Bull. Nov. Comp. Center, Comp. Science, 23 (2005), 85–99
© 2005 NCC Publisher

Symbolic verification method for definite iterations over tuples of altered data structures*

V. A. Nepomniaschy

Abstract. In order to extend the area of application of the symbolic verification method [19, 20, 21, 22, 23], definite iterations over tuples of altered data structures are introduced and reduced to the standard definite iterations. This reduction is extended to definite iterations including the exit statement. The generalization of the symbolic verification method allows us to apply it to pointer programs. As a case of study, programs over doubly-linked lists are considered. A program that merges in-place ordered doubly-linked lists is verified by the symbolic method.

1. Introduction

The axiomatic approach to program verification is based on the Hoare method and consists of the following stages: annotating programs by pre-, post-conditions and loop invariants; generating verification conditions with the help of proof rules and proving the verification conditions [6]. The loop invariant synthesis is an important problem [8] which is far from being overcome [26]. In the functional approach to program verification, loops are annotated by functions expressing the loop effect [10]. However, the synthesis of such a function remains a difficult problem in most cases [14]. Attempts to extract loop invariants from programs by using special tools [3] are found to be successful only for quite simple kinds of invariants.

Two ways of overcoming the difficulties of program verification are considered. In papers on code certification [16, 28], verification of simple but important program properties is considered in the framework of the axiomatic approach. However, the invariant synthesis problem imposes an essential limitation on the practical use of this method [28]. An other way of overcoming the difficulty is to use loops of a special form which allows to simplify the verification process. A special form of loops called simple is proposed in [1]. The simple loops are similar to for-loops in that they contain the only control variable called the loop parameter, and also its alteration in a given finite domain does not depend on other variables modified by the loop body. Therefore, the simple loops are definite iterations because of their termination for all values of input variables. Although the reduction of *for*-loops to *while*-loops is often used for verification, attempts to use the

*This work was supported in part by the Russian Foundation for Basic Research under grant 04-01-00114.

specific character of *for*-loops in the framework of the axiomatic approach should be noted [4, 5, 7]. In the framework of the functional approach, a general form of the definite iteration as an iteration over all elements of an arbitrary structure has been proposed in [27] where spreading of such iterations in practical programming has been justified.

A symbolic method for verification of for-loops with the statement of assignment to array elements as a loop body has been proposed in [17, 18], and it has also been extended to the definite iterations over data structures without restrictions on the iteration bodies in [19, 20]. This method is based on using a replacement operation that represents the loop effect in a symbolic form and allows us to express the loop invariant. The symbolic method uses a special technique for proving verification conditions containing the replacement operation. Along with data structures defined in [27], the symbolic method allows us to use hierarchical data structures that are constructed from given structures [20]. Moreover, the symbolic verification method is applied to a new kind of definite iterations called iterations over tuples of data structures [21, 22]. These iterations allow us to represent in a compact and natural form iterations with several input data structures. This method has been successfully applied to verification of programs over arrays and files [20, 21, 22]. However, an attempt to apply this method to pointer programs has revealed the following problem. The data structures can be modified by the iteration body, and, therefore, the loops are not simple. In [23] the symbolic verification method has been extended to definite iterations over altered data structures and applied to verification of programs over singly-linked lists.

The purpose of this paper is to develop the symbolic method for a satisfactory solution of this problem in the case of iterations over tuples of altered data structures and to apply it to pointer program verification. An outline of the symbolic verification method for definite iterations over hierarchical data structures and over tuples of unaltered data structures is given in Section 2 and Section 3, respectively. Definite iterations over tuples of altered data structures and their reduction to standard definite iterations are described in Section 4. Definite iterations over tuples of altered data structures that contain the exit statement are introduced, and their reduction to the standard definite iterations is described in Section 4, too. Application of the symbolic method to verification of programs over doubly-linked lists is presented in Section 5, where a list merge program is considered. Advantages and prospects of the symbolic verification method are discussed in Section 6.

2. Definite iteration over hierarchical data structures

We introduce the following notation. Let $\{s_1, \ldots, s_n\}$ be a multiset consisting of elements $s_1, \ldots, s_n$, $U_1 - U_2$ be the difference of multisets U_1 and U_2,

$U_1 \bigcup U_2$ be the union of multisets, and $|U|$ be the power of a finite multiset U. The empty set is denoted by $\emptyset$. Let $[v_1, \ldots, v_m]$ denote a vector consisting of elements $v_i\,(1 \leq i \leq m)$ and $\varnothing$ denote the empty vector. A concatenation operation $con(V_1, V_2)$ is defined in the usual fashion for vectors V_1 and V_2. For a function $f(x)$ we assume that $f^0(x) = x, f^i(x) = f(f^{i-1}(x))$ $(i = 1, 2, \ldots)$.

Let us remind the notion of a data structure [27]. Let $memb(S)$ be a finite multiset of elements of a structure S, $empty(S)$ be a predicate "$memb(S)$ is empty", $choo(S)$ be a function which returns an element of $memb(S)$, $rest(S)$ be a function which returns a structure S' such that $memb(S') = memb(S) - \{choo(S)\}$. The functions $choo(S)$ and $rest(S)$ will be undefined if and only if $empty(S)$. This definition abstracting from the way of the determination of the functions $choo(S)$ and $rest(S)$, is quite flexible. For example, if a tree is defined as a data structure, a tree traversal method is fixed. So, such different traversal methods result in different data structures.

