Модел. и анализ информ. систем. Т. **18**, № **4** (2011) 157–167

УДК 519.681

Верификация Си-программ: объяснение условий корректности и стандартная библиотека

Промский А.В.[1]

Институт систем информатики имени А.П. Ершова СО РАН

e-mail: promsky@iis.nsk.su

получена 27 сентября 2011 года

Ключевые слова: верификация, спецификация, аксиоматическая семантика, язык C-light, ACSL, условие корректности, стандартная библиотека

Представлены два направления развития проекта по верификации Си-программ, разрабатываемого в ИСИ СО РАН. Во-первых, аксиоматическая семантика языка C-kernel была расширена семантической разметкой. Метки, выводимые непосредственно исчислением Хоара, соответствуют различным аспектам условий корректности (УК). Эти метки могут быть извлечены из термов и преобразованы в пояснения на естественном языке. Пояснения, понятные обычному пользователю системы верификации, могут быть полезны для понимания УК и локализации ошибок. Во-вторых, было специфицировано подмножество стандартной библиотеки языка Си. Спецификации написаны на языке ACSL и соответствуют модели памяти языка C-light. В работе представлены примеры использования этих двух подходов.

1. Введение

Проект по верификации Си-программ успешно развивается в Лаборатории теоретического программирования ИСИ СО РАН (см. [7, 8, 9]). Входной язык C-light представляет собой значительное подмножество стандарта C99. Его формальное определение было задано в виде структурной операционной семантики. В свою очередь, в языке C-light было выделено ограниченное ядро — язык C-kernel, обладающее непротиворечивой аксиоматической семантикой. При этом процесс верификации разбивается на два этапа. На первом шаге аннотированная C-light программа транслируется в эквивалентную программу на языке C-kernel program. Далее с помощью аксиоматической семантики выводятся условия корректности.

Успешное развитие теоретических основ проекта позволяет перейти к решению ряда практических проблем. Одна из них состоит в локализации ошибок и интерпретации условий корректности. Обычно аксиоматический метод Хоара сопоставляет

[1]Работа поддержана Лаврентьевским грантом СО РАН *Верификация программ на языке C с эффективной локализацией ошибок* и частично поддержана грантом РФФИ № 11-01-00028-а.

условия линейным участкам в исходной программе. Ложность условия сигнализирует о потенциальной ошибке на соответствующем участке (или в его аннотациях), но не позволяет узнать ее точное местоположение. Ситуация также усугубляется сложностью условий для языков подобных Си, что препятствует их ручному анализу.

Для решения этой проблемы был адаптирован подход, ранее представленный в работе Денни и Фишера [3]. Идея состоит в систематическом расширении правил Хоара так, чтобы само исчисление могло порождать объяснения для условий и сопоставлять фрагменты условий отдельным операциям в программе.

Другая практическая проблема заключается в отсутствии формальных спецификаций для стандартной библиотеки языка Си. Язык спецификаций ACSL [1] выглядит наиболее подходящим инструментом для этой задачи. В настоящий момент в рамках нашего проекта предложены спецификации для ряда библиотечных типов и функций, относящихся к файловому вводу-выводу, работе с памятью, операциям над строками и математическим функциям.

Работа состоит из 7 разделов. В разд. 2 кратко описан язык C-kernel. Семантическая разметка и преобразования меток рассмотрены в разд. 3. Специфицирование стандартной библиотеки обсуждается в разд. 4. Примеры, иллюстрирующие эти подходы, даются в разд. 5. Разд. 6 посвящен обзору родственных работ. Полученные результаты и планы по дальнейшему развитию рассмотрены в заключении.

2. Язык C-kernel

Язык C-kernel существенно ограничивает входной язык C-light. Позвольте кратко напомнить, что представляет собой сам C-light [7] по сравнению со стандартом C99. Во-первых, не поддерживается небольшое количество конструкций, чья семантика существенно зависит от низкоуровневых деталей реализации (например, битовые поля и объединения). Во-вторых, мы фиксируем ряд аспектов поведения, не определяемых самим стандартом. Например, выражения вычисляются слева-направо и побочные эффекты срабатывают сразу.

