Mechanical Generation of Invariants for FOR-Loops

Stefan Kauer[1] and Jürgen F H Winkler[2]
[1] EADS Deutschland GmbH, Business Unit Defence Electronics, Claude-Dornier-Str.
D-88090 Immenstaad, Germany
Stefan.Kauer@eads.com
[2] Friedrich Schiller University, Institute of Informatics, Ernst-Abbe-Platz 2,
D-07743 Jena, Germany
winkler@informatik.uni-jena.de

Abstract. In the mechanical verification of programs containing loops it is often necessary to provide loop invariants additionally to the specification in form of pre- and postcondition. In this paper we present a method for the mechanical generation of invariants for a class of FOR-loops. The invariant is derived from the postcondition and the final bound of the loop only. The method is applicable if the final bound of the FOR-loop is of a simple form. This is often the case in practice. The incorporation of this method into an automatic program verifier would make the task of the SW engineer easier, because he has only to provide a pre-post-specification for a FOR-loop.

Keywords: mechanical verification, mechanical generation of loop invariants, FOR-loop.

1 Introduction

Program verification involves a great amount of mechanical formula manipulation. If done by hand, this is tedious and, even worse, error prone. Most of the theorems (verification conditions (VC)), which have to be proved, are quite trivial and can therefore be proved automatically by an automatic theorem prover. If all VCs are generated and proved automatically we speak of automatic program verification. A tool which performs automatic program verification is then called an automatic program verifier (APV). Examples of such tools are Boogie [BCD06], FPP [KW99a; Win97] and NPPV [Gumxx]. Other tools use a combination of automatic and interactive theorem proving and can therefore be called semi-automatic program verifiers (SAPV). Examples are KeY [ABB05], SPARK [Bar00] and Theorema [JKP03].

Most tools for the verification of concrete programs use the assertion based method (ABM) for the specification of the required behavior of the program (e.g. Boogie, FPP, KeY, NPPV, SPARK, Theorema). The specification is given by a pair (pre, post) of assertions which refer to entities of the program, and may also refer to entities which belong to the specification only.

ABM allows also the verification of program fragments and therefore can be used by the SW engineer in a continuous manner during program development, and not only for the verification of a finished program in one big step. In this situation, the use of an APV is especially convenient.

In ABM automatic verification on the basis of (pre, post) is rather straightforward for statements like declarations, assignment, IF, and CASE[1]. The verification of loops usually requires also an invariant [Dij76; Hoa72; Tur49; Win98], and for WHILE-loops additionally a termination function [Dij76; Flo67; Tur49]. It would be easier if loops could also be verified by giving only (pre, post). This can be done in two ways: (1) by computing wp(loop, post) resp. sp(pre, loop) and use this in the general verification condition pre $\Rightarrow$ wp(loop, post) resp. sp(pre, loop) $\Rightarrow$ post, or (2) by computing an invariant (and in case of a WHILE-loop a termination function) and perform then the verification using the corresponding VCs. Automatic computation of invariants from the code is seen as difficult in the general case [Bac06: 3]. Stefan Kauer has developed methods for the mechanical verification of classes of loops which are only specified by (pre, post) [Kau99]. For FOR-loops his method is based on the heuristic "replacing a constant in the postcondition by a variable (RCPV)" for the computation of an invariant. For WHILE-loops his method computes wp(loop, post). In this paper we report about the method for the computation of invariants for FOR-loops. For the verification of FOR-loops we use the proof rule of [Win98] which is less restrictive than that of [Hoa72].

An annotated FOR-loop (AFL) in Ada syntax looks like

```
   -- PRE
FOR i in LO..UP LOOP  BODY  END LOOP                    (1)
   -- POST
```

where i is the loop variable, the value of LO is the lower bound and the value of UP is the upper bound of the loop. (1) is an upwards counting loop. Many languages contain also downwards counting loops. In this paper we deal mainly with upwards counting FOR-loops. Downwards counting FOR-loops do not pose new problems and can be treated in an analogous manner [KW00; Win98].

Basic idea. RCPV tries to derive an invariant INV from the postcondition POST and the final bound of the FOR-loop only; the final bound of a FOR-loop is the upper bound for upwards counting loops and is the lower bound for downwards counting loops. Especially, BODY is not used for the derivation of INV, but it is used to check whether INV is really an invariant of the loop. By not using BODY we avoid the problem that a loop with an incorrect BODY may lead to the generation of invalid invariants resulting in redundant work for the APV. Another aspect of only using POST and the final bound is that nested loops have not to be treated as a special case as e.g. in [Weg74], but can be treated in a recursive manner. The method for the generation of hypothetical invariants is formulated as an algorithm which can be used in an APV.

