Bull. Nov. Comp. Center, Comp. Science, 37 (2014), 93–105
© 2014 NCC Publisher

Experiments on self-applicability in the C-light verification system. Part 2

A. V. Promsky

Abstract. Successful development of the theoretical foundations of C program verification in the project C-light allowed us to address an interesting practical task. We would like to develop a self-applicable verification system for C programs. The first step towards this goal was a series of experiments to verify some fragments of the input analyzer/translator. Now we are ready to address the verification condition generator. As a basis of our approach, we chose the metageneration approach proposed by Moriconi and Schwartz. Metageneration allows us to build automatically a recursively defined generator from a Hoare logic and also ensures its consistency and completeness. This paper discusses the metageneration approach as well as the experiments to verify its implementation in C-light.

Keywords: verification, specification, C-light, ACSL, MetaVCG.

1. Introduction

The answer to the question whether a verification system is reliable can consist of two main parts:

1. Since the verification methods are based on some mathematical concepts (sets, relations, calculi, etc.), their properties can be formally proved. For example, the proof of axiomatic semantics soundness is quite a traditional practice [1]. However, these proofs are usually performed by hand. The assistance of automatic theorem provers can contribute significantly to their trustworthiness. The examples of such "mechanical" proofs are much more uncommon, though some researchers have obtained remarkable results [9, 10].
2. Program implementations of those theoretical methods should also be checked. And again, in addition to usual testing, formal verification is desirable. In particular, if a verification system is implemented in a target language, then its self-verification could be an ultimate check. Speaking about the C language, we are not aware of such a self-applied system.

In the Laboratory of Theoretical Programming (IIS) we are developing the C-light verification system [6]. The C-light language covers the major part of the previous standard (C99). In order to avoid the problems of a

Hoare logic for the full C, we translate the input programs in a restricted core called C-kernel. The verification condition generator (VCG) for C-kernel produces lemmas (verification conditions, VCs), while the interactive prover Simplify tries to discharge them. Taking into account the importance of correctness, we formally proved some properties of the steps of our approach [8].

On our way to a self-verifiable system we have already taken two steps. First, the specifications written in ACSL [2] were developed for a part of the Standard C library [11]. Every meaningful C program relies on library routines, so these annotations are an important prerequisite. Second, some parts of the parsing/translating module of our system were verified [12]. In fact, it is implemented in C++ using API of the compiler Clang. Thus the complete verification is unachievable; however the translator is rich in code expressible in C-light, which made it a good subject of study.

And now we are concentrated on the VCG stage of our system. At first sight, we could use the same unsophisticated approach as we applied for the translator. Nevertheless, there is an alternative way which also has some benefits.

In 1981, Moriconi and Schwartz [7] proposed a method which forms a *meta verification condition generator* (MetaVCG). It 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.

In the presence of this meta-stage, the classical three-block scheme (input analyzer/VCG/prover) of a verification system changes slightly (Fig. 1).

In addition to theoretical correctness, this method has other advantages for our project. In the future, we plan to enrich C-light with the remaining C constructs or to use it as a basis for the related languages (Objective C, C++). Also the specific axiomatic logics for restricted classes of programs are of great interest. We could mention here a Hoare system for linear algebra developed in our laboratory long ago. The rules of that system can replace some nested loops (inherent in matrix handling) by logical constructs which are much more convenient for proving than usual loop invariants. As a more concrete example, let us consider a code

$swap(x, y, buf) \equiv$
```
memcpy(buf, x, m);
memcpy(x, y, m);
memcpy(y, buf, m);
```

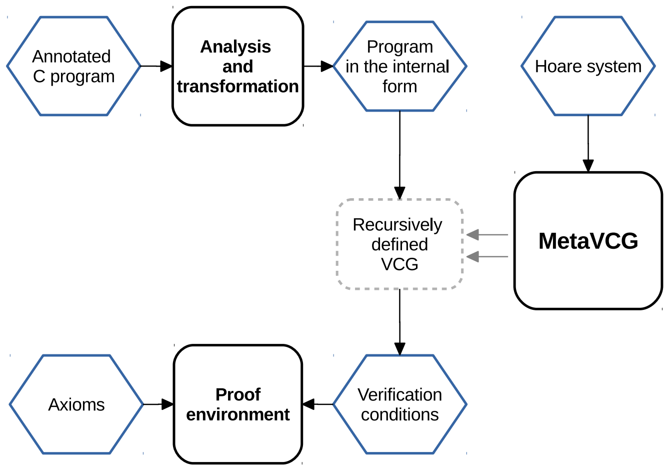

Figure 1. A "meta-stage" in the verification process

which can be found in some library routines. Its treatment by general rules will involve a triple instantiation of specifications for `memcpy` leading to a cumbersome quantified VC. In the meantime, we can enrich the Hoare system by the following axiom:

