Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

A.V. Promsky

A FORMAL APPROACH TO THE ERROR LOCALIZATION

Preprint
169

Novosibirsk 2012

At first sight the program verification is trustworthy and convenient, which is guaranteed by its well-developed formal basis. However, this method works smoothly in an ideal case only: a program and its annotations are correct, a domain theory is complete and a theorem prover is powerful enough. In case any of these prerequisites do not hold, the verification becomes much more complicated. Apart from possible defects in the foundations (which are not discussed here), one of the reasons is that verification, as a process, still relies on informal methods or implementation tricks. Error tracing is one of them. In this paper we would like to address this issue by giving a review of error localization and explanation methods developed in the C-light project. Their application is illustrated by examples.

**Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова**

А.В. Промский

ФОРМАЛЬНЫЙ ПОДХОД К ЛОКАЛИЗАЦИИ ОШИБОК

**Препринт
169**

Новосибирск 2012

На первый взгляд верификация программ — это надежный метод, что гарантируется ее проработанной формальной базой. Однако она работает без проблем только в идеальном случае, когда программа и ее спецификации корректны, теория проблемной области полна, и доказатель теорем достаточно мощный. При нарушении любого из этих требований верификация значительно усложняется. Помимо возможных проблем в формальных основах (не рассматриваются в данной работе) причиной может быть то, что верификация, как процесс, по прежнему использует неформальные или полуформальные методы. К ним относится локализация ошибок. Данная работа является обзором методов локализации/объяснения ошибок, разработанных в проекте C-light для решения этой проблемы. Их применение показано на примерах.

INTRODUCTION

In the field of verification system design we can distinguish two mainstreams. The first one is represented by systems oriented to higher performance or to the search of possibly wider classes of errors[1]. However, their efficiency usually results from application of less strict methods and algorithms. As an example we can mention the ESC/Java tool which uses incomplete or in some cases even unsound methods.

The opposite camp is represented by academical research (oriented to verification teaching) or by the systems suitable for narrower classes of program properties or program errors. The use of more strict and reliable approaches is typical for such systems. As a counterexample to the mentioned ESC/Java we can note that the proof system Isabelle/HOL was used to develop complete and sound semantics for the restricted dialects Java Card and Java^{light}.

The C program verification project is being developed in the Theoretical programming laboratory of IIS. It possesses two key features. First, we use a multi-level approach in order to overcome numerous obstacles of C program verification. Of course, this feature does not make our project unique. In the corresponding Section we will mention some projects which share the same ideology. However, the second feature consists in formal justification of methods we have chosen. The soundness of our formalisms and algorithms favorably distinguish our project from those which often prefer efficiency to correctness.

While the principal stages of our approach are well developed and justified, still there are less examined parts of the verification process. Among them we can mention the interpretation of verification results and error localization. The poor coverage of these issues in literature reveals that they are inherent in other works too. This paper is a survey of methods and results we developed and obtained in this field recently.

The rest of the paper requires basic knowledge of our approach. It can be described by two schemes. First, our input language, C-light, which is a wide subset of the standard C [8], possesses structured operational semantics. Fig. 1 demonstrates its difference from the classical Plotkin's approach. Indeed, we do not map the variable names directly onto their values. Instead, several consequent mappings are used. We call these mappings *meta-variables* since they play the role of variables in Hoare's calculus. So, when the control flow

[1]The notion of "industrial-strength verification"is quite suitable here.

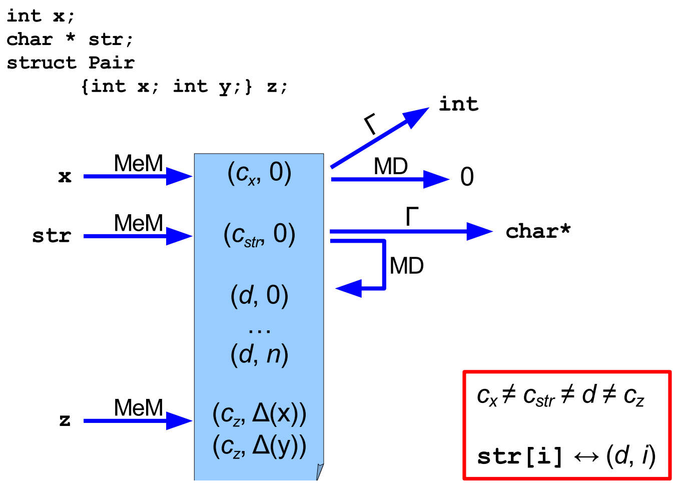


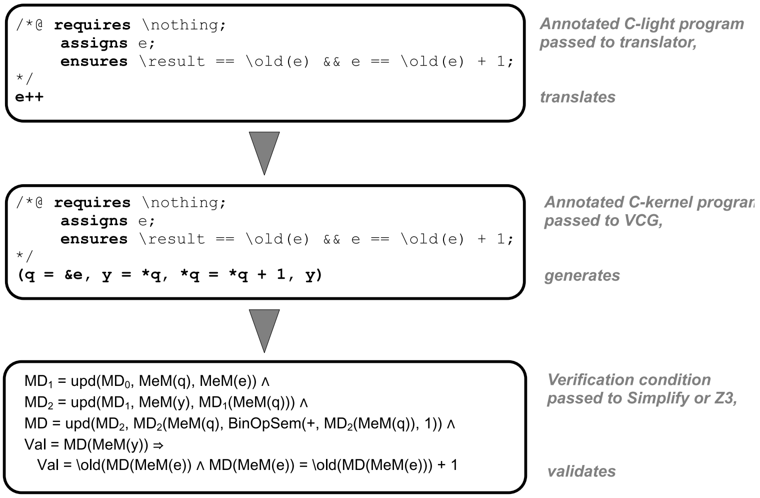
Fig. 1. The memory of the C-light abstract machine