Related work. Soon after the seminal work on program verification by Floyd and Hoare [Flo67; Hoa69] began a phase of intensive work on developing methods for the determination of loop invariants [e.g. Weg74; Cap75; KM76; MW77; Mis78; Bas80; Tam80; Ell81; Gri82; BD84; GDM85; Pai86; CEG99; Kau99; CEG00; BMM01; FQ02]. More recently several methods have been presented to determine especially

[1] This refers primarily to the generation of the VCs. The verification proper may still be rather difficult even for very simple statements: {True} skip; {Goldbach's conjecture}.

polynomial invariants [MS03; MSS04; JK05; PS05; KR06]. Some methods are for application by hand [e.g. Cap75; Mis78], some work in a semi-mechanized manner [e.g. Weg74; Tam80; BMM01; FQ02] and some are fully mechanized [e.g. Kau99; PS05; JK05; KR06].

The different approaches exploit the annotated loop in different ways:

Some methods use the loop only, i.e. derive invariants from the code [e.g. KM76; Bas80; Tam80; Ell81; GDM85; Pai86; MS03; MSS04; JK05; PS05; KR06]. In [CEG99, CEG00] the loop is instrumented in order to output interesting variables ("trace variables"). The method then tries to infer an invariant from the values of the trace variables for several executions of the loop. This is a special case of deriving an invariant from the loop, because the values of the trace variables are determined by the loop.

Another approach is to derive the invariant from the specification [Mis78; Gri82]. Misra [Mis78] mentions two approaches: "A loop invariant could be a proposition about "what has been done" or a proposition about "what remains to be done" ". Gries [Gri81, Gri82] derives an invariant of the kind "what has been done" from POST. Gries attributes this methodology to Dijkstra [Dij76]. Whereas Misra uses the invariant for the verification of an existing loop, Gries uses the invariant for the development of the loop itself.

Wegbreit [Weg74] and Kauer [Kau99] derive INV from POST and the loop-condition. The method of Kauer is inspired by [Gri82], but is mechanized, and is tailored to FOR-loops and to the verification of an existing loop. The details are the topic of this paper.

Most methods work with WHILE-loops. Since FOR-loops can be transformed into WHILE-loops these methods can also be applied to FOR-loops. If e.g. the method of [Weg74] is applied to the WHILE-loop corresponding to example (4) in sect. 3.1 no loop invariant seems to be produced, despite the fact that the candidates are also derived from POST and the loop-condition.

The rest of the paper is organized as follows. In section 2 we present the verification scheme for FOR-loops. Section 3 contains the method for the computation of an invariant and some examples of its application. Section 4 concludes the paper.

2 An Improved Proof Rule for FOR-loops

The proof rule for FOR-loops in [Win98] is based on that in [Hoa 72]. The main differences are that [Win98] does not require $\boldsymbol{I}([\,])$ to hold before the first execution of the loop body. The invariant $\boldsymbol{I}([\mathrm{LO} \,..\, i])$ must only hold after executions of the loop body. Secondly, the loop variable may occur in the invariant, and thirdly, the proof rule also works for loops with zero repetitions. The strategy for the handling of the invariant is based on the following observations:

(1) the invariant is intended to be an assertion which is established by any execution of the loop body, especially the last one; therefore, it seems not necessary that the invariant holds before the FOR-loop. Collins calls such an invariant a "post-invariant" [Col88];

(2) the invariant of a FOR-loop is typically an inductive assertion which involves the loop variable. In [Hoa72] the loop variable must not occur in the invariant;

(3) in some programming languages the loop variable is declared locally in the loop and does not exist outside the loop [e.g. Algol 68, Ada, C#]. If the invariant contains the loop variable and must hold before the loop, this could lead to illegal uses of the loop variable;

(4) there are examples in which it seems difficult to derive $\boldsymbol{I}([\,])$ mechanically from $\boldsymbol{I}([LO\,..\,i])$. One example for this is [Win 98: 9]:

```
v := 5;
     -- v=5
FOR i IN 1 .. 10
LOOP  v := i;                                   (1a)
     --  inv ???
END LOOP;
     -- v=10
```

It is easy to see that $\boldsymbol{I}([1..i]) \equiv v = i$ is an invariant which fulfills (1a).

$\boldsymbol{I}([1..i])^{i}_{10} \equiv \boldsymbol{I}([1..10]) \equiv v=10$ is sufficient to establish the postcondition. We then have to determine $\boldsymbol{I}([\,])$ such that

$$[\, v=5 \Rightarrow \boldsymbol{I}([\,])\,] \wedge [\, \boldsymbol{I}([\,]) \Rightarrow wp(\text{“v:=1;“}, v=1\,] \qquad (1b)$$

holds. If we try $\boldsymbol{I}([\,]) \equiv \boldsymbol{I}([1..i])^{i}_{pred(1)} \equiv v=0$ we observe that it does not work: because $[v=5 \Rightarrow v=0] \equiv$ False. On the other hand, $\boldsymbol{I}([\,]) \equiv$ true does the trick; it is maximal in that it is the weakest solution of (1b). But it is not derived mechanically from $\boldsymbol{I}([1..i])$.

Apart from these differences, the verification scheme is expressed in a form suitable for automatic verification using ABM, whereas the proof rule in [Hoa72] is formulated as a logical derivation rule.

