First publ. in: Tools and algorithms for the construction and analysis of systems : 12th International Conference, TACAS 2006, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006; proceedings / Holger Hermanns ... (eds.). - Berlin [u.a.] : Springer, 2006. - S. 318-333. - (Lecture notes in computer science ; 3920). - ISBN 978-3-540-33056-1

A Region Graph Based Approach to Termination Proofs

Stefan Leue and Wei Wei

Department of Computer and Information Science, University of Konstanz,
D-78457 Konstanz, Germany
{Stefan.Leue, wei}@inf.uni-konstanz.de

Abstract. Automated termination proofs are indispensable in the mechanic verification of many program properties. While most of the recent work on automated termination proofs focuses on the construction of linear ranking functions, we develop an approach based on region graphs in which regions define subsets of variable values that have different effects on loop termination. In order to establish termination, we check whether (1) any region will be exited once it is entered, and (2) no region is entered an infinite number of times. We show the effectiveness of our proof method by experiments with Java code using a prototype implementation of our approach.

1 Introduction

Automated termination proofs are indispensable in the mechanic verification of many program properties. Our interest in automated termination proofs comes from the precursory work on determining communication buffer boundedness for communicating finite state machine based models such as they occur in UML RT models [6, 7, 8]. In UML RT models the action code of a transition in a state machine can contain arbitrary program code, for instance Java code. When the action code contains a program loop within which some messages are sent, we need the information of how many times the loop iterates in order to determine how many messages are sent along the transition.

Automated termination proving has recently received intensive attention [10, 12, 2, 1, 4, 3], in particular those approaches based on transition invariants [11]. Most of the recent work [10, 1] focuses on the construction of linear ranking functions. However, loops may not always possess linear ranking functions, c.f. Example 1 in Section 2.

We develop a method to prove termination for an important class of loops, *deterministic multiple-path linear numerical loops with conjunctive conditions*, whose subclasses are also studied in [10, 12]. Given a loop, we construct one or more region graphs in which regions define subsets of variable values that have different effects on loop termination. In order to establish termination, we check for some generated region graph whether (1) any region will be exited once it is entered, and (2) no region is entered an infinite number of times. We show the effectiveness of our proof method by experiments with Java code using a prototype implementation of our approach.

Related Work. [10] gives a complete and efficient linear ranking function synthesis method for loops that can be represented as a linear inequality system. It considers nondeterministic update of variable values to allow for abstraction. However, it does not

apply to multiple-path loops. [1] can discover linear ranking functions for any linear loops over integer variables based on building ranking function templates and checking satisfiability of template instantiations that are Presburger formulas. The method is complete but neither efficient nor terminating on some loops. [2] gives a novel solution to proving termination for polynomial loops based on finite difference trees. In fact it applies only to those polynomial loops whose behavior is also polynomial, i.e., the considered guarding function value at any time can be represented as a polynomial expression in terms of the initial guarding function value. Note that Example 1 does not have a polynomial behavior. [12] proves the decidability of termination for linear single-path loops over real variables. However, the decidability of termination for integer loops remains a conjecture. [4] suggests a constraint solving based method of synthesizing nonlinear ranking functions for linear and quadratic loops. The method is incomplete due to the Lagrangian Relaxation of verification conditions that it takes advantage of.

Outline. We define loops, regions, and region graphs in Sections 2 and 3. The region graph based termination proof methods are explained for three subclasses of loops: (1) G^1P^1 in Section 4, (2) G^1P^* in Section 5, and (3) G^*P^1 in Section 6. We generalize these methods to handle the whole loop class that we consider in this paper in the end of Section 6. Experimental results are reported in Section 7 before a conclusion in Section 8.

2 Loops

We formalize the class of loops that we consider in this paper. We call this class *deterministic multiple-path linear numerical*[1] *loops with conjunctive conditions*, or G^*P^* (multiple-guard-multiple-path) in short. Loops in G^*P^* have the following syntactic form:

while lc **do**
  $pc^1 \rightarrow \bar{x}' = U^1\bar{x} + \bar{u}^1$
  ...
  $pc^p \rightarrow \bar{x}' = U^p\bar{x} + \bar{u}^p$
od

where

- $\bar{x} = [x_1, ..., x_n]^T$ is a column variable vector where T is transposition of matrices. $x_1, ..., x_n$ can be either integer variables or real variables. We use $\bar{x}' = [x'_1, ..., x'_n]^T$ to denote the new variable values after one loop iteration.
- $lc = \bigwedge_{i=1}^{m} lc^i$ is the loop condition. Each conjunct lc^i is a linear inequality in the form $\bar{a}^i\bar{x} \geq b^i$ where $\bar{a}^i = [a^i_1, ..., a^i_n]$ is a constant row vector of coefficients of variables and b^i is a constant. We call $\bar{a}^i\bar{x}$ a *guard*. We know that values of $\bar{a}^i\bar{x}$ are always bounded from below during loop iterations.

