Automated error localization for C programs using deductive verification

Dmitry Kondratyev

A.P. Ershov Institute of Informatics Systems

C-light and C-kernel

- Correct methods / algorithms at each step.
- Solution:

  > *"Restrictions that contribute to provability are what make a programming language good."* Tony Hoare

- C-light language
  - covers the majority of C99 (C0 — completely, Misra C — almost);
  - sets the calculation order;
  - doesn't have some low-level operations.
- C-kernel language
  - is defined in terms of operational semantics;
  - axiomatic semantics is correct with respect to operational one.

Translation from C-light to C-kernel

The main idea of this translation is to localize side effects.

As a result, all instructions and expressions are translated into a form where only variables and constants are their arguments.

For example, the following expression:

$$f(e_1,\ \ldots,\ e_{i-1},\ e_i,\ e_{i+1},\ \ldots,\ e_n)$$

is translated to

$$(x\ =\ e_i,\ f(e_1,\ \ldots,\ e_{i-1},\ x,\ e_{i+1},\ \ldots,\ e_n)),$$

where

- e_i is not variable or constant;
- $e_{i+1},\ \ldots,\ e_n$ – variables and constants;
- f – a function or $+$, $-$, $*$, $/$, $<$, $>$, $<=$, $=>$, $!=$, $==$;
- x is a new variable of the same type as e_i.

It is *Ops* translation rule.

Kinds of C-kernel expressions with memory access

Kinds of lvalues in the C-kernel language:

1. var
2. $*pointer\_var$
3. $var.structure\_field$
4. $(*pointer\_var).structure\_field$
5. $array[index]$
6. $*(array\_of\_pointers[index])$

Kinds of rvalues with reference operators:

1. $\&var$
2. $\&(var.structure\_field)$
3. $\&(array[index])$

Memory allocation:

1. $new(type)$
2. $delete(pointer)$

Memory model of the C-light language

MeM is a mapping from an object to its address

MD is a mapping from an object's address to its value.

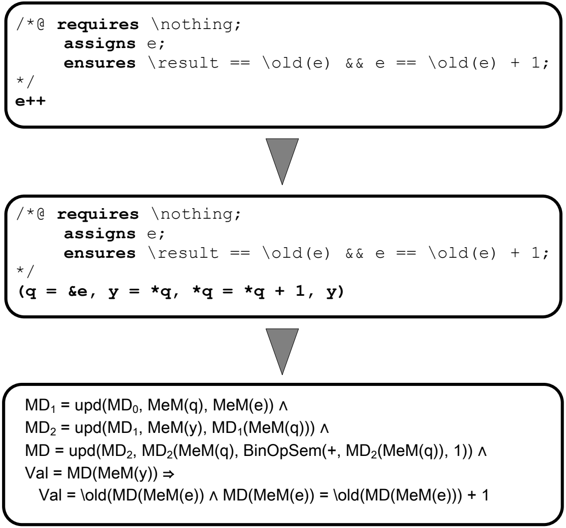
If MD contains a pair $(adr\ val')$ (where val' is some value), then the mapping $upd(MD,\ addr,\ val)$ differs from MD by replacing the pair $(adr\ val')$ with the pair $(adr\ val)$

If $addr$ does not belong to the domain of MD, then the mapping $upd(MD,\ addr,\ val)$ differs from MD by adding the pair $(adr\ val)$

