Bull. Nov. Comp. Center, Comp. Science, 31 (2010), 123–138
© 2010 NCC Publisher

Error-tracing axiomatic semantics for C-kernel*

A. V. Promsky

Abstract. The classical Hoare logic links separate verification conditions (VCs) to linear paths of a program. The real verification condition generators (VCG) link VCs to line numbers at best. It can be insufficient, since VCs contain information neither about the evaluation order nor about correspondence between their fragments and specific operations. Verification of complex programs is inevitably confronted by difficulties of VC interpretation and error localization. This paper describes an extension of axiomatic semantics of the C-kernel language. The extended calculus will be able to derive VCs accompanied with formal derivation protocols useful in VC understanding and error tracing.

1. Introduction

The C programming language still holds the top positions in the popularity rating of programming languages[1]. Thus the formal verification of C programs is an actual problem.

The C program verification project, which is being developed in the Theoretical programming laboratory of IIS, has a long history. The details can be found in [6, 7, 10, 8]. The input language, known as C-light, represents a considerable subset of the standard C99. The formal definition of C-light was developed by means of the structural operational semantics. We also have chosen a restricted core of C-light, known as C-kernel, and proposed a sound axiomatic semantics for it. So, the verification process in our project consists of two stages. On the first step, an annotated C-light program is translated into an equivalent C-kernel program by means of formal rewriting rules. Then, the axiomatic semantics is used to derive the VCs which must be proved to establish correctness of the original C-light program. Two main advantages of our approach can be mentioned. First, the two-stage technique allows us to overcome many obstacles originating in the C low-level nature. Second, correctness of each stage is formally proved.

Since the theoretical part of our approach is well-developed, we can concentrate on some practical problems. One of them concerns the error localization and interpretation of VCs. Usually, the Hoare axiomatic method

*Partially supported by Lavrentiev's grant "C program verification with efficient error localization" from SB RAS.

[1]According to `http://www.tiobe.com/index.php/content/paperinfo/tpci/` it competes with Java for the 1st place.

compares VCs to linear paths in a program. The falsity of a condition gives warning of an error on the corresponding path (or in its specification), but does not reveal the exact position of a bug. In addition, when we deal with a language like C, the complexity of verification conditions grows considerably. The automatic theorem provers can smooth over this problem during the condition proof, but if a user tries to analyze such a condition, he can reach a deadlock.

In order to overcome this obstacle, we applied a method presented in paper [2] by Denney and Fischer. The idea consists in a systematic extension of the Hoare rules by labels so that the calculus itself can be used to build up explanations and traces of VCs. The purpose of this paper is to give an overview of the Hoare logic of C-kernel enriched by the labeling technique. The algorithms which render the labels according to various aspects of VC analysis will also be discussed.

The rest of the paper is organized as follows: Section 2 briefly describes the C-kernel language. An overview of the extended Hoare logic of C-kernel is given in Section 3 together with label rendering methods. An example in Section 4 illustrates this technique. Section 5 presents an overview of related projects. Section 6 is a conclusion of the paper.

2. The C-kernel language

The C-kernel language imposes a set of severe restrictions on C-light. So, let us briefly remind what is the C-light language[2]. Ultimately, the restrictions imposed by C-light on the standard C99 are divided into two groups. First, we avoid the constructions whose semantics significantly relies on low-level implementation information. For example, those include bit-fields and unions. Second, we deliberately fix some undefined aspects of the C language. For example, expressions are evaluated in the “right-to-left” order and side effects take place immediately. We also restrict the use of some legal C-light constructions avoiding programmers’ tricks, such as flexible array members or `goto`s into blocks.

Declarations. The declarator lists in C-kernel are allowed in function declarations only. Every other declaration introduces a single object.

The initializers of composite objects must be in a fully bracketed form.

The storage specifiers `static` and `auto` are mandatory.

Expressions. Every C-kernel expression different from a function call implies at most one memory modification, i.e. only the function calls contain multiple side effects.

The legal C-kernel operations are the following:

[2]Details can be found in [6, 9]

1) primary:	`()` `[]` `.` `->`
2) unary:	`&` `*` `-` `sizeof`
3) binary:	`*` `/` `%` `+` `-` `<` `>` `<=` `>=` `==` `!=`
4) assignment:	`=`
5) memory handling:	`new` `delete`
7) type casts.	

We traditionally use C++ memory operations in order to avoid references to the standard library when semantics is discussed.

The base C-kernel expressions are *normalized* ones. They are built by usual C rules involving only primary, unary, binary and type cast operations. Thus, no memory change can take place in a normalized expression.