The verification scheme for FOR-loops used in this paper is:

$$\begin{aligned}
&[\, PRE \Rightarrow LO, UP \in Ti\,] \wedge \\
&[\, PRE \wedge LO > UP \Rightarrow POST\,] \wedge \\
&[\, PRE \wedge LO \le UP \Rightarrow wp(\, BODY^{i}_{LO}, INV^{i}_{LO}\,)\,] \wedge \\
&[\, LO \le i < UP \wedge INV \Rightarrow wp(\, BODY^{i}_{i+1}, INV^{i}_{i+1}\,)\,] \wedge \\
&[\, LO \le UP \wedge INV^{i}_{UP} \Rightarrow POST\,]
\end{aligned} \qquad (2)$$

where

PRE	is the precondition
POST	is the postcondition
INV	is the invariant
Ti	is the value set of the type of the loop variable i
[...]	denotes universal quantification over the program variables and the specification variables

This form of the verification scheme assumes that

(r1) the evaluation of LO and UP has no side effects

(r2) any evaluation of LO, UP or any of their subexpressions at any point in the FOR-loop yields the same value as in the initial evaluation at the beginning of the

execution of the FOR-loop. This means especially that LO and UP are not written to in BODY and that they do not contain calls of functions which are not referentially transparent.

Both restrictions hold for many loops used in practice. Restriction (r2) is not severe; [Win98; KW00] contain a scheme which does not require restriction (r2) by introducing two fresh variables vlo and vup which are assigned the values of LO and UP before beginning the repetitions of the loop body. Since the method for the computation of the invariant does not depend on the exact form of the loop verification scheme, we use the simpler form of the verification scheme for the examples in this paper.

In [KW00] we show that (2) implies the correctness of the loop (1) and that the correctness of (1) implies the existence of an invariant INV, which satisfies (2).

3 A Method for Computing Invariants of FOR-Loops

3.1 Basic Idea

The general wp-rule for a FOR-loop cannot always be solved exactly. Usually, some weaker form of correctness is used which uses a loop invariant [Gri81; Win98]. This means that the engineer has to determine a suitable invariant. If such an invariant can be computed mechanically the task of the engineer will be easier. In this section we present a method for the mechanical generation of invariants of FOR-loops, which are annotated by PRE and POST only.

The method is based on the heuristic "*replacing a constant in the postcondition by a variable(RCPV)*" [Gri81: 199], where in our case the variable is always the loop variable. The heuristic RCPV is typically applied by replacing the final bound in POST by the loop variable. For upwards counting loops the final bound is UP, and for downwards counting loops it is LO. In the following we show the derivation of the method for upwards counting loops. How it works for downwards counting loops is presented in [KW00].

The method works in two steps:

(1) try to derive a predicate HI (hypothetical invariant) from the AFL. There are cases in which the method does not generate a predicate HI, e.g. if UP is a non-linear expression. From a practical point of view those cases are rare. A check of [BG91] gave the following result: in most FOR-loops UP has one of the forms: (a) variable, (b) sum of two variables, or (c) sum of a variable and a constant. The Fortran program RWPL (= Randwertproblemlöser = boundary value problem solver), written by M. Hermann and D. Kaiser of our department, contains 1015 FOR-loops (DO-loops in Fortran), of which 998, i.e. almost all, are appropriate for our method.

(2) try to prove $(\text{FOR-rule})^{INV}_{HI}$. There are three possible answers:

a) the proof succeeds, i.e. HI is an invariant and the loop is correct.

b) the refutation succeeds. This can be due to the following reasons:

b1) the loop is correct, but HI is not an invariant.

b2) the loop is not correct. In this case HI may or may not be an invariant.

c) neither proof nor refutation succeed, i.e. the prover "gives up" or does not terminate. In this case it is unknown whether the loop is correct or incorrect, or whether HI is or is not an invariant.

Only in case a) does the method say that the loop is correct.

The idea behind this method is that most FOR-loops compute their final result by computing a sequence of partial intermediate results which approximate the final result better and better. The final result is described by POST and often depends on a characteristic constant or variable which usually is UP. $POST^{UP}_{i}$, which then depends on i, often characterizes these partial results.

A very simple example is a loop for the summation of the first 100 natural numbers:

```
    -- PRE: s = 0  ∧  s ∈ int32
FOR i in 1..100 LOOP s := s+i; END LOOP                    (3)
    -- POST: s = ⟨Σj: 1..100: j⟩  ∧  s ∈ int32
```

In (3) we assume that the type of s is int32. In (3) RCPV can be applied directly and gives the HI

$$HI \equiv POST^{100}_{i} \equiv (s = \langle \Sigma j: 1..100: j \rangle \wedge s \in int32)^{100}_{i}$$
$$\equiv s = \langle \Sigma j: 1..i: j \rangle \wedge s \in int32$$

The loop (3) with the invariant HI satisfies (2) and therefore, HI is an invariant of (3) and (3) is correct. Since $\langle \forall i \in 0..100: \langle \Sigma j: 1..i: j \rangle \in int32 \rangle$ holds, we could have ommitted `s ∈ int32` in (3). We included it for documentation purposes.

