Proving Non-Termination

Ashutosh K. Gupta
MPI-SWS
agupta@mpi-sws.mpg.de

Thomas A. Henzinger
EPFL
tah@epfl.ch

Rupak Majumdar
UC Los Angeles
rupak@cs.ucla.edu

Andrey Rybalchenko
MPI-SWS
rybal@mpi-sws.mpg.de

Ru-Gang Xu
UC Los Angeles
rxu@cs.ucla.edu

Abstract

The search for proof and the search for counterexamples (bugs) are complementary activities that need to be pursued concurrently in order to maximize the practical success rate of verification tools. While this is well-understood in safety verification, the current focus of liveness verification has been almost exclusively on the search for termination proofs. A counterexample to termination is an infinite program execution. In this paper, we propose a method to search for such counterexamples. The search proceeds in two phases. We first dynamically enumerate lasso-shaped candidate paths for counterexamples, and then statically prove their feasibility. We illustrate the utility of our nontermination prover, called TNT, on several nontrivial examples, some of which require bit-level reasoning about integer representations.

Categories and Subject Descriptors D.2.4 [*Software*]: Software Engineering—Program Verification; D.4.5 [*Software*]: Operating Systems—Reliability

General Terms Reliability, Verification

Keywords Program verification, Model checking, Testing, Non-termination, Recurrent Sets

1. Introduction

It has become fairly undisputed that the main value of verification tools lies in the discovery of feasible bugs, not in the proof of program correctness (Godefroid 2005). Evidence of a bug presented to a programmer is often a more convincing demonstration of the utility of a tool than a proof of correctness, which even in the best case is relative to certain semantic assumptions that may not hold, and relative to partial specifications. Moreover, during software development, a tool can repeatedly demonstrate its utility by reporting bugs even on incomplete programs, while a proof can be provided only at the end (if at all). Thus, it is important that both the precision and efficiency of verification tools be geared towards finding bugs, not proofs.

POPL'08, January 7–12, 2008, San Francisco, California, USA.

Consequently, in safety verification there has been much focus on checking whether a counterexample is spurious or genuine, and on classifying which counterexamples are most likely to be genuine bugs (Kremenek and Engler 2003). A counterexample to a safety property is a finite program executions that leads to an error. The discovery and feasibility analysis of such executions lies at the heart of many recent tools (Ball and Rajamani 2002; Henzinger et al. 2002; Godefroid et al. 2005; Gulavani et al. 2006).

In contrast, in liveness verification the main focus to date has been on finding proofs, i.e., evidence of program termination (Colón and Sipma 2002; Bradley et al. 2005; Cousot 2005; Cook et al. 2006). A liveness violation is a non-terminating program execution. Unfortunately, since techniques to prove termination are incomplete, failure to prove termination does not immediately indicate the existence of a non-terminating execution. Therefore, a failed termination proof produces a possible counterexample, which may or may not be genuine. Currently, these counterexamples must be inspected manually to determine if they are indeed bugs. This is unsatisfactory: experience with static program analysis (for safety properties) indicates that a major obstacle to practicality of static analysis tools is the presence of *false positives*—warnings that do not correspond to real errors. Furthermore, classical objects of study in temporal verification are auxiliary assertions geared towards proving temporal properties, e.g., invariants and ranking functions. Auxiliary assertions that demonstrate existence of (non-terminating) executions have not received adequate attention. Therefore, dual to the search for termination proofs, we must also develop tools that demonstrate feasible *non-terminating* executions.

We present a method that searches precisely and efficiently for non-terminating program executions. In particular, our algorithm looks for feasible "lassos". A *lasso* consists of a finite program path called *stem* that is followed by a finite program path called *loop*. The loop must form a syntactic cycle in the control-flow or call graph of the program The lasso is feasible if an execution of the stem can be followed by infinitely many executions of the loop. In general, the method is incomplete, as not all non-terminating program executions are induced by lassos (the infinite behavior may be non-periodic). However, by limiting our attention to lassos, we can concentrate on finding the most common non-termination bugs quickly.

Our algorithm proceeds in two phases. The first phase generates candidate lassos, and the second checks each lasso for possible non-termination.

Our tool generates candidate lassos exhaustively by systematically executing all paths of the concrete program until some control location is re-visited. Backtracking is controlled by collecting symbolic constraints during concrete program execution. This method has two advantages. First, the stem followed by one execution of the loop is guaranteed to be feasible. Second, every such lasso is guaranteed to be generated. Alternatively, a termination prover could be used to supply a candidate lasso whenever its termination proof fails.

The second phase proves the feasibility of a given lasso. A lasso is feasible if and only if there exists a *recurrent* set of states, i.e., a set of states that is visited infinitely often along the infinite path that results from unrolling the lasso. We formulate and solve the existence of a recurrent set as a template-based constraint satisfaction problem. The constraint satisfaction problem for recurrent sets turns out to be equivalent to constraint systems for invariant generation (Colón et al. 2003; Sankaranarayanan et al. 2004). Therefore, techniques for constraint-based invariant generation apply directly to the problem of checking non-termination. The precision of our analysis can be adjusted by choosing an appropriate constraint theory. We can either apply a more precise bit-level analysis that takes into account many of the machine-dependent characteristics of programs such as overflow, or a less precise integer-level (arithmetic) analysis that is geared towards more algorithmic reasons for non-termination.

We have implemented a tool to test for non-termination of C programs, called TNT. Since our emphasis is on *feasible* bugs, the lasso generation algorithm in TNT is based on dynamic execution of the program, with symbolic constraints gathered on the side for systematic exploration of the state space (Godefroid et al. 2005; Sen et al. 2005). The feasibility check of a lasso is symbolic, and can utilize precise information about data structures and address resolution along the lasso provided by the dynamic exploration. While dynamic exploration has its own drawbacks (path explosion), we believe the interaction of efficient dynamic and static analysis in TNT provides an interesting design point in the search for liveness bugs.

We provide several examples where our tool succeeds to prove non-termination. The bit-level analysis can prove the non-termination of a standard implementation of binary search that is caused by arithmetic overflows. The integer constraint analysis shows a non-terminating path in an implementation of the Mondriaan memory protection scheme (Witchel et al. 2002). Dynamic exploration of the program state space was especially helpful to find the non-termination bug in Mondriaan, as the non-terminating execution required precise, bit-accurate reasoning about a three-level permissions table data structure.

Our non-termination checker can also be used whenever at runtime a program does not terminate within a given time bound. The non-termination checker can capture the state and the current execution path and try to prove online if the current execution is doomed to run forever. Therefore, a non-termination checker is the liveness analogue to runtime monitors for safety (Sen et al. 2004; d'Amorim and Rosu 2005; Pnueli et al. 2006).

A similar non-termination analysis has been independently developed by Velroyen (2007).

2. Example

Non-Terminating Binary Search. Joshua Bloch recently pointed out that many existing implementations of binary search can produce `ArrayOutOfBounds` exceptions because they ignore arithmetic overflows (Bloch 2006). For example, the version that used to be in Java's standard library is affected by this bug. Specifically, the statement

```
mid = (lo + hi)/2;
```

that computes the midpoint of the range can overflow for large values of `lo` and `hi`, thus, producing a negative value that is used as an array index.

```
 1: int bsearch(int a[], int k,
                unsigned int lo,
                unsigned int hi) {
 2:     unsigned int mid;
 3:     while (lo < hi) {
 4:             mid = (lo + hi)/2;
 5:             if (a[mid] < k) {
 6:                     lo = mid + 1;
 7:             } else if (a[mid] > k) {
 8:                     hi = mid - 1;
 9:             } else {
10:                     return mid;
11:             }
12:     }
13:     return -1;
14: }
```

Figure 1. Broken binary search implementation.

The version of binary search shown in Figure 1 is similar to the original, except that we have replaced the signed integers of the original version with *unsigned* integers. (This is similar to the signature of binary search in C's stdlib which uses the unsigned type `size_t` for the indices.) This time, while the array index will remain within bounds, the arithmetic overflow can lead to an infinite loop.

We illustrate the non-termination by considering an execution of a path through the `while` loop that follows the first branch of the conditional statement visiting lines 2–6. The corresponding sequence of statements consists of

```
{lo<hi}; mid=(lo+hi)/2; {a[mid]<k}; lo=mid+1;
```

Here, we used curly braces `{ }` to represent evaluation of conditional expressions. We maliciously choose the following initial values

$$\texttt{lo} = 1, \qquad \texttt{hi} = MAXINT, \qquad \texttt{a[0]} < \texttt{k},$$

where $MAXINT$ is the maximum value of an unsigned integer. The first statement on the path, which evaluates the loop condition, returns true for the chosen inputs. Then, we update the value for the variable `mid`. Due arithmetic overflow, we obtain $\texttt{mid} = 0$. Thus, we come back to the head of the loop with $\texttt{lo} = 1$ and $\texttt{hi} = MAXINT$, which is our starting program state. Thus, if an adversary can choose the input values to the program then she can force the program to enter an infinite loop.

