, хранящуюся в поле table_length. Если в нулевом массиве некоторый идентификатор ID1 имеет индекс i, то в первом массиве индекс i должен иметь идентификатор ID2, сопоставленный идентификатору ID1. Таким образом хранится информация о сопоставленных идентификаторах.

D. Анализ шаблонов и программы

Напомним, что для обозначения программного кода в заключении правила вывода используется целевой язык, совпадающий с языком Си, и имеющий некоторые модификации. Так как шаблоны поступают на вход метагенератору, то необходимо производить их лексический, синтаксический и семантический анализ. В дальнейшем под словом анализ будем понимать прохождение всех этих трех видов анализа. Было принято решение использовать уже существующие инструменты для анализа и построения внутреннего представления шаблонов. Данное решение было принято по следующим причинам:

1) Многие из этих инструментов предлагают довольно удобное API для работы с внутренним представлением программ. Отметим, что эти API совершенствовались в течение многих лет.

2) Многие из этих инструментов имеют большие тестовые базы, на которых они проверялись. Было бы сложно создать такие большие тестовые базы только собственными силами.

В качестве такого инструмента для лексического анализа и построения внутреннего представления было выбрано API на языке программирования C++, предоставляемое компилятором Clang и виртуальной машиной LLVM [6]. Этот инструментарий в полной мере обладает всеми перечисленными выше преимуществами [15] и, кроме того:

1) API на языке программирования C++ позволяет использовать возможности объектно-ориентированного анализа и дизайна при разработке анализатора. Использование объектно-ориентированного анализа и дизайна позволяет значительно облегчить разработку практически любых программных систем. Также использование объектно-ориентированного анализа и дизайна при разработке анализатора дает возможность легко вносить изменения в уже реализованный анализатор.

2) API на языке программирования C++, предоставляемое компилятором Clang, позволяет легко работать с внутренним представлением программ. Например, это API предоставляет возможность достаточно легко обойти всю программу, пользуясь ее внутренним представлением.

3) Задача анализа шаблонов предполагает много работы со строковыми данными. Язык программирования C++, на котором предоставляет свой API компилятор Clang, дает возможность удобно работать со строковыми данными по сравнению со многими другими языками программирования (например, по сравнению с языком программирования Си).

4) Сейчас компилятор Clang активно поддерживается, и нет оснований полагать, что его поддержка в скором времени прекратится.

5) Язык программирования C++, на котором предоставляет свой API компилятор Clang, дает возможность снабдить спецификациями фрагменты собственного кода.

При работе метагенератора на стадии анализа происходит считывание конструкций `label_pattern` и формирование внутреннего представления, в котором типу метки соответствует ее текстовый шаблон. Также, на стадии анализа происходит считывание правил вывода и, соответственно, содержащихся в них конструкций `label`, и формирование внутреннего представления правил вывода. Расширение языка семантическими метками привело к созданию анализатора для конструкций `label`, который в ходе работы создает структуры `label` и добавляет указатели на них их к соответствующим узлам `term_node`.

Так как API Clang имеет встроенный лексический и синтаксический анализатор Си-программ, то использование API Clang позволило не заниматься реализацией этой функциональности. На первом этапе, анализатор с помощью инструментария, предоставляемого API Clang, строит внутреннее представление шаблона. Так как специальные синтаксические конструкции в языке шаблонов являются расширениями языка Си, то для их обработки штатными средствами API Clang необходимо вводить специальные функции, одноименные с рассматриваемыми конструкциями. Тогда специальные синтаксические конструкции попадают во внутреннее представление как вызовы этих специальных функций. На втором этапе анализатор с помощью API Clang строит свое внутреннее представление шаблона, которое более удобно для наших задач и которое основывается на структуре pattern_node. При этом вызовы специальных функций интерпретируются как специальные синтаксические конструкции в языке шаблонов. Таким образом, расширение языка Си специальными синтаксическими конструкциями не вызвало сложностей при анализе языка шаблонов. В результате работы анализатора шаблонов получается внутреннее представление шаблонов – множество деревьев, узлами которых является структуры `pattern_node`.