[1] With numerical loops we will not consider the rounding and overflow problems as usually considered while analyzing programs.

- Each $pc^i \rightarrow \bar{x}' = U^i\bar{x} + \bar{u}^i$ is a path with a path condition pc^i which is a conjunction of linear inequalities. We require that $\bigvee_{i=1}^{p} pc^i = \mathit{true}$, which guarantees a complete specification of the loop body. We further require that, for any i and j such that $i \neq j$, $pc^i \wedge pc^j = \mathit{false}$. This means that only one path can be taken at any given point in time.
- Each U^i is a constant matrix of dimension $n \times n$. Each $\bar{u}^i$ is a constant column vector of dimension n. They together describe how values of variables are updated along the i-th path.

If a loop has only one single path, then the loop body can be written as $\bar{x}' = U^1\bar{x} + \bar{u}^1$, in which we leave out the path condition *true*. Here are some examples of G^*P^* loops.

Example 1. This loop is an example of a loop without linear ranking functions [10]:

while $x \geq 0$ **do**
  $x' = -2x + 10$
od

Example 2. This is a loop with two paths:

while $x \geq -4$ **do**
  $x \geq 0 \rightarrow x' = -x - 1$
  $x < 0 \rightarrow x' = -x + 1$
od

Example 3. This loop has more than one inequality in its loop condition:

while $x_1 \geq 1 \wedge x_2 \geq 1$ **do**
  $\begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$
od

The three examples above represent three interesting subclasses of G^*P^* that are studied in the paper: (1) G^1P^1 are single-guard-single-path loops such as Example 1; (2) G^1P^* are single-guard-multiple-path loops such as Example 2; and (3) G^*P^1 are multiple-guard-single-path loops such as Example 3.

We say that a loop is *terminating* if it terminates on *any* initial assignment of variable values.

3 Region Graph

We define regions, positive and negative regions, still regions, and region graphs.

Definition 1. *Given a loop, a* region *is a set of vectors of variable values such that*

- *all the vectors in the region satisfy the loop condition.*
- *it forms a convex polyhedron, i.e., it can be expressed as a system of linear inequalities.*

We will also call a vector of variable values a *point*. We say that the loop iteration is *at some point* when the variables have the same values as in the point.

Definition 2. *Given a loop and a guard in the loop condition, a* positive (negative, still, resp.) region with respect to the guard *is a region such that, starting at any point in the region, the value of the guard is decreased (increased, unchanged, resp.) after one iteration.*

For instance, a positive region of Example 1 with respect to the guard x is $\{v \mid v > 10/3\}$, a negative region with respect to x is $\{v \mid 0 \leq v < 10/3\}$, and the only still region with respect to x is $\{10/3\}$. Moreover, if x is an integer variable, then there is no still region with respect to x. In the remainder, when we mention a positive (or negative or still) region, we will omit the respective guard if it is clear from the context.

Definition 3. *Given a loop and two regions R_1 and R_2 of the loop, there is a* transition *from R_1 to R_2 if and only if, starting at some point p in R_1, a point p' in R_2 is reached after one iteration. R_1 is the* origin *of the transition. R_2 is the* target *of the transition.*

In the definition, if R_1 and R_2 are distinct, then we say that R_1 is *exited* at p and R_2 is *entered* at p'. A transition is a *self-transition* if it starts and ends in one same region. We define that a self-transition on a region means that the region is neither exited nor entered.

For instance, there is a transition from the positive region $\{v \mid v > 10/3\}$ to the negative region $\{v \mid 0 \leq v < 10/3\}$ of Example 1 because $-2 \times 4 + 10 = 2$ while 4 is in the positive region and 2 is in the negative region.

Definition 4. *Given a loop, a* region graph *is a pair $< \mathbb{R}, \mathbb{T} >$ such that*

- *$\mathbb{R}$ is a finite set of pairwisely disjoint regions such that the union of all the regions is the complete set of points satisfying the loop condition.*
- *$\mathbb{T}$ is the complete set of transitions among regions in $\mathbb{R}$.*

In general, a region graph may contain regions that are neither positive, nor negative, nor still. However, the region graphs constructed by our termination proving methods contain only positive, negative, or still regions. A loop may have infinitely many region graphs.

Definition 5. *Given a region graph, a* cycle *is a sequence of transitions $< T_1, ..., T_n >$, where $n \geq 2$, such that*

- *for any two successive transitions T_i and T_{i+1}, the origin of T_{i+1} is the target of T_i;*
- *the origin of T_1 is the target of T_n.*

The condition $(n \geq 2)$ in the above definition excludes self-transitions to be cycles. Cycles such as $< T_1, T_2, T_3 >$, $< T_3, T_1, T_2 >$ and $< T_2, T_3, T_1 >$ are regarded as one

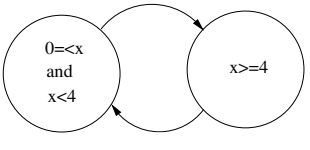

Fig. 1. A Region Graph

same cycle. A *simple cycle* is a cycle that cannot be further decomposed into smaller cycles. In the remainder, all the cycles under consideration are simple cycles.

