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

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

To understand this analysis requires a sufficient knowledge of the C programming language and what an x86 emulator does. It is also very helpful to have some basic understanding of the x86 programming language. When we examine termination at the concrete C / x86 level false assumptions that slip through the cracks of the Turing Machine level are exposed.

CODE SAMPLE 1

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

void Infinite_Loop()
{
  HERE: goto HERE;
}

void Infinite_Recursion()
{
  Infinite_Recursion();
}

void DDD()
{
  HHH(DDD);
}

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

Analysis of CODE SAMPLE 1

Every C programmer that knows what an x86 emulator is knows that when **HHH** emulates the machine language of **`Infinite_Loop`, `Infinite_Recursion`, and `DDD`** that it must abort these emulations so that itself can terminate normally.

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

Simulating termination analyzers report on the behavior that their finite string input specifies thus **HHH** must report that **DDD** correctly emulated by **HHH** remains stuck in recursive simulation unless this simulation is aborted.

Now we examine that behavior of DDD correctly emulated by x86 emulator HHH at the x86 programming language level. When **we stipulate that the only measure of a correct emulation is the semantics of the x86 programming language** then we see that when DDD is correctly emulated by HHH that its call to HHH(DDD) cannot possibly return.

```
_DDD()
[00002163] 55         push ebp      ; housekeeping
[00002164] 8bec       mov ebp,esp   ; housekeeping
[00002166] 6863210000 push 00002163 ; push DDD
[0000216b] e853f4ffff call 000015c3 ; call HHH(DDD)
[00002170] 83c404     add esp,+04
[00002173] 5d         pop ebp
[00002174] c3         ret
Size in bytes:(0018) [00002174]
```

When DDD is correctly emulated by any pure function x86 emulator HHH calls an emulated HHH(DDD) this call cannot possibly return. This prevents the emulated DDD from ever reaching past its own machine address of **0000216b** and halting.

HHH must report that it needs to abort its emulaton of DDD. HHH can't correctly report that DDD doesn't need to be aborted at the point in the execution trace where DDD does need to be aborted. The behavior of DDD(DDD) is at a different point in the execution trace after HHH has already aborted the emulated DDD.

Halt Deciders compute the mapping from their actual finite string input to a Boolean output on the basis of the actual behavor specified by this input. The behavior of this input must include and cannot ignore the recursive emulation specified by the fact that DDD is calling its own emulator.

A partial halt decider is a computable function that computes the mapping from its finite string input to a Boolean value corresponding to the behavior that this finite string actually specifies. It does this for a limited set of inputs.

The following algorithm is used by 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 x86utm operating system is a proxy for a UTM and uses C functions as proxies for Turing Machines and the x86 language as a proxy for the Turing Machine description language. This makes every single detail of the halting problem 100% concrete thus totally eliminating any false assumptions.

The halting problem proof is understood to be the logical impossibility of specifying a halt decider **HHH** that correctly reports the halt status of input **DD** that is defined to do the opposite of whatever value that **HHH** reports. **Of course this is impossible.**

HHH(DD) has the classic halting problem proof relationship to its input. **HHH(DD)** has the same behavior as the above **DDD** correctly simulated by **HHH**. This prevents **DD** correctly simulated by **HHH** from reaching past its own first line. This makes the classic halting problem question moot:

What Boolean value can HHH correctly return when input DD is defined to do the opposite of every value that HHH returns? DD correctly emulated by HHH cannot possibly reach this paradoxical point at its own second line.

```
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.

```
_DD()
[00002133] 55         push ebp      ; housekeeping
[00002134] 8bec       mov ebp,esp   ; housekeeping
[00002136] 51         push ecx      ; make space for local
[00002137] 6833210000 push 00002133 ; push DD
[0000213c] e882f4ffff call 000015c3 ; call HHH(DD)
[00002141] 83c404     add esp,+04
[00002144] 8945fc     mov [ebp-04],eax
[00002147] 837dfc00   cmp dword [ebp-04],+00
[0000214b] 7402       jz 0000214f
[0000214d] ebfe       jmp 0000214d
[0000214f] 8b45fc     mov eax,[ebp-04]
[00002152] 8be5       mov esp,ebp
[00002154] 5d         pop ebp
[00002155] c3         ret
Size in bytes:(0035) [00002155]
```

The same reasoning that applied to DDD correctly simulated by HHH applies here. When **we stipulate that the only measure of a correct emulation is the semantics of the x86 programming language** then we see that when DD is correctly emulated by HHH that its call to HHH(DD) cannot possibly return.

As we can see from DDD correctly emulated by HHH the behavior of the input to HHH(DD) is different than the behavior of the directly executed DD(DD). Computable functions including halt deciders are only accountable for the actual behavior of their actual inputs. No one is free to overrule the semantics of the x86 language.

HHH must report that it needs to abort its emulaton of DD. HHH can't correctly report that DD doesn't need to be aborted at the point in the execution trace where DD does need to be aborted. The behavior of DD(DD) is at a different point in the execution trace after HHH has already aborted the emulated DD.

Because the call from DD correctly simulated by HHH to HHH(DD) cannot possibly return this DD cannot possibly reach past its own first line. This makes the paradoxical portion of DD unreachable making it moot.

HHH uses the same non-halt status criteria that it uses to detect infinite recursion to detect and reject that DD correctly simulated by HHH would halt. HHH returns 0 to it caller to indicate it rejected its input as non-halting.

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.

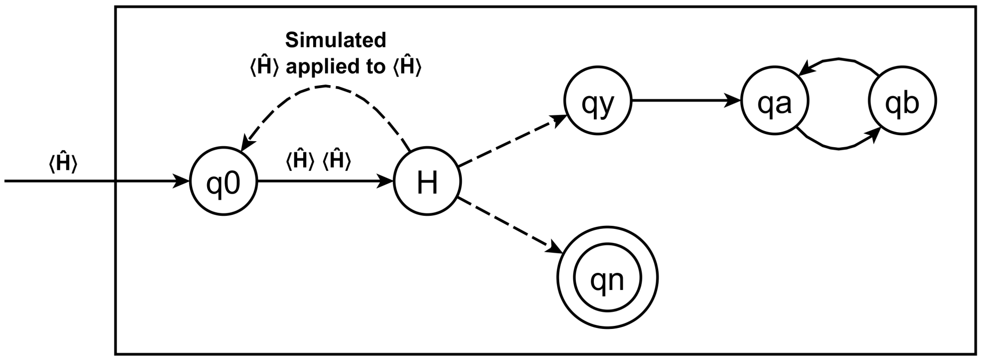
Original Linz Turing Machine H
H.q0 ⟨M⟩ w ⊢* H.qy // M applied to w halts
H.q0 ⟨M⟩ w ⊢* H.qn // M applied to w does not halt

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.

"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)

Simulating Partial Halt Decider Applied to Linz Proof
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
Several fully operational simulating termination analyzers with sample inputs.

[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