Модел. и анализ информ. систем. Т. **20**, № **6** (2013) 52–63

УДК 519.681.3

Автоматическая верификация C-программ на основе смешанной аксиоматической семантики[1]

Марьясов И.В., Непомнящий В.А., Промский А.В., Кондратьев Д.А.

Институт систем информатики им. А. П. Ершова СО РАН
630090, г. Новосибирск, пр. Лаврентьева, 6
Новосибирский государственный университет
630090, г. Новосибирск, ул. Пирогова, д. 2

e-mail: {ivm, vnep, promsky}@iis.nsk.su, apple-66@mail.ru

получена 10 ноября 2013

Ключевые слова: верификация, операционная семантика, аксиоматическая семантика, язык C, язык C-light, язык C-kernel, частичная корректность, ACSL, LLVM, Simplify

Развитие проекта C-light привело к применению новых формализмов и реализации методов, которые облегчают процесс верификации C-программ. Смешанная аксиоматическая семантика предлагает выбор между упрощенными и общими правилами вывода условий корректности (УК) в зависимости от программных объектов и их свойств. Инфраструктура LLVM значительно упрощает реализацию анализатора и транслятора C-light программ. Метод семантических меток, предложенный ранее, теперь может быть безопасно использован в условиях корректности при их доказательстве. Рассмотрен пример из соревнования по верификации, иллюстрирующий применение нашей системы.

1. Введение

В настоящее время верификация C-программ является актуальной проблемой, поскольку язык C широко применяется в системном программировании. Рассмотрим два проекта по верификации C-программ, которые идеологически сходны с нашим.

Один подход был предложен в рамках проекта WHY [10] института INRIA. В действительности WHY — это платформа, подходящая для верификации многих императивных языков. Определен промежуточный язык под тем же названием WHY, в который транслируются входные программы. Эта трансляция нацелена на генерацию условий корректности (УК), независимых от доказателей теорем. Платформа WHY служит основой системы Frama-C, которая предоставляет статический анализ и дедуктивную верификацию.

[1]Эта работа частично поддержана грантом РФФИ 11-01-00028-а.

В Microsoft Research [8] разрабатывается проект VCC. Программы транслируются в логические формулы с помощью инструмента Boogie, который сочетает в себе промежуточный язык Boogie PL и генератор условий корректности. Эти условия проверяются SMT-решателем Z3. Boogie PL не ограничивается поддержкой только языка C. Например, он также используется в проекте Spec#. Однако трансляция в другой язык является определенным недостатком подхода, так как не представлено доказательство корректности такой трансляции (это же справедливо и для проекта WHY).

C-light является выразительным подмножеством языка C. Для верификации C-light программ ранее был предложен двухуровневый подход [3] и метод смешанной аксиоматической семантики [4].

На первой стадии мы транслируем исходную C-light программу в программу на языке C-kernel. Язык C-kernel является подмножеством C-light. На второй стадии генерируются условия корректности по правилам смешанной аксиоматической семантики.

Наш двухуровневый подход к верификации C-программ имеет теоретическое обоснование. Были доказаны теоремы о корректности трансляции из C-light в C-kernel и о корректности смешанной аксиоматической семантики C-kernel [4]. Другим преимуществом метода является полная операционная и аксиоматическая формализация языка C-light. С одной стороны, это позволяет нам выражать различные свойства (например, разделение памяти). С другой стороны, это приводит к громоздким условиям корректности. Для преодоления этой проблемы используется метод смешанной аксиоматической семантики [4], который является комбинацией двухуровневого метода верификации C-программ и смешанной аксиоматической семантики языка C-kernel. Слово «смешанная» означает, что имеются несколько правил вывода для одной и той же программной конструкции, которые однозначно применяются в зависимости от контекста. Во многих случаях использование специальных правил вывода позволяет упростить условия корректности.

В [15] была описана схема расширяемой мультиязыковой системы анализа и верификации СПЕКТР. В настоящей работе мы представляем систему, в которой реализованы наши методы в контексте верификации C программ.

Диаграмма на рис. 1 иллюстрирует схему системы верификации C-light программ.

