Модел. и анализ информ. систем. Т. **21**, № **6** (2014) 71–82

УДК 519.681

Разработка самоприменимой системы верификации. Теория и практика

Кондратьев Д.А.[1], Промский А.В.[2]

[1]*Новосибирский государственный университет*
630090, Россия, г. Новосибирск, ул. Пирогова, 2
[2]*Институт систем информатики имени А.П. Ершова СО РАН*
630090, Россия, г. Новосибирск, пр. Лаврентьева, 6

e-mail: apple-66@mail.ru, promsky@iis.nsk.su

получена 13 сентября 2014

Ключевые слова: верификация, спецификация, аксиоматическая семантика, язык C-light, условие корректности, метагенерация

По сравнению с традиционным тестированием дедуктивная верификация предлагает более формальный способ доказательства корректности программ. Но как установить корректность самой системы верификации? Теоретические основы логик Хоара исследовались в классических работах, где были получены различные результаты по непротиворечивости и полноте. Однако нам не известны реализации этих теоретических методов, проверенные чем-либо отличным от обычного тестирования. Иными словами, нас интересует система верификации, которая может быть применена к самой себе (хотя бы частично). В наших исследованиях мы обратились к методу *метагенерации*, который выглядит многообещающим для этой задачи.

1. Введение

Ответ на вопрос о надежности системы верификации может состоять из двух частей:

1. Так как методы верификации основаны на математических концепциях (множества, отношения, исчисления и т.д.), их свойства можно формально доказывать. Например, доказательство непротиворечивости аксиоматической семантики — это вполне традиционное упражнение [1]. Однако такие доказательства, как правило, проводятся вручную. Автоматические доказатели теорем могли бы значительно повысить надежность этих результатов. Примеров их применения гораздо меньше, хотя некоторыми исследователями получены интересные результаты [7, 8].

2. Программные реализации этих теоретических методов также следует проверять. В дополнение к обычному тестированию формальная верификация была бы полезной. В частности, если система реализована на исследуемом языке, то ее самоверификация стала бы наилучшей проверкой. Что касается языка Си, нам неизвестна подобная самоприменимая система.

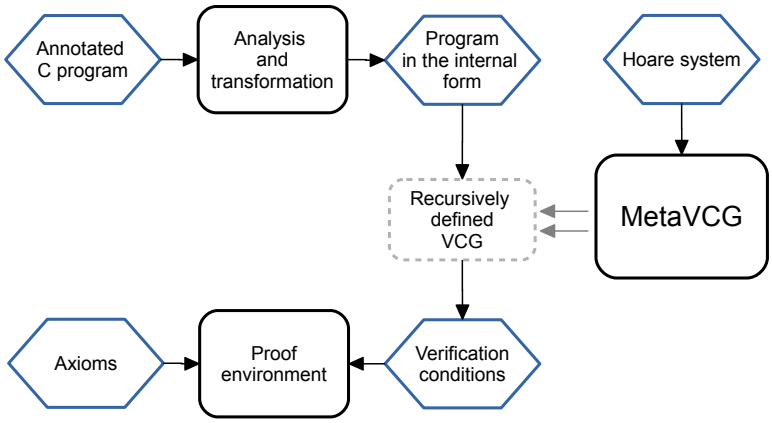

Рис. 1. «Мета-этап» в процессе верификации

В лаборатории теоретического программирования ИСИ СО РАН разрабатывается системы C-light [4]. Язык C-light покрывает б**о**льшую часть предыдущего стандарта (C99). Чтобы избежать проблем логики Хоара для полного Си, мы транслируем входные программы в небольшое ядро — язык C-kernel. Условия корректности (УК), порождаемые генератором для C-kernel, передаются для доказательства системе Simplify. Помня о важности корректности системы, мы формально установили соответствующие свойства для всех этапов нашего подхода [6].

На пути к самоприменимой системе мы уже предприняли некоторые шаги. Во-первых, спецификации на языке ACSL были разработаны для части стандартной библиотеки [9]. Любая осмысленная Си-программа использует библиотеку, поэтому эти аннотации играют важную роль. Во-вторых, некоторые фрагменты входного анализатора/транслятора нашей системы были верифицированы [10]. Хотя он реализован на C++ с помощью API компилятора Clang, значительная часть его функционала вкладывается в C-light, что делает его хорошим тестовым примером.

