Termination Proofs for Systems Code [*]

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Andrey Rybalchenko
Max-Planck-Institut für Informatik and EPFL
rybal@mpi-sb.mpg.de and andrey.rybalchenko@epfl.ch

Abstract

Program termination is central to the process of ensuring that systems code can always react. We describe a new program termination prover that performs a path-sensitive and context-sensitive program analysis and provides capacity for large program fragments (i.e. more than 20,000 lines of code) together with support for programming language features such as arbitrarily nested loops, pointers, function-pointers, side-effects, etc. We also present experimental results on device driver dispatch routines from the Windows operating system. The most distinguishing aspect of our tool is how it shifts the balance between the two tasks of *constructing* and respectively *checking* the termination argument. Checking becomes the hard step. In this paper we show how we solve the corresponding challenge of *checking* with *binary reachability analysis*.

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 termination, model checking, program verification, formal verification

1. Introduction

Reactive systems (e.g. operating systems, web servers, mail servers, database engines, etc) are usually constructed from a set of components that we expect will always terminate. Cases where these functions unexpectedly do not return to their calling context leads to non-responsive systems. Device driver dispatch routines, for example, must eventually return to their caller. Consider the function in Figure 1 which is called from several dispatch routines within the Windows serial enumeration device driver. This code calls other serial-based device drivers by passing I/O request packets via the kernel routine `IoCallDriver` (line 50, `pIrp` is a pointer to the request packet and `FdoData->TopOfStack` is the pointer to another serial-based device driver). In the case where the other device driver returns a return-value that indicates success, but places 0 in `PIoStatusBlock->Information`, the serial enumeration driver will fail to increment the value pointed to by `nActual` (line 66), possibly causing the driver to infinitely execute this loop and not return to its calling context. The consequence of this error is that the computer's serial devices could become non-responsive. Worse yet, depending on what actions the other device driver takes, this loop may cause repeated acquiring and releasing of kernel resources (memory, locks, etc) at high priority and excessive physical bus activity. This extra work stresses the operating system, the other drivers, and the user applications running on the system, which may cause them to crash or become non-responsive too.

This example demonstrates how a notion of termination is central to the process of ensuring that reactive systems can always react. Until now no automatic termination tool has ever been able to provide a capacity for large program fragments (>20,000 lines) together with accurate support for programming language features such as arbitrarily nested loops, pointers, function-pointers, side-effects, etc. In this paper we describe such a tool, called TERMINATOR.

TERMINATOR's most distinguishing aspect, with respect to previous methods and tools for proving program termination, is how it shifts the balance between the two tasks of *constructing* and respectively *checking* the termination argument. The classical method is to construct an expression defining the *rank* of a state and then to check that its value decreases in every transition from a reachable state to a next one. The construction of the ranking function is the hard part and forms a task that needs to be applied to the whole program. The checking part is relatively easy. In our method, the task of constructing ranking functions is the relatively easy part; they are constructed on demand based on the examination of only a few selected paths through the program.

Furthermore, TERMINATOR is not required to construct only *one* correct termination argument but rather a set of guesses of possible arguments, some of which may be bad guesses. That is, this set need not be the exact set of the 'right' ranking functions but only a *superset*. We find the same monotonicity of the refinement of the termination argument as with iterative abstraction refinement for safety (the set of predicates need not be the exact set of 'right' predicates but only a superset).

Checking the termination argument is the hard part of our method. This is because the termination argument is now a set of ranking functions, not a single ranking function. With a single ranking function one must show that the rank decreases from the pre- to post-state after executing each single transition step. In our setting it is not sufficient to look at a single transition step. Instead, we must consider all *finite sequences of transitions*. We must show that, for every sequence, one of the ranking functions decreases

[*] The second and third author were supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-320-4/06/0006. . . $5.00.

```
NTSTATUS
Serenum_ReadSerialPort(CHAR * PReadBuffer, USHORT Buflen,
                       ULONG Timeout, USHORT * nActual,
                       IO_STATUS_BLOCK * PIoStatusBlock,
                       const FDO_DEVICE_DATA * FdoData)
{
    NTSTATUS status;
    IRP * pIrp;
    LARGE_INTEGER startingOffset;
    KEVENT event;
    SERIAL_TIMEOUTS timeouts;
    ULONG i;

    startingOffset.QuadPart = (LONGLONG) 0;
    //
    // Set the proper timeouts for the read
    //

    timeouts.ReadIntervalTimeout = MAXULONG;
    timeouts.ReadTotalTimeoutMultiplier = MAXULONG;
    timeouts.ReadTotalTimeoutConstant = Timeout;
    timeouts.WriteTotalTimeoutMultiplier = 0;
    timeouts.WriteTotalTimeoutConstant = 0;

    KeInitializeEvent(&event, NotificationEvent, FALSE);

    status = Serenum_IoSyncIoctlEx(IOCTL_SERIAL_SET_TIMEOUTS, FALSE,FdoData->TopOfStack,
                   &event, &timeouts, sizeof(timeouts), NULL, 0);

    if (!NT_SUCCESS(status)) {
       return status;
    }

    Serenum_KdPrint(FdoData, SER_DBG_SS_TRACE, ("Read pending...\n"));

    *nActual = 0;

    while (*nActual < Buflen) {
        KeClearEvent(&event);

        pIrp = IoBuildSynchronousFsdRequest(IRP_MJ_READ, FdoData->TopOfStack,
                                            PReadBuffer, 1, &startingOffset,
                                            &event, PIoStatusBlock);

        if (pIrp == NULL) {
            Serenum_KdPrint(FdoData, SER_DBG_SS_ERROR, ("Failed to allocate IRP\n"));
            return STATUS_INSUFFICIENT_RESOURCES;
        }

        status = IoCallDriver(FdoData->TopOfStack, pIrp);

        if (status == STATUS_PENDING) {

            status = KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);

            if (status == STATUS_SUCCESS) {
               status = PIoStatusBlock->Status;
            }
        }

        if (!NT_SUCCESS(status) || status == STATUS_TIMEOUT) {
           Serenum_KdPrint (FdoData, SER_DBG_SS_ERROR, ("IO Call failed with status %x\n", status));
           return status;
        }

        *nActual += (USHORT)PIoStatusBlock->Information;
        PReadBuffer += (USHORT)PIoStatusBlock->Information;
    }

    return status;
}
```

Figure 1. Utility function containing a termination bug. This function is used by several dispatch routines in the Windows serial enumeration device driver, SERENUM.SYS

```
$T$ := $\emptyset$ (* termination argument: union of well-founded relations *)
repeat
    (* check binary reachability for $T$ *)
    if $R_I^+ \subseteq T$ then
        report “Terminating”
    else
        $\rho$ := a binary relation such that $\rho \subseteq R_I^+$ but $\rho \not\subseteq T$
    (* find rank function for $\rho$ if it exists *)
    if $\rho$ is not a well-founded relation then
        report “Not Terminating”
    else
        (* construct termination argument *)
        $W$ := a ranking relation, i.e. $\rho \subseteq W$ and $W$ well-founded
        $T$ := $T \cup W$
end.
```

Figure 2. Algorithm underlying TERMINATOR (from [13]). The binary reachability analysis, which checks the inclusion $R_I^+ \subseteq T$, is described in Section 3. If the check fails, it returns a binary relation ρ. The binary relation ρ is represented by a sequence of statements (with a loop), i.e. a program. The construction of a ranking function for this program and of the corresponding ranking relation W is implemented via techniques explained in [22].

between the pre- and post-state. In other words: we must first find all pairs of states s_1 and s_2 such that s_1 is reachable from the program's initial state and s_2 is reachable from s_1; and we must then show that the value of one the ranking functions decreases from s_1 to s_2. We call this task *binary reachability analysis*. Previously, it was not known whether binary reachability analysis could be made practical. The challenge raised by our approach was to show that this is indeed the case.