Let *CV* stand for constants and variable identifiers. Then, the syntax of an arbitrary C-kernel expression looks like:

Expr ::= *f*`(`*CV*`,...,` *CV*`)` | *NExpr* `=` *f*`(`*CV*`,...,` *CV*`)` |
NExpr `=` `new` *Type* | *NExpr* `=` `new` *Type* `[`*NExpr*`]` |
`delete` *NExpr* | `delete []` *NExpr* |
NExpr `=` *NExpr*

Statements. The legal C-kernel statements are as follows:

Stat ::= *Expr*`;` |
l`:` *Stat* | `goto` *l*`;` | `return;` | `return` *NExpr*`;` |
`if (`*NExpr*`)` *Stat* `else` *Stat* | `while (`*NExpr*`)` *Stat* |
`{`*Stat* ... *Stat*`}`

As it was said above, the C-kernel language is an intermediate one, and the user does not write in it explicitly. Instead, an input C-light program is automatically translated into C-kernel. Let us consider an example which will also be used in Section 4. The annotated C-light program looks like

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

The annotations are irrelevant here, so we use abbreviations. The corresponding C-kernel program is as follows:

```
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  }
```

In fact, the translation process does not add line numbers. Nevertheless, we will need them later. Note that the `for` statement is replaced by the `while` statement, the `break` statement is replaced by `goto`, and the postfix increment `i++` requires an additional pointer `q1`.

3. An extended Hoare logic for C-kernel

In theory, program verification is quite an easy process: a verification condition generator (VCG) takes a program that is "marked-up" with logical annotations and produces a number of VCs that are simplified, augmented with a domain theory, and finally discharged by a theorem prover. In practice, however, many things can go wrong: the program may be incorrect or unsafe, the annotations may be incorrect or incomplete, the simplifier may be too weak, the domain theory may be incomplete, and the prover may run out of resources. In each of these cases, users are typically confronted only with failed VCs (i.e., the failure to prove them automatically) but receive no additional information about the causes of the failure. They must thus analyze the VCs, interpret their constituent parts, and relate them through the applied Hoare rules and simplifications to the corresponding source code locations. This is often difficult to achieve.

Using the idea from [2], we will extend the Hoare rules for the C-kernel language [7, 9] by "semantic mark-up" so that we can use the calculus itself to build up explanations of the VCs. This mark-up takes the form of structured labels that are attached to the formulas in the Hoare rules. The labels are maintained through different processing steps, in particular simplification, and are then extracted from the final VCs and rendered as natural language explanations.

3.1. Preliminaries

Here we should remind important information about the underlying abstract machine of C-light and general structure of Hoare rules.

Instead of the classical program state, which maps the program variable names onto their possible values, we define a state as a map over meta-variables. In turn, the meta-variables are higher level objects (functions and Cartesian products) which model the abstract memory. Formally a *state* of C-light abstract machine is a map over the following meta-variables: (1) the metavariable MeM which maps the names of usual program variables onto their memory addresses; (2) the metavariable MD which maps the program object addresses onto their values; (3) the mapping Γ which retrieves the types of program objects from their addresses; (4) the metavariable STD which handles the `typedef`s and structure tags; (5) the metavariable Val which contains the value of the last evaluated expression.

Such a complex state is a for low-level nature of the C language. It allows us to handle local variable redefinitions or aliasing.

A typical Hoare triple for C-kernel looks like

$$\mathcal{E}nv \Vdash \{P\}\ S\ \{Q\}\ \ ,$$

where P and Q are (labeled) pre- and postcondition, respectively, S is a legal C-kernel program, and *environment* $\mathcal{E}nv$ is a triple $\langle cf, nl, IAx\rangle$. Here, cf is the name of a current function (it will be used in semantics of `return`). A nonnegative integer nl corresponds to a current *nesting level.* Finally, a set of Hoare triples IAx denotes the inductive hypotheses (the semantics of `goto` and function calls relies greatly on it). We will use the superscripts $\mathcal{E}nv^i$ to access the environment components.

Note. Usually, a program S is represented as source strings, not a syntax tree. The line numbers are retrieved by a VCG that implements the Hoare logic. For clarity, we also use a traditional notation. As a result, it may seem that the line numbers appear magically in the semantic labels.

3.2. Concepts and labels

Here we focus on error localization and VC understanding. While the error tracing idea is obvious (file names, line numbers and, ideally, source columns), the explanation of a VC requires understanding of the role it can play. The first approximation appears if we recall that, after simplification, VCs usually look like Horn clauses, i.e. $H_1 \wedge \ldots \wedge H_n \Rightarrow C$. Here, the unique conclusion C of the VC can be considered as its *purpose*. However, for a meaningful explanation of the VC *structure*, we need a more detailed characterization of the sub-formulas.

Concepts. The basic information for explanation generation is a set of underlying concepts, which depends on the particular aspect of the VCs to be explained.

Hypotheses consist of assertions and control flow predicates. *Assertions* include pre- and post-conditions (labels asm_pre and asm_post), function pre- and post-conditions (asm_fpre and asm_fpost), and loop invariants. Since a loop invariant serves as a hypothesis in two different positions, we distinguish asm_inv and ass_inv_exit. *Control flow predicates* reflect the program control flow. For both `if` statement and `while` loop, the controlling expressions occurring in a program are required in both their original and negated forms, so that we get four different concepts: then, else, while_t, and while_f.

