УДК: 519.681

Название: Комплексный подход к локализации ошибок при верификации Си-программ

Автор(ы):

Кондратьев Д.А. (Новосибирский государственный университет),

Промский А.В. (Институт систем информатики им. А.П. Ершова СО РАН)

Аннотация: Отличительной чертой проекта C-light является максимальное использование формальных непротиворечивых методов. Это касается не только фундаментальных базисов для верификации, таких как операционная семантика Плоткина или логика Хоара, но и реализационных аспектов. Одним из таких моментов является локализация потенциальных ошибок. В большинстве известных систем верификации локализация ошибок просто реализуется программистами как часть функционала системы без подведения какой-либо формальной основы под этот процесс. В свете недавнего расширения проекта C-light техникой семантической разметки и выбора инструментария LLVM/Clang для входного блока системы мы приводим обзор наших наработок для решения этой задачи.

Ключевые слова: семантика, спецификация, верификация, трансляция, локализация ошибок, разметка, Си, C-light, LLVM, Clang

1. Введение. В области разработки систем верификации программ[1] можно выделить два основных направления. К первому относятся системы, ориентированные на максимальную производительность или на поиск как можно более многочисленных типов ошибок[2]. Однако зачастую эффективность поиска ошибок является следствием применения менее строгих методов и алгоритмов. Особенно показателен пример системы ESC/Java, где применялись не только неполные, но в некоторых случаях и противоречивые методы.

Противоположный лагерь представляют академические исследования, ориентированные в основном на обучение методам верификации, или системы, поддерживающие более узкие классы свойств программ и/или ошибок в них. Для таких систем характерно использование более строгих и надежных подходов. Так, если сравнивать их с упомянутой ESC/Java, то можно заметить, что для ограниченных диалектов Java Card и Java^{light} при помощи системы Isabelle/HOL были разработаны полные и непротиворечивые семантики.

В проекте C-light, развиваемом в Лаборатории теоретического программирования ИСИ СО РАН, мы, конечно же, стремимся к широкому охвату языка Си и верифицируемых свойств в рамках дедуктивного подхода Хоара. Однако приоритетной для нас является максимальная корректность используемых методов.

Для того чтобы сформулировать основную тему данной работы, нам понадобится крат-

[1]Мы намеренно не указываем язык, так как это верно не только для Си, но и для, скажем, Java, C# и т.д.

[2]В зарубежной литературе используется термин "industrial-strength verification".

