

INTRODUCTION TO THE ACTIVE ELEMENT MACHINE

MICHAEL STEPHEN FISKE

Aemea, San Francisco, CA, 94129
e-mail address: mf@aemea.org

ABSTRACT. A new computing machine, called an active element machine (AEM), and programming language is presented. This computing model is motivated by the positive aspects of dendritic integration, inspired by biology, and traditional programming languages based on the register machine. Distinct from the traditional register machine, the fundamental computing elements – active elements – compute simultaneously. Distinct from traditional programming languages, all active element commands have an explicit reference to time. These attributes make the AEM an inherently parallel machine, enable the AEM to change its architecture (program) as it is executing its program. Using a random bit source from the environment and the Meta command, a simple AEM program generates an arbitrary real number in $[0, 1]$. Exploiting the randomness from the environment, this example is extended to an AEM that can recognize an arbitrary binary language $L \subseteq \{0, 1\}^*$. Finally, an AEM program finds the Ramsey number $r(3, 3)$, illustrating how parallel AEM algorithms and time in the commands help compute an NP-hard problem.

1. INTRODUCTION

A new computing machine, called an active element machine (AEM), and programming language is presented. This computing model is motivated by the positive aspects of dendritic integration, inspired by biology, and traditional programming languages based on the register machine. Distinct from the traditional register machine, the fundamental computing elements – active elements – compute simultaneously. Distinct from traditional programming languages, all active element commands have an explicit reference to time. These attributes make the AEM an inherently parallel machine and enable the AEM to change its architecture (program) as it is executing its program.

1.1. Wilfrid Rall’s Models of Dendritic Integration. Wilfrid Rall’s research [11] in neurophysiology influenced the development of the active element machine – in particular, his work on dendritic integration and how this contributes to computation. Rall’s mathematical models are thorough and complicated; Rall modelled the non-linearities of the neuron and much of his work focussed on dendritic integration.

Our goal was to capture the critical computational properties of dendritic integration that use its computational power while keeping the mathematics as simple as possible.

1998 ACM Subject Classification: F.1.1.

Key words and phrases: active, computing, element, machine, programming language.

Another goal was to assure that the mathematics and computing mechanism were simple enough to implement in silicon and other kinds of hardware ([4], [5]).

Our third goal was to make the machine and language simple enough to design AEM programs implicitly, explicitly or both. An AEM program may be designed implicitly using evolutionary methods. An AEM program may be explicitly written by one or more persons. Early and current neural network models [7], [9] are complicated to program or do not have a simple programming language for designing the network. For the above reasons, implicit and explicit programmability were important criteria that influenced the design of the active element machine.

1.2. Register Machine Computation. Another part of this development comes from the formal model of the Turing machine [15] and the subsequent von Neumann architecture. (This section contains some rhetorical content as a means to motivate new notions.) Today's computers do not conceptually work much differently than these early models. Perhaps, the biggest difference is that today's computers are much faster. In the current notion of an *algorithm*, the relevant concepts of the Turing computing model (see [15] and definitions 4.3, 4.4, 4.5, 4.6) are:

- There are a finite number of alphabet symbols $A = \{a_1, \dots, a_n\}$ read and written to a tape.
- There are a finite number of machine states $Q = \{q_1, \dots, q_m\}$.
- The Turing program η is a finite set of rules that stays fixed: *The rules do not change as the program executes.*
- The execution of one rule represents a computational step. During this computational step, one of the rules is selected, based on the current alphabet symbol pointed to by the tape head and the current machine state. The output of the rule specifies that a new alphabet symbol or the same symbol is written to the tape, the machine moves to a new state or stays in the current state and that the tape head moves one square to the left or right.
- *Computational steps are executed sequentially with no explicit reference to time.*

In light of the above, it seems natural for the Turing machine to lead to the register machine (see [1], [10], [14]). In the register machine, a program is a finite number of instructions that are executed in a linear sequence. Further, the contents of a register is changed in one computational step, which is analogous to writing a new symbol on the tape during one computational step of the Turing machine. In the register machine, there is also no explicit reference to an absolute or relative time. And usually only one register machine instruction is executed at a time, which creates a computational bottleneck.

