Bull. Nov. Comp. Center, Comp. Science, 35 (2013), 85–99

Experiments on self-applicability in the C-light verification system*

A. V. Promsky

Abstract. Development of the C-light verification system is accompanied by various case studies. We have already demonstrated the applicability of our system to some examples from verification competitions. Those programs are connected to verification-difficult issues but, as a rule, they are represented by artificial or trivial pieces of code. Now we can address the more realistic tests. The first series of experiments is based on fragments of the input analyzer/translator of the C-light system. The trials include axiomatization of problematic domains in the prover Simplify, the development of ACSL annotations and inductive reuse of already specified standard library routines. Thus the first step to a self-applicable C program verification system has been taken.

Keywords: C-light, ACSL, Simplify, standard library, axiomatic theory, specification, verification.

1. Introduction

As opposed to traditional testing, the deductive verification represents a formal way to examine the program correctness. But what about correctness of the verification system itself?

The answer to this question 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 obtained remarkable results [9, 10].
2. The program implementations of those theoretical methods should also be checked thoroughly. And again, in addition to usual testing, formal verification here looks desirable. In particular, if the verification system is implemented in the 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.

*Partially supported by RFBR under grant 11-01-00028.

In the Laboratory of Theoretical Programming (IIS) we are developing the C-light verification system. The C-light language covers the major part of the previous standard (C99). In order to avoid the problems of Hoare's logic for the full C, we translate the input programs in a restricted core called C-kernel. The verification condition generator 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 composing parts of our approach [8]. We plan to check those proofs in the future using a higher order logics prover (like HOL, for example).

Practical testing of a verification system is related to the choice of case studies. Previously, we demonstrated [7, 11] that our prototype system is powerful enough to verify some programs from a recognized collection of "verification challenges" [6] or examples from verification competition suite [3]. These programs are usually very simple or artificial.

Now it is time for more realistic experiments, which is possible thanks to recent studies. First, the method of formal verification condition explanation and error localization was developed for C-light [12]. Whereas VCs for simple programs are comprehensible even for a "pen-and-paper" proof, verification of a real code requires the automatic assistance. Even so, counterexamples given by a prover may be unexpectedly complex. The approach mentioned above simplifies analysis when something goes wrong. Second, the specifications written in ACSL [2] were developed for a part of the Standard C library [13]. Every meaningful C program relies on library routines, so these annotations are an important prerequisite.

The self-verification goal has suggested a new test suite to us. The code examples considered in this paper are fragments of the input module of our system. This module translates a C-light program into an equivalent C-kernel program. In fact, it is implemented in C++ using API of the compiler Clang. Thus the complete verification is unachievable; however it is rich in code expressible in C-light, which makes our translator a good test subject.

The rest of the paper is organized as follows: Section 2 contains three examples from the test suite with our comments attached. An overview of verification prerequisites is given in Section 3. They include ACSL annotations and logical axiomatizations. The verification results are summarized in Section 4. Section 5 is a conclusion of the paper.

2. The chosen fragments of a translator

Here we present three functions from the translator source code. We also give some explanations, so that reader could match them against annotations. The annotations themselves will be described later in Section 3.