$$\{x = x_0 \wedge y = y_0\}\ swap(x, y, buf)\ \{x = y_0 \wedge y = x_0\}$$

Though it works only for those programs which contain a fragment *swap*(..), its application can simplify the proof considerably. The MetaVCG approach can relieve us of necessity to rewrite the VCG each time manually.

As for the immediate benefit, it allowed us to distance from C++ API of Clang. The implementation of MetaVCG described in the paper is given in C-light, so we can perform a more complete verification in comparison with the translator.

2. Meta verification condition generation

The method of metageneration proposed by Moriconi and Schwartz [7] consists of two steps. A general axiomatic definition is first transformed into a normal form which, in turn, develops into a recursively defined VCG.

2.1. Preliminary definitions

Following Moriconi and Schwartz, we will use metavariables $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{R}$, Γ, ... to denote partially interpreted first-order formulas. These formulas can contain uninterpreted predicate symbols P, Q, R, ... and formulas from the underlying theory. For example, $\mathcal{P}$ could denote P, $P \wedge x = 5$, or $x = 5$. We assume that the symbols P, Q, R,... may be instantiated by formulas in the underlying theory.

We also need a binary relation $\Leftarrow$ on uninterpreted predicate symbols. For a Hoare sentence of the form

$$\{\mathcal{P}(P_1, ..., P_m)\}\ S\ \{\mathcal{Q}(Q_1, ..., Q_n)\},$$

where the predicate symbols $P_1, ..., P_m$ and $Q_1, ..., Q_n$ are logically free in $\mathcal{P}$ and $\mathcal{Q}$, respectively, we have

$$P_i \Leftarrow Q_j, \quad \text{for } i \in \{1, ..., m\} \text{ and } j \in \{1, ..., n\}.$$

Intuitively, a relation $P \Leftarrow Q$ indicates that the binding of the predicate symbol P depends upon the binding of Q. The relation $\Leftarrow$ is defined with respect to a set of the Hoare triples. The notation $\overset{+}{\Leftarrow}$ denotes the *transitive closure* of $\Leftarrow$.

Similarly, for a rule of the form

$$\frac{\{P_1\}\ S_1\ \{\mathcal{Q}_1\}, ..., \{P_n\}\ S_n\ \{\mathcal{Q}_n\}, \Gamma}{\{\mathcal{P}\}\ S\ \{Q\}} \tag{1}$$

the relation $\ll$ defines the dependence of a proof concerning S on proofs of $S_1, ..., S_n$. In particular, we have $S \ll S_i$, for $i \in \{1, ..., n\}$. For a Hoare axiom system, we define the transitive closure $\ll +$ in the obvious manner.

We use the function $FreePreds$ to denote the set of *logically free* predicate symbols. $FreePreds$ applied to a first-order formula denotes its logically free symbols. Then inductively,

$$FreePreds(\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}) = FreePreds(\mathcal{P}) \cup FreePreds(\mathcal{Q})\ ,$$

and for a proof rule R of the form (1)

$$FreePreds(R) = \bigcup_{i=1}^{n} FreePreds(\{P_i\}\ S_i\ \{\mathcal{Q}_i\}) \\ \cup\, FreePreds(\Gamma) \cup FreePreds(\{\mathcal{P}\}\ S\ \{Q\}).$$

We will use the function $FragVars$ to denote the set of "fragment variables" in the program fragment S of a Hoare triple $\{P\}\ S\ \{Q\}$. For example, $FragVars$ applied to "`if` (B) S_1 `else` S_2" has the value $\{B, S_1, S_2\}$. If applied to an entire Hoare rule, $FragVars$ yields a set containing the fragment variables from every Hoare sentence in the rule.

Finally, we define the notion of a bound occurrence of an uninterpreted predicate symbol in a rule. For a rule R, a predicate symbol in $FreePreds(R)$ is *bound* in R iff it is in $FragVars(R)$. Otherwise, the occurrence is said to be *free* in R.

