Verification of Definite Iteration over Hierarchical Data Structures

V.A. Nepomniaschy

Institute of Informatics Systems
Russian Academy of Sciences, Siberian Division
6, Lavrentiev ave., Novosibirsk 630090, Russia
Fax: (07-383-2)32-34-94
vnep@iis.nsk.su

Abstract. A method of verifying a definite iteration over hierarchical data structures is proposed. The method is based on a replacement operation which expresses the definite iteration effect in a symbolic form and belongs to a specification language. The method includes a proof rule for the iteration without invariants and inductive proof principles for proving verification conditions which contain the replacement operation. As a case study, a parallel replacement operation for arrays is considered in order to simplify the proof of verification conditions. Examples which illustrate the application of the method are considered.

1 Introduction

Formal program verification that means the proof of consistency between programs and their specifications is successfully developed. The axiomatic style of verification is based on the Hoare method [7] which includes the following stages: constructing pre-, post-conditions and loop invariants; deriving verification conditions with the help of proof rules and proving them. The functional style of verification proposed by Mills and others [1,9] assumes that each loop is annotated with a function expressing the loop effect. The functions are closely related to the loop invariants but a difference can be noticed [4]. In both the approaches loop annotation remains a difficult problem especially for programs over complex data structures such as arrays, files, trees and pointers. Simplifying verification of such programs is an important problem. Difficulties of verification have been noted both for the functional approach to programs over arrays [10] and for the axiomatic one to programs over pointers [5].

One way to simplify verification of such programs is to impose restrictions to loop forms. Loops can be divided in two groups called definite and indefinite iterations. For- and while-loops of Pascal are typical examples. A definite iteration is an iteration over all elements of a list, set, file, array, tree or other data structure. A general form of the definite iteration is proposed in [16]. In [2,6,8] some advantages of for-loops over unordered and linear ordered sets are discussed for the axiomatic approach, and proof rules which take into account the specific

J.-P. Finance (Ed.): FASE'99, LNCS 1577, pp. 176–187, 1999.

character of the for-loops are proposed. In [16] the functional method for verifying the definite iteration in the general form is described. However, annotation of the definite iteration remains a difficult problem for programs over complex data structures.

In [12,13] we proposed a symbolic method of verifying the definite iteration of such a form as loops over linear ordered sets which had the statement of assignment to array elements as the loop body. The main idea of the method is to use the symbols of invariants instead of the invariants in verification conditions and a special technique based on the loop properties for proving the verification conditions. In [14] we extended the symbolic method to the definite iteration over data structures without restrictions on the loop bodies.

The purpose of this paper is to develop a symbolic method for the definite iteration over hierarchical data structures, which allows us to represent some cases of while-loops. In Section 2 the notion of hierarchical data structures is defined and useful functions over the structures are introduced. A definite iteration and a replacement operation which represents the effect of the iteration by means of a symbolic form are described in Section 3 along with a proof rule without invariants for the iteration. Inductive proof principles for proving assertions which contain the replacement operation are presented in Section 4. A case of study of loop bodies over arrays and a parallel replacement operation which allows us to simplify verification condition proofs are considered in Section 5. Examples which illustrate the application of the symbolic method are given in Section 6. In conclusion, results and prospects of the symbolic verification method are discussed. Appendix contains proofs of two theorems from Section 5.

This work is supported in part by INTAS-RFBR grant 95-0378.

2 Data Structures

We introduce the following notation. Boolean operations are denoted by symbols $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication), $\neg$ (negation), $\leftrightarrow$ (equivalence). We suppose all free variables to be bound by universal quantifiers in axioms, theorems and other formulas. Let $\{s_1, s_2, \ldots, s_n\}$ be a multiset (sometimes a set) which consists of elements $s_1, \ldots, s_n$. Here $s \in T$ denotes the membership of s in the multiset T. Let $T_1 - T_2$ be the difference of multisets T_1 and T_2. For the function $f(x)$ we denote $f^0(x) = x$, $f^i(x) = f(f^{i-1}(x))$ $(i = 1, 2, \ldots)$.

Let us remind the notion of a data structure which contains a finite number of elements [16]. Let $memb(S)$ be the multiset of the elements of the structure S, and $\mid memb(S) \mid$ be the power of the multiset $memb(S)$. The following three operations are defined for the structure S: $empty(S)$ is a predicate whose value is true if $memb(S)$ is empty and false otherwise; $choo(S)$ is a function which returns an element s of $memb(S)$; $rest(S)$ is a function which returns a structure S' of the same type as S such that $memb(S') = memb(S) - \{choo(S)\}$. The functions $choo(S)$ and $rest(S)$ will be undefined if and only if $empty(S)$. Typical examples of the structures are sets, sequences, lists, strings, arrays, files and trees.

