UDC 004.052.42

Verification of Definite Iteration over Arrays with a Loop Exit in C Programs

Maryasov I. V. (A. P. Ershov Institute of Informatics Systems SB RAS)

Nepomniaschy V. A. (A. P. Ershov Institute of Informatics Systems SB RAS)

Kondratyev D. A. (A. P. Ershov Institute of Informatics Systems SB RAS)

This work represents the further development of the method for definite iteration verification [6]. It extends the mixed axiomatic semantics method [1] suggested for C-light program verification. This extension includes a verification method for definite iteration over unchangeable arrays with a loop exit in C-light programs. The method includes an inference rule for the iteration without invariants, which uses a special function that expresses loop body. This rule was implemented in verification conditions generator, which is the part of our C-light verification system. To prove generated verification conditions an induction is applied which is a challenge for SMT-solvers. At proof stage the SMT-solver Z3 is used in our verification system. To overcome mentioned difficulty a rewriting strategy for verification conditions is suggested. It allows to verify the definite iteration automatically using Z3. Also the paper describes the application of the theorem prover PVS for automatic proving of such verification conditions. An example, which illustrates the application of these methods, is considered.

This research is partially supported by RFBR grant 15-01-05974.

Keywords: C-light, loop invariants, mixed axiomatic semantics, definite iteration, arrays, Z3, PVS, specification, verification, Hoare logic

1. Introduction

C program verification is an important task nowadays. A lot of projects (e. g. [2, 3]) propose different solutions. But none of the mentioned above suggests any methods for loop verification without invariants whose construction is a challenge. Therefore, the user has to provide these invariants. In many cases it is a difficult task. Tuerk [12] suggested to use pre- and post-conditions for while-loops but the user still has to construct them himself.

Our method describes a class of loops, which can be verified automatically without any invariants or loop pre- and post-conditions. It deals with a definite iteration of a special form. We extend our mixed axiomatic semantics of the C-light language [1] with a new rule

for verification of such iterations. This extension includes a verification method for definite iteration over unchangeable arrays with a loop exit in C-light programs. The method includes an inference rule for the iteration without invariants, which uses a special function that expresses loop body. This rule was implemented in verification conditions generator, which is the part of our C-light verification system.

At the proof stage, the SMT-solver Z3 [9] and the proof assistant PVS [11] are used. A rewriting strategy for the induction based verification conditions was suggested to prove them in Z3.

2. Definite Iteration and Replacement Operation

The method of loop invariants elimination for definite iteration was suggested in [10]. It includes four cases:

1. Definite iteration over unchangeable data structures without loop exits.
2. Definite iteration over unchangeable data structures with a loop exit.
3. Definite iteration over changeable data structures with a loop exit.
4. Definite iteration over hierarchical data structures with a loop exit.

The first case was considered in [6]. Our paper deals with the second case.

Let us remind the notion of data structures, which contain a finite number of elements. Let $memb(S)$ be the multiset of elements of the structure S and $|memb(S)|$ be the power of the multiset $memb(S)$. For the structure S the following operations are defined:

1. $empty(S) = true$ iff $|memb(S)| = 0$.
2. $choo(S)$ returns an element of $memb(S)$ if $\neg empty(S)$.
3. $rest(S) = S'$, where S' is a structure of the type of S and $memb(S') = memb(S) \setminus \{choo(S)\}$ if $\neg empty(S)$.

Sets, sequences, lists, strings, arrays, files, and trees are typical examples of the data structures.

Let $\neg empty(S)$, then $vec(S) = [s_1, s_2, \ldots, s_n]$ where $memb(S) = \{s_1, s_2, \ldots, s_n\}$ and $s_i = choo(rest^{i-1}(S))$ for $i = 1, 2, \ldots, n$.