Axioms about *MeM* and *MD*

1. $MD(NULL) = void$
2. $MeM(obj) \neq NULL$
3. $upd(MD,\ NULL,\ val) = MD$
4. $upd(MeM,\ obj,\ NULL) = MeM$
5. $delete(MD, NULL) = MD$
6. $(upd(MD,\ addr,\ val))(addr) = val$
7. $(upd(MD, adr_1, val))(adr_2) = MD(adr_2)\ if\ adr_1 \neq adr_2$
8. $upd(MD,\ MeM(obj),\ MD(MeM(obj))) = MD$
9. $upd(MeM,\ obj\ MeM(obj)) = MeM$
10. $(upd(MeM, obj, addr))(obj) = addr$
11. $(upd(MeM, obj_1, adr))(obj_2) = MeM(obj_2)\ if\ obj_1 \neq obj_2$
12. $(delete(MD, addr))(addr) = void$
13. $(delete(MD, adr_1))(adr_2) = MD(adr_2)\ if\ adr_1 \neq adr_2$
14. $(delete(MeM, obj))(obj) = void$
15. $(delete(MeM, obj_1))(obj_2) = MeM(obj_2)\ if\ obj_1 \neq obj_2$
16. $delete(upd(MD, addr, val), addr) = MD$
17. $delete(upd(MeM, obj, addr), obj) = MeM$

The C-lightVer system: overview



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

Annotated C-light program passed to translator,

translates

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

Annotated C-kernel program passed to VCG,

generates

$MD_1 = upd(MD_0, MeM(q), MeM(e)) \wedge$

$MD_2 = upd(MD_1, MeM(y), MD_1(MeM(q))) \wedge$

$MD = upd(MD_2, MD_2(MeM(q), BinOpSem(+, MD_2(MeM(q)), 1)) \wedge$

$Val = MD(MeM(y)) \Rightarrow$

$\quad Val = \backslash old(MD(MeM(e)) \wedge MD(MeM(e)) = \backslash old(MD(MeM(e))) + 1$

Verification condition passed to Simplify or Z3,

validates

Java Program Verification Challenges

Jacobs B., Kiniry J., Warnier M. Java Program Verification Challenges. Lecture Notes in Computer Science. 2003. Volume 2852. pp. 202–219.
DOI: `https://doi.org/10.1007/978-3-540-39656-7_8`

Translator from C-light to C-kernel

```
56. for(;  i > 0 ; i++)
57. {
58.  k++;
59.  continue;
60.  j++;
61. }

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

Reverse translator

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

Reverse translator