We introduce a number of useful functions related to the structure S in the case of $\neg empty(S)$. We denote $s_i = choo(rest^{i-1}(S))$ for $i = 1, \ldots, n$ provided $\neg empty(rest^{n-1}(S))$ and $empty(rest^n(S))$. So, $memb(S) = \{s_1, s_2, \ldots, s_n\}$ and $last(S)$ is a partial function such that $last(S) = s_n$. Let $str(s)$ denote a structure S which contains the only element s.

Let $M = [m_1, \ldots, m_k]$ denote a vector which consists of elements $m_i (1 \leq i \leq k)$. We will use $pred(M, j)\,(j = 1, \ldots, k)$ to denote the set $\{m_i \mid 1 \leq i < j\}$ and the empty set if $j > 1$ and $j = 1$, respectively. We will consider the vector $M = [m_1, \ldots, m_k]$ as a structure such that $choo(M) = m_1, rest(M) = [m_2, \ldots, m_k]$ (if $k \geq 2$), $empty(rest(M))$ (if $k = 1$). We consider $m \in M$ to be a shorthand for $m \in memb(M)$. Let $con(M_1, M_2)$ be a concatenation operation of vectors M_1 and M_2.

For the structure S we assume that $vec(S) = [s_1, \ldots, s_n]$ provided $\neg empty(S)$, $memb(S) = \{s_1, \ldots, s_n\}$ and $s_i = choo(rest^{i-1}(S))(i = 1, \ldots, n)$. A function $head(S)$ returns the structure such that $vec(head(S)) = [s_1, \ldots, s_{n-1}]$ provided $vec(S) = [s_1, \ldots, s_n]$.

Let us introduce a concatenation operation $con(S_1, S_2)$ for the structures S_1 and S_2 by means of a recursive definition. The explicit definition of the operation can be complicated for some structures, for example, trees. The recursive definition is as follows: $con(S_1, S_2) = S_2$ if $empty(S_1)$, $choo(con(S_1, S_2)) = choo(S_1)$ and $rest(con(S_1, S_2)) = con(rest(S_1), S_2)$ if $\neg empty(S_1)$. We consider $con(S, s)$, $con(s, S)$, $con(S_1, S_2, S_3)$ to be shorthands for $con(S, str(s))$, $con(str(s), S)$, $con(con(S_1, S_2), S_3)$, respectively.

The following theorems express some important properties of the concatenation operation for the structures.

Th1. $\neg empty(S) \rightarrow con(choo(S), rest(S)) = S$.

Th1 is immediate from the definition of *con*.

Th2. $con(vec(S_1), vec(S_2)) = vec(con(S_1, S_2))$.

Th3. $\neg empty(S) \rightarrow con(head(S), last(S)) = S$.

Th2 and Th3 are proved with the help of the induction by $\mid memb(S_1) \mid$ and $\mid head(S) \mid$, respectively.

Let us consider data structures $S_1, \ldots, S_m$. We will use $T(S_1, \ldots, S_m)$ to denote a term constructed from S_i with the help of the functions $choo, last, rest$, $head$, str, con. Sometimes we will omit all S_i in $T(S_1, \ldots, S_m)$. For a term T which represents a data structure, we will denote the function $\mid memb(T) \mid$ by $lng(T)$. The function can be calculated by the following rules: $lng(S_i) = \mid memb(S_i) \mid, lng(con(T_1, T_2)) = lng(T_1) + lng(T_2)$ $lng(rest(T)) = lng(head(T)) = lng(T) - 1, lng(str(s)) = 1$.

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

- terms which do not contain $S_1, \ldots, S_m$;
- the functions $choo(S_i), rest(S_i), last(S_i)$ and $head(S_i)$ for all $i = 1, \ldots, m$;
- the predicate $empty(S_i)(i = 1, \ldots, m)$;

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

Note that the undefined value ω of the functions $choo(S)$ and $rest(S)$ means $empty(S)$. The restriction that we impose on terms T_j in the recursive definition of $STR(S_1, \ldots, S_m)$ ensures termination of the definition process.

3 Definite Iteration over Structures and Replacement Operation

We recall the notion of the definite iteration over structures from [16]. Let us consider the statement
(1) **for** x **in** S **do** $v := body(v, x)$ **end**
where S is the structure, x is the variable called the loop parameter, v is the data vector of the loop body $(x \notin v), v := body(v, x)$ represents the loop body computation. We suppose that the loop body uses variables from v(and x), does not change the loop parameter x and iterates over all elements of the structure S. So, the loop body terminates for every $x \in memb(S)$.

Operational semantics of iteration (1) is defined as follows. Let v_0 be the vector of initial values of variables from the vector v. The result of the iteration is $v = v_0$ if $empty(S)$. If $\neg empty(S)$ and $vec(S) = [s_1, \ \ldots, s_n]$, the loop body iterates sequentially for x defined as $s_1, \ s_2, \ \ldots, \ s_n$.