Объявления. Списки деклараторов в C-kernel разрешены только в объявлениях функций. Инициализаторы составных объектов должны быть в полноскобочной форме. Спецификаторы класса памяти задаются явно.

Выражения. Основная цель здесь состоит в том, чтобы в любом выражении, отличном от вызова функции, было не более одного побочного эффекта. Поэтому в C-kernel запрещены составные присваивания и операция последовательного вычисления. Также традиционно мы вводим операции `new` и `delete` в стиле C++ для работы с памятью [2].

Базовыми в языке C-kernel являются *нормализованные* выражения ($NExp$). Они строятся по стандартным правилам языка Си с помощью первичных, унарных и бинарных операций и приведений типов. Тем самым изменений состояний памяти в них не происходит.

[2]Это делалось для некоторого упрощения семантики, хотя нынешнее специфицирование библиотеки позволит избавиться от искусственных элементов.

Программа на C-light	**Эквивалентная C-kernel программа**

`void NegFirst(int ia[], int Length)`
`{`
`  //@ pre ...`
`  int i;`
`  for (i = 0; i < Length; i++)`
`  {`
`  //@ inv ...`
`  if (ia[i] < 0) {`
`  ia[i] = -ia[i];`
`  break;`
`  }`
`  }`
`  //@ post ...`
`}`
	`1 void NegFirst(int ia[], int Length) {`
	`2 //@ pre ...`
	`3 auto int i;`
	`4 i=0;`
	`5 while(i < Length){`
	`6 //@ inv ...`
	`7 if (ia[i]<0){`
	`8 ia[i] = -ia[i];`
	`9 goto L;`
	`10 }`
	`11 else {}`
	`12 auto int* q1;`
	`13 q1 = &i;`
	`14 *q1 = *q1 + 1;`
	`15 }`
	`16 L:;`
	`17 //@ post ...`
	`18 }`

Таблица 1. Пример трансляции из C-light в C-kernel

Пусть CV обозначает имена констант и переменных. Тогда синтаксис выражений языка C-kernel можно описать следующим образом:

$$Exp ::= f\texttt{(}CV\texttt{,...,}\ CV\texttt{)} \mid NExp\ \texttt{=}\ f\texttt{(}CV\texttt{,...,}\ CV\texttt{)} \mid NExp\ \texttt{=}\ NExp \mid NExp\ \texttt{=}\ \texttt{new}\ Type\ \texttt{[}NExp\texttt{]}_{opt} \mid \texttt{delete []}_{opt}\ NExp$$

Операторы. Язык C-kernel допускает следующие операторы:

$$Stat ::= Exp\texttt{;} \mid l\texttt{:}\ Stat \mid \texttt{goto}\ l\texttt{;} \mid \texttt{return}\ NExp_{opt}\texttt{;} \mid \texttt{if (}NExp\texttt{)}\ Stat\ \texttt{else}\ Stat \mid \texttt{while (}NExp\texttt{)}\ Stat \mid \texttt{\{}Stat\ \ldots\ Stat\texttt{\}}$$

Несмотря на то, что C-kernel является крайне ограниченным подмножеством, любую C-light программу можно перевести в эквивалентную программу в C-kernel. В Табл. 1 приведен пример такой трансляции.

3. Расширение логики Хоара с помощью меток

Верификация методом Хоара хорошо работает только в идеальном случае, когда программа и ее спецификации корректны, теория проблемной области полна и автоматический доказатель теорем обладает достаточной мощью. Тогда программа может быть верифицирована автоматически с минимальным участием пользователя. Но на практике некоторое из перечисленных условий (а зачастую все) бывает не выполнено. В этом случае пользователь получает набор недоказанных/ложных условий корректности, но, как правило, не получает информации о причинах неудачи. Он должен проанализировать условия, проинтерпретировать их составные части и соотнести их с исходными местами в программе.