2.1. The function `deleteSpaces`

This function takes a null-terminated string as its input and erases all blank symbols in the head and tail of the string. Those symbols include *the space character* and *escape sequences*. The function `deleteSpaces` is used actively during the specification analysis. Indeed, the ACSL specifications takes the form of comments and may often contain extra blank symbols between the lexemes "`/*`" and "`*/`" and specification bodies.

```
#include <ctype.h>
#include <string.h>

/*@ requires \valid_string(str);
    assigns \nothing;
    ensures \base_addr(\old(str)) <= \base_addr(str) &&
          \base_addr(str) <= \base_addr(\old(str)) +
          strlen(\old(str));
    ensures strlen(\result) == length &&
          length <= strlen(\old(str));
    ensures \result == NULL ||
           strncmp(\result, str, length - 1) == 0;
*/
char* deleteSpaces(char *str)
{
    int length = strlen(str);
    /*@ loop invariant 0 <= length < strlen(str) &&
                           (str[length - 1] == ' ' ||
                            str[length - 1] == '\f' ||
                            str[length - 1] == '\n' ||
                            str[length - 1] == '\r' ||
                            str[length - 1] == '\t' ||
                            str[length - 1] == '\v' ||
                            isalnum(str[length - 1) == 0);
    */
    while (isspace(str[length - 1])) --length;
    /*@ loop invariant 0 <= length < strlen(\old(str)) &&
                    \old(str) <= str < \old(str) +
                         strlen(\old(str));
    */
    while (*str && isspace(*str))
        ++str, --length;
    char* result = (char*)malloc(1 + length);
    if (NULL != result)
    {
```

```
        strncpy(result, str, length);
        result[length] = '\0';
    }
    return result;
}
```

The implementation is quite straightforward. First, we need to calculate the length of the resulting string. This can be done in two loops. We begin with the input string length. During the first loop, we decrease the variable `length` until we find the rightmost *basic source character*. Thus all tailing blank symbols are implicitly removed from consideration. In the second loop, we also decrease `length` and increase the string pointer until it points to the leftmost basic source character. After that we allocate memory for the resulting string and copy the specification body into it.

2.2. The function `replace`

As an intermediate program representation in our system, we use a special prefix notation. Thus the verification condition generator, which is a stand-alone program, can easily parse it using a recursive descent. In the prefix form the *type specifiers* an *type qualifiers* are separated by the character `'_'` (`unsigned_int`, for example). The LLVM infrastructure and compiler Clang allow us to obtain a string containing information about types, which uses blank symbols as separators. So we need to replace them all with `'_'`.

```
#include <string.h>
#include <ctype.h>

/*@ requires \valid_string(str);
    assigns str[0..length(str)-1];
    ensures \forall int i; (0 <= i < length(str) &&
                            isspace(\old(str[i]))
                           )
                           ==>
                           str[i] == '_';
*/
void replace(char* str)
{
    int length = strlen(str);
    int i;
    /*@ loop invariant
            \forall int j; (0 <= j <= i) &&
                            isspace(\old(str[j]))
                           )
```

```
                            ==>
                            str[j] == '_';
    for (i = 0; i < length; i++)
        if (isspace(str[i])) str[i] = '_';
}
```

The function `replace` takes a string `str` as an input. A single loop is used to check all characters replacing spaces with `'_'`. In fact, the resulting string does not require another conversion when the prefix form is generated.

2.3. The function `getBlockID`

According to the axiomatic semantics of C-kernel, every compound statement should have a unique identifier. During translation into prefix form, a construction of the form {A_1; A_2; ...; A_N;}, where A_i are statements, is transformed into block(ID, A_1, A_2, ..., A_N), where ID is a unique name. The automatic generation of such names is a duty of the function `getBlockID`. The names have the form `BLOCK`n, where $n \in$ [1..`UINT_MAX`). The value `UINT_MAX` is taken from file "`limits.h`".

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

/*@ requires 1 <= id <= UINT_MAX;
    assigns id;
    behavior somewhere_in_the_middle:
        assumes 1<= \old(id) < UINT_MAX;
        ensures id == \old(id) + 1 &&
                strcmp(\result, strcat("BLOCK\0",
                                       ltoa(\old(id)))) == 0;
    behavior too_many_blocks:
        assumes \old(id) == UINT_MAX;
        ensures \result == NULL;
    complete behaviors somewhere_in_the_middle, too_many_blocks;
    disjoint behaviors somewhere_in_the_middle, too_many_blocks;
*/
char* getBlockID()
{
    static unsigned int id = 1;
    char* result = NULL;
    if (id < UINT_MAX)
    {
```