В данный момент мы исследуем этап генерации УК. Можно было бы применить тот же лобовой способ, что и для транслятора. Однако есть альтернативный подход, обладающий рядом достоинств.

В 1981 г. Морикони и Шварц [5] предложили *метагенератор условий корректности*. Получая на вход логику Хоара, он автоматически порождает рекурсивный алгоритм генерации УК. Аксиоматические правила в *нормальной* форме должны удовлетворять некоторым ограничениям. Многие известные аксиоматические правила им не удовлетворяют, поэтому был предложен алгоритм эквивалентной трансляции из менее ограниченной *общей* формы в нормальную. Для метода были установлены полнота и непротиворечивость, что гарантирует *корректность* порождаемого генератора относительно исходной аксиоматической семантики.

При наличии стадии метагенерации обычная трехблочная схема системы верификации приобретает модифицированный вид (см. рис. 1).

Помимо теоретической корректности этого метода есть и ряд преимуществ на практике. Во-первых, появляется возможность автоматизированного порождения расширенных или специализированных версий генераторов УК. Цель расширения

вполне очевидна: пополнение языка C-light остающимися конструкциями стандартного Си или переход к родственным языкам (Objective C, C++). Под специализацией мы понимаем добавление контекстных аксиоматических правил, которые работают только для некоторых классов программ, но при этом упрощают верификацию. Примером здесь может служить разработанная в лаборатории система вывода для программ линейной алгебры. Правила этой системы способны заменять некоторые вложенные циклы (характерные для обработки матриц) на логические конструкции, работать с которыми проще, чем с обычными инвариантами циклов. В качестве более конкретного примера рассмотрим фрагмент

$$swap(x, y, buf) \equiv \texttt{memcpy(buf, x, m); memcpy(x, y, m); memcpy(y, buf, m);}$$

который встречается в функциях стандартной библиотеки. При обработке его по общим правилам произойдет трехкратная подстановка спецификаций для функции `memcpy` с конкретизацией параметров, что приведет к громоздкому УК с кванторами. С другой стороны, мы могли бы добавить к логике Хоара следующую аксиому:

$$\{x = x_0 \wedge y = y_0\}\ swap(x, y, buf)\ \{x = y_0 \wedge y = x_0\}.$$

Хотя она работает только при наличии в программе фрагмента $swap(..)$, ее применение может радикально упростить доказательство. При этом подход метагенерации избавляет нас от ручного кодирования этой аксиомы в генераторе.

Другое достоинство этого подхода состоит в том, что он позволил абстрагироваться от API Clang и реализовать метагенератор на языке C-light. Это позволяет более полно проводить его верификацию, что и является основной целью.

2. Метагенерация условий корректности

Подход метагенерации, предложенный в [5], состоит из двух шагов. Вначале аксиоматическая семантика в общей форме преобразуется в нормальную форму, по которой, в свою очередь, строится рекурсивная процедура генерации УК. В данном разделе мы сосредоточимся в основном на описании нормальной формы, поскольку общая более приближена к классическим примерам.

2.1. Предварительные определения

По аналогии с [5] мы используем метапеременные $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{R}$, Γ, ... для обозначения частично-интерпретированных формул первого порядка. Они могут содержать неинтерпретированные предикатные символы P, Q, R, ... и формулы конкретной предметной логики. Например, $\mathcal{P}$ может обозначать P, $P \wedge x = 5$ или $x = 5$. Вместо символов P, Q, R,... могут быть подставлены формулы предметной логики.

Также вводится бинарное отношение $\Leftarrow$ на неинтерпретированных предикатных символах. Для тройки Хоара вида

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

где символы $P_1, ..., P_m$ и $Q_1, ..., Q_n$ логически свободны в $\mathcal{P}$ и $\mathcal{Q}$ соответственно, мы имеем

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

Интуитивно запись $P \Leftarrow Q$ означает, что связывание символа P зависит от связывания Q. Отношение $\Leftarrow$ определяется по всему множеству троек Хоара. Запись $\stackrel{+}{\Leftarrow}$ обозначает транзитивное замыкание $\Leftarrow$.