In this paper, we show that one can make binary reachability analysis practical. Furthermore, through experiments with TERMINATOR on Windows device drivers, we demonstrate that it is effective at proving termination arguments for industrial systems code.

2. TERMINATOR

The algorithm for the incremental construction of termination arguments underlying TERMINATOR is outlined in [13]. In this section we briefly describe TERMINATOR's design and explain the role that *binary reachability* plays in it.

TERMINATOR iterates between two procedures: its *binary reachability analysis* check the candidate termination argument, while its rank function synthesis engine incrementally constructs the termination argument.

See Figure 2. We assume a program P with a transition relation R and a set of initial states I. We define the binary reachability relation R_I^+ as the transitive closure of R restricted to reachable states. It consists of the pairs of states (s_1, s_2) such that s_2 is reachable from s_1 in at least one step and s_1 itself is reachable from an initial state s_0. Formally,

$$R_I^+ \triangleq \{(s_1, s_2) \mid \exists s_0 \in I.\, (s_0, s_1) \in R^* \wedge (s_1, s_2) \in R^+\}$$

Binary reachability analysis is a procedure that checks whether R_I^+ is contained in a given binary relation T:

$$R_I^+ \subseteq T$$

In the case that the inclusion does not hold, there exists a non-empty sequence of statements $\tau_1, \ldots, \tau_i, \ldots \tau_n$ with an execution sequence $s_0 \rightarrow_{\tau_1} s_1 \ldots s_{i-1} \rightarrow_{\tau_i} s_i \ldots s_{n-1} \rightarrow_{\tau_n} s_n$ such that the pair of states (s_i, s_n) is not in T, formally $(s_i, s_n) \in R_I^+$ but $(s_i, s_n) \notin T$. By the form of T in our setting, the two states s_i and s_n will always have the same program location; thus, the statements $\tau_{i+1}, \ldots \tau_n$ form a cycle in the program. Let ρ be the relation consisting of all pairs of states (s_i, s_n) that are connected by an execution sequence of the form described above (induced by the sequence of statements $\tau_1, \ldots, \tau_i, \ldots \tau_n$). That is, the relation ρ is the counterexample to the inclusion $R_I^+ \subseteq T$. We have $\rho \subseteq R_I^+$ and $\rho \not\subseteq T$.

TERMINATOR applies a rank synthesis tool based on [22] to ρ in attempt to construct a ranking function (thus proving its well-foundedness). From this we can construct a corresponding ranking relation W, which consists simply of the pairs of states with decreasing, positive rank. Thus, the constructed ranking relation W contains the relation ρ and is well-founded. See Figure 3 for an example.

Note that the union T of ranking relations, however, is in general not well-founded (it is only *disjunctively well-founded* in the terminology of [23]). This is why it would not be sufficient to show the inclusion $R \subseteq T$, and why we must instead prove $R_I^+ \subseteq T$. Our approach thus follows the framework of [20] where temporal reasoning (here, about termination) is reduced to first order reasoning using auxiliary assertions (here, the union T of ranking relations).

Example. Consider Figure 4. In order to prove this program terminating, TERMINATOR incrementally constructs a relation T that must eventually contain the binary reachability relation of that program. It is sufficient to specify (and to construct) the subrelations T^ℓ with pairs of states at the same program location ℓ where ℓ is one of 7, 12, 28 and 33. These locations form a set of cutpoints, in the terminology of Floyd [15]. We then can prove

$$R_I^+ \cap \{(s, t) \mid s(\mathsf{pc}) = t(\mathsf{pc}) = \ell\} \subseteq T^\ell$$

for each ℓ. The overall termination argument, T, can be constructed as the union of the T^ℓs together with a relation that expresses all of non-composable pairs. Formally,

$$T \triangleq T^7 \cup T^{12} \cup T^{28} \cup T^{33} \cup T^{\text{DIFF}}$$

where T^{DIFF} is a large set of trivially well-founded relations location such as $\{(s, t) \mid s(\mathsf{pc}) = 3 \wedge t(\mathsf{pc}) = 2\}$, $\{(s, t) \mid s(\mathsf{pc}) = 3 \wedge t(\mathsf{pc}) = 5\}$, etc. More precisely:

$$T^{\text{DIFF}} \triangleq \bigcup_{i \neq j} \{(s, t) \mid s(\mathsf{pc}) = i \wedge t(\mathsf{pc}) = j\}$$

The relations T^ℓ constructed by TERMINATOR are listed in Figure 5. Note that $T^{28} = T_0^{28} \cup T_1^{28} \cup T_2^{28}$ is a union of well-founded relations but is itself not well-founded.

TERMINATOR starts with T_0^ℓ, the empty relation denoted by `false`, and adds ranking relations T_1^ℓ, T_2^ℓ etc. until there are no more counterexamples for ℓ. This means that the relation T^ℓ formed by their union contains every pair (s_i, s_n) of states at location ℓ such that s_i is reachable from a state s_1 at the start of `main` and s_n is reachable from s_i by any non-empty sequence of execution steps.

TERMINATOR's binary reachability analysis is designed to support programs with pointer aliasing, as found in Figure 4. Notice that the termination argument at program location 28, T^{28}, does not specify the relationships between `x`, `y`, `p`, and `q`. This is an example of the separation of concerns in TERMINATOR. Binary reachability analysis tracks the aliasing of `*p` but the termination argument does not specify the aliasing relationships. Furthermore, the construction of the termination argument does not need to track them.

In order to prove that T^{33} contains all pairs of states at location 33, the binary reachability analysis must derive and prove that `b==true` is a program invariant at location 36. It thus subsumes

```
1: do {
2:   if (z>x) {
3:     x++;
4:   } else {
5:     z++;
6:   }
7: } while (x<y);
```

(a) Multipath loop

```
1: do {
2:   assume(z>x);
3:   x++;
7: } while (x<y);
```

(b) Loop representing the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 7$ through the loop in **(a)**.

$$\begin{array}{rcl} Q((x_0,y_0,z_0),(x_1,y_1,z_1)) & \triangleq & z_0 > x_0 \\ & \wedge & x_1 = x_0 + 1 \\ & \wedge & y_1 = y_0 \\ & \wedge & z_1 = z_0 \\ & \wedge & x_1 < y_1 \end{array}$$

(c) The relation Q represents the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 7$ through the loop in **(a)**. The expression (x_0, y_0, z_0) is used to represent program states at the beginning of the loop body, and (x_1, y_1, z_1) is used to represent the resulting state after executing the body. Three iterations of the loop in **(b)**, for example, can be represented by $Q \circ Q \circ Q$.

Figure 3. TERMINATOR uses binary reachability to search for possibly not well-founded paths and uses a rank function synthesis engine to try and show that they are spurious counterexamples (i.e. that they are well-founded). For example: TERMINATOR might find the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 1$ in the loop in **(a)**. This path can be represented either as another program loop found in **(b)** or the relation Q from **(c)**. TERMINATOR's rank function synthesis engine proves the well-foundedness of Q by finding the ranking function $f(x, y, z) \triangleq y - x$. The *ranking relation* W is defined as $\{(s,t) \mid f(t) > 0 \wedge f(t) \geq f(s) + 1\}$, i.e. $\{(s,t) \mid t(\mathtt{y}) - t(\mathtt{x}) > 0 \wedge t(\mathtt{y}) - t(\mathtt{x}) \geq s(\mathtt{y}) - s(\mathtt{x}) + 1\}$.

the synthesis of program invariants, the task of standard reachability analysis. Again, this is an example of the separation of concerns in TERMINATOR, since the construction of T^{33} does not involve deriving and proving the invariant.