```
        int maxNumberLength = 11;
        char* number = (char*)malloc(maxNumberLength);
        if (NULL != number)
        {
            int blockLength = 6;
            int resultLength;
            int numberLength;

            sprintf(number, "%u", id);
            numberLength = strlen(number);
            resultLength = blockLength + numberLength;

            result = (char*)malloc(resultLength);
            if (NULL != result)
            {
                char block[] = "BLOCK";
                strncpy(result, block, blockLength);
                strncat(result, number, numberLength);
                ++id;
            }
            free(number);
        }
    }
    return result;
}
```

The function `getBlockID` has no parameters. A static variable `id` of type `unsigned int` is declared and initialized to `1`. Every subsequent invocation of `getBlockID` increases the value of `id` until the upper limit `UINT_MAX` is reached. The current value of `id` is used as a suffix n in `BLOCK`n. We use the library `sprintf` to obtain string representations for integers, as well as standard routines `strncpy` and `strncat` for a string copy and concatenation, respectively. If either `id == UINT_MAX` or any of two memory allocations was unsuccessful, `getBlockID` returns `NULL`. The use of the static variable guarantees originality of block names.

3. Verification prerequisites

To verify a C-light program, we need some additional information. First of all, a program should have specifications. All library types and functions addressed in the program require their own specifications. Finally, the concepts from the current problematic domain should be axiomatized so that the prover Simplify could process them.

3.1. ACSL annotations

We use ACSL as a specification language. The paper's volume does not allow us to give a detailed description of this language. Instead, we explain the specifications of our examples. More information about ACSL can be found in [2].

deleteSpaces. The ACSL annotations are written in special C comments beginning with `'/*@'` in contrast to simple comments. The most important ACSL concept is the function contract. The function `deleteSpaces` on Page 87 is preceded by such a contract.

The function pre-condition is given in the `requires` clause. We need a correctly allocated string `str` as an input. This is exactly what the predicate `\valid_string` says.

It is possible to specify that global variables do not change during the execution of `deleteSpaces`. The term `\nothing` in the `assigns` clause means that the function has no visible side-effects.

The function post-condition is expressed by the `ensures` clause. In the presence of several clauses, it means that they are combined by conjunction. The only reason to use many entries is the reader's convenience. The built-in function `\old` is used to address the function arguments in the pre-state (i.e. before the invocation). The function `\base_addr` denotes the pointer to the string head.

So, the post-condition demands that, at the end of the function body, the pointer `str` points somewhere within the limits of the original string. Also the length of the resulting string, which is stored in the variable `length`, can be shorter than the original string length. Finally, either the resulting value is a null pointer or it points to the same substring as `str` does at the end.

The function body contains two loops, so two loop invariants are introduced. The first one states that, at every iteration, the value of `length` should be nonnegative and less than the current length of the string `str` and also the character `str[length - 1]` shold be a blank symbol. Actually, pattern matching for escape sequences in invariant is unnecessary, since it can be deduced from specifications of the library function `isspace`. But, to complete the picture, we made it explicit. Also, we suppose that programs are created by a "wise" user, so we do not take into account exotic escape sequences, like `'\a'` or `'\b'`. The second invariant claims that the length of a new string does not exceed the length of the string `str` and the pointer `str` is within the limits of the original string.

replace. This function is simpler than the previous one, but, in terms of logic, it is more complex because it applies to quantifiers.

The pre-condition on Page 88 is the same — `\valid_string`.

However, some global objects can change during execution of `replace` and this fact is reflected in the `assigns` clause. Moreover, in ACSL it specifies that a function is not allowed to change memory locations other than those explicitly listed. Also note that ACSL provides a convenient way to specify the sets (regions) of values (objects).

The post-condition simply says that every blank symbol of the original string is replaced by the character `'_'`.

Finally, the loop invariant can be expressed by a generalized form of the post-condition.

getBlockID. As we have seen earlier, the function `getBlockID` can either return a valid block name or end up with failure. Any attempt to describe the variety of results together with the reasons can lead to a complicated post-condition. It is possible to express that differently, by using ACSL's *behaviors*. A function can have several behaviors in addition to a general specification. A behavior can have additional `ensures` clauses, but it can also have `assumes` clauses which indicate when the behavior is triggered.

The shared pre-condition (Page 89) expresses the expected limits for `id`.

The `assigns` clause says that `id` is the only object which can change during execution of `getBlockID`.

Behavior `somewhere_in_the_middle` corresponds to a situation when the value of `id` is within capacity of the type `unsigned int`. Then a new value of `id` is by one greater than the previous value. Note that the statement about the resulting string required a trick. Indeed, we used the library function names in specifications (like `strcmp`) thus turning them into logical functions. This works well for the functions with fixed parameter lists. However, we appealed to the function `sprintf` to transform an integer into its string representation, and that function is *variadic*. To avoid inevitable problems, we use the logical function `ltoa` for the same task. This logical name has no direct implementation in the standard library, but it serves as an inverse to the library function `atol`. One would think that the function `itoa` (as a reverse to `atoi`) could be appropriate. But the variable `id` is unsigned, so we need the type `long int` to avoid dangerous coercions from `unsigned int` into `int`.

Behavior `too_many_blocks` models the situation when increment of `id` results in integer overflow. So, we return the `NULL` pointer instead.

In addition, we stipulate that this pair of behaviors is exhaustive.

To conclude this section, let us note the convenience of ACSL. Like its ancestor, the specification language JML, it is based on the idea of using the target language syntax for its own expressions. An immediate gain is that specifications become rather understandable for "ordinary" programmers, who, as a rule, are not eager to learn formalisms used in the verification theory.

3.2. Library specifications and axiomatics

Recently we developed the ACSL specifications for a subset of the Standard C library. Moreover, basing on these specifications, several library functions were successfully verified [7]. Since our three programs are concentrated mainly on character and string manipulations (and memory management, sometimes), let us consider the corresponding examples.

It should be noted that specifications for the library are not limited to functional contracts only. ACSL provides a way to develop logical functions and predicates which can be used later during the proof stage.

ctype.h Here, let us restrict ourselves to the minimal locale. Then, the following logical predicates should be obvious, especially for the reader familiar with the C language.

