Dynamical Systems that Heal

Michael Stephen Fiske
Aemea Institute
mf@aemea.org

Abstract

Malware plays a key role in attacking critical infrastructure. With this problem in mind, we introduce systems that heal from a broader perspective than the standard digital computer model: Our goal in a more general theory is to be applicable to systems that contain subsystems that do not solely rely on the execution of register machine instructions. Our broader approach assumes a dynamical system that performs tasks.

Our primary contribution defines a principle of self-modifiability in dynamical systems and demonstrates how it can be used to heal a malfunctioning dynamical system. As far as we know, to date there has not been a mathematical notion of self-modifiability in dynamical systems; hitherto there has not been a formal system for describing how to heal damaged computer instructions or to heal differential equations that perform tasks.

1. Introduction

Malware plays a key role in attacking computer systems [1]. Typically, malware sabotages the purpose of the machine instructions executing on a digital computer. Also, there is no computational mechanism in the register machine model [2, 3, 4][1] for healing (self-repairing) damaged instructions.

Our goal is to study systems that heal from a broader perspective than the register machine model so that our methods may also be applied to systems [5], which have subsystems that do not solely rely on executing digital computer instructions. Some systems can be implemented with *analog machines* such as an Archimedean screw [6].[2]

At any moment, a register machine's state lies in a countable topological space; on the other hand, some physical systems [7] are more aptly modelled by a flow on a metric space [8] that is a continuum.[3]

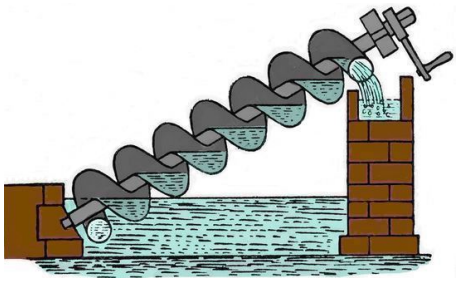

Figure 1. An Archimedean Screw.

We assume a system exists that performs tasks: for example, the energy system in mitochondria, or an autonomous transportation system. Our primary goal is to develop and better understand methods of repairing a system when the system is malfunctioning. In the field of biology, a *malfunctioning system*, in some cases, is called a *disease*. With our goal in mind, we ask the following two questions:

> What is *self-modifiability*?
>
> How can *self-modifiability* be mathematically modelled as a means for healing a system?

As far as we know, hitherto there has not been a notion of self-modifiable differential equations; to date there has not been a scientific language or formal system for describing how to heal differential equations or computer instructions. This is an important notion to develop so that we can design and improve systems that protect our critical infrastructure and health: biological systems, energy systems, GPS, manufacturing systems, and transportation systems.

[1] On pages 124-163 of [2], Knuth describes a specific register machine model, called MIX. Processor architectures, such as the Intel Core i7 and ARM Cortex-A8, are covered in [4]; these architectures specify physical realizations of the register machine model.

[2] Figure 1 shows an Archimedean screw: In [6], equation (10) models the volume of fluid flow passing through an Archimedean screw; equation (10) is an analytic equation over the real numbers.

[3] A *continuum* is often defined as a compact, connected, metric space or a compact, connected, Hausdorff space [9]. Sometimes a continuum is not compact. The real numbers are a continuum.

URI: https://hdl.handle.net/10125/103442
978-0-9981331-5-7

Our primary contribution defines the principle of self-modifiability in dynamical systems and demonstrates how it can be used to heal a malfunctioning dynamical system. A computational model is described that can simultaneously execute multiple machine instructions and a software simulation is provided. The machine's simultaneous execution property enables it to repair instructions that have been damaged or removed; specifically, this machine uses self-modifiability (via `meta` instructions) to help heal damaged instructions.

Another contribution provides examples of how self-modifiability can be understood in terms of classical mathematics. Computation is typically implemented with a dynamical system that performs a task. We define *meta variables* and *meta operators*, and show how to add or replace variables and differential equations. *Meta variables and meta operators provide a general method for healing dynamical systems, based on differential equations.*

2. Motivating Healing

Figure 2 shows a doorbell circuit [10]. This system can be damaged by shorting the two inputs of the speaker; alternatively, degradation of the insulation around the speaker leads can lead to a malfunctioning doorbell system.

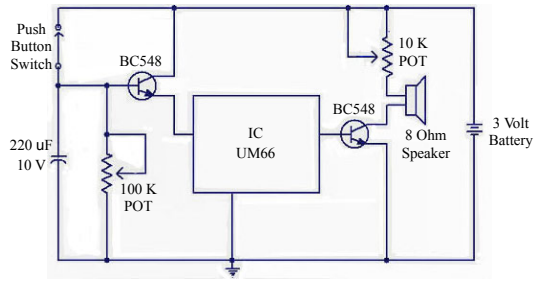

Figure 2. Doorbell Circuit

The circuit can be modelled with a set of differential equations. When the speaker leads are shorted or cut, the differential equations have changed. Ideally, we are searching for methods to repair the equations. Our goal is to heal them back to differential equations that model a working doorbell system.

3. The Principle of Self-Modifiability

A computer program can be viewed as a discrete dynamical system [11]. A set of differential equations specifies a different type of dynamical system that can sometimes model analog machines. When a dynamical system that performs a task is changed so that the task is no longer adequately performed, it is desirable to have a process or mechanism to heal the system. This means that part of the system should detect a change and part of the system should *self-modify* the damaged system back to the original system that adequately performed the task. A dynamical system that can change itself is called *self-modifiable*.

A more concrete way to think about this is to view a dynamical system as a system that is governed by a collection of rules. In the case of differential equations, each equation is a rule; in a computational machine, each machine instruction is a rule. A dynamical system is self-modifiable if it can change its own rules: this means the dynamical system can add new rules to itself; it can replace current rules with new rules; or it can delete rules. Usually, a self-modifiable dynamical system is *non-autonomous* because the rules governing its behavior can change as a function of time.

In section 5, we describe a computational machine that can add new rules when a "dynamical event" occurs during the machine's execution. Overall, our machine uses "meta commands" to change its own rules.