Consider the statement

$$\textbf{for x in S do v} := \textbf{body}(\mathbf{v}, \mathbf{x})\ \textbf{end}$$

where S is a structure, x is the variable of the type "an element S", v is a vector of loop variables which does not contain x and *body* represents the loop body computation, which

does not modify x and S, and which terminates for each $x \in memb(S)$. The loop body can contain only the assignment statements, the **if** statements and the **break** statements. Such **for** statement is named a definite iteration.

The operational semantics of such statement is defined as follows. Let v_0 be the vector of initial values of variables from v. If $empty(S)$ then the result of the iteration $v = v_0$. Otherwise, if $vec(S) = [s_1, s_2, \ldots, s_n]$, then the loop body iterates sequentially for x taking the values $s_1, s_2, \ldots, s_n$.

To express the effect of the iteration let us define a replacement operation $rep(v, S, body, n)$, where $rep(v, S, body, 0) = v$, $rep(v, S, body, i) = body(rep(v, S, body, i-1), s_i)$ for all $i = 1, 2, \ldots, n$ if $\neg empty(S)$.

A number of theorems, which express important properties of the replacement operation, were proved in [10].

The inference rule for definite iterations has the form:

$$\frac{E, SP \vdash \{\exists v'\ P(v \leftarrow v')\ \wedge v = rep(v', S, body)\}\ \mathbf{A};\{Q\}}{E, SP \vdash \{P\}\ \mathbf{for\ x\ in\ S\ do\ v} := \mathbf{body(v, x)\ end\ A};\{Q\}}$$

Here A are program statements after the loop. We use forward tracing: we move from the program beginning to its end and eliminate the leftmost operator (at the top level) applying the corresponding rule of the mixed axiomatic semantics [1] of C-light. E is the environment which contains an information about current function (its identifier, type and body) which is verified, an information about current block and label identifier if **goto** statement occurred earlier. SP is program specification which includes all preconditions, postconditions, and invariants of loops and labeled statements.

Let S be a one-dimensional array of n elements. Consider the special case of definite iteration

$$\mathbf{for\ (i = 0; i < n; i++)\ v} := \mathbf{body(v, i)\ end}$$

where $\mathbf{v} := \mathbf{body(v, i)}$ consists of assignment statements, **if** statements and **break** statements.

In order to generate verification conditions we have to determine v, $body(v, i)$, and the function rep.

Let the loop body has the form

$$\{\mathbf{x_1} = \mathbf{expr_1}(\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_k});$$
$$\mathbf{x_2} = \mathbf{expr_2}(\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_k});$$
$$\ldots$$
$$\mathbf{x_k} = \mathbf{expr_k}(\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_k});\}$$

where $expr_j (j = 1, 2, \ldots k)$ are some C-light expressions.

The vector v of loop variable consists of all variables from left parts of assignment statements: $v = (x_1, x_2, \ldots, x_k)$. From the statements before the loop, we can get the initial value of v and obtain the first axiom for *rep*:

$$rep(v, S, body, 0) = (x_{1_0}, x_{2_0}, \ldots, x_{k_0})$$

where $x_{j_0}, j = 1, 2, \ldots, k$ are the initial values of x_j before the loop execution.

At the next step, we make consecutive substitutions

$$x_1 = expr_1(x_1, x_2, \ldots, x_k);$$

$$x_2 = expr_2(expr_1(x_1, x_2, \ldots, x_k), x_2, \ldots, x_k);$$

$$\ldots$$

$$x_k = expr_k(expr_1(x_1, x_2, \ldots, x_k), expr_2(expr_1(x_1, x_2, \ldots, x_k), x_2, \ldots, x_k), \ldots, x_k);$$

And then in the right parts $rep((x_1, x_2, \ldots, x_k), S, body, i-1)$ is substituted for x_j.