If we comment out the code at line 11 in Figure 4 then TERMINATOR fails to prove termination and produces the following path through the program, denoted by line numbers (notice that the call to `Ack` is not reachable in the first iteration of the loop).

stem = 20→ 21→ 23→ 24→ 25→ 27→ 29→ 39→ 40→
41→ 42→ 44→ 27→ 29→ 30→ 3→ 5→ 9→ 12
cycle = 3→ 5→ 6→ 7→ 3→ 5→ 9→ 12

The path is a lasso consisting of two parts, a stem and a cycle; the cycle is iterated forever if it is entered by a state that results from executing the stem. Notice that in general, the stem and the cycle can contain the unrolling of one or more loops in the program.

3. Binary reachability analysis

In this section we describe the design and an implementation of binary reachability analysis. We proceed as follows:

1. We give a characterization of R_I^+ by the least fixpoint of a function F.
2. Guided by the structure of F, we show a transformation on P. This transformation creates a new program $\widehat{P}$ that implements F. This means that the least fixpoint of F is equivalent to the set of reachable states in $\widehat{P}$.
3. We describe a transformation of $\widehat{P}$ that creates $\widehat{P}_T$. An error location in $\widehat{P}_T$ is not reachable if and only if the inclusion $R_I^+ \subseteq T$ holds. This allows us to use a temporal safety prover to check the validity of T.
4. Finally, we describe how TERMINATOR transforms error paths found by the safety checker in $\widehat{P}_T$ to an input for TERMINATOR's rank function synthesis engine.

3.1 Fixpoint characterization

We first define a function F whose least fixpoint is R_I^+. The domain of F consists of binary relations over states of the given program. The least element is the transition relation restricted to initial states.

$$\bot \triangleq \{(s_1, s_2) \mid s_1 \in I \wedge (s_1, s_2) \in R\}$$

Before formalizing F, we define an auxiliary function $\mathsf{id}_{(2)}$ that restricts the identity relation over the program states to the image of its input relation, formally,

$$\mathsf{id}_{(2)}(X) \triangleq \{(s_2, s_2) \mid \exists s_1.\ (s_1, s_2) \in X\}.$$

Let $\circ$ denote the relational composition operator:

$$X \circ Y \triangleq \{(s_1, s_3) \mid \exists s_2.\ (s_1, s_2) \in X \wedge (s_2, s_3) \in Y\}.$$

The function F takes a binary relation X as input. It returns the relational composition of the union of X and the identity relation restricted to the second component, with the transition relation R of the program.

$$F(X) \triangleq (X \cup \mathsf{id}_{(2)}(X)) \circ R$$

In effect F either copies the righthand component of X into the left before passing it to R, or else it simply passes X itself to R.

THEOREM 1. *The binary reachability relation R_I^+ of the program P is equal to the least fixpoint of the function F on the domain of binary relations with the least element $\bot$:*

$$R_I^+ = \mathsf{lfp}(F, \bot).$$

3.2 Reachability characterization

We define a transformation $\widehat{P}$ of the program P and an equivalence relation $\simeq$ between pairs of states in P and states of $\widehat{P}$. The transformation reflects the structure of the function F, expressed in P's programming language. We will establish a connection between the least fixpoint of F over $\bot$ and the set of reachable states of $\widehat{P}$.

Let $V = \{\mathsf{v}_1, \ldots, \mathsf{v}_n, \mathsf{pc}\}$ be the set of program variables in P, including the program counter. The set of variables $\widehat{V}$ of transformed program $\widehat{P}$ contains V and a duplicate set of variables $‘V = \{‘\mathsf{v}_1, \ldots, ‘\mathsf{v}_n, ‘\mathsf{pc}\}$. These are used in $\widehat{P}$ to record values of V in previous states (We will discuss how TERMINATOR create these variables for pointer and records expressions in Section 4.2).

We say that a state $\widehat{s}$ in $\widehat{P}$ is equivalent to a pair of states (s_1, s_2) in P, written $\widehat{s} \simeq (s_1, s_2)$, if the following conditions hold.

$$\begin{array}{ll} \widehat{s}(‘\mathsf{v}_1) = s_1(\mathsf{v}_1) & \widehat{s}(\mathsf{v}_1) = s_2(\mathsf{v}_1) \\ \cdots & \cdots \\ \widehat{s}(‘\mathsf{v}_n) = s_1(\mathsf{v}_n) & \widehat{s}(\mathsf{v}_n) = s_2(\mathsf{v}_n) \\ \widehat{s}(‘\mathsf{pc}) = s_1(\mathsf{pc}) & \widehat{s}(\mathsf{pc}) = s_2(\mathsf{pc}) \end{array}$$

```
 1 int Ack(int x, int y)
 2 {
 3     if (x>0) {
 4         int n;
 5         if (y>0) {
 6             y--;
 7             n = Ack(x,y);
 8         } else {
 9             n = 1;
10         }
11         x--;
12         return Ack(x,n);
13     } else {
14         return y+1;
15     }
16 }
17
18 void main()
19 {
20     int x = nondet();
21     int y = nondet();
22
23     int * p = &y;
24     int * q = &x;
25     bool b = true;
26
27     while(x<100 && 100<y && b)
28     {
29         if (p==q) {
30             int k = Ack(nondet(),nondet());
31             (*p)++;
32             while((k--)>100)
33             {
34                 if (nondet()) {p = &y;}
35                 if (nondet()) {p = &x;}
36                 if (!b) {k++;}
37             }
38         } else {
39             (*q)--;
40             (*p)--;
41             if (nondet()) {p = &y;}
42             if (nondet()) {p = &x;}
43         }
44         b = nondet();
45     }
46 }
```

Figure 4. Example program. `nondet()` is used to represent non-deterministically chosen integers and Booleans

ℓ	well-founded binary relations T_k^ℓ
7	$T_0^7(s,t) \triangleq$ `false`
	$T_1^7(s,t) \triangleq t(\mathtt{y}) > -1 \wedge t(\mathtt{y}) \leq s(\mathtt{y}) - 1$
12	$T_0^{12}(s,t) \triangleq$ `false`
	$T_1^{12}(s,t) \triangleq t(\mathtt{x}) > -1 \wedge t(\mathtt{x}) \leq s(\mathtt{x}) - 1$
28	$T_0^{28}(s,t) \triangleq$ `false`
	$T_1^{28}(s,t) \triangleq t(\mathtt{y}) > 100 \wedge t(\mathtt{y}) \leq s(\mathtt{y}) - 1$
	$T_2^{28}(s,t) \triangleq t(\mathtt{x}) < 100 \wedge t(\mathtt{x}) \geq s(\mathtt{x}) + 1$
33	$T_0^{33}(s,t) \triangleq$ `false`
	$T_1^{33}(s,t) \triangleq t(\mathtt{k}) > 100 \wedge t(\mathtt{k}) \leq s(\mathtt{k}) - 1$

Figure 5. The termination argument (a union of well-founded binary relations T_k^ℓ between states at the same cutpoint location ℓ) incrementally constructed and then checked by TERMINATOR, thus proving the termination of the program in Figure 4. TERMINATOR starts with T_0^ℓ, the empty relation denoted by `false`.

```
                          L:  if (nondet()) {
                                  ‘v1 = v1;
                                  ...
                                  ‘vn = vn;
L: stmt     ⟹                     ‘pc = L;
                              } else {
                                  skip;
                              }
                              stmt
```

Figure 6. Transformation used to construct $\widehat{P}$ from P.