4. Related Work

Related work on self-repair and self-modification is split between biology and computer science. Biology focuses on understanding biological complexity [12, 13] and DNA repair [14]. Computer science emphasizes better implementations of code repair and AI.

DNA repair [15] has been extensively studied for over fifty years. The models of DNA explain repair at the biomolecular level [14], but they do not propose a self-modifiability principle, based on adding new rules or replacing rules; and they do not provide a mathematical model of healing a dynamical system.

In [16], self-modifying systems are proposed as a means for understanding the development of complexity in biology: "**the popular belief is that the increase of complexity in evolution poses no deep or unsolved problem.**" Subsequently, the author observes, "**the increase of complexity becomes a deep problem**", and paradoxically states, "We shall see that **in a formal system complexity can never increase at all.**"

In [17], a semantic code search method is described whose goal is to repair register machine programs. The method uses human-coded fragments in a database to repair buggy `C` programs, written by novice students. It has a 19 percent success rate on benchmarked bugs. It has no principle of self-modifiability. In [18], "self-modifiable algorithms" perform AI: explicit time and quantum randomness [19] are not part of the model. Self-modifiable differential equations are not proposed.

5. The Active Element Machine

Dendritic integration [20, 21] inspired the active element machine (AEM) model [22]. Active elements compute simultaneously. AEM commands specify time, while standard compilers [23] and language theory [24] do not specify time. In native hardware, an AEM can simultaneously execute 100,000 instructions or more, and eliminate the von Neumann bottleneck [25].

First, we summarize the AEM.[4] Then a small AEM program is described that computes if a complete monochromatic graph on 3 vertices exists. Then we show how self-modifiability can repair a program.

An AEM is composed of computational objects called *active elements* or *elements*. There are three kinds of elements: *input*, *computational* and *output* active elements. Input elements process information received from the environment or another AEM. Computational elements receive pulses from the input elements and other computational elements and transmit new pulses to computational and output active elements. Output elements receive pulses from input and computational elements. Every element is active in the sense that each one can receive and transmit pulses simultaneously.

Each pulse has an *amplitude* and a *width*. The width represents how long the amplitude lasts as input to the element that receives the pulse. If element E_i simultaneously receives pulses with amplitudes summing to a value greater than E_i's threshold and E_i's refractory period has expired, then E_i fires. When E_i fires, it sends pulses to other elements. If E_i fires at time t, a pulse reaches element E_k at time $t+\tau_{ik}$ where τ_{ik} is the *delay* (transmission time) from E_i to E_k.

In section 8.2, a specification of the AEM language defines four commands and two keywords. An *element* command creates a new active element or updates an element's parameters: `(element (time 2) (name L) (threshold -3) (refractory 2) (last 0))`.[5] At time $t = 2$, if element `L` does not exist, then `L` is created. Element `L`'s threshold is set to -3. `L`'s refractory period is set to 2, and `L`'s last time fired is set to 0. After time $t = 2$, element `L` exists indefinitely with the same threshold and refractory values until a new element command with name value `L` executes at a later time.

A *connection* command connects two elements: `(connection (time 2) (from C) (to L) (amp -7) (width 6) (delay 3))`. At time $t = 2$, a connection from element `C` to element `L` is created. The pulse amplitude is set to -7. The pulse width is set to 6. The transmission time is set to 3.

[4] A more comprehensive description is in sections 8.1 and 8.2.

[5] In input element commands, parameters *threshold*, *refractory* and *last* may be omitted.

A *fire* command fires an input element and sends input, by transmitting pulses to other active elements: `(fire (time 3) (name C))` fires element `C` at $t = 3$.

In `(name v)`, value `v` uniquely identifies the active element in element and fire commands. In expression `(from v) (to w)`, values `v` and `w` uniquely identify a connection: `(connection (time 2) (from v) (to w) (amp 7) (width 6) (delay 3))`.

A *meta* command can self-modify an AEM program: `(meta (name E) (window 1 5) (C (args a b)))`. If element `E` fires (a dynamical event) during the time window $[1, 5]$, this meta command executes and creates command `C` with arguments `a` and `b`.

Suppose an element command is contained in a meta command and an element command with the same name value already exists in the AEM program. If the meta command executes, then the element command in the `meta` command replaces the one in the AEM program. A similar replacement occurs for `fire` commands if the name values are the same. A similar replacement occurs for a `connection` command in a `meta` command if the `from` and `to` values are the same.

Keyword `dT` represents an infinitesimal [26] amount of time that helps coordinate almost simultaneous events. `dT` $< q$ for every rational $q > 0$, and `dT` > 0.

5.1. A Simple AEM Program

We define a program with 4 input active elements `A`, `B`, `C`, and `D` and one computational element `E`.

```
(element (time -dT) (name E) (threshold 9)
         (refractory 4) (last 0) )
(element (time -dT) (name A))
(element (time -dT) (name B))
(element (time -dT) (name C))
(element (time -dT) (name D))

(connection (time -1) (from A) (to E) (amp 5)
                      (width 4+2dT) (delay 3-dT))
(connection (time -1) (from B) (to E) (amp 9)
                      (width 6) (delay 2+dT))
(connection (time -1) (from C)  (to E) (amp -4)
                      (width 5) (delay 5-dT))
(connection (time -1) (from D) (to E) (amp 2)
                      (width 4+2dT) (delay 6-dT))

(fire (time 0) (name A))  (fire (time 0) (name B))
(fire (time 0) (name C))  (fire (time 0) (name D))
(fire (time 0) (name E))
```

The fire commands cause elements `A`, `B`, `C`, `D` and `E` to fire at $t = 0$. The parameters of the elements and their connections are shown in tables 1 and 2.

Table 1. Element Parameter Values

element	threshold	refractory	last firing time
E	9	4	0

Figure 3 shows the resulting pulses sent from input elements `A`, `B`, `C`, `D` to element `E`. The pulse sent from

Table 2. Connection Parameter Values

name	from	to	amplitude	width	delay
AE	A	E	5	$4+2$dT	$3-$dT
BE	B	E	9	6	$2+$dT
CE	C	E	-4	5	$5-$dT
DE	D	E	2	$4+2$dT	$6-$dT

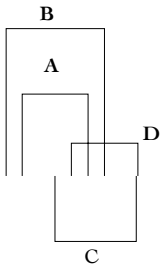

Figure 3. Pulses Sent to Element E.

element B reaches element E at time $2+$dT. The pulse sent from A reaches E at time $3-$dT. Figure 3 also shows a *time geometry* of the pulses sent to element E.