Концепции	**Примеры меток**	**Аспекты условий корректности**
Гипотезы • Утверждения • Управляющие предикаты	asm_pre, asm_inv, then, while_t	*Гипотезы* отражают предположения о том, что некоторые логические утверждения выполнены в тех или иных точках программы. Это могут быть как исходные предусловия, так и управляющие выражения операторов наподобие `while` и `if`.
Заключения	ens_post, ens_inv_iter	*Заключения* отражают основное предназначение условий корректности, состоящее в *гарантировании* выполнения тех или иных утверждений в заданных местах программы.
Уточнители • Подстановки • Присваивания	sub, upd, alloc, init	*Уточнители* вводят более детальную характеризацию для гипотез и заключений, отражая способ получения под-формулы. Как правило, они соответствуют выражениям и операциям в программе.
Индуктивные уточнители	call, pres_inv	*Индуктивные уточнители* отражают второстепенное предназначение условия корректности. Например, условия корректности для вложенного цикла концептуально связаны с предназначением условий и для охватывающего цикла.

Таблица 2. Роль под-формул условия корректности для верификации

На базе идеи Денни и Фишера [3] предлагается расширить правила Хоара с помощью «семантической разметки» так, чтобы само исчисление порождало объяснения для условий корректности. Эти структурированные метки прикрепляются к формулам в правилах Хоара. Метки, пройдя различные этапы обработки, извлекаются из окончательных условий корректности и преобразуются в объяснения на естественном языке (в данной работе — на английском).

Концепции и метки. При определении меток стоит разделить две задачи. Задача локализации ошибок достаточно проста. Метка, очевидно, должна содержать информацию о месте срабатывания правила Хоара (имя файла, строка и, возможно, колонка). Более сложная задача объяснения условия корректности требует понимания *роли* условия в процессе верификации.

В первом приближении можно воспользоваться тем фактом, что условия корректности, как правило, имеют вид Хорновских клозов: $H_1 \wedge \ldots \wedge H_n \Rightarrow C$. Тогда заключение C можно охарактеризовать как *предназначение* этого условия. Далее, для осмысленного объяснения структуры условия корректности необходимо предложить более детальную характеризацию его под-формул. Базисная информация при генерации объяснения представляет собой множество *концепций*, зависящее от того, какой аспект условий корректности необходимо объяснить. В Табл. 2 дается краткий обзор концепций, предложенных в настоящее время. Более подробный обзор можно найти в [10].

Мы используем нотацию из [3] для вывода помеченных термов вида $\ulcorner t \urcorner^l$, где каждый терм t может быть помечен некоторой меткой l. Сами метки имеют вид $c(o, n)$. Концепция c (см. примеры из второй колонки в табл. 2) описывает роль данного помеченного терма. Местоположение o отражает происхождение терма (номер стро-

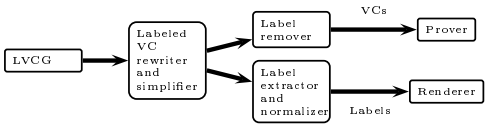

Рис. 1. Метки и помеченные условия корректности

ки или диапазон строк). Список меток (возможно, пустой) n записывает некоторую уточняющую информацию.

Примеры правил Хоара для C-kernel с метками. Основным достоинством данного метода является то, что семантические метки не меняют структуру исходных правил для C-kernel. Мы просто «украшаем» термы метками. Метка может быть добавлена к отдельной под-формуле, группе формул или тройке Хоара. Обычно метки над под-формулами соответствуют гипотезам, заключениям и уточнителям. Группы формул и тройки помечаются индуктивными уточнителями.

Исходная логика Хоара для C-kernel была представлена в [8], а ее вариант с метками в [10]. Рассмотрим для примера правило для композиции:

$$\frac{\mathcal{E}nv \Vdash \{P\}\ S_1\ \{\lceil R \rceil^{\mathsf{ens\_post}}\} \qquad \mathcal{E}nv \Vdash \{\lceil R \rceil^{\mathsf{asm\_pre}}\}\ S_2\ \{Q\}}{\mathcal{E}nv \Vdash \{P\}\ S_1\, S_2\ \{Q\}}$$

Таким образом, метки $\mathsf{ens\_post}$ and $\mathsf{asm\_pre}$ отражают роли промежуточного утверждения R. Необходимо гарантировать (ensure), что оно является постусловием в первой тройке, а также необходимо предположить (assume), что оно является предусловием во второй.