Let $\widehat{I}$ be the set of initial states of $\widehat{P}$ defined as follows.

$$\widehat{I} \triangleq \{\widehat{s} \mid \exists s_0 \in I.\ \widehat{s} \simeq (s_0, s_0)\}$$

See Figure 6, which shows a transformation on program statements. We construct $\widehat{P}$ by applying this transformation on each statement of the program P. Let $\mathsf{post}_{\widehat{P}}$ denote the post operator of the program $\widehat{P}$.

There is an analogy between transformed statements and the structure of the function F:

- The assignment statements in Figure 6 correspond to the application $\mathsf{id}_{(2)}(X)$,
- The `if`-conditional with non-deterministic choice between the branches corresponds to the union $X \cup \mathsf{id}_{(2)}(X)$,
- The relational composition with R is reflected by i) having the original statement of the program P after the conditional and ii) the connection between states $\widehat{s}$ and pairs of states (s_1, s_2).

We formalize the analogy in the following theorem.

THEOREM 2. *The least fixpoint of the function F on the domain of binary relations with the least element $\bot$ is equal to the set of states of the transformed program $\widehat{P}$ reachable after at least one step:*

$$\mathsf{lfp}(F, \bot) = \mathsf{post}^+_{\widehat{P}}(\,\widehat{I}\,).$$

The equality holds under the assumption that we identify a state $\widehat{s}$ with a pair of states (s_1, s_2) if they are $\simeq$-equivalent.

The statement of Theorem 2 involves the following technicalities:

- We assume that there are no statements in P whose destination location is the initial location of P. This is because we do not instrument the first instruction in P in order to model $\bot$. The tranformation will not be correct, however, if this first instruction is reachable later during the program's execution.
- We assume that the operator $\mathsf{post}_{\widehat{P}}$ treats the compound statements

```
L: if (nondet()) {...} stmt
```

of the program $\widehat{P}$ as monolithic ones. This means that the intermediate states at locations of $\widehat{P}$ added due to the transformation are not considered to be elements of $\mathsf{post}^{+}_{\widehat{P}}(\widehat{I}\ )$.

We can now implement the binary reachability analysis in three steps:

- Create the transformed program $\widehat{P}$,
- Compute the set of reachable states $\mathsf{post}^{+}_{\widehat{P}}(\widehat{I}\ )$, and
- Check the inclusion between the computed set and T.

3.3 Reachability characterization

In practice, we would like to stop the reachability computation as soon as it becomes evident that the inclusion does not hold. In this section we describe an additional transformation, applied on the program $\widehat{P}$, that addresses this issue.

The additional transformation takes the program $\widehat{P}$ and produces $\widehat{P}_T$ by replacing each compound statement of $\widehat{P}$ (except the initial statement):

```
    L:  if (nondet()) {...} stmt
```

by the statement:

```
    L:  if (!(T_L)) { ERROR: skip; }
        if (nondet()) {...} stmt
```

To construct the Boolean expression $T_{\mathtt{L}}$ we first assume that the relation T is represented by an assertion over variables ‘V and V, which denote the values of the program variables in the first and the second component of the pairs $(s_1, s_2) \in T$. Note that the program counter variable `pc` does not appear in the program text of C programs. Hence, we cannot insert the assertion T into the program text directly. To overcome this, we use the expression $T_{\mathtt{L}}$ that is obtained by substituting `L` for `pc` in T. For example, for $T = ((\texttt{‘pc} = \texttt{L12} \wedge \texttt{pc} = \texttt{L12}) \implies (\texttt{x} > -1 \wedge \texttt{x} < \texttt{‘x})$ and location `L28` we obtain the following conditional:

```
    L28: if ( ! (! ( ‘pc==L12 && L28==L12 )
                 || ( x>-1 && x<‘x ))
             )
         {
             ERROR: skip;
         }
```

We will revisit this example in Section 4.1.

THEOREM 3. *The inclusion $R_I^{+} \subseteq T$ holds if and only if the location* ***ERROR*** *is not reachable in the program $\widehat{P}_T$.*

Now, we can apply a temporal safety checker on the program $\widehat{P}_T$ to prove the non-reachability of the location `ERROR`. For safety checkers based on counterexample-guided abstraction refinement, the formulation of $\widehat{P}_T$ (particularly with several optimizations to be described in Section 4) gives ample opportunity for good abstractions.

3.4 Analyzing error paths of $\widehat{P}_T$

We assume that a temporal safety checker can produce an error path if the location `ERROR` is reachable. Next, we describe the interpretation of such an error path π in the context of TERMINATOR's algorithm, which we have described in Section 2.

We need to extract a counterexample ρ to the inclusion $R_I^{+} \subseteq T$ from the error path π. We observe that π must, at some point, traverse through the positive branch of the conditional added by the transformation in Figure 6. We split π at the latest appearance of such statement into *stem* and *cycle*. We then remove all statements that were added by the program transformation from the stem and the cycle. Let R_{stem} and R_{cycle} be the transition relations of the stem and the cycle, respectively. We produce the relation ρ, see Figure 2, as follows.

$$\rho \triangleq \{(s_2, s_3) \mid \exists s_1 \in I.\ (s_1, s_2) \in R_{stem} \wedge (s_2, s_3) \in R_{cycle}\}$$

The resulting relation ρ is represented as a conjunction of atomic assertions computed via a symbolic simulation of the path in P.

We assume that the safety checker also outputs an aliasing configuration between pointer variables that together with the error path π witnesses the reachability of the location `ERROR` in $\widehat{P}_T$. We encode this information into ρ by an additional conjunction.

If the rank function synthesis step fails for the relation ρ, the stem and the cycle constitute a possible counterexample to termination of the program P.

Example. Consider the following simple program:

```
1    void main() {
2        int x = nondet();
3        int * p = nondet();
4        if (p==&x) {
5            do {
6                x--;
7            } while(*p>0);
8        }
9    }
```

With a termination argument $T^5 = \texttt{false}$ the (false) counterexample to termination will be the $stem = 2 \to 3 \to 4 \to 5$ and $cycle = 6 \to 7 \to 5$.

The rank function synthesis engine knows nothing about pointers (i.e. the meaning of `*p`). However, if we symbolically simulate this path while constructing the mathematical relationship, we can see that `*p` and `x` represents the same value. Therefore, when constructing the mathematical relation representing this path, we can simply use the same mathematical variable. Let v_0 and v_1 be the values that both `*p` and `x` have before and after the execution of the stem/cycle statements. We can define the stem and cycle represent mathematical relations R_{stem} and R_{cycle} as:

$$\begin{aligned} R_{stem}(v_0, v_1) &\triangleq \ \texttt{true} \\ R_{cycle}(v_0, v_1) &\triangleq \ v_1 = v_0 - 1 \wedge v_1 > 0 \end{aligned}$$

R_{cycle} is well-founded, where the ranking function is v. This is mapped back to a relation over program variables by choosing either `x` or `*p`. That is, we could either use $T^5(s,t) \triangleq t(\mathtt{x}) > 0 \wedge t(\mathtt{x}) < s(\mathtt{x})$ or $T^5(s,t) \triangleq t(*\mathtt{p}) > 0 \wedge t(*\mathtt{p}) < s(*\mathtt{p})$ for the refinement of the termination argument.

Note that, even if we incorrectly assume that two C expressions always alias, this does not cause an unsoundness in TERMINATOR. The only constraint on T^5 is that it is well-founded. If we use the argument $T^5(s,t) \triangleq t(\mathtt{x}) > 0 \wedge t(\mathtt{x}) < s(\mathtt{x})$, the binary reachability will find cases where `x` and `*p` do not alias if such examples exist.

4. Optimizations

In this section we describe several optimizations that TERMINATOR applies during the program transformation described in Section 3. The first optimization exploits the fact that we can construct and check termination arguments for one location at a time. The remaining optimizations prune away executions of $\widehat{P}_T$ that the temporal safety checker does not need to consider when trying to proving the non-reachability of the location `ERROR`. We implement these optimizations as additional program transformations and perform them during the construction of the program $\widehat{P}_T$.