reaches the declaration of the static variable x, the meta-variable MeM allocates a new address c_x for the variable x. The meta-variable MD assigns the default zero to this new object and the meta-variable Γ sets its type to `int`.

At the first stage we translate the annotated[2] C-light program into an equivalent C-kernel program. The C-kernel language, which is a restricted core of C-light, possesses a sound axiomatic semantics. At the second stage the verification condition generator (VCG) produces a set of verification conditions (VC) which are interpreted in the corresponding domain theory. Finally, the theorem prover Simplify is used to prove (or disprove) these lemmas. All stages are shown in Fig 2.

These stages are sufficient if the verification is successful (i.e. all VCs are true). Otherwise, we need to interpret the wrong VCs and find the corresponding errors in the program. To succeed in these tasks we adopted an idea which was pioneered in [3]. The idea is to enrich the axiomatic semantics for C-kernel by special labels. These structural labels establish the necessary connection between VC sub-formulas and program locations. They can be

[2]We use ACSL [1] as our specification language.



```
/*@ requires \nothing;
    assigns e;
    ensures \result == \old(e) && e == \old(e) + 1;
*/
e++
```

```
/*@ requires \nothing;
    assigns e;
    ensures \result == \old(e) && e == \old(e) + 1;
*/
(q = &e, y = *q, *q = *q + 1, y)
```

$\mathrm{MD}_1 = \mathrm{upd}(\mathrm{MD}_0, \mathrm{MeM}(q), \mathrm{MeM}(e)) \wedge$
$\mathrm{MD}_2 = \mathrm{upd}(\mathrm{MD}_1, \mathrm{MeM}(y), \mathrm{MD}_1(\mathrm{MeM}(q))) \wedge$
$\mathrm{MD} = \mathrm{upd}(\mathrm{MD}_2, \mathrm{MD}_2(\mathrm{MeM}(q), \mathrm{BinOpSem}(+, \mathrm{MD}_2(\mathrm{MeM}(q)), 1)) \wedge$
$\mathrm{Val} = \mathrm{MD}(\mathrm{MeM}(y)) \Rightarrow$
$\quad \mathrm{Val} = \backslash\mathrm{old}(\mathrm{MD}(\mathrm{MeM}(e)) \wedge \mathrm{MD}(\mathrm{MeM}(e)) = \backslash\mathrm{old}(\mathrm{MD}(\mathrm{MeM}(e))) + 1$

Fig. 2. The C-light verification project: simplified view

extracted from VCs, normalized and automatically translated into explanations written in a natural language. A more recent study addresses the attempt to promote those labels into the theorem prover Simplify, which is not originally designed to support them.

The review of these studies and application of corresponding methods form the rest of the paper (Sections 2 and 3). A survey of related works is given in the Conclusion.

This research is partially supported by the grant 11-01-00028-a from RFBR.

1. EXTENDING HOARE LOGIC BY LABELS

In practice users are often confronted with failed VCs but receive no additional information about the causes of the failure. They must thus analyze the VCs, interpret their constituent parts, and refer to the corresponding source code locations. 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 or precise error localization.

The first part of this Section is a review of the labeling and explanatory techniques developed in our research. Their detailed description can be found in [11, 13]. The second subsection represents a more recent result, when we tried to preserve the labels during the proof stage. As we will see in Section 3, combination of all these methods simplifies significantly the analysis of VCs and error localization.

Using the idea from [3], we will extend the Hoare rules by "semantic mark-up" so that the calculus itself builds 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 and are then extracted from the final VCs and rendered as natural language explanations.

1.1. 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 roles 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 the `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 the specification language. That is why, comparing with [3], we use the words "assumes" and "ensures" in the concepts instead of "asserts" and "establishes".

Labels. We will use the notation from [3] 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 [3], where a language without labels was examined. We hope that the reader can distinguish between these semantic labelings and program labels depending on the context.

1.2. Extended Hoare logic

A typical Hoare triple looks like $\mathcal{E}nv \Vdash \{P\}\ S\ \{Q\}$, where P and Q are (labeled) pre- and postcondition, respectively, S is a program, and *environment* $\mathcal{E}nv$ is a triple $\langle cf, nl, IAx\rangle$. Here, cf is the name of current function, a nonnegative integer nl corresponds to a current *nesting level* and a set of Hoare triples IAx denotes the inductive hypotheses. We will use the superscripts $\mathcal{E}nv^{i}$ to access the environment components.

