Simulating Halt Decider Applied to the Halting Theorem

MIT Professor Michael Sipser has agreed that the following verbatim paragraph is correct (he has not agreed to anything else in this paper):

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.

A simulating halt decider computes the mapping from its input finite strings to an accept or reject state on the basis of the actual behavior specified by this input as measured by its correct simulation of this input.

We start with Sipser's definitions of H and D:

> On input (M, w), where M is a TM and w is a string, H halts and accepts if M accepts w. Furthermore, H halts and rejects if M fails to accept w. In other words, we assume that H is a TM, where
>
> H(⟨M,w⟩ = { accept if M accepts w
> { reject if M does not accept w
>
> Now we construct a new Turing machine D with H as a subroutine. This new TM calls H to determine what M does when the input to M is its own description ⟨M⟩. Once D has determined this information, it does the opposite. That is, it rejects if M accepts and accepts if M does not accept.
>
> D(⟨M⟩) = { accept if M does not accept ⟨M⟩
> { reject if M accepts ⟨M⟩ (Sipser 1997:165)

We encode the Sipser D and define the behavior of Sipser H as C functions.

```
//
// Sipser_H returns 1 when its input would halt and return 1
// otherwise Sipser_H returns 0
//
int Sipser_D(int (*M)())
{
  if ( Sipser_H(M, M) )
    return 0;
  return 1;
}


int main()
{
  Output((char*)"Input_Halts = ", Sipser_D(Sipser_D));
}
```

When H correctly simulates D it finds that D remains stuck in recursive simulation
(a) D calls H that simulates D with an x86 emulator
(b) that calls a simulated H that simulates D with an x86 emulator
(c) that calls a simulated H that simulates D with an x86 emulator ...
Until the executed H recognizes this repeating state, aborts its simulation of D and returns 0.
The first page of the Appendix has all of the details about this.

D calls simulating halt decider H which computes the mapping from its input D to an accept or reject state on the basis of the behavior of its correct simulation of D. When H correctly determines that this simulated input would remain stuck in recursive simulation H aborts this simulation and reports non-halting by returning 0. When D reverses this decision it returns 1. This is used to correctly fill in the "?" in the Sipser Figure 4.6 (see below) with "accept".

Simulating halt decider H recognizes instances of recursive simulation using the same criteria that it uses in its dynamic behavior pattern that recognizes infinite recursion:

```
void Infinite_Recursion(u32 N)
{
  Infinite_Recursion(N);
}

int main()
{
  Output((char*)"Input_Halts = ", H(Infinite_Recursion, (ptr)0x777));
}

_Infinite_Recursion()
[000013fa] 55              push ebp
[000013fb] 8bec            mov ebp,esp
[000013fd] 8b4508          mov eax,[ebp+08]
[00001400] 50              push eax
[00001401] e8f4ffffff      call 000013fa
[00001406] 83c404          add esp,+04
[00001409] 5d              pop ebp
[0000140a] c3              ret
Size in bytes:(0017) [0000140a]
```

H detects that _Infinite_Recursion() calls itself with no condtional branch instructions between the beginning of _Infinite_Recursion() and the call to itself that could escape repeated recursion.

	⟨M_1⟩	⟨M_2⟩	⟨M_3⟩	⟨M_4⟩ ...	⟨D⟩ ...
M_1	accept	reject	accept	reject	accept
M_2	accept	accept	accept	accept	accept
M_3	reject	reject	reject	reject	reject
M_4	accept	accept	reject	reject	accept
...					
D	reject	reject	accept	accept	?
...					

Figure 4.6 (Sipser 1997:167)

Sipser, Michael 1997. Introduction to the Theory of Computation. Boston: PWS Publishing Company (165-167)

Complete halt deciding system (Visual Studio Project) Sipser version.
(a) x86utm operating system
(b) x86 emulator adapted from libx86emu to compile under Windows
(c) Several halt deciders and their sample inputs contained within Halt7.c
(d) The execution trace of Sipser_H applied to Sipser_D is shown in Halt7_Sipser.txt
https://liarparadox.org/2022_10_08.zip

Appendix

```
int Sipser_D(int (*M)())
{
  if ( Sipser_H(M, M) )
    return 0;
  return 1;
}

int main()
{
  Output((char*)"Input_Halts = ", Sipser_D(Sipser_D));
}

_Sipser_D()
[000012ae] 55             push ebp
[000012af] 8bec           mov ebp,esp
[000012b1] 8b4508         mov eax,[ebp+08]
[000012b4] 50             push eax
[000012b5] 8b4d08         mov ecx,[ebp+08]
[000012b8] 51             push ecx
[000012b9] e880fdffff     call 0000103e
[000012be] 83c408         add esp,+08
[000012c1] 85c0           test eax,eax
[000012c3] 7404           jz 000012c9
[000012c5] 33c0           xor eax,eax
[000012c7] eb05           jmp 000012ce
[000012c9] b801000000     mov eax,00000001
[000012ce] 5d             pop ebp
[000012cf] c3             ret
Size in bytes:(0034) [000012cf]
```