Conclusions capture the primary purpose of a VC, which includes ensuring that different types of assertions hold at given locations. As in the case of hypotheses, invariants are used in two different forms, the *entry form* (or base case) ens_inv and the *step form* ens_inv_iter.

Qualifiers further characterize hypotheses and conclusions by recording how a sub-formula was produced. Different *substitution* concepts reflect substitutions of the underlying Hoare calculus. The *assignment* concept sub captures the origin and the effect of assignments and array updates on the form of the resulting VCs.

Contributors capture the secondary purpose of a VC; this arises when a recursive call of VCG (applied to a nested program structure) produces VCs that are conceptually connected to the purpose of the larger structure. For example, all VCs emerging from the premise $\{I \wedge e\}\ S\ \{I\}$ of the classical *while*-rule contribute to the proof of preservation of the invariant I over the loop body S independent of their primary purpose.

Note. In our project, we have chosen ACSL as a specification language. That is why, comparing with [2], we use the words "assumes" and "ensures" in the concepts instead of "asserts" and "establishes".

Labels. We will use notation from [2] to derive labeled terms $\ulcorner t \urcorner^{l}$, where each term t can be marked with a label l. Labels will have the form $c(o, n)$. Here the concept c describes the role the labeled term plays and thus determines how it is rendered. The location o records where it originates; it refers either to an individual line number or to a range. The optional list of labels n nested inside contains further qualifying information which applies either directly to the top-level term, or has been extracted from sub-terms during normalization and extraction.

Note. This notion of a label can be confusing when a usual C label arises. It was not a problem in [2], where a language without labels was examined. We hope that a reader can distinguish between these semantic labelings and program labels depending on the context.

3.3. Extended Hoare rules

When the deductive verification is studied, an interesting question arises. Most Hoare systems presented in papers do not contain any rule for a program as a whole (which is called a *compilation unit* in C). In rare cases, simple programs à la Pascal were examined. The reason is obvious: if we do not model an operating system, it is not clear what should be a Hoare triple for the entire program. In fact, such rules are hidden in VCGs implementing the corresponding Hoare systems.

However, in the context of VCs explanation such a semi-formal rule deserves our attention. Let F denote a function definition. Let D stand for an arbitrary external nonfunctional declaration. A C-kernel program is a sequence of such F_i and D_j accompanied by the function `main`:

$$\mathcal{P}(D_1, ..., D_n, F_1, ..., F_m, \texttt{main}) \ .$$

We assume that subscripts of the declarations D_i correspond to their relative positions within the program. Let Hyp stand for the following set:

$$\Big\{ \{pre(F_i)\}\ name(F_i)(\overline{v_i})\ \{post(F_i)\} \ \Big|\ i = 1, ..., m \Big\} \ ,$$

where $\overline{v_i}$ is the parameter list of F_i. Then the starting rule looks like

$$\frac{\begin{array}{c} (name(F_1), 1, Hyp) \Vdash \{\ulcorner pre(F_1) \urcorner^{\mathsf{asm\_pre}}\}\ body(F_1)\ \{\ulcorner post(F_1) \urcorner^{\mathsf{ens\_post}}\} \\ \ldots \\ (name(F_m), 1, Hyp) \Vdash \{\ulcorner pre(F_m) \urcorner^{\mathsf{asm\_pre}}\}\ body(F_m)\ \{\ulcorner post(F_m) \urcorner^{\mathsf{ens\_post}}\} \\ (\texttt{main}, 1, Hyp) \Vdash \{\ulcorner pre(\texttt{main})\}\ body(\texttt{main})\ \{\ulcorner post(\texttt{main}) \urcorner^{\mathsf{ens\_post}}\} \\ (\emptyset, 0, \emptyset) \Vdash \{\mathsf{true}\}\ D_1\ ...\ D_n\ \{\ulcorner pre(\texttt{main}) \urcorner^{\mathsf{ens\_pre}}\} \end{array}}{\mathcal{P}(D_1, ..., D_n, F_1, ..., F_m, \texttt{main})}$$

Thus, this rule reduces verification of the entire program to verification of its separate functions and forms the set of inductive hypotheses Hyp to handle the function calls. Note that verification of each function starts with the nesting level equal to one. The last Hoare triple in the premise guarantees that execution of external declarations D_i precedes the execution of `main`.

It is the first time when semantic labeling appears in a rule. The labels reflect the purpose of each sub-formula: pre-conditions are assumed to hold at the beginning and post-condition must be ensured.

Consequence rule. The extended version of this standard rule looks like:

$$\frac{\ulcorner P\urcorner^{\mathsf{asm\_pre}} \Rightarrow P_1 \qquad \mathcal{E}nv \Vdash \{P_1\}\ S\ \{\ulcorner Q_1\urcorner^{\mathsf{ens\_post}}\} \quad \ulcorner Q_1\urcorner^{\mathsf{asm\_post}} \Rightarrow Q}{\mathcal{E}nv \Vdash \{\ulcorner P\urcorner^{\mathsf{ens\_pre}}\}\ S\ \{Q\}} \ .$$

As you can see, we must ensure that Q_1 is an intermediate post-condition and also serves as an assumption for Q.