```
#include "limits.h"
#include "stdio.h"

// if locale = "C"
/*@ predicate ISDIGIT(int _c) = (_c >= '0' && _c <= '9');
    predicate ISLOWER(int _c) = (_c >= 'a' && _c <= 'z');
    predicate ISUPPER(int _c) = (_c >= 'A' && _c <= 'Z');

    predicate ISALPHA(int _c) = (ISUPPER(_c) || ISLOWER(_c));
    predicate ISALNUM(int _c) = (ISALPHA(_c) || ISDIGIT(_c));
    ...
*/
```

As it was said before, the ACSL language encourages the use of the C code in its expressions. Now, having these definitions, we can simply specify the standard function which checks whether a character is alphanumerical.

```
/*@ requires 0 <= c <= UCHAR_MAX || c == EOF;
    ensures \result == ISALNUM(c);
*/
int isalnum(int c);
```

As a precondition, we use the comparison of a character with the standard constants[1]. The postcondition states that the returning value is equal to the value of the predicate `ISALNUM`. Actually, these parameter tests and predicate definitions can constitute an appropriate implementation of `isalnum` (that is why we omitted its body). So, its verification was effortless indeed.

[1] Defined in `limits.h` and `stdio.h`.

string.h From the verifier's point of view, this could be the best part of the library. Indeed, the functions declared in `string.h` are not trivial (like those from `ctype.h`, for example), but at the same time they are merely manipulations on the arrays of characters. This means that they admit portable implementations which do not rely on the system calls or assembly language.

For example, the implementation of the function `strcpy` accompanied by its ACSL specification looks like[2]:

```
/*@ requires \valid_range(s1,0,strlen(s2)) &&
             valid_string(s2);
    assigns s1[0..strlen(s2)];
    ensures strcmp(s1,s2) == 0 && \result == s1;
    ensures \base_addr(\result) == \base_addr(s1);
*/
char *strcpy(char *restrict s1, const char *restrict s2)
{
    char *os1 = s1;
    //@ ghost int i = 0;
    /*@ loop invariant \forall integer j;
                       0 <= j <= i ==> s1[j] == s2[j]; */
    while (*s1++ = *s2++) { /*@ ghost i++; */ }
    return (os1);
}
```

stdlib.h One of the most frequently used library parts is the memory management. Here we begin to face the execution environment dependance. As a result, only a partial verification is achievable. It is partial in the sense that we rely on the hypotheses about system calls or assembly language fragments which cannot be validated. However, as an exercise training we tried to verify some artificial implementations with a simple memory model. One of them is as follows:

```
char HEAP[HEAP_SIZE];

/*@ requires size > 0 && \valid(HEAP);
    assigns next_free;
    ensures next_free == \old(next_free) + size;
    ensures \result ==
            next_free >= HEAP_SIZE ? NULL : (HEAP + next_free);
*/
void *malloc(size_t size) {
```

[2]Here we do not demand the strings be 0-terminated.

```
    static int next_free = 0;
    next_free += size;
    if (next_free >= HEAP_SIZE) return NULL;
    return (HEAP + (next_free - size));
}
```

In this model, the heap is just an array of bytes of an appropriate length. Every successful allocation increases the global index `next_free` by the size of the allocated space. Such an implementation does not provide any heap overhead which helps to determine the size of the region being deallocated. Thus, the corresponding implementation of `free` can only be

```
void free(void* ptr) { return; }
```

4. Verification

Since all verification requirements are met, the rest of the process is quite routine. Programs are translated into the C-kernel language (in the prefix notation), the verification condition generator (VCG) produces VCs which are passed, in turn, to the prover Simplify.

Let us consider the course of life for one of VCs, say a VC contributing to the proof of the loop

```
while (*str && isspace(*str))
    ++str, --length;
```

in the function `deleteSpaces`.

An equivalent code in C-kernel is as follows:

```
while (1) {
    auto int x;
    if(*str) x = isspace(*str) else x = 0;
    if (x) {
        str += 1;
        length -= 1;
    }
    else goto l;
}
l:;
```

The nature of logical AND operator[3] in C together with restrictions on C-kernel requires such a bulky rewriting. On the other hand, this intermediate

[3]It allows for incomplete evaluation if the first argument is sufficient for the whole expression.

translation step is hidden from a user. We demonstrate it only to complete the picture.

Note that the method of assigning unique names to auxiliary variables (like `x` above) during translation is similar to the generation of block identifiers. Thus the verification of `getBlockID` can be useful to us.

To verify a loop is to show that its body preserves a loop invariant. So the VCG will take the invariant from Page 87 and the control expression (can be omitted due to obvious reasons) as a pre-condition. The same invariant will serve as a post-condition. There are two `if`-statements, so the proof tree has four leaves — VCs. One of them after simplifications has the following form:

```
(IMPLIES
    (AND (<= 0 (select MD1 (select MeM1 length)))
         (< (select MD1 (select MeM1 length))
            (strlen (old (select MD1 (select MeM1 str)))))
         (<= (old (select MD1 (select MeM1 str)))
            (select MD1 (select MeM1 str)))
         (< (select MD1 (select MeM1 str))
            (+ (old (select MD1 (select MeM1 str)))
               (strlen (old (select MD1 (select MeM1 str))))))
         (EQ MeM (store MeM1 x |@nc0|))
         (EQ MD2 (store MD1 |@nc0| |@omega|))
         (EQ (select MD2 (select MD2 (select MeM str))) |@0|)
         (EQ MD3 (store MD2 (select MeM x) 0))
         (NEQ 0 (select MD3 (select MeM x)))
         (EQ MD4 (store MD3 (select MeM str)
                            (+ 1 (select MD3 (select MeM str)))))
         (EQ MD (store MD4 (select MeM length)
                           (- (select MD4 (select MeM length)) 1))))
    (AND (<= 0 (select MD (select MeM length)))
         (< (select MD (select MeM length))
            (strlen (old (select MD (select MeM str)))))
         (<= (old (select MD (select MeM str)))
             (select MD (select MeM str)))
         (< (select MD (select MeM str))
            (+ (old (select MD (select MeM str)))
               (strlen (old (select MD (select MeM str)))))))
)
```

As you can see, the original program variables (`x`, `str`, ...) acquire wrappings built of names MeM_i or MD_j. These are so called meta-variables which model the memory in our C-light abstract machine. Details can be found in [8]; suffice it to say that meta-variables behave like arrays or mappings. The Simplify possesses built-in theories for arrays and uninterpreted functions.

Table. Verification results

Experiment	Derived VCs and triples	Approved by Simplify
`deleteSpaces`	7 VCs and 1 dummy triple	✓
`replace`	4 VCs	✓
`getBlockID`	4 VCs	✓

We may also express some of the logical predicates and functions from Section 3.2 in the LISP-notation, so that Simplify could process strings[4]. They can be combined in a file and passed to Simplify as external axioms. After that Simplify can easily prove our VC:

```
Simplify-1.5.4.exe -ax user.ax vc.lisp
1: Valid.
```

The other VCs are validated by analogy. The summary of verification results is given in the table. The dummy triple obtained for `deleteSpaces` needs some explanation. As we have just seen, the `while` loop in the C-kernel program has the tautological control expression. Its negation (FALSE) will become a pre-condition for the resting part of the function `deleteSpaces` (from the loop exit point up to the end) thus making the corresponding Hoare's triple trivially true. However, we must keep that triple in the proof tree because the control is transferred there (`goto l;`). So we have to find an invariant for the label `l`. As soon as an invariant is found, we can safely neglect all VCs that stem from the proof sub-tree of our dummy triple.

5. Conclusion and future work

The deductive verification is a way to establish formally 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 paper describes our first step towards the "verified verifier". A series of experiments was performed in order to verify some parts of a translator from C-light into C-kernel. The work included the development of ACSL annotations and axiomatic theories for problematic domains. Three of our case studies were illustrated here.

To emphasize the actuality, let us note that studies related to this field are almost unknown. In many cases researchers use different languages to

[4]For example, a prover should be informed about the properties of `strlen`.

implement their systems (like the functional O'Caml in WHY [5]). Others are concentrated on verification of different applications (for example, Hyper-V is the main subject of study in the VCC project [4]).

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 introductory section we also mentioned one more area of possible research. The formal semantics for C-light and C-kernel could be embedded in some prover based on the higher order logics. After that, some theorems earlier proved manually could be revised with such an 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. – Available at http://www.frama-c.cea.fr/download/acsl_1.4.pdf

[3] Bormer T., Brockschmidt M., Distefano D., Ernst G., Filliâtre J.-C., Grigore R., Huisman M., Klebanov V., Marché C., Monahan R., Mostowski W., Polikarpova N., Scheben C., Schellhorn G., Tofan B., Tschannen J., Ulbrich M. The COST IC0701 verification competition 2011 // Revised Selected Papers of Int. Conf. FoVeOOS 2011. – Lect. Notes Comput. Sci. – 2011. – Vol. 7421. – P. 3–21.

[4] 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.

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

[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] Maryasov I.V., Nepomnyaschy V.A., Promsky A.V., Kondratyev D.A. 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. – P. 50–59.

[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. – Iss. 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, N 13. – P 1173–1214. – Available at http://isabelle.in.tum.de/Bali/papers/CPE01.html.

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

[12] Promsky A.V. A formal approach to the error localization. – 2012. – 33 p. – (Prep. / A.P. Ershov Institute of Informatics Systems, Novosibirsk, Russia; N 169).

[13] Promsky A.V. C program verification: verification condition explanation and standard library // Automatic Control and Computer Sciences. – 2012. – Vol. 46, N 7. – P. 394–401.