1.3. Explicit Representation of Time. The register machine is a programmable machine but the program is fixed during program execution. There is also no notion of explicit time in the register machine model, only the order in which instructions are executed. Rall's research does not address programmability and has no notion of commands. His models used time, dendritic integration and adaptability of the synapses. The active element machine explicitly represents time in the machine commands which enables the following useful properties.

- Parallel algorithms can be implemented in a natural way, since each active element performs computation and all of them simultaneously compute.
- Explicit time in the active element commands enhances control over the active element machine computation because the synchronization of computation among different groups of active elements can be coordinated. This coordination helps avoid race conditions that can occur in the standard programming languages that implement concurrent processes.
- The machine can change its own architecture (program) with the Meta command while it is executing.
- The Meta command enables the active element machine's complexity to increase over time.

In [8], Edward Lee proposes using explicit time in a computing model and computing applications:

This paper argues that to realize its full potential, the core abstractions of computing need to be rethought to incorporate essential properties of the physical systems, most particularly the passage of time. It makes a case that the solution cannot be simply overlaid on existing abstractions, The emphasis needs to be on repeatable behavior rather than on performance optimization.

2. MACHINE ARCHITECTURE

An active element machine is composed of computational primitives called active elements. There are three kinds of active elements: Input, Computational and Output active elements. Input active elements process information received from the environment or another active element machine. Computational active elements receive messages from the input active elements and other computational active elements firing activity and transmit new messages to computational and output active elements. The output active elements receive messages from the input and computational active elements firing activity. The firing activity of the output active elements represents the output of the active element machine. Every active element is an active element in the sense that each one can receive and transmit messages simultaneously.

Each active element receives messages, formally called pulses, from other active elements and itself and transmits messages to other active elements and itself. If the messages received by active element, E_i , at the same time sum to a value greater than the threshold, then active element E_i fires. When an active element E_i fires, it sends messages to other active elements.

Let \mathbb{Z} denote the integers. Define the extended integers as $\overline{\mathbb{Z}} = \{m + kdT : m, k \in \mathbb{Z} \text{ and } dT \text{ is a fixed infinitesimal}\}$. For more on infinitesimals, see [13].

Definition 2.1. *Machine Architecture*

Γ , Ω , and Δ are index sets that index the input, computational, and output active elements, respectively. Depending on the machine architecture, the intersections $\Gamma \cap \Omega$ and $\Omega \cap \Delta$ can be empty or non-empty. A machine architecture, denoted as $\mathcal{M}(\mathcal{I}, \mathcal{E}, \mathcal{O})$, consists of a collection of input active elements, denoted as $\mathcal{I} = \{E_i : i \in \Gamma\}$; a collection of computational active elements $\mathcal{E} = \{E_i : i \in \Omega\}$; and a collection of output active elements

$\mathcal{O} = \{E_i : i \in \Delta\}$. Each computational and output active element, E_i , has the following components and properties:

- A threshold θ_i
- A refractory period r_i where $r_i > 0$.
- A collection of pulse amplitudes $\{A_{ki} : k \in \Gamma \cup \Omega\}$.
- A collection of transmission times $\{\tau_{ki} : k \in \Gamma \cup \Omega\}$, where $\tau_{ki} > 0$ for all $k \in \Gamma \cup \Omega$.
- A function of time, $\Psi_i(t)$, representing the time active element E_i last fired. $\Psi_i(t) = \sup\{s : s < t \text{ and } g_i(s) = 1\}$, where $g_i(s)$ is the output function of active element E_i and is defined below. The sup is the least upper bound.
- A binary output function, $g_i(t)$, representing whether active element E_i fires at time t . The value of $g_i(t) = 1$ if $\sum A_{ki}(t) > \theta_i$ where the sum ranges over all $k \in \Gamma \cup \Omega$ and $t \geq \Psi_i(t) + r_i$. In all other cases, $g_i(t) = 0$. For example, $g_i(t) = 0$, if $t < \Psi_i(t) + r_i$.
- A set of firing times of active element E_k within active element E_i 's integrating window, $W_{ki}(t) = \{s : \text{active element } E_k \text{ fired at time } s \text{ and } 0 \leq t - s - \tau_{ki} < \omega_{ki}\}$. Let $|W_{ki}(t)|$ denote the number of elements in the set $W_{ki}(t)$. If $W_{ki}(t) = \emptyset$, then $|W_{ki}(t)| = 0$.
- A collection of input functions, $\{\phi_{ki} : k \in \Gamma \cup \Omega\}$, each a function of time, and each representing pulses coming from computational active elements, and input active elements. The value of the input function is computed as $\phi_{ki}(t) = |W_{ki}(t)|A_{ki}(t)$.
- The refractory periods, transmission times and pulse widths are positive integers; and pulse amplitudes and thresholds are integers. The time t – that these parameters are a function of i.e. $\theta_i(t), r_i(t), A_{ki}(t), \omega_{ki}(t), \tau_{ki}(t)$ – is an element of the extended integers $\overline{\mathbb{Z}}$.