```
43. /* begin reverse 70-76 */
44. for(;  i > 0 ; i++)
45. {
46.   k++;
47.   continue;
48.   j++;
61. }
62. /* end reverse */
```

Maryasov I.V., Nepomniaschy V.A., Promsky A.V., Kondratyev D.A.
Automatic C Program Verification Based on Mixed Axiomatic Semantics. Automatic Control and Computer Sciences. 2014. Volume 48. Issue 7. pp. 407–414.
DOI: `https://doi.org/10.3103/S0146411614070141`

Inference rule for array update

$$\frac{\{P\}\ \mathbf{prog};\ \{Q(MD \leftarrow upd(MD, MeM(a,i), rval))\}}{\{P\}\ \mathbf{prog};\ \ \mathbf{a[i]} = \mathbf{rval}\ \{Q\}}$$

Inference rule for *if* statement

$$\frac{\{P\}\ \textbf{prog}\ \{B\}\ \mathbf{S_1}\ \{Q\}, \{P\}\ \textbf{prog}\ \{\neg B\}\ \mathbf{S_2}\ \{Q\}}{\{P\}\ \textbf{prog};\ \textbf{if B}\ \mathbf{S_1}\ \textbf{else}\ \mathbf{S_2}\ \{Q\}}$$

Inference rule for invariants

$$\frac{\{P\}\ \mathbf{prog}\ \{\lceil INV \rceil \to Q\}}{\{P\}\ \mathbf{prog};\ \{INV\}\ \{Q\}}$$

Inference rule for *while* loop

$$\frac{\begin{array}{l}\{P\}\ \textbf{prog};\ \{I\},\\ \{I \land B\}\ \textbf{S}\ \{I\},\\ \qquad I \land \neg B \to Q\end{array}}{\{P\}\ \textbf{prog};\ \textbf{while B inv I do S}\ \{Q\}}$$

Inference rule for empty program

$$\frac{P \rightarrow Q}{\{P\}\ \{Q\}}$$

Why verification may fail?

- The program may be incorrect or unsafe;
- The annotations may be incorrect or incomplete;
- The simplifier may be too weak;
- The underlying theory may be incomplete;
- The prover may run out of resources.

In each of these cases, users are typically confronted only with failed VCs but receive no additional information about the causes of the failure.

Idea (following Denney and Fischer)

- The Hoare rules are extended by "semantic mark-up"so that the calculus itself can be used to build up explanations of the VCs.
- This mark-up takes the form of structured labels that are attached to the terms used in the Hoare rules, so that the VCG produces labeled versions of the VCs.
- The labels are maintained through the different processing steps:
  - simplification
  - extraction from the final VCs
  - rendering into natural language explanations

Denney E., Fischer B. Explaining Verification Conditions. Lecture Notes in Computer Science. 2008. Volume 5140. pp. 145–159. DOI: `https://doi.org/10.1007/978-3-540-79980-1_12`

Labels

We will derive labeled terms

$$\ulcorner t \urcorner^{l},$$

where

- each term t can be adorned with a label l
- labels will have the form $c(o, n)$
- c — type
- o — location
- n — optional list of labels

Term supplemented with multiple labels:

Let us consider term supplemented with multiple labels:

$$\lceil\lceil\lceil\lceil\lceil t\rceil^{l_1}\rceil^{l_2}\rceil^{\cdots}\rceil^{l_{k-1}}\rceil^{l_k}$$

We may replace multiple labels by first label:

$$\lceil t\rceil^{l_1}$$

where

$$n(l_1) = list(l_2, \ldots . l_{k-1}, l_k)$$

Verification condition structure

Horn clause:

$$H_1 \wedge \ldots H_n \to C$$

where

- $H_1, \ \ldots \ H_n$ — hypotheses;
- C — conclusion.

Review of concepts of semantic labels

Concepts	**Examples**	**Aspects of verification conditions**
Hypotheses - Assertions - Control predicates	asm_pre, asm_inv, then, while_t	*Hypotheses* represent the assumptions that some logic statements hold in some program points. Among them can be both the original preconditions and the control expressions of the statements `while` and `if`.
Conclusions	ens_post, ens_inv_iter	*Conclusions* reflect the main purpose of the verification conditions, which is in the *ensurance* that some assertions hold in the given program points.
Qualifiers - Substitutions - Assignments	sub, upd, alloc, init	*Qualifiers* introduce more detailed characterization for hypothesis and conclusions by recording how a subformula was produced.
Inductive qualifiers	call, pres_inv	*Inductive qualifiers* give the secondary purpose of the verification conditions. For example, the verification conditions for the inner loop conceptually relate to the purpose of the conditions for the nested loop as well.

Inference rule for *update* supplemented with semantic labels

$$\frac{\{P\}\ \mathbf{prog};\ \{Q(MD \leftarrow \lceil upd(MD, MeM(a,i), rval)\rceil^{upd})\}}{\{P\}\ \mathbf{prog};\ \mathbf{a[i]} = \mathbf{rval}\ \{Q\}}$$

Inference rule for *if* supplemented with semantic labels

$$\frac{\{P\}\ \mathbf{prog}\ \{\lceil B \rceil^{\mathit{then}}\}\ \mathbf{S_1}\ \{Q\}, \{P\}\ \mathbf{prog}\ \{\lceil \neg B\ \rceil^{\mathit{else}}\}\ \mathbf{S_2}\ \{Q\}}{\{P\}\ \mathbf{prog};\ \mathbf{if}\ \mathsf{B}\ \mathbf{S_1}\ \mathbf{else}\ \mathbf{S_2}\ \{Q\}}$$

Inference rule (*invariant*) supplemented with semantic labels

$$\frac{\{P\}\ \textbf{prog}\ \{\lceil \mathit{INV} \rceil^{asm\_inv} \rightarrow \lceil Q \rceil^{ens\_post}\}}{\{P\}\ \textbf{prog};\ \{\mathit{INV}\}\ \{Q\}}$$

Inference rule for *while* supplemented with semantic labels

$$\frac{\begin{array}{l} \{\lceil P\ \rceil^{\mathit{asm\_pre}}\}\ \textbf{prog};\ \{\lceil I\rceil^{\mathit{ens\_inv}}\}, \\ \lceil\{\lceil I\rceil^{\mathit{asm\_inv}} \wedge \lceil B\rceil^{\mathit{while\_t}}\}\ \textbf{S}\ \{\lceil I\rceil^{\mathit{ens\_inv\_iter}}\}\rceil^{\mathit{pres\_inv}}, \\ \qquad \lceil I\rceil^{\mathit{asm\_inv\_exit}} \wedge \lceil\neg B\rceil^{\mathit{while\_f}} \rightarrow \lceil Q\rceil^{\mathit{ens\_post}} \end{array}}{\{P\}\ \textbf{prog};\ \textbf{while B inv I do S}\ \{Q\}}$$

Inference rule for *empty* supplemented with semantic labels

$$\frac{\lceil P \rceil^{asm\_pre} \rightarrow \lceil Q \rceil^{ens\_post}}{\{P\}\ \{Q\}}$$

`Negate_first` example

