Implementing the MetaVCG approach in the C-light system

Dmitry Kondratyev
A. P. Ershov Institute of Informatics Systems
Novosibirsk, Russia
Email: apple-66@mail.ru

Alexei Promsky
A. P. Ershov Institute of Informatics Systems
Novosibirsk, Russia
Email: promsky@iis.nsk.su

***Abstract*—Among the problems that can confront a verification system developer, the addition of new axiomatic rules is of great interest. Not only theoretical properties of a Hoare logic (first of all, soundness and completeness) can be endangered by such activity, but also the recoding of verification condition generator (VCG) is required in practice. The error-prone process of manual reprogramming can compromise the very idea of verification system. Can we trust a program verified by (possibly) faulty system? While the self-verified system is still a challenge (though some steps towards it are already taken), as for its VCG-part, there is a method providing a greater level of reliability. In 1980, Moriconi and Schwartz represented the MetaVCG approach, regretfully abandoned. In this article, we would like to describe our efforts to implement this approach in the C-light system.**

***Keywords*—Verification, specification, axiomatic semantics, the C-light language, ACSL, MetaVCG**

I. Introduction

The axiomatic semantics serves as a formal basis for deductive program verification [1]. Its implementation usually takes the form of verification condition generator (VCG). When verification is studied for a simple model language, the VCG can be implemented once and for all. But in practice, it may require expansion. What could be the reasons for the expansion?

First, the program language itself may be enriched by new constructions. For example, in our project of the C program verification we introduced a subset, called C-light [8], as a starting point. This subset is quite representative, but still does not cover some low-level aspects of the C language. During the project evolution we added new constructions to the C-light. And yet, new expansions are to come. First of all, the newest standard of the C language is of great interest. Also, the C-light may serve as a basis for the kindred languages (Objective C, C++).

Second, the practical verification can require development of specialized versions of VCG for narrow classes of programs. Compared to the expansion of the language, such specialization (in fact, restriction) is not so obvious. However, it can play crucial role if we try to simplify the verification. Two examples can clarify this point.

As a start, let us consider a pattern code

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

which can be found in some library routines (`qsort`, for example). Its treatment by general axiomatic 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.

Another example relates to the linear algebra programs. A Hoare system for this applied area was developed in the Theoretic programming Lab of our institute quite ago [10]. Given, `M` is a two-dimensional matrix and $e(k, i)$ is an expression depending on matrix indices k and i, consider the following Hoare triple:

$\{Q(M \leftarrow rep(M, mat(e_1, e_2, e_3, e_4), e(s, t)))\}$
```
  for(k = e1; k <= e2; k++)
    for(i = e3; i <= e4; i++)
      M[k][i] = e(k,i);
```
{Q}

where matrix $rep(M, mat(e_1, e_2, e_3, e_4), e(s, t)))$ results from replacement of all elements corresponding to sub-matrix $mat(e_1, e_2, e_3, e_4)$ by expression e. The usual treatment of loops in axiomatic semantics is based on the loop invariants, which have always been one of the annoying features of this approach. But, as you can see here, two invariants for two nested loops were replaced by application of some logical constructs (rep and mat). A complete logical axiomatization of them can be found in [9].

An interim conclusion here is that the expansion of axiomatic semantics (and its VCG implementation, correspondingly) is inevitable in practice. Moreover, an enormous VCG, which covers all possible classes of programs, is hardly an option. A collection of specialized VCG could be a better solution. Thus, any method that can facilitate the reprogramming of the base VCG or development of the new ones, is of great interest for us.

Fortunately, there is a quite appropriate technique. In 1981, Moriconi and Schwartz [6] proposed a method which forms a

This research is partially supported by the Russian Foundation for Basic research (project no. 15-01-05974)

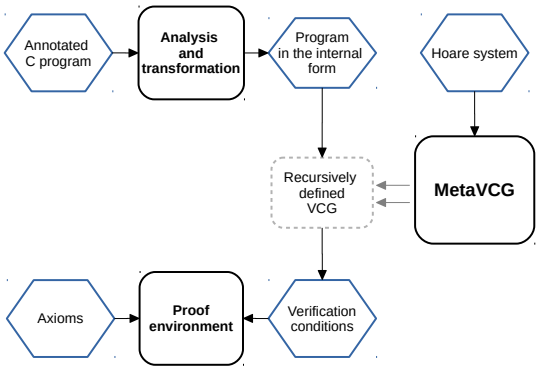

Fig. 1. A "meta-stage" in the verification process

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).