We observe that a sufficient condition for such non-terminating loops is the existence of a *lasso-shaped* execution. That is a feasible program execution that reaches the head of the loop (the conditional $\texttt{lo} < \texttt{hi}$ on line 3) with some state s, then executes the body of the loop (lines 4-11) and returns to the same state s (modulo the live variables, i.e., variables whose values will be read in the future). In this case, we can unroll the execution of the loop arbitrarily many times by starting at s, executing the loop, and returning to s.

We can search for such paths by exploring the program *symbolically*. In symbolic execution, the program is executed on symbolic instead of, or in addition to, concrete inputs. During the execution, we gather constraints on symbolic inputs. Any satisfying assignment to the symbolic constraints is guaranteed to execute the program along the currently executed path. Let X be the set of program variables. Then, for a program path π, we write $\rho_\pi(X, X')$ to denote the symbolic constraint generated by symbolic execution when traversing π. It relates the values X' of the program variables at the end of the path to their original values X.

We assume that symbolic execution completes a loop in the control flow graph of the program, say along the path $\ell_0 \xrightarrow{stem} \ell \xrightarrow{loop} \ell$. Let ℓ_0 be the entry point of the program, ℓ be the head of a loop, and $stem$ and $loop$ be respectively the executed path up to the loop and along the loop body. To detect an infinite loop, we can check for the existence of a satisfying assignment to a query:

$$\rho_{stem}(X_0, X) \wedge \rho_{loop}(X, X') \wedge X = X' \qquad (1)$$

While we write $X = X'$ to enforce the search for the same state, we only require equality of all *live* variables. For the program `bsearch` and the path that takes the first branch of the conditional, the generated query is

$true \wedge$	constraints up to line 3
$\texttt{lo} < \texttt{hi} \wedge \texttt{mid} = (\texttt{lo} + \texttt{hi})/2 \wedge$	
$\texttt{a[mid]} < \texttt{k} \wedge \texttt{lo}' = \texttt{mid} + 1 \wedge$	constraints for the loop
$\texttt{lo}' = \texttt{lo} \wedge \texttt{hi}' = \texttt{hi}.$	set live variables equal

This query can be resolved by a decision procedure that treats variables and arithmetic in a bit-accurate way (Cook et al. 2005; Ganesh and Dill 2007; Xie and Aiken 2005). A satisfying assignment to the variables will provide an input that causes non-termination.

Unbounded Ranges and Recurrent Sets. Unfortunately, modeling the bit-accurate semantics leads to too many cases of non-termination, as a significant portion of C code is written without considering overflow or with implicit assumptions about the size of integer inputs to the program. For example, under the bit-accurate semantics, the following loop does not terminate:

```
n := input();
for (i = 0; i <= n; i++) body;
```

An offending input is when `n` is $MAXINT$, the maximum representable number in the machine. When `i` is incremented after it reaches `n`, the value rolls back to either 0 or $MININT$, depending on whether `i` is declared as unsigned or signed. Then, the loop condition `i <= n` continues to hold. Such loops occur in library functions that sort inputs. However, there is always an implicit assumption that the arrays we shall sort in practice are not that large (or the memory allocator will fail to allocate the array even before the sorting routine is called).

These non-terminating programs represent an important class of bugs, especially for security and denial-of-service related vulnerabilities where an attacker can exploit the overflow. Equally important, we also want to find non-terminating executions in the still important *abstracted* semantics, where numbers are modeled as unbounded, mathematical integers. These non-terminating executions can point out *algorithmic* problems in code.

In the abstracted semantics, the condition in Equation (1) is sufficient for non-termination, but clearly not necessary. The *same* state may not be reached in any iteration of the loop, but the execution can nevertheless be non-terminating. For example, consider the loop:

```
loop() {
  i = input(); y = input();
  if (y!=0)
    while (i >= 0) { i = i - y; }
}
```

The loop is non-terminating if the initial value of i is non-negative, and `y` is negative. In this case, every execution of the loop body produces a new state where the new value of `i` is the old value of `i` minus the value of `y`. Thus, there is no unrolling of the loop body for which we can satisfy the requirement from Equation (1). Instead, we generalize the test for non-termination and look for a *recurrent set*.

A recurrent set R is a set of states at the head of the loop that satisfies the following properties:

1. R entails the loop condition, say p,
2. some reachable state s satisfies R, and
3. for any state s satisfying R, some successor of s after executing the loop body is again in R.

Clearly, if there is such an R, then we can find a non-terminating execution of the program. In case of the loop statement in the function `loop`, the set $i \geq 0 \wedge y \leq 0$ forms a recurrent set. The first two conditions are easily checked. For condition (3), we notice that

$$\forall \texttt{i}, \texttt{y}\ \exists \texttt{i}', \texttt{y}'.\ (\texttt{i} \geq 0 \wedge \texttt{y} \leq 0) \rightarrow \\ (\texttt{i}' = \texttt{i} - \texttt{y} \wedge \texttt{y}' = \texttt{y}) \wedge (\texttt{i}' \geq 0 \wedge \texttt{y}' \leq 0).$$

Thus, we can conclude that there is a non-terminating execution of the loop. We have reduced the search for non-terminating executions to the search for recurrent sets.

Reduction to Constraint Solving. We shall use a template-based approach for computing recurrent sets. The approach reduces the search for recurrent sets to constraint solving over a suitable domain. We restrict our attention to the important class of *linear* programs, where the transition relation is a linear function of the variables (and their updated values). Consider the lasso-shaped execution, where the stem consists of the first two instructions and the lasso consists of the remaining three instructions:

```
i=input();  y=input();  {y!=0};  {i>=0};  i=i-y;
```

Let us assume that the recurrent set is defined by a parametric linear inequality together with the loop guard. That is, for the recurrent set we assume the template

$$\texttt{i} \geq 0 \wedge a\texttt{i} + b\texttt{y} \geq c,$$

where a, b, and c are unknown parameters. They define a recurrent set if the condition (3) is satisfied:

$$\forall \texttt{i}, \texttt{y}\ \exists \texttt{i}', \texttt{y}'.\ \texttt{i} \geq 0 \wedge a\texttt{i} + b\texttt{y} \geq c \rightarrow \\ \texttt{i} \geq 0 \wedge \texttt{i}' = \texttt{i} - \texttt{y} \wedge \texttt{y}' = \texttt{y} \wedge \\ \texttt{i}' \geq 0 \wedge a\texttt{i}' + b\texttt{y}' \geq c.$$

These constraints are similar to constraints generated in template-based invariant generation (Colón et al. 2003). Existing non-linear constraint solving techniques can be used to solve them. We provide the formal details in Section 5. After solving these constraints for a, b, and c, we get a recurrent set

$$\texttt{i} \geq 0 \wedge \texttt{y} \leq 0.$$

This set is reachable for the initial condition $\texttt{i} \geq 0 \wedge \texttt{y} < 0$. Any solution to these constraints demonstrates an input that causes non-termination.

Summary. Our non-termination checker TNT consists two components. The first component performs a reachability computation on the state space of the program to enumerate all possible lasso shaped executions. The second component attempts to infer a recurrent set for each lasso-shaped execution that is discovered by the first component. In our implementation, the enumeration of lassos is performed using directed test generation. Existence of a recurrent set implies that the current lasso induces a non-terminating execution. If no recurrent set is found (either because the loop terminates or because the template is too weak), the generation of lassos continues.

3. Preliminary Definitions

We develop our algorithm for an abstract imperative programming language. For ease of exposition, this language ignore features such as references, dynamic memory allocation, and function calls.

A *program* $P = (X, \mathcal{L}, \mathcal{L}^{\circlearrowleft}, \ell_0, \mathcal{T})$ consists of a set X of variables, a set $\mathcal{L}$ of control locations, a set of cutpoint locations $\mathcal{L}^{\circlearrowleft} \subseteq \mathcal{L}$, an initial location $\ell_0 \in \mathcal{L}$, and a set $\mathcal{T}$ of transitions. Each transition $\tau \in \mathcal{T}$ is a tuple (ℓ, ρ, ℓ'), where $\ell, \ell' \in \mathcal{L}$ are control locations, and ρ is a *transition relation* over the variables from $X \cup X'$. The variables from X denote values at control location ℓ, and the variables from X' denote the (updated) values of the variables from X at control location ℓ'. The sets of locations and transitions naturally define a directed graph, called the *control-flow graph* (CFG) of the program (Aho et al. 1986). Though, unlike Aho et al. (1986) we put the transitions at the edges of the graph. We assume that for any infinite path through the control-flow graph, there exists at least one control location in $\mathcal{L}^{\circlearrowleft}$ that is visited by the path infinitely many times. In our examples, we shall write programs using a C-like syntax, but these can be easily processed into (abstract) programs.