4.1 Specialization of $\widehat{P}_T$

The termination argument T for the program P is a conjunction of termination arguments T^{ℓ} for each cutpoint ℓ. Each termination

```
 1  void main()
 2  {
 3    int x,y;
 4
 5 S: if(y>=1) {
 6      while(x>=0)
 7      {
 8 L:      x = x+y;
 9      }
10    }
11  }
```

Program P

```
0.1  int ‘x, ‘y, ‘pc;
0
1    void main()
2    {
3      int x, y;
4
4.1    ‘x = x;
4.2    ‘y = y;
4.3    ‘pc = L0;
5    S: if (y>=1) {
6         while (x>=0)
7         {
8.1  L:     if (‘pc==L
8.2            && L==L
8.2            && !( x>=0
8.3                 && x<=‘x-1
8.4                 )
8.5            ) {
8.6           ERROR: ;
8.7         }
8.8         if (nondet()){
8.9           ‘x = x;
8.10          ‘y = y;
8.11          ‘pc = L;
8.11        }
8           x = x+y;
9         }
10      }
11    }
```

Program $\widehat{P}_T$

Figure 7. Example of the program transformation for checking the binary reachability query ($\texttt{‘pc} = \texttt{L} \wedge \texttt{pc} = \texttt{L}) \implies (x \geq 0 \wedge x \leq \text{‘}x - 1)$ at the location L.

argument T^ℓ is of the form

$$(\texttt{‘pc} = \ell \wedge \texttt{pc} = \ell) \implies (T_1^\ell \vee \cdots \vee T_n^\ell).$$

Assume that TERMINATOR is processing the location ℓ. When the program transformation creates a statement of the program $\widehat{P}_T$ at the location $\ell' \neq \ell$ then the corresponding expression T is equivalent to false, since the conjunction $T^\ell \wedge \texttt{pc} = \ell'$ is valid. Thus, we can drop the statement

```
if ( ! T_L ) { ERROR: skip; }
```

at all locations different from ℓ (see the example in Section 3.3). Furthermore, we can also drop the statement

```
if (nondet()) { ... }
```

at these locations, since the states $\widehat{s}$ that are created by taking the positive branch of the above conditional at location $\ell' \neq \ell$ cannot cause the computation to reach the location ERROR, because $\widehat{s} \notin T^\ell$.

We apply this optimization on all examples in the remainder of this section. See Figure 7 for an example of specialization of $\widehat{P}_T$.

4.2 Pre-variables for C programs

The program transformation described in Section 3 implicitly requires that we create a pre-variable for every heap and stack location addressable using C expressions from the program variables in scope. However, in practice this is impossible.

Assume that TERMINATOR is processesing a termination argument for a given cutpoint. For each variable x of scalar type, *i.e.*, int, char, long, etc., in scope of the cutpoint we count the number n of dereference operators in the type definition of the variable x. For example, if x is defined as int **x; then $n = 2$. For each $0 \leq i \leq n+1$ we create the following pre-variable.

$$\texttt{‘}\underbrace{\texttt{p}\ldots\texttt{p}}_{i\text{ times}}\texttt{x}$$

and we insert the assignment statement

$$\texttt{‘}\underbrace{\texttt{p}\ldots\texttt{p}}_{i\text{ times}}\texttt{x = }\underbrace{\texttt{*}\ldots\texttt{*}}_{i\text{ times}}\texttt{x;}$$

in the conditional from Figure 6, during the program transformation.

```
 1 void main()
 2 {
 3     int ...;
 4 L0:
 5 L1: while (...) {
 6         ...
 7 L2:     while (...) {
 8             ...
 9         }
10         ...
11     }
12     if (...)
13          goto L0;
14 }
```

Figure 8. Example program with three cutpoints L0, L1, and L2.

Note that it is sound for TERMINATOR not to create duplicate-variables for locations addressable in the original C program, this only serves to make TERMINATOR more complete.

TERMINATOR also creates pre-variables for field access expressions, say x->f, that appear in the termination argument. It creates a pre-variable ‘x_f and the corresponding assignment statement ‘x_f = x->f;.

4.3 Structured programs

When proving the termination argument for a cutpoint ℓ within a structured program it is not necessary to prove termination at ℓ for executions that leave the ℓ loop and then return—these executions will be covered when proving termination of the outer loop. To implement this optimization we insert

```
if (‘pc == L) { exit(); }
```

into the source code of the transformed program at exit points of the loop that ℓ represents.

We illustrate this optimization on Figures 8 and 9. Assume that we are in the process of inferring a termination argument for the cutpoint that corresponds to the while-loop at the location L2. By inserting the conditional statement at line 9.1 we exclude from consideration states $\widehat{s}$, where $\widehat{s}(\texttt{pc}) = \texttt{L2}$ that appear on computations leaving the inner loop and coming back to the location L2.

4.4 Weak binary reachability

In large programs, paths to a cutpoint can be very long. These paths may execute many instructions that are not relevant to the termination analysis at the cutpoint. We observed examples of such paths on some dispatch routines when applying TERMINATOR. TERMINATOR can abstract away the prefixes of such paths in two different ways:

1. It could ignore all program statements that are executed between the start of the main function and the call to the function containing the cutpoint under consideration.

```
 1 void main()
 2 {
 3     int ...;
 4 L0:
 5 L1: while (...) {
 6         ...
 7 L2:     while (...) {
 7.1           if ( ! T_L ) { ... }
 7.2           if (nondet()) {...; ‘pc = L2;}
 8             ...
 9         }
 9.1       if (‘pc == L2) { exit(); }
10         ...
11     }
12     if (...)
13         goto L0;
14 }
```

Figure 9. Transformed program for cutpoint `L2` with the `return`-statement at line 9.1 added due to optimization.

2. It could also ignore all program statements within the function containing the cutpoint that appear between the first statement and the statement that corresponds to the cutpoint.

Note that these optimizations must ensure that values that would be initialized during the code that is being skipped must still be initialized (with non-determinatic values) in the abstraction. In most temporal safety checkers this is automatic. We introduce two approximations R_1^+ and R_2^+ of the binary reachability relation R_I^+ that correspond to the above abstractions. Then, we define the corresponding program transformations that given a program $\widehat{P}_T$ create programs $\widehat{P}_T^1$ and $\widehat{P}_T^2$ which implement the first and the second approximation respectively.

Let ℓ_{entry} be the entry location of the function containing the cutpoint that we are analyzing. We define the first approximation R_1^+ as follows.

$$\begin{aligned} R_1^+ \triangleq \{(s_1, s_2) \mid \exists s_0.\ & s_0(\mathsf{pc}) = \ell_{entry} \wedge \\ & (s_0, s_1) \in R^* \wedge \\ & (s_1, s_2) \in R^+ \wedge \\ & s_1(\mathsf{pc}) = s_2(\mathsf{pc}) = \ell\} \end{aligned}$$

The second approximation R_2^+ is the transitive closure of R restricted to the cutpoint. Note that, if R_I^+ is restricted to states where the pc equals ℓ, $R_I^+ \subseteq R_1^+ \subseteq R_2^+$.

The program $\widehat{P}_T^1$, which represents the approximation R_1^+, is obtained from $\widehat{P}$ by inserting a call to the function containing the cutpoint as the first instruction in the function `main`. We illustrate $\widehat{P}_T^1$ in Figure 10(b).

The program $\widehat{P}_T^2$, which represents the approximation R_2^+, is obtained from $\widehat{P}_T^1$ inserting a `goto`-statement at the beginning of the function containing the cutpoint. The destination label of the `goto`-statement is the cutpoint location. We illustrate $\widehat{P}_T^2$ in Figure 10(c).