When H correctly simulates D it finds that D remains stuck in recursive simulation

```
Sipser_H: Begin Simulation   Execution Trace Stored at:111fa8
 machine   stack     stack     machine    assembly
 address   address   data      code       language
 ========  ========  ========  =========  =============
[000012ae][00111f94][00111f98] 55         push ebp      // Begin Sipser_D
[000012af][00111f94][00111f98] 8bec       mov ebp,esp
[000012b1][00111f94][00111f98] 8b4508     mov eax,[ebp+08]
[000012b4][00111f90][000012ae] 50         push eax      // push Sipser_D
[000012b5][00111f90][000012ae] 8b4d08     mov ecx,[ebp+08]
[000012b8][00111f8c][000012ae] 51         push ecx      // push Sipser_D
[000012b9][00111f88][000012be] e880fdffff call 0000103e // call Sipser_H
Sipser_H: Infinitely Recursive Simulation Detected Simulation Stopped
```

We can see that the first seven instructions of Sipser_D simulated by Sipser_H precisely match the first seven instructions of the x86 source-code of Sipser_D. This conclusively proves that these instructions were simulated correctly.

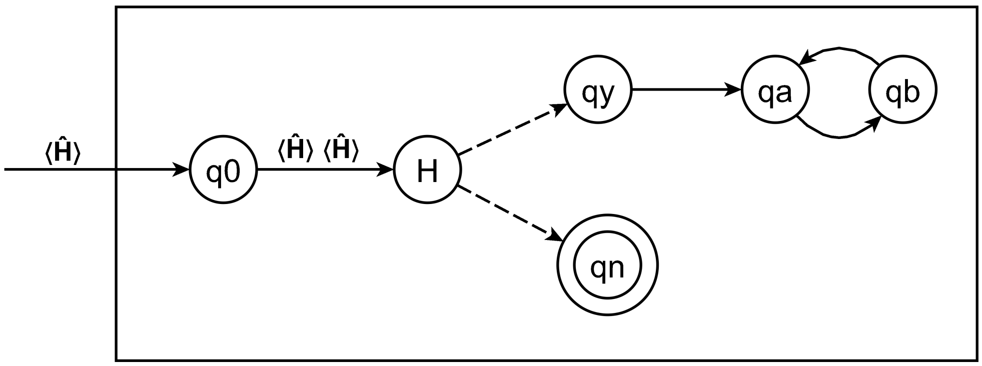
Anyone sufficiently technically competent in the x86 programming language will agree that the above execution trace of Sipser_D simulated by Sipser_H shows that Sipser_D will never stop running unless Sipser_H aborts its simulation of Sipser_D.

Sipser_H detects that Siper_D is calling itself with the exact same arguments that Siper_H was called with and there are no conditional branch instructions from the beginning of Sipser_D to its call to Sipser_H that can possibly escape the repetition of this recursive simulation.

Peter Linz Halting Problem Proof adapted to use a simulating halt decider

When we see the notion of a simulating halt decider applied to the embedded copy of Linz H within Linz Ĥ then we can see that the ⟨Ĥ⟩ ⟨Ĥ⟩ input to embedded H specifies recursive simulation that never reaches its own final state of ⟨Ĥ.qy⟩ or ⟨Ĥ.qn⟩.

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



$\hat{H}.q_0$ ⟨Ĥ⟩ ⊢* H ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qy ∞
If ⟨Ĥ⟩ ⟨Ĥ⟩ correctly simulated by H would reach its own final state of ⟨Ĥ.qy⟩ or ⟨Ĥ.qn⟩.

$\hat{H}.q_0$ ⟨Ĥ⟩ ⊢* H ⟨Ĥ⟩ ⟨Ĥ⟩ ⊢* Ĥ.qn
If ⟨Ĥ⟩ ⟨Ĥ⟩ correctly simulated by H would never reach its own final state of ⟨Ĥ.qy⟩ or ⟨Ĥ.qn⟩.

When Ĥ is applied to ⟨Ĥ⟩ // subscripts indicate unique finite strings
Ĥ copies its input $\langle\hat{H}_0\rangle$ to $\langle\hat{H}_1\rangle$ then H simulates $\langle\hat{H}_0\rangle$ $\langle\hat{H}_1\rangle$

Then these steps would keep repeating: (unless their simulation is aborted)
$\hat{H}_0$ copies its input $\langle\hat{H}_1\rangle$ to $\langle\hat{H}_2\rangle$ then H_0 simulates $\langle\hat{H}_1\rangle$ $\langle\hat{H}_2\rangle$
$\hat{H}_1$ copies its input $\langle\hat{H}_2\rangle$ to $\langle\hat{H}_3\rangle$ then H_1 simulates $\langle\hat{H}_2\rangle$ $\langle\hat{H}_3\rangle$
$\hat{H}_2$ copies its input $\langle\hat{H}_3\rangle$ to $\langle\hat{H}_4\rangle$ then H_2 simulates $\langle\hat{H}_3\rangle$ $\langle\hat{H}_4\rangle$...

Since we can see that the input: $\langle\hat{H}_0\rangle$ $\langle\hat{H}_1\rangle$ correctly simulated by H would never reach its own final state of $\langle\hat{H}_0.qy\rangle$ or $\langle\hat{H}_0.qn\rangle$ we know that $\langle\hat{H}_0\rangle$ specifies a non-halting sequence of configurations.

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