Let us remind a definition of useful functions related to the structure S [19]. Let $vec(S)$ denote a vector $[s_1, \ldots, s_n]$ such that $s_i = choo(rest^{i-1}(S))$ $(i = 1, \ldots, n)$ in the case of $\neg empty(S)$ and $memb(S) = \{s_1, \ldots, s_n\}$. The vector $vec(S)$ is empty if $empty(S)$. The function $vec(S)$ defines such an unfolding of the structure S that uniquely gives its use. Structures S_1 and S_2 are called equivalent $(S_1 = S_2)$ when $vec(S_1) = vec(S_2)$. The following functions $head(S)$ and $last(S)$ will be undefined in the case of $empty(S)$. A function $head(S)$ returns a structure such that $vec(head(S)) = [s_1, \ldots, s_{n-1}]$ if $vec(S) = [s_1, \ldots, s_n]$ and $n \geq 2$. If $n = 1$, then $empty(head(S))$. Let $last(S)$ be a partial function such that $last(S) = s_n$ if $vec(S) = [s_1, \ldots, s_n]$. Let $str(s)$ denote a structure S which contains the only element s. A concatenation operation $con(S_1, S_2)$ is defined in [19] so that

$$con(vec(S_1), vec(S_2)) = vec(con(S_1, S_2)).$$

Let $con(s, S) = con(str(s), S)$ and $con(S, s) = con(S, str(s))$. In the case of $\neg empty(S)$, $con(choo(S), rest(S)) = con(head(S), last(S)) = S$. Moreover, the property $head(rest(S)) = rest(head(S))$ provided $\neg empty(rest(S))$ results from [19].

Our aim is to introduce parameters in the definition of the structure S. To this end we remind the rules for construction of a hierarchical structure S from given structures $S_1, \ldots, S_m$ [20]. We will use $T(S_1, \ldots, S_m)$ to denote a term constructed from the data structures S_i $(i = 1, \ldots, m)$ with the help of the functions *choo*, *last*, *rest*, *head*, *str*, *con*. For a term T which represents a data structure, we denote the function $|memb(T)|$ by $lng(T)$. The function can be calculated by the following rules:

$lng(S_i) = |memb(S_i)|$,

$lng(con(T_1, T_2)) = lng(T_1) + lng(T_2)$,

$lng(rest(T)) = lng(head(T)) = lng(T) - 1$,
$lng(str(s)) = 1$.
Let a hierarchical data structure $S = STR(S_1, \ldots, S_m)$ be defined by the functions $choo(S)$ and $rest(S)$ constructed with the help of conditional $if - then - else$, superposition and Boolean operations from the following components:

— terms not containing $S_1, \ldots, S_m$;

— the predicate $empty(S_i)$ and the functions $choo(S_i), rest(S_i), last(S_i)$, $head(S_i)\ (i = 1, \ldots, m)$;

— terms of the form $STR(T_1, \ldots, T_m)$ such that $\sum_{i=1}^{m} lng(T_i) < \sum_{i=1}^{m} lng(S_i)$;

— an undefined element ω.

Note that the undefined value ω of the functions $choo(S)$ and $rest(S)$ means $empty(S)$.

Let us suppose that the iteration body consists of a sequence of assignment and conditional statements. The iteration body is represented as the vector assignment statement $v := body(v, x)$, where x is the iteration parameter, v is a vector of other variables, $body(v, x)$ is a vector of conditional expressions constructed with the help of the operation $if - then - else$. Such a representation is formed by a sequence of suitable substitutions which replace both conditional statements by conditional expressions and a sequence of assignment statements by one vector assignment statement.

Let us consider the following definite iteration over an unaltered data structure S:

$$\textbf{for } x \textbf{ in } S \textbf{ do } v := body\,(v, x) \textbf{ end} \qquad (1)$$

where x is a variable called a loop parameter, v is a data vector of the loop body ($x \notin v$) and the iteration body $v := body(v, x)$ does not change the structure S. The result of this iteration is an initial value v_0 of the vector v if $empty(S)$. Let $\neg empty(S)$ and $vec(S) = [s_1, \ldots, s_n]$. Then the iteration body iterates sequentially for x defined as $s_1, \ldots, s_n$.

Let us remind the definition of the replacement operation $rep(v, S, body)$ which presents the effect of iteration (1) [19]. Let its result for $v = v_0$ be a vector v_n such that $n = 0$ provided $empty(S)$, $v_i = body(v_{i-1}, s_i)$ for all $i = 1, \ldots, n$ provided $\neg empty(S)$ and $vec(S) = [s_1, \ldots, s_n]$. The following theorem which results from [19] presents useful properties of the replacement operation.

Theorem 1.
1.1. Iteration (1) is equivalent to the multiple assignment statement
$v := rep(v, S, body)$.
1.2. $rep(v, con(S_1, S_2), body) = rep(rep(v, S_1, body), S_2, body)$.
1.3. $rep(v, str(s), body) = body(v, s)$.

Corollary 1.
1.1. $\neg empty(S) \rightarrow rep(v, S, body) = body(rep(v, head(S), body), last(S))$.
1.2. $\neg empty(S) \rightarrow rep(v, S, body) = rep(body(v, choo(S)), rest(S), body)$.

The replacement operation allows us to formulate the following proof rule without invariants for iteration (1). Let $R(y \leftarrow exp)$ be the result of substitution of an expression *exp* for all occurrences of a variable y into a formula R. Let $R(vec \leftarrow vexp)$ denote the result of a synchronous substitution of the components of an expression vector *vexp* for all occurrences of corresponding components of a vector *vec* into a formula R.

rl1. $\{P\}prog\{Q(v \leftarrow rep(v, S, body))\} \vdash$
$\{P\}prog;$ **for** x **in** S **do** $v := body(v, x)$ **end** $\{Q\}$,

where P is a pre-condition, Q is a post-condition which does not depend on the loop parameter x, *prog* is a program fragment, and {P} *prog* {Q} denotes partial correctness of a program *prog* with respect to P and Q.