A *state* of the program P is a valuation of the variables from X. The set of all states is denoted $\mathsf{V}.X$. We shall represent sets of states using formulas over X. We write $s \models \psi$ if the state $s \in \mathsf{V}.X$ satisfies the formula ψ. A formula ψ over X represents the set $\{s \in \mathsf{V}.X \mid s \models \psi\}$. For a formula ρ over $X \cup X'$ and a valuation $(s, s') \in \mathsf{V}.X \times \mathsf{V}.X'$, we write $(s, s') \models \rho$ if the valuation satisfies the constraint ρ. An *execution* of the program P is a sequence $\langle \ell_0, s_0\rangle, \langle \ell_1, s_1\rangle, \dots, \langle \ell_k, s_k\rangle \in (\mathcal{L} \times \mathsf{V}.X)^*$. We require that ℓ_0 is the initial location, and for each $i \in \{0, \dots, k-1\}$ there is a transition $(\ell_i, \rho, \ell_{i+1}) \in \mathcal{T}$ such that $(s_i, s_{i+1}) \models \rho$. A *path* of the program P is a sequence $\pi = (\ell_0, \rho_0, \ell_1), (\ell_1, \rho_1, \ell_2), \dots, (\ell_{k-1}, \rho_{k-1}, \ell_k)$ of transitions, where ℓ_0 is the initial location and the sequence of transitions form a path in the CFG. The path π is *feasible* if there is an execution $\langle \ell_0, s_0\rangle, \dots, \langle \ell_k, s_k\rangle$ such that for each $i \in \{0, \dots, k-1\}$, we have $(s_i, s_{i+1}) \models \rho_i$. A location ℓ is *reachable* if some feasible path ends in ℓ. A state s is reachable at location ℓ if $\langle \ell, s\rangle$ appears in some computation. Feasibility is extended in a natural way to paths that are *infinite*. That is, an infinite path $\pi = (\ell_0, \rho_0, \ell_1), \dots, (\ell_{k-1}, \rho_{k-1}, \ell_k), \dots$ is feasible if there is an infinite execution sequence $\langle \ell_0, s_0\rangle, \dots, \langle \ell_k, s_k\rangle \dots$ for all $i \in \mathbb{N}$, we have $(s_i, s_{i+1}) \models \rho_i$. For a finite path $(\ell_0, \rho_0, \ell_1) \dots (\ell_{k-1}, \rho_{k-1}, \ell_k)$, we shall write a "compound transition" (ℓ_0, ρ, ℓ_k) where $\rho = \rho_0 \circ \dots \circ \rho_{k-1}$, and $\circ$ is the relational composition operator defined by

$$(\phi \circ \psi)(X, X') = \exists X''.\ \phi(X, X'') \wedge \psi(X'', X').$$

A *lasso* at a cutpoint location $\ell \in \mathcal{L}^{\circlearrowleft}$ consists of two sequences of transitions, which are referred to as *stem* and *loop*:

$$\mathit{lasso} \;=\; \ell_0 \xrightarrow{\mathit{stem}} \ell \xrightarrow{\mathit{loop}} \ell$$

The *stem* is a path from the initial location ℓ_0 to the location ℓ. The *loop* consists of a path that starts and ends at the cutpoint location ℓ by following a cyclic path through the control-flow graph. A lasso *induces* an infinite execution if the infinite path $\mathit{stem}(\mathit{loop})^\omega$ obtained by traversing the stem and then unrolling the loop infinitely many times is feasible.

4. Generating Lassos

Our algorithm for detecting non-terminating executions has two parts: one part generates lassos in the control-flow graph, and the second part checks if each generated lasso induces an infinite execution.

```
      input
        P: program
      vars
        s, τ, ℓ, ℓ' : program state, transition, and control locations
        stem, loop : sequences of transitions
      begin
1       ℓ := ℓ0
2       CHOOSE s ∈ V.X
3       stem := ε
4       loop := ε
5       try
6         repeat                                  (∗ selecting stem ∗)
7           τ, ⟨ℓ, s⟩ := CHOOSENEXT(ℓ, s)
8           stem := stem • τ
9         until ℓ ∈ L^↺ and CHOOSE {true, false}
10        ℓ' := ℓ                         (∗ fixing cutpoint location ∗)
11        repeat                                  (∗ selecting loop ∗)
12          τ, ⟨ℓ, s⟩ := CHOOSENEXT(ℓ, s)
13          loop := loop • τ
14        until ℓ = ℓ' and CHOOSE {true, false}
15        if NONTERMLASSO(stem, loop) then
16          s := initial state witnessing non-termination
17          return "non-terminating execution starts from s"
18        else
19          raise CHOICEFAILURE
20      catch CHOICEFAILURE do
21        backtrack
22      catch BACKTRACKINGEXHAUSTED do
23        return "program terminates"
      end.
```

Figure 2. Algorithm NONTERM for testing non-termination. The operator $\bullet$ adds a transition at the end of a given sequence. The functions CHOOSE and CHOOSENEXT are backtrackable.

We now describe the algorithm NONTERM that searches for lassos in the control-flow graph of the program. It is shown in Figure 2.

Non-Deterministic Search. In NONTERM, we use the variables ℓ, s, and τ to store the current location, program state, and transition that leads to the current location. The search for lassos is divided into two phases, which follows the lasso structure. During the first phase, we construct the stem part, see lines 6–9 in Figure 2. It is chosen nondeterministically and also fixes the cutpoint location ℓ'. The second phase, see lines 11–14, nondeterministically chooses a loop at ℓ'.

Our high-level exposition combines nondeterministic choice and backtracking to achieve exhaustive enumeration of all possible lassos. We use a nondeterministic choice operator CHOOSE, which selects an arbitrary element from a given set. We assume that CHOOSE raises the CHOICEFAILURE exception when applied on the empty set. The function CHOOSENEXT determines how a stem/loop is extended. When applied on a program state s at location ℓ, it returns a program transition that starts from ℓ, a successor location and state under this transition. The definition of CHOOSENEXT is shown in Figure 3. The CHOOSE and CHOOSENEXT operations can be effectively implemented using symbolic execution and depth first search.

We assume that if no more backtracking is possible, i.e., if all possible choices have been explored by the algorithm, then the exception BACKTRACKINGEXHAUSTED is raised. In this case, we report that the program is terminating, see line 22.

Check. A discovered lasso is analyzed w.r.t. the non-termination. We use symbolic constraint-based methods for this analysis, which

```
input
  ℓ : control location
  s : program state
vars
  S : set of state-transition pairs
begin
  S := {(τ, ⟨ℓ′, s′⟩) | τ = (ℓ, ρ, ℓ′) ∈ T and (s, s′) ⊨ ρ}
  return CHOOSE S
end.
```

Figure 3. Auxiliary function CHOOSENEXT for the nondeterministic selection of an outgoing transition, a successor location and state. The function CHOOSE raises the CHOICEFAILURE exception when applied on the empty set. We implicitly assume the fixed program P which determines the possible states s' and transitions τ.

is motivated by the following considerations. Symbolic methods can effectively explore all possible executions that follow the stem and then unroll the loop part, despite their large or unbounded number. Even if a particular execution that is analyzed by the algorithm is terminating, there may be a similar one—an execution that traverses the same stem and loop, but has a different valuation of the program variables—that is non-terminating.

We apply a non-termination checker by calling the function NONTERMLASSO, see line 15. If the proof of non-termination succeeds, then we assume that it yields an initial state of a non-terminating execution. This state is reported as evidence of non-termination, and the algorithm NONTERM succeeds. In general, the transition relation of the lasso may be nondeterministic, e.g., contain some input statements. In this case, we also require that NONTERMLASSO computes a sequence of valuations for the inputs that are read by the program during the lasso traversal. In Section 5, we describe algorithms for proving non-termination that provide a range of precision/efficiency trade-offs.

A failed non-termination check guides the algorithm into the search for a different lasso. We rely on backtracking to explore the alternative choices of the calls to CHOOSE and CHOOSENEXT.

Correctness. The correctness of the algorithm NONTERM relies on two components: the exhaustiveness of the search process for lassos, and the soundness of NONTERMLASSO, i.e., it only gives the positive result if there exists an infinite execution induced by the lasso.

