STᴏP : Scalable Termination analysis of (C) Programs

Guillaume Andrieu [1]

University of Lille
Villeneuve d'Ascq, France

Christophe Alias [2]

LIP/INRIA
University of Lyon
Lyon, France

Laure Gonnord[3]

LIFL
University of Lille
Villeneuve d'Ascq, France

Abstract

In this paper we describe a general method to prove termination of C programs in a scalable and modular way. The program to analyse is reduced to the smallest relevant subset through a termination-specific slicing technique. Then, the program is divided into pieces of code that are analysed separately, thanks to an external engine for termination. The result is implemented in the prototype STᴏP over our previous toolsuite WTC ([2]) and preliminary results shows the feasibility of the method.

Keywords: Static Analysis, Termination, Ranking functions, Modularity, Interprocedural Analysis, Slicing.

1 Introduction

Although proving program termination is known to be undecidable, recent progress in program analysis made it possible to predict the termination of an always increasing class of sequential programs. The standard approach of the seminal paper [9] which consists in finding a function from the states of the program to some well-founded set, which strictly decreases at each program point, remains standard. In [2], we proposed a general algorithm to discover

[1] Email: andrieuguillaume42@gmail.com
[2] Email: christophe.alias@ens-lyon.fr
[3] Email: laure.gonnord@lifl.fr

multidimensional ranking functions from flowcharts programs, by means of the resolution of linear programing (LP) instances. However, our experiments showed two main problems to scale to larger C programs :

- Although our C parser, namely C2fsm ([8]) was designed to handle a large part of C syntax, many syntactical variants are not handled. Most of them could be ignored to prove the termination.
- Moreover, the size of the LP problems increases with the number of code lines, and thus quickly become intractable.

In this paper, we propose an effective modular interprocedural termination analysis, which relies on the previous method, but enables to analyse a broad range of C programs, and validate this approach on a large benchmark of the literature. Our method relies on classical methods from static analysis and compilation, such as slicing and summaries, but as far as we know, is the first attempt in proving termination of a large set of C programs in a modular way.

The rest of the paper is organized as follows. In section 2, we introduce our motivation of a challenging program of the literature. In section 3 we quickly introduce our notation and theoretical foundations. In section 4 we describe our method, and we evaluate it on a large bench of middle-sized programs in section 6. We end with related works (section 7) and a conclusion.

2 Motivating example

In this paper we detail our method on an implementation of the merge sort of an array. The code is taken from [5] and is depicted in Figure 1. For sake of lisibility, we drawed boxes around innerloops, and commented the end of outer loops.

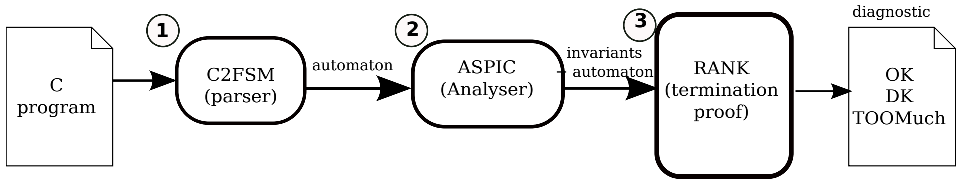

Fig. 2. WTC toolchain

Calling our tool suite (Figure 2) detailed in [2] and [8] gives an intermediate automaton of 12 variables, 80 transitions and 9 locations. Aspic successfully computes the invariants but Rank fails with `TWO_MUCH_VARIABLES` error because the size of the underlying linear programming problem is too big.