Input active elements that are not computational active elements have the same characteristics as computational active elements, except they have no inputs ϕ_{ki} coming from active elements in this machine. In other words, they don't receive pulses from active elements in this machine. Input active elements are assumed to be externally fireable. An external source such as the environment or an output active element from another distinct machine $\mathcal{M}(\mathcal{I}', \mathcal{E}', \mathcal{O}')$ can cause an input active element to fire. The input active element can fire at any time as long as the current time minus the time the input active element last fired is greater than or equal to the input active element's refractory period.

An active element, E_i , can be an input active element and a computational active element. Similarly, an active element can be an output active element and a computational active element. Alternatively, when an output active element, E_i , is not a computational active element, where $i \in \Delta - \Omega$, then E_i does not send pulses to active elements in this machine.

Example 2.2. *Overlapping pulses with different firing times*

Consider the four element machine where X, Y , and Z are input active elements and B is a computational active element. The parameters of the elements and their connections are shown in tables 1 and 2.

Input elements X, Y , and Z are externally fired at times 4, 1 and 2, respectively. At time $t = 3$, pulses created by Y and Z are travelling to B but have not yet arrived. B

Table 1: Element Parameter Values

Element	Threshold	Refractory	Firing Times
X		1	4
Y		1	1
Z		1	2
B	10	2	5

X , Y , and Z show no thresholds, since they are input elements.

Table 2: Connection Parameter Values

Connection	From	To	Amplitude	Width	Transmission Time
XB	X	B	3	4	1
YB	Y	B	4	2	3
ZB	Z	B	4	2	2

does not fire. At time $t = 4$, pulses created by Y and Z arrive at B . The input to B is $A_{YB}(4) + A_{ZB}(4) = 8$ which does not exceed B 's threshold. B does not fire. At time $t = 5$, pulses created by Y and Z are still at B because their pulse widths are 2. Also the pulse from X arrives. The input to B is $A_{XB}(5) + A_{YB}(5) + A_{ZB}(5) = 11$ which exceeds B 's threshold. B fires at time $t = 5$. At time $t = 7$, the refractory period of B has expired. The pulses created by Y and Z have passed through B . The pulse from X is still at B . The input to B is $A_{XB}(7) = 3$ which does not exceed B 's threshold. B does not fire at time $t = 7$.

This paragraph summarizes the machine architecture. If $g_i(s) = 1$, this means active element E_i fired at time s . The *refractory period*, r_i , is the amount of time that must elapse after active element E_i just fired before E_i can fire again. The *transmission time*, τ_{ki} , is the amount of time it takes for active element E_i to find out that active element E_k has fired. The *pulse amplitude*, A_{ki} , represents the strength of the pulse that active element E_k transmits to active element E_i after active element E_k has fired. After this pulse reaches E_i , the *pulse width* ω_{ki} represents how long the pulse lasts as input to active element E_i . At time s , the *connection* from E_k to E_i represents the triplet $(A_{ki}(s), \omega_{ki}(s), \tau_{ki}(s))$. If $A_{ki} = 0$, then there is no connection from active element E_k to active element E_i .