На стадии доказательства используется автоматический доказатель теорем Simplify [9]. В случае, когда доказатель не смог проверить истинность условия корректности, пользователь может подавать дополнительные аксиомы. Если все условия корректности доказаны, то программа частично корректна. В противном случае пользователь должен исправить программу или ее спецификацию и повторить процесс верификации в системе.

2. Трансляция из C-light в C-kernel

В ходе развития проекта был выбран новый подход для реализации транслятора из C-light в C-kernel, подробно описанный в [2]. Было принято решение использовать уже существующие инструменты для лексического анализа и построения внутреннего представления аннотированных C-light программ. В качестве такого

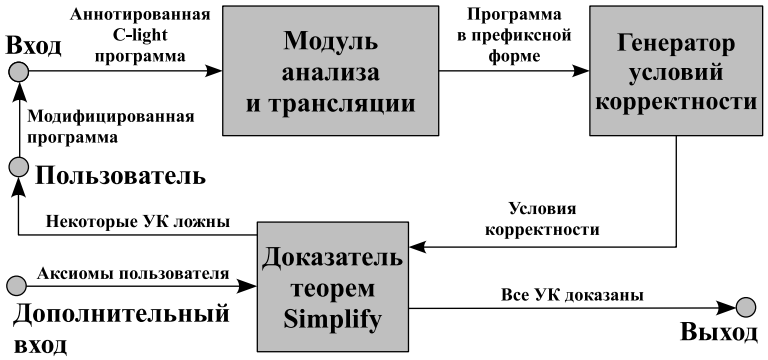

Рис. 1. Система верификации C-light

инструмента было выбрано API на языке программирования C++, предоставляемое компилятором Clang и виртуальной машиной LLVM. Этот инструментарий обладает следующими преимуществами:

1. API позволяет использовать возможности объектно-ориентированного анализа и дизайна при разработке транслятора. Это позволяет облегчить разработку и дает возможность довольно легко вносить изменения в уже реализованный транслятор. Задача внесения изменений является актуальной в связи с работой по расширению языка C-light [3].

2. API позволяет использовать возможности объектно-ориентированного анализа и дизайна при разработке транслятора. Это позволяет облегчить разработку и дает возможность довольно легко вносить изменения в уже реализованный транслятор. Задача внесения изменений является актуальной в связи с работой по расширению языка C-light [3].

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

При рассмотрении текста программы, являющейся реализацией транслятора, важно рассмотреть классы из API Clang, объекты которых использовались в этой программе, а также классы, на которых строится реализация правил трансляции из [3].

В API Clang внутреннее представление программы называется AST. Для работы с ним API представляет довольно большой набор классов. Задачи трансляции удобнее всего решать, используя те из них, которые отвечают за реализацию шаблона проектирования `Visitor`. В классической книге [1] назначение паттерна `Visitor` описывается так: «Описывает операцию, выполняемую с каждым объектом из некоторой структуры». Из довольно большого числа классов, которые отвечают за реализацию паттерна `Visitor` в API Clang, использовались следующие.

`RecursiveASTVisitor` — это класс, который позволяет в прямом порядке поиском в глубину обойти все AST и посетить каждый узел. Этот класс предоставляет

возможность наследоваться от него и переопределить методы, вызывающиеся при обработке нужных узлов AST.

`SourceLocation` — это класс, который отвечает за местоположение того или иного объекта в исходном коде программы. Отметим, что при реализации правил трансляции с помощью класса `SourceLocation` можно запомнить участки кода, в которых были произведены изменения, что дает возможность создать протокол, по которому можно от конструкций в C-kernel программе вернуться к конструкциям C-light программы. Такой протокол важен для задачи локализации ошибок из [16].