The whole computation time is 12 seconds on a `Intel Core2 @ 1.60GHz`. In the following, we will show how our method SToP , finally manages this code to prove its termination.

```
int main() {
  int n,i,j,k,l,t,h,m,p,q,r;
  int up;    /* really boolean */
  int a[2*n+1]
  up = 1;  p = 1;

 loop4 : do{ // sorting a
    h = 1;
    m = n;
    if(up == 1){
      i = 1;
      j = n;
      k = n+1;
      l = 2*n;
    } else {
      k = 1;
      l = n;
      i = n+1;
      j = 2*n;
    }
  loop5 : do{
      if(m >= p)
        q = p;
      else q = m;
      m = m-q;
      if(m >= p)
        r = p;
      else r = m;
      m = m-r;

    loop0 : while(q>0 && r>0) {
        if(a[i] < a[j]){
          a[k] = a[i];
          k = k+h;
          i = i+1;
          q = q-1;
        } else {
          a[k] = a[j];
          k = k+h;
          j = j-1;
          r = r-1;}


    loop1 : while(r > 0){
        a[k] = a[j];
        k = k+h;
        j = j-1;
        r = r-1;      }

    loop2 : while(q > 0){
        a[k] = a[i];
        k = k+h;
        i = i+1;
        q = q-1;
      }

      h = -h;
      t = k;
      k = l;
      l = t;
    } while(m > 0);//end of loop5

    up = 1-up;
    p = 2*p;

  } while(p<n);//end of loop4

//final copy of the array in the first half of a
  if(up == 0){
    i = 1;

    loop3 : while(i <= n){
      a[i] = a[i+n];
      i = i+1;
    }

    }//end if test up==0

  return 0;
}
```

Fig. 1. Our motivating example: iterative merge sort

3 Preliminaries

3.1 Grammar for intraprocedural analysis

We define in Figure 3 a rather classical "mini while" grammar for programs, named G. Expressions are simple Boolean and numerical expressions without side-effects. The semantics of the programs is rather classical too and we do not detail it.

On this kind of programs, the problem of proving termination is undecidable, but like in [2] we will provide a sound (but not complete) analysis that is able to deal with a large bench of problems. Basically, the method described in this paper is based on the fact that while loops can be proven to be terminating quite independently of the other ones, provided that we keep an execution context precise enough, thanks to the notion of *summary*.

$\langle$prog$\rangle$::=$\langle$declaration list$\rangle$ $\langle$statement list$\rangle$
$\langle$declaration list$\rangle$::=$\langle$decl$\rangle$ $\langle$declaration list$\rangle$| (empty)
$\langle$decl$\rangle$::=**bool** $\langle$ident$\rangle$ | **int** $\langle$ident$\rangle$
$\langle$statement list$\rangle$::=$\langle$statement$\rangle$ $\langle$statement list$\rangle$ | (empty)
$\langle$statement$\rangle$ ::= $\langle$assignment$\rangle$ | $\langle$ifstat$\rangle$ | $\langle$forstat$\rangle$ | $\langle$whilestat$\rangle$
|$\langle$assertstat$\rangle$ | $\langle$breakstat$\rangle$
$\langle$assignment$\rangle$ ::= $\langle$var$\rangle$ **:=** $\langle$expr$\rangle$
$\langle$ifstat$\rangle$ ::= **if** $\langle$expr$\rangle$ **then** $\langle$statement list$\rangle$ **else** $\langle$statement list$\rangle$
$\langle$forstat$\rangle$ ::= **for** $\langle$var$\rangle$ **from** $\langle$expr$\rangle$ **to** $\langle$expr$\rangle$ **do** $\langle$statement list$\rangle$
$\langle$whilestat$\rangle$ ::= **while** $\langle$formula$\rangle$ **do** $\langle$statementlist$\rangle$
$\langle$breakstat$\rangle$ ::= **break**
$\langle$assert stat$\rangle$ ::= **assert** $\langle$formula$\rangle$

Fig. 3. A simple grammar G for programs

3.2 Summaries

Definition 3.1 Let $C \in G$ be a statement and $R_C(x_0, y_0, x, y)$ the relationship between the values of the variables before and after C. $C' \in G$ is said to be an abstraction of C if $R_C \subseteq R_{C'}$.

