Simulating Termination Analyzer H is Not Fooled by Pathological Input D

The notion of a simulating termination analyzer is examined at the concrete level of pairs of C functions. This is similar to AProVE: Non-Termination Witnesses for C Programs: The termination status decision is made on the basis of the dynamic behavior of the input. This paper explores what happens when a simulating termination analyzer is applied to an input that calls itself.

In computer science, termination analysis is program analysis which attempts to determine whether the evaluation of a given program halts for each input. This means to determine whether the input program computes a total function.
https://en.wikipedia.org/wiki/Termination_analysis

To understand this analysis requires a sufficient knowledge of the C programming language. An x86 emulator works just like a C language interpreter except that it uses the compiled machine language of a function instead of its C source-code. That HHH is built from an x86 emulator allows it to simulate other C functions as if it was a C language interpreter.

x86utm is an multi-tasking operating system that enables one C function to simulate another C function in debug step mode. Termination analyzers can simulate inputs that call this same termination analyzer.

```
typedef void (*ptr)();
int HHH(ptr P);

void Infinite_Recursion()
{
  Infinite_Recursion();
  return;
}

void DDD()
{
  HHH(DDD);
  return;
}

int main()
{
  HHH(Infinite_Recursion);
  HHH(DDD);
}
```

Every C programmer that knows that when **HHH** emulates the machine language of, `Infinite_Recursion` that it must abort this emulation so that itself can terminate normally.

When this is construed as non-halting criteria then simulating termination analyzer **HHH** is correct to reject this input as non-halting by returning 0 to its caller.

We get the same repetitive pattern when DDD is correctly emulated by HHH. HHH emulates DDD that calls HHH(DDD) to do this again.

HHH simulates DDD that calls HHH(DDD). This requires HHH to simulate itself simulating DDD. HHH examines the execution trace of its simulated input. HHH must ignore the execution trace of itself otherwise it would determine that `Infinite_Recursion()` halts on the basis that HHH halts.

When we examine the infinite set of every HHH/DDD pair such that:
HHH_1 One step of DDD_1 is correctly emulated by HHH_1
HHH_2 Two steps of DDD_2 are correctly emulated by HHH_2
HHH_3 Three steps of DDD_3 are correctly emulated by HHH_3
...
HHH_∞ The emulation of DDD_∞ by HHH_∞

DDD correctly emulated by any HHH never reaches its of "return" instruction. When we understand that this "return" instruction is the final halt state of DDD then we can see that no DDD emulated by any HHH ever halts.

This means that each HHH can take a wild guess that its input never halts and be necessarily correct. It need not be a wild guess. There is a repeating pattern. HHH simulates DDD that calls HHH(DDD) to repeat the process.

Professor Hehner recognized this repeating process before I did.

> From a programmer's point of view, if we apply an interpreter to a program text that includes a call to that same interpreter with that same text as argument, then we have an infinite loop. A halting program has some of the same character as an interpreter: it applies to texts through abstract interpretation. Unsurprisingly, if we apply a halting program to a program text that includes a call to that same halting program with that same text as argument, then we have an infinite loop. **(Hehner:2011:15)**

Thus Professor Hehner derived the essence of this halt status criteria:
This algorithm is used by all the simulating termination analyzers:
<MIT Professor Sipser agreed to ONLY these verbatim words 10/13/2022>

> If simulating halt decider H correctly simulates its input D
> until H correctly determines that its simulated D would never
> stop running unless aborted then
>
> H can abort its simulation of D and correctly report that D
> specifies a non-halting sequence of configurations.

</MIT Professor Sipser agreed to ONLY these verbatim words 10/13/2022>

The author independently derived the above algorithm. The non-halting criteria was independently discovered in 2016 and applied to a simulating halt decider in 2017. This became fully operational software in 2020.

As soon as simulating termination analyzer HHH correctly determines that it must abort DDD to prevent its own infinite execution it is necessarily correct to abort DDD and return 0 indicating that it has rejected this input as non-halting.

Computable functions are the formalized analogue of the intuitive notion of algorithms, in the sense that a function is computable if there exists an algorithm that can do the job of the function, i.e. **given an input of the function domain it can return the corresponding output.**
https://en.wikipedia.org/wiki/Computable_function

A halt decider computes the mapping from an input finite string to the behavior that this finite string specifies. No halt decider ever reports on the actual behavior of the computation that itself is contained within.

For the three years that my work has been extensively reviewed this has been the most difficult point for people to understand. Everyone remains convinced that HHH must report on the behavior of the computation that itself is contained within and not the behavior that its finite string input specifies.

```
typedef void (*ptr)();
int HHH(ptr P);

int DD()
{
  int Halt_Status = HHH(DD);
  if (Halt_Status)
    HERE: goto HERE;
  return Halt_Status;
}

int main()
{
  HHH(DD);
}
```

When we understand that
(a) Decider **HHH** must report on the behavior that its input actually specifies.