Аналогично для правила вида

$$\frac{\{P_1\}\ S_1\ \{\mathcal{Q}_1\}, ..., \{P_n\}\ S_n\ \{\mathcal{Q}_n\}, \Gamma}{\{\mathcal{P}\}\ S\ \{Q\}}$$

отношение $\ll$ обозначает зависимость доказательства для S от доказательств для $S_1, ..., S_n$. В частности $S \ll S_i$ для $i \in \{1, ..., n\}$. По всей системе Хоара можно определить транзитивное замыкание $\ll +$.

Также используем функцию $FreePreds$ для обозначения множества *логически свободных* предикатных символов в формуле. $FreePreds$, примененная к тройке Хоара $\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}$, обозначает объединение $FreePreds(\mathcal{P}) \cup FreePreds(\mathcal{Q})$, а для правила вывода это будет объединение множеств предикатных символов, полученных для каждой посылки и заключения.

Функция $FragVars$ обозначает множество *фрагментных переменных* во фрагменте S из тройки Хоара $\{P\}\ S\ \{Q\}$. Например,

$$FragVars(\texttt{if}\ (B)\ S_1\ \texttt{else}\ S_2) = \{B, S_1, S_2\}.$$

Будучи примененной к правилу вывода, $FragVars$ выдаст множество, содержащее фрагментные переменные из каждой тройки Хоара в правиле.

Наконец, определим понятие связанного вхождения неинтерпретированного предикатного символа в правиле вывода. Для правила R предикатный символ из множества $FreePreds(R)$ называется *связанным в* R, если он принадлежит $FragVars(R)$. Иначе его вхождение считается *свободным в* R.

2.2. Нормальная форма правил

Рассмотрим следующее

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

$$\frac{\{P_1\}\ S_1\ \{\mathcal{Q}_1\}\ , ...,\ \{P_n\}\ S_n\ \{\mathcal{Q}_n\},\ \Gamma}{\{\mathcal{P}\}\ S\ \{Q\}},$$

которое удовлетворяет следующим ограничениям:

1. P_1,..., P_n и Q — это предикатные символы, свободные в N.

2. Γ — это формула предметной логики, свободные предикатные символы которой могут включать только символы из $FreePreds(N)$ или $FragVars(S)$.

3. $\cup_{1 \le i \le n} FragVars(S_i) \subseteq FragVars(S)$.

4. *Порядок зависимостей.* Тройки Хоара в посылках правила N должны удовлетворять двум ограничениям на зависимости:

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

b. $T \stackrel{+}{\Leftarrow} U \wedge \neg(\exists R)U \stackrel{+}{\Leftarrow} R\ \supset\ U \equiv Q \vee U$ bound in N

5. *Монотонность*. Пусть $\mathcal{P}[P \leftarrow \mathbf{false}, P \in s]$ обозначает $\mathcal{P}$ с подстановкой **false** вместо каждого предикатного символа P из множества s. Тогда формула $\mathcal{P}$ должна удовлетворять следующему:

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

Это же ограничение должно выполняться для Γ и каждой $\mathcal{Q}_i$. □

Для аксиом это определение вырождается до тройки вида $\{\mathcal{P}(Q)\}\ S\ \{Q\}$, где постусловие Q — это единственный предикатный символ, который может быть свободным в аксиоме, а также выполнено ограничение: $\mathcal{P}[Q \leftarrow \mathbf{true}] \ \vee \ \neg\mathcal{P}[Q \leftarrow \mathbf{false}]$.

При этом на всю систему правил в нормальной форме вводятся два дополнительных ограничения: (i) Любая терминальная строка σ языка программирования может быть не более чем одним языковым фрагментом S, определяемым аксиомой или правилом в нормальной форме. (ii) Отношение $\ll +$ должно быть иррефлексивным (это необходимо для завершимости процесса генерации УК).