It is obvious that if we manage to prove that C' terminates, then C terminates. Thanks to linear relation analysis, we are able to compute such abstractions of code behaviours. We again use Aspic ([8]) with the appropriate option to obtain a polyhedral over-approximation $\widetilde{R_C}(x_0, y_0, x, y)$ of the behaviour of a given code C.

Definition 3.2 [Summary] Let C be a code and $R_C(x_0, y_0, \ldots, x, y)$ be an over-approximation of the relation between initial variables $x_0, y_0, \ldots$ and final variables $x, y, \ldots$. Then the following code is called a *summary* of C :

```
x0 = x; y0 = y; ...
x = random(); y = random();...
if(!R_C(x0,y0, ..., x, y, ...)) break
```

The *summary* of C is obviously an abstraction of C and thus can be used for proving the termination of C.

4 Intraprocedural modular termination

This section presents a termination procedure able to handle large programs. Our termination procedure is based on the following remarks.

- The program terminates if and only if all the conditional loops (referred as “while loops” in the following) terminate. Hence, the termination analysis can focus on the smallest program *slice* which contains the while loops and the minimum subset of instructions to preserve their behaviour. This preprocessing is called *termination-specific slicing* and described in subsection 4.1. In general, the slicing reduces drastically the program size. However, this may not be sufficient (see for example our motivating example) and the following analysis is generally required.

- Basically, a while loop terminates if and only if (i) the body terminates for each iteration and (ii) the while loop itself terminates. This simple idea allows to decompose a big termination problem into several termination sub-problems, small enough to be processed successfully by WTC. The detail of the decomposition is presented in subsection 4.2.

4.1 Step 1. Termination-Specific Slicing

The termination-specific slicing step is a preprocessing which extracts the program subset relevant for proving termination. This subset must contains the while loops and the instructions required to preserve their behaviour.

```
if(up == 0){
  i = 1;
  while(i <= n){
    a[i] = a[i+n];
    i = i+1;
  }
}
```

(a) Before slicing

```
if (up == 0) {
  i = 1;
  while(i <= n){
    i = (i + 1);
  }
}
```

(b) After slicing

Fig. 4. Example for slicing taken from our motivating example

Consider the example given in figure 4. Clearly, for each variable used in the while condition, the definition sites must be kept (here, `i=1` and `i=i+1`). But the variables read by these definitions sites must be kept as well, and so on. This process boils down to a backward traversal of the data dependencies, starting from the variables used in the while condition. Also, control dependencies must be taken into account. On the example, the while loop will be executed only when the `then` branch is chosen. In turn, this depends on the `up` variable, on which the same process must be applied. These are the basic ideas of program slicing, for which many variants has been developed [13] for various purposes as reverse engineering, program comprehension,...

Our termination-specific slicing algorithm is depicted in Algorithm 1.

- Do_Slice(function_body), the main function, selects the set of while statements of the current function_body and runs the main slicing function, Slice. Do_Slice is applied to each function body of the program.
- Slice(stmt_set) proceeds the definition sites of stmt_set with a depth-first traversal, flooding the connex part of stmt_set in the meaning of Definition_Sites. Notice that the function mark(stmt) adds the *line* of

stmt to the slice. When stmt is a control structure (if, for,while,...), only the control lines are added (*e.g.* if, else, {, }), not the body.

- Definition_Sites(stmt) computes the set of statements whose stmt depends immediately, in the meaning of data- and control-dependence. More precisely, Definition_Sites computes the *reaching definitions* [1] $\mathrm{RD}_{\mathrm{stmt}}(\texttt{xi})$ of each variable `xi` read by stmt. The result is a set of assignments which can possibly define the value of `xi` read by stmt. Also, the immediate compound control structure of stmt, Compound(stmt), is added if it exists. This way control dependencies are taken into account.

