Proof Pearl: The Termination Analysis of TERMINATOR

Joe Hurd*

Computing Laboratory
Oxford University
joe.hurd@comlab.ox.ac.uk

Abstract. TERMINATOR is a static analysis tool developed by Microsoft Research for proving termination of Windows device drivers written in C. This proof pearl describes a formalization in higher order logic of the program analysis employed by TERMINATOR, and verifies that if the analysis succeeds then program termination logically follows.

1 Introduction

TERMINATOR [2] is a static analysis tool developed by Microsoft Research to prove termination of Windows device drivers written in C. The device drivers are typically thousands or tens of thousands of lines of code running at the privilege level of the kernel, and an infinite loop will cause the computer to freeze.

TERMINATOR works by modifying the program to reduce the termination problem into a safety property. Given a program location l and a finite set of well-founded relations $R_1, \ldots, R_n$ on program states at location l (l-states), TERMINATOR inserts the statement

```
already_saved_state := false;
```

at the beginning of the program, and the statements

```
if (already_saved_state) {
  if ¬(R_1 state saved_state ∨ ··· ∨ R_n state saved_state) {
    error("possible non-termination");
  }
}
else if (*) {
  saved_state := state;
  already_saved_state := true;
}
```

just before the program location l. A static analysis tool is run on this modified program to verify that there is no execution trace leading to the error statement. In the case of TERMINATOR, this verification step is performed by the SLAM

* Supported by a Junior Research Fellowship at Magdalen College, Oxford.

static analysis tool [1], also developed by Microsoft Research. Since the statement `if (*)` is interpreted as demonic non-deterministic choice, the error statement being unreachable guarantees that between the ith and jth time that program location l is reached, the l-state goes down in at least one of the well-founded relations $R_1, \ldots, R_n$.

If for every program location there exists a set of well-founded relations that allow this modification and check to succeed, then it is possible to conclude that the program must always terminate [6]. It is this logical step that is formally verified in the remainder of this paper. Section 2 presents a higher order logic formalization of programs and termination; Section 3 adds to the model the information generated by a successful TERMINATOR analysis; and Section 4 formally verifies that it is sufficient to guarantee program termination. Finally Section 5 extends the model to verify two optimizations that are implemented in the TERMINATOR tool. The HOL4 theorem prover [3] was used for this proof pearl, and all the theorems presented have been mechanically checked.[1]

Although irrelevant for the correctness proof, it is interesting to see how TERMINATOR automatically constructs the well-founded relations $R_1, \ldots, R_n$. The SLAM tool called by TERMINATOR works by Counter-Example Guided Abstraction and Refinement (CEGAR), and thus in the case that the property is false provides a concrete execution trace leading to an error statement. TERMINATOR starts with no relations,[2] repeatedly runs SLAM to generate error traces, and each time adds a well-founded relation that would rule it out. The well-founded relations are heuristically generated by an external tool called RANKFINDER [5]. The hope is that eventually enough well-founded relations are chosen that the error statement can be proven to be unreachable.

2 Formalizing Termination

Programs are formalized in higher order logic as nondeterministic state machines equipped with a function mapping states to program locations (this captures the intuition that the program counter is part of the state):

$$\begin{aligned}&(\text{'state,'location}) \text{ program} \equiv \\ &\quad <|\ \mathsf{states} : \text{'state} \to \text{bool};\ \mathsf{location} : \text{'state} \to \text{'location}; \\ &\qquad \mathsf{initial} : \text{'state} \to \text{bool};\ \mathsf{transition} : \text{'state} \to \text{'state} \to \text{bool}\ |> \ .\end{aligned}$$

Note that the 'state and 'location can be any higher order logic types, and in particular the states set can be infinite.

The set of all program locations is simply the range of the location function:

$$\mathsf{locations}\ p \equiv \mathsf{image}\ p.\mathsf{location}\ p.\mathsf{states}\ .$$

[1] The proof script can be downloaded from `http://www.gilith.com/research/papers/terminatorScript.sml`.

[2] Note that for program locations that are executed at most once, SLAM will be able to prove the error statement is unreachable even when there are no relations.

Well-formed programs must be closed w.r.t. their set of states, and their set of program locations must be finite:[3]

$$\begin{array}{l}\mathsf{programs} \equiv \\ \quad \{\ p : (\text{'state,'location})\ \mathsf{program}\ \mid \\ \qquad \mathsf{finite}\ (\mathsf{locations}\ p)\ \wedge\ p.\mathsf{initial} \subseteq p.\mathsf{states}\ \wedge \\ \qquad \forall s, s'.\ p.\mathsf{transition}\ s\ s' \Rightarrow s \in p.\mathsf{states} \wedge s' \in p.\mathsf{states}\ \}\ .\end{array}$$