TERMINATOR first analyzes $\widehat{P}_T^2$. If $\widehat{P}_T^2$'s error location is not reachable then TERMINATOR proceeds with the next cutpoint. If, however, an error path is found, then TERMINATOR switches its focus to $\widehat{P}_T^1$. If $\widehat{P}_T^1$'s error location is not reachable then, again, TERMINATOR proceeds with the next cutpoint. If an error path in $\widehat{P}_T^1$ is found then TERMINATOR must revert to the original $\widehat{P}_T$. Note that predicates found by a predicate-abstraction based temporal safety checker during the analysis of $\widehat{P}_T^2$ can be reused with $\widehat{P}_T^1$, etc.

```
 1 void main()
 2 {
 3   ...
 4   g();
 5   ...
 6 }
 7
 8 void g()
 9 {
10   ...
11   f();
12   ...
13 }
14
15 void f()
16 {
17   ...
18 L: while
19   ...
20 }
```

(a)

```
1  void main()
2  {
2.1  f();
2.2  exit;
3    ...
4    g();
5    ...
6  }
7
8  void g()
9  {
10   ...
11   f();
12   ...
13 }
14
15 void f()
16 {
17   ...
18 L: while
19   ...
20 }
```

(b)

```
1    void main()
2    {
2.1    f();
2.2    exit;
3      ...
4      g();
5      ...
6    }
7
8    void g()
9    {
10     ...
11     f();
12     ...
13   }
14
15   void f()
16   {
16.1   goto L;
17     ...
18     L: while
19     ...
20   }
```

(c)

Figure 10. (a) Example program with a cutpoint at location `L` in the function `f`. **(b)** Transformed program $\widehat{P}_T^1$ for cutpoint `L`. The safety checker does not consider the function `g`. **(c)** Transformed program $\widehat{P}_T^2$ for cutpoint `L`. The safety checker considers neither the function `g` nor the initial part of `f`.

```
1  void main()
2  {
3      int x=nondet(), y=nondet(), z=nondet();
4      if (y>0) {
5          do {
6              if (nondet()) {
7                  x = x + y;
8              } else {
9                  z = x - y;
10             }
11         } while (x<y && y<z);
13     }
14 }
```

Figure 11. An example program

This optimization does not affect TERMINATOR's precision: TERMINATOR produces counterexamples to termination only when analyzing $\widehat{P}_T$. This optimization speeds up the analysis of programs that terminate, and slows down the checking of loops and recursive functions that do not guarantee termination.

5. A complete example

In this section we work through a TERMINATOR-style termination proof search on the program in Figure 11. As there is a only one loop we need simply to find and prove a termination argument for one program location (program location 5).

First iteration. We start the proof search with the termination argument T^5 initialized to the empty argument, i.e. $T^5(s, t) \triangleq$ `false`. In order to check the validity of the termination argument T^5, the binary reachability procedure will try to prove the safety of a program that it constructs. Note that, in this case, weak binary reachability is not powerful enough. This is due to the fact that we must know `y>0` during the analysis of the loop. For this reason we will skip directly to strong binary reachability. Recall that our implementation of binary reachability analysis produces a new program and performs reachability analysis on it. The new program in this case is:

```
0.1  int ’pc = 0;
0.2  int ’x, ’y, ’z;
1    void main()
2    {
3        int x=nondet(), y=nondet(), z=nondet();
4        if (y>0) {
5            do {
5.1              if (’pc==5) {
5.2                  if (!(false)) {
5.3                      ERROR: skip;
5.4                  }
5.5              }
5.6              if (’pc==0) {
5.7                  if (nondet()) {
5.8                      ’x = x;
5.9                      ’y = y;
5.10                     ’z = z;
5.11                     ’pc = 5;
5.12                 }
5.13             }
6                if (nondet()) {
7                    x = x + y;
8                } else {
9                    z = x - y;
10               }
11           } while (x<y && y<z);
13       }
14   }
```

Note that $T^5 = \texttt{false}$ is used in the conditional at line 5.2. A temporal safety checker will find that the program location ERROR (i.e. location 5.3) is reachable, with one possible counterexample being: $3 \rightarrow 4 \rightarrow 5 \rightarrow 5.1 \rightarrow 5.6 \rightarrow 5.7 \rightarrow 5.8 \rightarrow 5.9 \rightarrow 5.10 \rightarrow 5.11 \rightarrow 5.12 \rightarrow 5.13 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 11 \rightarrow 5 \rightarrow 5.1 \rightarrow 5.2 \rightarrow 5.3$. This counterexample can be broken up into a representative stem and cycle. We perform this by splitting the stem from the cycle at the occurence of line 5.8, and then removing all of the line numbers introduced by instrumentation. This leaves us with:

stem $= 3 \rightarrow 4 \rightarrow 5$
cycle $= 6 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 11 \rightarrow 5$

The stem and cycle represent the mathematical relations R_{stem} and R_{cycle}:

$$\begin{array}{rcl} R_{stem}((x_0,y_0,z_0),(x_1,y_1,z_1)) & \triangleq & y_1 > 0 \\ & \wedge & x_1 = x_0 \\ & \wedge & z_1 = z_0 \end{array}$$

$$\begin{array}{rcl} R_{cycle}((x_0,y_0,z_0),(x_1,y_1,z_1)) & \triangleq & x_1 = x_0 + y_0 \\ & \wedge & z_1 = z_0 \\ & \wedge & y_1 = y_0 \\ & \wedge & x_1 < y_1 \\ & \wedge & y_1 < z_1 \end{array}$$

The counterexample that we pass to the rank function synthesis engine is the relation

$$\rho_1(s_1,s_2) \triangleq \exists s_0 \in I.\ (s_0,s_1) \in R_{stem} \wedge (s_1,s_2) \in R_{cycle}$$

The rank function synthesis engine can prove this relation well-founded, and returns the following ranking relation computed during the proof which over-approximates ρ_1:

$$T^5_1(s,t) \triangleq t(\mathtt{x}) < t(\mathtt{y}) \wedge t(\mathtt{x}) \geq s(\mathtt{x}) + 1$$

That is, we know that T^5_1 is well-founded and that $\rho_1 \subseteq T^5_1$—think of T^5_1 as the *generalization* or *core-reason* as to why the path represented by ρ_1 is a spurious termination counterexample. T^5_1 is then added to our termination argument:

$$T^5(s,t) \triangleq T^5_1(s,t) \vee \texttt{false}$$

Second iteration. We then start the process again: we are trying to prove termination at program location 5 and the current termination argument, T^5, equals $T^5(s,t) \triangleq T^5_1(s,t) \vee \texttt{false}$ and where:

$$T^5_1(s,t) \triangleq t(\mathtt{x}) < t(\mathtt{y}) \wedge t(\mathtt{x}) \geq s(\mathtt{x}) + 1$$

The binary reachability procedure will produce the following program:

```
0.1  int ’pc = 0;
0.2  int ’x, ’y, ’z;
1    void main()
2    {
3        int x=nondet(), y=nondet(), z=nondet();
4        if (y>0) {
5            do {
5.1              if (’pc==5) {
5.2                  if (!( (x<y && x>=’x)
5.3                      || false
5.4                      ));
5.5              }
5.6              if (’pc==0) {
5.7                  if (nondet()) {
5.8                      ’x = x;
5.9                      ’y = y;
5.10                     ’z = z;
5.11                     ’pc = 5;
5.12                 }
5.13             }
6                if (nondet()) {
7                    x = x + y;
8                } else {
9                    z = x - y;
10               }
11           } while (x<y && y<z);
13       }
14   }
```