Также API Clang позволяет легко реализовать транслятор из C-kernel в префиксную форму. Префиксная форма представляет собой линейную запись древовидной структуры, в которой любая программа может быть представлена. Необходимо, чтобы генератор условий корректности как можно меньше отвлекался на синтаксические особенности языка программирования, поэтому он получает текст программы в максимально упрощенном виде. Префиксная форма вполне удовлетворяет данным требованиям. В трансляторе из C-light в префиксную форму совершается обход всего AST. Этот обход начинается от деклараций самого высокого уровня. При реализации транслятора была использована такая возможность API Clang: в тех узлах AST, которые являются декларациями функций, это API позволяет получить списки деклараций, операторов и выражений, составляющих тело функции. Элементы этих списков тоже являются узлами AST. Далее совершается обход той части AST, которая имеет корнем эти узлы.

Для заданной аннотированной Си-программы важно знать, можно ли при ее верификации использовать смешанную аксиоматическую семантику. Идея, описанная в [13], заключается в том, чтобы выбрать классическую схему для доступа к значениям тех объектов, к которым не осуществляется доступ по их адресу. В анализаторе поиск таких объектов реализован довольно просто при помощи API Clang. Сначала составляется список всех переменных данной программы. Потом каждая переменная из этого списка проверяется на применение к ней операции взятия адреса. Такая проверка реализована путем обхода AST с использованием наследником класса `RecursiveASTVisitor`.

В случае, когда программа оказалась некорректной относительно своих спецификаций, необходимо найти источник проблем в программе. Эта задача решается с помощью трассировки недоказанных условий корректности в исходную программу. Для этого достаточно создать протокол, позволяющий от конструкций C-kernel программы вернуться к конструкциям исходной C-light программы. Для реализации такого протокола транслятор после каждого применения правила трансляции добавляет в код аннотированной C-kernel программы метаинформацию.

Рассмотрим пример добавления метаинформации при применении одного из правил трансляции. Код до применения правила трансляции:

```
56. for(;  i > 0 ; i++){
57.     k++;
58. continue;
59. j++;
60. }
```

Здесь числа 56 – 60 — номера строк, добавленные для наглядности. Код после применения правила трансляции:

```
78. /* begin changes BCE5 17 79-84 */
79. for(; i > 0; i++){
80.     j++;
81. goto l;
82. k++;
83. l:
84. }
85. /* end changes */
```

Обратный транслятор из C-kernel в C-light работает итеративно. На каждой итерации происходит просмотр информации обо всех применениях правил трансляции. При этом просмотре происходит поиск метаинформации, в котором содержится наибольший номер применения правила трансляции. Потом происходит применение правила трансляции, обратного к рассматриваемому. Рассматриваемая метаинформация заменяется на новую.

Рассмотрим пример исполнения одной из итераций. Пусть на данной итерации в метаинформации, записанной в нижеприведенном участке кода, содержится наибольший номер применения правила трансляции.

```
69. /* begin changes BCE5 19 70-75 */
70. for(; i > 0; i++){
71.     j++;
72. goto l;
73. k++;
74. l:
75. }
76. /* end changes */
```

После исполнения итерации вышеприведенный участок кода заменится на:

```
43. /* begin reverse 70-75 */
44. for(;  i > 0 ; i++){
45.     k++;
46. continue;
47. j++;
48. }
49. /* end reverse */
```

При обнаружении ошибок, возникающих при верификации, обратный транслятор получает данные о диапазоне строк аннотированной C-kernel программы, в которых эту ошибку следует искать. Далее, обратный транслятор исполняет все итерации и в результате их исполнения получает исходную аннотированную C-light программу с добавленной им метаинформацией. В этой метаинформации обратный транслятор отыскивает самый узкий диапазон, в который вкладывается рассматриваемый диапазон строк аннотированной C-kernel программы. Метаинформация, в которой был найден искомый диапазон, ограничивает некоторый участок программы с помощью комментария `/* begin reverse ... */` и соответствующего комментария `/* end reverse */`. Этот участок и содержит ошибку.

3. Генерация условий корректности

Генератор условий корректности основан на смешанной аксиоматической семантике языка C-kernel [4]. Он реализован на C++. Входным файлом генератора является выходной файл транслятора из C-light в C-kernel, который содержит аннотированную C-kernel программу в префиксной форме. Программа и ее спецификация хранятся в древовидной структуре, обеспечивающей удобство выполнения необходимых подстановок согласно правилам смешанной аксиоматической семантики.