```
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 ...
}
```

Annotations of *Negate_ first*

Precondition of *Negate_ first*:

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

Postcondition of *Negate_ first*:

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

Invariant of *Negate_first* loop

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

Negate_first with error

```
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 ...
}
```

C-kernel representation of Negate_first

```
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 ...
}
```

Unproven verification condition

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

Kondratyev D., Promsky A. An integrated approach to the error localization during C program verification. System Informatics. 2013. Issue 1. pp. 79–96.
DOI: `https://doi.org/10.31144/si.2307-6410.2013.n1.p79-96`
(In Russian)

Conjunct from unproven verification condition

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

Text templates for labels

then → *assumption that "then"-branch is chosen at line ...*

→ means correspondance between label and text template

Label extraction

Denney and Fischer proposed extracting labels from the VC in ascending order of the number of the lines corresponding to the label. This is how a list of labels is generated and used to generate the VC explanation.

An algorithm for extracting labels from the VC was implemented in the C-lightVer system, which differs from the approach of Denney and Fischer. This algorithm is based on the representation of the VC as a depth-first traversal tree. Labels are added to the list used to generate a VC explanation in the order of that depth-first traversal.

Generation of explanation of verification condition

The result of the label extraction algorithm is a list of labels used to generate the VC explanation. After labels are extracted from the VC, the VC explanation is generated.

The algorithm for generating VC explanation is based on a sequential traversal of the resulting list of labels. For each label that is visited during the traversal, the text of its line-number-filled template is added to the text explaining the VC.

Explanation of unproven verification condition

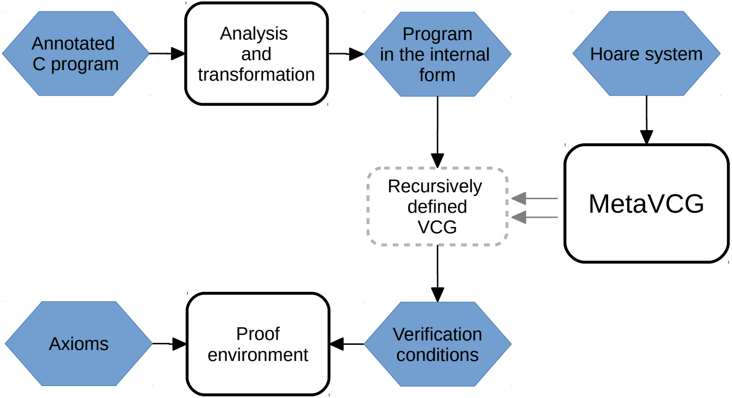
```
This VC corresponds to lines 6-10 in the function
"NegateFirst". Its purpose is to contribute the loop
invariant preservation on each iteration.
Hence, given
  - assumption that loop invariant holds at line 6,
  - assumption that the loop condition holds at line 6,
  - assumption that "then"-branch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

MetaVCG: origins

- Can the correctness of a VCG be guaranteed not only by testing/verification but also by its construction?
- Basing on classical results by E.W. Dijkstra, R.L. London, D.C. Luckham etc., M. Moriconi and R. Schwartz proposed in 1981 a method for mechanically constructing VCGs from a useful class of Hoare logics.
- Any VCG constructed by the method is shown to be sound and deduction-complete w.r.t the associated Hoare logic.