A temporal safety checker will find that ERROR is reachable, and will produce the following counterexample:

stem $= 3 \rightarrow 4 \rightarrow 5$
cycle $= 6 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 5$

The representative relations R_{stem} and R_{cycle} are defined as:

$$\begin{array}{rcl} R_{stem}((x_0,y_0,z_0),(x_1,y_1,z_1)) & \triangleq & y_1 > 0 \\ & \wedge & x_1 = x_0 \\ & \wedge & z_1 = z_0 \end{array}$$

$$\begin{array}{rcl} R_{cycle}((x_0,y_0,z_0),(x_1,y_1,z_1)) & \triangleq & z_1 = x_0 - y_0 \\ & \wedge & x_1 = x_0 \\ & \wedge & y_1 = y_0 \\ & \wedge & x_1 < y_1 \\ & \wedge & y_1 < z_1 \end{array}$$

We can define a second counterexample relation, ρ_2, as:

$$\rho_2(s_1,s_2) \triangleq \exists s_0 \in I.\ (s_0,s_1) \in R_{stem} \wedge (s_1,s_2) \in R_{cycle}$$

ρ_2 is also well-founded—the ranking relation produced is:

$$T^5_2(s,t) \triangleq t(\mathtt{y}) < t(\mathtt{z}) \wedge t(\mathtt{z}) \leq s(\mathtt{z}) - 1$$

We again refine the termination argument:

$$T^5(s,t) \triangleq T^5_2(s,t) \vee T^5_1(s,t) \vee \texttt{false}$$

Third iteration. We are now ready to try and prove the validity of the termination argument:

$$T^5(s,t) \triangleq T_2^5(s,t) \vee T_1^5(s,t) \vee \texttt{false}$$

The program that the binary reachability procedure produces is:

```
0.1  int ’pc = 0;
0.2  int ’x, ’y, ’z;
1    void main()
2    {
3        int x=nondet(), y=nondet(), z=nondet();
4        if (y>0) {
5            do {
5.1              if (’pc==5) {
5.2                  if (!( (y<z && z<=’z)
5.3                      || (x<y && x>=’x)
5.4                      || false
5.5                      ));
5.6              }
5.7              if (’pc==0) {
5.8                  if (nondet()) {
5.9                      ’x = x;
5.10                     ’y = y;
5.11                     ’z = z;
5.12                     ’pc = 5;
5.13                 }
5.14             }
6                if (nondet()) {
7                    x = x + y;
8                } else {
9                    z = x - y;
10               }
11           } while (x<y && y<z);
13       }
14   }
```

The location `ERROR` is not reachable in this program and, therefore, the final termination argument for program location 5 is valid:

$$T^5(s,t) \triangleq T_2^5(s,t) \vee T_1^5(s,t) \vee \texttt{false}$$

where

$$T_2^5(s,t) \triangleq t(\mathtt{y}) < t(\mathtt{z}) \wedge t(\mathtt{z}) \leq s(\mathtt{z}) - 1$$

$$T_1^5(s,t) \triangleq t(\mathtt{x}) < t(\mathtt{y}) \wedge t(\mathtt{x}) \geq s(\mathtt{x}) + 1$$

Because there is only one cutpoint in this program, we can reason that the final whole-program termination argument is:

$$R_I^+ \subseteq T_2^5 \cup T_1^5 \cup T^{\mathrm{DIFF}}$$

6. Experimental results

We have integrated Terminator into the Static Driver Verifier (SDV) formal verification tool [2, 21], which is distributed as a part of the Microsoft Windows Device Driver Development Kit. SDV uses a temporal safety checker to prove properties of device drivers. SDV provides a set of safety properties together with an abstract model of the environment in which device drivers execute. We were able to reuse SDV's environment model in the new integration after two minor modifications. The new property that our SDV/Terminator integration proves of a device driver is that dispatch routines, when called, always return back to the environment's call-site. The environment uses non-deterministic choices to model the possibility of calling any of the device driver's dispatch routines, providing coverage for all of the dispatch routines in the device driver.

We applied SDV/Terminator to the standard 23 Windows OS device drivers used within Microsoft to test SDV. Each of these 23 drivers provides from 5 to 10 dispatch routines.

The results of these experiments are displayed in Figure 12. The results indicate the scalability of Terminator to programs

Driver	Run-time (seconds)	True bugs found	False bugs reported	Lines of code	Cutpoint set size
1	12	0	1	1K	3
2	8	0	0	1K	8
3	410	0	1	8K	26
4	1475	0	1	7.5K	24
5	123292	1	11	5.5K	50
6	196	1	3	5K	29
7	4174	0	0	8K	23
8	210	0	11	5K	27
9	1294	0	5	6K	38
10	158	0	0	8K	21
11	13	0	0	2.5K	6
12	204	0	0	2.5K	16
13	257	1	1	7.5K	26
14	5	0	0	1K	2
15	141	0	1	6.5K	18
16	22	0	0	1.5K	2
17	800	1	6	4K	35
18	1503	1	0	6.5K	31
19	209	0	3	3K	28
20	4099	0	2	10K	63
21	1461	1	4	16K	56
22	114762	0	5	34K	65
23	158746	2	10	35K	75

Figure 12. Results of experiments using an integration of Terminator with the Windows Static Driver Verifier[21] product (SDV) on the standard 23 Windows OS device drivers used to test SDV. Each device driver exports from 5 to 10 dispatch routines, all of which must be proved terminating.

with up to 35,000 lines of code. In practice TERMINATOR spends effectively 100% of its time in the binary reachability analysis. For this reason, the results in Figure 12 also demonstrate the accuracy and scalability of TERMINATOR's binary reachability analysis.

The termination violations reported by TERMINATOR are split into two categories in Figure 12: *true bugs* and *false bugs*. The false bugs are due to inaccuracies in TERMINATOR's analysis, which can be categorized accordingly:

Heaps: A majority of the false bugs were caused by loops in which the program is walking a linked-list data structure. For example:

```
do { f(p); p = p->next} while (p != NULL);
```

TERMINATOR currently does not have a rank function synthesis mechanism that accurately models operations occurring in the program that modify the shape of the heap. TERMINATOR's rank function synthesis module can determine that a counter pointed to by a pointer is decremented, but it is unable to reason effectively about the effects of instructions on heap-sizes.

Note that, in cases where the drivers are using high-level operations on kernel-level data-structures (such as queues and stacks), these were not reported as bugs by TERMINATOR. This is due to the fact that we were able to model the size of the structures using arithmetic in the SDV environment model.

Bit operations: Our implementation of binary reachability analysis overapproximates the meaning of the C bit operations such as `&`[1], meaning that TERMINATOR can return false counterexamples in cases where the termination condition requires a more precision treatment of these operations.

Note that in most cases the false bugs caused by linked-lists and bit operations are easily recognizable by the developer of the driver. Another interesting aspect is that we saw no false bugs due to inaccuracies in the environment model. This gives us hope that, with improvements to the handling of bit-vectors and heaps, TERMINATOR could achieve an unprecedented level of accuracy for automatic program verification.

The true bugs in Figure 12 are termination counterexamples found by TERMINATOR that have been confirmed as bugs by developers in the Windows kernel team. The bug in driver number 5 is the example used in Section 1 (Figure 1). In this case TERMINATOR returned a path with 2531 steps (formally, a sequence of 2531 statements). The path was from the start of the environment model's `main` function through the driver and into the loop in Figure 1.

7. Related work

Automatic program termination is a research topic dating back to Turing [24] and before. Techniques have been developed in a number of contexts, including term rewriting (e.g. [11, 16]) and logic and functional programming (e.g. [9, 18, 19]). Inspired by the success of automatic program verification for temporal safety properties (e.g. [2, 4, 17]), researchers have found a renewed interest in techniques for proving program termination of imperative programs (e.g. [6, 7, 10, 14]).