The following corollary is evident from Theorem 1.1.

Corollary 2. The proof rule $rl1$ is derived in the standard system of proof rules for usual statements including the multiple assignment statement.

Projections of vectors $body(v, x)$ and $rep(v, S, body)$ on a variable y are denoted by $body_y(v, x)$ and $rep_y(v, S, body)$, respectively.

3. Iterations over tuples of unaltered data structures

Let us remind a definition of a definite iteration over a tuple of unaltered data structures $S_1, \ldots, S_m$ (possibly hierarchical) [21]. We will use a function $sel(x_1, \ldots, x_m)$ for selection of one structure from the structures $S_1, \ldots, S_m$, where $x_i \in memb(S_i) \bigcup \{\omega\}$ $(i = 1, \ldots, m)$. The function $sel(x_1, \ldots, x_m)$ returns an integer j $(1 \leq j \leq m)$ such that $x_j \neq \omega$, and also $sel(\omega, \ldots, \omega)$ is undefined. In the case of $x_j \neq \omega$ and $x_i = \omega$ for all $i \neq j$ $(i = 1, \ldots, m)$, $sel(x_1, \ldots, x_m) = j$ and the definition of the function *sel* can be omitted.

Let us consider the iteration over a tuple of structures $S_1, \ldots, S_m$ of the form

for x_1 **in** $S_1, \ldots, x_m$ **in** S_m **do** $t := sel(x_1, \ldots, x_m); v := body\,(v, x_t, t)$ **end** (2) where v is a data vector ($x_i \notin v$ for all $i = 1, \ldots, m$) and the iteration body does not change the structures $S_1, \ldots, S_m$. If $empty(S_i)$ for each $i = 1, \ldots, m$, then the iteration result is an initial value v_0 of the vector v. Otherwise, we assume $x_i = choo(S_i)$ for each $i = 1, \ldots, m$, where $choo(S_i) = \omega$ provided $empty(S_i)$. A new value v_1 of the vector v is defined so that $v_1 = body(v_0, x_t, t)$. The structure S_t is replaced by the structure $rest(S_t)$, and the other structures S_i ($i \neq t$) are not changed. The process is applied to v_1 and the resulted structures until all structures become empty. The resulted value v_d ($d = \sum_{i=1}^{m} \mid memb(S_i) \mid$) of the vector v is assumed to be the result of iteration (2).

Here the purpose is to reduce iteration (2) to iteration (1) with the help of hierarchical structures. We use the following notation in order to define a hierarchical structure

$$S = STR(S_1, \ldots, S_m)$$

from the structures $S_1, \ldots, S_m$ and the function $sel(x_1, \ldots, x_m)$. Let
$EMPTY = (empty(S_1) \wedge \ldots \wedge \ empty(S_m))$,
$t_1 = sel(choo(S_1), \ldots, choo(S_m))$,
$REST = STR(S_1, \ldots, rest(S_{t_1}), \ldots, S_m)$ provided $\neg EMPTY$. Then
$(choo(S), rest(S)) =$ **if** $EMPTY$ **then** (ω, ω)**else** $((choo(S_{t_1}), t_1), REST)$.
Notice that this definition is consistent with the definition of hierarchical structures from Section 2, since the quantifiers bounded by the set $\{1, \ldots, m\}$, can be expressed by applying conjunction or disjunction m times, and $empty(S) \equiv (choo(S) = \omega)$. The following theorem is similar to Theorem 1 [21].

Theorem 2. Iteration (2) over a tuple of unaltered data structures $S_1, \ldots, S_m$ is equivalent to the iteration

$$\textbf{for } (x, \tau) \textbf{ in } S \textbf{ do } v := body\,(v, x, \tau) \textbf{ end} \tag{3}$$

where the hierarchical structure $S = STR(S_1, \ldots, S_m)$ is defined by means of the function $sel(x_1, \ldots, x_m)$.

Let us consider the following definite iteration over a tuple of unaltered data structures with a body including the statement of termination of the iteration EXIT:

```
for $x_1$ in $S_1, \ldots, x_m$ in $S_m$ do $t := sel(x_1, \ldots, x_m); v := body(v, x_t, t);$
if $cond(x_1, \ldots, x_m)$ then $EXIT$ end                                    (4)
```

where $x_i \notin v$ ($i = 1, \ldots, m$), the condition

$$cond(x_1, \ldots, x_m)(x_i \in memb(S_i) \cup \{\omega\})$$

does not depend on variables from v, and the iteration body does not change the structures S_j $(j = 1, \dots, m)$.

We will use b_1 to denote $cond(choo(S_1), \dots, choo(S_m))$. Let us describe operational semantics of iteration (4). If $EMPTY$, then the result of iteration (4) is an initial value v_0 of the vector v. Otherwise, the result of iteration (4) is $v_1 = body(v_0, choo(S_{t_1}), t_1)$, when b_1 is true. If b_1 is false, then this process is continued with $v = v_1$ and the structure $rest(S_{t_1})$ instead of S_{t_1}, when the other structures S_i $(i \neq t_1)$ are not changed. A value v_l resulting from this process is the result of iteration (4).