2.2. The normal form for rules

Let us consider the following

Definition. A *normal form rule* is any instance N of

$$\frac{\{P_1\}\ S_1\ \{\mathcal{Q}_1\}\ ,...,\ \{P_n\}\ S_n\ \{\mathcal{Q}_n\},\ \Gamma}{\{\mathcal{P}\}\ S\ \{Q\}}$$

that satisfies the following constraints:

1. P_1,..., P_n and Q are predicate symbols free in N.
2. $FreePreds(\Gamma) \subseteq FreePreds(N) \cup FragVars(S)$.
3. The fragment variables of each S_i must be bound in S. That is, it must be the case that $\cup_{1 \leq i \leq n} FragVars(S_i) \subseteq FragVars(S)$.
4. *Dependency ordering.* The Hoare-triple premises of N must satisfy two dependency constraints.

   a. $P_i \overset{+}{\Leftarrow} P_j\ \supset\ i < j$

   b. $T \overset{+}{\Leftarrow} U \wedge \neg(\exists R) U \overset{+}{\Leftarrow} R\ \supset\ U \equiv Q \vee U$ bound in N.

5. *Monotonicity.* Let $\mathcal{P}[P \leftarrow \mathbf{false}, P \in s]$ denote $\mathcal{P}$ with the proper substitution of **false** for each predicate P in the set s. Then, the following constraint on $\mathcal{P}$ must be satisfied:

   $$\mathcal{P}[P_1, ..., P_n, Q \leftarrow \mathbf{true}]\ \vee\ \forall s \subseteq \{P_1, ..., P_n, Q\}\ \neg\mathcal{P}[P \leftarrow \mathbf{false}, P \in s].$$

   This constraint must hold for Γ and for each Q_i.

Two constraints are imposed on a system of the normal form rules: (i) Any terminal string σ in the programming language can be an instance of at most one language fragment S defined by a normal form axiom or an inference rule. (ii) The relation $\ll +$ must be irreflexive.

Constraint 4 ensures that VCG will be able to compute instantiations for all free uninterpreted predicate symbols in the rules. In particular, constraint 4a requires an ordering of free predicate symbols that is made apparent by the following schema:

$$\frac{\{P_1\}\ S_1\ \{\mathcal{Q}_1(P_2, ..., P_n)\}\ ,...,\ \{P_i\}\ S_i\ \{\mathcal{Q}_i(P_{i+1}, ..., P_n)\}\ ,...,\ \{P_n\}\ S_n\ \{\mathcal{Q}_n\},\ \Gamma}{\{\mathcal{P}(P_1, ..., Q)\}\ S\ \{Q\}}.$$

This has the effect of eliminating dependency cycles, such as a premise of the form $\{P\}$ ... $\{P\}$ or a pair of premises of the form $\{P\}$ ... $\{R\}$ and $\{R\}$ ... $\{P\}$. Given this ordering, constraint 4b ensures that the tail of every dependency chain is either expressible as a function of the postcondition Q or is bound in a program fragment.

Constraint 5 is necessary for completeness of a VCG, i.e. it guarantees that VCG is able to compute the weakest precondition $wp(S, Q)$ for given S and Q. It is done by imposing a monotonicity constraint on rules, which eliminates the rules where certain "changes of sign" exist between the preconditions of the premises and the precondition in the conclusion.

2.3. The general form for rules and its translation into the normal one

The normal form constraints serve two purposes. First, a recursively defined VCG can be built up automatically for the normal rules. Indeed, since the preconditions of premises are individual predicate symbols, they can be substituted by the weakest preconditions for the corresponding programs and postconditions. For a rule of the form (1), the recursive function wp is defined as follows:

$$\begin{aligned} wp(S, Q) = \mathcal{P}[P_1 \leftarrow wp(S_1, \mathcal{Q}_1), ..., P_n \leftarrow wp(S_n, \mathcal{Q}_n)] \wedge \\ (\forall \overline{v})\Gamma[P_1 \leftarrow wp(S_1, \mathcal{Q}_1), ..., P_n \leftarrow wp(S_n, \mathcal{Q}_n)], \end{aligned}$$

where $[P_1 \leftarrow t_1, ..., P_n \leftarrow t_n]$ denotes n subsequent substitutions performed from left to right, and $\overline{v}$ is a set of free logical variables of Γ.