We associate a function $body(v, x)$ with the right part of the body of iteration (1) such that the body has the same form $v := body(v, x)$. To present the effect of iteration (1), let us define a replacement operation $rep(v, S, body)$ to be a vector v_n such that $v_0 = v$, $n = 0$ provided $empty(S)$, $v_i = body(v_{i-1}, s_i)$ for all $i = 1, \ \ldots, \ n$ provided $\neg empty(S)$ and $vec(S) = [s_1, \ \ldots, s_n]$. It should be noted that in the expression $rep(v, S, body)$ all variables of the term *body* are considered to be bound in the term. Therefore, substitutions for the occurrences of the variables are not performed in the term *body*.

Important properties of the replacement operation are expressed by the following theorems.

Th4. $rep(v, con(S_1, S_2), body) = rep(rep(v, S_1, body), S_2, body)$.

Th4 is proved with the help of the induction by $\mid memb(S_1) \mid$.

Th5. $\neg empty(S) \rightarrow rep(v, S, body) = body(rep(v, head(S), body), last(S))$.

Th5 follows from Th3 and Th4. Let us denote the iteration (1) by $iter(v, S)$.

Th6. $iter(v, S)$ is equivalent to the multiple assignment $v := rep(v, S, body)$.

Proof. We use the induction by $\mid memb(S) \mid$. If $\neg empty(S)$, then $iter(v, S)$ is equivalent to the program **begin** $iter(v, head(S))$; $v := body(v, last(S))$ **end**. It remains to use Th5.

To describe a proof rule for the iteration (1), we introduce the following notation. Let P, Q, inv and $prog$ denote a pre-condition, a post-condition, an invariant, and a program fragment, respectively. $\{P\}$ $prog$ $\{Q\}$ denotes partial correctness of the program $prog$ with respect to the pre-condition P and the

post-condition Q. Let $R(y \leftarrow exp)$ (or $R(exp1 \leftarrow exp)$) be a result of substitution of an expression exp for all the occurrences of a variable y (or an expression $exp1$) into a formula R. Let $R(vec \leftarrow vexp)$ denote a result of the synchronous substitution of the components of an expression vector $vexp$ for all the occurrences of corresponding components of a vector vec into a formula R.

The replacement operation allows us to formulate a proof rule without invariants for the iteration (1). Indeed, let us consider the following proof rule.
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 the post-condition Q does not depend on the loop parameter x.

Let PROOF denote the standard system of proof rules for usual statements including multiple assignment. The following corollary from Th6 justifies the proof rule rl1.
Corollary 1. The proof rule rl1 is derived in the standard system PROOF.

We will generalize the definite iteration (1) allowing for the output from the loop body under a condition. The condition can depend on the loop parameter x but it does not depend on the variables from the data vector v. For this purpose we will define the iteration (1) over hierarchical data structures which is equivalent to the generalized one.

Let us consider the statement
(2) **for** x **in** S_0 **do if** $cond(x)$ **then** $EXIT$; $v := body(v, x)$ **end**
where S_0 is arbitrary data structure, the condition $cond(x)$ does not depend on the variables from v, and $EXIT$ is the statement of termination of the loop.

Let a hierarchical data structure $S = STR(S_0)$ be defined with respect to the structure S_0 and to the condition $cond$ as follows:
$(choo(S), rest(S)) =$ **if** $empty(S_0) \vee cond(choo(S_0))$ **then** (ω, ω)
else $(choo(S_0), STR(rest(S_0)))$.

Th7. The generalized iteration (2) is equivalent to the iteration (1) with $S = STR(S_0)$.

Proof. We will use the induction by $\mid memb(S_0) \mid = n$. Let us suppose $n > 0$ and the iterations to be equivalent if $\mid memb(S_0) \mid < n$. So, $\neg empty(S_0)$. In the case when $\neg cond(choo(S_0))$, the iteration (2) is equivalent to the program
$v := body(v, choo(S_0))$; **for** x **in** $rest(S_0)$ **do**
if $cond(x)$ **then** $EXIT$; $v := body(v, x)$ **end**.
From $\neg cond(choo(S_0))$, it follows that $\neg empty(S)$ and the iteration (1) is equivalent to the program
$v := body(v, choo(S_0))$; **for** x **in** $rest(S)$ **do** $v := body(v, x)$ **end**.
It remains to notice that the programs are equivalent because
$choo(S) = choo(S_0), rest(S) = STR(rest(S_0)), \mid memb(rest(S_0)) \mid = n - 1$, and the inductive hypothesis is applicable.

4 Induction Principles

Verification conditions including the replacement operation are generated by means of the proof rule rl1. To prove the verification conditions, we need a special technique.