3. FOUR ACTIVE ELEMENT MACHINE COMMANDS AND SYNTAX

This section shows how to explicitly program an active element machine with four AEM commands: `Element`, `Connection`, `Fire`, and `Program`.

Definition 3.1. *AEM Program*

In Backus-Naur form, an AEM program is defined as follows.

```

<AEM_program> ::= <cmd_sequence>

<cmd_sequence> ::= "" | <AEM_cmd><cmd_sequence>
                | <program_def><cmd_sequence>

<AEM_cmd> ::= <element_cmd> | <fire_cmd> | <meta_cmd>

```

| <cnct_cmd> | <program_cmd>

Definition 3.2. *AEM Symbols and Extended Integer Expressions*

In Backus-Naur form, the AEM symbols are defined as follows.

```

<ename> ::= "" | <int> | <symbol>
<symbol> ::= <symbol_string> | (<ename> . . . <ename>)
<symbol_string> ::= "" | <char_symbol><str_tail>
<str_tail> ::= "" | <char_symbol><str_tail> | 0<str_tail>
              | <pos_int><str_tail>

<char_symbol> ::= <letter> | <special_char>
<letter> ::= <lower_case> | <upper_case>
<lower_case> ::= a | b | c | d | e | f | g | h | i | j | k | l | m
              | n | o | p | q | r | s | t | u | v | w | x | y | z
<upper_case> ::= A | B | C | D | E | F | G | H | I | J | K | L | M
              | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<special_char> ::= "" | _

```

These rules represent the extended integers, addition and subtraction.

```

<int> ::= <pos_int> | <neg_int> | 0
<neg_int> ::= - <pos_int>
<pos_int> ::= <non_zero><digits>
<digits> ::= <numeral> | <numeral><digits>
<non_zero> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<numeral> ::= "" | <non_zero> | 0
<aint> ::= <aint> <math_op> <d> | <d> <math_op> <aint> | <d>
<math_op> ::= + | -
<d> ::= <int> | <symbol_string> | <infinitesimal>
<infinitesimal> ::= dT

```

Definition 3.3. *Element*

An **Element** command specifies the time when an active element's values are updated or created. This command has the following Backus-Naur syntax.

```

<element_cmd> ::= (Element (Time <aint>) (Name <ename>)
                  (Threshold <int>)(Refractory <pos_int>)(Last <int>))

```

The keyword **Time** tags the time integer value s at which the element is created or updated. If the name symbol value is **E**, the keyword **Name** tags the name **E** of the active element. The keyword **Threshold** tags the threshold $\theta_E(s)$ assigned to **E**. **Refractory** tags the refractory value $r_E(s)$. The keyword **Last** tags the last time fired value $\Psi_E(s)$.

The following is an example of an element command.

```
(Element (Time 2) (Name H) (Threshold -3) (Refractory 2) (Last 0))
```

At time 2, if active element **H** does not exist, then it is created. Active element **H** has its threshold set to -3 , its refractory period set to 2, and its last time fired set to 0. After time 2, active element **H** exists indefinitely with threshold = -3 , refractory = 2 until a new **Element** command whose name value **H** is executed at a later time; in this case, the **Refractory**, **Threshold** and **Last** values specified in the new command are updated.

Definition 3.4. Connection

A **Connection** command creates or updates a connection from one active element to another active element. This command has the following Backus-Naur syntax.

```
<cncnt_cmd> ::= (Connection (Time <aint>)(From <ename>)(To <ename>)
                [(Amp <int>)(Width <pos_int>)(Delay <pos_int>)] )
```

The keyword **Time** tags the time value s at which the connection is created or updated. The keyword **From** tags the name F of the active element that sends a pulse with these updated values. The keyword **To** tags the name T of the active element that receives a pulse with these updated values. The keyword **Amp** tags the pulse amplitude value $A_{FT}(s)$ that is assigned to this connection. The keyword **Width** tags the pulse width value $\omega_{FT}(s)$. The keyword **Delay** tags the transmission time $\tau_{FT}(s)$.