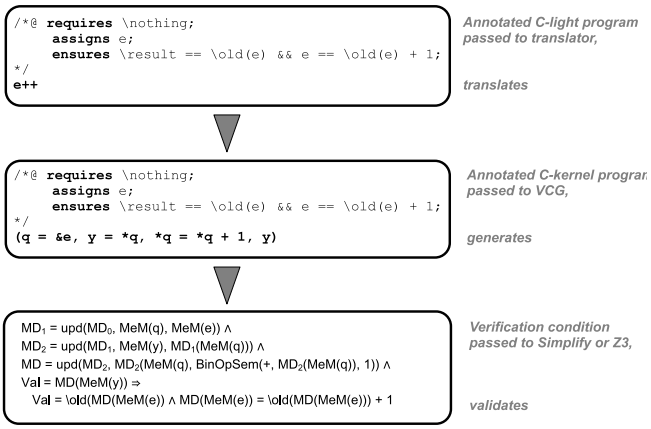

Рис. 1: Схема верификации в проекте C-light

кий обзор проекта C-light. Итак, язык C-light является представительным подмножеством стандарта C99. Детальный обзор его допустимых конструкций можно найти в [3]. Здесь же отметим следующее. В ходе эволюции языка Си во многом преследовались цели эффективности и максимальной переносимости программ. Это привело к тому, что стандарт для некоторых конструкций не дает детальной или вообще детерминированной семантики, оставляя ее на усмотрение разработчиков компилятора. Поэтому семантика некоторых низкоуровневых конструкций C-light обладает высоким уровнем абстракции (например, адресация или раскладка значений в памяти), а такой тип данных, как объединение (union), запрещен. Также в C-light фиксирован порядок вычисления выражений, а побочные эффекты срабатывают сразу. Для C-light была разработана структурированная операционная семантика, фактически являющаяся его формальным определением. Несмотря на ограничения, язык C-light все же слишком широк для классической логики Хоара. Поэтому в нем было выделено ограниченное ядро C-kernel, и предложены формальные правила перевода из C-light в C-kernel. Была доказана формальная корректность такого перевода. Это позволило предложить простую аксиоматическую семантику C-kernel в классическом стиле Хоара, что, в свою очередь, позволило доказать ее непротиворечивость относительно операционной семантики. Таким образом, для C-light используется двухэтапная схема верификации, представленная на рис. 1. Отметим, что в качестве языка спецификаций выбран язык ACSL.

Как видно из обзора, корректность каждого из этапов в нашем подходе была формально обоснована. Вместе с тем схема на рис. 1 не описывает весь процесс верификации. В частности, в ней не представлена интерпретация результатов верификации пользователем. Особый интерес представляет случай, когда доказатель теорем (prover) обнаружил ложные или недоказанные условия корректности, и необходимо найти источник проблем

в программе. В подавляющем большинстве случаев трассировку условий обратно в исходную программу реализуют как часть системы, но не предлагают никакой формализации для нее. Для решения этой задачи в нашем проекте было разработано расширение аксиоматической семантики языка семантическими метками, которые позволяют формализовать соотнесение условий корректности и фрагментов программ, а также позволяют автоматически порождать объяснения для условий корректности на естественном языке.

Наконец, разработка формализма для локализации ошибок снова показала, что корректность должна быть установлена не только для теоретических методов, но и для практических аспектов, связанных с реализацией системы верификации. Идеальным решением была бы реализация системы на языке C-light и ее (само)верификация. Эта амбициозная задача еще только формулируется в планах. Пока же в контексте задачи локализации ошибок при верификации разрабатывается входной анализатор/транслятор с помощью нового инструментария LLVM/Clang. При этом для той его части, что вкладывается в C-light, будут предложены формальные спецификации на языке ACSL. Это позволит провести его частичную верификацию.

Итак, в данной работе мы представляем обзор теоретических и практических результатов, полученных нами в контексте решения задачи локализации ошибок при верификации Си-программ. В разд. 2 обсуждаются причины выбора нового инструментария, и описана архитектура анализатора/транслятора. Разд. 3 посвящен расширению логики Хоара для языка C-kernel семантическими метками и их использованию для локализации ошибок и интерпретации условий корректности. Обзор родственных работ дается в разд. 4.

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

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. На смену использовавшейся ранее классической связке Flex/Yacc(Bison), пришла относительно новая связка LLVM/Clang. О причинах такого выбора и архитектуре разработанного прототипа транслятора говорится в первой и второй частях данного раздела, соответственно. Вопрос сопоставления фрагментов C-kernel программы и фрагментов исходной C-light программы затронут при рассмотрении одного из классов API Clang. Это сопоставление важно для задачи локализации ошибок.

2.1. Выбор инструментария. Было принято решение не реализовывать данный транслятор «с нуля», а использовать уже существующие инструменты для лексического анализа и построения внутреннего представления аннотированных C-light программ. Данное решение было принято по следующим причинам:

1. Многие из этих инструментов имеют очень хорошо продуманную структуру, и поэто-

му, предоставляют довольно удобное API для работы со внутренним представлением программ. Если бы была предпринята попытка своими силами разработать собственное API для тех же целей, то оно вероятнее всего не было бы удобней API, предоставляемых этими инструментами, так как эти API совершенствовались в течение многих лет.

2. Многие из этих инструментов гарантируют, что они поддерживают все возможности языка Си, описанные в стандарте ISO/IEC 9899:1999. Для реализации нашего транслятора этот факт очень важен, так как язык C-light является очень широким подмножеством языка Си, и поэтому очень схож с ним. Попытка реализовать поддержку всех возможностей языка C-light получилась бы бесполезной тратой времени ввиду того, что уже существуют инструменты, поддерживающие все возможности языка Си.