Here the purpose is to reduce iteration (4) to iteration (1) with the help of hierarchical structures. Let us define a hierarchical structure $T = STR(S_1, \dots, S_m)$ with the help of the function $sel(x_1, \dots, x_m)$ and the condition $cond(x_1, \dots, x_m)$.
Let $(choo(T), rest(T)) =$ **if** $EMPTY$ **then** (ω, ω) **else** $((choo(S_{t_1}), t_1),$
if b_1 **then** ω **else** $REST)$. The following theorem is similar to Corollary 1 [21].

Theorem 3. Iteration (4) over a tuple of unaltered data structures $S_1, \dots, S_m$ is equivalent to the iteration

$$\textbf{for } (x, \tau) \textbf{ in } T \textbf{ do } v := body(v, x, \tau) \textbf{ end} \qquad (5)$$

where the hierarchical structure $T = STR(S_1, \dots, S_m)$ is defined with the help of the function $sel(x_1, \dots, x_m)$ and the condition $cond(x_1, \dots, x_m)$.

4. Iterations over tuples of altered data structures

The definite iteration over tuples of altered data structures has the form (2), where the structure S_i can depend on variables from the vector v $(i = 1, \dots, m)$. Let w_i denote a vector consisting of all variables on which the structure $S_i = S_i(w_i)$ depends $(i = 1, \dots, m)$. If S_i does not depend on variables from v, then w_i is the empty vector and it can be omitted. Let $Init$ denote an admissible set consisting of initializations of variables from v. The set $Init$ can depend on a program containing iteration (2).

Let $v_j (w_{ij}$, respectively) denote a vector consisting of values of variables from v (w_i, respectively). Let us say that v_j extends $w_{ij} (v_j \supset w_{ij})$ if, for each variable y from the vector w_i and its value $y_j \in w_{ij}$, the property $y_j \in v_j$ holds.

Let us define operational semantics of iteration (2) for a vector v_0 consisting of initial values of variables from v such that $v_0 \in Init$. Let $S_{i0} = S_i(w_{i0})$ provided $w_{i0} \subset v_0$ $(i = 1, \dots, m)$, $d = \sum_{i=1}^{m} \mid memb(S_{i0}) \mid$.

We introduce the following notation:
$vec_0(S_{i0}) =$ **if** $\rceil empty(S_{i0})$ **then** $vec(S_{io})$ **else** $\emptyset$,
$t_j = sel(choo(vec_{j-1}(S_{10})), \dots, choo(vec_{j-1}(S_{m0})))$,
$vec_j(S_{i0}) =$ **if** $i \neq t_j$ **then** $vec_{j-1}(S_{i0})$ **else** $rest(vec_{j-1}(S_{i0}))$,

$VEC_{j0} = [vec_j(S_{10}), \dots, vec_j(S_{m0})]$,
$v_j = body(v_{j-1}, choo(vec_{j-1}(S_{t_j0})), t_j)\ (j = 1, \dots, d)$.

We impose a restriction RTR1 on the iteration (2) such that at the j-th step of the iterative process the iteration body does not change VEC_{j0} for $j = 1, \dots, d-1$. Therefore, after the j-th step of the iterative process, the vector of undelivered elements of the structure S_i coincides with $vec_j(S_{i0})$ when $v = v_{j-1}$ and $x_i = choo(vec_{j-1}(S_{i0}))\ (i = 1, \dots, m)$.

The result of the iterative process is defined to be v_d. The following claim follows immediately from the operational semantics of iteration (2).

Claim 1. Iteration (2) with an initial value $v_0 \in Init$ of the vector v provided $S_i = S_i(w_i)$ and $w_{i0} \subset v_0\ (i = 1, \dots, m)$ is equivalent to the program

$$w_1 := w_{10}; \dots, w_m := w_{m0};\ \textbf{for}\ x_1\ \textbf{in}\ S_1(w_{10}), \dots, x_m\ \textbf{in}\ S_m(w_{m0})$$
$$\textbf{do}\ t := sel(x_1, \dots, x_m); v := body(v, x_t, t)\ \textbf{end}. \tag{6}$$

It should be noted that, in the case of $w_k = \emptyset$, the statement $w_k := w_{k0}$ is omitted in (6) and $S_k(w_{k0})$ is replaced by S_k.

Let us define operational semantics of iteration (4) for a vector $v = v_0 \in Init$ in the case when the structure S_i depends on variables from the vector $w_i\ (i = 1, \dots, m)$. Let $b_j = cond(choo(vec_{j-1}(S_{10})), \dots, choo(vec_{j-1}(S_{m0})))$ $(j = 1, \dots, d)$. We impose a restriction RTR2 on iteration (4) such that at the j-th step of the iterative process the iteration body does not change VEC_{j0} when $\rceil b_1 \wedge \dots \wedge \rceil b_j\ (j = 1, \dots, d-1)$. v_d is defined as the result of the iterative process in the case of $\rceil b_1 \wedge \dots \wedge \rceil b_d$. Otherwise, there exists j such that
$2 \le j \le d \wedge \rceil b_1 \wedge \dots \wedge \rceil b_{j-1} \wedge b_j$ or $j = 1 \wedge b_1$. In this case v_j is defined to be the result of the iterative process.

The following claim follows immediately from the operational semantics of iteration (4).