Apart from automatic creation of VCGs, this approach has another benefit for us. An ultimate goal of our project is the development of a self-applicable verification system for the C language [12]. However, at the moment we use the C++ API of the compiler Clang for the "Analysis and transformation" block on Fig. 1. An intermediate translation of Clang AST into the C structures allowed us to implement MetaVCG using the C-light language, thus, promising a more complete self-verification. Some experiments were successfully conducted [13].

In the rest of this paper we give an overview of the MetaVCG approach as well as its implementation in our system.

II. Meta verification condition generation

The method of metageneration proposed by Moriconi and Schwartz [6] consists of two steps. A general axiomatic definition is first transformed into a normal form which, in turn, develops into a recursively defined VCG.

A. 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 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.

B. 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 $\cup_{1 \le i \le n} FragVars(S_i) \subseteq FragVars(S)$.

4) *Dependency ordering*. The Hoare-triple premises of N must satisfy two dependency constraints.
   a. $P_i \stackrel{+}{\Leftarrow} P_j \supset i < j$
   b. $T \stackrel{+}{\Leftarrow} U \wedge \neg(\exists R)U \stackrel{+}{\Leftarrow} R \supset U \equiv Q \vee U$ 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:

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

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.

C. 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:

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

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 [6].

On the other hand, the normal form constraints narrow the class of admissible Hoare systems. Note that axiomatic semantics for C-kernel [5], which is an intermediate language of our project, 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 (2), (4) and equivalent normal (3), (5) form correspondingly:

$$\frac{\{P \wedge B\}\ S_1\ \{Q\}, \quad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \texttt{if}\ (B)\ S_1\ \texttt{else}\ S_2\ \{Q\}} \tag{2}$$

$$\frac{\{P_1\}\ S_1\ \{Q\}, \quad \{P_2\}\ S_2\ \{Q\}}{\{B \supset P_1 \wedge \neg B \supset P_2\}\ \texttt{if}\ (B)\ S_1\ \texttt{else}\ S_2\ \{Q\}} \tag{3}$$

$$\frac{\{P \wedge B\}\ S\ \{P\}, \quad P \wedge \neg B \supset Q}{\{P\}\ \texttt{while}\ (B)\ S\ \{Q\}} \tag{4}$$

$$\frac{\{P_1\}\ S\ \{P\}, \quad P \wedge \neg B \supset Q, \quad P \wedge B \supset P_1}{\{P\}\ \texttt{while}\ (B)\ S\ \{Q\}} \tag{5}$$

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.

III. 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

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

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.

A. 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 II-A) 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 [5] but they do not alter principally the logical structure of the familiar Hoare sentences.

B. 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 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 post-condition, 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 two functions in the rest of this Section. As for the main tree comparison algorithm (programs against patterns), at the moment we use a "greedy" algorithm. Such algorithm can be applied successfully thanks to the simplicity

of the C-light language. Perhaps, consideration of the complete C or complex applied program domain will require a more general approach.

C. Verification examples

When comparing the pattern tree node with a node of the program tree we also match the node attributes, including their identifiers. For convenience, we call the current nodes of the corresponding trees simply as `pattern`, and `code`. If both nodes have identifiers, we must check whether the identifier of `pattern` was early matched against any program construction identifier. This comparison is carried out using a scan of table `match_identifiers`. If the validation fails, then the identifier of `pattern` is associated with the identifier of `code`. So, we want to place this information in the table `match_identifiers`, stored in `pattern`. We use the function `add_identifier`, the annotated definition of which looks like[1]

```
/*@
requires \valid(pattern) && \valid(code);
assigns pattern->table_length;
assigns pattern->match_identifiers[0..1]
           [\old(pattern->table_length)]
           [0..\max(strlen(pattern_identifier),
                        63)];
ensures strncmp(pattern->match_identifiers[0]
                        [pattern->table_length],
                pattern->identifier, 63);
ensures strncmp(pattern->match_identifiers[1]
                        [pattern->table_length],
                pattern->identifier, 63);
ensures pattern->table_length =
                 \old(pattern->table_length)+1;
*/
void add_identifier(struct pattern_node* pattern,
                    struct program_node* code)
{
    strncpy(pattern->match_identifiers[0]
                          [pattern->table_length],
            pattern->identifier, 63);
    strncpy(pattern->match_identifiers[1]
                          [pattern->table_length],
            code->identifier, 63);
    pattern->table_length++;
}
```