Ограничение 4 гарантирует, что генератор УК будет способен вычислять вхождения всех свободных неинтерпретированных предикатных символов в правилах. В частности, 4a фактически вводит следующий порядок на предикатных символах:

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

Тем самым удаляются циклические зависимости, такие как посылка вида $\{P\}$ ... $\{P\}$ или пара посылок $\{P\}$ ... $\{R\}$ и $\{R\}$ ... $\{P\}$. При таком порядке 4b гарантирует, что конец любой цепочки зависимостей либо выражается функцией от постусловия Q, либо связан в программном фрагменте.

Ограничение 5 необходимо для полноты порождаемого генератора УК, т.е. гарантирует, что генератор способен вычислять слабейшее предусловие $wp(S, Q)$ для заданных S и Q. Требование монотонности отсеивает правила, в которых есть некоторые «смены знака» между предусловиями посылок и предусловием заключения.

2.3. Общая форма правил и ее перевод в нормальную

Ограничения нормальной формы правил служат двум целям. Во-первых, по правилам можно автоматически построить рекурсивный генератор УК. Действительно, поскольку предусловия в посылках — это одиночные предикатные символы, то вместо них можно подставить слабейшие предусловия для соответствующих программных фрагментов и постусловий. В общем случае для правила вида

$$\frac{\{P_1\}\ S_1\ \{\mathcal{Q}_1\}, ..., \{P_n\}\ S_n\ \{\mathcal{Q}_n\}, \Gamma}{\{\mathcal{P}\}\ S\ \{Q\}}$$

рекурсивная функция wp определяется как

$$wp(S, Q) = \mathcal{P}[P_1 \leftarrow wp(S_1, \mathcal{Q}_1), ..., P_n \leftarrow wp(S_n, \mathcal{Q}_n)] \wedge \\ (\forall \overline{v})\Gamma[P_1 \leftarrow wp(S_1, \mathcal{Q}_1), ..., P_n \leftarrow wp(S_n, \mathcal{Q}_n)],$$

где $[P_1 \leftarrow t_1, ..., P_n \leftarrow t_n]$ обозначает n подстановок, проводимых последовательно слева направо, а $\overline{v}$ — это множество свободных логических переменных в Γ.

Во-вторых, ограничения и определение функции *wp* позволяют доказать непротиворечивость и полноту порождаемого генератора как системы ввода, относительно исходной системы в нормальной форме [5]. Заметим, что это не означает непротиворечивости и полноты нормальной системы относительно операционной семантики языка. Эти результаты устанавливаются отдельно. В частности, доказательство непротиворечивости аксиоматической семантики языка C-kernel можно найти в [6].

С другой стороны, ограничения нормальной формы сильно сужают класс допустимых логик Хоара, да и сами правила выглядят довольно непривычно. Поэтому Морикони и Шварц предложили менее ограниченную *общую форму* правил и алгоритм перевода из общей формы в нормальную. Объем статьи позволяет рассмотреть их лишь кратко. Так, из ограничений нормальной формы сохраняются только 1–3 и 4b (плюс модифицированное свойство монотонности). Тем самым устраняется неудобный порядок на посылках. Далее, предусловия в посылках становятся более разнообразными: помимо одиночных предикатных символов допускаются формулы предметной логики или конъюнкции первых двух вариантов. Алгоритм трансляции состоит в собирании предусловий, отличных от одиночных символов, в виде конъюнкций с удалением дубликатов, а также в переупорядочивании посылок для выполнения ограничения 4a.

Для иллюстрации приведем правила для операторов `if` и `while` в общей (слева) и эквивалентной нормальной форме:

$$\frac{\{P \wedge B\}\ S_1\ \{Q\}, \quad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \texttt{if}\ (B)\ S_1\ \texttt{else}\ S_2\ \{Q\}} \qquad \frac{\{P_1\}\ S_1\ \{Q\}, \quad \{P_2\}\ S_2\ \{Q\}}{\{B \supset P_1 \wedge \neg B \supset P_2\}\ \texttt{if}\ (B)\ S_1\ \texttt{else}\ S_2\ \{Q\}}$$

$$\frac{\{P \wedge B\}\ S\ \{P\}, \quad P \wedge \neg B \supset Q}{\{P\}\ \texttt{while}\ (B)\ S\ \{Q\}} \qquad \frac{\{P_1\}\ S\ \{P\}, \quad P \wedge \neg B \supset Q, \quad P \wedge B \supset P_1}{\{P\}\ \texttt{while}\ (B)\ S\ \{Q\}}$$

2.4. Язык шаблонов

Генератор УК, построенный по системе Хоара в нормальной форме, пытается сопоставить свободные предикатные символы и фрагментные переменные конкретным аннотациям и программным конструкциям. Для записи аксиом и правил в общей форме, подаваемых на вход метагенератору, предлагается язык шаблонов.