(3) is a very special loop because the upper bound is the fixed number 100. Often the upper bound will be a program variable whose value is constant in the FOR-loop. Such a more general FOR-loop is given in (4).

```
    -- PRE: s=0 ∧ 0≤n≤65535 ∧ n=N
FOR i in 1..n LOOP s := s+i; END LOOP                      (4)
    -- POST: s=⟨Σj: 1..n: j⟩ ∧ 0≤n≤65535 ∧ n=N
```

We assume that s and n are of type int32. N is a specification variable which is used to guarantee that the value of n after the loop is the same as before the loop. If we compute HI mechanically as $POST^{n}_{i}$ we obtain

$$HI \equiv POST^{n}_{i} \equiv (s=\langle \Sigma j: 1..n: j \rangle \wedge 0 \le n \le 65535 \wedge n=N)^{n}_{i}$$
$$\equiv s=\langle \Sigma j: 1..i: j \rangle \wedge 0 \le i \le 65535 \wedge i=N$$

We observe immediately that HI is not an invariant of (4) because i has not always the value N (if N > 1).

A strategy for avoiding this problem is to apply the substitution $(n \mapsto i)$ only to those conjuncts of POST which are not also a conjunct of PRE. In the example this results in

$$HI \equiv (s=\langle \Sigma j: 1..n: j \rangle)^{n}_{i} \wedge 0 \le n \le 65535 \wedge n=N$$
$$\equiv s=\langle \Sigma j: 1..i: j \rangle \wedge 0 \le n \le 65535 \wedge n=N$$

The loop (4) with HI as invariant satisfies (2).

A method for the identification of common conjuncts is given in sect. 3.3.

3.2 Bound Transformation

The method developed so far works only if UP is a constant or a variable. This is a severe restriction. One idea is to insert the assignment "vup := UP;" immediately before the loop, where vup is a fresh variable, and then use vup as the upper bound. But this does not work in general, since vup does not occur in POST. Replacing a nonoccurring variable does not change POST, so that POST itself had to be considered as an invariant, which does not work in most cases.

In the loop (5) UP is not a variable but a more complicated expression.

```
   -- PRE: s = 0
FOR i in 1..n+m LOOP s := s+i; END LOOP                    (5)
   -- POST: s = ⟨Σj: 1..n+m: j⟩
```

In (5) we have omitted the type constraints on s, n and m because they are not significant for the current discussion. In general, the value of n+m is constrained by the type of s: in (4) the constraint for n guarantees that $s \in int32$.

The introduction of vup and the application of RCPV results in

```
vup := n+m;
   -- PRE: s = 0 ∧ vup = n+m
FOR i in 1..vup LOOP s := s+i; END LOOP                    (6)
   -- POST: s = ⟨Σj: 1..n+m: j⟩
```

Since vup does not occur in POST we obtain

$HI \equiv POST^{vup}_{i} \equiv s = \langle \Sigma\, j: 1..n+m: j \rangle^{vup}_{i} \equiv s = \langle \Sigma j: 1..n+m: j \rangle \equiv POST.$

It is easy to see that HI is not an invariant.

It seems therefore better to try to transform the given loop L1 into an equivalent loop L2 with upper bound UP2, such that $UP2 = v$ and $v \in free(UP1) \cap free(POST)$ holds.

In the following we distinguish between expressions such as LO and UP and their value, which we denote by s1(LO) and s1(UP) where s1 is the state just before the first execution of BODY.

In this paper we use transformations t which induce a translation of the range s1(LO1) .. s1(UP1) by a constant k = s1(e), where e is some arithmetic expression. The resulting range is then s1(LO1)+s1(e) .. s1(UP1)+s1(e) = s1(LO2) .. s1(UP2).

This means that the number of executions of BODY is the same in the transformed loop L2. If we use a suitable transformation t* in BODY, which compensates for the translation t of the values of the loop variable, we obtain a loop which is semantically equivalent to the given loop L1. This leads to the following scheme (for upward counting loops):

L1:
```
   -- PRE
FOR i in LO1..UP1 LOOP BODY END LOOP
   -- POST
```

L2:
```
   -- PRE
FOR i in t(LO1)..r(t(UP1))
LOOP BODY(i ↦ t*(i)) END LOOP
   -- POST
```

r(•) is a function which reduces (simplifies) arithmetic expressions.

In L2 a translation t(•) is applied to LO and UP and a second translation t*(•) to all occurrences of the loop variable i in BODY. The idea is that t*(•) neutralizes t(•) and that therefore the BODY of L2 is executed for the same values of the loop variable i as the BODY of L1. That this is really the case is shown below.

We apply the loop transformations only in such cases in which r(t(UP1)) has the form "v" or "–v". Since UP2 = r(t(UP1)) has this simple form we get the hypothetical invariant

HI = $POST^{v}_{i}$, if r(t(UP1)) has the form "v ", or

HI = $POST^{v}_{(-i)}$, if r(t(UP1)) has the form "-v" .