Table 3 shows the sum of the pulses received by E at various times. At time $t = 3$, E does not fire because E's refractory period is 4 and the last time E fired was $t = 0$.

Table 3. Sum of Input Pulses to E

Time	2	3	4	5	8	9	12
Sum of Pulses	0	14	14	10	7	-2	0

At $t = 4$, E fires because the sum of E's input is 14 and 14 is greater than E's threshold of 9. Since E fires at $t = 4$, E cannot fire again until $t = 8$ or afterward. At time $t = 8$, the sum of E's input is 7, and 7 is less than E's threshold. Hence, E does not fire a second time.

5.2. An AEM that Computes Colorings of K_5.

We describe an AEM program that computes a simple graph theory problem. In section 5.3, we show how to use meta commands to help repair commands that have been removed from a valid AEM program.

K_5 is the complete graph on 5 vertices, $\{1, 2, 3, 4, 5\}$. In figure 4, each edge $\{j, k\}$ is colored red or blue. The red edges are $\{1, 2\}$, $\{2, 3\}$, $\{3, 4\}$, $\{4, 5\}$, and $\{1, 5\}$. The blue edges are $\{1, 3\}$, $\{1, 4\}$, $\{2, 4\}$, $\{2, 5\}$, and $\{3, 5\}$.

We specify an AEM program that determines whether a coloring of K_5 contains a complete monochromatic subgraph on 3 vertices (triangle). Monochromatic means the subgraph has only blue edges or only red edges. In figure 4, the coloring does not

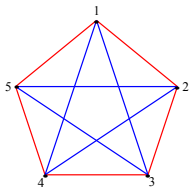

Figure 4. A Red-Blue Coloring of Graph K_5

contain a monochromatic subgraph on 3 vertices.

This graph problem comes from Ramsey theory [27], which has extensive applications [28] in computer science, ergodic theory, information theory, logic, and number theory. Ramsey number $r(j, l)$ is the least integer n such that if the edges of the complete graph K_n are colored with only red and blue, then there always exists a complete subgraph K_j containing only red edges or there exists a complete subgraph K_l with only blue edges. Computing $r(j, l)$ is an NP-hard problem [29]. $r(j, l)$ grows so fast that $r(5, 5)$ is unknown.

Below are 5 *red edge* input element commands. The one named R12 represents that edge $\{1, 2\}$ is red.

```
(element (time 0) (name R12) (threshold 1)
                  (refractory 1) (last -1))
(element (time 0) (name R23) (threshold 1)
                  (refractory 1) (last -1))
(element (time 0) (name R34) (threshold 1)
                  (refractory 1) (last -1))
(element (time 0) (name R45) (threshold 1)
                  (refractory 1) (last -1))
(element (time 0) (name R15) (threshold 1)
                  (refractory 1) (last -1))
```

Below are 5 *blue edge* input element commands, where Bij ranges over {B13, B14, B24, B25, B35}.

```
(element (time 0) (name Bij) (threshold 1)
                  (refractory 1) (last -1))
```

Below are 5 commands that fire the *red edge* elements, and 5 commands that fire the *blue edge* elements.

```
(fire (time 0) (name R12)) (fire (time 0) (name R23))
(fire (time 0) (name R34)) (fire (time 0) (name R45))
(fire (time 0) (name R15)) (fire (time 0) (name B13))
(fire (time 0) (name B14)) (fire (time 0) (name B24))
(fire (time 0) (name B25)) (fire (time 0) (name B35))
```

For each red edge $\{j, k\}$, there is a meta command

```
(meta (name Rjk) (window 0 1) (connection
      (time 0) (from Rjk) (to Rjk)
      (amp 2) (width 1) (delay 1)))
```

where Rjk is one of R12, R23, R34, R45, R15.

For each blue edge $\{j, k\}$, there is a meta command

```
(meta (name Bjk) (window 0 1) (connection
      (time 0) (from Bjk) (to Bjk)
      (amp 2) (width 1) (delay 1)))
```

where Bjk is one of B13, B14, B24, B25, B35.

For each 3 distinct vertices $\{i, j, k\}$ in $\{1, 2, 3, 4, 5\}$, create red element R_ijk and blue element B_ijk.

```
(element (time 0) (name R_ijk) (threshold 5)
                  (refractory 1) (last -1))
```

```
(element (time 0) (name B_ijk) (threshold 5)
                  (refractory 1) (last -1))
```

For each set of 3 distinct vertices $\{i, j, k\}$, compute if all three edges $\{i, j\}$, $\{i, k\}$, $\{j, k\}$ are blue.

```
(connection (time 0) (from Bij) (to B_ijk)
            (amp 2) (width 1) (delay 1) )
(connection (time 0) (from Bjk) (to B_ijk)
            (amp 2) (width 1) (delay 1) )
(connection (time 0) (from Bik) (to B_ijk)
            (amp 2) (width 1) (delay 1) )
```

For each set of three distinct vertices $\{i, j, k\}$, compute if all three edges $\{i, j\}$, $\{i, k\}$, $\{j, k\}$ are red.

```
(connection (time 0) (from Rij) (to R_ijk)
            (amp 2) (width 1) (delay 1) )
(connection (time 0) (from Rjk) (to R_ijk)
            (amp 2) (width 1) (delay 1) )
(connection (time 0) (from Rik) (to R_ijk)
            (amp 2) (width 1) (delay 1) )
```

5.3. Simulating an AEM that Self-Repairs

We built a software tool that simulates the active element machine. We compiled our `C` code [30] to a dynamic library (59,552 bytes), named `AEM.dylib`. Library `AEM.dylib` runs on macOS Big Sur, and can execute all four AEM commands.

The AEM can simultaneously execute multiple commands, while `C` code compiles to register machine instructions that can only execute sequentially. By finessing the use of `dT` in our AEM programs, we can avoid simulations that produce different computing results than an actual AEM program, implemented in hardware, would produce. Before demonstrating self-repair, we describe two software simulations of our AEM program defined in section 5.2.