THEOREM 1 (Correctness). *If the algorithm* NONTERM *terminates on an input program P, and returns "non-terminating execution starts from s" then P has an infinite execution starting from state s. If* NONTERM *terminates on an input program P, and returns "program terminates" then the execution of P terminates starting from any initial state.*

PROOF[Sketch] The first case immediately follows from the correctness of NONTERMLASSO. For the second case, we observe that if the BACKTRACKINGEXHAUSTED exception is raised then NONTERM enumerated all possible lassos. From the correctness of NONTERMLASSO, we conclude program termination. □

Since the problem of detecting non-termination is undecidable, algorithm NONTERM may not terminate on all programs. This could happen, for example, for programs with infinite, aperiodic executions.

5. Proving Feasibility of Lassos

In this section, we propose algorithms for proving *non-termination* of lassos, which we use to implement the function NONTERMLASSO in our algorithm NONTERM for testing non-termination.

(Non-)Well-Foundedness. First, we describe conditions when a relation can induce infinite sequences. Subsequently, we extend it to deal with lassos.

A binary relation $\rho(X, X')$ over states is *not well-founded* at a state s if there exists an infinite sequence $s_1, s_2, \dots$ such that $s_1 = s$ and for each $i \geq 1$ we have $(s_i, s_{i+1}) \models \rho$. The relation is *well-founded* at s if there is no such infinite sequence. We are interested in finding the initial states of infinite sequences induced by relations that are not well-founded.

Let $lasso = \ell_0 \xrightarrow{stem} \ell \xrightarrow{loop} \ell$ be a lasso where ρ_{stem} and ρ_{loop} are transition relations of the stem and loop, respectively. The lasso induces an infinite execution if the transition relation of the loop part is not well-founded at a state that is reachable by traversing the stem. Formally, the lasso induces an infinite execution if the relation

$$\exists X.\ \rho_{stem}(X, X') \wedge \rho_{loop}(X', X'')$$

is not well-founded.

5.1 Recurrent Sets

We now provide a condition for checking that a relation is not well-founded. We formulate our condition in terms of *recurrent sets*. Let ρ be a relation. A state s' is called a ρ-successor of a state s if $(s, s') \models \rho$. A set of states $\mathcal{G}(X)$ is *recurrent* for ρ if for each state $s \models \mathcal{G}(X)$, there exists a ρ-successor state s' such that $s' \models \mathcal{G}(X)$.

PROPOSITION 1 (Recurrent sets and non-well-foundedness). *A relation $\rho(X, X')$ is not well-founded if and only if there exists a non-empty recurrent set of states, i.e., if for some $\mathcal{G}(X)$, we have:*

$$\exists X.\ \mathcal{G}(X), \tag{2}$$

$$\forall X\ \exists X'.\ \mathcal{G}(X) \rightarrow \rho(X, X') \wedge \mathcal{G}(X'). \tag{3}$$

PROOF If a non-empty recurrent set exists, then we generate an infinite sequence by picking an element satisfying $\mathcal{G}(X)$ (this is possible by Condition (2)), and then constructing an infinite sequence by iteratively applying Condition (3). If a relation is not well-founded, let $s_1, s_2, \dots$ be an infinite sequence induced by the relation. We define $\mathcal{G}(X)$ to be the set $\{s_1, s_2, \dots\}$. □

We illustrate recurrent sets by example. Consider the relation ρ over the variables x, y and x', y' such that

$$x \geq 0 \wedge x' = x + y \wedge y' = y + 1.$$

We observe that for the construction of an infinite sequence induced by ρ it is necessary that the value of variable x is always positive. One possibility to ensure this condition is to start with a positive value of x and increase it at each step. Hence, we obtain a recurrent set

$$\mathcal{G}_1(x, y) \ = \ x \geq 0 \wedge y \geq 0.$$

An alternative recurrent set admits infinite sequences in which the value of x may decrease initially, but never decreases below zero (here, $|y|$ denotes the absolute value of y):

$$\mathcal{G}_2(x, y) \ = \ x \geq 0 \wedge x \geq \frac{1}{2}|y|(|y| + 1).$$

An example infinite sequence for the recurrent set $\mathcal{G}_2(x, y)$ is

$$\langle 6, -3 \rangle,\ \langle 3, -2 \rangle,\ \langle 1, -1 \rangle,\ \langle 0, 0 \rangle,\ \langle 0, 1 \rangle,\ \langle 1, 2 \rangle,\ \langle 3, 3 \rangle,\ \dots .$$

When analyzing the non-termination of a lasso, we need to construct a recurrent set for the loop of the lasso that is moreover reachable by traversing the stem.

PROPOSITION 2 (Recurrent set for lasso). *A lasso $\ell_0 \xrightarrow{stem} \ell \xrightarrow{loop} \ell$ induces an infinite execution if there exists a recurrent set $\mathcal{G}(X')$ for the relation $\rho_{loop}(X', X'')$ such that*

$$\exists X\ \exists X'.\ \rho_{stem}(X, X') \wedge \mathcal{G}(X'). \tag{4}$$

5.2 From Recurrent Sets to Constraint Systems

We now describe two symbolic analyses to construct recurrent sets satisfying the conditions of Proposition 2 for a given lasso. The first, *bitwise* analysis assumes that the state space is finite, and, without loss of generality, encoded using Boolean variables. The second, *linear arithmetic* analysis assumes that every program transition along the lasso can be represented as a (rational) linear constraint over the program variables.

The bitwise analysis enables the precise treatment of low-level features of programming languages, e.g., bit-wise operations and arithmetic modulo fixed widths. The linear arithmetic analysis is a useful abstraction for programs when bit-level precision is not required. In either case, we show how we can reduce the search for recurrent sets to automatic constraint solving.

Bitwise Analysis. For the bitwise analysis, we assume that program variables range over Booleans, and that the transition relation of the lasso is given by a Boolean formula over propositional variables. Since the state space is finite, a lasso induces an infinite execution if and only if some state is repeated infinitely many times in the course of the execution. Therefore, we can restrict the search to *singleton* recurrent sets (i.e., recurrent sets which have exactly one state). Given a lasso $\ell_0 \xrightarrow{stem} \ell \xrightarrow{loop} \ell$, we therefore look for a state s that is reached at ℓ by executing the transition $stem$, and after executing the transition *loop* returns back to itself. We encode this condition by the constraint

$$\exists X, X', X''.\ \rho_{stem}(X, X') \wedge \rho_{loop}(X', X'') \wedge X'' = X', \tag{5}$$

The function NONTERMLASSO returns a positive result if this constraint is satisfiable, and can be implemented using Boolean satisfiability solving. The valuation of X is an initial state that witnesses non-termination. This is similar to bounded model checking procedure for liveness (Clarke et al. 2001). The constraints can be resolved by a bit-precise decision procedure such as Cogent (Cook et al. 2005) or STP (Ganesh and Dill 2007), that eventually reduces the checks to Boolean satisfiability.

Notice that the constraint in Equation (5) may not be satisfied for a syntactic loop in the program, but only for some unrolling of this loop. Consider the program

```
while (x == y) { x = !x; y = !y; }
```

which has an infinite loop if x and y are initially equal. For this program, the constraint

$$(\mathtt{x} = \mathtt{y}) \wedge (\mathtt{x} = \neg\mathtt{x} \wedge \mathtt{y} = \neg\mathtt{y})$$

obtained from Equation (5) is unsatisfiable. However, since NONTERM exhaustively generates all lassos, we shall eventually consider a lasso where the loop is unrolled once

```
{x==y}; x = !x; y = !y; {x==y}; x = !x; y = !y;
```

We observe that a singleton recurrent set exists for this lasso.

Linear Arithmetic Analysis. The linear arithmetic analysis assumes that the program transitions are representable using conjunctions of linear inequalities over the program variables.

Our algorithm follows a constraint-based approach for the synthesis of auxiliary assertions for temporal verification, e.g., linear and non-linear invariants and ranking functions (Colón et al. 2003; Sankaranarayanan et al. 2004; Cousot 2005; Kapur 2006). We extend its applicability to synthesizing recurrent sets.

The constraint-based approach to the generation of auxiliary assertions reduces the computation of an assertion to a constraint solving problem. The reduction is performed in three steps. First, a *template* that represents an assertion to be computed is fixed in an *a priori* chosen language. The parameters in the template are the unknown coefficients that determine the assertion. Second, a set of *constraints* over these parameters is defined. The constraints encode the validity of the assertion. This means that every solution to the constraint system yields a valid assertion. Third, an *assertion* is obtained by solving the resulting constraint system.