Let $prop(STR(S_1, \ldots, S_m))$ denote a property expressed by a first-order logic formula only with free variables $S_1, \ldots, S_m$. The formula is constructed from function symbols, variables and constants by means of Boolean operations and first-order quantifiers. The function symbols include *memb*, *empty*, *vec*, *choo*, *rest*, *last*, *head*, *str*, *con*. To prove such properties, we present an induction principle.

Induction principle 1. The property $prop(STR(S_1, \ldots, S_m))$ holds for all structures $S_1, \ldots, S_m$ if the following two conditions hold:
1) $empty(S_1) \wedge \ldots \wedge empty(S_m) \rightarrow prop(STR(S_1, \ldots, S_m))$,
2) for all $S_1, \ldots, S_m$ such that $\neg empty(S_i)$ for appropriate $i = 1, \ldots, m$, there exist terms $T_1, \ldots, T_m$ for which $\sum_{i=1}^{m} lng(T_i) < \sum_{i=1}^{m} lng(S_i)$ and $prop(STR(T_1, \ldots, T_m)) \rightarrow prop(STR(S_1, \ldots, S_m))$.

The validity of this principle can be proved with the help of induction by $n = \sum_{i=1}^{m} lng(S_i)$.

Let $prop(rep(v, S, body))$ denote a property expressed by a first-order logic formula with the only free variable S. The formula is constructed from the replacement operation $rep(v, S, body)$, function symbols, variables and constants by means of Boolean operations, first-order quantifiers and substitution of constants for variables from v.

Induction principle 2. The property $prop(rep(v, S, body))$ holds for each structure S if the following two conditions hold:
1) $empty(S) \rightarrow prop(rep(v, S, body))$,
2) for each S such that $\neg empty(S)$ there exists a term $T(S)$ such that $lng(T(S)) < lng(S)$ and $prop(rep(v, T(S), body)) \rightarrow prop(rep(v, S, body))$.

Corollary 2 follows from induction principle 2 with $T(S) = rest(S)$ and theorems Th1, Th4.

Corollary 2. The property $prop(rep(v, S, body))$ holds for each structure S if the following two conditions hold for each structure S:
1) $empty(S) \rightarrow prop(rep(v, S, body) \leftarrow v)$.
2) $\neg empty(S) \wedge prop(rep(v, rest(S), body)) \rightarrow prop(rep(v, S, body) \leftarrow rep(body(v, choo(S)), rest(S), body))$.

Corollary 3 follows from induction principle 2 with $T(S) = head(S)$ and Th5.

Corollary 3. The property $prop(rep(v, S, body))$ holds for each structure S if the following two conditions hold for each structure S:
1) $empty(S) \rightarrow prop(rep(v, S, body) \leftarrow v)$.
2) $\neg empty(S) \wedge prop(rep(v, head(S), body)) \rightarrow prop(rep(v, S, body) \leftarrow body(rep(v, head(S), body), last(S)))$.

5 Case of Study: Iterations over Arrays

At first, we recall the known notion $upd(A, ind, exp)$ which denotes an array resulted from the array A by replacing its element indexed by ind with the value of the expression exp [15]. A notion $upd(A, IND, EXP)$ with $IND = [ind_1, \ldots, ind_m]$ and $EXP = [exp_1, \ldots, exp_m]$ is its generalisation which denotes an array obtained from the array A by the evaluation of the expression vector EXP and after that, by the sequential replacement of its ind_j-th element with the value of the expression exp_j for all $j = 1, \ldots, m$. Let Mz denote the projection of a vector M on a variable z.

In the section we assume that the iteration body contains a vector of variables consisted of a variable x, an array A and a vector v of other variables. So, the body of the iteration (1) has the form $(A, v) := body(A, v, x)$. We also assume that $body_A(A, v, x)$ can be represented by $upd(A, IND, EXP)$ for appropriate vectors $IND(x)$ and $EXP(A, v, x)$ where if $A[ind]$ is in $exp_j(A, v, x)$ $(1 \le j \le m)$, then ind has the form $r_i(x)(1 \le i \le t)$. So, we impose a restriction on IND and EXP such that ind_j and r_i do not depend on variables from v. Notice that the representation of $body_A$ by upd is natural, since such a loop body usually contains the statements of the form $A[ind] := exp$ which can be jointly represented by the statement $A := upd(A, IND, EXP)$.