During the first simulation, the following firing activity occurs. At time 0, elements `B13`, `B14`, `B24` `B25`, `B35`, `R12`, `R15`, `R23`, `R34`, and `R45` fire. None of the elements `B_ijk` or `R_ijk` ever fire: thus, K_5's coloring does not contain a blue or red triangle.

If $\{3, 5\}$ is colored red instead of blue, command `(fire (time 0) (name R35))` replaces `(fire (time 0) (name B35))`. Also, `(element (time 0) (name R35) (threshold 1) (refractory 1) (last -1))` replaces `(element (time 0) (name B35) (threshold 1) (refractory 1) (last -1))`.

During the second simulation, the new AEM program produces the following firing activity. At time $t = 0$, elements `B13`, `B14`, `B25`, `B24`, `R12`, `R15` `R23`, `R34`, `R35` and `R45` fire. Also, element `R_345` fires at $t = 1$: This means that the new coloring of K_5 has a red triangle $\{3, 4, 5\}$.

Now we show how the new AEM program can be protected. Suppose the new AEM program is sabotaged by removing the connection command:

```
(connection (time 0) (from R35) (to R_345)
                     (amp 2) (width 1) (delay 1) )
```

When element `R35` fires, the pulse of amplitude 2 is not sent to `R_345` and the input sum to `R_345` never goes above 4, so `R_345` never fires even though the new coloring of K_5 contains the red triangle $\{3, 4, 5\}$.

We can heal the removal of this connection with the following meta command:

```
(meta (name R35)  (window -dT 1)
      (connection (time 0) (from R35) (to R_345)
                  (amp 2) (width 1) (delay 1) )
```

Similarly, for each connection from `Rij` to `R_ijk`, we can heal the removal with the meta command:

```
(meta (name Rij)  (window -dT 1)
      (connection (time 0) (from Rij) (to R_ijk)
                  (amp 2) (width 1) (delay 1) )
```

We can add a similar meta command for each connection from `Rjk` to `R_ijk` and for each connection from `Rik` to `R_ijk`. We can also protect the *blue connections* by adding a meta command for each connection from `Bij` to `B_ijk`.

```
(meta (name Bij)   (window -dT 1)
      (connection  (time 0) (from Bij) (to B_ijk)
                   (amp 2) (width 1) (delay 1) )
```

We can add a similar meta command to heal broken connections for each connection from `Bjk` to `B_ijk` and for each connection from `Bik` to `B_ijk`.

5.4. Healing Principle for AEM programs

A *Turing computable* property is a property that can be computed by a register machine. Since AEM programs can execute Turing computable algorithms [22], the healing examples demonstrated in 5.3 can be extended with more general methods. For example, a new output active element $\mathtt{O}_c$ can be adjoined to an AEM program for the sole purpose of detecting some Turing computable property `c` has changed in the damaged AEM program. One can adjoin meta commands of the form `(meta (name` $\mathtt{O}_c$`) h)` where if active element $\mathtt{O}_c$ fires, the firing of $\mathtt{O}_c$ indicates that property `c` is awry. Hence, command `h` self-modifies the AEM program so that adjoining command `h` at least partially heals the damaged AEM program.

Before we provide a formal definition, we give two examples of properties, named ϕ_1 and ϕ_2.

ϕ_1 : The number of non-zero connections in program `P` that contain expression `(to A)` is 7.

ϕ_2 : Suppose `P` is the AEM defined in section 5.1. If elements `A`, `B`, `C`, and `D` all fire at time s, then element `E` will fire before or at time $s + 4$.

Let $\mathcal{P}$ be the set of all AEM programs. A *property* of all programs $\mathcal{P}$ is defined as a characteristic function $\phi : \mathcal{P} \to \{0, 1\}$. When $\phi(\mathtt{P}) = 1$, then property ϕ holds

on program P. If $\phi(\texttt{P}) = 0$, then property ϕ is violated on P. ϕ is a *Turing computable property* if ϕ is Turing computable on $\mathcal{P}$. For a finite set of programs $\mathcal{F} \subset \mathcal{P}$, a restriction $\phi|_{\mathcal{F}} : \mathcal{F} \to \{0,1\}$ can be Turing computable, yet ϕ is not necessarily Turing computable on $\mathcal{P}$.

A *subprogram* S of an AEM program P is any subset of commands selected from P. An *extension program* of S is a finite set of AEM commands adjoined to S.

Theorem 1. *AEM Healing Theorem*

Let P be an AEM program. Suppose ϕ is a Turing computable property of program P. Suppose that a subprogram S of P can compute whether ϕ holds on P or is violated. Then there is an extension program H of S such that if P is damaged and H is not damaged, then H can restore program (heal) P, and S can verify that ϕ has the same value on the original program P.

Proof. Starting with program S, we explain how to build program H that can restore program P. Add a new unique active element with name value X (not in P) to program S. (The name value X must not occur in P because any firing activity of X must not interfere with firing activity of other elements in P.) For each command C in P, adjoin the following meta command to program S: `(meta (name X) C)`.

Command `(fire (time t0) (name X))` is also adjoined to S, where `t0` is the time that program H loads and starts executing. When X fires, the commands in P are restored by the meta commands in H that were adjoined to program S. Trivially, ϕ has the same value on the restored program as the original program P. □

The proof shows that the `meta` command can help restore a damaged program; and self-modifiability of AEM programs plays a fundamental role. In some cases, a Turing computable property ϕ_i can be the ith bit of a one-way hash function [31] applied to program P that checks if tampering of P has occurred.

6. Self-Modifiable Differential Equations

Before we show how to self-modify a differential equation, we describe some ordinary differential equations. Then we develop a formal language for building differential equations. Finally, we show a damaged one and illustrate how to heal it.