Заметим, что классический способ описания логики Хоара (как в разд. 2.1.) хорош для понимания теории, но не особо гибок на практике. Поэтому мы не оговариваем жестко, что символы P, Q, R выражают предикаты, а S_i — это операторы. Можно использовать любые символы, а принадлежность тому или иному классу выражается в виде явной синтаксической обертки. Например, конструкция `any_code(S)` может быть сопоставлена любой последовательности (включая пустую) элементов языка программирования, тогда как `exist_code(S)` соответствует единственной конструкции. Конструкция `simple_expression` описывает любое выражение, не содержащее вызовов функций или приведений типов.

В качестве иллюстрации использования языка шаблонов приведем аксиому для присваивания в языке C-kernel

```
{(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)}
```

и правило для оператора `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_exp(e)) any_code(S) {any_predicate(Q)}
```

Имена MD, MeM, STD относятся к модели памяти языка C-light и не влияют на структуру троек Хоара, поэтому позвольте нам не останавливаться подробно на их описании. Детали можно найти в [4].

3. Реализация и верификация метагенератора

Следует отметить основное отличие нашей текущей стратегии от исходной идеи Морикони и Шварца. Наш метагенератор — это функция от двух аргументов и в процессе ее работы происходит переход к однопараметрической функции. Так, если H — это система Хоара и AP — аннотированная программа, которую нужно верифицировать, то

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

где VCG_H — собственно генератор УК, динамически построенный для H. Это не самое лучшее решение с точки зрения эффективности, так как при каждом сеансе верификации (аргумент AP) мы заново строим генератор, даже если система Хоара остается той же (например семантика языка C-kernel). С другой стороны, это позволяет нам верифицировать одну программу (метагенератор) вместо двух, одна из которых появляется без каких-либо спецификаций. На текущем этапе теоретических исследований такой подход нас устраивает. В будущем мы можем использовать генератор как отдельное компилируемое приложение.

Также отметим, что мы не ограничиваем наш метагенератор только стратегией *слабейшего предусловия*, как Морикони и Шварц. Исчисление *сильнейшего постусловия* также может быть применено.

3.1. Реализация метагенератора

Аргументы метагенератора — система Хоара и аннотированная программа преобразуются во внутреннее представление. Напомним, что для синтаксического анализа используется API компилятора Clang, поэтому преобразование включает перевод в структуры, совместимые с C-light.

В качестве примера рассмотрим тип данных `pattern_node`, в котором представляются аксиомы и заключения правил Хоара.

```
struct pattern_node
{
    int is_omitted;

    int has_category;
    char* category;

    int has_identifier;
    char identifier[64];

    int has_type;
    char* type;

    int has_value;
    char* value;

    int is_matched;
    int table_length;
    char match_identifiers[2][1000][64];