In summary, our work differs from the previous research in two ways:

- Our method and tool seems to be the first automatic program termination analysis that is successfully applied to industrial systems code.
- In the previous approaches for proving the termination of imperative programs, finding a ranking function is the main task (proving that the ranking function actually decreases in every computation step of the program is relatively easy). In our approach the main task is the check of the termination argument, for which we have designed a binary reachability analysis.

One of the earliest abstraction-based verification tools, SYNTOX [5] can check the termination of Pascal programs using greatest-fixpoint iteration. This is an alternative approach with a different flavor than ours. The problem here is to develop practical over-approximation techniques for greatest-fixpoint iteration.

There are other abstraction-based methods for termination that are targeted at more specialized settings (e.g. [1] or [25]). These approaches do not implement abstraction refinement, meaning that loss of information under a (fixed) abstraction cannot be recovered.

We finally come to the comparison with model checking for *finite-state* systems. It has been observed that for a given finite-state system, termination can be reduced to a safety property[2]. The algorithm in [3] makes this explicit using a program transformation, but this fact implicitly underlies all model checking algorithms for termination in finite-state systems. The idea is to search for a repeated occurrence of the same state on some trace. Interestingly, there exists a way in which our method generalizes the model checking algorithm for termination of finite-state systems. Namely, we can phrase the algorithm as the binary reachability analysis that checks the inclusion $R_I^+ \subseteq T$ for one particular binary relation T, fixed independently of the finite-state system; T consists of all pairs of different states.

$$T = \{(s_1, s_2) \mid s_1 \neq s_2\} \quad = \bigcup_{s_1 \neq s_2} \{(s_1, s_2)\}$$

The relation T is a finite union of well-founded relations (finite because of the limitation to finite-state systems; in fact, T is the largest relation with this property). Hence, termination can be shown by checking the inclusion of the binary reachability relation of the finite-state system in T, i.e. $R_I^+ \subseteq T$.

8. Conclusion

In this paper we have introduced a method and a tool, TERMINATOR, that supports the following set of features:

Scalability. As the experimental results demonstrate, TERMINATOR is able to analyze the termination of device driver dispatch routines with up to 35,000 lines of code. This is due to the fact that TERMINATOR's binary reachability analysis implements a form of counterexample-guided abstraction refinement that leverages the locality of each binary reachability query.

Applicability. TERMINATOR supports most of the language features required in at least one application area (device drivers): arbitrary loop nesting, side-effects and aliasing, function-pointers, etc. This is due to binary reachability which tracks these details independently.

Automation. TERMINATOR is completely automatic. It does not require the user to provide ranking functions or proof hints. This is due to TERMINATOR's counterexample-guided argument refinement mechanism, which leverages binary reachability.

Precision. TERMINATOR implements a precise path-sensitive and context-sensitive program analysis.

Counterexample generation. TERMINATOR provides counterexamples to failed termination proofs. This is, again, a feature that

[1] See [12] for more information on we might support these operations more accurately

[2] In contrast, for an infinite-state system, we have only shown that termination can be reduced to the *existence of* a safety property. The difficulty of an automatic proof is that the tool has to find that safety property.

is a direct consequence of Terminator's binary reachability analysis.

Terminator achieves this milestone by shifting the burden away from the construction of termination arguments and to the checking of termination arguments. Terminator constructs termination arguments which are the disjunction of (possibly many) simple well-founded relations. Each of these relations is drawn, on demand, from a simple and fast analysis on a single path through the program. Terminator's binary reachability analysis, on the other hand, must perform the arduous task of actually checking that the disjunction of arguments covers over all possible pairs of states within all possible traces through the program. The experiments that we have performed with Terminator show that this task can be solved with satisfying accuracy and scalability.

Future work In the future we would like to investigate ways in which binary reachability and Terminator's method of refinement for termination arguments can be used when proving liveness properties of concurrent programs. We would also like to investigate methods of accelerating the production of counterexamples in Terminator's binary reachability analysis using tools such as CBMC [8].

Terminator could potentially be used in ways beyond simply proving termination. For example, Syntox [5] is used to derive debugging information, namely, to derive states that must inevitably reach the error state, a property that can be phrased in terms of termination. Terminator's analysis could potentially be used in a similar fashion.

Acknowledgments We would like to thank the following people and groups for useful discussions regarding this work: Wolfgang Ahrendt, Domagoj Babic, Tom Ball, Clark Barrett, Andreas Blass, Aaron Bradley, Koen Claessen, Marsha Chechick, East London Massive, Jürgen Giesl, Alexey Gotsman, Yuri Gurevich, Arie Gurfinkel, Alan Hu, Joe Hurd, Shuvendu Lahiri, the formal methods group at the National Security Agency, John Matthews, Sriram Rajamani, Zvonimir Rakamaric, Mooly Sagiv, Mary Sheeran, David Wahlstedt, Angela Wallenburg, the Windows SDV team and Windows kernel team, Andrei Voronkov, and many others.

References

[1] I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *VMCAI'2005: Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *LNCS*. Springer, 2005.

[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys'06: European Systems Conference*, 2006.

[3] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *FMICS'02: Formal Methods for Industrial Critical Systems*, volume 66(2) of *ENTCS*, 2002.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI'2003: Programming Language Design and Implementation*, pages 196–207. ACM Press, 2003.

[5] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *PLDI'1993: Programming Language Design and Implementation*, pages 46–55. ACM Press, 1993.

[6] A. Bradley, Z. Manna, and H. Sipma. Linear ranking with reachability. In *CAV'05: Conference on Computer Aided Verification*, volume 3576 of *LNCS*. Springer, 2005.

[7] A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI'2005: Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *LNCS*. Springer, 2005.

[8] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS'04: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[9] M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.

[10] M. Colón and H. Sipma. Practical methods for proving program termination. In *CAV'2002: Computer Aided Verification*, volume 2404 of *LNCS*, pages 442–454. Springer, 2002.

[11] E. Contejean, C. Marché, B. Monate, and X. Urbain. Proving Termination of Rewriting with C*i*ME. In *Extended Abstracts of the 6th International Workshop on Termination, WST'03*, pages 71–73, June 2003.

[12] B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In *CAV'05: Conference on Computer Aided Verification*, 2005.

[13] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *SAS'2005: Static Analysis Symposium*, volume 3672 of *LNCS*, pages 87–101. Springer, 2005.

[14] P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI'2005: Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *LNCS*. Springer, 2005.

[15] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[16] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In *RTA'2004: Rewriting Techniques and Applications*, volume 3091 of *LNCS*, pages 210–220. Springer, 2004.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL'2004: Principles of Programming Languages*, pages 232–244. ACM Press, 2004.

[18] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL'2001: Principles of Programming Languages*, volume 36, 3 of *ACM SIGPLAN Notices*, pages 81–92. ACM Press, 2001.

[19] N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. TermiLog: A system for checking termination of queries to logic programs. In *CAV'97: Computer-Aided Verification*, LNCS, pages 444–447. Springer, 1997.

[20] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: Safety*. Springer, 1995.

[21] Microsoft Corporation. Windows Static Driver Verifier. Available at www.microsoft.com/whdc/devtools/tools/SDV.mspx, July 2004.

[22] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI'2004: Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *LNCS*, pages 239–251. Springer, 2004.

[23] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS'2004: Logic in Computer Science*, pages 32–41. IEEE, 2004.

[24] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *London Mathematical Society*, 42(2):230–265, 1936.

[25] E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In *ESOP'2003: European Symp. on Programming*, volume 2618 of *LNCS*, pages 204–222. Springer, 2003.