A region graph of Example 1 is illustrated in Figure 1, assuming that x is an integer variable. There is one cycle passing the two regions.

The basic idea of our region graph based termination proofs is stated in Theorem 1.

Theorem 1. *Given a loop and one of its region graphs, the loop is terminating, if and only if, during loop iterations starting with any variable values, we have*

- *once a region is entered, it will be exited eventually.*
- *and no region is entered infinitely often.*

Proof (sketched). During loop iterations starting with some variable values, we construct a sequence of points by recording the variable values before each iteration. If the two conditions in the theorem are satisfied, then there exists no infinite sequence of points during loop iterations, and vice versa. ■

In the next sections we will show how to construct region graphs for proving termination.

4 Proving Termination for G^1P^1

We first show how to prove termination based on region graphs for loops in the simplest class G^1P^1. The concepts and methods described in this section can also apply to more general subclasses with little adaption as explained in the subsequent sections.

4.1 Constructing Region Graphs

Given a G^1P^1 loop as below,

while $\bar{a}\bar{x} \geq b$ **do**
  $\bar{x}' = U\bar{x} + \bar{u}$
od

we construct a region graph as follows in a straightforward way:

- The only positive region is defined by the system of the linear inequalities (1–3) in Figure 2 if it has solutions. Otherwise, there is no positive region.
- The only negative region is defined by the system of the linear inequalities (4–6) if it has solutions. Otherwise, there is no negative region.
- The only still region is defined by the system of the linear inequalities (7–9) if it has solutions. Otherwise, there is no still region.

$\bar{a}\bar{x} \geq b$	(1)	$\bar{a}\bar{x} \geq b$	(4)	$\bar{a}\bar{x} \geq b$	(7)
$\bar{x}' = U\bar{x} + \bar{u}$	(2)	$\bar{x}' = U\bar{x} + \bar{u}$	(5)	$\bar{x}' = U\bar{x} + \bar{u}$	(8)
$\bar{a}\bar{x} > \bar{a}\bar{x}'$	(3)	$\bar{a}\bar{x} < \bar{a}\bar{x}'$	(6)	$\bar{a}\bar{x} = \bar{a}\bar{x}'$	(9)

Fig. 2. Region defining linear inequality systems

– For a region R_1 defined by an inequality system I_1 and a region R_2 defined by I_2, there is a transition from R_1 to R_2 if the following system of inequalities has solutions: $\bigwedge_{e \in I_1} e \wedge \bigwedge_{e \in I_2} e[\bar{x} \mapsto \bar{x}', \bar{x}' \mapsto \bar{x}'']$ where $e[\bar{x} \mapsto \bar{x}', \bar{x}' \mapsto \bar{x}'']$ is the same inequality as e except that $\bar{x}$ is substituted with $\bar{x}'$ and $\bar{x}'$ is substituted with $\bar{x}''$ simultaneously.

The constructed region graph for Example 1 is exactly the one in Figure 1, assuming that x is an integer variable. The right region is positive and defined by the inequalities (10–12). The left region is negative and defined by the inequalities (13–15). There is a transition from the positive region to the negative region because the system of the inequalities (16–21) has solutions.

$$x \geq 0 \quad (10) \qquad x \geq 0 \quad (16)$$
$$x' = -2x + 10 \quad (11) \qquad x' = -2x + 10 \quad (17)$$
$$x > x' \quad (12) \qquad x > x' \quad (18)$$
$$x \geq 0 \quad (13) \qquad x' \geq 0 \quad (19)$$
$$x' = -2x + 10 \quad (14) \qquad x'' = -2x' + 10 \quad (20)$$
$$x < x' \quad (15) \qquad x' < x'' \quad (21)$$

Fig. 3. The above linear inequality systems define regions and a transition in a region graph of Example 1

Construction of region graphs can be fully automated since feasibility of linear inequality systems can be checked using linear optimization tools such as a linear programming problem solver.

Next, we propose a method of proving termination by studying region graphs.

4.2 Checking Regions

One of the two termination conditions in Theorem 1 is that any region will be eventually exited once it is entered. For any region without a self-transition, after it is entered, it will be exited after one iteration. For any positive region with a self-transition, the runtime values of variables cannot stay in the region forever. This is because the respective guard value is always decreased during self-transitions and also bounded from below as imposed by the loop condition. On the contrary, negative and still regions with self-transitions introduce the potential of staying in one region forever.

Every time that the self-transition of a negative region is taken, the respective guard value is increased. However, if the guard value has an upper bound within the region, then the self-transition cannot be continuously taken forever. In such a case, we call this region a *bounded* region.