Переписывание помеченных термов и преобразование меток в объяснения. Условия корректности (как обычные, так и помеченные) становятся громоздкими при выводе и должны быть упрощены до этапа доказательства. Цель этапа переписывания состоит в удалении избыточных меток и минимизации области действия оставшихся, при этом необходимо сохранить достаточное количество меток для объяснения неудач с доказательством.

Помеченные правила переписывания базируются на обычных непомеченных правилах (например, см. [9]), но не переиспользуют их в неизменном виде, поскольку метки требуют более аккуратной работы. Например, с одной стороны, мы можем безопасно удалить метки из тождественно истинных под-формул, поскольку они не требуют объяснения. С другой стороны, метки из тождественно ложных подформул удаляются избирательно, чтобы сохранить информацию для объяснения неудачи.

Окончательной стадией работы с метками является преобразование их в объяснения, доступные человеку. Составные шаги излагаемого подхода представлены на Рис. 1 и включают в себя: (1) нормализацию условий корректности с использованием помеченной системы перевода; (2) извлечение меток; (3) нормализацию самих меток, с целью их соответствия шаблонам порождения объяснения; (4) генерацию текстовых объяснений с помощью набора шаблонов (реализованы на языке ML).

4. Спецификация стандартной библиотеки

К данному моменту разработаны ACSL спецификации для ряда библиотечных средств. Примеры заголовочных файлов и соответствующих функций включают:

`ctype.h` : character classification functions of the form `is...()`
`stdio.h` : `fopen`, `fclose`, `getc`, `fgetc`, `fgets`, `setbuf`, ...
`stdlib.h`: `malloc`, `calloc`, `free`, `qsort`, ...
`string.h`: `memset`, `strlen`, `strcmp`, `strcpy`, `strcat`, ...

Для иллюстрации рассмотрим некоторые функции из `stdio.h`, которые будут использованы в примере. В текущий момент мы используем высокий уровень абстракции для файлов. Например, мы не моделируем содержимое структуры `FILE`; все атрибуты потока доступны только через абстрактные функции.

```
/*@ requires valid_string(filename) && valid_string(mode);
    assigns \nothing;
    behavior update:
       assumes mode == \eq(mode, "r+b");
       ensures \result == res(filename);
    behavior truncate_or_update:
       assumes mode == \eq(mode, "w+b");
       ensures \result == rew(filename);
    ...
    behavior failure:
       ensures \result == NULL;
    complete behaviors update, truncate_or_update, ..., failure;
*/
FILE *fopen(const char restrict *filename, const char restrict *mode);

/*@ requires \valid(stream) && size >= 0 && nmemb >= 0;
    assigns *ptr, stream;
    behavior wrong_read:
       assumes size == 0 || nmemb == 0;
       ensures \result == 0 && stream == \old(stream);
    behavior good_read:
       assumes size > 0 || nmemb > 0;
       ensures \exists int n; n == \max(length(\old(stream))/size, nmemb) &&
       stream == get(\old(stream), n, size) &&
       \forall 0 <= i < n *ptr[i] == buf(stream)[i] && \result == n;
    complete behaviors wrong_read, good_read;
*/
size_t fread(void * restrict ptr, size_t size,
             size_t nmemb, FILE * restrict stream);
```

Таким образом, у функции `fopen` есть общее предусловие (ветвь `requires`), в котором происходит проверка допустимости аргументов. Каждое частное поведение соответствует режиму работы с файлом и обладает собственными пред- и постусловием (ветви `assumes` и `ensures` соответственно). В каждом случае открытый файл

моделируется логической функцией. Например, функции $res()$ и $rew()$ можно считать аналогами процедур `reset` и `rewrite` языка Паскаль. При ошибке функция `fread` возвращает ноль. Успешное чтение моделируется логической функцией get, которая является обобщенной моделью для функции `fgetc`. По аналогии функция `fwrite` моделируется логической функцией put, которая, в свою очередь, обобщает `fputc`.