Правила смешанной аксиоматической семантики сконструированы следующим образом:

1. Мы двигаемся от начала программы к ее концу и удаляем самый левый оператор (на верхнем уровне), применяя соответствующее правило (прямое прослеживание).

2. На каждом шаге мы применяем одно и только одно правило (однозначность вывода).

Также был реализован алгоритм трансляции инвариантов циклов из C-light в C-kernel [4].

Идея смешанной аксиоматической семантики состоит в разработке нескольких вариантов правила вывода для одной и той же программной конструкции, которые применяются в зависимости от ее контекста. Например, у нас имеются два правила для операции присваивания. Первое — для неразделяемых переменных (т. е. к значениям которых нет доступа через указатели), второе — для разделяемых (т. е. к значениям которых имеется доступ через указатели). С целью выявления таких переменных и выбора соответствующего правила был разработан алгоритм поиска разделяемых и неразделяемых переменных [5].

Рассмотрим общее правило вывода для операции простого присваивания:

$$\frac{\begin{array}{c}E \vdash \{\exists MD'\ P(MD \leftarrow MD') \wedge \\ MD = upd(MD', addr(val(e, MD'), MD'), \\ cast(val(e_0, MD'), type(e_0), type(e)))\} A; \{Q\}\end{array}}{E \vdash \{P\}\, \mathbf{e} = \mathbf{e_0};\ A; \{Q\}}\ .$$

Здесь e_0 не содержит вызовов функций и операторов приведения типов. Запись $P(V \leftarrow V')$ обозначает замену всех вхождений метапеременной V в формулу P на V'.

Вариант этого правила, когда x — неразделяемая переменная, имеет вид:

$$\frac{\begin{array}{c}E \vdash \{\exists x'\ P(x \leftarrow x') \wedge \\ x = cast(val(e_0(x \leftarrow x'), MD), type(e_0), type(x))\} A; \{Q\}\end{array}}{E \vdash \{P\}\, \mathbf{x} = \mathbf{e_0};\ A; \{Q\}}\ .$$

На завершающей стадии все условия корректности записываются в выходной файл в формате, удовлетворяющем входному языку Simplify.

4. Пример — поиск максимума в массиве

В настоящее время для экспериментов в нашей системе мы используем программы из соревнования COST IC0701 [7] и 1-го соревнования верифицированного программного обеспечения [11]. Отличительная особенность сравнения эффективности систем верификации состоит в том, что они сравниваются не только по производительности (измеренной в (милли)секундах). Важнейшим вопросом является *способность* инструментария верифицировать определенный класс программ. Эксперименты показывают, что наша система достаточно мощная, чтобы справиться с рассматриваемыми примерами.

Рассмотрим здесь один из примеров (информацию по другим можно найти в [13]). Дан непустой массив `a` целых чисел. Функция `max()` должна вернуть индекс максимального элемента в `a`. Мы использовали спецификации, предложенные командой Dafny [12]. Также мы представили их в форме, принимаемой Simplify. Однако на практике аннотации на языке ACSL транслируются в данный синтаксис программой из раздела 2.

Аннотированная программа на языке C-kernel имеет вид:

```
/* (AND (NEQ a |@NULL|)
        (> length 0))
*/
int max(int* a, int length)
{
    auto int x = 0;
    auto int y = length - 1;

    /* (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])
    */
    while (x != y)
    {
        if (a[x] <= a[y]) {x = x + 1; } else {y = y - 1;}
    }

    return x;
```

```
}
/* (AND (<= 0 x)
        (< x length)
        (FORALL (i) (IMPLIES (AND (<= 0 i) (< i length))
                             (>= a[x] a[i]))))
*/
```