Claim 2. Iteration (4) with an initial value $v_0 \in Init$ of the vector v provided $S_i = S_i(w_i)$ and $w_{i0} \subset v_0\ (i = 1, \dots, m)$ is equivalent to the program

$$w_1 := w_{10}; \dots, w_m := w_{m0};\ \textbf{for}\ x_1\ \textbf{in}\ S_1(w_{10}), \dots, x_m\ \textbf{in}\ S_m(w_{m0})\ \textbf{do}$$
$$t := sel(x_1, \dots, x_m); v := body(v, x_t, t);\ \textbf{if}\ cond(x_1, \dots, x_m)\ \textbf{then}\ EXIT$$
$$\textbf{end}. \tag{7}$$

5. Case of study: iterations over doubly-linked lists

5.1. Specification means

In order to generate verification conditions of pointer programs, we will use the method from [11]. A section of a heap which presents a computer memory is associated with a pointer type. Let L be a set of heap elements to which pointers can refer. An element to which a pointer p refers, is denoted by $p\uparrow$ in programs or by $\subset p\supset$ in specifications, or by $L\subset p\supset$ in specifications when the element belongs to L. The predicate $\subset p\supset\in L$ is denoted by $pnto(L,p)$. Let L be a set of records with the field k. We use $upd(l,k,e)$ to denote an element resulted from the element l by replacing the field $l.k$ with the value of the expression e. Let $upd(L,\subset p\supset,k,e)$ be a set resulted from the set L by replacing the field $L\subset p\supset .k$ with the value of the expression e. To generate verification conditions, we will use for the statement $q\uparrow .k := e$ its equivalent form $L := upd(L,\subset q\supset,k,e)$ in the case of $pnto(L,q)$.

In this section we assume that a set L that forms a double-linked list consists of records with the fields *key*, *next* and *prev*. The *key* field contains an integer that serves as an identification name for an element. The *next* and *prev* fields contain a pointer or nil.

The predicate $reach_n(L,r,q)$ $(reach_p(L,r,q)$, respectively) means that the element $\subset q\supset$ is reached from the element $\subset r\supset$ in the set L via pointers from the *next* (*prev*, respectively) field. Let $root_n(L)$ $(root_p(L)$, respectively) be a pointer to a head element of the set L with respect to n-reachability (p-reachability, respectively), i.e. such an element from which all other elements of the set L can be reached via pointers from the *next* (*prev*, respectively) field. Let $l = last_n(L)$ $(l = last_p(L)$, respectively) be such an element of the set L that the field $l.next$ ($l.prev$, respectively) contains *nil* or a pointer to an element which does not belong to the set L.

The predicate $dset(L)$ means that the set L is doubly-linked, i.e. there exist pointers $root_n(L)$ and $root_p(L)$ as well as elements $last_n(L)$ and $last_p(L)$ such that $last_p(L) = \subset root_n(L)\supset$ and $last_n(L) = \subset root_p(L)\supset$. Notice that there exist the only pointer $root_n(L)$ $(root_p(L)$, respectively) and the only element $last_n(L)$ $(last_p(L)$, respectively) for the doubly-linked set L. A doubly-linked set L can be considered as a structure L such that $choo(L) = \subset root_n(L)\supset$ and $rest(L)$ results from the set L by removing the element $choo(L)$.

The predicate $dlist(L)$ means that the set L is a doubly-linked list, i.e. $dset(L)$ and $last_n(L).next = last_p(L).prev = nil$. An other useful kind of doubly-linked sets, so-called semilists, is defined by the predicate $dpset(L)$ which means $dset(L)$ and $last_n(L).next = nil$.

Let us define several useful operations over doubly-linked sets. A doubly-linked set which contains the only element l is denoted by $dset(l)$. Let us consider disjoint doubly-linked sets L_1 and L_2 such that

$$\neg pnto(L_1, last_n(L_2).next) \text{ and } \neg pnto(L_2, last_p(L_1).prev).$$

We define their concatenation as a doubly-linked set $L = con(L_1, L_2)$ such that $L = L_1' \cup L_2'$, where L_1' (L_2', respectively) results from L_1 (L_2, respectively) by placing the pointer $root_n(L_2)$ $(root_p(L_1)$, respectively) into the field $last_n(L_1).next$ ($last_p(L_2).prev$, respectively). Let us extend the definition of $con(L_1, L_2)$ such that $con(L_1, L_2) = L_i$, where i=1 if $L_2 = \emptyset$, and i=2 if $L_1 = \emptyset$. We consider $con(L, l)$ and $con(l, L)$ to be a short form for $con(L, dset(l))$ and $con(dset(l), L)$, respectively.

A sequence which is the projection of the doubly-linked set L on the *key* field in the direction given by pointers in the *next* field, is denoted by $L.key$. In the case of the empty set L, let $L.key$ be the empty sequence.

For a sequence seq of different integers we denote by $sord(seq)$ a predicate whose value is true, if the sequence seq has been sorted in the order $<$, and false otherwise. Let $set(seq)$ be the set of all elements of the sequence seq.

5.2. Merging ordered doubly-linked lists

The following annotated program prog1 merges in-place ordered doubly-linked lists L_1 and L_2 into an ordered list L, where sets of keys of elements of L_1 and L_2 are disjoint.

```
$\{P\}z := nil;\ y_1 := root_n(L_1);\ y_2 := root_n(L_2);$
**for** $x_1$ **in** $L_1, x_2$ **in** $L_2$ **do** $t := sel(x_1, x_2);$
**if** $z \neq nil$ **then begin** $z\uparrow.next := y_t; y_t\uparrow.prev := z$ **end**;
$z := y_t; y_t := x_t.next;$ **if** $x_1 = \omega \vee x_2 = \omega$ **then** $EXIT$ **end** $\{Q\}$,
```