Moriconi M., Schwartz R.L. Automatic construction of verification condition generators from hoare logics. Lecture Notes in Computer Science. 1981. Volume 115. pp. 363–377.
DOI: `https://doi.org/10.1007/3-540-10843-2_30`

The C-lightVer system: overview



MetaVCG: soundness and completeness

- Metagenerator takes a Hoare logic as an input and automatically derives a recursively defined VCG. The axiomatic rules must be given in a *normal* form with several constraints.
- Many axiomatic rules do not satisfy them, so the authors provided an equivalence-preserving transformation from a more liberal *general* form into a normal one.
- The soundness and completeness were proved for their method, thus providing that a produced VCG is *correct* w.r.t. the original axiomatic definition.

MetaVCG: the pattern language

```
{P} prog {INV},
    {INV /\ e} S {INV},
INV /\ (not e) => Q
|-
{any_predicate(P)} any_code(prog)
{any_predicate(INV)}
while(simple_expression(e)) any_code(S)
{any_predicate(Q)}
```

Kondratyev D.A., Promsky A.V. Developing a self-applicable verification system. Theory and practice. Automatic Control and Computer Sciences. 2015. Volume 49. Issue 7. pp. 445–452. DOI: `https://doi.org/10.3103/S0146411615070123`

label construct

```
{(label P asm_pre)} prog {(label INV ens_inv)},
(label
    {(label INV asm_inv) /\ (label e while_t)} S
        {(label INV ens_inv_iter)}
    pres_inv
),
(label INV asm_inv_exit) /\ (label (not e) while_f) =>
    (label Q ens_post)
|-
{any_predicate(P)} any_code(prog)
{any_predicate(INV)}
while(simple_expression(e)) any_code(S)
{any_predicate(Q)}
```

Kondratyev D. Implementing the Symbolic Method of Verification in the C-Light Project. Lecture Notes in Computer Science. 2018. Volume 10742. pp. 227–240.
DOI: `https://doi.org/10.1007/978-3-319-74313-4_17`

Defining *label* construct

$$(label\ c\ text)$$

where

- *c* – new type of semantic label;
- *text* – new text template for semantic label.

Text template may contain special control characters:

- %*begin* – line number of begin of correspompding source code;
- %*end* – line number of end of corresponding source code.

For example, let us consider definition of *then* label:

```
(label
  then
  assumption that "then"-branch is chosen at lines %begin-%end
)
```

Related works

- Denney E., Fischer B. Explaining Verification Conditions. Lecture Notes in Computer Science. 2008. Volume 5140. pp. 145–159. DOI: `https://doi.org/10.1007/978-3-540-79980-1_12`
- Maryasov I.V., Nepomniaschy V.A., Promsky A.V., Kondratyev D.A. Automatic C Program Verification Based on Mixed Axiomatic Semantics. Automatic Control and Computer Sciences. 2014. Volume 48. Issue 7. pp. 407–414. DOI: `https://doi.org/10.3103/S0146411614070141`
- Kondratyev D., Promsky A. An integrated approach to the error localization during C program verification. System Informatics. 2013. Issue 1. pp. 79–96. DOI: `https://doi.org/10.31144/si.2307-6410.2013.n1.p79-96` (In Russian)
- Kondratyev D.A., Promsky A.V. The Complex Approach of the C-lightVer System to the Automated Error Localization in C-Programs. Automatic Control and Computer Sciences. 2020. Volume 54. Issue 7. pp. 728-739. DOI: `https://doi.org/10.3103/S0146411620070093`

Automated error localization for C programs using deductive verification

Dmitry Kondratyev

A.P. Ershov Institute of Informatics Systems