    int children_count;
    struct pattern_node* children[1000];
};
```

Так как мы имеем дело с аксиоматической семантикой, то очевидно, что первая и последняя вершины в списке потомков — это пред- и постусловие соответственно. Каждая вершина-шаблон имеет атрибуты (категория, идентификатор, тип), которые служат в процессе сопоставления. Также есть таблица соответствия программных и шаблонных имен, заполняемая непосредственно в процессе сопоставления. Структура `program_node`, которая служит для представления дерева программы в метагенераторе, в целом аналогична типу `pattern_node`.

Таким образом метагенератор строит дерево аннотированной программы и набор шаблонов для логики Хоара. В соответствии с направлением вывода он выбирает первую/последнюю программную конструкцию и пытается найти подходящий шаблон. Для найденного шаблона он рекурсивно применяет посылки соответствующего правила[1] Хоара.

Объем статьи не позволяет привести основной алгоритм метагенератора, поэтому в следующем разделе ограничимся двумя вспомогательными функциями. Отметим только, что сейчас при сопоставлении используется «жадный» алгоритм. Корректность его применения гарантируется простотой аксиоматической семантики языка C-kernel[2] и ограничениями языка C-light (например запрет на передачу управления в составные операторы извне). При переходе к полному Си или к C++, возможно, будут применяться более общие подходы.

[1]Введение направления вывода подразумевает преобразование аксиом в эквивалентные правила.

[2]Собственно ради этого и введен этап трансляции из C-light.

3.2. Примеры верифицированных фрагментов

При сопоставлении вершины дерева шаблонов с вершиной дерева программы мы также сравниваем атрибуты вершин, включая их идентификаторы. Для краткости будем обозначать текущие вершины деревьев как `pattern` и `code`. Если обе вершины имеют идентификаторы, то мы должны проверить, был ли идентификатор `pattern` ранее сопоставлен идентификатору какой-либо программной конструкции. Для проверки сканируется таблица `match_identifiers`. При неудаче идентификатор `pattern` связывается с идентификатором `code` и соответствующая запись добавляется в таблицу `match_identifiers`, хранящуюся в `pattern`. Для этого используется функция `add_identifier`, аннотированное определение которой выглядит как

```
/*@ requires \valid(pattern) && \valid(code);
    assigns pattern->table_length;
    assigns pattern->match_identifiers[0..1]
                    [\old(pattern->table_length)]
                    [0..\max(strlen(pattern_identifier), 63)];
    ensures strncmp(pattern->match_identifiers[0]
                                    [pattern->table_length],
                    pattern->identifier, 63);
    ensures strncmp(pattern->match_identifiers[1]
                                    [pattern->table_length],
                    pattern->identifier, 63);
    ensures pattern->table_length = \old(pattern->table_length)+1;
*/
void add_identifier(struct pattern_node* pattern,
                    struct program_node* code)
{
    strncpy(pattern->match_identifiers[0][pattern->table_length],
            pattern->identifier, 63);
    strncpy(pattern->match_identifiers[1][pattern->table_length],
            code->identifier, 63);
    pattern->table_length++;
}
```

Другой пример относится непосредственно к процессу сопоставления деревьев. В частности, необходимо сопоставлять такой атрибут, как `category`. Для этого используется функция `compare_categories`. Если поля `category` текущих вершин деревьев совпадают, то возвращается значение 1, иначе 0.

```
/*@ requires \valid(pattern) && \valid(code) &&
             pattern->has_category == 1;
    behavior comparable:
        assumes strlen(pattern_category) == strlen(code_category)
                && \forall int i; 0 <= i <= strlen(code_category)
                        ==>
                        pattern_category[i] == code_category[i];
        ensures \result == 1;

    behavior incomparable:
        assumes \exist int i;
```

```
                        0 <= i <= \min(strlen(pattern_category),
                                       strlen(code_category)) &&
                        pattern_category[i] != code_category[i];
        ensures \result == 0;
*/
int compare_categories(struct pattern_node* pattern,
                        struct program_node* code)
{
    int result = 1;