The boundedness of a negative (or positive or still, respectively) region can be checked at the same time when the region is created during region graph construction. For instance, the system of the inequalities (13–15) defines the negative region of Example 1. We can use an optimizer to determine the maximum of guard values under the constraint of the inequalities (13–15) while checking feasibility, by adding the objective function $max : x$. In this example, the negative region is bounded since x has an upper bound 3 within the region.

Having an unbounded negative region, however, does not imply that the runtime values of variables can stay in the region forever. Consider Example 4 whose negative region is unbounded and defined by the inequalities (22–25). Note that the difference of the guard values before and after one iteration is the value of x_2 before the iteration. By Inequality (24) we know that the value of x_2 is always decreased in this region and cannot remain positive forever. This implies eventual leaving of the region. We call such a region a *slowdown* region.

Example 4. This loop has an unbounded negative region.

while $x_1 \geq 0$ **do**

$$\begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

od

$$x_1 \geq 0 \quad (22)$$

$$x_1' = x_1 + x_2 \quad (23)$$

$$x_2' = x_2 - 1 \quad (24)$$

$$x_1' > x_1 \quad (25)$$

$$x_2' \geq x_2 \quad (26)$$

Checking whether a negative region R is a slowdown region can be done by checking the feasibility of a linear inequality system. The checked inequality system describes a subregion of R in which the difference of the respective guard value is increased or unchanged after one iteration. If no such a subregion exists, then R is a slowdown region. For instance, the negative region of Example 4 is a slowdown region because the system of the inequalities (22–26) has no solutions.

We generalize the concept of slowdown regions using an idea similar to the concept of finite difference trees [2]. For an unbounded negative region and an arbitrary natural number n, we build a finite chain $d_0, d_1, ..., d_n$ where the root d_0 is the difference of the respective guard values before and after one loop iteration within the region, and d_1 is the difference of d_0 before and after one iteration within the region, i.e, the "difference of difference", and so forth. When any d_i of the $d_0, d_1, ..., d_n$ is decreased within the region, the region is a slowdown region since d_i dominates the change of d_0, making it impossible to remain positive forever.

4.3 Checking Cycles

Eventual exiting of regions is not enough to show termination. We must make sure that no region is entered an infinite number of times.

In a region graph, if there are no cycles, then no region is entered infinitely often. The region graph in Figure 1 of Example 1 does not have this property. There is a cycle passing the positive region and the negative region. If this cycle can be taken forever, then both regions are entered infinitely often.

We observe that, for Example 1, if the negative region is entered at some point p, then it will be entered at the next time at such a point p' that the value of the guard x at p is greater than the value of x at p'. Because of the loop condition $x \geq 0$, we know that the cycle cannot be taken forever. So, no region is entered infinitely often.

We generalize the above idea by the following definition.

Definition 6. *A cycle is* progressive on a region R *if one of the following is satisfied:*

- *Along the cycle, every time that R is entered, the respective guard value is greater than the guard value at the last time that R is entered. In such a case, we say that the cycle is* upward progressive *if R is bounded.*
- *Along the cycle, every time that R is entered, the respective guard value is smaller than the guard value at the last time that R is entered. In such a case, we say that the cycle is* downward progressive.

It is easy to prove that the following cycles are progressive: (1) a cycle passing the positive region and the still region, and (2) a cycle passing the negative region and the still region if the negative region is bounded.

For other types of cycles, we can check their progressiveness by checking feasibility of a set of linear inequality systems. We have at most six choices: checking whether the cycle is upward (or downward) progressive on the positive (or negative or still) region. For the purpose of illustration, we show how to check downward progressiveness on negative regions. The idea can be easily adapted for other choices and other cases.

Given a G^1P^1 loop as below,

while $\bar{a}\bar{x} \geq b$ **do**
  $\bar{x}' = U\bar{x} + \bar{u}$
od

we assume that there is a cycle passing the positive region and negative region in its constructed region graph. If both regions have no self-transitions, then we can use the linear inequality system in Figure 4 to describe the behavior in which the respective guard value is not decreased every time that the negative region is entered along the cycle. The inequalities (27–29) define that the negative region is entered at a point $\bar{x}$. The inequalities (30–32) define that the positive region is then entered at $\bar{x}'$. The inequalities (33–35) define that the negative region is re-entered at $\bar{x}''$. Inequality (36) imposes that the guard value at $\bar{x}''$ is no smaller than the guard value at $\bar{x}$. If the inequality system has no solutions, then the guard value is always decreased and the cycle is downward progressive on the negative region.