3. Очень многие из этих инструментов можно использовать не просто для анализа C-light программ, а для анализа C-light программ, снабженных ACSL спецификациями. Это возможно благодаря тому, что с точки зрения этих инструментов ACSL спецификации являются самыми обычными комментариями в Си-программе и никак не нарушают синтаксис и семантику Си-программ. Более того, многие из этих инструментов не вырезают комментарии, и поэтому позволяют создавать анализаторы ASCL спецификаций.

4. Многие из этих инструментов хорошо протестированы на наличие ошибок. Это подтверждается большими тестовыми базами, на которых проверялись эти инструменты. Было бы сложно создать такие большие тестовые базы только собственными силами.

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

- API на языке программирования C++ позволяет использовать возможности объектно-ориентированного анализа и дизайна при разработке транслятора. Использование объектно-ориентированного анализа и дизайна позволяет значительно облегчить разработку практически любых программных систем. Также использование

объектно-ориентированного анализа и дизайна при разработке транслятора дает возможность довольно легко вносить изменения в уже реализованный транслятор. Задача внесения изменений в уже реализованный транслятор является актуальной в связи с тем, что в настоящее время активно ведутся работы по расширению языка C-light.

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

- Задача трансляции исходного кода C-light программ в исходный код C-kernel программ предполагает много работы со строковыми данными. Язык программирования C++, на котором предоставляет свой API компилятор Clang, дает возможность легко работать со строковыми данными по сравнению со многими другими языками программирования (например, по сравнению с языком программирования Си).

- Сейчас компилятор Clang активно поддерживается, и нет оснований полагать, что его поддержка в скором времени прекратится. Значит, с большой уверенностью можно предполагать, что API компилятора Clang является надежным инструментом для лексического анализа и построения внутреннего представления Си-программ.

- Язык программирования C++, на котором предоставляет свой API компилятор Clang, дает возможность снабдить спецификациями код методов классов, определенных при реализации транслятора по аналогии с ACSL-спецификациями аннотированных C-light программм. Благодаря этому появляется возможность провести формальную верификацию транслятора.

Так как API Clang имеет встроенный лексический анализатор и парсер Си-программ, то использование API Clang позволило не заниматься реализацией этой функциональности. В тексте программы, являющейся реализацией транслятора, лексический анализ и парсинг C-light программ занимает всего 35 строк. Если бы в тексте этой программы не использовалось API Clang, то, по проведенным оценкам, лексический анализ и парсинг C-light программ занимал бы в этом тексте приблизительно 7000 строк. Получается, что при использовании API Clang удалось добиться сокращения части кода, отвечающей за рассматриваемую функциональность, приблизительно в 200 раз.

Отметим, что использование API Clang позволяет добиться существенного уменьшения текста программы, реализующей транслятор. Текст этой программы занимает приблизительно 11000 строк. Если бы в этом тексте не использовалось бы API Clang, то, по

проведенным оценкам, текст этой программы занимал бы приблизительно 22000 строк. Итак, с помощью API Clang удалось сократить количество строк кода транслятора приблизительно в 2 раза. Все эти данные подтверждают правильность принятых решений.

2.2. Архитектура транслятора. Транслятор аннотированных C-light программ в аннотированные C-kernel программы был реализован на языке программирования C++, так как при реализации этого транслятора активно использовалось API на языке программирования C++, предоставляемое компилятором Clang. При создании транслятора активно использовались принципы объектно-ориентированного анализа и дизайна. Поэтому при рассмотрении текста программы, являющейся реализацией транслятора, интересно рассмотреть классы из API компилятора Clang, объекты которых использовались в этой программе, а также другие классы, на которых строится реализация правил трансляции. Особенно важно рассмотреть классы, отвечающие за внутреннее представление программы и работу с ним.