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 are 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`:

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

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

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

Thus, this rule reduces the verification of the entire program verifying 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. 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:

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

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 [9] for details). Let us consider the declaration of a global integer variable and the declaration of a local array with initialization:

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

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

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 the 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 the 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:

$$\mathcal{E}nv \Vdash \{\mathcal{U}pd(Q, Expn)\}\ Expr\ \{Q\}\ . \tag{1}$$

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

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

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

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

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

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]

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

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:
$\alpha = (\text{MeM} \leftarrow SI(\text{MeM}, \text{MD}, \mathcal{E}nv^2, \overline{x}));$
$\beta = (\text{MeM}\gamma \leftarrow \text{MeM});$
$\delta = (\text{MD} \leftarrow \text{upd}(\text{MD}, \text{MeM}(lval, \mathcal{E}nv^2), z));$
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. Variable renaming allows us to avoid universal quantification on the local level.

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

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

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

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

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:

$$\frac{\mathcal{E}nv \Vdash \{P\}\ S_1\ \{\ulcorner R \urcorner^{\textsf{ens\_post}}\} \qquad \mathcal{E}nv \Vdash \{\ulcorner R \urcorner^{\textsf{asm\_pre}}\}\ S_2\ \{Q\}}{\mathcal{E}nv \Vdash \{P\}\ S_1\ S_2\ \{Q\}} \ . \tag{2}$$

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 a C-kernel program will make this rule useless. As we will see later, the Hoare rule for labeled statements will provide successful applicability of (2).

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

$$\frac{\mathcal{E}nv(nl \leftarrow nl + 1) \Vdash \{P\}\ \mathit{Statements}\ \{Q'\}}{\mathcal{E}nv \Vdash \{P\}\ \{\mathit{Statements}\}\ \{Q\}} \ , \tag{3}$$

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, restrictions in the rule (2) 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:

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

where $\mathcal{E}nv_1^3 = \mathcal{E}nv^3 \cup \Big\{ (\{I_i\}\ \texttt{goto}\ \ l_i\texttt{;}\ \{\mathsf{false}\}, \mathcal{E}nv^2) \ \Big|\ \ i = 1, \ldots, n \Big\}$ 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 $\{I_i\}\ \texttt{goto}\ \ l_i\texttt{;}\ \{\mathsf{false}\}$ 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:

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

The evaluating function $\mathcal{E}val$ is defined in [9] 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:

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

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 semantics of **loop** is based on a classical Hoare rule:

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

The labeling reflects the rule meaning. The invariant should be ensured at the loop entry point and is thus labeled with ens_inv. The individual subformulas 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.

[4]Note the explicit braces in the premise.

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:

$$\mathcal{E}nv \Vdash \{\ulcorner BR(I, m, \mathcal{E}nv^2)\urcorner^{\mathsf{ens\_inv}}\}\ \texttt{goto}\ l\texttt{;}\ \{\mathsf{false}\}\ , \tag{4}$$

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 (3). Obviously, when performing a jump, we can leave several nesting blocks. The successive application of *Reduce* is implemented by definition:

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

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

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

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.

2. 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.

The labeled rewriting rules do not simply reuse the usual unlabeled ones because labels need careful handling. For example, we can safely remove labels from trivially true sub-formulas because these require no explanations. On the other hand, labels from trivially false labeled sub-formulas are removed selectively, since there must be a context to explain the failure. A detailed review of labeled rewriting and simplification rules can be found in [3]. In the meantime we will discuss one of rewriting aspects below.

The final generation of actual explanations, i.e., turning the (labeled) VCs into a human-readable text, is called rendering. It relies on the building blocks described in Fig. 3 and comprises four steps: (1) VC normalization

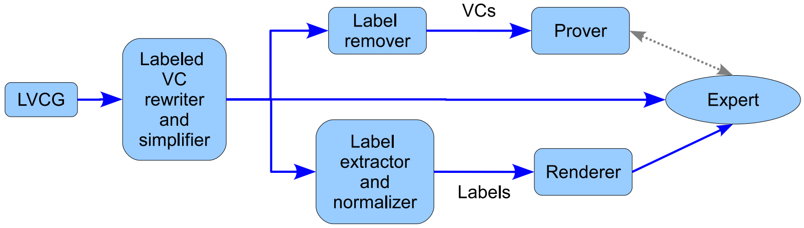

LVCG
Labeled VC rewriter and simplifier
Label remover
VCs
Prover
Expert
Label extractor and normalizer
Labels
Renderer

Fig. 3. Labels and labeled VCs

using the labeled rewrite system; (2) label extraction; (3) label normalization to fit the labels to the explanation templates; (4) text generation using the explanation templates.

2.1. Verification condition simplification revisited

The two-level access model resolves the problems of composite objects and aliasing but leads to an explosion in the size of verification conditions (VC). In [10] we discussed some of the simplification strategies. Those strategies were intuitively clear, but their implementation required non-trivial analysis algorithms. The newly developed labeling technique gave us an opportunity to simplify them. So, we would like to return to an example from [10] in order to demonstrate the changes.

Let us consider a simple program:

```
int x, y, z;

x = 1;
y = 2;
z = 3;
z = 4;
```

Actually, this program is so simple that it could be verified in the classical Hoare logic without the use of metavariables. Let us compare the VCs obtained in each approach. For such program we can use `true` as a precondition.