L2 is never really executed but is only used to determine HI and a corresponding proof rule. On the level of proof rule and proof we assume the usual mathematical sets of numbers, i.e. when applying t and t* we do not have to watch for range violations and can apply the usual laws of arithmetic.

The transformation t depends on UP and v and is a mapping $E \times Var \rightarrow (E \rightarrow E)$, i.e. $t(UP,v) \in E \rightarrow E$, where E is the set of arithmetic expressions. When UP and v are known from the context we also write t(e) instead of t(UP,v)(e). t(UP,v) is applied to both LO1 and UP1. In order to find t for a given expression UP1 and a given variable v we determine the syntactic transformation necessary to semantically neutralize all terms apart from v or -v.

E.g. if UP = m+10 we obtain t(UP, m)(e) = e -10 and for UP = n*a +b we obtain t(UP, n)(e) = (e-b)/a and t(UP, b)(e) = e - n*a . We obtain the corresponding t* by modifying t(e) analogously wrt e.

Not all such transformations lead to a translation of s1(LO1) .. s1(UP1). For UP1 = 2*n we obtain t(UP1,n)(e) = e/2 . If LO1 = 0 then the original range is 0..2*n and the transformed range is 0..n whose lengths are different for n >0. On the other hand, if we rewrite 2*n as n+n, we could transform the range 0..2*n into -n..n which has the same length.

This means that the transformation of the range is only possible if UP1 has a suitable form. In this paper we limit the possible transformations to the cases in the following table:

Table 1. Definition of the mappings t and t*

	Form of UP			
	o1 v	e1 o2 v	o1 v o2 e1	e1 o2 v o3 e2
t(UP,v)(e)	e	e - e1	e $o2^{-1}$ e1	e - e1 $o3^{-1}$ e2
r(t(UP,v)(UP))	o1 v	o2 v	o1 v	o2 v
t*(UP,v)(e)	e	e + e1	e o2 e1	e + e1 o3 e2
t*(UP,v)(t(UP,v)(e))	e	e - e1 + e1	e $o2^{-1}$ e1 o2 e1	e - e1 $o3^{-1}$ e2 + e1 o3 e2

where $v \notin free(e1) \cup free(e2)$, $o1 \in \{+, -, \varepsilon\}$, $o2, o3 \in \{+, -\}$, $+^{-1} = -$, $-^{-1} = +$, and e1 and e2 are parenthesized expressions. If e.g. UP = v-a+b we assume that UP has been transformed into v-(a-b) or an equivalent parenthesized form. As already mentioned in sect. 3.1 these restrictions seem not severe from a practical point of view.

Since the expressions in table 1 operate over the mathematical sets of numbers the usual algebraic rules apply. It is easy to see that the last row implies that $t^*(UP,v)(t(UP,v)(e))$ is semantically equivalent to e.

In order to show the equivalence of L1 and L2 we need one last property: $t(UP,v)(\bullet)$ and $t^*(UP,v)(\bullet)$ must be constant throughout the FOR-loop. This is guaranteed by the restriction (r2) in sect. 2. A consequence of this is: there is a $k \in \mathbb{Z}$ such that $s(t(UP,v)(e)) = s(e) + k$ for any arithmetic expression e and for any state s during the execution of the loop, where k can be derived from table 1. Analogously, we have $s(t^*(UP,v)(e)) = s(e) - k$.

With these properties we can now show that in L1 and in L2 BODY is executed for the same sequence of values of the iteration expression. For L1 we obtain $BODY^{s1(LO1)..s1(UP1)}$. For L2 we obtain

$$
\begin{aligned}
& \{BODY^{i}_{t^*(i)}\}^{s1(t(LO1))..s1(t(UP1))} \\
= \; & BODY^{s1(t^*(s1(t(LO1)))) \,..\, s1(t^*(s1(t(UP1))))} && \text{-- } s(t(e)) = s(e) + k \\
= \; & BODY^{s1(t^*(s1(LO1)+k)) \,..\, s1(t^*(s1(UP1)+k))} && \text{-- } s(t^*(e)) = s(e) - k \\
= \; & BODY^{s1(s1(LO1)+k)-k \,..\, s1(s1(UP1)+k)-k} && \text{-- } s(a+b) = s(a) + s(b),\ s(s(e)) = s(e) \\
= \; & BODY^{s1(LO1) \,..\, s1(UP1)}
\end{aligned}
$$