where $sel(x_1, x_2) =$ **if** $x_1.key < x_2.key$ **then** 1 **else** 2,
$y_t =$ **if** $t = 1$ **then** y_1 **else** y_2,
$P : L_1 = L_{10} \wedge L_2 = L_{20} \wedge dlist(L_{10}) \wedge dlist(L_{20}) \wedge L = L_{10} \cup L_{20} \wedge sord(L_{10}.key) \wedge sord(L_{20}.key) \wedge set(L_{10}.key) \cap set(L_{20}.key) = \emptyset$,
$Q : dlist(L) \wedge sord(L.key) \wedge set(L.key) = set(L_{10}.key) \cup set(L_{20}.key)$.
It should be noted that the program prog1 has variables $L_1, L_2, L, y_1, y_2, z, x_1, x_2, t$, and the elements $z \uparrow$ and $y_t \uparrow$ can be written in the form $L \subset z \supset$ and $L \subset y_t \supset$, respectively. Moreover, y_i is a pointer to a scanned element of the list L_i $(i = 1, 2)$, and z is a pointer to an element of the set L which has been selected by the function *sel* at the previous step of the iterative process.

The iteration contained in prog1 is considered for the initialization $Init$: $L_1 = L_{10}, L_2 = L_{20}, L = L_{10} \cup L_{20}, z = nil, y_1 = root_n(L_{10}), y_2 = root_n(L_{20})$, where $dlist(L_{10})$ and $dlist(L_{20})$ hold because the precondition P is true.

Claim 3. The restriction RTR2 holds for the iteration from prog1 under the condition of the initialization $Init$.

Proof. Let $v = v_0 \in Init$. We will use the induction on $j = 1, \ldots, d-1$. If $j = 1$, then $z = nil$ and the set $L = L_{10} \cup L_{20}$ does not change after the first step of the iterative process. Therefore, the restriction RTR2 holds for $j = 1$. Let us suppose that $j > 1$ and $\rceil b_1 \wedge \ldots \wedge \rceil b_j$. Two elements $L \subset z \supset$ and $L \subset y_t \supset$ of the set L can be changed at the j-th step of the iterative process. Detecting that the element $L \subset z \supset$ has been selected at the (j-1)-th step and the element $L \subset y_t \supset$ has been selected at the j-th step, we see that these elements do not belong to VEC_{j0}. Claim 3 follows from this and the inductive hypothesis.

By Claim 2, the program prog1 is equivalent to the following program prog2 for the initialization $Init$:
$\{P\}z := nil;\ y_1 := root_n(L_1);\ y_2 := root_n(L_2); L_1 := L_{10}, L_2 := L_{20};$
for x_1 **in** L_{10}, x_2 **in** L_{20} **do** $t := sel(x_1, x_2);$
if $z \neq nil$ **then begin** $z\uparrow.next := y_t; y_t\uparrow.prev := z$ **end**;
$z := y_t; y_t := x_t.next;$ **if** $x_1 = \omega \vee x_2 = \omega$ **then** $EXIT$ **end** $\{Q\}$.

By Theorem 3, the program prog2 is equivalent to the following program prog3:
$\{P\}z := nil;\ y_1 := root_n(L_1);\ y_2 := root_n(L_2); L_1 := L_{10}, L_2 := L_{20};$
for (x, τ) **in** S **do if** $z{\neq}nil$ **then begin** $z\uparrow.next{:=}\ y_\tau; y_\tau\uparrow.prev := z$ **end**;
$z := y_\tau; y_\tau := x.next$ **end** $\{Q\}$,
where the hierarchical structure $S = STR(L_{10}, L_{20})$ is defined in the following way:
$(choo(S), rest(S)) =$ **if** $EMPTY$ **then** (ω, ω) **else** $((choo(L_{t_10}), t_1),$
if b_1 **then** ω **else** $REST)$,
$t_1 = sel(choo(L_{10}), choo(L_{20})), b_1 = (choo(L_{10}) = \omega \vee choo(L_{20}) = \omega)$,
$REST =$ **if** $t_1 = 1$ **then** $STR(rest(L_{10}), L_{20})$ **else** $STR(L_{10}, rest(L_{20}))$.

The body of the iteration that is contained in prog3 can be written in the form $(L, y_1, y_2, z) := body(L, y_1, y_2, z, x, \tau)$, where $body(L, y_1, y_2, z, x, \tau) =$
(**if** $z \neq nil$ **then** $upd(upd(L, \subset z \supset, next, y_\tau), \subset y_\tau \supset, prev, z)$ **else** L,
if $\tau = 1$ **then** $x.next$ **else** y_1, **if** $\tau = 1$ **then** y_2 **else** $x.next$, y_τ).

The following verification condition VC is generated from prog3 with the help of the proof rule rl1.
$VC : P(L, L_1, L_2, L_{10}, L_{20}) \to Q(L', L_{10}, L_{20})$,
where $L' = rep_L((L, root_n(L_1), root_n(L_2), nil), S, body)$.

In order to prove the condition VC by induction, we replace doubly-linked lists by semilists. The verification condition VC immediately follows from the property
$prop(L, L_1, L_2, L_{10}, L_{20}) = (P'(L, L_1, L_2, L_{10}, L_{20}) \to Q'(L', L_{10}, L_{20}))$,
where $P'(L, L_1, L_2, L_{10}, L_{20}) : L_1 = L_{10} \wedge L_2 = L_{20} \wedge dpset(L_{10}) \wedge dpset(L_{20}) \wedge$
$L = L_{10} \cup L_{20} \wedge \rceil pnto(L_{20}, choo(L_{10}).prev) \wedge \rceil pnto(L_{10}, choo(L_{20}).prev) \wedge$
$sord(L_{10}.key) \wedge sord(L_{20}.key) \wedge set(L_{10}.key) \cap set(L_{20}.key) = \emptyset$,
$Q'(L, L_{10}, L_{20}) : dpset(L) \wedge sord(L.key) \wedge choo(L).key = choo(L_{t_10}).key \wedge$
$choo(L).prev = choo(L_{t_10}).prev \wedge set(L.key) = set(L_{10}.key) \cup set(L_{20}.key)$.

