Mila

Giaco

Saumya

Kevin

Gregg

Modeling metamorphism by abstract interpretation

Roberto Giacobazzi

16.09.2010 - SAS2010

The problem

Malware analysis: signature checking

- Malware refers to malicious software
- Signature checking: identify a sequence of instructions which is unique to a malware (virus signature) then scan program for signatures
- Example: *Chernobyl* signature:

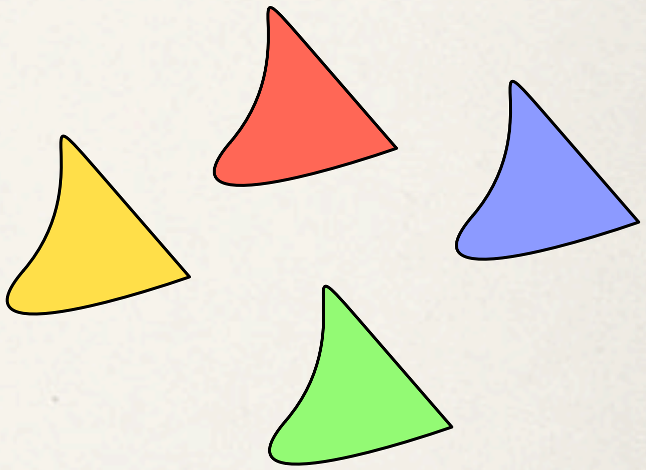
```
E800 0000 005B 8D4B 4251 5050

0F01 4C24 FE5B 83C3 1CFA 882B
```

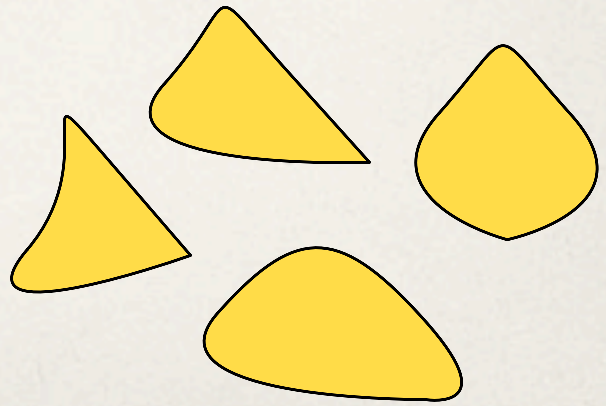
- Cumbersome, inaccurate, easy to foil....

Anti-anti malware

- How can we escape signature checking?
    - *...by dynamically modifying malware structure!*
- Polymorphic malware contain decryption routines which decrypt encrypted constant parts of their body.



- Metamorphic malware typically do not use encryption, but mutates (obfuscate) forms in subsequent generations.



Metamorphism as obfuscation

From Chernobyl CIH 1.4

```
Loop:
        pop   ecx
        jecxz SFModMark
        mov   esi, ecx
        mov   eax, 0d601h
        pop   edx
        pop   ecx
        call  edi
        jmp   Loop
```

```
Loop:
        pop   ecx
        nop
        jecxz SFModMark
        xor   ebx, ebx
        beqz  N1
N1:     mov   esi, ecx
        nop
        mov   eax, 0d601h
        pop   edx
        pop   ecx
        nop
        call  edi
        xor   ebx, ebx
        beqz  N2
N2:     jmp   Loop
```

Metamorphism as obfuscation

From Chernobyl CIH 1.4

```
Loop:
        pop   ecx
        jecxz SFModMark
        mov   esi, ecx
        mov   eax, 0d601h
        pop   edx
        pop   ecx
        call  edi
        jmp   Loop
```

```
Loop:
        pop   ecx
        nop

        call  edi
        xor   ebx, ebx
        beqz  N2
N2:     jmp   Loop

        nop
        mov   eax, 0d601h
        pop   edx
        pop   ecx
        nop

        jecxz SFModMark
        xor   ebx, ebx
        beqz  N1
N1:     mov   esi, ecx
```

Metamorphism as obfuscation

From Chernobyl CIH 1.4

```
Loop:
        pop   ecx
        jecxz SFModMark
        mov   esi, ecx
        mov   eax, 0d601h
        pop   edx
        pop   ecx
        call  edi
        jmp   Loop
```

```
Loop:
        pop   ecx
        nop
        jmp L1
L3:     call edi
        xor   ebx, ebx
        beqz N2
N2:     jmp   Loop
        jmp L4
L2:     nop
        mov   eax, 0d601h
        pop   edx
        pop   ecx
        nop
        jmp L3
L1:     jecxz SFModMark
        xor   ebx, ebx
        beqz N1
N1:     mov   esi, ecx
        jmp L2
L4:
```

Metamorphism: an example

Malware evolution

```
mov [ebp - 3], eax
```

```
push ecx
mov ecx,ebp
add ecx,33
mov [ecx-36],eax
pop ecx
```

```
push ecx
mov ecx,ebp
add ecx,33
push esi
mov esi,ecx
sub esi,34
mov [esi-2],eax
pop esi
pop ecx
```

```
push ecx
mov ecx, ebp
push eax
mov eax, 33
add ecx, eax
pop eax

push esi
mov esi, ecx
push edx

mov edx, 34
sub esi, edx
pop edx
mov [esi - 2], eax
pop esi
pop ecx
```

```
push ecx
mov ecx, [ebp + 10]
mov ecx, ebp
push eax
add eax, 2342
mov eax, 33
add ecx, eax
pop eax
mov eax, esi
push eax
mov esi, ecx
push edx
xor edx, 778f
mov edx, 34
sub esi, edx
pop edx
mov [esi-2], eax
pop esi
pop ecx
```