The postcondition for the classical logic is $\mathtt{x} = 1 \wedge \mathtt{y} = 2 \wedge \mathtt{z} = 4$. Applying

$$\begin{bmatrix} c_1 = \mathsf{naddr}(\mathrm{MD}_1) & \wedge \\ \mathrm{MeM}_2 = \mathit{upd}(\mathrm{MeM}_1, \mathtt{x}, c_1) & \wedge \\ \Gamma_2 = \mathit{upd}(\Gamma_1, c_1, \mathtt{int}) & \wedge \\ \mathrm{MD}_2 = \mathit{upd}(\mathrm{MD}_1, c_1, \omega) & \wedge \\ c_2 = \mathsf{naddr}(\mathrm{MD}_2) & \wedge \\ \mathrm{MeM}_3 = \mathit{upd}(\mathrm{MeM}_2, \mathtt{y}, c_2) & \wedge \\ \Gamma_3 = \mathit{upd}(\Gamma_2, c_2, \mathtt{int}) & \wedge \\ \mathrm{MD}_3 = \mathit{upd}(\mathrm{MD}_2, c_2, \omega) & \wedge \\ c_3 = \mathsf{naddr}(\mathrm{MD}_3) & \wedge \\ \mathrm{MeM} = \mathit{upd}(\mathrm{MeM}_3, \mathtt{z}, c_3) & \wedge \\ \Gamma = \mathit{upd}(\Gamma_3, c_3, \mathtt{int}) & \wedge \\ \mathrm{MD}_4 = \mathit{upd}(\mathrm{MD}_3, c_3, \omega) & \wedge \\ \mathrm{MD}_5 = \mathit{upd}(\mathrm{MD}_4, \mathrm{MeM}(\mathtt{x}), 1) & \wedge \\ \mathrm{MD}_6 = \mathit{upd}(\mathrm{MD}_5, \mathrm{MeM}(\mathtt{y}), 2) & \wedge \\ \mathrm{MD}_7 = \mathit{upd}(\mathrm{MD}_6, \mathrm{MeM}(\mathtt{z}), 3) & \wedge \\ \mathrm{MD} = \mathit{upd}(\mathrm{MD}_7, \mathrm{MeM}(\mathtt{z}), 4) & \end{bmatrix} \Longrightarrow \begin{bmatrix} \mathrm{MD}(\mathrm{MeM}(\mathtt{x})) = 1 & \wedge \\ \mathrm{MD}(\mathrm{MeM}(\mathtt{y})) = 2 & \wedge \\ \mathrm{MD}(\mathrm{MeM}(\mathtt{z})) = 4 & \end{bmatrix}$$

Fig. 4. The verification condition

the Hoare assignment rule 4 times, we obtain the true assertion:

$$\mathsf{true} \Longrightarrow (1 = 1 \wedge 2 = 2 \wedge 3 = 3)\ .$$

Two-level access in the C-kernel logic immediately complicates the precondition: $\mathrm{MD}(\mathrm{MeM}(\mathtt{x})) = 1 \wedge \mathrm{MD}(\mathrm{MeM}(\mathtt{y})) = 2 \wedge \mathrm{MD}(\mathrm{MeM}(\mathtt{z})) = 4$. The verification condition (VC) from [10] is presented in Fig. 4. In order to simplify its proof we proposed some rewriting strategies. Many of them were based on the following observations:

1. The VC carefully accumulates information about program objects. New memory locations (c_1, c_2, c_3) are reserved, object types are identified correspondingly to declaration (`int`) and the default indefinite values (ω) are assigned to automatic variables.
2. The primary goal of the VC (as described in postcondition) concerns only the final values of the variables without appealing to their addresses or types.
3. So, we can safely omit all information unnecessary for the proof.

Well, it sounds obvious, but the implementation of such ‘simplifier’ is not so easy. Indeed, simple examination of the conjunct

$$\mathrm{MD}_2 = \mathit{upd}(\mathrm{MD}_1, c_1, \omega)$$

tells nothing about its necessity. We must trace the whole chain of MD_i to understand that this is an intermediate update of the metavariable MD. By the way, the names MD_i themselves are not hints, since they are chosen only for the reader's convenience. As we noticed in the introductory section, we try to avoid the situation when our formal algorithms are built on semi-formal information or implementation tricks[5].

Again, allow us to formulate the argument from [3] in favor of labeling techniques. In the classical Hoare's approach a verification condition does not contain any information about the process of its own generation. For example, if we replace the first assignment by `x = 2` we will receive a false VC

$$\mathsf{true} \Longrightarrow (1 = 2 \wedge 2 = 2 \wedge 3 = 3) \ ,$$

but the conjunct $1 = 2$ does not reveal the assignment which procreated it. In the logical formulas (VCs) all connections to the source locations are lost. Thus, the analysis in rewriting strategies can be as complex as the direct VC proof.

On the contrary, our labels are introduced by formal inference rules and provide all necessary hints. According to the new labeled semantics for C-kernel our VC takes the form represented in Fig. 5. And now we can handle all sub-formulas in a more formal way.