Генератор УК выдает 4 условия, рассмотрим одно из них:

```
(IMPLIES
 (AND
   (AND (<= 0 x)
        (< x length)
        (<= 0 y)
        (< y length)
        (>= y x)
        (FORALL (i) (IMPLIES (AND (<= 0 i) (<= i x))
                             (OR (<= (select a i) (select a x))
                                 (<= (select a i) (select a y)))))
        (FORALL (i) (IMPLIES (AND (<= y i) (<= i (- length 1)))
                             (OR (<= (select a i) (select a x))
                                (<= (select a i) (select a y))))))
   (NOT (NEQ x y)))
 (AND (<= 0 x)
      (< x length)
      (FORALL (i) (IMPLIES (AND (<= 0 i) (< i length))
                           (>= (select a x) (select a i))))))
```

Оно может быть доказано с помощью Simplify с использованием только стандартной семантики выбора элементов массива.

5. Работа с условиями корректности

В ходе экспериментов дальнейшее развитие получили исследования по формальной локализации ошибок и объяснению условий корректности.

Напомним, что основная идея подхода состоит в расширении правил вывода Хоара специальными семантическими метками. Метки отражают вклад синтаксических конструкций в вывод условия и выделяют роли под-формул относительно предназначения всего условия.

В качестве иллюстрации приведем пример размеченного правила для композиции:

$$\frac{\mathcal{E}nv \Vdash \{P\}\ S_1; \{\ulcorner R \urcorner^{\mathsf{ens\_post}}\} \qquad \mathcal{E}nv \Vdash \{\ulcorner R \urcorner^{\mathsf{asm\_pre}}\}\ S_2; \{Q\}}{\mathcal{E}nv \Vdash \{P\}\ S_1\, S_2; \{Q\}}\ . \tag{1}$$

Согласно меткам, мы в явном виде должны убедиться, что промежуточное утверждение R является постусловием и предусловием относительно операторов S_1 S_2, соответственно. Достоинством метода является то, что исходные правила Хоара для C-kernel не меняются. Мы просто «навешиваем» на них метки.

Далее, при выводе мы получаем помеченные УК. Метки из них можно извлечь, нормализовать и, применив специальные текстовые шаблоны, породить по ним объяснения для УК на обычном языке.

Новое расширение этого подхода связано с задачей локализации ошибок. Ранее для доказательства УК метки из него извлекались. В случае ошибки доказатель выдавал некий контрпример и пользователь вручную должен был соотнести его с исходной формулой и пояснением для исходной формулы. Для реалистичных (т. е. сложных) УК это становится непростой задачей.

Поэтому была предложена идея по явному внедрению меток в УК, так чтобы истинность логических формул не менялась и контрпримеры содержали метки. Напомним, что помимо описания «смысла» подформул метки хранят в себе и информацию об исходных точках программы (т.е. локализуют подформулы). При этом желательно, чтобы эта идея работала даже в достаточно «примитивном» доказателе Simplify.

В качестве инструмента для работы с метками в логических формулах используются шаблоны доказателя Simplify. Вначале транслятор переводит помеченные логические термы в так называемые S-выражения. А именно, встретив терм вида

$$\ulcorner e_1 \; op \; e_2 \urcorner^{c(o,n)},$$

он переводит его в

$$\texttt{(L\_}op'\ e_1\ e_2\ \texttt{(label\_}c\ o\ n\ \texttt{))},$$

где op' — это подходящее LISP-представление для op. Например, знак равенства $=$ заменяется на `L_EQ`. Наконец, мы должны установить связь между новыми именами `L_`op' и стандартными ключевыми словами. Так, для знака $=$ будет использован шаблон

```
(FORALL (e1 e2 dum)
    (PATS (L_EQ e1 e2 dum))
    (EQ e1 e2))
```

Таким образом, каждое УК, переданное Simplify, должно быть доказано в таком окружении. К счастью, множество функциональных / операционных имен и логических связок конечно. Мы можем создать библиотеку таких шаблонов для использования в наших экспериментах с C-light программами. Более того, пользователю не нужно изучать эти шаблоны. Когда он получает контрпример, единственное, что он должен сделать — это найти помеченные термы (достаточно прямолинейно) и найти соответствующие части в объяснении УК.