$$\bar{a}\bar{x} \geq b \quad (27) \qquad \bar{a}\bar{x}' \geq b \quad (30) \qquad \bar{a}\bar{x}'' \geq b \quad (33)$$
$$\bar{x}' = U\bar{x} + \bar{u} \quad (28) \qquad \bar{x}'' = U\bar{x}' + \bar{u} \quad (31) \qquad \bar{x}''' = U\bar{x}'' + \bar{u} \quad (34)$$
$$\bar{a}\bar{x}' > \bar{a}\bar{x} \quad (29) \qquad \bar{a}\bar{x}' > \bar{a}\bar{x}'' \quad (32) \qquad \bar{a}\bar{x}'' > \bar{a}\bar{x}''' \quad (35)$$
$$\bar{a}\bar{x} \leq \bar{a}\bar{x}'' \quad (36)$$

Fig. 4. A linear inequality system for checking progressiveness

If one of the regions above has a self-transition, then we do not know precisely at which point this region is exited after being entered. In such a case, we have to overapproximate the exit point. Assume that both regions have a self-transition. The linear inequality system to check downward progressiveness is shown in Figure 5. Note that the negative region is entered at a point $\bar{x}$ as defined by the inequalities (37–39), and it is exited at $\bar{p_{x'}}$ as defined by the inequalities (41–43). An additional inequality (40) guarantees that the successor $\bar{s_x}$ of $\bar{x}$ satisfies the loop condition because loop

iterations cannot continue otherwise. Inequality (44) relates the entry point and the exit point by imposing that the guard value at $\bar{x}$ is no larger than the guard value at $\bar{p_{x'}}$ due to the effect of self-transitions of a negative region. Note that the "equal" part cannot be dropped since it is still possible to leave the negative region immediately without taking the self-transition. The inequalities (45–52) describe the entering and the exiting of the positive region similarly.

$$\bar{a}\bar{x} \geq b \quad (37) \qquad \bar{a}\bar{x}' \geq b \quad (45) \qquad \bar{a}\bar{x}'' \geq b \quad (53)$$

$$\bar{s_x} = U\bar{x} + \bar{u} \quad (38) \qquad \bar{s_{x'}} = U\bar{x}' + \bar{u} \quad (46) \qquad \bar{x}''' = U\bar{x}'' + \bar{u} \quad (54)$$

$$\bar{a}\bar{s_x} > \bar{a}\bar{x} \quad (39) \qquad \bar{a}\bar{x}' > \bar{a}\bar{s_{x'}} \quad (47) \qquad \bar{a}\bar{x}'' > \bar{a}\bar{x}''' \quad (55)$$

$$\bar{a}\bar{s_x} \geq b \quad (40) \qquad \bar{a}\bar{s_{x'}} \geq b \quad (48) \qquad \bar{a}\bar{x} \leq \bar{a}\bar{x}'' \quad (56)$$

$$\bar{a}\bar{p_{x'}} \geq b \quad (41) \qquad \bar{a}\bar{p_{x''}} \geq b \quad (49)$$

$$\bar{x'} = U\bar{p_{x'}} + \bar{u} \quad (42) \qquad \bar{x}'' = U\bar{p_{x''}} + \bar{u} \quad (50)$$

$$\bar{a}\bar{x}' > \bar{a}\bar{p_{x'}} \quad (43) \qquad \bar{a}\bar{p_{x''}} > \bar{a}\bar{x}'' \quad (51)$$

$$\bar{a}\bar{p_{x'}} \geq \bar{a}\bar{x} \quad (44) \qquad \bar{a}\bar{x}' \geq \bar{a}\bar{p_{x''}} \quad (52)$$

Fig. 5. A linear inequality system for checking progressiveness

The progressiveness of each individual cycle is sufficient to show no infinite number of entering of any region only if any two cycles do not pass a same region (see [9] for the proof). Otherwise, this condition is insufficient.

Definition 7. *Given a region graph, if two cycles pass one same region, then we say that these two cycles* interfere *with each other on this region. The region is called an* interfered region *of both cycles.*

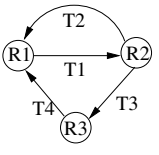

Fig. 6. Two interfering cycles

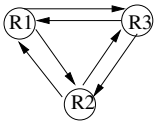

Fig. 7. Three interfering cycles

Consider the region graph in Figure 6 where transitions are distinctly named for convenience. Two cycles $< T_1, T_2 >$ and $< T_1, T_3, T_4 >$ interfere with each other on R_1 and R_2.

We say that a cycle is *completed* when, starting from a region in the cycle, the region is re-entered along the cycle. Furthermore, a cycle C is *uninterruptedly completed* if no other cycle is completed during the completion of C. If a cycle C_1 interferes with some other cycle C_2 on a region R, then a completion of C_1 can be interrupted at R to enter C_2 and resumed from R after C_2 is completed. In such a case, even if C_1 is progressive on some region R', R' may still be entered infinitely often since the respective guard value can be arbitrary when the completion of C_1 is resumed from R after one interruption. However, the following case deserves special attention.

Definition 8. *A region R is a* base region *if the following is satisfied. For any cycle C that passes R, all the cycles that interfere with C also pass R. The set of cycles $\{C \mid C \text{ passes } R\}$ is called an* orbital cycle set.