Аналогично производится и анализ программ, поступающих на вход метагенератору. Так как целевым языком нашей системы является язык Си, то вышеперечисленные преимущества API Clang определили выбор инструментария для решения задачи анализа программ. Он проходит в два этапа, аналогичных этапам анализа шаблонов. Более того, данные этапы являются более простыми, чем аналогичные этапы при анализе шаблонов, так как при анализе программ не приходится обрабатывать специальные синтаксические конструкции, хранящие метаинформацию. В результате работы анализатора программы получается ее внутреннее представление – дерево, узлами которого является структуры `program_node`. Итак, использование API Clang позволило снизить трудоемкость разработки анализатора шаблонов и анализатора программы.

E. Функция match_trees

При генерации условий корректности последовательность программных конструкций просматривается от конца к началу. При этом просмотре на каждом шаге выбирается последняя программная конструкция. Функция `match_trees` проверяет, соответствует ли выбранная программная конструкция определенному шаблону или нет [4]. Рассматриваем набор правил вывода и из набора правил выбираем то, которое позволяет провести унификацию программной

конструкции с шаблоном. Происходит перебор правил вывода, и, соответственно, шаблонов.

Функция match_trees вызывается для каждого из них и для выбранной программной конструкции. Это происходит до тех пор, пока не будет найден шаблон, соответствующий выбранной программной конструкции, или пока не будут перебраны все правила вывода, и, соответственно, все шаблоны.

Функция match_trees принимает два аргумента: шаблон и программную конструкцию. Она возвращает 1, если ее аргументы сопоставимы, и 0 в обратном случае. Первым аргументом функция match_trees принимает переменную pattern – указатель на структуру pattern_node. Переменная pattern указывает на корень дерева, в котором хранится шаблон. А вторым аргументом функция match_trees принимает переменную code – указатель на структуру program_node. Переменная code указывает на корень дерева, в котором хранится программная конструкция.

Таким образом, функция match_trees сопоставляет два дерева: дерево шаблона и дерево программной конструкции. Соответственно, функции match_trees передаются указатели на узлы этих деревьев, являющиеся их корнями. Для удобства будем называть узел дерева шаблона, указатель на который передан функции match_trees, узлом pattern, а узел дерева программной конструкции, указатель на который передан функции match_trees, – узлом code. Атрибуты этих узлов хранятся в полях структур pattern_node и program_node соответственно. Корректное сопоставление синтаксических конструкций any_code, exist_code, simple_expression программным конструкциям обеспечивается с помощью алгоритма, реализованного в функции match_trees.

Отметим, что при сопоставлении используется «жадный» алгоритм. Корректность его применения гарантируется введенным выше дополнительным ограничением на язык написания шаблонов, а также простотой аксиоматической семантики языка C-kernel и ограничениями языка C-light (например, запретом на передачу управления в составные операторы извне).

Сначала функция пытается сопоставить все атрибуты узлов program_node и code, кроме потомков этих узлов. Если эти атрибуты удалось сопоставить, то функция пытается сопоставить потомков узла pattern и потомков узла code. При этом происходят рекурсивные вызовы функции match_trees. В таких вызовах в качестве первого аргумента функции передается один из некоторых потомков узла pattern, а в качестве второго – один из некоторых потомков узла code. Если и потомков узлов pattern и code удалось сопоставить, то можно сделать вывод, что шаблон и программная конструкция сопоставимы, и функция возвращает значение 1.

Тогда в узле pattern атрибут is_matched принимает значение 1, а в атрибуте match_identifiers хранится таблица соответствия идентификаторов шаблона идентификаторам программной конструкции. Использование данной таблицы облегчает применение правила вывода, происходящее после успешного сопоставления. Таким образом, заполнение таблицы match_identifiers в ходе работы функции match_trees является преимуществом рассматриваемого подхода.

Отметим, что в реализации функции match_trees нет привязки к конкретному языку, она подходит к широкому классу языков программирования. Таким образом, данный подход соответствует концепции Морикони и Шварца [7], в которой метагенерация также рассматривается безотносительно от используемого языка программирования.