For each **if** statement of the form $\mathbf{if}\,(\mathbf{e(i, x_1, x_2, \ldots, x_k)})\{\mathbf{A};\}\,\mathbf{else}\,\{\mathbf{B};\}$, where **A** and **B** are compound statements consisting of assignment statements, two axioms are added to the output of verification conditions generator:

$$\forall x_1 \forall x_2 \ldots \forall x_k \; e(i, x_1, x_2, \ldots, x_k) \Rightarrow A^*$$

$$\forall x_1 \forall x_2 \ldots \forall x_k \; \neg e(i, x_1, x_2, \ldots, x_k) \Rightarrow B^*$$

where A^* and B^* are obtained by consecutive substitutions as described above.

The **break** statement could appear at the top level of the loop or in the **if** statement. The first case is obvious, it means that the loop iterates no more than once and all the statements after **break** in the loop body are ignored. Therefore, the function *rep* is defined for $i = 0, 1$.

The second case means that for some j such that $0 < j \leq n$ a loop exit occurs and such j is defined by the condition of the **if** statement. Therefore, for all i such that $j \leq i \leq n$

$$rep((x_1, x_2, \ldots, x_k), S, body, i) = rep((x_1, x_2, \ldots, x_k), S, body, j)$$

In this case the following axiom is added:

$$\forall x_1 \forall x_2 \ldots \forall x_k \; e(i, x_1, x_2, \ldots, x_k) \Rightarrow (A^* \wedge (\forall l \, i < l \Rightarrow A^*))$$

For the case when the **break** statement is located in the **else** statement, the negation of e is used.

3. Example

Let us demonstrate the application of our method. Consider the following function **search**. For a given integer *key* it finds its first occurrence in the given array of integers *arr* consisting of *length* elements. If *key* does not occur in *arr* the function returns -1.

The annotated (in SMT-LIB v2 syntax of Z3) C-light program has the form:

```
/* (assert (> length 0)) */
int search(int* arr, int length)
{
    auto int result = -1;
    for (i = 0; i < length; i++)
     if (key == arr[i])
     {
      result = i;
      break;
     };
    return result;
}
/* (assert (or
            (forall ((i Int))
             (implies (and (< -1 i) (< i length))
                      (and (not (= key (select arr i))) (= result -1))))
            (exists ((r Int))
             (implies (and (< -1 r) (< r length))
                      (forall ((i Int))
                       (implies (and (< -1 i) (< i r))
                                (and (not (= key (select arr i)))
                                     (= key (select arr r))
                                     (= result r)))))))) */
```

In this function $v = (result)$ and its initial value before the iteration is -1. Also note that in Z3 all functions must be total. Therefore, the first axiom is:

```
(declare-fun rep (Int Int (Array Int Int) Int) Int)
(assert (forall ((i Int) (key Int) (arr (Array Int Int)) (result Int))
         (implies (< i 1)
                  (= (rep result key arr i) -1))))
```

According to our method described in Sec. 2, the second and the third axioms are:

```
(assert (forall ((i Int) (key Int) (arr (Array Int Int)) (result Int))
         (implies (and (< 0 i) (= (select arr (- i 1)) key))
```

```
                (and (= (rep result key arr i) (- i 1))
                     (forall ((j Int))
                      (implies (< i j)
                               (= (rep result key arr j) (- i 1))))))))
(assert (forall ((i Int) (key Int) (arr (Array Int Int)) (result Int))
         (implies
                (and (< 0 i)
                     (not (= (select arr (- i 1)) key)))
                (= (rep result key arr i) (rep result key arr (- i 1))))))
```

Z3 is the SMT-solver but our task is to check a validity of verification conditions, not satisfiability. Therefore, the verification conditions generator produces the negation of the verification condition:

```
(assert (not
         (forall ((result Int) (key Int) (length Int) (arr (Array Int Int)))
          (implies (and
                    (> length 0)
                    (= result (rep result key arr length)))
                   (or
                    (forall ((i Int))
                     (implies (and (< -1 i) (< i length))
                              (and (not (= key (select arr i)))
                                   (= result -1))))
                    (exists ((r Int))
                     (implies (and (< -1 r) (< r length))
                              (forall ((i Int))
                               (implies (and (< -1 i) (< i r))
                                        (and (not (= key (select arr i)))
                                             (= key (select arr r))
                                             (= result r))))))))))))
```

And then we expect the answer “unsat” which means that the negation is unsatisfiable therefore the verification condition is true.