В API, предоставляемым компилятором Clang на языке программирования C++, внутреннее представление программы называется `AST`. Для работы с этим внутренним представлением API компилятора Clang предлагает довольно большой набор классов. Задачи трансляции удобнее всего решать, используя те из этих классов, которые отвечают за реализацию шаблона проектирования `Visitor`. В классической книге назначение [1] паттерна `Vistor` описывается так: “Описывает операцию, выполняемую с каждым объектом из некоторой структуры” т.е. паттерн `Visitor` позволяет осуществить обход `AST` и выполнить при этом обходе ряд операций для реализации правил трансляции. При использовании паттерна `Visitor` можно не опасаться появления зависимостей от внутреннего устройства `AST`. Использование в нашем трансляторе классов, которые отвечают за реализацию шаблона проектирования `Visitor`, позволяло при его разработке концентрироваться не на обходе дерева, а на реализации правил трансляции.

Из довольно большого числа классов, которые отвечают за реализацию паттерна `Visitor` в API, предоставляемом компилятором Clang, при разработке нашего транслятора использовались следующие:

`ASTConsumer` — это интерфейс, который должны реализовывать те, кто использует `AST`. `ASTConsumer` позволяет создать уровень абстракции, на котором можно быть независимым от того, кто предоставляет `AST`. Таким образом, используя `ASTConsumer`, можно не заботиться о том, кто предоставил `AST` — парсер или класс, десериализующий `AST` из какого-либо файла. Для того чтобы начать обход `AST`, удобнее всего наследоваться от класса `ASTConsumer` и переопределить его методы. При реализации транслятора был создан класс `MyASTConsumer`, наследник класса `ASTConsumer`. В классе `MyASTConsumer` был переопределен метод `HandleTopLevelDecl` класса `ASTConsumer`. После переопределения этого

метода, была получена возможность обойти все декларации самого высокого уровня. В нашем трансляторе обход всех деклараций самого высокого уровня является началом обхода `AST`.

`RecursiveASTVisitor` — это класс, который позволяет в прямом порядке поиском в глубину обойти все `AST` и посетить каждый узел. При реализации транслятора была использована в том числе и такая функциональность этого класса: если выбрать декларацию, то этот класс позволяет обойти ту часть `AST`, которая имеет корнем эту декларацию. Именно эта возможность была использована при реализации транслятора в том месте, где вызывается метод `TraverseDecl`, и аргументом ему передается декларация, полученная в начале обхода `AST`. Также этот класс предоставляет важнейшую возможность наследоваться от него и переопределить методы, вызывающиеся при обработке нужных узлов `AST`. Например, можно переопределить метод VisitDecl, который вызывается при обходе каждого узла `AST`, являющегося декларацией. Таким образом, использование класса `RecursiveASTVisitor` заметно облегчило обход `AST` при реализации правил трансляции.

`SourceLocation` — это класс, который отвечает за местоположение того или иного объекта в исходном коде программы. Например, объект этого класса можно получить у заданной деларации и оператора. При нахождении в определенном узле AST во время обхода AST с помощью наследника класса `RecursiveASTVisitor` можно понять, какому месту в исходном коде программы соответствует этот узел. Таким образом, с помощью класса `SourceLocation` можно установить соответствие между исходным кодом и синтаксическими конструкциями в программе. Также очень важно, что при реализации правил трансляции с помощью класса `SourceLocation` можно запоминать участки кода, в которых были произведены изменения, что дает возможность создать протокол, по которому можно от конструкций в тексте C-kernel программы вернуться к конструкциям текста C-light программы. Такой протокол позволяет сообщить не только о том, что программа не корректна, но и указать в исходном тексте C-light программы на причины.

`Rewriter` — это класс, который позволяет изменять исходный код программы, загруженный в специальные буферы. При реализации правил трансляции все необходимые изменения в исходном коде программы осуществляются с помощью класса `Rewriter`. Класс `Rewriter` использует объекты классов `SourceLocation`, чтобы узнать, в каких местах исходного кода производить изменения. `Rewriter` позволяет осуществлять операции вставки, копирования, замены, т.е. все те операции, которые производятся во время работы с текстом программы при применении правил трансляции. После применения правил трансляции у объекта класса `Rewriter` можно запросить объект класса `RewriterBuffer`. Из объекта класса `RewriterBuffer` можно получить уже модифицированный исходный код.