A sufficient condition for a program p being terminating is that the $p.\mathsf{transition}$ relation is well-founded. However, this is too strong, and excludes terminating programs that have unreachable loops in their transition relation. Instead a notion of program execution traces is introduced:

$$\mathsf{traces}\ p \equiv \{\ t : \text{'state lazy\_list}\ \mid\ t_0 \in p.\mathsf{initial} \wedge \forall t_i, t_{i+1} \in t.\ p.\mathsf{transition}\ t_i\ t_{i+1}\ \}\ .$$

The type α lazy_list of possibly infinite lists is already defined in the HOL4 theorem prover; this work only required some extra constants to support syntactic constructs such as the above universal quantification over adjacent elements of the list. Now a program can be defined to terminate if it has no infinite execution traces:

$$\mathsf{terminates}\ p \equiv \forall t \in \mathsf{traces}\ p.\ \mathsf{finite}\ t\ .$$

3 The TERMINATOR Program Analysis

The previous section presented a simple formalization of programs and defined a termination predicate on them. This section completes the formalization of the main verification goal by defining what it means for a TERMINATOR program analysis to succeed.

At a particular location l of a program p, the result of a successful TERMINATOR analysis (as described in the Introduction) is formalized in higher order logic as

$$\begin{array}{l}\mathsf{terminator\_property\_at\_location}\ p\ l \equiv \\ \quad \exists R, n. \\ \qquad (\forall k \in \{0, \ldots, n-1\}.\ \mathsf{well\_founded}\ (R\ k))\ \wedge \\ \qquad \forall t \in \mathsf{traces}\ p.\ \forall x_i < x_j \in \mathsf{trace\_at\_location}\ p\ l\ t. \\ \qquad\quad \exists k \in \{0, \ldots, n-1\}.\ R\ k\ x_j\ x_i\ ,\end{array}$$

where $\mathsf{trace\_at\_location}\ p\ l\ t$ filters the execution trace t leaving only the states corresponding to the location l:

$$\mathsf{trace\_at\_location}\ p\ l\ t \equiv \mathsf{filter}\ (\lambda s.\ p.\mathsf{location}\ s = l)\ t\ .$$

[3] The infinite non-terminating program skip; skip; skip; $\cdots$ visits each program location precisely once and thus the TERMINATOR analysis trivially succeeds.

The TERMINATOR analysis for a whole program succeeds if it succeeds at every location:

$$\mathsf{terminator\_property}\ p \equiv \forall l \in \mathsf{locations}\ p.\ \mathsf{terminator\_property\_at\_location}\ p\ l\ .$$

4 Verifying TERMINATOR

At this point the formalization is complete, and the correctness statement for the TERMINATOR analysis can be expressed as

$$\forall p \in \mathsf{programs}.\ \mathsf{terminator\_property}\ p \Rightarrow \mathsf{terminates}\ p\ .$$

How to prove this verification goal? The first step is to fix a program location l, and prove that no trace can visit l infinitely often.

The proof is easiest to explain by contradiction: suppose a program trace is filtered to give in an infinite list of l-states $x_0, x_1, x_2, \ldots$. The TERMINATOR analysis results in well-founded relations $R_0, \ldots, R_{n-1}$ such that for every $i < j$ there exists $0 \leq k < n$ satisfying $R_k\ x_j\ x_i$.

The proof proceeds by induction on n. If $n = 0$ then the contradiction is immediate because there is no well-founded relation available to compare x_0 and x_1. For $n > 0$ construct an undirected graph $G = (V, E)$ with vertex set $V = \mathbb{N}$ and edge relation

$$E\ i\ j = i < j \wedge R_{n-1}\ x_j\ x_i\ .$$

The next step is formalize a result of Ramsey Theory [4] that every infinite graph has an infinite subgraph that is either complete (i.e., every vertex is connected to every other) or empty (i.e., there are no edges).[4] Here is the higher order logic theorem:

$$\begin{aligned}
\vdash\ & \forall V, E.\ \mathsf{infinite}\ V \Rightarrow \\
& \exists M \subseteq V.\ \mathsf{infinite}\ M\ \wedge \\
& \quad ((\forall i, j \in M.\ i < j \Rightarrow E\ i\ j)\ \vee\ (\forall i, j \in M.\ i < j \Rightarrow \neg E\ i\ j))\ .
\end{aligned}$$

What do the two cases mean for the graph G? If the infinite subgraph $G' = (M, E)$ is complete, then the subsequence of vertices in M is an R_{n-1} infinite descending sequence: a contradiction since R_{n-1} is a well-founded relation. If instead G' is empty, then the relation R_{n-1} is never used and the problem reduces to $n - 1$ well-founded relations: the contradiction is provided by the inductive hypothesis. The final theorem is

$$\begin{aligned}
\vdash\ & \forall p \in \mathsf{programs}.\ \forall l \in \mathsf{locations}\ p. \\
& \quad \mathsf{terminator_property_at_location}\ p\ l \Rightarrow \\
& \quad \forall t \in \mathsf{traces}\ p.\ \mathsf{finite}\ (\mathsf{trace_at_location}\ p\ l\ t)\ .
\end{aligned}$$

[4] Formalizing Ramsey Theory in higher order logic is not novel; perhaps the earliest example is Harrison's HOL88 theory in 1994, now ported to HOL Light.