6.1. ODEs that Compute OR, AND, NOT

We define ordinary differential equations that can compute the Boolean operator OR, where $\texttt{OR}(1,0) = \texttt{OR}(0,1) = \texttt{OR}(1,1) = 1$, and $\texttt{OR}(0,0) = 0$. Consider equations $\frac{dx}{dt} = 0$; $\frac{dy}{dt} = 0$; and $\frac{dz}{dt} = x + y - xy - z$. The initial values x_0 and y_0 are Boolean inputs 0 or 1. Variable z computes the output. z's initial value is always $z_0 = 0$. We tested that z converges to the correct Boolean output for all 4 pairs of initial values $x_0, y_0 \in \{0,1\}$, using Julia [32]. For example, table 4 assumes initial conditions $x_0 = 0$; $y_0 = 1$; and $z_0 = 0$.

Table 4. $\frac{dx}{dt} = 0$. $\frac{dy}{dt} = 0$. $\frac{dz}{dt} = x + y - xy - z$.

t	0	.25	.49	.81	1.22	2.34	3.07	4.0
$x(t)$	0	0	0	0	0	0	0	0
$y(t)$	1	1	1	1	1	1	1	1
$z(t)$	0	.22	.56	.71	.82	.90	.95	.98

If $x_0 = 0$, $y_0 = 1$ and $z_0 = 0$, then $\frac{dz}{dt}|_{t=0} = 1$ and $\frac{dz}{dt} = 1 - z$ when $t > 0$. Thus, $\frac{dz}{dt} > 0$ when $z < 1$, and $\lim_{t\to\infty} z(t, x_0 = 0, y_0 = 1, z_0 = 0) = 1$. By symmetry, $\lim_{t\to\infty} z(t, x_0 = 1, y_0 = 0, z_0 = 0) = 1$. Also, $\lim_{t\to\infty} z(t, x_0 = 1, y_0 = 1, z_0 = 0) = 1$. Lastly, the relevant fixed points[6] (x, y, z) of $\frac{dx}{dt} = 0$; $\frac{dy}{dt} = 0$; $\frac{dz}{dt} = x + y - xy - z$ are $(0,1,1)$, $(1,0,1)$, $(0,0,0)$, $(1,1,1)$, and have the same input-output values as OR.

Equations $\frac{dx}{dt} = 0$; $\frac{dy}{dt} = 0$; and $\frac{dz}{dt} = xy - z$ compute $\texttt{AND}(0,0) = \texttt{AND}(1,0) = \texttt{AND}(0,1) = 0$, and $\texttt{AND}(1,1) = 1$. Relevant fixed points are $(0,1,0)$, $(1,0,0)$, $(0,0,0)$, $(1,1,1)$. With input x and output z, $\frac{dx}{dt} = 0$ and $\frac{dz}{dt} = 1 + xz - (x + z)$ compute NOT.

6.2. Generalizing the Meta Command

In the AEM, the firing of an element in a meta command adds a new connection or new element, or replaces some of the parameters in an existing element or connection. When meta commands create new elements and connections, these new computational objects are representable by one or more new variables. This is evident from the machine architecture definition in 8.1 of the appendix. In short, we identify two notions that are critical to self-modifying a differential equation.

1. A formal language specifies how to self-modify a differential equation. *Meta operators* are formal objects that self-modify an equation.

2. A *meta variable* helps detect an event. A detectable event triggers an execution of a *meta operator*. A meta operator alone is not sufficient for defining self-modification. A self-modifiable dynamical system must also know at what time a meta operator executes.

[6] The fixed points are all (x, y, z) such that $\frac{dx}{dt} = \frac{dy}{dt} = \frac{dz}{dt} = 0$, e.g. $(2, 2, 0)$. Sometimes fixed points are called equilibrium points.

6.3. Meta Variables

We define a *meta variable*, a *detectable event*, and a *meta execution time*. *Standard variables* are variables that occur in an ordinary differential equation. In $\frac{dz}{dt} = 1 + xz - (x + z)$, x and z are standard variables. Define $\omega_1, \ldots \omega_n$ as meta variables. Define $x_1, \ldots, x_n$ as standard variables.[7] Each f_i is a function. $\frac{d\omega_i}{dt} = f_i(x_1, \ldots, x_n)$ is a *meta equation*.

Let X be a topological space, where derivatives can be defined. At any time t, a standard variable's value $x_i(t)$ and a meta variable's value $\omega_i(t)$ both lie in X. Let A be a Lebesgue measurable [33] subset of X.[8] A *signed characteristic function* on A is $\chi_A : X \to \{-1, 1\}$. If x is in A, then $\chi_A(x) = 1$; if x is not in A, then $\chi_A(x) = -1$. X is the domain of χ_A, and $\{-1, 1\}$ is the range. Let θ_ω be a threshold. Composing χ_A and meta variable ω, a *detectable event* occurs when

$$\int_{t=0}^{t=s} \chi_A(\omega(t))\ dt \geq \theta_\omega \quad \text{for some time } s. \quad (1)$$

Define *meta execution time* τ_ω as the infimum[9] of all times s that satisfy inequality (1). A meta operator $\mathcal{M}_\omega$, bound to meta variable ω, executes at time τ_ω.[10]

6.4. Meta Operators

Meta operators enable us to build self-modifiable differential equations. A *system* of differential equations is a set of differential equations. For example, $\mathcal{S} = \{\frac{dx}{dt} = 0,\ \frac{dy}{dt} = 0,\ \frac{dz}{dt} = x + y - xy - z\}$ is a system.

Our formal language for meta operators works as follows. A *create operator* $\mathcal{C}$ creates an empty system, and assigns a name with syntax $\mathcal{C}$(`time`, `name`). $\mathcal{C}(0, \mathcal{S})$ creates an empty system $\mathcal{S} = \{\}$ at time 0.

An *initialize operator* $\mathcal{I}$ declares a variable with a name and its type; assigns an initial value; and places the variable in a system. $\mathcal{I}$'s syntax is $\mathcal{I}$(`time`, `variable_name`, `variable_type`, `initial_value`, `system_name`). For example, $\mathcal{I}(0, x,$ `standard`, $0, \mathcal{S})$ creates x at time 0; defines x as a standard variable; and assigns x the initial value of 0 (i.e., $x_0 = 0$). Argument $\mathcal{S}$ indicates that x is a variable in a system named $\mathcal{S}$.