When the AEM clock reaches time s , F and T are name values that must be the name of an element that already has been created or updated before or at time s . Not all of the connection parameters need to be specified in a connection command. If the connection does not exist beforehand and the **Width** and **Delay** values are not specified appropriately, then the amplitude is set to zero and this zero connection has no effect on the AEM computation. Observe that the connection exists indefinitely with the same parameter values until a new connection is executed at a later time between **From** element F and **To** element T .

The following is an example of a connection command.

```
(Connection (Time 2) (From C) (To L) (Amp -7) (Width 1) (Delay 3))
```

At time 2, the connection from active element C to active element L has its amplitude set to -7 , its pulse width set to 1, and its transmission time set to 3.

Definition 3.5. Fire

The **Fire** command has the following Backus-Naur syntax.

```
<fire_cmd> ::= (Fire (Time <aint>) (Name <ename>)) )
```

The **Fire** command fires the active element indicated by the **Name** tag at the time indicated by the **Time** tag. This command is primarily used to fire input active elements in order to communicate program input to the active element machine.

An example is `(Fire (Time 3) (Name C))`, which fires active element C at $t=3$.

Definition 3.6. Program

The **Program** command is convenient when a sequence of commands are used repeatedly. This command combines a sequence of commands into a single command. It has the following definition syntax.

```
<program_def> ::= (Program <pname> [(Cmds <cmds>)] [(Args <args>)]
                  <cmd_sequence> )
```

```
<pname> ::= <ename>
```

```
<cmds> ::= <cmd_name> | <cmd_name><cmds>
```

```
<cmd_name> ::= Element | Connection | Fire | Meta | <pname>
```

```
<args> ::= <symbol> | <symbol><args>
```

The **Program** command has the following execution syntax.

```
<program_cmd> ::= (<pname> [(Cmds <cmds>)] [(Args <args_cmd>)] )
```

```
<args_cmd> ::= <ename> | <ename><args_cmd>
```

The `FireN` program is an example of definition syntax.

```
(Program FireN (Args t E)
 (Element (Time 0) (Name E)(Refractory 1)(Threshold 1)(Last 0))
 (Connection (Time 0) (From E) (To E)(Amp 2)(Width 1)(Delay 1))
 (Fire (Time 1) (Name E))
 (Connection (Time t+1) (From E) (To E) (Amp 0))
 )
```

The execution of the command `(FireN (Args 8 E1))` causes element `E1` to fire 8 times at times 1, 2, 3, 4, 5, 6, 7, and 8 and then `E1` stops firing at time = 9.

4. COPY AND NAND PROGRAM EXAMPLES

The first example describes an active element copy program. The second example describes a nand program.

Example 4.1. *Copy Program*

This active element program copies an element's firing state to another element.

```
(Program copy (Args s t b a)
 (Element (Time s-1) (Name b) (Threshold 1) (Refractory 1) (Last s-1))
 (Connection (Time s-1) (From a) (To b) (Amp 0) (Width 0) (Delay 1))
 (Connection (Time s) (From a) (To b) (Amp 2) (Width 1) (Delay 1))
 (Connection (Time s) (From b) (To b) (Amp 2) (Width 1) (Delay 1))
 (Connection (Time t) (From a) (To b) (Amp 0) (Width 0) (Delay 1))
 )
```

When the copy program is called, active element `b` will start firing if `a` fired during the window of time $[s, t)$. Further, a connection is set up from `b` to `b` so that `b` will keep firing indefinitely. This enables `b` to *store* active element `a`'s firing state.

Example 4.2. *Nand Program*

This active element program computes a nand circuit.

```
(Program nand (Args s x y l h)
 (Element (Time s) (Name x) (Threshold 0) (Refractory 1) (Last s))
 (Element (Time s) (Name y) (Threshold 0) (Refractory 1) (Last s))
 (Element (Time s) (Name h) (Threshold -3) (Refractory 2) (Last s))
 (Element (Time s) (Name l) (Threshold 3) (Refractory 2) (Last s))
 (Connection (Time s) (From x) (To l) (Amp 2) (Width 1) (Delay 1))
 (Connection (Time s) (From x) (To h) (Amp -2) (Width 1) (Delay 1))
 (Connection (Time s) (From y) (To l) (Amp 2) (Width 1) (Delay 1))
 (Connection (Time s) (From y) (To h) (Amp -2) (Width 1) (Delay 1))
 )
```

`B` and `C` are input elements. `L` (represents a 0 output) and `H` (represents a 1 output) are output elements. The call `(nand (Args -1 B C L H))` creates the connections between `B`, `C`, `L` and `H`. All four cases are verified where $|$ denotes the Sheffer stroke.