Another case study directly concerns the tree matching process. When comparing the pattern tree node with a node of the program tree we also need to correlate the node attributes, including attribute `category`. The function `compare_categories` implements such comparison. If the `category` fields of the current tree nodes are equal, then this function returns 1. Otherwise, it returns 0.

```
/*@
requires \valid(pattern) && \valid(code) &&
             pattern->has_category == 1;
behavior comparable:
assumes strlen(pattern_category) ==
            strlen(code_category)
            &&
            \forall int i;
            0 <= i <= strlen(code_category)
            ==>
            pattern_category[i] ==
                  code_category[i];
    ensures \result == 1;

behavior incomparable:
    assumes
       \exist int i;
       0 <= i <= \min(strlen(pattern_category),
                      strlen(code_category))
       &&
       pattern_category[i] != code_category[i];
    ensures \result == 0;
*/
int compare_categories(
      struct pattern_node* pattern,
      struct program_node* code)
{
    int result = 1;

    if ((pattern->has_category) &&
        (strcmp(pattern->category,
                code->category)
        != 0))
    {
        result = 0;
    }
    return result;
}
```

Unlike the code nodes, a pattern node may omit the information about syntactic constructions it can be compared to, i.e. its `category` field can be empty. Usually it takes place when a pattern can match any sequence (including empty) of program constructs. However, such situation is handled somewhere on the outer level, thus we implicitly suppose that `pattern->has_category == 1` when `compare_categories` is invoked.

Based upon specifications of string routines `strlen`, `strncpy` and `strcmp` from our earlier work [11] the verification of these two functions is quite straightforward. The other parts of MetaVCG involve loop invariants or recursive function calls and, thus, are bulky to be described here.

IV. Conclusion

In our project, we are pursuing two objectives. First, we need a collection of VCGs for various classes of programs to simplify the verification. This will make verification available not only for theoreticians, but also for ordinary programmers. Second, we would like to guarantee the correctness of our method as much as possible. 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.

In order to make the creation of such "convenient and verified verifier" more feasible we adapted the MetaVCG approach. 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

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

in our previous works [11]. Finally, a series of experiments was performed in order to verify the MetaVCG.

It is difficult to carry out a qualitative comparison with related works. It appears that the approach of Moricony and Schwartz has not been used by other researchers. And, despite the fact that there are many verification projects, the 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 complete self-verification. As for the development of specialized VCG, the method of finite iterations, developed by Prof. V.A. Nepomnyaschy [7], is a high priority candidate for implementation. It resembles the axiomatic semantics for linear algebra, mentioned in Introduction, and relieves us of loop invariants.

REFERENCES

[1] Apt K.R., Olderog E.R. Verification of sequential and concurrent programs. — Berlin etc.: Springer, 1991. — 450 p.

[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. — LNCS. — 2009. — Vol. 5674. — P. 23-42.

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

[5] I.V. Maryasov, V.A. Nepomnyaschy, A.V. Promsky, D.A. Kondratyev. Automatic C Program Verification Based on Mixed Axiomatic Semantics // Proc. Fourth Workshop "Program Semantics, Specification and Verification: Theory and Applications". — Yekaterinburg, Russia, June 24, 2013. — pp. 50–59.

[6] 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.

[7] Nepomniaschy V.A. Verification of finite iterations over data structures // Programming. — 2002. — N 1. — pp. 3–12. (In Russian)

[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] Nepomniaschy V.A., Ryakin O.M. Applied methods of program verification. — Moscow, 1988. — 256 p. (In Russian)

[10] Nepomniaschy V.A., Sulimov A.A. Verification of the linear algebra programs in the system SPECTRUM // Cybernetics and system analysis. — 1992. — N 5. — pp. 136–144. (In Russian)

[11] A.V. Promsky. C Program Verification: Verification Condition Explanation and Standard Library // Automatic Control and Computer Sciences, 2012, Vol. 46, No. 7, pp. 394–401.

[12] Promsky A.V. Experiments on self-applicability in the C-light verification system // Bull. Nov. Comp. Center, Comp. Science, 35 (2013), 85–99.

[13] A.V. Promsky Experiments on self-applicability in the C-light verification system. Part 2 // Bull. Nov. Comp. Center, Comp. Science, 37 (2014), 93–105.