The use of labels has an additional outcome. We slightly simplified the Hoare's rule for variable declaration. Indeed, in the previous version of the semantics we were forced to postulate explicitly that every new object has a unique address (note the terms "$c_i = \mathsf{naddr}(\mathrm{MD}_j)$" in Fig. 4). Now that every allocation is decorated by the label alloc, we can retrieve such information when it is needed. For example, if we are verifying a program on pointers, we can search all terms labeled by alloc and supply the proof environment with axioms about the originality of every address.

Apart from the removal of unnecessary information, simplifying strategies include splitting into sub-goals and antecedent unfolding. All details can be found in [10]. As a final illustration here let us consider one of the resulting sub-VCs:

$$upd(\mathrm{MD}_7, upd(\mathrm{MeM}_3, \mathsf{z}, c_3)(\mathsf{z}), 4)(upd(\mathrm{MeM}_3, \mathsf{z}, c_3)(\mathsf{z})) = 4.$$

The standard semantics of *upd* immediately provides the truth of this assertion. The comparison with the starting VC in Fig. 5 shows that such

[5]For example, the information on how the verification condition generator introduces new names.

$$\begin{bmatrix} \ulcorner \mathsf{true} \urcorner^{\mathsf{asm\_pre}(1)} & \wedge \\ \ulcorner \mathrm{MeM}_2 = \mathit{upd}(\mathrm{MeM}_1, \mathtt{x}, c_1) \urcorner^{\mathsf{alloc}(1)} & \wedge \\ \Gamma_2 = \mathit{upd}(\Gamma_1, c_1, \mathtt{int}) & \wedge \\ \ulcorner \mathrm{MD}_2 = \mathit{upd}(\mathrm{MD}_1, c_1, \omega) \urcorner^{\mathsf{init}(1)} & \wedge \\ \ulcorner \mathrm{MeM}_3 = \mathit{upd}(\mathrm{MeM}_2, \mathtt{y}, c_2) \urcorner^{\mathsf{alloc}(1)} & \wedge \\ \Gamma_3 = \mathit{upd}(\Gamma_2, c_2, \mathtt{int}) & \wedge \\ \ulcorner \mathrm{MD}_3 = \mathit{upd}(\mathrm{MD}_2, c_2, \omega) \urcorner^{\mathsf{init}(1)} & \wedge \\ \ulcorner \mathrm{MeM} = \mathit{upd}(\mathrm{MeM}_3, \mathtt{z}, c_3) \urcorner^{\mathsf{alloc}(1)} & \wedge \\ \Gamma = \mathit{upd}(\Gamma_3, c_3, \mathtt{int}) & \wedge \\ \ulcorner \mathrm{MD}_4 = \mathit{upd}(\mathrm{MD}_3, c_3, \omega) \urcorner^{\mathsf{init}(1)} & \wedge \\ \ulcorner \mathrm{MD}_5 = \mathit{upd}(\mathrm{MD}_4, \mathrm{MeM}(\mathtt{x}), 1) \urcorner^{\mathsf{asgn}(2)} & \wedge \\ \ulcorner \mathrm{MD}_6 = \mathit{upd}(\mathrm{MD}_5, \mathrm{MeM}(\mathtt{y}), 2) \urcorner^{\mathsf{asgn}(3)} & \wedge \\ \ulcorner \mathrm{MD}_7 = \mathit{upd}(\mathrm{MD}_6, \mathrm{MeM}(\mathtt{z}), 3) \urcorner^{\mathsf{asgn}(4)} & \wedge \\ \ulcorner \mathrm{MD} = \mathit{upd}(\mathrm{MD}_7, \mathrm{MeM}(\mathtt{z}), 4) \urcorner^{\mathsf{asgn}(5)} & \end{bmatrix} \Longrightarrow \ulcorner \begin{array}{|l|} \mathrm{MD}(\mathrm{MeM}(\mathtt{x})) = 1 \wedge \\ \mathrm{MD}(\mathrm{MeM}(\mathtt{y})) = 2 \wedge \\ \mathrm{MD}(\mathrm{MeM}(\mathtt{z})) = 4 \end{array} \urcorner^{\mathsf{ens\_post}(5)}$$

Fig. 5. The labeled verification condition

simplification makes sense.

2.2. Keeping labels in theorem provers

In our previous work [11, 13], we examined only the explanatory application of labels. Correspondingly, the VCs had been striped of labels before they were fed to a theorem prover (Simplify, for example). It is not a problem when Simplify validates a VC as true.

However, if Simplify discards a VC, then there are two possibilities.

- In some cases this prover is not powerful enough, giving an empty counterexample or the whole formula as a counterexample. Then the user must analyze the wrong VC manually. Here, explanations built by label renderer (see Fig. 3) are of great help.
- Simplify can give a stronger counterexample built out of some subformulas of the original formula. This complicates the error localization task. Remember that the explanation which contains links to the source code, is built for the whole VC. Thus, the user has to match the sub-formulas against the original formula in order to find the corresponding parts of the explanation. Taking into account the Lisp-like syntax of Symplify and higher order of meta-variables (see Fig. 1) it is not an easy task.