An orbital cycle set can have more than one base region. For instance, in Figure 6 both R_1 and R_2 are base regions of the orbital set consisting of two cycles. In contrast no region in Figure 7 is a base region.

Orbital sets have an interesting property as follows. Given a base region and its corresponding orbital set, between two successive times that the base region is entered, some cycle in the orbital set is uninterruptedly completed. The proof is sketched here. It is trivial to show that a cycle is completed between two successive times that the base region is entered. Assume that this completion is interrupted at some region R and resumed after some other cycle C is completed. Because C is also in the same orbital set, the base region must be entered while completing C, which contradicts that there is no entering of the base region in-between.

Lemma 1. *Given an orbital cycle set O, any region in any cycle in O is entered only a finite number of times during loop iterations if all the cycles in O are uniformly upward or uniformly downward progressive on some base region* (see [9] for the proof).

4.4 Determining Termination

Based on the previous discussion, we suggest a termination proving algorithm for loops in G^1P^1 as follows. Given a loop,

1. Check the existence of a still region[2]. If it exists, then check whether it has a self-transition. If the self-transition exists, then return "UNKNOWN".
2. Check the existence of a negative region. If neither a negative region nor a still region exists, then return "TERMINATING". In such a case, the loop has linear ranking functions (see Theorem 2).
3. If the negative region exists, then check whether it has a self-transition. If the self-transition exists and the region is unbounded, then check whether it is a slowdown region. If it cannot be determined to be a slowdown region, then return "UNKNOWN".
4. Complete construction of the region graph by constructing the positive region and the rest of the transitions.
5. Check if there are any cycles. If no cycle exists, then return "TERMINATING".
6. Construct all the orbital cycle sets. If there is any interfering cycle that does not belong to any orbital set, then return "UNKNOWN".
7. Check if all the simple cycles are progressive. If there is one simple cycle whose progressiveness cannot be determined, then return "UNKNOWN". With presence of an orbital set, check whether all the cycles in the set are progressive on one base region and agree on the direction of progress (upward or downward). If it is satisfied, then return "TERMINATING".

[2] Remember that the boundedness of a region is checked at the same time that the region is created.

All the steps in this algorithm are arranged in an optimal order so that no unnecessary step is taken. Since all the constructions and checks are performed by automatic translation into linear inequality systems and and automated solving of these systems, the algorithm requires no human intervention.

Complexity. Let N be a parameter to the algorithm as the upper bound on the length of finite difference chains built to check slowdown regions. The number of the linear inequality systems constructed by the algorithm is no more than $16 + N$. Each constructed inequality system has a size linear in the number of variables. If all the variables used in the loop are real variables, then solving of a linear inequality system is polynomial. Otherwise, it is NP-complete. However, in practice constructed inequality systems are usually very small. For the class of loops that have linear ranking functions, the algorithm in [10] needs to construct only one linear inequality system to determine termination, which seems much more efficient than our method. However, we can show that, for any G^1P^1 loop with linear ranking functions, its constructed region graph contains only one positive region as stated in Theorem 2 (see [9] for the proof). So, for any G^1P^1 loop that has linear ranking functions, our algorithm only generates 2 inequality systems to check the existence of a negative region and a still region.

Theorem 2. *A G^1P^1 loop has linear ranking functions if and only if its constructed region graph contains no negative region and no still region.*

Soundness. The algorithm is sound. The proof is sketched in [9]. The basic idea is to show that, if the algorithm returns "TERMINATING" for a loop, then the two termination conditions in Theorem 1 are satisfied by the constructed region graph.

Completeness. The algorithm is incomplete and may return "UNKNOWN". Although termination for G^1P^1 loops in which all the variables are real variables is decidable, the decidability of termination for G^1P^1 loops that have integer variables remains a conjecture [12]. Furthermore, our algorithm can prove termination for a large set of G^1P^1 loops whose iterations change the guard value in one of the patterns as informally illustrated in Figure 8. The horizontal axes represent passage of time and the vertical axes represent change of guard values. The left pattern corresponds to existence of linear ranking functions. The middle one corresponds to existence of slowdown regions. The right one corresponds to progressiveness of cycles.

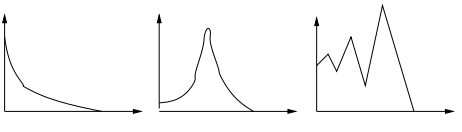

Fig. 8. Patterns in which the guard value changes

In the next two sections, we will generalize the idea of determining termination for G^1P^* and G^*P^1 loops.

5 Proving Termination for G^1P^*

All the ideas in the previous section can be used for G^1P^* loops without too much adaption except that some concepts are generalized with path conditions.