    if ((pattern->has_category) &&
        (strcmp(pattern->category, code->category) != 0))
    {
        result = 0;
    }
    return result;
}
```

В отличие от «программных» вершин, вершина-шаблон может опускать информацию о синтаксических конструкциях, которым она может быть сопоставлена, т.е. поле `category` может быть пустым. Это обычно происходит, когда шаблон может соответствовать любой последовательности (включая пустую) программных конструкций. Однако такая ситуация обрабатывается где-то на внешнем уровне, поэтому неявно предполагается, что `pattern->has_category == 1` при вызове функции `compare_categories`.

Используя спецификации для строковых функций `strlen`, `strncpy` и `strcmp` из предыдущих исследований [9], эти функции были верифицированы с помощью доказателя Simplify.

4. Заключение

Дедуктивная верификация призвана формально устанавливать корректность программ. Очевидно, что сам метод верификации при этом обязан быть корректным. Помимо теоретической непротиворечивости проверке подлежит и его воплощение в коде. Если система написана на целевом языке, то появляется шанс верифицировать ее своими же силами. Особый интерес эта задача представляет для такого популярного языка, как Си.

В этой работе мы описали некоторые шаги на пути к такому «верифицированному верификатору». Прежде всего модифицированный вариант метода метагенерации был реализован на языке C-light. Затем были разработаны его ASCL-спецификации, которые основаны на библиотечных спецификациях из наших предыдущих работ. Наконец была проведена серия экспериментов по верификации метагенератора.

Заметим, что попытка сравнения с аналогичными отечественными или зарубежными работами вызывает определенные сложности. Зачастую разработчики систем используют для реализации совершенно другие языки (например функциональный язык O'Caml в известной системе WHY [3]). Либо же исследователи концентрируют

свои усилия на верификации других приложений (например, гипервизор Hyper-V является основным объектом исследования команды из проекта VCC [2]).

Мы планируем расширять работу по специфицированию и верификации компонентов нашей системы. В настоящий момент только ограниченный функционал выразим на языке Си. Возможно, потребуется отказ от использования C++ и API компилятора Clang в пользу языка Си для достижения главной цели — полной верификации.

Во введении было упомянуто и другое возможное направление будущих исследований. Формальные семантики языков C-light и C-kernel можно представить в некотором автоматическом/интерактивном доказателе на основе логик высшего порядка. Это позволит перепроверить теоремы об их свойствах на новом уровне.

Список литературы

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

2. 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 // Proc. TPHOLs. 2009. LNCS. 2009. Vol. 5674. P. 23–42.

3. Filliâtre J.C., Marché C. Multi-prover verification of C programs // Proc. ICFEM 2004. LNCS. 2004. Vol. 3308. P. 15–29.

4. Maryasov I.V., Nepomnyaschy 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.

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

6. Непомнящий В.А., Анureев И.С., Михайлов И.Н., Промский А.В. Ориентированный на верификацию язык C-light // Системная информатика: Сб. науч. тр. Новосибирск: Издательство СО РАН, 2004. Вып. 9: Формальные методы и модели информатики. С. 51–134. [Nepomnyaschy V.A., Anureev I.S., Mikhaylov I.N., Promsky A.V. Orientirovannyy na verifikatsiyu yazyk C-light // Sistemnaya informatika: Sb. nauch. tr. Novosibirsk: Izdatel'stvo SO RAN, 2004. Vyp. 9: Formal'nye metody i modeli informatiki. S. 51–134 (in Russion)].

7. Norrish M. C formalised in HOL: Thes. doct. phylosophy (computer sci.). Cambridge, 1998. 150 p.

8. Oheimb D. von. Hoare logic for Java in Isabelle/HOL // Concurrency and Computation: Practice and Experience. 2001. 13(13).

9. Promsky A.V. C Program Verification: Verification Condition Explanation and Standard Library // Automatic Control and Computer Sciences. 2012. Vol. 46, No. 7. P. 394–401.

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

Towards the 'Verified Verifier'. Theory and Practice

Kondratyev D. A.[1], Promsky A. V.[2]

[1]*Novosibirsk State University, ul. Pirogova, 2, Novosibirsk, 630090, Russia*
[2]*A.P. Ershov Institute of Informatics Systems SB RAS, Acad. Lavrentjev pr., 6, Novosibirsk, 630090, Russia*

Keywords: verification, specification, axiomatic semantics, the C-light language, verification condition, MetaVCG

As opposed to traditional testing, the deductive verification represents a formal way to examine the program correctness. But what about the correctness of the verification system itself? The theoretical foundations of Hoare's logic were examined in classical works, and some soundness/completeness theorems are well-known. However, we practically are not aware of implementations of those theoretical methods which were subjected to anything more than testing. In other words, our ultimate goal is a verification system which can be self-applicable (at least partially). In our recent studies we addressed ourselves to the *metageneration* approach in order to make such a task more feasible.

Сведения об авторах:
Кондратьев Дмитрий Александрович
Новосибирский государственный университет, студент;

Промский Алексей Владимирович,
Институт систем информатики имени А.П. Ершова СО РАН, ст. науч. сотр.