An *adjoin operator* $\mathcal{A}$ adjoins a new differential equation to a system. $\mathcal{A}$'s syntax is $\mathcal{A}$(`time`, `equation`, `system_name`). $\mathcal{A}$ is executed at a time specified by the first argument. In simpler cases, the time is explicitly stated. In other cases, the time is the greatest lower bound of all times s which satisfy integral inequality (1) in section 6.3. For example, execute 7 meta operators shown below: $\mathcal{C}(-1, \mathcal{S})$ $\mathcal{I}(0, x,$ `standard`, $0, \mathcal{S})$ $\mathcal{A}(0, \frac{dx}{dt} = 0, \mathcal{S})$ $\mathcal{I}(0, y,$ `standard`, $0, \mathcal{S})$ $\mathcal{A}(0, \frac{dy}{dt} = 0, \mathcal{S})$ $\mathcal{I}(0, z,$ `standard`, $0, \mathcal{S})$ $\mathcal{A}(0, \frac{dz}{dt} = x + y - xy - z, \mathcal{S})$. Afterward, system $\mathcal{S} = \{\frac{dx}{dt} = 0,\ \frac{dy}{dt} = 0,\ \frac{dz}{dt} = x + y - xy - z\}$.

A *replace operator* $\mathcal{R}$ replaces a variable with an equation or variable, or $\mathcal{R}$ replaces an equation with another equation. $\mathcal{R}$'s syntax is $\mathcal{R}$(`time`, `old_exp`, `new_exp`, `grammar`, `system_name`). The argument `time` behaves the same as `time` in the adjoin operator. Sometimes the 2nd argument `old_exp` represents the current variable that will be replaced by a new variable or equation, indicated by the 3rd argument `new_exp`. Sometimes `old_exp` represents an equation that will be replaced by a new equation `new_exp`. The 4th argument `grammar` is a pattern matching scheme. For replacement to occur, an expression in `old_exp`, must be accepted by a grammar, specified in `grammar`. It may be a semi-Thue grammar [34].[11] If `grammar` is $\emptyset$ or omitted, a replacement occurs at time specified by `time`. Overall, $\mathcal{R}$ plays a similar role to a meta command in an AEM.

6.5. Repairing a Damaged Equation

Adding a small amount of noise to the `OR` ODE so that $\frac{dx}{dt} = 0.1$ and $\frac{dy}{dt} = 0.1$, a Julia simulation is shown in table 5 with initial values $x_0 = y_0 = 0$. Table 5 shows that z moves to an incorrect output value because x and y move away from their initial values $x_0 = y_0 = 0$.

Table 5. $\frac{dx}{dt} = 0.1$. $\frac{dy}{dt} = 0.1$. $\frac{dz}{dt} = x + y - xy - z$.

t	0	.86	1.72	3.11	5.43	8.12	10.0
$x(t)$	0	.086	.17	.31	.54	.81	1.0
$y(t)$	0	.086	.17	.31	.54	.81	1.0
$z(t)$	0	.055	.17	.38	.68	.91	.98

Define equation $\frac{d\omega}{dt} = \frac{dx}{dt}$ with meta variable ω. Set $A = \mathbb{R}$. Set initial value $\omega(0) = 0$. Then $\int_0^s \chi_A(\omega(t))\ dt = \int_0^s \omega(t)\ dt$. Also, $\omega(s) = \omega(0) + \int_0^s \frac{d\omega}{dt}\ dt = \int_0^s \frac{dx}{dt}\ dt$. Now $\int_0^s \chi_A(\omega(t))\ dt = \int_0^s \int_0^s \frac{dx}{dt}\ dt$.

[7] Variables have type *standard* or type *meta*.

[8] Set A is measurable so that an integral is well-defined.

[9] The infimum of a set of real numbers is the greatest lower bound.

[10] Integrating $\omega(t)$'s orbit and executing $\mathcal{M}_\omega$ is analogous to a meta command executing when an active element A fires, due to the sum of A's input pulses exceeding A's threshold.

[11] Also, see pages 220-223 in [24].

Set $\theta_\omega = 10^{-6}$. Define $\mathcal{R}_\omega(\inf_{s>0} \int_0^s \omega(t)dt \geq \theta_\omega, \frac{dx}{dt} = 0, \frac{dx}{dt} = x(1-x), \emptyset, \mathcal{S})$. If there is physical noise $\frac{dx}{dt} = \delta > 0$, noise accumulates so that $\int_0^s \omega(t)\ dt$ first reaches θ_ω at time s. At time s, $\mathcal{R}_\omega$ self-modifies to $\mathcal{S} = \{\frac{dx}{dt} = x(1-x), \frac{dy}{dt} = 0, \frac{dz}{dt} = x + y - xy - z\}$. This change repairs noise in equation $\frac{dx}{dt}$ near $x = 0$.

$\mathcal{R}_\omega$ will not repair noise near $x = 1$. One can repair noise around both $x_0 = 0$ and $x_0 = 1$, by splitting A, for some $\epsilon > 0$, into $A_0 = (-\epsilon, \epsilon)$ and $A_1 = (1-\epsilon, 1+\epsilon)$, and creating two replacement operators $\mathcal{R}_{\omega_0}$ and $\mathcal{R}_{\omega_1}$.

7. Summary & Research Questions

By studying the AEM model, simulating the AEM with some simple programs, and extrapolating the use of meta commands to differential equations, we defined a principle of self-modifiability in computation and in differential equations. We also demonstrated self-modifiable dynamical systems that heal.

Long-term research should address these questions:

- How does a system self-reflect so that it detects a deleterious change? (For example, put a microphone in the doorbell system.[12])
- Can we build self-reflection into a mathematical theory so that it becomes a general principle on how to heal a malfunctioning system?
- Assuming that information theory [35, 36] will play a fundamental role in self-reflection, when is a broken system beyond self-repair? That is, when, if ever, has too much information been lost so that healing is impossible?
- How can self-repair be designed to adequately function while actively being attacked by a sentient adversary?

Acknowledgments

I would like to thank A.F. Mayer, and the anonymous peer reviewers for their helpful comments.

References

[1] Abhijit Mohanta, Anoop Saldanha. Malware Analysis and Detection Engineering. 1st Edition, Apress, 2020.