Here is an intermediate conclusion. When we remove labels from VCs we have to perform an additional task. The whole idea of labels was to promote them up to separate operations. But theorem provers accept logical formulas only, not labels.

There are two solutions. In the long-term outlook we could turn to a more powerful prover (like Z3, for example), which provides its own API. It may allow us to develop a driver for such prover. This driver will handle all necessary labels and links in a transparent manner from the user's point of view.

On the other hand, while we prefer the "simpler" prover Simplify, we could use a less elegant, but still working approach. We could introduce labels into logical terms as dummy arguments, which do not impact the truth.

According to this solution we replace the label remover (see Fig. 3) by the "label introducer" (LI). It translates labeled logical terms, given in prefix notation, into a so-called S-expression. Precisely, when LI meets the following term

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

it transforms it into

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

where op' is an appropriate Lisp representation for op. For example, the equality sign '=' will be replaced by `L_EQ`. Finally, we need to provide a connection of new names `L_`op' with the standard keywords. This is done via Simplify patterns. So, if the '=' sign was present in the original VC, the final formula will be accompanied by

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

This is the main shortcoming of proposed solution. Besides the growing length of formulas, every VC passed to Simplify must be proven in such an environment. Fortunately, the set of functional/operational names and logical connectives is finite. We may create a library of such patterns to use in our experiments with C-light programs. Moreover, the user does not need to examine these patterns. When he obtains a counterexample, the only task he must do is to find the label-terms (quite straightforward) and find the corresponding parts in the VC explanation. The following Sect. demonstrates such experiments.

3. CASE STUDIES

Let us try to apply these methods in order to find errors in programs. At the moment we are focused on experiments with the Standard Library functions, but they are quite complex to serve as an obvious demonstration. Instead, let us appeal to our successful experience with a well-known collection of verification challenges [6]. Those programs are simple (if not primitive) but they represent some theoretical obstacles for deductive verification. We have already demonstrated that we can verify them [10], now we deliberately introduce some bugs in them. Also these examples will give us some idea about applicability of proposed methods.

3.1. An incorrect expression

Consider the following program (with omitted annotations):

```
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 ...
}
```

In theory, this program should search an array. When the first negative element is reached, its sign changes and the loop aborts. As you can see, the minus sign is dropped in the assignment `ia[i] = -ia[i]` in the loop body.

Actually, this issue deserves some additional discussion. As a rule, programmers switch off the strict mode of their compilers in order to avoid a flood of unimportant warning messages. So, a typical compiler may simply ignore that assignment (since it does nothing) without giving any notice. In such case, the user will have to test program trying to understand what went wrong. Now, lets us see whether our verification methods can be of any use.

First of all, we need formal specifications which correctly reflect the intended behavior of the program.

$$\begin{aligned}
\text{pre}: \;& \exists \mathit{old} : \texttt{int[]}.\ \mathrm{MD}(\mathrm{MeM}(\texttt{ia})) \neq \mathsf{null} \wedge \\
& \qquad\qquad\quad \mathrm{MD}(\mathrm{MeM}(\texttt{ia})) = \mathrm{MD}(\mathrm{MeM}(\mathit{old}))
\end{aligned}$$

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

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

Now we translate the original program into a corresponding C-kernel program:

```
void NegateFirst(int ia[], int Length) {
    //@ pre ...
    auto int i;
    i=0;
    while(i < Length){
        //@ inv ...
        if (ia[i]<0){
            ia[i] = ia[i];
            goto L;
        }
        else {}
        auto int* q1;
        q1 = &i;
        *q1 = *q1 + 1;
    }
    L:;
    //@ post ...
}
```

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

The LVCG produces five labeled VCs and one trivially true Hoare triple. Four VCs are proved successfully by Simplify or Z3 provers. One of them corresponds to the path from the function entry point up to the loop entry point. Its labeled form is

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

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.
```

This is the explanatory aspect of the labeling technique. Though its necessity in the case of true VC is questionable, it may be useful for verification teaching.

In the meantime, the VC corresponding to *then*-branch of the `if` statement proved false. It was produced after the application of rules (1) and (4) from Section 2.1. Its labeled form is

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

Here, to make the formula observable, the post-condition and the loop invariant are also given in symbolic form. If we try to prove this VC, we will see that the conjunct

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

contradicts the conclusion. In principle, this is enough to localize the bug. However, the Simplify prover does not produces the counterexamples in such infix and well-formatted form. Instead, the user will encounter something like that:

```
(L_EQ md
    (upd md1 (index (mem ia) (md1 (mem i)))
         (md1 (index (mem ia) (md1 (mem i)))))
    (label_upd 8))
```

Again, those who are familiar with the C-light abstract machine and the Lisp-like syntax of Simplify can understand it. Nevertheless, with more realistic programs the complexity of formulas grows considerably, becoming a challenge even for a specialist. Besides, our aim is to hide all technical and prover-dependant details as much as possible.

This is where the explanatory method begins to work providing a possibly better solution. Indeed, in applying our label rendering algorithms we will obtain the following explanation for the whole VC:

```
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 "thenbranch is chosen at line 7,
  - substitution for MD
     originating in array update at line 8,
show that label invariant holds at line 9.
```

So, instead of deciphering the Lisp-like terms appropriate for Simplify, the user can address the descriptions written in natural language. Knowing the label of a troublesome conjunct he can check the corresponding program location (the boxed sub-sentence).

3.2. Missed initialization

In Sect. 2.1 we discussed the case when the default initialization can be irrelevant to verification goals and, thus, can be omitted. On the contrary, in the following example it will play a crucial role.

This is also one of challenging programs from [6] and the corresponding problem is known as aliasing. Our successful verification was demonstrated in [10]. Now, let us remove one of strings (by a comment) so that access to uninitialized object takes place:

```
#include "stdio.h"

struct C {
    struct C *a;
    int i;
};

int m(void) {
    struct C c;
    c.a = &c;
    // c.i = 2;
    return c.i + (c.a)->i;
};
```

```
int main(void){
    printf("%d", m());
    return 0;
}
```

Just like in the previous case study, the error here can or cannot be signaled by the compiler depending on the warning level switch. In any case this program can be compiled and executed with unpredictable results.

Specifications for the function `m`[6] have the form:

Pre(`m`) : true
Post(`m`) : $\mathsf{Val} = 4$

An intermediate C-kernel counterpart for the function `m` looks like

```
int m(void)
{
    auto struct C c;
    auto struct C* x1;
    auto struct C** y1;

    x1 = &c;
    y1 = &c.a;
    *y1 = x1;

    auto int ret_val;
    auto int x2;
    auto int x3;

    x2 = (c.a)->i;
    x3 = c.i;
    ret_val = x3 + x2;

    return ret_val;
};
```

Expressions in the C-kernel language are very simple. This leads to expansion in the length of intermediate programs but allows us to simplify axiomatic semantics.

[6]Verification of the function `main` is of little interest.

The body of `m` forms a single linear area. The axiomatic semantics produces a single VC, which after simplification looks like

$$\left[\begin{array}{l} \mathsf{MD}_1 = \mathit{upd}(\mathsf{MD}_0, c1, \omega)\wedge \\ \mathsf{MD}_2 = \mathit{upd}(\mathsf{MD}_1, c1, c1)\wedge \\ \mathsf{MD}_3 = \mathit{upd}(\mathsf{MD}_2, \mathsf{c}, c1)\wedge \\ \mathsf{MD}_4 = \mathit{upd}(\mathsf{MD}_3, \mathsf{mb}(c1, \mathsf{a}), \mathsf{val}(\&\mathsf{c}, \mathsf{MD}_4))\wedge \\ \mathsf{Val} = \mathsf{BinOpSem}(+, \mathsf{MD}_4(\mathsf{mb}(c1, \mathtt{i})), \mathsf{val}(\mathtt{(c.a)\text{->}i}), \mathtt{int}) \end{array}\right] \Rightarrow \mathsf{Val} = 4.$$

We omitted labels for brevity.

Despite the fact that this program is simpler than the previous example, it represents a more complex class of errors. In Sect. 3.1 the prover Simplify explicitly gave a counterexample to demonstrate the falsity of VC. But here Simplify signals that it is unable to prove the following formula:

```
(IMPLIES
  (AND
    (DISTINCT i a)
    (DISTINCT c_1 c_2)
    (EQ
      MD
      (upd MD1_5 (get MeM i)
        (+ (get MD1_5 (get MeM i)) 2)
      )
    )
    (EQ
      MD1_5
      (upd
        MD1_4
        (get MD1_4 (get MeM a))
        (+ (get MD1_4 (get MD1_4 (get MeM a))) 2)
      )
    )
    (EQ MD1_4 (upd MD1_3 (get MeM a) (get MeM i)))
    (EQ (get MD1_2 c_2) |@undef|)
    (EQ MD2_2 MD1_3)
    (EQ MeM (upd MeM1_3 a c_2))
    (EQ MeM1_3 (upd MeM1_2 i c_1))
    (EQ MD1_3 (upd MD1_2 c_2 0))
    (EQ (get MD1_1 c_1) |@undef|)
    (EQ MD2_1 MD1_2)
    (EQ MD1_2 (upd MD2_1 c_1 0))
    (EQ
      MeM1_2
      (upd (upd MeM1_1 i |@undef|) a |@undef|)
    )
  )
  (EQ val 4)
)
```

The problem is that the undefined value ω is being matched against natural numbers. So, error localization required some additional work here. For example, the logical axioms establishing the connection between undefined values and explicit falsity were added to the proof environment. Then manual analysis was applied in order to find which conjunct introduced an undefined value. Its label revealed the declaration `struct C c;`. So, like any compiler we can only give a warning that an uninitialized variable (structure member) is used in the statement `return c.i + (c.a)->i;`

The examples in this Sect. demonstrate the domains of fully automatical and semi-automatical applications of our error localization method. Indeed, errors can be divided into two classes. The first case corresponds to a program, which is in some sense complete, but incorrect. In such a case we can precisely show the wrong construction. On the contrary, in the second case we can only isolate some region where something is missed, but cannot automatically deduce it.

An additional disadvantage consists in the restriction of these formal methods to the C-kernel language. In order to trace possible errors back to the original C-light source the user must refer to translation protocols and manually establish the necessary links. Automatic solution (based on the formal methods) of this task is one of the goals in our future work.

4. RELATED WORK

Let us mention 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 [4]. In fact, WHY is a platform appropriate for verification of many imperative languages. An intermediate language of the same name WHY is defined, and input programs are translated into it. This translation is aimed at the 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 and 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 [2]. Programs are translated into logical formulas using the Boogie tool 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 is presented[7]. At the moment, the VCC developers are focused on the verification of the Hyper-V hypervisor, so the information about other case studies is insufficient.

Apart from these two projects, there are many other works dedicated to C program verification. A more extensive review can be found in [9, 10]. On the contrary, works concerning VC understanding and formal error tracing are not numerous. Here we can mention the following three. First, the INRIA project Centaur [5]. Generated VCs were analyzed while searching for the initial conditional expressions which were used in `if` statements and loops. This search involves some algorithms from the program debugging field. The language under study is quite simple.

A more recent study [3] has inspired this paper greatly. Denney and Fischer extended the Hoare rules by labels to build up explanations of VCs. Labels are maintained through different processing steps and rendered as natural language explanations. 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 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. [7] also extend an underlying logic with labels to

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

represent explanatory semantic information, but their use of labels is different. Labels are introduced when a 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.

5. CONCLUSION

Deductive verification was designed as a way of trustworthy proof of program correctness compared to the more probabilistic nature of traditional testing. Naturally, the question of its reliability arises. While the theoretical foundations have been justified in classical papers quite long ago, in practice the whole question rests on the correctness of verification system implementation. Obviously, an ideal solution is to implement it in the target programming language and then apply it to self-verification. This is a non-trivial task for a language such as C. There are two main obstacles.

First, any sensible C program uses the standard library. This library does not possess a complete formal specification, let alone its verification. The first steps in this direction were not taken until recently. Apart from the experiments within the framework of WHY/Frama-C (not represented in papers), some work is in progress in our C-light project [12].

Second, the formal verification basis applies only to VCs generation and their proofs. The processes of verification results interpretation (especially, the negative results) and error localization are formalized insufficiently.

Results. This paper describes the combination of tracing and explanatory techniques in our C program verification project. The Hoare logic of the C-kernel language was 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. In addition a method of label logical promotion is proposed. It allows us to turn labels into logical terms so that a theorem prover can safely handle them. The application points of these methods during verification process are shown in Fig. 5.

Outlook. Obviously, the labeling 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 should be implemented. An interesting opportunity here is provided by APIs developed for the LLVM infrastructure and its Clang front-end. In particular, they

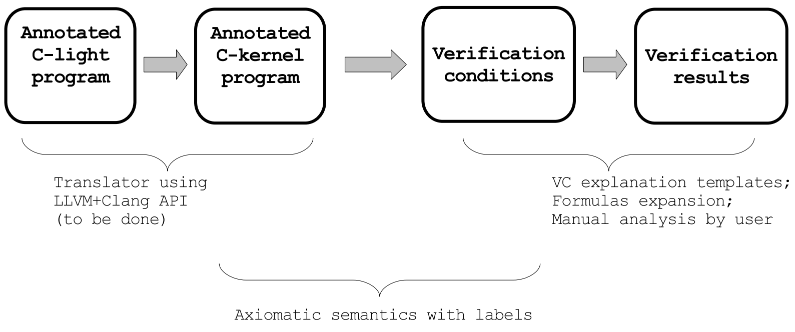

Annotated C-light program
Annotated C-kernel program
Verification conditions
Verification results
Translator using LLVM+Clang API (to be done)
VC explanation templates; Formulas expansion; Manual analysis by user
Axiomatic semantics with labels

Fig. 6. Methods for error localization

allow to associate the translating handlers with nodes of abstract syntax tree. The place of these future studies is also shown in Fig. 5.

REFERENCES

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. Jacobs B., Kiniry J.L., Warnier M. Java program verification challenges // Proc. FMCO 2002. — Lect. Notes Comput. Sci. — 2003. — Vol. 2852. — P. 202–219.
7. 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.
8. Programming languages — C: ISO/IEC 9899:1999. — 1999. — 566 p.
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.
11. Promsky A.V. Error-tracing semantics for C-kernel // Bull. Nov. Comp. Center, Comp. Science, 31 (2010), 123-138.
12. 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. — pp. 83–92.
13. Promsky A.V. Verification Condition Understanding // Proc. Ershov Informatics Conference (PSI Series, 8-th Edition). — June 27 – July 1, 2011, Akademgorodok, Novosibirsk, Russia. — pp. 295–300.

А.В. Промский

ФОРМАЛЬНЫЙ ПОДХОД К ЛОКАЛИЗАЦИИ ОШИБОК

Препринт
169

Рукопись поступила в редакцию 21.11.2012
Рецензент И. С. Ануреев
Редактор Т. М. Бульонкова

Подписано в печать 10.12.2012
Формат бумаги 60×84 1/16 Объем 1,8 уч.-изд.л., 2,0 п.л.
Тираж 60 экз.

Центр оперативной печати “Оригинал 2”, г. Бердск, 49-а, оф. 7,
тел./факс 8 (241) 5 38 77