В свою очередь, логические функции и файловый тип также необходимо аксиоматизировать. Полная ACSL-теория для файлов слишком обширна для данной статьи. Отметим лишь, что файл f моделируется через свои составные части $left(f)$ и $right(f)$ относительно файлового указателя. Ниже приведен пример некоторых свойств:

```
/*@ axiomatic EOF {
    logic FILE* f;
    axiom eof-1: eof(left(f)) ==> buf(left(f)) == \omega;
    axiom eof-2: eof(f) <==> right(f) == empty_file;
    ... */
```

Смысл аксиом очевиден: когда достигнут конец файла, содержимое файлового буфера не определено (ω), а правая часть пуста.

Таким образом, язык ACSL предоставляет полный набор средств для логического специфицирования функций языка Си.

5. Примеры

Объяснение условий корректности. Рассмотрим функцию `NegFirst` из Табл. 1. Ее предусловие и инвариант выглядят как

$$\text{pre}: \ \exists old : \texttt{int[]}.\ \mathrm{MD}(\mathrm{MeM}(\texttt{ia})) \neq \mathsf{null} \wedge \mathrm{MD}(\mathrm{MeM}(\texttt{ia})) = \mathrm{MD}(\mathrm{MeM}(old))$$

$$\begin{aligned}\text{inv}: \ & 0 \le \mathrm{MD}(\texttt{i}) \le \mathrm{MD}(\texttt{Length}) \wedge (\forall j.\ 0 \le j < \mathrm{MD}(\texttt{i}) \Rightarrow \\ & \quad (\mathrm{MD}(\mathrm{MeM}(\texttt{ia}, j)) \ge 0 \wedge \mathrm{MD}(\mathrm{MeM}(\texttt{ia}, j)) = \mathrm{MD}(\mathrm{MeM}(old, j)))\ .\end{aligned}$$

Мы не рассматриваем здесь постусловие, поскольку оно не используется в условии ниже (подробнее о верификации этого примера можно узнать в [9]). Согласно спецификациям `NegFirst` ищет первый отрицательный элемент массива, меняет его знак и завершает работу. Для краткости здесь использована обычная математическая нотация вместо ASCL-формата.

Генератор условий корректности порождает 5 условий и одну тождественно истинную тройку Хоара. Об их доказательстве в системах Simplify и Z3 можно также узнать в [9]. Рассмотрим задачу объяснения. Даже самое короткое из условий оказывается достаточно сложным для анализа пользователем. Это условие соответствует пути от точки входа функции до точки входа в цикл. Его помеченный вариант выглядит как

$$\left(\begin{array}{ll} \ulcorner \mathrm{pre}(MeM \leftarrow MeM_1)(MD \leftarrow MD_1) \urcorner^{\mathsf{asm\_pre}(2)} & \wedge \\ \ulcorner MeM = upd(MeM_1, (\texttt{i}, 1), nc) \urcorner^{\mathsf{alloc}(3)} & \wedge \\ \ulcorner MD_2 = upd(MD_1, nc, \omega) \urcorner^{\mathsf{init}(3)} & \wedge \\ \ulcorner MD = upd(MD_2, MeM(\texttt{i}, 1), 0) \urcorner^{\mathsf{asgn}(4)} & \end{array}\right) \Rightarrow \ulcorner \mathrm{inv} \urcorner^{\mathsf{ens\_inv}(6)}\ .$$

Опять же для краткости предусловие и инвариант записаны в нем в символической форме. Читатель может подставить исходные формулы, чтобы оценить окончательный объем условия. Что означает эта формула? Какую роль она играет в процессе верификации? Ответ на эти вопросы дает преобразование меток в следующее пояснение:

```
This VC corresponds to lines 2-6 in the function NegateFirst. Its purpose is to
ensure that the loop invariant at line 6 holds at the loop entry point. Hence,
given
  - assumption that function precondition holds at line 2,
  - substitution for MeM originating in object allocation at line 3,
  - substitution for MD originating in object initialization at line 3,
  - substitution for MD originating in assignment at line 4,
show that the loop invariant at line 6 holds at the loop entry point at line 5.
```

Как мы надеемся, это объяснение (написанное на естественном языке) сможет помочь в понимании данного условия или локализации возможных ошибок.