[2] Donald Knuth. The Art of Computer Programming. Fundamental Algorithms. Volume 1, 3rd Edition. Addison-Wesley, 1997.

[3] H. Abelson, G. Sussman. Structure and Interpretation of Computer Programs. MIT Press, 491–610, 1996.

[4] John Hennessy, David Patterson. Computer Architecture. 5th Edition, Morgan Kaufmann, 2012.

[5] John Maxwell. "On Governors." Proceedings of the Royal Society, 100, 1868.

[6] Arash YoosefDoost, William Lubitz. "Archimedean Screw Design." Energies 14, 7812. MDPI, 2021.

[7] Michael Tabor. Chaos and Integrability in Nonlinear Dynamics. John Wiley, 1989.

[8] Misha Gromov. Metric Structures for Riemannian and Non-Riemannian Spaces. Birkhäuser, 1999.

[9] James R. Munkres. Topology. Prentice-Hall, 1975.

[10] Circuits Today. Circuit IC UM 66. Nov. 9, 2018.

[11] M.S. Fiske. "Toward a Mathematical Understanding of the Malware Problem." Proc. of the 53rd Hawaii Intl. Conf. on System Sciences. Jan. 7-10, 2020.

[12] Daniel W. McShea. "Complexity and evolution: What everybody knows." Biology and Philosophy. Springer, 6, 303–324, July 1991.

[13] J. Lukeš, J. Archibald, et. al. "How a Neutral Evolutionary Ratchet Can Build Cellular Complexity." IUBMB Life. 63(7), 528–537, July 2011.

[14] E.C. Friedberg, et. al. DNA Repair and Mutagenesis. ASM Press, 2nd Edition, 2006.

[15] D.E. Rasmussen, R.B. Painter. "Evidence for repair of ultra-violet damaged DNA in cultured mammalian cells." Nature. 203, 1360–1362, 1964.

[16] George Kampis. Self-Modifying Systems in Biology and Cognitive Science. Pergamon Press, 1991.

[17] Y. Ke, et. al. "Repairing Programs with Semantic Code Search." 2015 30th IEEE / ACM Intl. Conf. on Automated Software Engineering. 295–305, 2015.

[18] Eugene Eberbach. "Selected Aspects of the Calculus of Self-Modifiable Algorithms Theory." ICCI 1990. LNCS 468, Springer, 1991.

[19] Michael S. Fiske. "Turing Incomputable Computation." Turing-100. EasyChair 10, 69–91, 2012.

[20] Wilfrid Rall. The Theoretical Foundation of Dendritic Function. Papers of Wilfrid Rall. MIT Press, 1995.

[21] Greg J. Stuart, Nelson Spruston. "Dendritic Integration: 60 years of progress." Nature Neuroscience. 18(12), 1713–1721, Dec. 2015.

[22] Michael S. Fiske. "The Active Element Machine." Proceedings of Computational Intelligence. Springer 391, 69–96, 2011.

[23] A.V. Aho, R. Sethi, J.D. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, 1988.

[24] J. Hopcroft, J.D. Ullman. Intro. to Automata Theory, Languages, and Computation. Addison-Wesley, 1979.

[25] J. Backus. "Can programming be liberated from the von neumann style?: a functional style and its algebra of programs." Comm. of ACM, 21(8), 613–641, 1978.

[26] Abraham Robinson. Non-standard Analysis. Princeton University Press, 1996.

[27] R. L. Graham, V. Rödl. Numbers in Ramsey Theory. Surveys in Combinatorics, LMS Lecture Note Series 123. Cambridge University Press, 1987.

[12] A.F. Mayer suggested putting a microphone in the doorbell circuit as an example of detecting a malfunctioning system.

[28] Vera Rosta. "Ramsey Theory Applications." The Electronic Journal of Combinatorics, Dec. 2004.

[29] S. Burr. "Determining Generalized Ramsey Numbers is NP-hard." Ars Combinatoria. 17, 21–25, 1984.

[30] Brian Kernighan, Dennis Ritchie. The C Programming Language. 2nd Edition, Prentice-Hall, 1988.

[31] NIST. Secure Hash Standard FIPS-180-4. 2012.

[32] Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral Shah. "Julia: A fresh approach to numerical computing." SIAM review. 59(1), 65–98, 2017.

[33] Terence Tao. An Introduction to Measure Theory. American Mathematical Society. Volume 126, 2011.

[34] Axel Thue. "Probleme über Veränderungen von Zeichenreihen nach gegebenen Regeln." Christiana Videnskabs-Selskabs Skrifter, I. 10, 1914.

[35] Claude E. Shannon. "A Mathematical Theory of Communication." Bell Systems Technical Journal. 27, 379–423, 1948.

[36] Thomas M. Cover, Joy A. Thomas. Elements of Information Theory. 2nd Ed., Wiley & Sons, 2006.

8. Appendix

8.1. Active Element Machine Architecture

Let $\mathbb{Z}$ be the integers, and fix infinitesimal $\mathtt{dT}$ [26]. Define extended integers $\overline{\mathbb{Z}} = \{m + k\ \mathtt{dT}\colon m, k \in \mathbb{Z}\}$. In 8.2, keyword $\mathtt{dT}$ is described, which establishes an order on $\overline{\mathbb{Z}}$: for example, $3 - 5\mathtt{dT} < 3 - 4\mathtt{dT}$.

Definition 1. *Machine Architecture*

Γ, Ω, and Δ are index sets that index the input, computational, and output active elements, respectively. 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. $\Psi_i(t) = \sup\{s : s < t$ 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 and is always defined here, whence Ψ_i is well-defined.
- 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, $W_{ki}(t) = \{s :$ active element E_k fired at time s 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, pulse amplitudes and thresholds are integer valued. At any moment, transmission times and pulse widths are extended integers ≥ 1. These parameters are a function of time: $\theta_i(t), r_i(t), A_{ki}(t), \omega_{ki}(t), \tau_{ki}(t)$. Time t is an extended integer.