Second, the constraints together with the definition of wp allow us to prove that VCG (as a proof system) is sound and complete w.r.t. the initial Hoare system in the normal form [7].

On the other hand, the normal form constraints narrow the class of admissible Hoare systems. Note that axiomatic semantics for C-kernel [6] does not satisfy these requirements. Moreover, the normal form rules look quite unusual, which is why Moriconi and Schwartz proposed a more liberal *general form* for rules as well as an algorithm of its translation into the normal one. Here we discuss them only briefly. The general form preserves the constraints (1–3) and (4b) (together with a modified monotonicity property). Thus an awkward order on the premises disappears. Further, the preconditions in premises may take more forms: not only singular predicate symbols but also formulas of the underlying theory, as well as the conjunctions of these two variants. The idea of the translation algorithm is as follows: we gather all preconditions that are different from the singular predicate symbols. Instead of them we will use "fresh" predicate symbols. The connection between these new symbols and old formulas is established by some implications where old formulas may be gathered in conjunctions (simultaneously removing duplicates). Finally, the new rule premises must be reordered to satisfy the constraint (4a).

To illustrate, let us consider the proof rules for `if` and `while` statements in the general (left column) and equivalent normal form:

$$\frac{\{P \wedge B\}\, S_1\, \{Q\}, \quad \{P \wedge \neg B\}\, S_2\, \{Q\}}{\{P\}\ \texttt{if}\ (B)\ S_1\ \texttt{else}\ S_2\ \{Q\}} \qquad \frac{\{P_1\}\, S_1\, \{Q\}, \quad \{P_2\}\, S_2\, \{Q\}}{\{B \supset P_1 \wedge \neg B \supset P_2\}\ \texttt{if}\ (B)\ S_1\ \texttt{else}\ S_2\ \{Q\}}$$

$$\frac{\{P \wedge B\}\, S\, \{P\}, \quad P \wedge \neg B \supset Q}{\{P\}\ \texttt{while}\ (B)\ S\ \{Q\}} \qquad \frac{\{P_1\}\, S\, \{P\}, \quad P \wedge \neg B \supset Q, \quad P \wedge B \supset P_1}{\{P\}\ \texttt{while}\ (B)\ S\ \{Q\}}.$$

An intermediate conclusion here is that axiomatic semantics for C-kernel fits the requirements of the general form. So, it can be translated in an equivalent normal system which, in turn, can be transformed into a recursive VCG. Thus the MetaVCG approach can be applied in our case.

3. Implementation and experiments

In this section, we discuss the composing parts of our adaptation of the MetaVCG approach. They include the development of the pattern language which is used to express the Hoare rules and axioms. The main (meta)generation algorithm has been written in C-light, thus making its partial verification possible. An example of the code is also presented here.

First of all, let us note the difference between the original idea of MetaVCG and our implementation. Our metagenerator is a two-parameter function and there is *currying* during its work. So, if H is a Hoare system and AP is an annotated program to be verified, then

$$\text{MetaVCG}(H, AP) = \text{VCG}_H(AP),$$

where VCG_H is an ordinary generator built for H. This is not a good solution from the point of view of efficiency since in every verification experiment (the argument AP) we rebuild the generator even if the Hoare system is the same (say, Hoare system for C-kernel). On the other hand, it allows us to verify a single program instead of two, one of which 'appears' prior to any specification. As long as we are concentrated on theoretical studies, this strategy serves our purposes quite well. In future, we may use the generator in a usual way as a stand-alone application or a plug-in.

We also do not restrict our MetaVCG to the *weakest precondition* strategy used by Moriconi and Schwartz. The *strongest postcondition* approach can be applied changing the direction of the program tree analysis.

3.1. The pattern language

A VCG built from a Hoare system in the normal form tries to instantiate those free predicate symbols and fragment variables with specific annotations and program constructs. Since a user provides MetaVCG with axioms and rules in a less restricted general form, we propose a pattern language to express them.

Let us note that the classic way to represent Hoare logics (like in Section 2.1) is good enough in theory but it is not so flexible in practice. That is why we do not require strictly that the symbols P, Q, R express predicates while S_i are program fragments. Any symbols can be used, and membership in a specific class is indicated by syntactic wrapping. For example, the construction `any_code(S)` can match any sequence (including empty) of programming language statements, whereas `exists_code(S)` corresponds to a singular construction. The construction `simple_expression` denotes any expression which does not contain function calls and type casts.