Declaration statement. In order to avoid unnecessary multiplication of rules, we use a generic Hoare rule, where a special function $\mathcal{HD}ec$ performs the case analysis:

$$\mathcal{E}nv \Vdash \{\mathcal{HD}ec(Q, Decl)\}\ Decl\ \{Q\} \ .$$

The volume of the paper does not allow us to show $\mathcal{HD}ec$ for every legal C-kernel declaration (see [7, 9] for details). Let us consider the declaration of a global integer variable and the declaration of a local array with initialization:

$$\begin{aligned}
&\mathcal{HD}ec(Q, \mathsf{static\ int}\ v\,\texttt{;}) = \\
&\qquad\qquad Q\ (\mathrm{MeM} \leftarrow \ulcorner upd(\mathrm{MeM}, (v, \mathcal{E}nv^2), nc)\urcorner^{\mathsf{alloc}}) \\
&\qquad\qquad\ \ (\mathrm{MD} \leftarrow \ulcorner upd(\mathrm{MD}, nc, 0)\urcorner^{\mathsf{init}}) \\
&\qquad\qquad\ \ (\Gamma \leftarrow upd(\Gamma, nc, \texttt{int}))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{HD}ec(Q,\ T\ a\texttt{[n] = \{}\ l_0, ..., l_k\texttt{\}}) = \\
&\qquad\qquad Q\ (\mathrm{MeM} \leftarrow \ulcorner upd(\mathrm{MeM}, \mathcal{E}nv^2, a, nc)\urcorner^{\mathsf{alloc}}) \\
&\qquad\qquad\ \ \ldots(\mathrm{MD} \leftarrow \ulcorner upd(\mathrm{MD}, (nc, i), l_i)\urcorner^{\mathsf{init}})\ \ldots \\
&\qquad\qquad\ \ \ldots(\mathrm{MD} \leftarrow \ulcorner upd(\mathrm{MD}, (nc, j), \omega)\urcorner^{\mathsf{init}})\ \ldots \\
&\qquad\qquad\ \ (\Gamma \leftarrow \mathrm{upd}(\Gamma, (nc, l), T))
\end{aligned}$$

where $0 \le i \le k$, $k+1 \le j \le n-1$.

So, when the control flow reaches a static variable declaration, the meta-variable MeM allocates a new address nc for the variable v. The meta-variable MD assigns the default zero to this new object and meta-variable Γ sets its type to `int`. In the case of an array, in addition to nc we have a set of offset locations (nc, Δ). Depending on initializer, some elements obtain the initial values l_i and some elements remain undefined (ω).

Expression statement. By analogy with declarations, we use a universal function (here, $\mathcal{U}pd$) to combine all legal expressions (except for function calls) in a single rule:

$$\mathcal{E}nv \Vdash \{\mathcal{U}pd(Q, Expn)\}\ Expr\ \{Q\} \ .$$

A fragment of $\mathcal{U}pd$ definition is as follows:

$\mathcal{U}pd(Q, a\texttt{[}i\texttt{]}\ \texttt{=}\ rval\texttt{;}) =$
$\quad Q\,(\mathrm{MD} \leftarrow \ulcorner upd(\mathrm{MD},\ (\mathrm{MeM}(a, \mathcal{E}nv^2), i), rval)^{\urcorner \mathsf{upd}});$

$\mathcal{U}pd(Q, lval\ \texttt{=}\ rval\texttt{;}) =$
$\quad Q\,(\mathrm{MD} \leftarrow \ulcorner upd(\mathrm{MD},\ (\mathrm{MeM}(lval, \mathcal{E}nv^2), 0), rval)^{\urcorner \mathsf{asgn}});$

$\mathcal{U}pd(Q, lval\ \texttt{=}\ \texttt{new}\ T\texttt{;}) =$
$\quad Q\,(\mathrm{MD} \leftarrow \ulcorner upd(\mathrm{MD},\ (\mathrm{MeM}(lval, \mathcal{E}nv^2)), nc)^{\urcorner \mathsf{alloc}})(\Gamma \leftarrow upd(\Gamma, nc, T));$

$\mathcal{U}pd(Q,\ \texttt{delete}\ ptr\texttt{;}) = Q(\mathrm{MD} \leftarrow \ulcorner upd(\mathrm{MD},\ \mathrm{MeM}(ptr, \mathcal{E}nv^2), \omega)^{\urcorner \mathsf{free}}).$

Note that the label upd signals about the array update, not about the meta-variable MD modification.