Приведем пример реализации такого правила трансляции из [2]: всякая декларация

вида "*type_spec declarator_list*; в которой отсутствует явная спецификация класса памяти, заменяется на декларацию вида "*storage type_spec declaratory_list*; где *storage* — это, в зависимости от места самой декларации, либо ключевое слово `static`, либо ключевое слово `auto`. Часть исходного кода транслятора, отвечающая за реализацию данного правила трансляции, приведена в листинге №1:

```
...

class MyASTVisitor : public RecursiveASTVisitor<MyASTVisitor>
{
 public:
    MyASTVisitor(Rewriter &R): TheRewriter(R){}

    bool VisitStmt(Stmt *s)
    {
       if (isa<DeclStmt>(s))
       {
          DeclStmt *declStmt = cast<DeclStmt>(s);
          Decl *d = *(declStmt->decl_begin());

          if (isa<VarDecl>(d))
          {
             VarDecl *varDecl = cast<VarDecl>(d);

             if (varDecl->hasLocalStorage())
             {
                TheRewriter.InsertText(varDecl->getLocStart(),
                                       "auto ", true, true);
             }
          }
       }

       return true;
    }

private:
```

```
    void AddBraces(Stmt *s);
    Rewriter &TheRewriter;
};

...

class MyASTConsumer : public ASTConsumer
{
 public:
    MyASTConsumer(Rewriter &R): Visitor(R){}

    virtual bool HandleTopLevelDecl(DeclGroupRef DR)
    {
       for (DeclGroupRef::iterator b = DR.begin(), e = DR.end();
            b != e; ++b)
            Visitor.TraverseDecl(*b);
       return true;
    }

 private:
    MyASTVisitor Visitor;
};

...
```

В примере можно увидеть, что классы `ASTConsumer`, `RecursiveASTVisitor`, `SourceLocation` и `Rewriter` действительно очень удобно использовать для реализации правил трансляции.

3. Использование семантической разметки. 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 1: Роль под-формул условия корректности для верификации

На базе идеи Денни и Фишера [7] 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, строку и, возможно, колонку). 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 подформул. Базисная информация при генерации объяснения пред-

ставляет собой множество *концепций*, зависящее от того, какой аспект условий корректности необходимо объяснить. В табл. 1 дается краткий обзор концепций, предложенных к настоящему времени. Более подробный обзор можно найти в [15].

Мы используем нотацию из [7] 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. 1) описывает роль данного помеченного терма. Местоположение o отражает происхождение терма (номер строки 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 подформуле, группе формул или тройке Хоара. Обычно, метки над подформулами соответствуют гипотезам, заключениям и уточнителям. Группы формул и тройки помечаются индуктивными уточнителями.

Исходная логика Хоара для C-kernel была представлена в [3], а ее вариант с метками в [15]. Рассмотрим для примера правило для композиции:

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

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. [14]), 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 подформул, 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. 2 и включают в себя: (1) нормализацию условий корректности с использованием помеченной си-

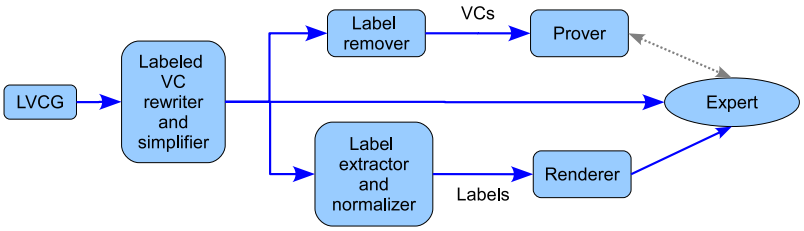

Рис. 2: Метки и помеченные условия корректности

стемы 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).