Algorithm 1 Termination-Specific Slicing Algorithm

```
 1: function Do_Slice(function_body)
 2:     while_loop_set = set of while statements
 3:     Slice(while_loop_set)
          //At this point, all the statements belonging to the slice are marked
 4: end function
 5: function Slice(stmt_set)
 6:     for all stmt ∈ stmt_set do
 7:         if is_not_marked(stmt) then
 8:             mark(stmt) //Add stmt to the slice
 9:             Slice(Definition_Sites(stmt))
10:         end if
11:     end for
12: end function
13: function Definition_Sites(stmt)
14:     if stmt = x := expr[x1,...,xn] then
          return $RD_{stmt}(x1) \cup \ldots \cup RD_{stmt}(xn) \cup$ Compound(stmt)
15:     end if
16:     if stmt = if cond[x1,...,xn] then S1 else S2 then
          return $RD_{stmt}(x1) \cup \ldots \cup RD_{stmt}(xn) \cup$ Compound(stmt)
17:     end if
18:     if stmt = for i from expr1[x1,...,xn] to expr2[y1,...,yp] do S then
          return $RD_{stmt}(x1) \cup \ldots \cup RD_{stmt}(yp) \cup$ Compound(stmt)
19:     end if
20:     if stmt = while cond[x1,...xn] do S then
          return $RD_{stmt}(x1) \cup \ldots \cup RD_{stmt}(xn) \cup$ Compound(stmt)
21:     end if
22: end function
```

Example (cont'd). Let us apply our slicing algorithm to the piece of code given figure 4. This code has been picked from the end of the motivating example. For the sake of the presentation, the while statement will be denoted by "while" and the if statement by "if".

- Do_Slice call Slice with while_loop_set = { while }. Initially, the slice is empty, no lines are marked. **Lines 3 and 6 are marked**, then Definition_Sites(while) is run and gives $RD_3(i) \cup RD_3(n) \cup \{\text{if}\} = \{\text{i=1}, \text{i=i+1}, \text{if}\}$.
- Slice is then called on {i=1,i=i+1,if}. i=1 is processed first. **Line 2 is marked** and Definition_Sites(i=1) is computed. i=1 does not read variables, thus we get $RD_2(\emptyset) \cup \{\text{if}\} = \{\text{if}\}$.

- Slice is then called on { if }. **Lines 1 and 7 are marked**. Definition_Sites(if) gives $RD_1(\text{up}) \cup \emptyset = \emptyset$. Indeed, `up` is not defined in this piece of code and if has no compound control structure. So, no additional recursive call is issued.
- Going back on the recursive calls, Slice({ i=i+1 }) is called. **Line 5 is marked**, then Definition_Sites(i=i+1) is computed and gives $RD_5(\text{i}) \cup \{\text{if}\} = \{\text{i=1}, \text{i=i+1}, \text{if}\}$.
- Slice is then called on {i=1,i=i+1,if}. All these statements were already processed, so no further recursive exploration is done.
- Do_Slice ends up with the slice defined by the marked lines **{1,2,3,5,6,7}**. This slice excludes a[i]=a[i+n] which, indeed, does not influence the termination of the while loop.

4.2 Step 2. Scalable Termination

This section presents a scalable termination procedure, which scatters the termination proof into several proof obligations, on small programs with a single conditional loop. These proof obligations will be small enough to be handled successfully with the termination method WTC, presented in [2].

Here, although WTC is a tool that can handle programs (with nested loops), we can see WTC as a termination engine that takes as input a single loop *and an execution context*, and tries to prove that this loops terminates under the given context. WTC can either answer `OK`, which means that the program actually terminates, or `DK`, which is inconclusive.

Basically, our algorithm is based on the structure of the program to analyse. As expressions have no side-effects, they terminate. Breaks and asserts terminate. A sequence of instructions terminates if each terminates (given their execution context). For tests, we have to prove the termination of each branch under their respective contexts. A `for` loop terminates if its body terminates (in our simple version of `for` loops). A (single) `while` loop terminates if we are able to exhibit a ranking function. This is done by a call to our tool `WTC`. To handle nested loops, we can replace the inner loop by a summary (3.2) of its behaviour, and call WTC on the outer loop with a lesser body.