Например, если пользователь забыл поставить восклицательный знак в условии цикла (пример из разд. 4), это приведет к опасной ситуации. Программа будет успешно скомпилирована и выполнена с непредсказуемым результатом (даже аварийным остановом). В рассмотренном выше УК соответствующий терм примет вид `(NOT(NOT y))`. Simplify просигнализирует, что `y` не может быть одновременно пропозициональной и целочисленной переменной. Если мы используем технику меток, контрпример будет содержать следующее:

$$\texttt{(L\_NOT (L\_NOT y (label\_loop-cond } n_1\texttt{)) (label\_loop-exit } n_2\texttt{))},$$

где n_1 и n_2 — соответствующие строки программы. Это значительно помогает найти источник проблемы.

В работе [16] были рассмотрены два примера использования данного подхода для задачи объяснения УК и локализации ошибок.

6. Заключение

В данной статье мы представили систему верификации C-light программ, основанную на наших новых формальных методах.

Во многих случаях сложность условий корректности, присущую общей аксиоматической семантике C-kernel, можно избежать. Формальный выбор между упрощенными и стандартными правилами Хоара реализован в смешанной аксиоматической семантике.

Успешная работа генератора условий корректности, а также метода локализации ошибок существенно зависит от первого модуля системы верификации (рис. 1). Из-за постоянного добавления новых возможностей (таких как ACSL) и методов приходится отказаться от традиционных инструментов (таких как Flex / Bison) в пользу более удобных. API и средства анализа, предоставляемые инфраструктурой LLVM и компилятором Clang, значительно уменьшили усилия в разработке нашей системы. Таким образом, мы можем рекомендовать эти инструменты для решения схожих проблем.

Мы также показали успешное применение наших методов к верификации программы. Несмотря на тот факт, что эта программа достаточно простая, она может представлять собой определенные трудности для систем верификации, основанных на логике Хоара, что подтверждается соревнованиями по верификации.

В дальнейшем мы планируем улучшить реализацию наших модулей, расширить набор примеров и провести эксперименты с SMT-решателем Z3.

Список литературы

1. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994. 395 p.

2. Кондратьев Д. А., Промский А. В. Комплексный подход к локализации ошибок при верификации Си-программ // Системная информатика. 2013. № 1. С. 79–96 (Kondratyev D.A., Promsky A.V. Kompleksny podkhod k lokalizatsii oshibok pri verifikatsii Si-programm // Sistemnaya informatika. 2013. N 1. P. 79–96 [in Russian]).

3. Непомнящий В.А., Анureев И.С., Михайлов И.Н., Промский А.В. Ориентированный на верификацию язык C-light // Системная информатика: сборник научных трудов. 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. Orientirovanny na verifikatsiyu yazyk C-light // Sistemnaya informatika: sbornik nauchnykh trudov. Novosibirsk: Izdatelstvo SO RAN, 2004. Vyp. 9: Formalnye metody i modeli informatiki. P. 51 – 134 [in Russian]).

4. Anureev I.S., Maryasov I.V., Nepomniaschy V.A. C-programs Verification Based on Mixed Axiomatic Semantics // Automatic Control and Computer Sciences. 2011. Vol. 45. Issue 7. P. 485–500.

5. Anureev I., Maryasov I., Nepomniaschy V. Revised Mixed Axiomatic Semantics Method of C Program Verification // Proc. of 3rd Workshop "PSSV: Theory and Applications". Nizhni Novgorod, 2012. P. 16–23.

6. Baudin P., Cuoq P., Filliâtre J.-C., Marché C., Monate B., Moy Y., Prevosto V. ACSL: ANSI/ISO C Specification Language.
`http://frama-c.com/download/acsl_1.4.pdf`

7. Bormer T., Brockschmidt M., Distefano D., Ernst G., Filliâtre J.-C., Grigore R., Huisman M., Klebanov V., Marché C., Monahan R., Mostowski W., Polikarpova N., Scheben C., Schellhorn G., Tofan B., Tschannen J., Ulbrich M. The COST IC0701 Verification Competition 2011 // Revised Selected Papers of Int. Conf. FoVeOOS 2011. LNCS. 2011. Vol.7421. P. 3–21.