Замечание. Отметим некоторое отличие нашей реализации от исходной идеи Морикони и Шварца. У них порожденный генератор является отдельной программой, которую можно скомпилировать и запустить. Наш метагенератор имеет два входа – правила вывода с аксиомам и программа. Следовательно, по нашей реализации в каждый сеанс верификации происходит порождение генератора по подаваемой на вход системе Хоара. Хотя эта схема не эффективна, но она позволяет нам верифицировать одну общую программу, а не две – метагенератор и порожденный генератор. Тем более, что по Морикони и Шварцу генератор порождается без спецификаций. Поэтому, с точки зрения создания самоприменимой системы верификации [10], наш подход имеет преимущество перед идеей Морикони и Шварца.

F. Реализация перехода от условия корректности к его объяснению

При работе метагенератора на стадии генерации условий корректности происходит применение правил вывода и, соответственно, переход к тройкам Хоара, чьи шаблоны были описаны в посылках правила вывода. При этом, в узлах деревьев спецификаций вместо метасимволов подставляется то, что им соответствует, а в структуру label записывается диапазон строк. Это не создает дополнительных сложностей, так как при применении правила вывода метагенератор знает, какой оператор рассматривается, и, соответственно, он знает и диапазон строк. В итоге, в тройках Хоара не остается программных фрагментов, и тогда предусловие и постусловие объединяются в одну формулу знаком импликации, образуя условие корректности. Таким образом, два дерева спецификаций объединяются в одно, соответствующее условию корректности. Далее, происходит упрощение УК по описанным выше правилам. При этом, в нашем подходе, в отличие от подхода Денни и Фишера, не применяется четвертая группа этих правил, т.е. группа правил, осуществляющих «просачивание» меток через конъюнкцию и (вложенную) импликацию. Таким образом, метки в дереве УК продолжают быть структурой, тоже представляющей собой дерево. Для

обоснования этого можно ввести отношение предок-потомок на метках, соответствующее такому же отношению на дереве спецификаций, не рассматривая при этом узлы дерева спецификаций, не имеющие меток. Следовательно, в нашем подходе, в отличие от подхода Дении и Фишера [2], метки в условиях корректности образуют некоторую иерархию, что используется при генерации текста на естественном языке, объясняющего смысл данного данного условия корректности. Для объяснения неудачи при доказательстве система верификации будет генерировать такой текст для каждого недоказанного УК. На этой стадии дерево меток обходится в глубину, и для каждой рассмотренной метки к общему тексту добавляется текст ее заполненного номерами строк шаблона. Таким образом в проекте C-light работает система локализации ошибок.

V. Заключение

При работе над проектом C-light возникла задача сделать систему верификации удобной не только для специалистов-теоретиков, но и для обычных программистов. Существенным вкладом в решение этой задачи стала бы возможность локализации и объяснения ошибок при наличии недоказанных УК. Иначе, пользователям системы верификации приходилось бы «вручную» анализировать недоказанные УК. Для решения этой задачи был выбран метод Денни и Фишера. Используемый в системе C-light метод метагенерации позволил удобным образом применить в нашем проекте предложенную Денни и Фишером концепцию семантических меток [2]. Согласно этой концепции полученные после генерации УК несут в себе дополнительную информацию, выраженную в виде семантических меток, которая позволяет локализовать и объяснить ошибки.

Обзор родственных работ. Тематика формальной локализации ошибок мало представлена в исследованиях по сравнению с тематикой дедуктивной верификации программ. Рассмотрим три проекта, основанных на формальном подходе к локализации ошибок. Во-первых, в проекте Centaur [3] УК анализируются для поиска условных выражений из исходных условных операторов и циклов. Во-вторых, в работе Денни и Фишера [2] рассмотрен метод семантической разметки, на котором базируется наш подход. В-третьих, в проекте Лейно [5] базовая логика расширена метками, предоставляющими пригодную для объяснения семантическую информацию. Отметим особенности этих проектов. В первом проекте используются некоторые алгоритмы из области отладки программ. Во втором проекте для порождения объяснений используются только метки, а не сами УК. В третьем проекте метки вводятся на этапе трансляции в промежуточный язык, который обрабатывается стандартным безметочным генератором. Также, во всех этих проектах используются более простые, чем Си, входные языки, что является их недостатком по сравнению с нашим проектом.