To illustrate this, let us consider the assignment axiom

```
{(any_predicate(Q))(MD <- upd(MD, loc(val(e, MeM..STD)),
                              cast(val(val(e', MeM..STD)),
                                   type(e', MeM, TP),
                                   type(e, MeM, TP))))
}
    e = simple_expression(e');
{any_predicate(Q)}
```

and the proof rule for the `while` statement

```
{P1} S {INV},
(INV /\ cast(val(val(e, MeM..STD)), type(e, MeM, TP), int) = 0)
    => Q,
(INV /\ cast(val(val(e, MeM..STD)), type(e, MeM, TP), int) != 0)
    => P1
|-
{any_predicate(INV)}
    while(simple_expression(e)) any_code(S)
{any_predicate(Q)}
```

To save the space, we show them as if they were already transformed from the general form. That is why two logical statements about predicates `Q` and `P1` appear in the `while`-rule premises. Only then the rule satisfies the constraints of the normal form. The names MD, MeM, and STD reflect our detailed memory model [6] but they do not alter principally the logical structure of the familiar Hoare sentences.

3.2. Implementation of MetaVCG

The arguments of metagenerator – Hoare axioms and rules together with an annotated program – are parsed and transformed into the corresponding internal representations. We have already mentioned that on the lower level the C++ API of the compiler Clang was enabled. Thus actually they are passed from the Clang representation into structures compatible with C-light.

As an example, let us consider the datatype `pattern_node` which represents axioms and conclusions of the Hoare rules.

```
struct pattern_node
{
    int is_omitted;

    int has_category;
    char* category;

    int has_identifier;
    char identifier[64];

    int has_type;
    char* type;

    int has_value;
    char* value;

    int is_matched;
    int table_length;
    char match_identifiers[2][1000][64];

    int children_count;
    struct pattern_node* children[1000];
};
```

Since we deal with axiomatic semantics, it is obvious that the first and the last node in the children list are a pre- and postcondition, correspondingly. Each node has attributes (category, identifier, type) which contribute to the matching process. In addition, there is a table of correspondence between the program and pattern names which is filled up during the matching. The program tree is based on the datatype `program_node` which, in general, is similar to `pattern_node`.

Thus the metagenerator builds a program tree for an annotated program and a collection of patterns for an applied Hoare system. According to the proof direction, it chooses the leftmost/rightmost program construction and tries to find an appropriate pattern. For a selected pattern it recursively applies to the premises of the corresponding Hoare rule.

The implementation of MetaVCG is quite large, so let us restrict ourselves to the main tree matching function in the rest of this section. At the moment we use a “greedy” algorithm. Such algorithm can be applied successfully thanks to the simplicity of the axiomatic semantics of C-kernel[1] and the constraints of C-light (for example, the control transfer into compound statements from the outside is forbidden). Perhaps, consideration of

[1]That was the reason to introduce the translation from C-light.

the complete C or C++ will require a more general approach.

3.3. An example of a verified code

The idea of verification condition generation consists in finding an appropriate proof rule for the current program construction. Application of a proof rule results in a formula of the underlying theory (VC) and/or a set of new Hoare triples for which the process should be recursively applied. As we have seen, during the meta-stage the axioms and rules (written in the pattern language) are transformed into the objects of the type `struct pattern_node` which are trees. The program constructions are also the trees of the analogous type `struct program_node`.

The following function implements the tree matching in our MetaVCG. Despite the size, its idea is quite straightforward. The inequality of the node properties signals that the nodes are incomparable. This is the case when, for example, a rule for an assignment is matched against, for example, a loop, or vice versa. Otherwise we recursively begin to compare children in the corresponding nodes. As noted above, the correspondence between program identifiers and names in a pattern is recorded in the table `match_identifiers`. For every comparison of the children pair, the parent node table serves as a base. That is why we copy it before each recursive function call. Moreover, during the children comparison, the inner table can be replenished with new records. Thus, we need to rewrite the main table every time.

```
int match_trees(struct pattern_node* pattern,
                struct program_node* code)
{
    pattern->is_matched = 1;

    if (compare_properties(pattern, code) == 0)
    {
        pattern->is_matched = 0;
    }
    else
    {
        int found = 1, found_omitted = 0, i = 0, j = 0;
        int code_count = code->children_count;
        int pattern_count = pattern->children_count;

        while (((i < code_count) ||
               (j < pattern_count)) && (found || found_omitted))
        {
            found = 0;
            if (j < pattern_count)
            {
                if (pattern->children[j]->is_omitted)
                {
                    found_omitted = 1;
                }
                else if (i < code_count)
                {
                    int match;
                    copy_table(pattern->children[j], pattern);

                    match = match_trees(pattern->children[j],
                                        code->children[i]);
                    if (match)
                    {
                        clear_table(pattern);
                        copy_table(pattern, pattern->children[j]);

                        found = 1;
                        if (found_omitted) found_omitted = 0;
                    }
                }
                else found_omitted = 0;
                j++;
            }
            else if (i < code_count) i++;
        }

        if ((!found) && (!found_omitted))
           pattern->is_matched = 0;
    }

    return pattern->is_matched;
}
```