Работа со стандартной библиотекой. Рассмотрим простую программу копирования файлов:

```
#include <stdio.h>

/*@ requires \nothing;
    assigns from, to, Buffer;
    ensures \value == 1 ||
            \value == 0 && (file(from) == file(to)) && eof(from) && eof(to);
*/
int main(){
   FILE * from, *to;
   char Buffer;

   if ((from = fopen("example.txt", "r+b")) == NULL) return 1;
   if ((to = fopen("example.bak", "w+b")) == NULL) return 1;

   /*@ invariant left(from) == file(to) &&
                 (eof(from) ==> fbuf(from) == \omega) */
   while(fread(&Buffer, 1, 1, from) != 0)
      fwrite(&Buffer, 1, 1, to);
   return 0;
}
```

Спецификации используют некоторые абстрактные функции и предикаты, рассмотренные ранее. Для краткости опустим здесь промежуточную программу на C-kernel, отметив только, что результаты вызовов `fopen` и `fread` будут сохранены в новых вспомогательных переменных, так как в C-kernel управляющие выражения должны быть нормализованными.

Генератор порождает 5 условий корректности. Два из них соответствуют ошибке при открытии файла и поэтому тривиальны. Среди оставшихся рассмотрим условие для тела цикла. После упрощения оно выглядит как:

$$\left[\begin{array}{l} \ulcorner inv \urcorner^{\mathsf{asm\_inv}} \Rightarrow \\ \ulcorner \left(\begin{array}{l} \ulcorner (pre(\texttt{fread}) \wedge post(\texttt{fread}))\alpha \urcorner^{\mathsf{ens\_specs(fread)}} \\ \Rightarrow \\ \left(\begin{array}{l} \ulcorner \mathrm{MD}(\mathrm{MeM}(x)) \neq 0 \urcorner^{\mathsf{then}} \Rightarrow \\ \ulcorner (pre(\texttt{fwrite}) \wedge post(\texttt{fwrite}))\beta \Rightarrow \ulcorner inv \urcorner^{\mathsf{ens\_inv\_iter}} \end{array} \right) \end{array} \right) \urcorner^{\mathsf{call(fwrite)}} \end{array} \right) \urcorner^{\mathsf{call(fread)}} \end{array}\right]^{\mathsf{pres\_inv}}$$

где $\alpha \equiv (\texttt{ptr} \leftarrow \mathrm{MeM}(\texttt{Buffer}), \texttt{stream} \leftarrow \texttt{from}, \texttt{size} \leftarrow 1, \texttt{nmemb} \leftarrow 1)$ and $\beta \equiv (\texttt{ptr} \leftarrow \mathrm{MeM}(\texttt{Buffer}), \texttt{stream} \leftarrow \texttt{to}, \texttt{size} \leftarrow 1, \texttt{nmemb} \leftarrow 1)$. Метки в точности описывают роль каждой из под-формул. Например, предполагается, что инвариант цикла выполнен в точке входа и *должен* быть выполнен после каждой итерации (соответственно метки asm_inv и ens_inv_iter). Термы для вызовов функций помечены должным образом. В целом же метка pres_inv сообщает, что все условие используется для доказательства сохранения инварианта.

По аналогии с предыдущим примером эти метки можно преобразовать в объяснение условия (объем статьи не позволяет привести его).

Свойства функций для файлового ввода/вывода из Разд. 4 были использованы для доказательства.

6. Родственные работы

Среди большого числа проектов по верификации Си-программ (подробный обзор в [9]) отметим два проекта, в которых также используется трансляция в промежуточный язык. Во-первых, проект WHY/Frama-C [4], развиваемый в INRIA. В нем исходные программы транслируются в языки WHY и CIL. Основная цель такой трансляции — генерация условий корректности независимо от доказателя теорем.

Во-вторых, проект VCC (A Verifier for Concurrent C), развиваемый в Microsoft Research [2]. Программы транслируются в логические формулы с помощью инструмента Boogie, который сочетает в себе промежуточный язык Boogie PL и генератор условий корректности. В качестве недостатка обоих проектов в сравнении с нашим отметим отсутствие формального доказательства корректности трансляции.