3.3 Determination of Common Conjuncts

According to the observation in (4) we present a refinement of the basic strategy by exempting common invariant conjuncts from RCPV. Common conjuncts often occur in programs with nested loops. One example is example 17 in [FKW02; KW99b]. A second example is the algorithm (7), which computes the ∞-norm p of the matrix a of size m×n, which is defined as: $p = \langle \text{Max } k: 1 \le k \le m: \langle \Sigma\ c: 1 \le c \le n: |a(k,c)| \rangle \rangle$ [GL89].

```
   -- PREo: {m,n} ≥ 1 ∧ p = 0
FOR i IN 1..m LOOP
   s := 0;
   -- PREi: s = 0 ∧
      --      p = ⟨Max k: 1..i-1: ⟨Σ c: 1..n: |a(k,c)|⟩⟩
   FOR j IN 1..n LOOP
      s := s+abs(a(i,j));                                          (7)
    END LOOP;
      -- POSTi: s = ⟨Σ c: 1..n: |a(i,c)|⟩ ∧
      --        p = ⟨Max k: 1..i-1: ⟨Σ c: 1..n: |a(k,c)|⟩⟩
   IF s>p THEN p := s; END IF;
      -- p = ⟨Max k: 1..i: ⟨Σ c: 1..n: |a(k,c)|⟩⟩
END LOOP;
   -- POSTo: p = ⟨Max k: 1..m: ⟨Σ c: 1..n: |a(k,c)|⟩⟩
```

PREi and POSTi have one conjunct in common, in which the upper bound must not be replaced by the loop variable to obtain an HI. This HI is an invariant of the inner loop.

We determine common conjuncts as follows

(a) transform PRE and POST into normal form NF

(b) determine the syntactically common conjuncts $C = C1 \wedge \ldots \wedge Cn$

(c) determine those Ci for which
noWrite(BODY, free(Ci)) $\vee$ [Ci $\Rightarrow$ wp(BODY, Ci)] holds;
noWrite(S, M) means that no variable in the set of variables M is written to in the statement S.
Let C_{com} be the conjunction of these Ci.

The condition in (c) seems to be unnecessarily complex because, from a theoretical point of view, trans(BODY, Ci) $\equiv$ [Ci $\Rightarrow$ wp(BODY, Ci)] is necessary and sufficient. From a practical point of view we have to bear in mind that a theorem prover may not be able to prove a theorem. On the other hand, noWrite(BODY, free(Ci)) can often be checked more easily, especially in the absence of function calls. Furthermore it is sufficient but not necessary. Therefore, using the combination of both conditions may classify more Ci as an invariant versus using either one alone.

The details of the normal form are given in [KW00].

If the method finds any invariant common conjuncts the normalized POST can be written as POST' $\wedge C_{com}$, where POST' does not contain any conjunct of C_{com}. The hypothetical invariant is then

$$HI \equiv POST'^{r(t(UP))}_{i} \wedge C_{com} \quad \text{or}$$
$$HI \equiv POST'^{r(-t(UP))}_{(-i)} \wedge C_{com}.$$

3.4 Adaptation of the Proof Rule

The bound transformation and the common conjuncts must now be considered in the proof rule for the FOR-loop. There are four factors which influence the adaptation of the proof rule:

a) direction of the loop: upwards / downwards

b) bound modification in BODY: bounds are modified / bounds are not modified

c) form of the transformed final bound: v / -v

d) occurrence of the loop variable in POST: $i \in free(POST)$ / $i \notin free(POST)$

The proof rule for the case

(upwards, not modified, v, $i \notin free(POST)$)

is given in (8).

$$\begin{aligned}
&[\ PRE \Rightarrow LO, UP \in Ti\ ] \ \wedge \\
&[\ PRE \wedge LO > UP \Rightarrow POST'\ ] \ \wedge \\
&[\ PRE \wedge LO \leq UP \Rightarrow wp(\ BODY^{i}_{LO}, POST'^{r(t(UP))}_{t(LO)}\ )\ ] \ \wedge \\
&[\ t(LO) \leq i < t(UP) \wedge POST'^{r(t(UP))}_{i} \wedge C_{com} \Rightarrow wp(\ BODY^{i}_{t*(i)+1}, POST'^{r(t(UP))}_{i+1}\ )\ ]
\end{aligned} \tag{8}$$

The proof rules for the other cases are given in [KW00].

3.5 Algorithm for the Application of the Method

We are now ready to put the pieces together and present the application of the method as an algorithm, which works for both upwards and for downwards counting AFLs.

```
-- INPUT: PRE, POST, i, LO, UP, BODY, UpwardsCounting?
AFLCorrect?: enum(proof, open) := open;
FinalBound: expression;

IF UpwardsCounting?
THEN FinalBound := UP; ELSE FinalBound := LO; END IF;

IF FinalBound is suitable (see table 1)
THEN  C_com: expression := true;
      HI: expression;
      Post': expression := POST;

      IF there is a common conjunct c with
         noWrite(BODY, free(c)) ∨ [c ⇒ wp(BODY,c)]
      THEN C_com := ⟨∧ c: c is common conjunct:
                      noWrite(BODY, free(c)) ∨
                      [c ⇒ wp(BODY, c)]⟩;
           POST' := con(set(POST) - set(C_com));
      END IF;

   -- create the set T of all possible translations
   -- t(FinalBound,v),
   -- where v ∈ free(FinalBound) ∩ free(POST);
   FOR EACH t ∈ T DO
      -- r(t(FinalBound)) = v  ∨ r(t(FinalBound))= -v
      IF r(t(FinalBound))= v
      THEN POST' := POST'^v_i;
      ELSE POST' := POST'^v_(-i);
      END IF;
      IF the AFL can be proved using POST' and C_com in the
         appropriate rule in sect. 3.4
      THEN AFLCorrect? := proof; EXIT;
      END IF;
   END FOR;
END IF;

-- OUTPUT: AFLCorrect?
```

The functions con(·) and set(·) are defined as follows:

$$\text{set}(C_1 \wedge \ldots \wedge C_n) = \{ C_1, \ldots, C_n \},$$

$$\text{con}(\{ C_1, \ldots, C_n \}) = C_1 \wedge \ldots \wedge C_n$$

The meaning of the three possible outcomes (proof, open, nontermination) has already been explained in section 3.1.