The VCG produces eleven VCs which can be proved using the specifications of auxiliary functions [5] and the Standard Library routines for strings [11].

4. Conclusion

The deductive verification is a way to establish formally the program correctness. Obviously, the verification method itself should be correct. Apart from theoretical soundness, its implementation also requires validation. The situation when a verification system is written in the target language gives us an opportunity to apply it to itself. This task is of great interest in the case of the C language.

This “work-in-progress” paper describes our steps towards the “verified verifier”. First of all, we adapted the metageneration approach and implemented it with some modification using the C-light language. Then the code was supplemented with ACSL annotations. Let us note that they rely deeply on specifications for the Standard C library (mainly string routines) developed in our previous works. Finally, a series of experiments was performed in order to verify the MetaVCG.

Let us note that studies related to the development of a self-applicable verification system are virtually unknown. In many cases researchers use different languages to implement their systems (like the functional O'Caml in WHY [4]). Others are concentrated on verification of different applications (for example, Hyper-V is studied in detail in the VCC project [3]).

We plan to continue our work on specification and verification of the components of our system. At the moment, only a restricted functionality is expressible in a pure C. Perhaps we will return from C++ API of the Clang compiler to the standard C in order to achieve an ultimate goal – the total verification.

In the introductory section, we also mentioned another possible research area. The formal semantics for C-light and C-kernel could be embedded in a prover based on the higher order logics. After that, some theorems earlier proved manually could be revised with such automatic assistance.

References

[1] Apt K.R., Olderog E.R. Verification of Sequential and Concurrent Programs. – Berlin etc.: Springer, 1991.

[2] 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` .

[3] 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.

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

[5] Kondratyev D., Promsky A. Towards the 'Verified Verifier'. Theory and practice // Proc. Fifth Workshop "Program Semantics, Specification and Verification: Theory and Applications". – Moscow, Russia, June 6, 2014. – P. 68–78.

[6] Maryasov I.V., Nepomnyaschy V.A., Promsky A.V., Kondratyev D.A. Automatic C program verification based on mixed axiomatic semantics // Proc. of Fourth Workshop "Program Semantics, Specification and Verification: Theory and Applications". – Yekaterinburg, Russia, June 24, 2013. – P. 50–59.

[7] Moriconi M., Schwartz R.L. Automatic Construction of Verification Condition Generators From Hoare Logics // Lect. Notes Comput. Sci. – Berlin etc., 1981. – Vol. 115. – P. 363–377.

[8] Nepomniaschy V.A., Anureev I.S., Mikhailov I.N., Promsky A.V. Verification-oriented language C-light // System Informatics. – Novosibirsk, SB RAS Publishing House, 2004. – Issue 9: Formal Methods and Informatics Models. – P. 51–134 (In Russian).

[9] Norrish M. C Formalised in HOL: Thes. Doct. Phylosophy (Computer sci.). – Cambridge, 1998.

[10] Oheimb D. von. Hoare logic for Java in Isabelle/HOL // Concurrency and Computation: Practice and Experience. – 2001. – Vol. 13(13). – URL: `http://isabelle.in.tum.de/Bali/papers/CPE01.html`.

[11] Promsky A.V. C program verification: verification condition explanation and standard library // Automatic Control and Computer Sciences. – 2012. – Vol. 46, No. 7. – P. 394–401.

[12] Promsky A.V. Experiments on self-applicability in the C-light verification system // Bulletin NCC. Series: Computer Science. – Novosibirsk, 2013. – IIS Special Iss. 35. – P. 85–99.