(b) The measure of this behavior is **DD** correctly simulated by **HHH** including its recursive call to **HHH(DD).**

Then we can see that **DD** correctly simulated **HHH** cannot possibly reach past its own first line.

We have covered the simplest possible example of pathological self-reference: An input that calls its own simulating termination analyzer.

Now we move on to the conventional halting problem proof relationship. As we see in the sidebar the paradoxical part of this input is simply unreachable. Thus DD correctly emulated by HHH has the same behavior pattern as DDD correctly emulated by HHH.

These pathological inputs themselves have no inputs so that HHH can be construed as a conventional termination analyzer.

Simulating (partial) halt decider applied to Peter Linz Halting Problem Proof

A simulating (partial) halt decider correctly predicts whether or not its correctly simulated input can possibly reach its own final state and halt. It does this by correctly recognizing several non-halting behavior patterns in a finite number of steps of correct simulation. Inputs that do terminate are simply simulated until they complete.

When a simulating (partial) halt decider correctly simulates N steps of its input it derives the exact same N steps that a pure UTM would derive because it is itself a UTM with extra features.

My reviewers cannot show that any of the extra features added to the UTM change the behavior of the simulated input for the first N steps of simulation:
(a) Watching the behavior doesn't change it.
(b) Matching non-halting behavior patterns doesn't change it
(c) Even aborting the simulation after N steps doesn't change the first N steps.

Because of all this we can know that the first N steps of input D simulated by simulating (partial) halt decider H are the actual behavior that D specifies to H for these same N steps.

computation that halts… "the Turing machine will halt whenever it enters a final state" (Linz:1990:234)

When we see (after N steps) that D correctly simulated by H cannot possibly reach its simulated final state in any finite number of steps of correct simulation then we have conclusive proof that D presents non-halting behavior to H.
A simulating (partial) halt decider must always stop its simulation and report non-halting when-so-ever it correctly detects that its correct simulation would never otherwise stop running. All halt deciders compute the mapping from their inputs to an accept or reject state on the basis of the actual behavior specified by this input.

When an input is defined to have a pathological relationship to its simulator this changes the behavior of this input. A simulating (partial) halt decider (with a pathological relationship) must report on this changed behavior to prevent its own infinite execution by aborting its simulation.

Summary of Linz Halting Problem Proof
The Linz halting problem proof constructs its counter-example input ⟨Ĥ⟩ on the basis of prepending and appending states to the original Linz H, (assumed halt decider) thus is named embedded_H.

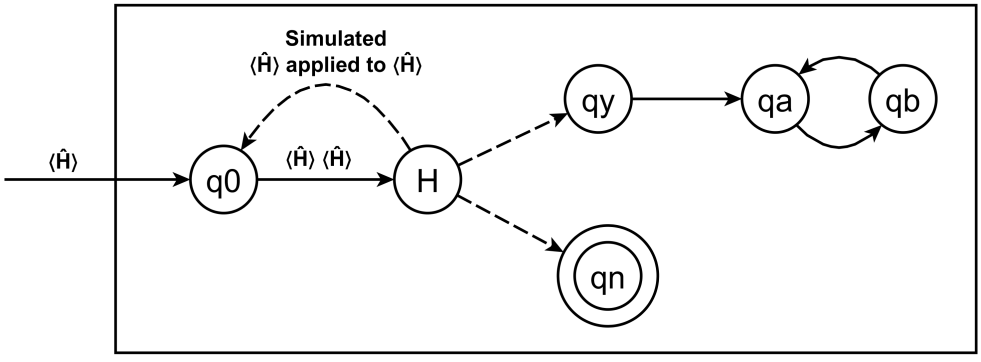
Original Linz Turing Machine H
H.q0 ⟨M⟩ w ⊢* H.qy // M applied to w halts
H.q0 ⟨M⟩ w ⊢* H.qn // M applied to w does not halt

The Linz term "move" means a state transition and its corresponding tape head action {move_left, move_right, read, write}.

(q0) is prepended to H to copy the ⟨M⟩ input of Ĥ. The transition from (qa) to (qb) is the conventional infinite loop appended to the (qy) accept state of embedded_H.
⊢* indicates an arbitrary number of moves.
⊢* specifies a wildcard sequence of state transitions

Ĥ.q0 ⟨M⟩ ⊢* embedded_H ⟨M⟩ ⟨M⟩ ⊢* Ĥ.qy ∞
Ĥ.q0 ⟨M⟩ ⊢* embedded_H ⟨M⟩ ⟨M⟩ ⊢* Ĥ.qn

Analysis of Linz Halting Problem Proof --- Copy of ⟨Ĥ⟩ simulated with ⟨Ĥ⟩



When Ĥ is applied to ⟨Ĥ⟩
Ĥ.q0 ⟨Ĥ⟩ ⊢* embedded_H ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qy ∞
Ĥ.q0 ⟨Ĥ⟩ ⊢* embedded_H ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qn

(a) Ĥ copies its input ⟨Ĥ⟩
(b) Ĥ invokes embedded_H ⟨Ĥ⟩ ⟨Ĥ⟩
(c) embedded_H simulates ⟨Ĥ⟩ ⟨Ĥ⟩
(d) simulated ⟨Ĥ⟩ copies its input ⟨Ĥ⟩
(e) simulated ⟨Ĥ⟩ invokes simulated embedded_H ⟨Ĥ⟩ ⟨Ĥ⟩
(f) simulated embedded_H simulates ⟨Ĥ⟩ ⟨Ĥ⟩
(g) goto (d) with one more level of simulation

Two complete simulations show a pair of identical TMD's are simulating a pair of identical inputs. We can see this thus proving recursive simulation that cannot possibly stop running unless aborted.