5.1 Constructing Region Graphs

Given a G^1P^* loop as below,

while $\bar{a}\bar{x} \geq b$ **do**
$pc^1 \rightarrow \bar{x}' = U^1\bar{x} + \bar{u}^1$
...
$pc^p \rightarrow \bar{x}' = U^p\bar{x} + \bar{u}^p$
od

the construction of region graphs is similar to the construction for G^1P^1 loops as follows:

- For each i-th path, we create a positive region, a negative region and a still region if their respective defining inequality system has solutions. Let the path condition be $pc^i = \bar{c}_1\bar{x} \geq d_1 \wedge ... \wedge \bar{c}_q\bar{x} \geq d_q$. The system of the linear inequalities (57–60) defines the positive region. The linear inequality systems to define the negative and the still region differ only in the relational operator in Inequality (60) accordingly.
- Transitions are built in exactly the same way as for G^1P^1.

$$\bar{a}\bar{x} \geq b \quad (57) \qquad \bar{x}' = U^j\bar{x} + \bar{u}^j \quad (59)$$

$$\bigwedge_{j=1}^{q} \bar{c}_j\bar{x} \geq d_j \quad (58) \qquad \bar{a}\bar{x} > \bar{a}\bar{x}' \quad (60)$$

5.2 Using Path Conditions

Path conditions can be used to determine eventual exiting of still regions and negative regions with self-transitions.

Consider Example 5. If the first path is taken, the guard value x_1 remains unchanged. However, the path cannot be taken forever. This is because the value of x_2 is always decreased every time that the path is taken and is bounded by 0 as imposed by the path condition.

Example 5. This is a loop with two paths.

while $x_1 \geq 0$ **do**
$x_2 \geq 0 \rightarrow \begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix}$
$x_2 < 0 \rightarrow \begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$
od

To generalize the idea, we define *drag regions* as follows.

Definition 9. *A negative region or a still region is a* drag region *with respect to the respective path condition* $pc = \bar{c}_1\bar{x} \geq d_1 \wedge ... \wedge \bar{c}_q\bar{x} \geq d_q$ *if, for some* $\bar{c}_j\bar{x}$ *in* pc*, the value of* $\bar{c}_j\bar{x}$ *is always decreased within the region.*

Drag regions can be checked by solving a linear inequality systems. The construction is similar to the linear inequality system for checking slowdown. Due to space limitations we do not give the full detail here.

Progressiveness of cycles can also be generalized when taking path conditions into consideration. For a region R with respect to a path condition $pc = \bar{c}_1\bar{x} \geq d_1 \wedge ... \wedge \bar{c}_q\bar{x} \geq d_q$, a cycle is progressive on R if, along the cycle, every time that R is entered, the value of some $\bar{c}_j\bar{x}$ in pc is smaller than the value of $\bar{c}_j\bar{x}$ at the last time that R is entered.

5.3 Determining Termination

The algorithm in Subsection 4.4 is modified for proving termination for G^1P^* loops as follows.

- Positive, negative, and still regions are created for all paths.
- When a still region has a self-transition, instead of returning "UNKNOWN", check whether it is a drag region. If not, return "UNKNOWN".
- For an unbounded negative region, check whether it is a drag region. If not, check whether it is a slowdown region. If not, return "UNKNOWN".
- Progressiveness is checked also with respect to path conditions.

Since the number of cycles is exponential in the number of loop paths, so is the number of linear inequality systems constructed by the modified algorithm. The size of each constructed inequality system is linear both in the number of loop paths and in the number of variables. The algorithm is sound and incomplete. In fact, termination of G^1P^* has been shown undecidable [12].

6 Proving Termination for G^*P^1

The basic idea to prove termination for a G^*P^1 loop (c.f. Example 3) is to check whether termination can be proved by the region graph constructed with respect to some guard in the loop condition. While analyzing the region graph with respect to a chosen guard, we also consider other guards in the loop condition as explained below.

Construction of region graphs. Choosing a guard in the loop condition, the construction of the region graph is similar to the construction for G^1P^1. The linear inequality system to define the positive region contains (1) all the inequalities in the loop condition, (2) variable update equations, and (3) the inequality that expresses the decrease of the chosen guard value. The inequality systems defining the negative region and the still region are constructed similarly.

Generalization of concepts. A negative or a still region is a drag region with respect to some guard that is not chosen for constructing the region graph if the value of the considered guard is decreased within the region. For a region R and some guard g that is not chosen for constructing the region graph, a cycle is progressive on R also if, along

the cycle, every time that R is entered, the value of g is smaller than the value of g at the last time that R is entered.

Determining termination. The algorithm to determine termination for a G^*P^1 loops is as follows. Given a G^*P^1 loop, a guard in the loop condition is chosen nondeterministically. The algorithm in Subsection 4.4 is then used to construct and check the region graph with respect to the chosen guard, with a slight modification which allows for checking drag regions and generalized progressiveness. If termination cannot be determined, then another guard is chosen. This procedure is repeated until termination is proved or all the guards have been checked.