We use a separate rule, when the result of a function call is assigned to a variable. Let $\overline{x}$ stand for the formal parameter list of f and $\overline{e}$ denote an actual argument list. Given that z is a fresh name (i.e., not occurring in the program and specifications), the rule looks like[3]

$$\frac{\ulcorner P^{\urcorner \mathsf{asm\_pre}} \Rightarrow \ulcorner (\ulcorner P'\alpha \wedge (Q'\gamma(\mathrm{Val} \leftarrow z))^{\urcorner \mathsf{ens\_specs}} \Rightarrow Q\gamma\beta\delta)^{\urcorner \mathsf{call}}}{\mathcal{E}nv \Vdash \{P\}\ lval\ \texttt{=}\ f(\overline{e});\ \{Q\}}\ ,$$

provided that for some P' and Q' $\{P'\}\ f(\overline{x})\ \{Q'\} \in \mathcal{E}nv^3$. The substitutions $\alpha, \beta, \gamma, \delta$ are as follows:
$\alpha = (\mathrm{MeM} \leftarrow SI(\mathrm{MeM}, \mathrm{MD}, \mathcal{E}nv^2, \overline{x}));$
$\beta = (\mathrm{MeM}\gamma \leftarrow \mathrm{MeM});$
$\delta = (\mathrm{MD} \leftarrow \mathrm{upd}(\mathrm{MD}, \mathrm{MeM}(lval, \mathcal{E}nv^2), z));$
γ changes all logical variables from P' and Q' with fresh names.

The function SI (Stack Initialization, see [9]) creates a new scope for the function parameters temporarily forbidding access to other local objects. The variable renaming allows us to avoid universal quantification on the local level.

The C language does not distinguish between functions and procedures. The procedures are functions returning `void`. Thus, the only difference in the following rule is that no substitution into Val takes place:

$$\frac{\ulcorner P^{\urcorner \mathsf{asm\_pre}} \Rightarrow \ulcorner (\ulcorner P'\alpha \wedge Q'\gamma^{\urcorner \mathsf{ens\_specs}} \Rightarrow Q\gamma\beta)^{\urcorner \mathsf{call}}}{\mathcal{E}nv \Vdash \{P\}\ f(\overline{e});\ \{Q\}}\ ,$$

provided that for some P' and Q' $\{P'\}\ f(\overline{x})\ \{Q'\} \in \mathcal{E}nv^3$.

[3]Let us restrict the presentation with a simple assignment case.

Composition. The classical Hoare rule for composition turns unsound in the presence of jumps. To avoid this, the composing parts should be restricted. Thus, in the following rule, we explicitly assume that neither S_1 nor S_2 contains labeled statements on the uppermost nesting level:

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

Considering that jumps into blocks are forbidden in C-light, this requirement guarantees that no jump from S_1 into S_2 or from S_2 into S_1 can take place. Of course, even a single label in the C-kernel program will make this rule useless. As we will see later, the Hoare rule for labeled statements will provide its successful applicability.

Compound statement. The rule for blocks should accurately model the corresponding stack manipulations:

$$\frac{\mathcal{E}nv(nl \leftarrow nl + 1) \Vdash \{P\}\ \mathit{Statements}\ \{Q'\}}{\mathcal{E}nv \Vdash \{P\}\ \{\mathit{Statements}\}\ \{Q\}}\ , \qquad (**)$$

where $Q' = Q(\mathrm{MeM} \leftarrow \mathit{Reduce}(\mathrm{MeM}, n))(\Gamma \leftarrow \mathit{Reduce}(\Gamma, n))$. The function *Reduce* guarantees that all local objects become inaccessible when we leave a block [9]. Except for forming a nesting scope, the compound statement does not change the control flow at all. Neither does it involve any exterior logical assertions. Thus, no semantic labeling is required.

Labels. As we already mentioned, the restrictions in the rule $(*)$ guarantee the absence of interference between S_1 and S_2. On the other hand, it also means that we cannot prove a Hoare triple unless all labels are found and excluded. Fortunately, since jumps into blocks are banned in C-light, we do not need to look for all program labels. Given a statement sequence, it is sufficient to handle all labels on the uppermost nesting level of this sequence. The following rule performs this task:

$$\frac{\begin{array}{c}\mathcal{E}nv_1 \Vdash \{P\}\ S_0\ \{\ulcorner I_1 \urcorner^{\mathsf{ens\_inv}}\} \qquad \mathcal{E}nv_1 \Vdash \{\ulcorner I_1 \urcorner^{\mathsf{asm\_inv}}\}\ S_1\ \{\ulcorner I_2 \urcorner^{\mathsf{ens\_inv}}\} \\ \dots \\ \mathcal{E}nv_1 \Vdash \{\ulcorner I_n \urcorner^{\mathsf{asm\_inv}}\}\ S_n\ \{R\} \qquad \mathcal{E}nv_1 \Vdash \{R\}\ S_{n+1}\ \{Q\}\end{array}}{\mathcal{E}nv \Vdash \{P\}\ S_0 \quad l_1 \colon S_1\ \dots \quad l_n \colon S_n \quad S_{n+1}\ \{Q\}}\ ,$$

where $\mathcal{E}nv_1^3 = \mathcal{E}nv^3 \cup \Big\{ \big(\{I_i\}\ \mathtt{goto}\ l_i;\ \{\mathsf{false}\}, \mathcal{E}nv^2\big)\ \Big|\ i = 1, \dots, n \Big\}$ and the nesting level of every l_i is equal to $\mathcal{E}nv^2$. Thus, we assume that for every label l_i there exists an assertion I_i which holds whenever control reaches l_i. The set of inductive hypotheses of the form $\{I_i\}$ `goto` l_i; $\{\mathsf{false}\}$ is used to handle the corresponding `goto`s.