Algorithm

The previous remarks are implemented in Algorithm 2. The main function ModularTerm takes as input a statement and a structure that is able to give an over approximation of the context of each statement of the program under analysis. These invariants are computed using linear relation analysis [7], implemented in the Aspic tool ([8]). ModularTerm proceeds the abstract syntactic tree with a depth-first traversal, evaluating each sub-tree to the value `OK` if it terminates, or `DK` if the analysis does not succeed to answer. If a sub-tree is evaluated to `DK`, the analysis fails and stops with `DK` output.

Algorithm 2 Scalable Algorithm for proving termination

```
function ModularTerm(statement,pcinvs)
    if statement = x:=expr then return $OK$
    end if
    if statement = st1;st2 then
        $res \leftarrow ModularTerm(\texttt{st1}, pcinvs)$
        if $res = OK$ then
            return $ModularTerm(\texttt{st2}, pcinvs)$
        else
            return $DK$
        end if
    end if
    if statement = if cond then S1 else S2 then
        $res \leftarrow ModularTerm(\texttt{st1}, pcinvs)$
        if $res = OK$ then
            $res' \leftarrow ModularTerm(\texttt{st2}, pcinvs)$
            return $res'$
        else
            return $DK$
        end if
    end if
    if statement = for i from exp1 to exp2 do S then
        return $ModularTerm(\texttt{S}, pcinvs)$
    end if
    if statement = while cond do S then
        $res \leftarrow ModularTerm(\texttt{S}, pcinvs)$
        if $res = DK$ then
            return $DK$
        else
            $context \leftarrow getContext(statement, pcinvs)$
            $res \leftarrow WTC(context, statement)$
            if $res = DK$ then
                return $DK$
            else
                Compute a summary of the loop and replace in the code.
                return $OK$
            end if
        end if
    end if
end function
```

Let us focus on `while` loops. To analyse loops, the algorithm :

(i) First tries to prove the loop body to terminate. If the loop body cannot be proved to terminate, then the analysis fails without having to analyse the loop itself.

(ii) If the analysis succeeds, the termination of the loop is checked by applying the WTC method described in [2] on the loop itself, under its *context*.

(iii) Then the loop is replaced by its *summary* (section 3.2).The analysis continues with this last code. Notice that the subsequent calls to WTC will have to deal with simpler programs with a single loop, which reduces scalability issues.

Notice that linear relation analysis expresses the invariants by a convex polyhedron, which over-approximate the actual possible values of the variables. As a consequence, the precondition of the loop needed to apply WTC (stored in *pcinvs*), as well as the summary of the loop are over-approximated. These approximations can make unprovable programs which could be proved successfully with WTC, on the cases where the ILP can be processed with a reasonable amount of resource (time and memory) (see Section 6).

Example (cont'd). Our termination procedure processes the loops 0, 1, 2, 5, 4 and 3 in this order (deep-first search). For the sake of the presentation, we will restrict the discussion to the first steps of the algorithm.

After instrumenting the code with loops labels and calling C2FSM and ASPIC on the resulting code, we find the following execution contexts for the loops :

- **loop0** : $\{r \geq 0, p \geq q, p \geq r, p \geq 1, m \geq 0, i + m + q + r \geq j + 1\}$
- **loop5** : $\{i + m + q \geq j + 1, m \geq 0, p \geq 1q \leq 0, r = 0\}$

WTC succeeds to prove the termination of **loop0** under its context (its finds the ranking function $q+r$), and also gives the following approximation for its behaviour : $\{j \leq j_0 \wedge i \geq i_0 \wedge h = h_0 \wedge n = n_0 \wedge i+q = i_0+q_0 \wedge j+r_0 = r+j_0\}$ whose summary is :

```
i0 = i; j0 = j; q0 = q; r0 = r; h0 = h; n0 = n;
if(!(<=j0 && i>= i0 && h=h0 && n=n0 && i+q=i0+q0 && j+r0=r+j0))
   break;
```

Loop1 and **loop2** are processed in the same way. At the end, the outer loop **loop5** no longer contains while loops, which were all substituted by an abstraction of their effect on the variables.