3.1. Пример. Рассмотрим следующую программу с внесенной ошибкой:

```
void NegateFirst(int ia[], int Length)
{
    //@ pre ...
    int i;
    //@ inv ...
    for (i = 0; i < Length; i++) {
        if (ia[i] < 0) {
            ia[i] = ia[i];
            break;
        }
    }
    //@ post ...
}
```

Оригинальная программа [14] должна была найти первый отрицательный элемент массива, изменить его знак и завершить работу. Как видно, в присваивании `ia[i] = -ia[i]` потерян знак "минус".

Спецификации, отражающие требуемое правильное поведение программы, выглядят как

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

$$\begin{aligned}\text{post}: \quad & \forall i.\ (0 \leq i \leq \mathrm{MD}(\texttt{Length}) \Longrightarrow \\ & ((\mathrm{MD}(\mathrm{MeM}(old, i)) < 0 \wedge \\ & (\forall j.\ 0 \leq j < i \Rightarrow \mathrm{MD}(\mathrm{MeM}(old, j)) \geq 0)) \Rightarrow\end{aligned}$$

$$\mathrm{MD}(\mathtt{MeM}(\mathtt{ia}, i)) = -\mathrm{MD}(\mathrm{MeM}(old, i)) \wedge$$
$$old[i] \geq 0 \Rightarrow \mathrm{MD}(\mathtt{MeM}(\mathtt{ia}, i)) = \mathrm{MD}(\mathrm{MeM}(old, i)))$$

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

Транслятор, описанный в разд. 2.2, порождает эквивалентную программу на C-kernel:

```
1   void NegateFirst(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  }
```

Номера строк не порождаются при трансляции, а добавлены для удобства.

Генератор условий корректности порождает 5 помеченых условий и одну тождественно истинную тройку Хоара. Четыре из условий автоматически доказываются в программе Simplify. Однако условие, соответствующее ветви *then* условного оператора, оказывается

ложным. Оно имеет вид

$$\left[\begin{array}{ll} \ulcorner \mathrm{inv}(\mathrm{MD} \leftarrow \mathrm{MD}_1) \urcorner^{\mathsf{asm\_inv}(6)} & \wedge \\ \ulcorner \mathrm{MD}_1(\mathrm{MeM}(\mathtt{i})) < \mathrm{MD}_1(\mathrm{MeM}(\mathtt{Length})) \urcorner^{\mathsf{while\_t}(6)} & \wedge \\ \ulcorner \mathrm{MD}_1(\mathrm{MeM}(\mathtt{ia}), \mathrm{MD}_1(\mathrm{MeM}(\mathtt{i}))) < 0 \urcorner^{\mathsf{then}(7)} & \wedge \\ \ulcorner \mathrm{MD} = \mathit{upd}(\mathrm{MD}_1, (\mathrm{MeM}(\mathtt{ia}), \mathrm{MD}_1(\mathrm{MeM}(\mathtt{i}))), & \\ \qquad \mathrm{MD}_1(\mathrm{MeM}(\mathtt{ia}), \mathrm{MD}_1(\mathrm{MeM}(\mathtt{i})))) \urcorner^{\mathsf{upd}(8)} & \\ \Rightarrow & \\ \ulcorner \mathrm{post} \urcorner^{\mathsf{ens\_inv}(9,\mathrm{L})} & \end{array}\right]^{\mathsf{pres\_inv}(6..10)}$$

В попытке его доказать мы заметим, что конъюнкт

$$\ulcorner \mathrm{MD} = \mathit{upd}(\mathrm{MD}_1, (\mathrm{MeM}(\mathtt{ia}), \mathrm{MD}_1(\mathrm{MeM}(\mathtt{i}))), \mathrm{MD}_1(\mathrm{MeM}(\mathtt{ia}), \mathrm{MD}_1(\mathrm{MeM}(\mathtt{i})))) \urcorner^{\mathsf{upd}(8)}$$

противоречит заключению. Метка `upd(8)` указывает, что проблема связана с обновлением элемента массива в строке 8. В свою очередь, связи, хранимые в объекте класса `SourceLocation` для данного узла `AST` (см. разд. 2.2), позволят указать и исходное присваивание в C-light программе.