Conditional statement. Among other things, the rule should reflect that, in C, the `if` statement forms a scope, so the nesting level is incremented:

$$\frac{\begin{array}{c}\mathcal{E}nv(nl \leftarrow nl+1) \Vdash \{P \wedge \lceil \mathcal{E}val(e)\rceil^{\mathsf{then}}\}\ S_1\ \{Q\} \\ \mathcal{E}nv(nl \leftarrow nl+1) \Vdash \{P \wedge \lceil \neg\mathcal{E}val(e)\rceil^{\mathsf{else}}\}\ S_2\ \{Q\}\end{array}}{\mathcal{E}nv \Vdash \{P\}\ \texttt{if}\ (e)\ S_1\ \texttt{else}\ S_2\ \{Q\}}\ .$$

The evaluating function $\mathcal{E}val$ is defined in [7] by induction over the structure of the expression e. For example, if e is a variable name `x` and the declaration of `x` belongs to the nesting level m, then $\mathcal{E}val(e) = \mathrm{MD}(\mathrm{MeM}(\texttt{x}, m))$.

Loops. The semantics of `while` is defined using an intermediate form:

$$\frac{\mathcal{E}nv \Vdash \{P\}\ \texttt{\{}\ \mathbf{loop}(e, S)\ \texttt{\}}\ \{Q\}}{\mathcal{E}nv \Vdash \{P\}\ \texttt{while}\ \texttt{(}e\texttt{)}\ \texttt{\{}\ S\ \texttt{\}}\ \{Q\}}\ .$$

Conceptually, **loop** means the same: "do something while a condition is true". However, it does not form a scope, whereas the `while` statement does. Thus, we avoid unnecessary complication in the rule[4]. In turn, the semantic of **loop** is based on a classical Hoare rule:

$$\frac{\mathcal{E}nv \Vdash \lceil\{\lceil I\rceil^{\mathsf{asm\_inv}} \wedge \lceil\ \mathcal{E}val(e)\rceil^{\mathsf{while\_t}}\}\ S\ \{\lceil I\rceil^{\mathsf{ens\_inv\_iter}}\}\rceil^{\mathsf{pres\_inv}}}{\mathcal{E}nv \Vdash \{\lceil I\rceil^{\mathsf{ens\_inv}}\}\ \mathbf{loop}(e, S)\ \{\lceil I\rceil^{\mathsf{asm\_inv\_exit}} \wedge \lceil\neg\mathcal{E}val(e)\rceil^{\mathsf{while\_f}}\}}\ .$$

The labeling reflects the rule meaning. The invariant should be ensured at the loop entry and is thus labeled with ens_inv. Individual sub-formulas of both the exit-condition $I \wedge \neg\mathcal{E}val(e)$ and the step-condition $I \wedge \mathcal{E}val(e)$ are labeled appropriately. In the triple of the premise, the incoming postcondition I is labeled with its purpose (i.e., the invariant is reinsured to hold after one loop iteration). Finally, the purpose of all VCs emerging from the loop body is to contribute to invariant preservation. That is why we labeled the entire triple with pres_inv.

Jumps. As long as the inductive hypotheses for program labels are gathered by the rule above, the Hoare rule for the goto statement is straightforward:

$$\mathcal{E}nv \Vdash \{\lceil BR(I, m, \mathcal{E}nv^2)\rceil^{\mathsf{ens\_inv}}\}\ \texttt{goto}\ l\texttt{;}\ \{\mathsf{false}\}\ ,$$

provided that for some assertion I and nesting level m

$$\big(\{I\}\ \texttt{goto}\ l\texttt{;}\ \{\mathsf{false}\}, m\big) \in \mathcal{E}nv^3\ .$$

Function BR (*BigReduce*) is a generalized form of *Reduce* mentioned in (∗∗). Obviously, performing a jump, we can leave several nesting blocks. The successive application of *Reduce* is implemented by definition:

[4]Note the explicit braces in the premise.

$$BE(\Phi, m, n) = \begin{cases} \Phi, & \text{if } m = n, \\ BR(\Phi(\mathit{Reduce}(\text{MeM}, n)/\text{MeM}, \mathit{Reduce}(\Gamma, n)/\Gamma), m, n-1), & \text{if } m < n. \end{cases}$$

By analogy, we can formalize the `return` statement:

$$\mathcal{E}nv \Vdash \{\ulcorner BR(Q(\text{Val} \leftarrow \mathcal{E}val(e)), 1, \mathcal{E}nv^2)\urcorner^{\mathsf{ens\_post}}\}\ \texttt{return}\ e;\ \{\mathsf{false}\}\ ,$$

provided that for some assertion Q $\{Q\}$ `return;` $\{\mathsf{false}\} \in \mathcal{E}nv^3$. When nothing is returned, the substitution into Val is empty.