8. Cohen E., Dahlweid M., Hillebrand M., Leinenbach D., Moskal M., Santen T., Schulte W., Tobies S. VCC: A Practical System for Verifying Concurrent C // Proc. of 22nd Int. Conf. TPHOLs. LNCS. 2009. Vol. 5674. P. 23–42.

9. Detlefs D., Nelson G., Saxe J. B. Simplify: A Theorem Prover for Program Checking. Palo Alto, 2003. 121 p. (HP Tech. Rep. HPL-2003-148).
`http://www.hpl.hp.com/techreports/2003/HPL-2003-148.pdf`

10. Filliâtre J.-C., Marché C. Multi-prover Verification of C Programs // Proc. of 6th ICFEM. LNCS. 2004. Vol. 3308. P. 15–29.

11. Klebanov V., Müller P., Shankar N., Leavens G. T., Wüstholz V., Alkassar E., Arthan R., Bronish D., Chapman R., Cohen E., Hillebrand M., Jacobs B., Leino K. R. M., Monahan R., Piessens F., Polikarpova N., Ridge T., Smans J., Tobies S., Tuerk T., Ulbrich M., Weiß B. The 1st Verified Software Competition: Experience Report // Proc. 17th Int. Symp. on Formal Methods. LNCS. 2011. Vol. 6664. P. 154–168.

12. Leino K.R.M. Dafny: An Automatic Program Verifier for Functional Correctness // Revised Selected Papers of 16th Int. Conf. LPAR-16. LNCS. 2010. Vol. 6355. P. 348–370.

13. Maryasov I.V., Nepomniaschy V.A., Promsky A.V., Kondratyev D.A. Automatic C Program Verification Based on Mixed Axiomatic Semantics // Proc. of 4th Workshop "PSSV: Theory and Applications". Yekaterinburg, 2013. P. 50–59.

14. Moura L. de, Bjørner N. Z3: An Efficient SMT Solver // Proc. of 14th Int. Conf. TACAS 2008. LNCS. 2008. Vol. 4963. P. 337–340.

15. Nepomniaschy V.A., Anureev I.S., Atuchin M.M., Maryasov I.V., Petrov A.A., Promsky A.V. C Program Verification in SPECTRUM Multilanguage System // Automatic Control and Computer Sciences. 2011. Vol. 45. Issue 7. P. 413–420.

16. Promsky A.V. A Formal Approach To The Error Localization. Novosibirsk, 2012. 33 p. (Preprint / RAS. Sib. branch. IIS; No 169).

Automatic C Program Verification Based on Mixed Axiomatic Semantics

Maryasov I. V., Nepomnyaschy V. A., Promsky A. V., Kondratyev D. A.

A.P. Ershov Institute of Informatics Systems SB RAS
prospect Akademika Lavrentjeva, 6, Novosibirsk, 630090, Russia
Novosibirsk State University, Pirogova Str., 2, Novosibirsk, 630090, Russia

Keywords: program verification, operational semantics, axiomatic semantics, C, C-light, C-kernel, partial correctness, ACSL, LLVM, Simplify

The development of the C-light project resulted in the application of new formalisms and implementation techniques which facilitate the verification process. The mixed axiomatic semantics proposes a choice between simplified and full-strength deduction rules depending on program objects and their properties. The LLVM infrastructure helps greatly in writing the C-light program analyzer and translator. The semantical labeling technique, proposed earlier, can now be safely kept in verification conditions during their proof. Two programs from the well-known verification benchmarks illustrate the applicability of the system.

Сведения об авторах:
Марьясов Илья Владимирович,
Институт систем информатики им. А. П. Ершова СО РАН,
младший научный сотрудник;
Непомнящий Валерий Александрович,
Институт систем информатики им. А. П. Ершова СО РАН, зав. лабораторией,
Новосибирский государственный университет, доцент;
Промский Алексей Владимирович,
Институт систем информатики им. А. П. Ершова СО РАН, научный сотрудник;
Кондратьев Дмитрий Александрович,
Новосибирский государственный университет, студент