Конечно, для этой тривиальной программы условие корректности и ложный конъюнкт оказываются достаточно обозримыми. Для реальных программ формулы значительно разрастаются. И здесь снова на выручку приходят семантические метки. Вместо анализа структуры условия корректности или его доказательства мы можем просто извлечь метки, нормализовать их и применить трансляционные шаблоны для порождения объяснения на естественном языке. В данном примере получается следующее:

```
This VC corresponds to lines 6-10 in the function "NegateFirst". Its
purpose is to contribute the loop invariant preservation on each
iteration.
Hence, given
  - assumption that loop invariant holds at line 6,
  - assumption that the loop condition holds at line 6,
  - assumption that "then"-branch is chosen at line 7,
  - substitution for MD
    originating in array update at line 8,
show that label invariant holds at line 9.
```

Пояснение для «подозрительной» подформулы выделено рамкой.

4. Родственные работы. Среди большого числа проектов по верификации Си-программ (подробный обзор в [14]) отметим два проекта, в которых также используется

трансляция в промежуточный язык. Во-первых, проект WHY/Frama-C [8], 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 [6]. 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 [9]. 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 [7] 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. [12] 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 генратором.

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

Также отметим некоторые работы по трансляции Си-программ. В [5] описана система `ctt` (Code Transformation Tool), предназначенная для трансформации Си-программ с целью распараллеливания и оптимизации. В [11] описана программа Linsert, транслирующая Си-программу в эквивалентную программу без побочных эффектов. Заметим, что хотя сами алгоритмы перевода имеют формальную основу, доказательства эквивалентности перевода отсутствуют (либо просто базируются на интуитивной трактовке стандарта). Более формальное обоснование предложено в [10], но оно связано не с формальной семантикой Си, а с сохранением тестовых покрытий.

5. Заключение. Дедуктивная верификация программ была задумана как способ достоверного доказательства корректности программы в отличие от тестирования, которое

имеет более вероятностный характер. Закономерно возникает вопрос ее надежности. И если теоретические основания верификации давно обоснованы в классических работах, то на практике все упирается в корректность реализации системы верификации. Очевидно, что идеальным решением было бы написать систему на целевом языке и проверить ее своими же средствами, т.е. получить «верифицированный верификатор». Для такого сложного языка, как Си, это нетривиальная задача. Основных препятствий здесь два.

Во-первых, любая осмысленная Си-программа использует стандартную библиотеку, которая не только не верифицирована, но даже не обладает полной формальной спецификацией. Шаги в этом направлении стали предприниматься только в последние годы. Помимо экспериментов в проекте WHY/Frama-C, не представленных в литературе, работа ведется и в рамках проекта C-light [16].

Во-вторых, формальная база дедуктивной верификации относится только к порождению условий корректности и их доказательству. Процессы же интерпретации результатов верификации пользователем (в особенности отрицательных результатов) и локализации ошибок формализованы слабо. Данная статья представляет обзор наших разработок на пути решения этой задачи. В соответствии с двухуровневой схемой верификации в проекте C-light работа ведется по двум направлениям.

Первое направление касается реализации транслятора из C-light в C-kernel. В частности, используется новый инструментарий. При этом для той части транслятора, которая выразима на языке C-light, будут предложены ACSL-спецификации, что позволит в будущем перейти к его верификации. Также для задачи локализации ошибок реализуется возможность обратной трансляции из C-kernel в C-light, что отсутствовало в предыдущих прототипах транслятора. Решение об использование инструментария Clang для этой цели оказалось оправданным. Оно позволило существенно сократить усилия при разработке транслятора. Поэтому можно рекомендовать при решении сходных задач трансляции применять инструментарий Clang.

Что касается второго этапа, то в аксиоматическую семантику языка C-kernel введено расширение в виде семантических меток. Структурированные метки отражают роли подтермов условий корректности в контексте верификации, а также аккумулируют в себе информацию об исходных фрагментах программы. Метки играют важную роль в процессе упрощения условий корректности, но основное их предназначение — это сохранение информации об исходных фрагментах программы в автоматическом доказателе и генерация объяснений для условий корректности на естественном языке.