We will define a parallel replacement operation $\widetilde{rep}(A, v, S, body)$ with respect to the array A as a special case of the replacement operation for which the reasoning technique can be simplifyed. Let us define the operation $\widetilde{rep}(A, v, S, body)$ to be a pair (A_n, v_n) such that $A_0 = A, v_0 = v,\ n = 0$ provided $empty(S), A_j = upd(A_{j-1}, IND(s_j), EXP(A, v_{j-1}, s_j)), v_j = body_v(A_{j-1}, v_{j-1},\ s_j)$ for all $j = 1,\ \ldots, n$ provided $\neg empty(S)$ and $vec(S) = [s_1,\ \ldots,\ s_n]$. Thus, the definition differs from the replacement operation definition from Section 3 by the fact that EXP included in upd depends on the initial value of the array A.

The parallel replacement operation is correct if it coincides with the replacement operation. A sufficient condition of its correctness gives the following theorem where $IND(D) = \{ind(s) \mid s \in D, ind \in IND\}$.

Th8. $\widetilde{rep}(A, v, S, body) = rep(A, v, S, body)$, if for every $j = 2, \ldots, n$ and $i = 1, \ldots, t$, $r_i(s_j) \notin IND(pred(vec(S), j))$.

Note that the condition of theorem 8 holds for $j = 1$ because $IND(pred(vec(S), 1))$ is the empty set.

We introduce the following notation. A set $IND(S) = \{ind_j(s) \mid s \in memb(S), 1 \le j \le m\}$ is called a replacement domain. The set $IND(S)$ is empty, if $empty(S)$. Let us define a maximal occurrence function $moc(S, k)$. The function $moc(S, k)$ will be undefined for $k \notin IND(S)$. If $k \in IND(S)$, then $moc(S, k) = (i, j)$ where i is a maximal index of the elements of the vector $vec(S) = [s_1, \ldots, s_n]$ such that there exists a number l for which $ind_l(s_i) = k, j$ is the maximal number among such the numbers l.

The following theorem gives a procedure for computing the parallel replacement operation with respect to an array A.

Th9. $\widetilde{rep_A}(A, v, S, body)[k] = A[k]$ if $k \notin IND(S)$.
$\widetilde{rep_A}(A, v, S, body)[k] = exp_j(A, \widetilde{rep_v}(A, v, head^{n-i+1}(S), body), s_i)$
if $k \in IND(S), vec(S) = [s_1, \ldots, s_n]$ and $moc(S, k) = (i, j)$.

Note that $vec(head^{n-i}(S)) = [s_1, \ldots, s_i]$, therefore $\neg empty(head^{n-i}(S))$ and $head^{n-i+1}(S)$ will be defined. Proofs of theorems Th8 and Th9 are given in Appendix.

Let us consider a special case where iteration (1) has the form
(3) **for** x **in** $[e1, e1+1, \ldots, e2]$ **do** $A := upd(A, IND(x), EXP(A, x))$ **end**
where $e2 \geq e1$. Then $S = vec(S), n = e2 - e1 + 1, pred(vec(S), j) =$
$\{e1, \ldots, e1+j-2\}$ $(j = 2, \ldots, n)$. The following corollaries follow from Th8 and Th9 for the iteration (3).
Corollary 4. $\widetilde{rep}(A, S, upd) = rep(A, S, upd)$ if for every $j = 2, \ldots, n$ and $i = 1, \ldots, t$, $r_i(e1 + j - 1) \notin IND(\{e1, \ldots, e1 + j - 2\})$.
Corollary 5. $\widetilde{rep}(A, S, upd)[k] = A[k]$ if $k \notin IND(S)$.
$\widetilde{rep}(A, S, upd)[k] = exp_j(A, e1 + i - 1)$ if $k \in IND(S)$ and $moc(S, k) = (i, j)$.

6 Examples

Example 1. Copying of ordered file with insertion.
To specify a copying program, we introduce the following notation. Let F and G be the files considered as structures; Ω denotes the empty file; $ord(F)$ is a predicate whose value is true if F was sorted in ascending order $\leq$ of elements and false otherwise. We assume that $ord(\Omega)$ and $\omega < y$ for each defined element y and the undefined element ω. $del(F, y)$ is a function which returns a file resulted from the file F by eliminating the first occurrence of the element y. If the element y is not contained in the file F, then $del(F, y) = F$. $hd(F, y)$ is a function which returns a file resulted from the file F by eliminating its tail which begins with the first occurence of the element y. $tl(F, y)$ is a function which returns a file resulted from the file F by eliminating its head which ends with the first occurence of the element y. If the element y is not contained in the file F, then $hd(F, y) = tl(F, y) = F$. $y > F$ is a predicate whose value is true, if $empty(F)$ or $\forall x \in memb(F)$ $y > x$ and false otherwise.

The following annotated program copies the sorted file F to the file G inserting an element w in its proper place.
$\{P\}$ $ins := false; G := \Omega$; **for** x **in** F **do** $(G, ins) := body(G, ins, x)$ **end**;
if $\neg ins$ **then** $G := con(G, w)$ $\{Q\}$
where ins is a Boolean variable, $body(G, ins, x) =$
if $w \leq x \wedge \neg ins$ **then** $(con(G, w, x), true)$ **else** $(con(G, x), ins)$, $P(F) = ord(F)$,
$Q(F, G) = (del(G, w) = F \wedge ord(G) \wedge w \in memb(G))$.