Our approach to generate recurrent sets follows these three steps. We use *templates* over linear inequalities to represents recurrent sets. We derive constraints over template parameters that encode the conditions in Equations (2)–(4) from Section 5.1. Then, we solve the constraints and obtain a recurrent set.

Recurrent sets are defined by universally quantified conditions. As for invariant generation in linear arithmetic, our main technical tool for the elimination of universal quantification will be Farkas' lemma from linear programming.

THEOREM 2 (Farkas' Lemma (Schrijver 1986)). *A satisfiable system of linear inequalities $Ax \leq b$ implies an inequality $cx \leq \delta$ if and only if there exists a non-negative vector λ such that $\lambda A = c$ and $\lambda b \leq \delta$.*

We now present the details of our algorithm for the computation of recurrent sets. We assume that the transition relations of the stem and loop parts are given by systems of inequalities. In particular, we assume that the transition relation of the loop is given by a guarded command with the guard $Gx \leq g$ and updates $x' = Ux + u$.[1] Our algorithm computes a recurrent set $\mathcal{G}$ that is an instantiation of a template consisting of a conjunction of linear inequalities:

$$\mathcal{G} = \ Tx \leq t.$$

First, we present a translation of Condition (3) into constraints over template parameters. We eliminate the existential quantification in Condition (3) by substituting the definition of the primed variables given by the loop update:

$$\forall x.\ \mathcal{G}(x) \rightarrow \rho_{loop}(x, Ux + u) \wedge \mathcal{G}(Ux + u),$$

which we write in matrix form:

$$\forall x.\ Tx \leq t \rightarrow Gx \leq g \wedge TUx \leq t - Tu.$$

Here, the template parameters T and t are existentially quantified. Next, we eliminate the universal quantification by encoding the validity of implication using Farkas' lemma:

$$\exists \Lambda \geq 0.\ \Lambda T = \begin{pmatrix} G \\ TU \end{pmatrix} \wedge \Lambda t \leq \begin{pmatrix} g \\ t - Tu \end{pmatrix}. \tag{6}$$

We can translate Condition (2) into the constraint

$$\forall \mu \geq 0.\ \mu T = 0 \rightarrow \mu t \geq 0.$$

The translation for Condition (4) is similar. Together with (6), this leads to the final constraint defining recurrent sets that contains quantifier alternation, from existential to universal, as well as non-linear constraints arising from the multiplication between template parameters and variables that encode the implication validity. The constraints are similar to those for invariant generation (Colón et al. 2003). We can use existing solving techniques based on instantiations and case splitting. Unfortunately, there is no practical constraint solver that supports quantifier alternation. We propose an alternative solution, which we can be implemented using a constraint

[1] We use low case x instead of X to denote program variables in this section to avoid confusion between matrices and vectors.

input
$S\left(\begin{smallmatrix} x \\ x' \end{smallmatrix}\right) \leq s$: transition relation of the stem
$Gx \leq g \wedge x' = Ux + u$: transition relation of the loop
$Tx \leq t$: template for recurrent set
vars
Φ : auxiliary constraint
σ, σ' : program states – valuations of x and x'
begin
$\Phi :=\ \exists \Lambda \geq 0.\ \Lambda T = \begin{pmatrix} G \\ TU \end{pmatrix} \wedge \Lambda t \leq \begin{pmatrix} g \\ t - Tu \end{pmatrix}$
try
CHOOSE $(T^*, t^*) \models \Phi$
if exist $(\sigma, \sigma') \models S\left(\begin{smallmatrix} x \\ x' \end{smallmatrix}\right) \leq s \wedge T^* x' \leq t^*$ **then**
return "recurrent set $T^* x \leq t^*$"
else
backtrack
catch CHOICEFAILURE **do**
return "no recurrent set for template $Tx \leq t$"
end.

Figure 4. Auxiliary function NONTERMLASSO for checking non-termination of linear arithmetic lassos.

solver that can iteratively enumerate all solutions. We enforce Conditions (2) and (4) by evaluating them for the values of T and t that the constraint solver computes for constraint (6). If the conditions are not satisfiable, then we require the solver to find alternative values for T and t. A constraint logic programming-based solver, e.g., clp(Q,R) (Holzbaur 1995), can implement this backtracking search. We summarize the described algorithm NONTERMLASSO in Figure 4.

THEOREM 3. *If the algorithm* NONTERMLASSO *terminates on an input lasso with stem* $S(\begin{smallmatrix} x \\ x' \end{smallmatrix}) \leq s$*, loop* $Gx \leq g \wedge x' = Ux + u$*, and template* $Tx \leq t$ *and returns "recurrent set"* $T^* x \leq t^*$ *then* $T^* x \leq t^*$ *is a recurrent set for the lasso.*

We illustrate the constraint generation process on the loop

$$x \geq 0 \wedge x' = x + y \wedge y' = y + 1,$$

which we write in matrix form as

$$\underbrace{\begin{pmatrix} -1 & 0 \end{pmatrix}}_{G} \begin{pmatrix} x \\ y \end{pmatrix} \leq \underbrace{0}_{g} \wedge \begin{pmatrix} x' \\ y' \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}}_{U} \begin{pmatrix} x \\ y \end{pmatrix} + \underbrace{\begin{pmatrix} 0 \\ 1 \end{pmatrix}}_{u}.$$

First, we assume a template

$$t_x x + t_y y \leq t \quad \wedge \quad s_x x + s_y y \leq s,$$

where t_x, t_y, t, s_x, s_y, and s are unknown parameters. Then, we have

$$TU = \begin{pmatrix} t_x & t_x + t_y \\ s_x & s_x + s_y \end{pmatrix} \quad \text{and} \quad t - Tu = \begin{pmatrix} t - t_y \\ s - s_y \end{pmatrix}.$$

Following (6), for

$$\Lambda = \begin{pmatrix} \lambda_{11} & \lambda_{12} \\ \lambda_{21} & \lambda_{22} \\ \lambda_{31} & \lambda_{32} \end{pmatrix}$$

we obtain the constraint system:

$$\exists \Lambda \geq 0.\ \Lambda \begin{pmatrix} t_x & t_y \\ s_x & s_y \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ t_x & t_x + t_y \\ s_x & s_x + s_y \end{pmatrix} \wedge \Lambda \begin{pmatrix} t \\ s \end{pmatrix} \leq \begin{pmatrix} 0 \\ t - t_y \\ s - s_y \end{pmatrix}.$$

We compute a solution

t_x	t_y	t	s_x	s_y	s
-1	0	0	0	-1	0

,

which defines the recurrent set $x \geq 0 \wedge y \geq 0$. It is straightforward to check that the validity of the corresponding implication in Condition (3) is established by Λ such that

$$\Lambda = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix},$$

for which the constraint below evaluates to true:

$$\Lambda \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ -1 & -1 \\ 0 & -1 \end{pmatrix} \wedge \Lambda \begin{pmatrix} 0 \\ 0 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

Our algorithm requires that a template for recurrent sets is provided. We propose an *iterative strengthening heuristic* to find a template for which a recurrent set exists. We start with a template that is a singleton conjunction, and incrementally add more conjuncts if the constraint solving fails. Our practical experience demonstrates that the solving fails quickly, thus, allowing us to continue with a stronger template when necessary.

5.3 Weaker Conditions for Recurrent Sets

A lasso induces an infinite execution if and only if each unrolling of its loop part is not well-founded at a state reachable by executing the stem. These unrollings determine weaker conditions on recurrent sets, which can sometimes lead to more succinct representations for recurrent sets.

We first illustrate the weakening by example, and then provide a formal account. Consider a lasso

$$\begin{aligned} \rho_{stem} = &\quad y' = 0, \\ \rho_{loop} = &\quad x \geq 0 \wedge x' = x + y \wedge y' = 1 - y. \end{aligned}$$

This lasso induces infinite executions, and a witnessing recurrent set is:

$$\mathcal{G}_1(x, y) = \quad x \geq 0 \wedge (y = 0 \vee y = 1).$$

We observe that the recurrent set contains a disjunction. In general, disjunctions are difficult for constraint solvers to reason about. However, we may also consider a loop relation ρ_{loop^2} obtained from the given one by unrolling it once:

$$\begin{aligned} \rho_{loop^2}(X, X') &= \rho_{loop} \circ \rho_{loop} \\ &= x \geq 0 \wedge x + y \geq 0 \wedge x' = x + 1 \wedge y' = y. \end{aligned}$$

For this relation we compute a recurrent set

$$\mathcal{G}_2(x, y) = \quad x \geq 0 \wedge y = 0.$$

This set is represented using a conjunction of atomic predicates. Any infinite sequence induced by the lasso $stem.loop^2$ is also induced by the original lasso.