3.4. Rewriting and rendering

VCs (labeled or unlabeled) become complex and need to be simplified aggressively before they can be proven. The purpose of the rewriting stage is to remove redundant labels, to minimize the scope of the remaining labels, and to keep enough labels to explain any unexpected failures, based on the assumption that the majority of VCs can be rewritten to true.

We use the auxiliary functions $|\cdot|$ to remove labels from terms, and $[\![\cdot]\!]$ to extract the labels of a term. $[\![\cdot]\!]$ is defined by

$$\begin{array}{lcl} [\![\ulcorner f(t_1, ..., t_n)\urcorner^l]\!] & = & l \otimes ([\![t_1]\!] \oplus ... \oplus [\![t_n]\!]) \\ [\![f(t_1, ..., t_n)]\!] & = & [\![t_1]\!] \oplus ... \oplus [\![t_n]\!] \end{array}$$

where $\oplus$ is list concatenation and the label composition operator $\otimes$ appends the inner labels to the list of labels nested in the outer label, i.e., $c(o, n) \otimes l = c(o, n \oplus l)$.

As in [2], the rewriting rules fall into several different groups. The first group contains rules such as $\ulcorner\mathsf{true}\urcorner^l \rightarrow \mathsf{true}$ that remove labels from the trivially true sub-formulas, because these require no explanations. The next group consists of rules such as $\ulcorner\mathsf{false}\urcorner^l \vee P \rightarrow P$ that *selectively* remove the trivially false labeled sub-formulas. The rules $\ulcorner P \wedge Q\urcorner^l \rightarrow \ulcorner P\urcorner^l \wedge \ulcorner Q\urcorner^l$ and $P \Rightarrow \ulcorner Q \Rightarrow R\urcorner^l \rightarrow P \wedge \ulcorner Q\urcorner^l \Rightarrow \ulcorner R\urcorner^l$ comprise the third group; they distribute labels over conjunction and (nested) implication, respectively, so that the label scopes are minimized in the final simplified VCs. The last group encodes knowledge about how the labels will be interpreted in the underlying domain. This group also contains an unnesting rule $\ulcorner\ulcorner t\urcorner^m\urcorner^n \rightarrow \ulcorner t\urcorner^{n\otimes m}$ that lifts nested labels to the top term, and so enables other rules to be applied, but keeps the nesting structure on the labels. This ensures that qualifiers remain nested properly and applied to the originally qualified term.

Rendering. We define the underlying structure and actual textual representation of explanations via a set of rules, where the right-hand side of each rule is an explanation template. These templates provide easy customization and fine-grained control of textual explanations.

The final generation of actual explanations, i.e., turning the (labeled) VCs into a human-readable text, is called rendering. It is independent of the actual aspect that is explained and can thus be reused. It relies on the building blocks described so far and comprises four steps: (1) VC normalization using the labeled rewrite system; (2) label extraction using $[\![\cdot]\!]$; (3) label normalization to fit the labels to the explanation templates; (4) text generation using the explanation templates.

The renderer contains code to interpret the templates and some glue code (e.g., sorting label lists by line numbers) that is spliced in to support the text generation.

At the moment, the collection of explanation templates is being implemented using the ML language.

In the conclusion of this section, we should note that the variety of explanatory concepts is not restricted to those presented here. Various aspects of VCs can be examined and various analysis methods are used in practical verification. For example, in [2] a *safety property* was also considered, thus introducing appropriate labels.

4. Case study

Let us consider the C-kernel variant of the function `NegateFirst` on page 126. Its annotations look like

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

$$\begin{array}{ll} \text{post:} & \forall i.\ (0 \leq i \leq \mathrm{MD}(\texttt{Length}) \Longrightarrow \\ & \quad ((\mathrm{MD}(\mathrm{MeM}(old, i)) < 0 \wedge (\forall j.\ 0 \leq j < i \Rightarrow \mathrm{MD}(\mathrm{MeM}(old, j)) \geq 0)) \Rightarrow \\ & \qquad\qquad \mathrm{MD}(\texttt{MeM}(\texttt{ia}, i)) = -\mathrm{MD}(\mathrm{MeM}(old, i)) \wedge \\ & \quad old[i] \geq 0 \Rightarrow \mathrm{MD}(\texttt{MeM}(\texttt{ia}, i)) = \mathrm{MD}(\mathrm{MeM}(old, i))) \end{array}$$

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

Obviously, they reflect the fact that `NegateFirst` searches for the first negative element in an array, changes its sign and exits. The original array content is stored in the auxiliary variable *old*.

The VCG produces five VCs and one trivially true Hoare triple. The comments about their successful proof in Simplify and Z3 can be found in

[10]. Here we focus on explanations. Even the shortest VC can be hard to comprehend. This VC corresponds to the path from the function entry point up to the loop entry point. Its labeled form is

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

For clarity, we kept the pre-condition and loop invariant in their symbolic form. The reader can substitute them with real formulas to estimate the volume of the final VC. What does this formula mean? What role does it play in the verification process? To answer these questions, we use our label rendering methods which produce the following explanation:

```
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.
```

We hope that this explanation (written in a natural language) can actually help in VC understanding or error localization if something goes wrong.