However, Z3 does not support proofs by induction, which appears inevitably in our veri-

fication method. In this example we get the answer "unknown" which means that Z3 is not able to determine whether the formula is satisfiable or not. Rustan Leino suggested a rewriting strategy and a heuristic for when to apply it to verify simple inductive theorems [5].

4. Working with Induction in Z3

To prove by induction some proposition $\forall n\ P(n)$ we try to prove the equivalent formula $\forall n\ (\forall k\ k < n \Rightarrow P(k)) \Rightarrow P(n)$. In this way we rewrite the verification condition for Z3 be able to prove it. We should add an extra axiom (induction hypothesis) of the form $\forall n\ \forall k\ k < n \Rightarrow P(k)$ and modify the verification condition by adding a base case of induction $P(1)$. In our case of definite iteration over unchangeable one-dimensional arrays the inductive variable is always the length of array. Therefore, the verification conditions generator is able to rewrite the verification condition which contains a *rep* function automatically.

In the example from Sec. 3 this extra axiom has the form:

```
(assert (forall ((result Int) (key Int) (length Int) (len Int)
                 (arr (Array Int Int)))
  (implies (and
            (> len 0)
            (> length len)
            (= result (rep result key arr len)))
           (or
            (forall ((i Int))
             (implies (and (<= 0 i) (< i len))
                      (and (not (= key (select arr i))) (= result -1))))
            (exists ((r Int))
             (implies (and (<= 0 r) (< r len))
                      (forall ((i Int))
                       (implies (and (<= 0 i) (< i r))
                                (and (not (= key (select arr i)))
                                     (= key (select arr r))
                                     (= result r)))))))))))
```

And the base case is when $length = 1$. Therefore we add the second verification condition which is obtained from the one in Sec. 3 by replacing the first conjunct `(> length 0)` with `(=`

```
length 1).
```

After adding this axiom and the second verification condition, Z3 immediately returns the answer "unsat" which means that the verification condition is true. Thus, the program is partially correct.

5. Using PVS to Prove Inductive Verification Conditions

Another approach to the implementation of our method is based on the meta verification condition generation (MetaVCG) [8] for building verification conditions and using PVS for its proving. Detailed information about the MetaVCG can be found in [4].

The PVS language allows us to define the theory, which is the input argument of PVS theorem prover. The PVS language is a functional programming language supplemented with constructs for defining higher-order logic sentences. Thus, a PVS theory contains type definitions, functions, and formulas. The PVS theorem prover can be applied to a certain formula of the theory or to all formulas of the theory.

Let us consider the process of verification of the example from Sec. 3. The following theory was generated:

```
search: THEORY
BEGIN
  rep(result:int, key:int, arr:ARRAY[nat->int], i:nat) : RECURSIVE int =
     IF (i < 1) THEN result
     ELSE (IF ((0 < i) AND (arr(i-1) = key)) THEN rep(i-1, key, arr, i-1)
           ELSE rep(result, key, arr, i-1) ENDIF) ENDIF
  MEASURE i
  vc: LEMMA
      FORALL (result:int, key:int, length:nat, arr:ARRAY[nat -> int]):
             ((length > 0) AND (result = rep(-1, key, arr, length)))
      IMPLIES
      ((FORALL (i:int): ((0 <= i) AND (i < length) AND (not (key = arr(i))))
       IMPLIES
       (result = -1))
      OR
      (EXISTS (r:int): ((0 <= r) AND (r < length))
```

```
        IMPLIES
        (FORALL (i:int): ((0 <= i) AND (i < r) AND (not (key = arr(i)))
                          AND (key = arr(r))))
         IMPLIES (result = r)))
END search
```

Note that it is necessary to provide a measure for the recursive function *rep*. The measure is a well-founded relation. As mentioned in Sec. 4 the length of array is the appropriate measure for the definite iteration over unchangeable arrays with a loop exit, therefore the measure could be provided to PVS automatically.