Claim 4. The property $prop(L, L_1, L_2, L_{10}, L_{20})$ holds.

Proof. Let us suppose $t_1 = 1$. We will use induction on $k = \mid memb(S) \mid$ $(k \geq 2)$. If $k = 2$, then $\mid memb(L_{10}) \mid = 1$ and $L' = con(L_{10}, L_{20})$. Indeed, the structure $STR(rest(L_{10}), L_{20})$ consists of the only element $(choo(L_{20}), 2)$ and $L' = rep_L((L_{10} \cup L_{20}, root_n(L_{10}), root_n(L_{20}), nil), S, body) =$
$upd(upd(L_{10} \cup L_{20}, \subset root_n(L_{10}) \supset, next, root_n(L_{20})), \subset root_n(L_{20}) \supset, prev,$
$root_n(L_{10}))$ by Theorem 1.3 and Corollary 1.2. Therefore, from $P'(L, L_1, L_2)$ it follows that $dpset(con(L_{10}, L_{20}))$ and $sord(con(L_{10}, L_{20}).key)$.
$Q'\ (L', L_{10},\ L_{20})$ follows immediately from this.

Let us suppose that $k > 2$ and $prop(L, L_1, L_2, L_{10}, L_{20})$ holds for $\mid memb(S) \mid < k$. Two cases are possible.
1. $t_2 = 1$. Then $L' = \{choo(L_{10})\} \cup L''$, where $L'' =$
$rep_L((rest(L_{10}) \cup L_{20}, root_n(rest(L_{10})), root_n(L_{20}), nil),$
$STR(rest(L_{10}), L_{20}), body) = rep_L((rest(L_{10}) \cup L_{20}, choo(rest(L_{10})).next,$
$root_n(L_{20}), root_n(rest(L_{10}))), STR(rest^2(L_{10}), L_{20}), body)$.
By the inductive hypothesis for L'', it follows from
$P'(L'', L_1, L_2, rest(L_{10}), L_{20})$ that $Q'(L'', rest(L_{10}), L_{20})$.
Therefore, $L' = con(choo(L_{10}), L'')$. $Q'(L', L_{10}, L_{20})$ follows from this,
$sord(L''.key), choo(L_{10}).key < choo(rest(L_{10})).key$ and
$\subset root_n(L'') \supset .key = choo(rest(L_{10})).key$.

2. $t_2 = 2$. Then $L' =$
$\{upd(choo(L_{10}), next, root_n(L_{20}))\} \cup upd(L'', \subset root_n(L'') \supset, prev, root_n(L_{10}))$,
where
$L'' = rep_L((rest(L_{10}) \cup L_{20}, root_n(rest(L_{10})), root_n(L_{20}), nil),$
$STR(rest(L_{10}), L_{20}), body) =$
$rep_L((rest(L_{10}) \cup L_{20}, choo(L_{10}).next, choo(L_{20}).next, root(L_{20})),$
$STR(rest(L_{10}), rest(L_{20})), body)$.
By the inductive hypothesis for L'', it follows from
$P'(L'', L_1, L_2, rest(L_{10}), L_{20})$ that $Q'(L'', rest(L_{10}), L_{20})$. Therefore,
$L' = con(choo(L_{10}), L'')$. $Q'(L', L_{10}, L_{20})$ follows from this,
$sord(L''.key)$, $choo(L_{10}).key < choo(L_{20}).key$ and $choo(L'') = choo(L_{20})$.

6. Conclusion

A generalization of the symbolic method that allows modification of data structures in tuples by the iteration body and termination of the iteration by EXIT statement under a condition is described in this paper. The generalization extends application domains of the symbolic method since generalized iterations allow us to represent a new important case of while-loops and to apply the method to verification of pointer programs with several input data structures.

Restrictions RTR1 and RTR2 imposed on iterations over tuples of altered data structures allow us to change only current and previously processed values of iteration parameters and to retain the important property of termination of the iterations. The idea of reduction of these iterations to the iterations over tuples of unaltered data structures by introducing special variables that store initial values of variables from the iteration body was found to be fruitful as demonstrated by Claims 1,2 and by the example in Section 5.2.

Instead of loop invariants, the symbolic method uses properties of the replacement operation which, as a rule, are simpler than the invariants. To represent the invariants, new notions related to a specific character of programs to be verified are often required. Proof of verification conditions including the replacement operation does not require introduction of such notions. Instead, it uses properties of both hierarchical data structures and the replacement operation that are expressed by Theorem 1 and Corollary 1.

Verification of pointer programs has been considered in [2, 9, 12, 24, 25] in the framework of axiomatic approach. Interesting examples of pointer program verification which include a program for in-place merge of ordered singly-linked lists have been given in [2], where a verification method based on the method proposed in [15] has been developed. An application of the tool Isabelle/HOL to pointer program verification using the method [2] has been described in [12]. Verification of a program over doubly-linked lists for elimination of elements with zero keys has been presented in [25], where a Hoare-like logic oriented to pointer program verification has been proposed. This logic has been formalized in [24] as the separation logic. The symbolic verification method has two advantages as compared with [2] and [25], since it does not use both loop invariants and special list representations. For Hoare-style verification of pointer programs, decidable logics and simulation of data structures including doubly-linked lists have been adapted in [9]. Such a new verification method uses loop and simulation invariants. Correctness proofs of some routines over singly-linked lists have been considered in [13] as a case study of a reliable library of object-oriented components.