Список литературы

1. Влиссидес Д., Гамма Э., Джонсон Р., Хелм Р. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2012. — 368 с.
2. Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В. На пути к верификации C программ. Часть 3. Перевод из языка C-light в язык C-light-kernel и его формальное обоснование. — Новосибирск, 2002. — 82 с. — (Препр. / РАН. Сиб. Отдние. ИСИ; N 97).
3. Промский А.В. Формальная семантика C-light программ и их верификация методом Хоара // Диссертация на соискание ученой степени кандидата физико-математических наук. — Новосибирск, 2004. — 157 с.
4. Baudin P., Filliâtre J.C., Marché C., Monate B., Moy Y., Prevosto V. ACSL: ANSI/ISO C Specification Language. [Электронный ресурс]. URL: `http://www.frama-c.cea.fr/download/acsl_1.4.pdf`
5. Boekhold M., Karkowski I., Corporaal H., Cilio A. A programmable ANSI C code transformation engine. — Delft, 1998. — (Tech. Rep. / Delft University of Technology; N 1-68340-44(1998)-08).
6. 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.
7. Denney E., Fischer B. Explaining Verification Conditions // Proc. AMAST 2008. — LNCS. — 2008. — Vol. 5140. — P. 145–159.
8. Filliâtre J.C., Marché C. Multi-prover verification of C programs // Proc. ICFEM 2004. — LNCS. — 2004. — Vol. 3308. — P. 15–29.
9. Fraer R. Tracing the origins of verification conditions. — Rocquencourt, 1996. — 17 p. — (Rapp. / INRIA; N 2840).
10. Harman M., Hu L., Hierons R., Wegener J., Sthamer H., Baresel A., Roper M. Testability transformation // IEEE Trans Software Eng. — 2004. — N 30(1). — P. 3–16.
11. Harman M., Hu L., Munro M., Zhang X.Side-effect removal transformation // Proc. IEEE Intern. Workshop Program Comprehension. — IEEE Computer Society Press, 2001. — P. 310–319.
12. 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.
13. Programming languages — C: ISO/IEC 9899:1999. — 1999. — 566 p.
14. Promsky A.V. Towards C-light program verification: Overcoming the obstacles // Proc.

PU-2009, 19–23 June, Altai Mountains, Russia, 2009. — P. 53–63.

15. Promsky A.V. Error-tracing semantics for C-kernel // Bull. Nov. Comp. Center, Comp. Science, 31 (2010). P. 123–138.
16. A. Promsky. C-light Program Verification: Error Tracing and Library Specification // Proc. Second Workshop "Program Semantics, Specification and Verification: Theory and Applications". — St. Petersburg, Russia, June 12–13, 2011. — P. 83–92.
17. Promsky A.V. Verification Condition Understanding // Proc. Ershov Informatics Conference (PSI Series, 8-th Edition). — June 27 – July 1, 2011, Akademgorodok, Novosibirsk, Russia. — P. 295–300.
18. Promsky A.V. A formal approach to the error localization // Preprint 169, A. P. Ershov Institute of Informatics Systems, Novosibirsk, Russia, 2012.

UDK: 519.681

Title: An integrated approach to the error localization during C program verification

Author(s):

Dmitry Alexandrovich Kondratyev (Novosibirsk State University),

Alexey Vladimirovich Promsky (A.P. Ershov Institute of Informatics Systems)

Abstract: Maximal employment of formal and sound methods is one of the distinctive features of C-light project. This concerns not only the fundamental verification bases, like Plotkin's operational semantics or Hoare's logics, but also implementation aspects. The localization of possible errors is one of them. In the majority of known verification systems such localization is simply coded by developers as a part of system functionality without any formal basis for it. The recent reinforcement of the C-light project by the semantical labeling technique and the choice of LLVM/Clang for the system input block allowed us to obtain some new results, which are surveyed in this paper.

Keywords: semantics, specification, verification, translation, error localization, labeling, the C language, C-light, LLVM, Clang