When we understand that embedded_H is accountable for the behavior of its input and not accountable for the behavior of the computation that itself is contained within then we understand that embedded_H is necessarily correct to transition to its own Ĥ.qn state.

"δ is the transition function" (Linz:1990:233) ...
"A Turing machine is said to halt whenever it reaches a configuration for which δ is not defined; (Linz:1990:234)

Non-halting behavior patterns can be matched in N steps. The simulated ⟨Ĥ⟩ halts only it when reaches its simulated final state of ⟨Ĥ.qn⟩ in a finite number of steps.

Execution trace of Ĥ applied to ⟨Ĥ⟩
(a) Ĥ.q0 The input ⟨Ĥ⟩ is copied then transitions to embedded_H
(b) embedded_H applied ⟨Ĥ⟩ ⟨Ĥ⟩ (input and copy) simulates ⟨Ĥ⟩ applied to ⟨Ĥ⟩
(c) which begins at its own simulated ⟨Ĥ.q0⟩ to repeat the process

Simulation invariant: ⟨Ĥ⟩ correctly simulated by embedded_H never reaches its own simulated final state of ⟨Ĥ.qn⟩.

When embedded_H correctly simulates the state transitions
specified by its input in the order that they are specified

⟨Ĥ⟩ ⟨Ĥ⟩ correctly simulated by embedded_H cannot possibly
reach its own simulated final state of ⟨Ĥ.qn⟩ and halt.

Therefore when embedded_H aborts the simulation of its input and transitions to its own final state of Ĥ.qn it is merely reporting this verified fact.

Conclusion
We have shown a 100% fully operational concrete example of a simulating termination analyzer applied to a pair of C functions that have the Halting Problem's pathological relationship to each other.

When it is understood that D correctly simulated by H cannot possibly halt and that H is reporting on the behavior of this correctly simulated input then H is correct to abort its simulation of D and report that this input does not halt.

The exact same reasoning applies to the Peter Linz Halting Problem proof. When embedded_H is applied to ⟨Ĥ⟩ ⟨Ĥ⟩ it transitions to Ĥ.qn indicating that its correctly simulated input cannot possibly reach its own simulated final state of ⟨Ĥ.qn⟩.

embedded_H is not allowed to report on the behavior of itself thus is not allowed to report on the behavior of Ĥ applied to ⟨Ĥ⟩. When we apply Linz H to ⟨Ĥ⟩ ⟨Ĥ⟩ it correctly reports that Ĥ applied to ⟨Ĥ⟩ will reach its own final state of Ĥ.qn and halt.

References

[1] Steffen Winterfeldt and others **libx86emu** (x86 emulation library)
1996-2017 https://github.com/wfeldt/libx86emu

[2] P Olcott, 2023. **The x86utm operating system:**
https://github.com/plolcott/x86utm **--- Above code samples in Halt7.c**
Above code samples are fully operational in the x86utm operating system.

[3] E C R Hehner. **Objective and Subjective Specifications**
WST Workshop on Termination, Oxford. 2018 July 18.
See https://www.cs.toronto.edu/~hehner/OSS.pdf

[4] Bill Stoddart. **The Halting Paradox**
20 December 2017
https://arxiv.org/abs/1906.05340
arXiv:1906.05340 [cs.LO]

[5] E C R Hehner. **Problems with the Halting Problem**, COMPUTING2011 Symposium on 75 years of Turing Machine and Lambda-Calculus, Karlsruhe Germany, invited, 2011 October 20-21; Advances in Computer Science and Engineering v.10 n.1 p.31-60, 2013
https://www.cs.toronto.edu/~hehner/PHP.pdf

[6] Linz, Peter 1990. **An Introduction to Formal Languages and Automata.**
Lexington/Toronto: D. C. Heath and Company. (317-320)

[7] Nicholas J. Macias. **Context-Dependent Functions:**
Narrowing the Realm of Turing's Halting Problem
13 Nov 2014
https://arxiv.org/abs/1501.03018
arXiv:1501.03018 [cs.LO]

[8] Jera Hensel , Constantin Mensendiek , and Jürgen Giesl
AProVE: Non-Termination Witnesses for C Programs
LuFG Informatik 2, RWTH Aachen University, Germany
https://link.springer.com/content/pdf/10.1007/978-3-030-99527-0_21.pdf