Input active elements that are not computational have the same properties as computational elements, except they receive no inputs ϕ_{ki} from elements in this machine. Input elements are externally firable, by an external source from the environment or an output element from a distinct machine $\mathcal{M}(\mathcal{I}', \mathcal{E}', \mathcal{O}')$. An input element can fire at any time after its refractory period has expired. An element can be an input and computational element; an element can be an output and computational element. When an output element E_i is not a computational element ($i \in \Delta - \Omega$), then E_i does not send pulses to elements in this machine.

If $g_i(s) = 1$, then active element E_i *fired* at time s. *Refractory period* r_i is the amount of time that must elapse after E_i just fired before E_i can fire again. τ_{ki} is the transmission time: If element E_k fires at time t, a pulse sent from E_k reaches E_i at time $t + \tau_{ki}$. *Pulse amplitude* A_{ki} is the height of the pulse that E_k sends to E_i after E_k has fired. After this pulse reaches E_i, *pulse width* ω_{ki} indicates how long the pulse lasts as input to E_i. If $A_{ki} = 0$, no connection exists from E_k to E_i.

8.2. Active Element Machine Language

A minimal programming language defines four commands: `element`, `connection`, `fire`, and `meta`.

Syntax 1. *AEM Program*

An AEM program is defined with Backus-Naur syntax. ϵ indicates a blank string that terminates an expression.

```
<AEM_program> ::= <aem_cmds>

<aem_cmds> ::= ϵ | <cmd><aem_cmds>

<cmd> ::= <e_cmd> | <c_cmd> | <f_cmd> | <m_cmd>
```

Syntax 2. *AEM Symbols & Extended Integers*

```
<ename> ::= <int> | <symbol>

<symbol> ::= <char><str> | (<ename> . . . <ename>)

<str> ::= ϵ | <char><str> | 0<str> | <pint><str>

<char> ::= <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> ::= _
```

The rules below cover extended integer arithmetic.

```
<int>  ::= <pint> | <nint> | 0
<nint> ::= - <pint>
<pint> ::= <nonzero><digits>
<digits> ::= <numeral> | <numeral><digits>
<nonzero> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<numeral> ::= ϵ | <nonzero> | 0
<aint> ::= <aint><sgn><d> | <d><sgn><aint> | <d>
<sgn> ::= + | -
<d> ::= <int> | <char><str> | <infinitesimal>
<infinitesimal> ::= dT
```

Command 1. `element`

An `element` command specifies the time when an active element is created or its parameter values are updated. Its Backus-Naur syntax is shown below.

```
<e_cmd> ::= (element (time <aint>)
              (name <ename>) <eth> <ere> <ela>)
eth ::= ϵ | (threshold <int>)
ere ::= ϵ | (refractory <pint>)
ela ::= ϵ | (last <int>)
```

Keyword `time` tags a time value s (extended integer) when the element is created or updated. If the name value is `E`, keyword `name` tags name `E` of the active element. Keyword `threshold` tags a threshold $\theta_E(s)$. Keyword `refractory` tags a refractory value $r_E(s)$, and `last` tags a last time fired value $\Psi_E(s)$.

Command 2. `connection`

A `connection` command creates or updates a connection from one active element to another active element. Its Backus-Naur syntax is shown below.

```
<c_cmd> ::= (connection (time <aint>)
                  (from <ename>) (to <ename>)
                  [(amp <int>) (width <pint>)
                  (delay <pint>)])
```

Keyword `time` tags a time value s when the connection is created or updated. Keyword `from` tags a name `E` of the active element that sends a pulse with these updated values. Keyword `to` tags a name `B` of the active element that receives a pulse with these updated values. Keyword `amp` tags a pulse amplitude value $A_{E,B}(s)$ that is assigned to this connection. Keyword `width` tags a pulse width value $\omega_{E,B}(s)$. Keyword `delay` tags a transmission time $\tau_{E,B}(s)$.

When the AEM clock reaches time `s`, symbols `E` and `B` are name values that must refer to an element that already has been created or updated before or at time s. Not all of the connection parameters have to be 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 0: this "zero" connection has no effect on the AEM computation. The connection exists indefinitely with the same parameter values until a new connection command is executed between `from` element `E` and `to` element `B`.

Command 3. `fire`

A `fire` command has the following syntax.

```
<f_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 can be used to fire input active elements.

Keyword 1. `dT`

Keyword `dT` represents a positive infinitesimal amount of time. If `m` and `n` are integers and $0 \leq$ `m` $<$ `n`, then `mdT` $<$ `ndT`. Also, `dT` > 0 and `dT` is less than every positive rational number; and $-$`dT` < 0 and $-$`dT` is greater than every negative rational number.

`dT` coordinates almost simultaneous events that are non-commutative or indeterminate: e.g. element `A` may be receiving a pulse from element `B` at the same time that a connection between them is removed.

Keyword 2. `clock`

Keyword `clock` evaluates to an integer, and is the current AEM time. `clock` is an instance of `<ename>`.

If the current AEM time is 5, then `(element (time clock) (name clock) (threshold 1) (refractory 1) (last -1))` executes as `(element (time 5) (name 5) (threshold 1) (refractory 1) (last -1))`.

After `(element (time clock) (name clock) (threshold 1) (refractory 1) (last -1) )` is created, then this command is executed with the current clock time. If this command is in the original AEM program before the clock starts at 0, then the following sequence of elements named 0, 1, 2, ... are created.

```
(element (time 0) (name 0) (threshold 1) (refractory 1) (last -1))
(element (time 1) (name 1) (threshold 1) (refractory 1) (last -1))
(element (time 2) (name 2) (threshold 1) (refractory 1) (last -1))
. . .
```

Command 4. `meta`

A `meta` command executes a command `<cmd>` when an active element fires within a window of time.

```
<m_cmd> ::= (meta (name <ename>) [<wtime>] <cmd>)
<wtime> ::= (window <aint> <aint>)
```

For example, `E` is the *name* of the active element in `(meta (name E) (window l w) (C (args t a))`. Keyword `window` tags a time interval, called a *window of time*. `l` and `w` are extended integers that locate the boundary points. If `w` > 0, the window of time is `[l, l+w]`. If `w` < 0, the window is `[l+w, l]`.

Command `C` executes each time that `E` fires during the window of time. If the window is omitted, then command `C` executes at any time that element `E` fires.