We now define the i-th weakening for recurrent sets. Given a binary relation ρ, we say that $\mathcal{G}$ is i-th recurrent for ρ, for $i \geq 1$, if it is recurrent for the relation ρ^i, which is obtained from ρ by unrolling it i-many times. Formally,

$$\forall X\ \exists X'.\ \mathcal{G}(X) \rightarrow \rho^i(X, X') \wedge \mathcal{G}(X'), \tag{7}$$

where

$$\rho^i = \begin{cases} \rho & \text{if } i = 1, \\ \rho \circ \rho^{i-1} & \text{if } i > 1. \end{cases}$$

The constraint-based non-termination check (Algorithm NONTERMLASSO) can be implemented using i-th recurrent sets. We observe that we can account for i-th weakening, where $i > 1$, by considering the (unrolled) loop relation:

$$\rho^i_{loop}(x, x') = \; GU^{i-1}x \leq g - \sum_{j=1}^{i-1} GU^{j-1}u \;\wedge$$
$$x' = U^i x + \sum_{j=1}^{i} U^{j-1}u.$$

As a heuristic, when proving non-termination, we first try to compute a recurrent set following the original definition, i.e., no weakening is applied. If the computation fails, then we continue with a weaker definition of recurrent sets until either an i-th recurrent set is computed or an upper bound on the number of weakening attempts is reached. The template strengthening heuristic from the previous section can be combined with weakening of recurrent sets. We increase the number of conjuncts in the template only after a sequence of weakening steps is explored up to an a priori given bound. Such a combination attempts to avoid the transition to more expensive constraint solving tasks by first trying to simplify recurrent sets through their weakening.

6. Experiences

6.1 Implementation

We have implemented TNT, a tool that implements the termination checking algorithm for C programs. TNT has an outer loop that performs concolic execution (Godefroid et al. 2005; Sen et al. 2005) by running the program on concrete as well as symbolic inputs. The constraints generated during concolic execution are bit accurate, and solved using the decision procedure STP (Ganesh and Dill 2007). In our symbolic execution, the heap is always kept concrete, we only allow symbolic constants with base types. The template-based search for recurrent sets is implemented using a Sicstus Prolog based constraint solver for invariant generation.

In preliminary experiments, we checked for non-termination of simple and small programs, including the programs from Section 2, and an abstraction of the non-termination bug from (Cook et al. 2006). In each case, the non-terminating loop was identified in a few seconds.

One limitation of the current implementation is that the heap is concrete. While this means that the recurrent sets benefit from precise address information, it is also a limitation of the tool to find potentially infinite executions that depend on shape assumptions on data structures. For example, we cannot catch infinite executions arising from acyclic list traversal routines when they are applied on circular lists as input. Integrating our work with symbolic shape information is left as future work.

6.2 Mondriaan Memory Protection

In addition to the simple examples, we ran our tool on an early implementation of the Mondriaan memory protection system (Witchel et al. 2002; Witchel 2004).

Mondriaan is a protection scheme that allows flexible memory sharing and fine-grained permission control between different user applications and a trusted supervisor mode running on an OS. Unlike usual virtual memory that operates at the page level, Mondriaan allows arbitrary permissions control at the granularity of individual memory words. In the Mondriaan implementation, memory is organized as a linear address space, divided into *user segments*. Each user segment determines a range of addresses and permissions associated with addresses in that range, and is defined as a triple $\langle b, l, p\rangle$ of a base address b, a length l, and a permission p. By setting the permissions of a *user segment*, a process can flexibly and safely share its address space with other processes. A privileged su-

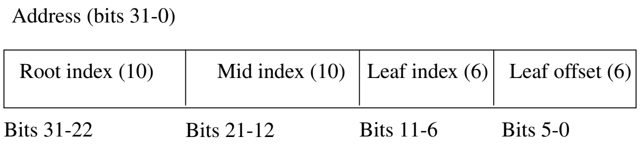

Figure 5. Modriaan permission table indexing

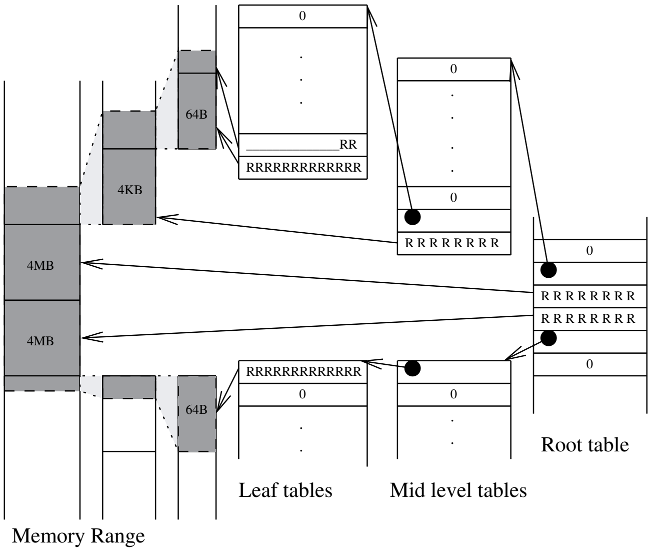

Figure 6. Typical permission table generated by `_mmpt_insert`

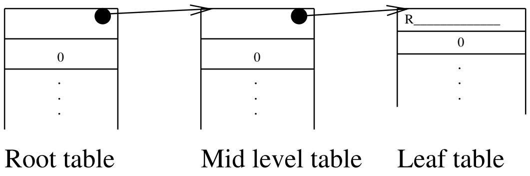

Figure 7. Table state after first call of `_mmpt_insert`

pervisor mode provides an API to modify permissions. The permissions associated with a user segment can be modified using this API by specifying the base word address, the length of the user segment in words, and the desired permission (none, read-only, read-write, or execute-read). Each *protection domain* (i.e., group of processes sharing the address space) maintains a permissions table, stored in privileged memory, that specifies the permission associated with each address of the address space.

The permissions table is organized as a compressed multi-level table, with three levels: a root table, a mid-level table, and a leaf table. Figure 5 shows how the table is indexed by a 32-bit address. The root table has 1024 entries (indexed by the top 10 bits in Figure 5), each of which maps a 4MB block of memory. Each mid-level table has 1024 entries (indexed by the next 10 bits), each of which maps a 4KB block. The leaf tables have 64 entries each (indexed by 6 bits), and each entry provides individual permissions for 16 four-byte words.

To save space, upper level table entries can either be pointers to lower level tables (or NULL), or directly hold a permissions vector for sub-blocks. In this case, the rightmost bit of the entry holds whether the entry is a pointer or a permissions vector (since addresses are word-aligned, the last two bits of an address are always zero). Permission vectors stored in upper levels of the table only store permissions for 8 sub-blocks (instead of 16 in the leaves). An auxiliary function `uentry_is_data` is used to check whether an entry is a pointer to a lower level table or a permissions vector.

We chose Mondriaan as our case study because (1) correctness is crucial, since the code runs in privileged mode in the kernel, (2) the code was complicated enough to warrant its author to annotate the code with properties (both as assertions and as comments), and to suggest it as a challenge for formal verification, and (3) the code is low-level, performing extensive bitwise operators that extract indices to arrays, and memory-intensive, traversing multiple levels of data structures, and both these features provide exceptional challenges to static analysis.

Initially, the Mondriaan implementation creates the root permission table with 1024 entries all filled with 0s (that is, no permissions associated with any address). Figure 9 shows source code to perform updates from an early version of Mondriaan that was provided to us by the author Emmett Witchel as a challenging verification problem. The code updates the permissions of a user segment in the permission table by taking as input a user segment and updating the permissions of the segment in the table.

The actual update is performed by the recursive procedure `_mmpt_insert`, shown in Figure 9, which takes as input a pointer to the permission table structure `mmpt`, a user segment (a base address, a pointer `len` to the length of the segment, and the desired protection `prot`), and additional control parameters such as a pointer to the current table (initially, the root table), the current level (root, mid-level, or leaf), and flags determining whether or not some table can be freed and whether or not to allocate. An insertion into the table for a user segment is performed as a call

$$\texttt{\_mmpt\_insert}(\texttt{mmpt}, \texttt{base}, \&\texttt{len}, \texttt{prot},\\ \texttt{mmpt}\rightarrow\texttt{tab}, 0, \&\texttt{nonzero}, 1);$$

The code is complicated, as it must consider several different cases in inserting new permissions to the table, and performs dynamic allocation or freeing of lower level tables when they are not necessary. Further, a user segment is broken into parts in the insertion process, and the insertion routine is called recursively for each part. Instead of explaining the code line by line, we will explain the functional behavior of `_mmpt_insert`.