The termination of **loop5** comes from the two first conditions at the beginning of **loop5**, whose global effect would be equivalent to the assignment `m = m - 2*min(p,m)`. As $p \geq 1$, m strictly decreases at each step of the loop and will end up with the value 0, ensuring the termination. A call to WTC ends with m as ranking function and gives a summary for **loop5**. In the same way, the process continues with the remaining **loop4** and **loop3**, and ends up with the final answer **OK**, meaning that the program always terminates.

5 Extensions

5.1 Interprocedural analysis

We add the notion of function calls and a `main`. Dealing with local variables, parameters, and function calls is rather classical. The main restrictions are that we avoid recursive functions and modifications of the parameters and global variables during the function calls.

⟨function⟩::= ⟨declaration list⟩ ⟨statement list⟩
⟨funcall⟩::= ⟨var⟩ **:=** ⟨functionname⟩ **(** ⟨var1⟩ **...** ⟨varn⟩ **)**

To handle functions, we first compute invariants for all program functions f: $pcinvs_f$. Then, if the current statement is a function call `x:=f(x1,...xn)`:

(i) Compute the projection of the call context at the current control point ($pcinvs_{caller}(progpoint)$) on the union of variables $x_1 \dots x_n$ and the global variables. This invariant is called *callcontext*.

(ii) Call recursively $ModularTerm$ on the function body whose context is modified by the call context : $pcinvs_{callee} \cap callcontext$.

5.2 Extensions to "full" C

For the moment, our tool is restricted to a subset of C and could not handle a general C program. However, it is always possible to extend the set of analysable programs with an appropriate set of normalizations, implemented as a preprocessing. We give thereafter two important normalizations which should enlarge drastically the class of analyzable programs.

For loops We assume the `for` loops to be Pascal-compliant: `for i from expr1 to expr2 do BLOCK(i);`. In C, the `for` loops can be much more general and used, for example, to iterate on linear data structures: `for(ptr=first; ptr != NULL; ptr=next(ptr)) BLOCK(ptr)`. It is easy to design a preprocessing which detects non-Pascal-compliant `for` loops, and turn them into a `while` loop: `ptr=first; while(ptr!=NULL) { BLOCK(ptr); ptr=next(ptr); }`

Pointers We assume the program to be pointer-free. In C, the use of pointers complicate the data-dependence analysis, hence the slicing (because of reaching definitions) and the invariant analysis. On the following program: `p = &a; q = &a; *p=1; *q=2; x = a;` The reaching definition of `a` in `x=a` should be $RD_{\texttt{x=a}}(a)$ = `*q=2`. Also, the invariant of `x=a` should implies $x = 2$. To make these analysis possible, the interferences must be analyzed carefully, thanks to pointer analysis [3]. However, the data-dependences are over-approximated, thus this will affect the size of slices. As for the invariant computation, the obtained data-dependencies should be analyzed carefully to produce an integer interpreted automaton leading to correct invariants. This analysis is not trivial and is left for future work.

6 Experimental results

We implemented our method as a driver over WTC, written in C++. The total number of LOC is 3000 for the driver itself, and 150 additional lines for statistics. The code intensively uses the source-to-source compiler infrastructure Rose ([12]) : functions for parsing, constructing flow graphs, searching in the code structure, pretty printing and instrumenting C codes.

Table 1 give an extract of the experiments we made with our tool SToP (the entire benchmark is available on `http://compsys-tools.ens-lyon.fr/stop`.

Benchmark	**#Progs**	**LoCB**	**LoCA**	**Regressions**	**WTC(s)**	**SToP(s)**
WTC1	50	28	21	3	0.92	2.45
WTC2*	6	55	39	0	3.66	3.52

Table 1
WTC1 and WTC2 denote middle-sized benchmarks taken from our previous experiments in [2]. WTC2 consist on a bench of 8 sorting functions from the literature, and WTC1 are also classical codes including `sipmamergesort`,our motivating example. For each benchmark, we give its number of programs, the average number of lines of codes before(`LoCA`) and after (`LoCB`) slicing, `Regressions` denote the number of regressions (examples that were proven `OK` with `WTC` and for which SToP now answers `DK`). We also give average execution times of each method.