PVS has special inference rules, which allow to use induction. In the case under consideration the construct `(induct-and-simplify "length")` is used. It tells the prover to apply the induction with the variable *length*. This leads to successful automatic proving of two formulas: the base case and the induction step.

6. Conclusion

This paper represents an extension of our system [7] for C-light program verification. In the case of definite iteration over unchangeable arrays with a loop exit, this extension allows us to generate verification conditions without loop invariants.

This generation is based on the described inference rule for the C-light **for** statement which introduces the replacement operation. It expresses definite iteration in the special form described in the paper.

The suggested rewriting strategy for the induction allowed us to prove obtained verification conditions in Z3. Also they are automatically proved in PVS.

The rewriting strategy allowed Z3 to prove automatically the partial correctness of the example from [6]. It iterates over an array of integers and for a given integer computes the number of its occurrences in this array. Also, PVS was able to prove automatically the partial correctness of this example.

We plan to improve the suggested algorithm of *rep* function determination for the case of nested **if** statements and to prove its correctness. Also, we will consider the case of loop elimination for changeable data structures and verify automatically classical array sorting programs.

References

1. Anureev I. S., Maryasov I. V., Nepomniaschy V. A. C-programs Verification Based on Mixed Axiomatic Semantics // Automatic Control and Computer Sciences. — 2011. — Vol. 45 — Issue 7. — P. 485–500.
2. Cohen E., Dahlweid M., Hillebrand M., Leinenbach D., Moskal M., Santen T., Schulte W., Tobies S. VCC: A Practical System for Verifying Concurrent C // Proc. of 22nd Int. Conf. TPHOLs. — LNCS. — 2009. — Vol. 5674. — P. 23–42.
3. Filliâtre J.-C., Marché C. Multi-prover Verification of C Programs // Proc. of 6th ICFEM. — LNCS. — 2004. — Vol. 3308. — P. 15–29.
4. Kondratyev D. A. The Extension of the MetaVCG Approach by Semantic Mark-up Concept // Proc. of the Int. Workshop-conf. "Tools & Methods of Program Analysis". — St. Petersburg, 2015. — P. 107–118.
5. Leino K. R. M. Automating Induction with an SMT Solver // Proc. of 13th Int. Conf. VMCAI. — LNCS. — 2012. — Vol. 7148. — P. 315–331.
6. Maryasov I. V., Nepomniaschy V. A. Loop Invariants Elimination for Definite Iterations over Unchangeable Data Structures in C Programs // Modeling and Analysis of Information Systems. — 2015. Vol. 22 — Issue 6. — P. 773–782.
7. 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. — Vol. 48 — Issue 7. — P. 407–414.
8. Moriconi M., Schwarts R. L. Automatic Construction of Verification Condition Generators From Hoare Logics // Automata, Languages and Programming. — LNCS. — 1981. — Vol. 115. — P. 363–377.
9. Moura L. de, Bjørner N. Z3: An Efficient SMT Solver // Proc. of 14th Int. Conf. TACAS 2008. — LNCS. — 2008. — Vol. 4963. — P. 337–340.
10. Nepomniaschy V. A. Verification of Definite Iteration over Hierarchical Data structures // Proc. of FASE/ ETAPS 1999. — LNCS. — 1999. — Vol. 1577. — P. 176–187.
11. Owre S., Rushby J. M., Shankar N. PVS: A Prototype Verification System // 11th Int. Conf. CADE 1992. — LNAI. — 1992. — Vol. 607. — P. 748–752.
12. Tuerk T. Local Reasoning about While-Loops // VSTTE 2010. Workshop Proceedings. — 2010. — P. 29–39.