As shown in Figure 6, `_mmpt_insert` splits the memory range from `base` to `base` + `len` in aligned 4MB blocks at the root level. For an update, this can result in one of the following two cases:

Exact Coverage. For every 4MB sub-block of the user segment that is aligned with the sub-block size (512KB), the permission vector is set in the corresponding entry in the root table. If the root entry was a pointer to a mid-level table, then the mid-level table is recursively deallocated.

Overlapped Coverage. Breaking the user segment into aligned 4MB blocks can leave "extra" blocks at the beginning and at the end, that is, the first and last blocks of the user segment may not be aligned with 512KB sub-blocks at the root level. For these blocks, `_mmpt_insert` recursively goes to the mid-level table (allocating the mid-level table if necessary) and sets permissions in the mid-level table.

Depending on the stored value in the root entry for the base address, `_mmpt_insert` first does following action:

1. If the root entry is 0 then it allocates a new mid level permission table of 1024 entries and fills the entries in the new table with zeros. The root entry is set as a pointer to this table.
2. If the root entry is a permission vector then it allocates a mid-level permission table and fills it with permission vectors. The root entry is set as a pointer to this table.
3. If the root entry is a already a pointer to a mid-level table then it follows the pointer to the mid-level table.

Then, `_mmpt_insert` computes the the memory range of the first or last section and makes a recursive call with parameters this memory range, pointer to this mid level table. With the same logic as for the root table, it divides the memory range in 4KB blocks and does the same operations for the permissions. For the possibly unaligned first and last block, it generates leaf tables. Permissions in leaf tables are set by a recursive call to `_mmpt_insert`. If leaf tables have unaligned blocks then permissions are approximated in sub-blocks of size 4 bytes in the leaf table.

Non-termination in `_mmpt_insert`. When we ran TNT on `_mmpt_insert`, it found a non-terminating execution. This bug is caused by calling `_mmpt_insert` with a user segment whose length is not a multiple of 4. To illustrate, consider two consecutive calls to `_mmpt_insert` just after initialization (found by TNT). The first call sets read permission for a user segment of 4 bytes with base address 0. The second one sets the permission for a user segment of 3 bytes with base address 0. The first call terminates, while the second call runs into infinite recursion. Figure 7 shows the state of memory table after first call.

Let us trace the code of `_mmpt_insert` line by line during the second call. Procedures called by `_mmpt_insert` are summarized in Figure 8.

In the second call, `_mmpt_insert` is called with a (constant) pointer to the `mmpt` structure, a base address `base` of 0, the address of the length variable `len` which contains the length 3, a pointer to the root table, and `level` $=$ 0 (signifying root entry). (We omit the other, unimportant parameters.) At line 4, the insertion routine checks that $*$`len` is not zero, that is, the current memory chunk needs to be filled in the table. At line 5, the helper procedure `make_idx` computes the index of the entry in the root table which corresponds to `base`. This extracts the top 10 bits of `base`, which turns out to be 0. At line 6, the last comparison in the conditional checks whether the value of `*len` is greater than the block size of the root entry (4MB). Since the length is 3, this check fails and control passes to line 19. The conditional on line 19 passes, since the passed table is not a leaf table, and the entry in the root table is a pointer to a mid-level table. (Recall the configuration of the permission table after the first call from Figure 7.)

Now at line 20, `_mmpt_insert` makes a recursive call with a pointer to the mid-level table that is pointed to by the zeroth entry of the root table, `tab`[`idx`]. The parameters `base` and `len` do not change, but `level` increases to 1, denoting that the current table is a mid-level table. In this recursive call, the control again comes to line 20, this time making another recursive call with a pointer to a leaf table, and with level equal to 2.

In this second recursive call, the conditions at lines 6, 19, and 21 all fail because the current table is a leaf table. Thus, control jumps to line 41. The `for` loop at line 41 does not execute at all, since `*len` (which is equal to 3) is smaller then the sub-block length (4 bytes) of the leaf table. Thus, control directly jumps to line 50 and makes a recursive call with base 0, length 3, a pointer to the root table, and level reset to 0. These are the same parameters which were passed at the start of execution of the procedure `_mmpt_insert`. This causes an infinite loop, as this recursion does not terminate.

`tab_base`(struct mmpt* mmpt,int base,int level)	Returns the lower boundary of memory range corresponds to the permission table of given level whose corresponding memory range contains `base`
`tab_len`(struct mmpt* mmpt,int level)	Returns the size of memory block corresponds to each entry of permission table of given level.
`tab_addr`(struct mmpt* mmpt, int base, int idx, int level)	Returns the lower boundary of memory range corresponds to the entry at index `idx` in permission table of given level whose corresponding memory range contains `base`
`tab_nentries`(struct mmpt* mmpt,int level)	Returns number of entries in the table of given level.
`subblock_len`(struct mmpt* mmpt,int level)	Returns sub block size of the level.
`make_idx`(struct mmpt* mmpt,int base,int level)	Returns the index of first entry in permission table whose corresponding memory range contains `base`
`uentry_is_data`(struct mmpt* mmpt,int entry)	Checks if the entry in the permission table is a pointer or a permission vector.
`entry_prot`(mmpt, permission_entry, i)	Returns i^{th} permission field in `permission_entry`

Figure 8. Summary of functions called by `_mmpt_insert`

```
1:     for (x = 0; x < 100; x++) { ... }
2:     while (y > 0) {
3:             y++;
4:             ...
5:     }
```

Figure 10. TNT can skip the analysis of the first loop, which is terminating, thus accelerating its search for non-terminating lassos. We assume that the omitted code fragments do not modify either `x` or `y`.

While this bug can be found using random testing followed by manual inspection in a debugger, the complexity of the code path (there are 3 levels of recursive calls involved) shows the utility of having a tool like TNT check the executions for possible non-termination over the current manual process of stepping through a debugger.

Proving Termination. The non-terminating execution occurred because of the unchecked assumption that the lengths of user segments are always a multiple of 4. We put this check in the code. This time, TNT timed out without identifying any infinite execution. Since our implementation does not implement a termination-check based acceleration, most of the time was spent in analyzing longer and longer symbolic traces of recursive calls.

We then proved termination of the corrected version by hand. The proof of termination involves a lexicographic ranking on the pair $(*\texttt{len}, 2-\texttt{level})$, as on every recursive call, either the length goes down or if the length remains the same, the distance of the level from the leaf tree goes down. The reasoning for termination uses crucially the invariant that the length is always a multiple of 4 to rule out the previous infinite execution.

We believe this example is a good challenge problem for termination checkers. Unfortunately, well-documented limitations of current termination checkers (to deal with bitwise operators, or shared data structures on the heap) make it difficult to prove termination of this program automatically with the current tools.

7. Acceleration for NonTerm

In this section we describe a practical extension for our non-termination checking algorithm NonTerm shown in Section 4. We describe how to avoid redundant non-termination checks by accelerating the traversal of terminating loops.

We first illustrate the proposed technique on the example code shown in Figure 10. The first loop statement terminates, thus, our algorithm should not take lassos that loop at line 1 into account and try to prove their non-termination. We can apply an existing termination prover to automatically check if the first loop terminates. The successul check allows one to unroll the first loop until it exits, and consider the resulting sequence as an initial stem. Such an unrolling avoids 100 attempts to prove non-termination of the first loop, and directly leads us to the interesting part of the program.

Given a lasso, the non-termination checking algorithm NonTermLasso, as described in the previous section, can fail to return a positive result. For complete checking algorithms, the failure is caused by the termination of the lasso. Incomplete algorithms can produce an indefinite result, which leaves open the possibility that the lasso may be terminating. Since the exploration of terminating loops does not advances the search for infinite executions, we propose a modification of NonTerm that removes such loops from consideration. See Figure 11 for the modified statements.

If the lasso can be proven terminating no matter what the input is, we lead the execution through the loop until it exits. In order words, we fully unroll the loop by executing the loop sequence, and rely on the proof of termination to guarantee the convergence of the unrolling. Thus, we eventually reach line 18.9. The resulting sequence will be used to seed the selection of the next stem to be considered, i.e., the next stem will be chosen to contain the sequence as a prefix. Thus, NonTerm can reach interesting parts of code by passing across loops in one step, without any interruption at each iteration.

If the termination property of the lasso cannot be determined, we continue our search for non-terminating lassos. During the subsequent iterations of NonTerm, we shall only consider lassos that have the current stem as a prefix. Thus, we ensure that the search makes progress.