3.6 Examples

In the following example (9) the natural numbers in the range m .. m-n (for n≤0) are summed up.

```
   -- PRE: s = 0 ∧ m ≥ 0 ∧ n≤0
FOR i IN m .. m-n LOOP
   s := s + i;                                                    (9)
END LOOP;
   -- POST: s = ⟨Σ j: m..m-n: j⟩
```

UP is suitable, $C_{com} \equiv$ true, free(UP) ∩ free(POST) = {m, n},
2 transformations are possible:
t1(m-n, m)(e) = e+n and t2(m-n, n)(e) = e-m.
t1 does not yield an invariant, but t2 gives

HI ≡ s = ⟨Σ j: m..m+i: j⟩

which is an invariant of the transformed loop. The transformed loop and HI satisfy (8).

The second example is from [PS05] and computes the sum of squares of the first n natural numbers. An equivalent AFL is (10).

```
   -- PRE: n≥0 ∧ n≤1860 ∧ n=N ∧ x=0
FOR y IN 0..n LOOP
   x := y*y + x;                                                  (10)
END LOOP;
   -- POST: x=(2n^3+3n^2+n)/6 ∧ x∈int32 ∧ n≥0 ∧ n≤1860 ∧ n=N
```

Additionally to [PS05] we assume that x ∈ int32 and use n in POST instead of y, which may not be in scope. Since UP is a simple variable we obtain directly

HI ≡ $x = (2y^3 + 3y^2 + y)/6 \wedge x \in \text{int32} \wedge n \geq 0 \wedge n \leq 1860 \wedge n = N$

HI is an invariant and the loop (10) together with HI satisfies (8).

4 Conclusion

We have developed a method for the mechanical generation of invariants for a practically relevant class of FOR-loops. The method can be incorporated into automatic program provers and would lead to a simplification of program verification using such a tool. By extending the suitable forms of the final bound the applicability of the method could be extended to further classes of FOR-loops.

Acknowledgments. We are grateful for the very useful hints of an anonymous referee which led to a number of improvements of the paper.

References

ABB05 Ahrendt, Wolfgang; Baar, Thomas; Beckert, Bernhard; et al.: The KeY tool. Software Syst Model 4 (2005) 32..54. DOI 10.1007/s10270-004-0058-x

Bar00 Barnes, John: High Integrity Ada - The SPARK Approach -. Addison-Wesley, 2000.

Bas80 Basu, S. K.: A Note on Synthesis of Inductive Assertions. IEEE TSE 6, 1 (1980) 32..39

BCD06 Barnett, M; Chang, B-Y E.; DeLine, R. et al.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. Springer, LNCS 4111, Berlin, 2006, pp. 364..387

BD84 Dunlop, D. D.; Basili, V. R.: A Heuristic for Deriving Loop Functions. IEEE TSE 10, 3 (1984) 275..285

BG91 Gonnet, G. H.; Baeza-Yates, R.: Handbook of Algorithms and Data Structures. Addison Wesley, Wokingham, 1991. ISBN-10: 0-201-41607-7

BMM01 Ball, T.; Majumdar, R.; Millstein, T.; Rajamani, S. K.: Automatic Predicate Abstraction of C Programs. ACM PLDI 2001, 203..213

Cap75 Caplain, Michel: Finding Invariant Assertions for Proving Programs. SIGPLAN Notices 10, 6 (1975) 165..171

CEG99 Ernst, M. D.; Cockrell, J.; Griswold, W. G.; Notkin, D.: Dynamically Discovering Likely Program Invariants to Support Program Evolution. ICSE '99, 213..224

CEG00 Ernst, M. D.; Czeisler, A.; Griswold, W. G.; Notkin, D.: Quickly Detecting Relevant Program Invariants. ICSE 2000, 449..458

Col88 Collins, W. J.: The Trouble with FOR-Loop Invariants. ACM SIGCSE Bull. 20, 1 (1988) 1..4

Dij76 Dijkstra, Edsger W.: A Discipline of Programming. Prentice-Hall, Inc., 1976.