- (1) At $t = 0$, active elements `B` and `C` do not fire i.e. $B | C = 0|0 = 1$. Since the threshold of `H` is -3 , `H` fires at time $t = 1$.

- (2) At $t = 0$, active element B fires and C does not fire i.e. $B | C = 1|0 = 1$. Since one pulse with amplitude -2 reaches H at $t = 1$ just as the refractory period expires and $-2 > -3$, then H fires at time $t = 1$.
- (3) At $t = 0$, active element B does not fire and C fires i.e. $B | C = 0|1 = 1$. Since one pulse with amplitude -2 reaches H at $t = 1$ just as the refractory period expires and $-2 > -3$, then H fires at time $t = 1$.
- (4) At $t = 0$, active elements B and C both fire i.e. $B | C = 1|1 = 0$. At time $t = 1$, two pulses with amplitude -2 reach H so H does not fire. At time $t = 1$, two pulses reach L, and each pulse has amplitude 2. L has threshold = 3, so L fires at $t = 1$.

ACKNOWLEDGEMENT

I would like to thank Michael Jones, Alan Langman, David Lewis, Alex Mayer, Lutz Mueller and Don Saari for their helpful advice. The original and complete version of the Active Element Machine will soon be available at <http://www.springerlink.com> in the Computational Intelligence series.

APPENDIX

In the following Turing Machine definition, the Turing program η is explicitly represented as a function instead of quintuples ([3], [15]).

Definition 4.3. *Turing Machine*

A Turing machine is a triple (Q, A, η) where

- Q is a finite set of states that does not contain the halt state. The states are sometimes represented as natural numbers $Q = \{2, \dots, K\}$. There is a unique halt state, represented as h or as 1.
- When machine execution begins, the machine is in an initial state s and $s \in Q$.
- A is a finite set of alphabet symbols that are read from and written to the tape.
- -1 and $+1$ represent advancing the tape head to the left or right square, respectively.
- η is a function where $\eta : Q \times A \rightarrow (Q \cup \{h\}) \times A \times \{-1, +1\}$. η acts as the program for the Turing machine. For each q in Q and α in A , $\eta(q, \alpha) = (r, \beta, x)$ describes how machine (Q, A, η) executes one computational step. When in state q and scanning alphabet symbol α on the tape:
 - Machine (Q, A, η) changes to state r .
 - Machine (Q, A, η) rewrites alphabet symbol α as symbol β on the tape.
 - If $x = -1$, then machine (Q, A, η) moves its tape head one square to the left on the tape and is subsequently scanning the symbol in this square.
 - If $x = +1$, then machine (Q, A, η) moves its tape head one square to the right on the tape and is subsequently scanning the symbol in this square.
 - If $r = h$, machine (Q, A, η) enters the halting state h , and the machine halts.

Definition 4.4. *Turing Machine tape*

The Turing machine tape T is represented as a function $T : \mathbb{Z} \rightarrow A$ where \mathbb{Z} denotes the integers. The tape T is M -bounded if there exists a bound $M > 0$ such that $T(k) = T(j)$ whenever $|k|, |j| \geq M$. The Turing machine definitions in [3] and [15] assume the initial tape, before program execution begins, is M -bounded and the tape contains only blank symbols, denoted here as $\#$, outside the bound. The symbol on the k th square of the tape is $T(k)$.

Definition 4.5. *Turing Machine Configuration with tape head location*

Let (Q, A, η) be a Turing machine with tape T . A configuration is an element of the set $C = (Q \cup \{h\}) \times \mathbb{Z} \times \{T : T \text{ is tape with range } A\}$. If (q, k, T) is a configuration, then k is called the tape head location.