В будущем планируется применить систему C-light для верификации программ определенных классов, например программ инженерной математики [17]. Метод метагенерации позволяет пользователям системы верификации использовать произвольный набор аксиом и правил вывода, а также произвольный набор семантических меток. Поэтому, интерес представляет использование узкоспециализированных аксиом и правил вывода для программ определенных классов, а также специфических меток для таких аксиом и правил вывода.

Список литературы

[1] Apt K.R., Olderog E.R. Verification of sequential and concurrent programs. – Berlin etc.: Springer, 1991. – 450 p.

[2] Denney E., Fischer B. Explaining Verification Conditions // Proc. AMAST 2008. – LNCS. – 2008. – Vol. 5140. – P. 145-159.

[3] Fraer R. Tracing the origins of verification conditions. – Rocquencourt, 1996. – 17 p. – (Rapp. / INRIA; N 2840).

[4] Kondratyev D. A., Promsky A. V. Towards the 'verified verifier'. Theory and practice // Proc. Fifth Workshop "Program Semantics, Specification and Verification: Theory and Applications". – Moscow, Russia, June 6, 2014. – P. 68-78.

[5] Leino K.R.M., Millstein T., Saxe J.B. Generating error traces from verification condition counterexamples // Science of Computer Programming. – 2005. – Vol. 55, N 1-3. – P. 209-226.

[6] Maryasov I. V., Nepomniaschy V. A., Promsky A. V., Kondratyev D. A. Automatic C Program Verification Based on Mixed Axiomatic Semantics // Proc. Fourth Workshop "Program Semantics, Specification and Verification: Theory and Applications". – Yekaterinburg, Russia, June 24, 2013. – P. 50-59.

[7] Moriconi M., Schwarts R.L. Automatic Construction of Verification Condition Generators From Hoare Logics // Lect. Notes Comput. Sci. – Berlin etc., 1981. – Vol. 115. – P. 363-377.

[8] Programming languages – C. – ISO/IEC 9899: 1999. – 566 p.

[9] Promsky A.V. C Program Verifcation: Verifcation Condition Explanation and Standard Library // Automatic Control and Computer Sciences, 2012, Vol. 46, No. 7, pp. 394-401.

[10] Promsky A.V. Experiments on self-applicability in the C-light verification system // Bull. Nov. Comp. Center, Comp. Science, 35 (2013), 85-99.

[11] Promsky A.V. Experiments on self-applicability in the C-light verification system. Part 2 // Bull. Nov. Comp. Center, Comp. Science, 37 (2014), 93-105.

[12] Promsky A.V. Towards C-light Program Verification: Overcoming the Obstacles // Proc. International Workshop on Program Understanding, 19-23 June, Altai Mountains, Russia, 2009. – pp. 53-63.

[13] Кондратьев Д. А., Промский А. В. Комплексный подход к локализации ошибок при верификации Си-программ // Системная информатика. 2013. № 1. С. 79-96.

[14] Кондратьев Д. А., Промский А. В. Разработка самоприменимой системы верификации. Теория и практика // Моделирование и анализ информационных систем. Т.21, №6 (2014), 70-81.

[15] Марьясов И.В., Непомнящий В.А., Промский А.В., Кондратьев Д.А. Автоматическая верификация С-программ на основе смешанной аксиоматической семантики // Моделирование и анализ информационных систем. Т.20, №6 (2013), 52-63.

[16] Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В. Ориентированный на верификацию язык C-light // Системная информатика. – 2004. – Вып. 9. – С. 51-134.

[17] Непомнящий В.А., Рякин О.М. Прикладные методы верификации программ. – М.: Радио и связь, 1988.