Two following verification conditions are generated by means of the proof rule rl1 and the standard system PROOF. We consider $rep(F)$ to be a shorthand for $rep((\Omega, false), F, body)$.
VC1: $P(F) \wedge \neg rep_{ins}(F) \rightarrow Q(F, con(rep_G(F), w))$,
VC2: $P(F) \wedge rep_{ins}(F) \rightarrow Q(F, rep_G(F))$.

To prove the verification conditions, we apply the property $prop(rep(F)) = prop1 \wedge prop2$ where $prop1 = (\neg rep_{ins}(F) \rightarrow rep_G(F) = F \wedge w > F), prop2 = (rep_{ins}(F) \rightarrow del(rep_G(F), w) = F \wedge w > hd(rep_G(F), w) \wedge w \in memb(rep_G(F)) \wedge w \leq choo(tl(rep_G(F), w)))$. The property $prop1$ specifies the case when the variable ins remains false, w exceeds all elements of the file F, and F is copied to the file G. The property $prop2$ specifies another case when the variable ins becomes true and the file F is copied to the file G with insertion of the element w in its proper place. We use Corollary 3 in order to prove the property $prop(rep(F))$.

Note that in [16] a mistake has been found in a version of the program with the help of the functional method. Formal verification of the correct program is not described in [16].

Example 2. Merging of ordered arrays with cleaning.
Let us consider two ordered arrays A_1 and A_2. The following structure $S = STR(A_1, A_2)$ is constructed by merging of the arrays in an ordered array.

$(choo(S), rest(S)) =$ **if** $\neg empty(A_1) \wedge \neg empty(A_2)$
then if $choo(A_1) \leq choo(A_2)$ **then** $(choo(A_1), STR(rest(A_1), A_2))$
else $(choo(A_2), STR(A_1, rest(A_2)))$
else if $\neg empty(A_1)$ **then** $(choo(A_1), rest(A_1))$
else if $\neg empty(A_2)$ **then** $(choo(A_2), rest(A_2))$
else (ω, ω).

The following annotated program merges the ordered arrays A_1 and A_2 in the ordered array A removing repetitive elements.
$\{P\} j := 1; A := \emptyset;$ **for** x **in** $STR(A_1, A_2)$ **do** $(A, j) := body(A, j, x)$ **end** $\{Q\}$
where $body(A, j, x) =$ **if** $j > 1 \wedge x = A[j-1]$ **then** (A, j)
else $(upd(A, j, x), j + 1), \emptyset$ denotes the empty array.

The annotations have the form:
$P(A_1, A_2) = \neg empty(A_1) \wedge \neg empty(A_2) \wedge ord(A_1) \wedge ord(A_2)$,
$Q(A, A_1, A_2) = (set(A_1) + set(A_2) = set(A) \wedge ord(A) \wedge difel(A))$
where $ord(A)$ is a predicate whose value is true, if A was sorted in ascending order of elements and false otherwise (we assume that $ord(A)$ for $\mid memb(A) \mid \leq 1$); $set(A)$ is a function which returns a set of all elements of A; $+$ is the union operation for sets; $difel(A)$ is a predicate whose value is true, if A does not contain equal adjacent elements and false otherwise.

The following verification condition is generated by means of the proof rule rl1 and the standard system PROOF. We consider $rep(S)$ to be a shorthand for $rep((\emptyset, 1), STR(A_1, A_2), body)$.
$VC : P(A_1, A_2) \rightarrow Q(rep_A(S), A_1, A_2)$.

To prove VC, we use the property $prop(S) = (memb(S) = memb(A_1) + memb(A_2) \wedge (ord(A_1) \wedge ord(A_2) \rightarrow ord(vec(S))))$ of the structure $S = STR(A_1, A_2)$. To prove the property, we use induction principle 1.

The verification condition VC follows from the properties $prop(S)$ and $prop(rep(S)) = (set(rep_A(S)) = set(vec(S)) \wedge difel(rep_A(S)) \wedge \mid rep_A(S) \mid = rep_j(S) - 1 \wedge (ord(vec(S)) \rightarrow ord(rep_A(S))))$. We use Corollary 3 in order to prove the property $prop(rep(S))$.