8. Conclusion

Termination checkers search for ranking functions; our non-termination checker searches for a recurrent set of states. Always one of the two must exist, and its presence or absence would be determined by a complete method. In practice, however, tools are incomplete. A failure to prove termination does not indicate definite non-termination, and vice versa. In particular, we looked at a non-termination proof for Mondriaan only after failed attempts to prove termination. This illustrates the need for complementing a termination checker with a non-termination checker, because the former, when failing, gives no information as to why it failed to dis-

```
18.1     else if TERMLASSO(stem, loop) then
18.2       repeat                                   (* unrolling loop *)
18.3         S := {s' | s --loop--> s'}
18.4         if S ≠ ∅ then
18.5           CHOOSE s ∈ S
18.6           stem := stem • loop
18.7         else
18.8           loop := ε
18.9           break
18.10      done
18.11    else
```

Figure 11. Acceleration of the algorithm NONTERM for testing termination. Lines 18.1—18.11 replace line 18 in Figure 2. We unroll the loop part of terminating lasso without intermediate checks for non-termination. Recall that the variable s holds the value of the current program state, which successors are computed during loop unrolling.

cover a ranking function. It may have failed because no such function exists within the assumptions and restrictions under which the checker operates, or because the program does not terminate.

Research in testing has focused on safety assertions, primarily because counterexamples are *finite* objects that can be recognized online. In contrast, infinite loops are identified using ad hoc techniques and guesses about the program run time, which is almost always performed through laborious manual inspection. Therefore, dual to the use of safety assertions in code, we must also develop specification techniques and automatic tools that assert and check absence of non-terminating executions. We believe that TNT is a step towards this class of tools.

Acknowledgments. Gupta and Rybalchenko are supported in part by Microsoft Research through the European Fellowship Programme. Majumdar and Xu were supported by the National Science Foundation grants CCF-0427202, CCF-0546170, and CNS-0720881. We thank Emmett Witchel for sharing and explaining the Mondriaan code and Patrice Godefroid for several useful comments on the draft.

References

A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.

Joshua Bloch. Nearly all binary searches and mergesorts are broken, June 2006. `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`.

A. Bradley, Z. Manna, and H. Sipma. The polyranking principle. In *Proc. ICALP*, LNCS 3580, pages 1349–1361. Springer, 2005.

E.M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1): 7–34, 2001.

M. Colón and H. Sipma. Practical methods for proving program termination. In *Proc. CAV*, LNCS 2404, pages 442–454. Springer, 2002.

M. Colón, S. Sankaranarayanan, and H.B. Sipma. Linear invariant generation using non-linear constraint solving. In *Proc. CAV*, LNCS 2725, pages 420–432. Springer, 2003.

B. Cook, D. Kroening, and N. Sharygina. COGENT: Accurate theorem proving for program verification. In *Proc. CAV*, LNCS 3576, pages 296–300. Springer, 2005.

B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. PLDI*, pages 415–426. ACM, 2006.

P. Cousot. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In *Proc. VMCAI*, LNCS 3385, pages 1–24. Springer, 2005.

M. d'Amorim and G. Rosu. Efficient monitoring of omega-languages. In *Proc. CAV*, LNCS 3576, pages 364–378. Springer, 2005.

V. Ganesh and D.L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. CAV*, LNCS 4590, pages 519–531. Springer, 2007.

P. Godefroid. The soundness of bugs is what matters (position statement). In *BUGS'2005 (PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools)*, 2005.

P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 213–223. ACM, 2005.

B. Gulavani, T.A. Henzinger, Y. Kannan, A. Nori, and S.K. Rajamani. Synergy: A new algorithm for property checking. In *Proc. FSE*, pages 117–127. ACM, 2006.

T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.

C. Holzbaur. *OFAI clp(q,r) Manual, Edition 1.3.3*. Austrian Research Institute for Artificial Intelligence, Vienna, 1995. TR-95-09.

D. Kapur. Automatically generating loop invariants using quantifier elimination. Technical Report 05431 (*Deduction and Applications*), IBFI Schloss Dagstuhl, 2006.

T. Kremenek and D.R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proc. SAS*, LNCS 2694, pages 295–315. Springer, 2003.

A. Pnueli, A. Zaks, and L.D. Zuck. Monitoring interfaces for faults. *Electr. Notes Theor. Comput. Sci.*, 144(4):73–89, 2006.

S. Sankaranarayanan, H.B. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *Proc. POPL*, pages 318–329. ACM, 2004.

A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.

K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In *Proc. ICSE*, pages 418–427. ACM, 2004.

K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272. ACM, 2005.

H. Velroyen. Automatic non-termination analysis of imperative programs. Master's thesis, Chalmers University of Technology, Aachen Technical University, 2007.

E. Witchel. *Mondriaan Memory Protection*. PhD thesis, Massachusetts Institute of Technologys, 2004.

E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proc. ASPLOS*, pages 304–316. ACM, 2002.

Y. Xie and A. Aiken. Saturn: A SAT-based tool for bug detection. In *Proc. CAV*, LNCS 3576, pages 139–143. Springer, 2005.

```
static void
_mmpt_insert(struct mmpt* mmpt, unsigned long base, unsigned long* len, int prot,
             tab_t* tab, int level, int* nonzero, int allocate_ok) {
  unsigned int idx; tab_t entry;
  if(*len == 0) return;
  idx = make_idx(mmpt, base, level);
  if(level < 2 && base == tab_base(mmpt, base, level + 1) && *len >= tab_len(mmpt, level + 1)) {
  // CASE A: Upper level, new region is aligned & spans at least one entry
     unsigned int entry_len;
     if(tab[idx] && !uentry_is_data(mmpt, tab[idx])) {
        look_for_nonzero(mmpt, (tab_t*)tab[idx], level, nonzero);
        table_free(mmpt, (void*)tab[idx], level + 1);
       tab[idx] = 0;
     }
     entry = tab[idx];
     entry_len = make_entry(mmpt, base, *len, prot, level, &entry);
     tab[idx] = entry;
     *len -= entry_len;
     base += entry_len;
     _mmpt_insert(mmpt, base, len, prot, mmpt->tab, 0, nonzero, allocate_ok);
  } else if(level < 2 && tab[idx] && !uentry_is_data(mmpt, tab[idx])) {
  // CASE B: Upper level, pointer entry
  // Recurse down through pointer
     _mmpt_insert(mmpt, base, len, prot, (tab_t*)tab[idx], level + 1, nonzero, allocate_ok);
  } else if(level < 2 && ((base & (subblock_len(mmpt, level)-1)) != 0 || *len < subblock_len(mmpt, level))) {
  // CASE C: Upper level, NULL or data entry, new region doesn't fit in
  // subblock (not aligned or not big enough)
     unsigned long upper_data_entry = tab[idx];
     unsigned int i;
     *nonzero |= (tab[idx] != 0);
     if(allocate_ok) {
        unsigned long sub_len;
        tab[idx] = (tab_t)xmalloc(tab_nentries(mmpt, level + 1) * sizeof(*mmpt->tab));
        memset((tab_t*)tab[idx], 0, tab_nentries(mmpt, level + 1) * sizeof(*mmpt->tab));
        for(i = 0; i < 1<<mmpt->lg_num_subblock[level]; ++i) {
           sub_len = subblock_len(mmpt, level);
           _mmpt_insert(mmpt, tab_base(mmpt, base, level+1) + i * subblock_len(mmpt, level), &sub_len,
                     entry_prot(mmpt, upper_data_entry, i), (tab_t*)tab[idx], level + 1, nonzero, allocate_ok);
        }
        _mmpt_insert(mmpt, base, len, prot, (tab_t*)tab[idx], level + 1, nonzero, allocate_ok);
     } else {
        unsigned int tlen = tab_len(mmpt, level + 1);
     // CASE D: Upper level, NULL or data entry, new region doesn't fit in
     // subblock (not aligned or not big enough), and not
     // allocating new tables
        if(*len < tlen) return;
        *len -= tlen;
        _mmpt_insert(mmpt, tab_addr(mmpt, base, idx+1, level), len, prot,
                     mmpt->tab, 0, nonzero, allocate_ok);
     }
  } else {
  // CASE E: Any level, NULL or data entry, fill in the rest of
  // this table and recurse for the remainder if necessary.
     for(; *len >= subblock_len(mmpt, level)
         && idx < tab_nentries(mmpt, level); idx++) {
        int entry_len;
        *nonzero |= (tab[idx] != 0);
        entry = tab[idx];
        entry_len = make_entry(mmpt, base, *len, prot, level, &entry);
        tab[idx] = entry;
        *len -= entry_len;
        base += entry_len;
     }
     _mmpt_insert(mmpt, base, len, prot, mmpt->tab, 0, nonzero, allocate_ok);
  }
}
```

Figure 9. Mondriaan insertion code