The final step of the verification is to deduce that if no program location is visited infinitely often then there are no infinite program traces:

$$\vdash \forall p \in \mathsf{programs}.\ \mathsf{terminator\_property}\ p \Rightarrow \mathsf{terminates}\ p\ .$$

Note that this relies on well-formed programs having a finite set of locations.

5 Optimizations

The previous section formally verified the core TERMINATOR program analysis, but the real tool also implements a number of optimizations to speed up the termination proof. In this section the verification is extended to include two of the most significant ones.

The first optimization occurs when there is only one relation that has been found (so far) at a program location l. Instead of the general program modification, TERMINATOR simply modifies the program to compare each l-state with the previous l-state, by inserting

```
already_saved_state := false;
```

at the beginning of the program, and

```
if (already_saved_state && ¬R state saved_state) {
  error("possible non-termination");
}
saved_state := state;
already_saved_state := true;
```

just before location l. The definition of a successful TERMINATOR program analysis at location l must therefore be weakened to

$$\begin{array}{l}\mathsf{terminator\_property\_at\_location}\ p\ l \equiv \\ \quad (\exists R. \\ \quad\quad \mathsf{well\_founded}\ R\ \wedge \\ \quad\quad \forall t \in \mathsf{traces}\ p.\ \forall x_i, x_{i+1} \in \mathsf{trace\_at\_location}\ p\ l\ t.\ R\ x_{i+1}\ x_i)\ \vee \\ \quad [\ldots \text{old definition of } \mathsf{terminator\_property\_at\_location}\ p\ l \ldots]\ .\end{array}$$

The second optimization that TERMINATOR implements is to skip the analysis for all but a *cut set* of program locations:

$$\begin{array}{l}\mathsf{cut\_sets}\ p \equiv \\ \quad \{L \mid L \subseteq \mathsf{locations}\ p\ \wedge \\ \quad\quad \forall t \in \mathsf{traces}\ p.\ \mathsf{infinite}\ t \Rightarrow \exists l \in L.\ \mathsf{infinite}\ (\mathsf{trace\_at\_location}\ p\ l\ t)\}\ .\end{array}$$

Intuitively, a set of program locations is a cut set if every infinite trace visits the cut set infinitely often. This is a semantic property, and in general is hard to prove.[5] In practice, TERMINATOR chooses a cut set to include all locations at the start of loops and functions that are called (mutually) recursively.

[5] Indeed, if the program is terminating, all location sets are cut sets!

Taking both these optimizations into account, the result of a successful TERMINATOR program analysis is captured by the following augmented definition:

$$\begin{aligned}&\mathsf{terminator\_property}\ p \equiv \\ &\quad \exists C \in \mathsf{cut\_sets}\ p.\ \forall l \in C.\ \mathsf{terminator\_property\_at\_location}\ p\ l\ .\end{aligned}$$

And the same correctness theorem is still true, requiring only modest changes to the proofs.

6 Summary

This proof pearl presented a formal verification of the termination analysis of the TERMINATOR static analysis tool. The correctness result is not obvious (at least to the author), and the proof is an interesting application of Ramsey Theory. Naturally, the mechanized proof uses the same concepts as the original published proof [6], but was made more self-contained to simplify both formalization and presentation. The HOL4 proof script is 500 lines long (including the Ramsey Theory lemmas), and took two days to get the formalization right and then another two days to complete the verification.

The verification of the TERMINATOR optimizations in Section 5 represents a first step toward a verified practical tool, but that is a long way off. An interesting next step would be a deep embedding of structured programs, so that the TERMINATOR program modification and cut set generation could be incorporated into the verification, as well as optimizations such as ignoring program traces that leave and come back into the current loop.

Acknowledgements

Byron Cook provided the initial stimulus for this work, and it was greatly improved in discussions with both him and Andreas Podelski. Comments from the anonymous TPHOLs referees greatly improved the paper.

References

1. 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 *Proceedings of the EuroSys 2006 Conference*, pages 73–85, April 2006.
2. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In Thomas Ball and Robert B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006)*, volume 4144 of *Lecture Notes in Computer Science*, pages 415–418, Seattle, WA, USA, August 2006. Springer.
3. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL (A theorem-proving environment for higher order logic)*. Cambridge University Press, 1993.
4. Bruce M. Landman and Aaron Robertson. *Ramsey Theory on the Integers*. American Mathematical Society, February 2004.

5. Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2004)*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer, January 2004.
6. Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004)*, pages 32–41, July 2004.