Consider the configuration $(p, 2, \dots \#\#\alpha\beta\#\#\dots)$. The 1st coordinate indicates that the Turing machine is in state p . The 2nd coordinate indicates that its tape head is currently scanning tape square 2, denoted as $T(2)$. The 3rd coordinate indicates that tape square 1 contains symbol α , tape square 2 contains symbol β , and all other tape squares contain the $\#$ symbol.

Definition 4.6. *Turing Machine Computational Step*

Given Turing machine (Q, A, η) in current configuration (q, k, T) such that $T(k) = \alpha$. After the execution of one computational step, the new configuration is determined by one and only one of the four cases

- (1) $(r, k - 1, S)$ if $\eta(q, \alpha) = (r, \beta, -1)$ for non-halting state r
- (2) $(r, k + 1, S)$ if $\eta(q, \alpha) = (r, \beta, +1)$ for non-halting state r
- (3) $(h, k + 1, S)$ if $\eta(q, \alpha) = (h, \beta, +1)$ for halting state h
- (4) $(h, k - 1, S)$ if $\eta(q, \alpha) = (h, \beta, -1)$ for halting state h

such that for all four cases the new tape $S(j) = T(j)$ whenever $j \neq k$ and $S(k) = \beta$. In cases (3) and (4), the machine execution halts.

If the machine is currently in configuration (q_0, k_0, T_0) and over the next n steps the sequence of machine configurations (points) is $(q_0, k_0, T_0), (q_1, k_1, T_1), (q_2, k_2, T_2), \dots, (q_n, k_n, T_n)$, then this execution sequence is sometimes called the next $n + 1$ computational steps.

REFERENCES

- [1] Harold Abelson and Gerald Jay Sussman. Structure and Interpretation of Computer Programs. Cambridge, MA. MIT Press, (1996)
- [2] S.A. Cook. The complexity of theorem proving procedures. Proceedings, Third Annual ACM Symposium on the Theory of Computing. ACM, New York, 151–158 (1971)
- [3] Martin Davis. Computability and Unsolvability. Dover Publications, New York, (1982)
- [4] Michael Stephen Fiske. Effector Machine Computation. US 7,398,260 B2 (2004). Provisional 60/456,715 (2003) (See <http://tinyurl.com/6l5wuhz> or <http://tinyurl.com/6zzly8p>)
- [5] Michael Stephen Fiske. Active Element Machine Computation. US Application 20070288668 (2007) (See <http://tinyurl.com/62gv8ke> or <http://tinyurl.com/6hz8by4>)
- [6] Michael R. Garey and David S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, (1979)
- [7] John Hertz, Anders Krogh and Richard G. Palmer. Introduction To The Theory of Neural Computation. Addison-Wesley Publishing Company. Redwood City, California, (1991)

- [8] Edward A. Lee. Computing Needs Time. Technical Report No. UCB/EECS-2009-30. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-30.html>, (electronic). Electrical Engineering and Computer Sciences, University of California at Berkeley (2009)
- [9] Warren S. McCulloch, Walter Pitts. A logical calculus immanent in nervous activity. *Bulletin of Mathematical Biophysics*. **5**, 115–133 (1943)
- [10] Marvin Minsky. *Computation: Finite and Infinite Machines* (1st edition). Englewood Cliffs, NJ. Prentice-Hall, Inc, (1967)
- [11] Wilfrid Rall. *The Theoretical Foundation of Dendritic Function. Selected Papers of Wilfrid Rall with Commentaries*. Edited by Idan Segev, John Rinzel, and Gordon Shepherd. MIT Press. Cambridge, Massachusetts, (1995)
- [12] F. P. Ramsey. On a problem of formal logic. *Proc. London Math. Soc. Series 2* **30**, 264–286 (1930)
- [13] Abraham Robinson. *Non-standard Analysis*. Revised Edition. Princeton, NJ. Princeton University Press, (1996)
- [14] H. E. Sturgis, and J. C. Shepherdson. Computability of Recursive Functions. *J. Assoc. Comput. Mach.* **10**, 217–255 (1963)
- [15] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc. Series 2* **42** (Parts 3 and 4), 230–265 (1936). A correction, *ibid.* **43**, 544–546 (1937).