The symbolic verification method is promising for applications. It is suggested to extend this method to definite iterations over tuples of altered data structures that contain the exit statement under a condition which can depend on variables from the iteration body.

References

[1] Abd-El-Hafiz S.K., Basili V.R. A knowledge-based approach to the analysis of loops // IEEE Trans. of Software Eng. — 1996. — Vol. 22, N 5. — P. 339–360.

[2] Bornat R. Proving pointer programs in Hoare logic // Proc. MPC 2000. — Lect. Notes Comput. Sci. — 2000. — Vol. 1837. P. 102–126.

[3] Ernst M.D., Cockrell J., Griswold W.G., Notkin D. Dynamically discovering likely program invariants to support program evolution // IEEE Trans. Software Eng. — 2001. — Vol. 27, N 2. — P. 99–123.

[4] Gries D., Gehani N. Some ideas on data types in high-level languages // Comm. ACM. — 1977. — Vol. 20, N 6. — P. 414–420.

[5] Hehner E.C.R., Gravell A.M. Refinement semantics and loop rules // Proc. FM'99. — Lect. Notes Comput. Sci. — 1999. — Vol. 1709. — P. 1497–1510.

[6] Hoare C.A.R. An axiomatic basis of computer programming // Comm. ACM. — 1969. — Vol. 12, N 10. — P. 576–580.

[7] Hoare C.A.R. A note on the for statement // BIT. — 1972. — Vol. 12, N 3. — P. 334–341.

[8] Hoare C.A.R. The verifying compiler: a grand challenge for computing research // Proc. PSI 2003. — Lect. Notes Comput. Sci. — 2003. — Vol. 2890. — P. 1–12.

[9] Immerman N. et al. Verification via structure simulation // Proc. CAV 2004. — Lect. Notes Computer Sci. — 2004. — Vol. 3114. — P. 281–294.

[10] Linger R.C., Mills H.D., Witt B.I. Structured Programming: Theory and Practice. — Addison–Wesley, 1979.

[11] Luckham D.C., Suzuki N. Verification of array, record and pointer operations in Pascal // ACM Trans. on Programming Languages and Systems. — 1979. — Vol. 1, N 2. — P. 226–244.

[12] Mehta F., Nipkow T. Proving pointer programs in higher-order logic // Proc. CADE-19. — Lect. Notes Comput. Sci. — 2003. — Vol. 2741. — P. 121–135.

[13] Meyer B. Towards practical proofs of class correctness // Proc. ZB 2003. — Lect. Notes Comput. Sci. — 2003. — Vol. 2651. — P. 359–387.

[14] Mills H.D. Structured programming: retrospect and prospect // IEEE Software. — 1986. — Vol. 3, N 6. — P. 58–67.

[15] Morris J.M. A general axiom of assignment // Lect. Notes of Internat. Summer School "Theoretical foundations of programming methodology". — D. Reidel, 1982. — P. 25–41.

[16] Necula G.C. Proof-carrying code // Proc. 24th Annual ACM Symp. on Principles of Programming Languages. — ACM Press, 1997. — P. 106–119.

[17] Nepomniaschy V.A. Loop invariant elimination in program verification // Programming and Computer Software. — 1985. — N 3. — P. 129–137 (English translation of Russian Journal "Programmirovanie").

[18] Nepomniaschy V.A. On problem–oriented program verification // Programming and Computer Software. — 1986. — N 1. — P. 1–9.

[19] Nepomniaschy V.A. Symbolic verification method for definite iteration over data structures // Information Processing Letters. — 1999. — Vol. 69. — P. 207–213.

[20] Nepomniaschy V.A. Verification of definite iteration over hierarchical data structures // Proc. FASE/ETAPS'99. — Lect. Notes Comput. Sci. — 1999. — Vol. 1577. — P. 176–187.

[21] Nepomniaschy V.A. Symbolic verification method for definite iteration over tuples of data structures // Joint NCC& IIS Bull., Ser.: Comput. Sci. — 2001. — Is. 15. — P. 103–123.

[22] Nepomniaschy V.A. Verification of definite iteration over tuples of data structures // Programming and Computer Software. — 2002. — N 1. — P. 1–10.

[23] Nepomniaschy V.A. Symbolic verification method for definite iteration over altered data structures // Programming and Computer Software. — 2005. — N 1. — P. 1–12.

[24] O'Hearn P., Reynolds J., Yang H. Local reasoning about programs that alter data structures // Proc.CSL 2001. — Lect. Notes Comput. Sci. — 2001. — Vol. 2142. — P. 1–19.

[25] Reynolds J.C. Reasoning about shared mutable data structure // Proc. Symp. in celebration of the work of C.A.R. Hoare, Oxford, 1999.

[26] Stark J., Ireland A. Invariant discovery via failed proof attempts // Proc. LOPSTR'98. — Lect. Notes Comput. Sci. — 1999. — Vol. 1559. — P. 271–288.

[27] Stavely A.M. Verifying definite iteration over data structures // IEEE Trans. Software Engineering. — 1995. — Vol. 21, N 6. — P. 506–514.

[28] Whalen M., Schumann J., Fischer B. Synthesizing certified code // Proc. FME 2002. — Lect. Notes Comput. Sci. — 2002. — Vol. 2391. — P. 431–450.