В сравнении с проектами по верификации, работы по объяснению условий корректности и формальной локализации ошибок менее многочисленны. Наибольший интерес представляют следующие три. Во-первых, проект Centaur [5]. В нем генерируемые условия корректности анализировались для поиска условных выражений, которые использовались в исходных условных операторах и циклах. Для поиска использовались некоторые алгоритмы из области отладки программ.

Более позднее исследование [3] во многом повлияло на данную работу. Денни и Фишер предложили расширить правила Хоара с помощью меток с целью автоматического порождения объяснений для условий корректности. Объяснения можно легко настраивать для отражения различных аспектов условий корректности. Подход полностью декларативный, и для порождения объяснения анализируются только метки, а не логический смысл самих условий или их доказательства.

Наконец, Лейно и др. [6] также расширили базовую логику за счет меток, представляющих семантическую информацию, пригодную для объяснения. В их случае метки вводятся на этапе трансляции в промежуточный язык, который в дальнейшем обрабатывается стандартным безметочным генератором.

В качестве недостатка всех трех перечисленных проектов отметим использование модельных входных языков, более простых, чем Си.

7. Заключение

Большинство систем верификации, основанных на логике Хоара, предлагают простую трассировку условий корректности, связывая номера строк с порождаемыми условиями. Этот подход не предлагает информацию о том, *какие еще* фрагменты программы концептуально связаны с условием корректности, *как* это условие порождено или *в чем* его предназначение. Поэтому он не может служить базисом для построения более информативных объяснений.

Результаты. В данной работе представлена комбинация методов локализации ошибок и объяснения условий корректности, планируемая к применению в проекте верификации Си-программ. Ранее нами были продемонстрированы теоретическая непротиворечивость и практическая надежность логики Хоара для языка C-kernel. Ряд программ с такими сложными для верификации особенностями, как множественные ссылки, побочные эффекты и передача управления были успешно верифицированы. Теперь же она формально расширена техникой разметки. Расширенное исчисление предоставит пользователю информацию для понимания условий корректности и поиска возможных ошибок.

Другой важный результат состоит в специфицировании стандартной библиотеки. В настоящий момент предложены ACSL-аннотации для ряда функций над файлами, памятью и строками.

Будущие исследования. Очевидно, что метод разметки имеет дело только с промежуточной стадией в нашем двухуровневом подходе. Поскольку исходные C-light программы транслируются в C-kernel, формальная обратная трансляция объяснений и местоположений ошибок также должна быть разработана.

Помимо техники разметки, необходимо расширять набор ACSL-спецификаций для библиотеки. Эта задача особенно важна для верификации библиотечных средств.

Список литературы

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

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

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

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

6. 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.

7. Nepomniaschy V.A., Anureev I.S., Mikhailov I.N., Promsky A.V. Towards verification of C programs. C-Light language and its formal semantics // Programming and Computer Software. 2002. Vol. 28(6). P. 314–323.

8. Nepomniaschy V.A., Anureev I.S., Promsky A.V. Towards verification of C programs: Axiomatic semantics of the C-kernel language // Programming and Computer Software. 2003. Vol. 29(6). P. 338–350.

9. Promsky A.V. Towards C-light program verification: Overcoming the obstacles // Proc. PU-2009, 19–23 June, Altai Mountains, Russia, 2009. P. 53–63.

10. Promsky A.V. Error-tracing semantics for C-kernel // Bull. Nov. Comp. Center, Comp. Science. 2010. 31. P. 123-138.

C Program Verification: VC Explanation and the Standard Library

A. V. Promsky

Keywords: verification, specification, axiomatic semantics, C-light language, ACSL, verification condition, standard library

The C program verification project is being developed in IIS. Its latest extension is twofold. First, the labeled variant of axiomatic semantics of the C-kernel language was proposed. The labels, introduced in the calculus, correspond to various concepts inherent in verification conditions (VC). These labels can be extracted from terms and rendered into explanations written in the natural language. User-friendly explanations can play a crucial role in VC understanding and error localization. Second, a subset of the C standard library was specified. The specifications written in ACSL correspond to the C-light memory model. The examples in the paper illustrate the use of these two techniques.

Сведения об авторе:
Промский Алексей Владимирович,
Институт систем информатики имени А.П. Ершова СО РАН, науч. сотр.