Ell81 Ellozy, H. A.: The Determination of Loop Invariants for Programs with Arrays. IEEE TSE 7, 2 (1981) 197..206

FKW02 Ein Vergleich der Programmbeweiser FPP, NPPV und SPARK. Ada-Deutschland Tagung 2002. Shaker Verlag, Aachen, 2002. p. 127..145. ISBN-10: 3-8265-9956-X

Flo67 Floyd, R. W.: Assigning Meaning to Programs. In: Schwartz, J. T. (ed.): Mathematical Aspects of Computer Science. AMS, 1967, pp. 19 .. 32. ISBN-10: 0-8218-1319-6

FQ02 Flanagan, C.; Qadeer, S: Predicate Abstraction for Software Verification. ACM POPL'02, 191..202

GDM85 Mili, A.; Desharnais, J.; Gagné, J.-R.: Strongest Invariant Functions: Their Use in the Systematic Analysis of While-Statements. Acta Informatica 22 (1985) 47..66

GL89 Golub, G.H.; Loan, C.F. van: Matrix Computations. John Hopkins Press, 1989

Gri81 Gries, D.: The Science of Programming. Springer, New York, 1981

Gri82 Gries, D.: A Note on a Standard Strategy for Developing Loop Invariants and Loops. Sci Comp Progr 2 (1982) 2007..214

Gumxx Gumm, H: New Paltz Program Verifier. http://www.mathematik.uni-marburg.de/~gumm/NPPV/nppv.html. Visited 2007.Feb.21

Hoa69 Hoare, C. A. R.: An Axiomatic Basis of Computer Programming. CACM 12, 10 (1969) 576..580, 583

Hoa72 Hoare, C. A. R.: A Note on the FOR Statement. BIT 12 (1972) 334..341

JK05 Kovács, L. I.; Jebelean, T.: An Algorithm for Automated Generation of Invariants for Loops with Conditionals. Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 2005

JKP03 Kovács, L. I.; Popov, N.; Jebelean, T.: Verification of Imperative Programs in Theorema. Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 2003

Kau99 Kauer, S.: Automatische Erzeugung von Verifikations- und Falsifikationsbedingungen sequentieller Programme. Dissertation, Friedrich Schiller University, 1999.Jan.27

KM76 Katz, S.; Manna, Z.: Logical Analysis of Programs. CACM 19, 4 (1976) 188..206

KR06 Rodríguez-Carbonell, E.; Kapur, D.: Automatic generation of polynomial invariants of bounded degree using abstract interpretation. Sci Comp Progr 64 (2007) 54..75. Available online 28 Sept 2006 at http://www.sciencedirect.com/; visited 2007.Jan.22

KW99a Kauer, S.; Winkler, J. F. H.: FPP: An Automatic Program Prover for Ada Statements. Workshop "Objektorientierung und sichere Software mit Ada". Karlsruhe, 1999.Apr.21-22

KW99b Kauer, S.; Winkler, J. F. H.: A Comparison of the Program Provers NPPV and FPP. Report Math / Inf / 1999 / 28, Friedrich Schiller University, Dept. of Math. & Comp. Sci., 1999

KW00 Kauer, S.; Winkler, J.F.H.: Automatic Generation of Invariants for FOR-Loops Based on an Improved Proof Rule. Report Math / Inf / 2000 / 26, Friedrich Schiller University, Dept. of Math. & Comp. Sci., 2000

Mis78 Misra, J.: Some Aspects of the Verification of Loop Computations. IEEE TSE 4, 6 (1978) 478..486

MS03 Müller-Olm, M.; Seidl, H.: Computing Polynomial Program Invariants. October 2, 2003. 2007Feb11 from: http://www.informatik.fernuni-hagen.de/forschung/ informatikberichte/pdf-versionen/310.pdf

MSS04 Sankaranarayanan, S.; Sipma, H. B.; Manna, Z.: Non-Linear Loop Invariant Generation using Gröbner Bases. ACM POPL 2004, 318..329

MW77 Morris, J. H. Jr.; Wegbreit, B.: Subgoal Induction. CACM 20, 4 (1977) 209..222

Pai86 Paige, R.: Programming with Invariants. IEEE Software 3, 1 (1986) 56..69

PS05 Seidl, H.; Petter, M.: Inferring Polynomial Invariants with Polyinvar. Technische Universität München, Garching, Germany. 2007.Feb.12 from: http://www2.cs.tum.edu/ ~petter/papers/nsad05.pdf

Tam80 Tamir, M.: ADI: Automatic Derivation of Invariants. IEEE TSE 6, 1 (1980) 40..48

Tur49 Turing, A.: Checking a Large Routine. In: Williams, L. R.; Campbell-Kelly, M. (eds.): The Early British Computer Conferences. MIT Press, Cambridge, 1989, 70..72

Weg74 Wegbreit, Ben: The Synthesis of Loop Predicates. CACM 17,2 (1974) 102..112

Win97 Winkler, J.F.H.: The Frege Program Prover. 42. Int. Wiss. Koll., Ilmenau, 1997. Vol.1 116..121

Win98 Winkler, J.F.H.: New Proof Rules for FOR-loops. Report Math / Inf / 98 / 13, Friedrich Schiller University, Dept. of Math. & Comp. Sci., 1998