The experiments were done on a Dual Core 2Ghz. As we expected, the execution times on middle-sized examples are much larger with SToP than `WTC`, mainly because of the cost of slicing and producing intermediate files. In future versions of SToP , we will investigate the opportunity of calling `WTC` on larger subprograms (some nested loops can be handled with a unique call to `WTC`). For sorting functions, the method shows it pertinence in terms of timing results and in terms of precision (it handles `sipmamergesort` in 22 seconds, but we had to manage the very last step by hand by providing a coarser abstraction for `loop5`). Thus SToP is able to deal with larger programs than WTC and seems to scale well. More experiments remains nevertheless to be done.

7 Related works

As far as we know, this is the first tentative of proving termination of C programs in a modular way. Previous papers prove terminations of kernels, and other papers such as [4] use what they call "C abstractors" to make benchmarks but no general algorithm is given. Our algorithm is a driver built upon WTC, a previous toolchain to prove the termination of (little) C codes. To improve its precision, we could use (instead of WTC) other methods and tools to prove fragments of code :

- The methods described in previous papers on termination such as [5], [11] or more specialised methods for other kinds of programs such as lists or trees ([10]). In these last cases, the numerical invariants that we precompute may not be precise enough, though.
- Instead of testing if a given code fragment terminates under a given over-approximation of its context, we can use the method described in [6] to compute (sufficient) termination preconditions, and check if these preconditions are satisfied under the given context.

As proving termination is undecidable, all these methods may positive results when other ones timeout or give `DK` answer, thus they could be used as "on-demand" back-ends.

8 Conclusion and Future works

In this paper we have presented a framework for proving the termination of C programs in a modular way. We used `WTC` as a tool for proving termination of simple loops, but the method is fully adaptable to other termination engines. The preliminary experimental results show that they are relatively few regression cases comparing to the previous work published in [2], and we are able to deal with larger programs such as `sipmamergesort` in reasonable time.

Future work include more tests to improve the reliability of our tool suite with respect to C variants, and on much bigger programs such as linux drivers and image processing codes. In particular, the size of the sub-programs generated by STOP expresses a trade-off execution time/precision which needs to be investiguated.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986-2006.

[2] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *17th International Static Analysis Symposium, SAS'10*, Perpignan, France, September 2010.

[3] W. Amme and E. Zehendner. Data dependence analysis in programs with pointers. *Parallel Computing*, 24(3–4):505–525, May 1998.

[4] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *17th International Conference on Computer Aided Verification (CAV)*, volume 3576 of *Lecture Notes in Computer Science*, pages 491–504. Springer Verlag, July 2005.

[5] Michael A. Colón and Henny B. Sipma. Practical methods for proving program termination. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 442–454. Springer Verlag, January 2002.

[6] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. Proving conditional termination. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 328–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages (POPL'78)*, Tucson.

[8] P. Feautrier and L. Gonnord. Accelerated Invariant Generation for C Programs with Aspic and C2fsm. In *Workshop on Tools for Automatic Program AnalysiS, TAPAS'10*, Perpignan, France, September 2010.

[9] Robert W. Floyd. Assigning meaning to programs. In *Symposium on Applied Mathematics*, volume 19, pages 19–32. A.M.S., 1967.

[10] Peter Habermehl, Radu Iosif, Adam Rogalewicz, and Tomas Vojnar. Proving termination of tree manipulating programs. In *Automated Technology for Verification and Analysis*, volume 4762 of *Lecture Notes in Computer Science*, pages 145–161. 2007.

[11] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'03)*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer Verlag, 2004.

[12] D. J. Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Proc. Letters*, 2000.

[13] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 1995.