Example 3. The array inversion.
The following annotated program presented by the iteration (3) inverts an array $A[1..m]$.
$\{P\}$ **for** k **in** S **do** $(A[k], A[m+1-k]) := (A[m+1-k], A[k])$ **end** $\{Q\}$
where $S = [1, 2, \ldots, trunc(m \setminus 2)]$, $trunc(s)$ is an integer nearest to s, A_0 is an initial value of the array A, $P(A) = m \geq 1 \wedge A[1..m] = A_0[1..m], Q(A) =$
$\forall i\ (1 \leq i \leq m \rightarrow A[i] = A_0[m+1-i])$. So, $IND = (ind_1, ind_2)$, $ind_1(k) = k$, $ind_2(k) = m+1-k$, $EXP = (exp_1, exp_2)$, $exp_1(A,k) = A[m+1-k]$, $exp_2(A,k) = A[k]$. Therefore, $exp_1(A,\ k) = A[r_1(k)]$, $r_1(k) = n+1-k$, $exp_2(A,\ k) = A[r_2(k)], r_2(k) = k$.

It follows from $2j \leq m$ that $j < m-j+2, r_1(j)$ and $r_2(j)$ are not contained in $IND(\{1, \ldots, j-1\})(j = 2, \ldots, trunc(m \setminus 2))$. Therefore, by Corollary 4, $\widetilde{rep}(A, S, body) = rep(A, S, body)$.

The verification condition $P(A) \rightarrow Q(\widetilde{rep}(A, S, body))$ is generated by the proof rule rl1 when the program *prog* is empty. The correctness proof of the verification condition with the help of Corollary 5 can be realized without induction.

7 Conclusion

The paper presents a symbolic method for verification of definite iteration over hierarchical data structures. The symbolic method differing substantially from the axiomatic and functional methods has some features related with these methods. For definite iteration the symbolic method uses a proof rule which has the form inherent in the axiomatic method, however, without invariants. To justify the proof rule, the axiomatic method is applied. The symbolic method, such as the functional one, makes use of a functional representation for the iteration body and for the iteration as the replacement operation.

Let us discuss peculiarities and advantages of the symbolic method. Axiomatization of data structures by means of the concatenation operation *con* plays an important role in the method. Indeed, useful algebraic properties of the operation *con* are expressed by Theorems 1, 2, 3. Moreover, the operation *con* is used to represent a key property of the replacement operation in Theorem 4 and to prove Theorem 5. The symbolic method is based on the replacement operation which allows loop invariants in proof rules to be eliminated. Instead of such an invariant, a suitable property of the replacement operation is used in a verification process. As a result, the verification process is substantially simplified because the property, as a rule, is simpler than the invariant. Besides, induction principle 2 is rather flexible and allows us to use different induction strategies, such as forward, backward and mixed, in proving the property. The use of properties of hierarchical data structures simplifies proving the verification conditions with the help of induction principle 1. Theorem 7 allows us to represent an important case of while-loops. We have proved a more complicated theorem which generalizes Theorem 7. This theorem is used for verification of programs over pointers.

Advantages of the symbolic method become prominent for programs over complex data structures. For arrays the change of the replacement operation by the parallel one allows us to eliminate or to simplify inductive reasoning. Notice that the parallel replacement operation has been introduced for a special case of arrays in [11] and has been generalized for them in [13]. A variant of the parallel replacement operation has been used for modelling synchronous computations in [3] which are represented by statements equivalent to for-loops over sets with vector assignments as the loop bodies. The statements are expressed by universal quantifiers bounded by the sets which are given by Boolean expressions.

The symbolic method of verification is promising for applications. To extend the range of its applicability in program verification systems, it is helpful to develop a proof technique which uses the peculiarities of problem domains. The induction principles developed in the framework of the symbolic method can be also useful for the functional method.

References

1. Basu, S.K., Misra, J.: Proving loop programs. IEEE Trans. on Software Engineering, Vol.1, No.1 (1975) 76–86 176
2. Basu, S.K., Misra, J.: Some classes of naturally provable programs. Proc. 2nd Int. Conf. on Software Engineering, IEEE Press (1976) 400–406 176
3. Chandy, K.M., Misra, J.: Parallel Program Design. Addison-Wesley (1988) 186
4. Dunlop, D.D., Basili, V.R.: Generalizing specifications for uniformly implemented loops. ACM Trans. on Programming Languages and Systems, Vol. 7, No. 1 (1985) 137–158 176
5. Fradet, P., Gaugne, R., Metayer, D.Le.: Static detection of pointer errors: an axiomatisation and a checking algorithm. Lecture Notes in Computer Sci., Vol. 1058 (1996) 125–140 176
6. Gries, D., Gehani, N.: Some ideas on data types in high-level languages. Communications of the ACM, Vol. 20, No. 6 (1977) 414–420 176
7. Hoare, C.A.R.: An axiomatic basis of computer programming. Communications of the ACM, Vol. 12, No. 10 (1969) 576–580 176
8. Hoare, C.A.R.: A note on the for statement. BIT, Vol. 12, No. 3 (1972) 334–341 176
9. Linger, R.C., Mills, H.D., Witt, B.I.: Structured Programming: Theory And Practice. Addison-Wesley (1979) 176
10. Mills, H.D.: Structured programming: retrospect and prospect. IEEE Software, Vol. 3, No. 6 (1986) 58–67 176
11. Nepomniaschy, V.A.: Proving correctness of linear algebra programs. Programming, No. 4 (1982) 63–72 (in Russian) 186
12. Nepomniashy, V.A.: Loop invariant elimination in program verification. Programming, No. 3 (1985) 3–13 (in Russian) 177
13. Nepomniaschy, V.A.: On problem-oriented program verification. Programming, No. 1 (1986) 3–13 (in Russian) 177, 186
14. Nepomniaschy, V.A.: Verification of definite iteration over data structures without invariants. Proc. 12th Intern. Symp. on Computer and Information Sci., Antalya, Turkey (1997) 608-614 177