Complexity, soundness and completeness. Let m be the number of guards in the loop condition and N be the parameter as the upper bound on the length of finite difference chains. In the worst case m region graphs are constructed and checked. For each region graph, the number of constructed linear inequality systems is no more than $14+2m+N$. The size of each inequality system is linear in both m and the number of variables. The algorithm is sound and incomplete. In fact it remains a conjecture that termination of G^*P^1 loops that have integer variables is decidable [12]. Furthermore, we conjecture that the algorithm can prove termination for any G^*P^1 loop that has linear ranking functions.

Proving termination for G^*P^*. In our paper we present incomplete approaches to prove termination for G^1P^* and G^*P^1. These two methods are orthogonal and can be easily combined to yield an approach to prove termination for the G^*P^* class.

7 Experimental Results

We implemented our method in a prototype tool named "PONES" (positive-negative-still). Finding a representative sample of realistic software systems that exhibit a large number of non-trivial loops that fall into our categorization is not easy, as it was also observed in [2]. Also, automated extraction of loop code and the resulting loop information has not yet been but will be implemented in the future. For the experiments described here, we manually collected program loops from the source code of Azureus[3] which is a peer-to-peer file sharing software written in Java. The software contains 3567 while- and for-loops. We analyzed the 1636 loops that fall into our categorization. There were only 3 loops in G^1P^* and 4 in G^*P^1. In fact, most of the loops were of the form "`while (i<j) i++`". The prevalent simplicity of the loops encountered corresponds to the desire of programmers to code loops that are easy to comprehend.

PONES failed to prove termination for 14 of the analyzed loops and proved termination within 65 milliseconds for each of all other loops on a Pentium IV 3.20GHz machine with 2GB memory. Manual inspection revealed that the 14 loops that PONES failed on are not terminating on arbitrary initial variable values but do terminate in the context of the Azureus software system which limits the range of the initial variable values.

[3] Available from sourceforge.net.

We propose that our analysis method can be improved by incorporating value analysis [5] to generate linear inequalities over variables as loop invariants. These inequalities are then used to shrink some regions in the constructed region graph in order to exclude those points that will never be reached during loop iterations.

As further future work we propose to generalize the concept of program loops as explicitly constructed by the *while* or *for* constructs to control flow cycles resulting from mutual and recursive function calls. These control flow cycles are usually more complex but we expect that our analysis can handle them nonetheless.

We cannot give a direct comparison with other termination proof methods because other works use different extraction and abstraction techniques than our method to collect loops from programs. It should also be noted that our method can be considered as being complementary to linear ranking function based approaches.

8 Conclusion

We propose a new termination proof method based on constructing and analyzing region graphs. The method is incomplete and efficient in practice. It can prove termination for some loops that have no linear ranking functions. We implemented the method in the PONES tool and conducted several experiments with Java programs. Future work includes: (1) the adaption of the method to approximate loop iteration times; (2) refining the method by discovering other useful information from loops; (3) analysis of loops with more general loop conditions, i.e., with the presence of disjunction; (4) abstraction of nested loops and control flow cycles into G^*P^* loops.

Acknowledgment. We thank Alin Stefanescu for his beneficial and helpful comments on our work and Daniel Butnaru for his assistance in programming the PONES prototype. We also thank anonymous reviewers for their useful comments and suggestions.

References

1. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination analysis of integer linear loops. In *Concurrency Theory, 16th International Conference, CONCUR 2005, Proceedings*, volume 3653 of *Lecture Notes in Computer Science*, pages 488–502. Springer, 2005.
2. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination of polynomial programs. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 113–129. Springer, 2005.
3. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In *Static Analysis, 12th International Symposium, SAS 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2005.
4. Patrick Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2005.
5. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th Symposium on Principles of Programming Languages (POPL 1978), Proceedings*, pages 84–97, 1978.

6. Stefan Leue, Richard Mayr, and Wei Wei. A scalable incomplete boundedness test for UML RT models. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 327–341. Springer, 2004.
7. Stefan Leue, Richard Mayr, and Wei Wei. A scalable incomplete test for buffer overflow of Promela models. In *Model Checking Software, 11th International SPIN Workshop, Proceedings*, volume 2989 of *Lecture Notes in Computer Science*, pages 216–233. Springer, 2004.
8. Stefan Leue and Wei Wei. Counterexample-based refinement for a boundedness test for CFSM languages. In *Model Checking Software, 12th International SPIN Workshop, Proceedings*, volume 3639 of *Lecture Notes in Computer Science*, pages 58–74. Springer, 2005.
9. Stefan Leue and Wei Wei. A region graph based approach to termination proofs. Technical report soft-06-01, University of Konstantz, 2006.
10. Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer, 2004.
11. Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), Proceedings*, pages 32–41. IEEE Computer Society, 2004.
12. Ashish Tiwari. Termination of linear programs. In *Computer Aided Verification, 16th International Conference, CAV 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 70–82. Springer, 2004.