5. Related work

Let us note here two C program verification projects which are ideologically similar to ours. First, a promising approach is proposed within the framework of INRIA project WHY [3]. In fact, WHY is a platform appropriate to verification of many imperative languages. An intermediate language of the same name WHY is defined, and the input programs are translated into it. This translation is aimed at generation of VCs independent of theorem provers. The WHY platform serves as a base for the toolset Frama-C that provides static analysis and deductive verification. Unsupported C features include arbitrary **goto**s, function pointers, arbitrary casts, unions, variadic functions, floating point computations. The verified program list includes rather simple programs (mainly in the field of search and sorting).

Second, the VCC (A Verifier for Concurrent C) project is being developed in Microsoft Research [1]. Programs are translated into logical formulas using a tool Boogie which combines an intermediate language Boogie PL and VC generator. VCs are validated in SMT solver Z3. Boogie PL is not limited to the C language support only. For example, it is used in the Spec# project. However, translation into a different language could be a disadvantage since no correctness proof was presented[5]. At the moment, the VCC developers are focused on verification of the Hyper-V hypervisor, so the information about other case studies is poor.

Besides these two projects, there are many other researches dedicated to C program verification. A more extensive review can be found in [9, 10]. On the contrary, the works concerning VC understanding and formal error tracing are not numerous. Here we can mention the following three. First, the INRIA project Centaur [4]. The generated VCs were analyzed in search of the initial conditional expressions which were used in the `if` statements and loops. This search involves some algorithms from the program debugging field. The language under study was quite simple.

A more recent study [2] has inspired greatly this paper. Denney and Fischer extended the Hoare rules by labels to build up explanations of VCs. The labels are maintained through different processing steps and rendered as natural language explanations. The explanations can easily be customized and can capture different aspects of VCs; the authors focused on labelings that explain their structure and purpose. The approach is fully declarative and the generated explanations are based only on analysis of labels rather than directly on the logical meaning of the underlying VCs or their proofs. The research was focused on simple languages, appropriate for automatic code generation.

Finally, Leino et al. [5] also extend an underlying logic with labels to represent explanatory semantic information, but their use of labels is different. Labels are introduced when the language is "desugared" into a lower-level form. This labeled language is then processed by a standard "label-blind" VCG. The authors use explanations for traces to safety conditions. This is sufficient for program debugging, which is their main motivation.

6. Conclusions and future work

Most verification systems based on the Hoare logic offer some basic tracing support by emitting the current line number whenever a VC is constructed. However, this does not provide any information as to which other parts of the program have contributed to the VC, how it has been constructed, or what is its purpose, and is therefore insufficient as a basis for informative explanations.

This paper is a first attempt to combine the tracing and explanatory techniques in our C program verification project. The Hoare logic of the C-kernel language has already displayed its theoretic soundness and practical

[5]The same is true for the WHY project.

reliability. Some programs with challenging features of C, such as aliasing, side effects and control transfer were successfully verified. Now it is being extended by labeling techniques in a formal way. The extended calculus will provide a user with information necessary to understand VCs and to find potential errors.

Obviously, this method concerns only the intermediate C-kernel stage of our two-level approach. Since the initial C-light programs are translated into C-kernel, the opposite translation of traces and explanations should be implemented. A uniform modeling approach proposed recently in [8] looks promising in this future work.

References

[1] 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. – Lect. Notes Comput. Sci. – 2009. – Vol. 5674. – P. 23–42.

[2] Denney E., Fischer B. Explaining Verification Conditions // Proc. AMAST 2008. – Lect. Notes Comput. Sci. – 2008. – Vol. 5140. – P. 145–159.

[3] Filliâtre J.C., Marché C. Multi-prover verification of C programs // Proc. ICFEM 2004. – Lect. Notes Comput. Sci. – 2004. – Vol. 3308. – P. 15–29.

[4] Fraer R. Tracing the origins of verification conditions. – Rocquencourt, 1996. – 17 p. – (Rapp. / INRIA; No. 2840).

[5] Leino K.R.M., Millstein T., Saxe J.B. Generating error traces from verification condition counterexamples // Science of Computer Programming. – 2005. – Vol. 55, No. 1–3. – P. 209–226.

[6] 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.

[7] 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.

[8] Nepomniaschy V.A., Anureev I.S., Atuchin M.M., Maryasov I.V., Petrov A.A., Promsky A.V. SPECTRUM-2: C program analysis and verificaion system // Proc. PSSV-2010, 14-15 June, Kazan, Russia, 2010. – P. 76–81. (In Russian)

[9] Promsky A.V. Formal semantics of C-light programs and their verification in Hoare logic / Ph.D thesis. – Novosibirsk, 2004. – 175 p. (In Russian)

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