15. Reynolds, J.C.: Reasoning about arrays. Comm. of the ACM, Vol. 22, No. 5 (1979) 290–299 182
16. Stavely, A.M.: Verifying definite iteration over data structures. IEEE Trans. on Software Engineering, Vol. 21, No. 6 (1995) 506–514 176, 177, 179, 184

8 Appendix: Proofs of Theorems Th8 and Th9.

Th8. It is sufficient to prove
(4) $EXP(A_{j-1}, v_{j-1}, s_j) = EXP(A, v_{j-1}, s_j)$ for all $j = 1, \ldots, n$.
Assertion (4) follows from
(5) $A_{j-1}[r_i(s_j)] = A[r_i(s_j)]$ for all $j = 1, \ldots, n$ and $i = 1, \ldots, t$.
Let us consider a generalization of (5) of the form
(6) $A_k[r_i(s_j)] = A[r_i(s_j)]$ for all $j = 1, \ldots, n, i = 1, \ldots, t, k = 0, 1, \ldots, j-1$.
To prove (6), we will use the induction by k. If $k = 0$, then (6) is true because $A_0 = A$. If $0 < k < j$, then
$A_k[r_i(s_j)] = upd(A_{k-1}, IND(s_k), EXP(A, v_{k-1}, s_k))[r_i(s_j)] = A_{k-1}[r_i(s_j)]$ because $s_k \in pred(vec(S), j), IND(s_k) \subseteq IND(pred(vec(S), j)), r_i(s_j) \notin IND(s_k)$.
It remains to apply the inductive hypothesis $A_{k-1}[r_i(s_j)] = A[r_i(s_j)]$.
Th9. We will use the induction by $\mid memb(S) \mid = n$. If $n = 0$, then $empty(S)$ and $IND(S)$ is the empty set. From this and Ax12, it follows that $k \notin IND(S)$ and Th9. Let us consider the case $n \neq 0$. So, $\neg empty(S)$. By theorem Th3, $S = con(head(S), last(S))$ and $IND(S) = IND(head(S)) \cup IND(last(S))$. From this it follows that
(7) $\widetilde{rep_A}(A, v, S, body)[k] = upd(\widetilde{rep_A}(A, v, head(S), body), IND(last(S)),$
$EXP(A, \widetilde{rep_v}(A, v, head(S), body), last(S)))[k]$.
Theorem Th9 is proved by the case analysis. Three cases are possible.

1. $k \notin IND(S)$. Then $k \notin IND(last(S))$. From this and (7) it follows that
(8) $\widetilde{rep_A}(A, v, S, body)[k] = \widetilde{rep_A}(A, v, head(S), body)[k]$.
It remains to apply the inductive hypothesis because $\mid memb(head(S)) \mid = n-1$ and $k \notin IND(head(S))$.

2. $k \in IND(last(S))$. Then there exists j such that
$1 \leq j \leq m, ind_j(s_n) = k$, and $\forall l(m \geq l > j \rightarrow ind_l(s_n) \neq k)$. From this and (7) it follows the conclusion of Th9 for $i = n$ of the form $\widetilde{rep_A}(A, v, S, body)[k] = exp_j(A, \widetilde{rep_v}(A, v, head(S), body), s_n)$.

3. $k \in IND(head(S)) \wedge k \notin IND(last(S))$. Then (8) follows from (7). It should be noted that $moc(S, k) = moc(head(S), k)$, because $k \notin IND(last(S))$. Let us assume $moc(head(S), k) = (i, j)$, where $1 \leq i < n, 1 \leq j \leq m$. It remains to apply the inductive hypothesis of the form $\widetilde{rep_A}(A, v, head(S), body)[k] = exp_j(A, \widetilde{rep_v}(A, v, head^{n-i}(head(S)), body), s_i)$ because
$\mid memb(head(S)) \mid = n-1$. Theorem Th9 follows from this and (8).