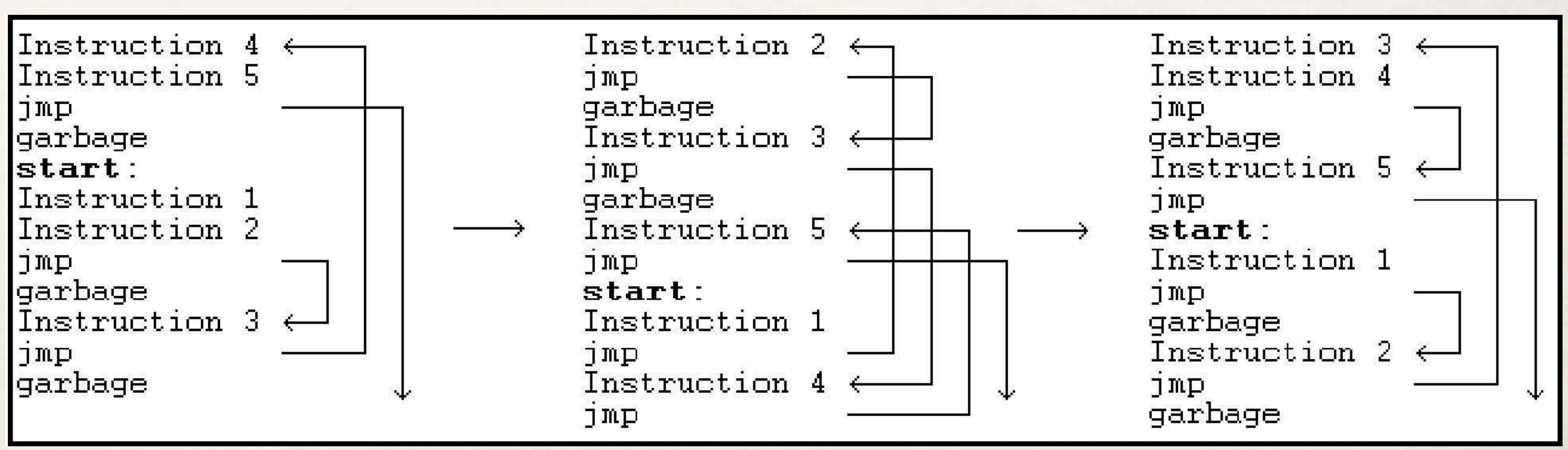
- How can we model and compute signatures for metamorphism?

Metamorphism: some (public) history

http://vx.netlux.org/

- Win32.Evol
    - swaps instructions with equivalents
    - inserts junk code between essential instructions
- Regswap (Win32)
    - same code different register names
- BadBoy (DOS) and Ghost (Win32)
    - same code different subroutine order (n! possible mutations: 10 modules ~3.6M possible signatures)
- Zmorph (Win95)
    - decrypt virus body instruction by instruction
    - push instructions on stack
    - insert and remove jumps
    - rebuild body on stack
- Zperm (Win95)
    - ....

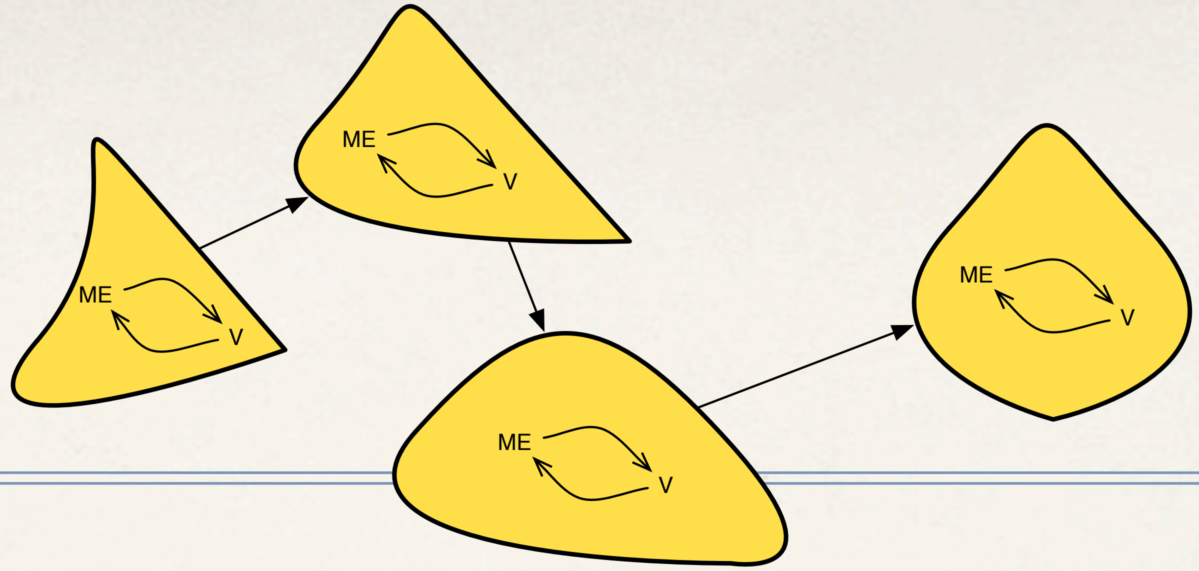
by Peter Szor



Attacking metamorphism

- Idea: *Behavior Monitors*
- Run suspect program in an emulator and extract a DB of relevant signatures (huge DB)
- Look for changes in file structure: Some viruses modify files in a consistent way (inaccurate)
- Disassemble and look for virus-like instructions: reverse engineering malware (expensive)

Problems

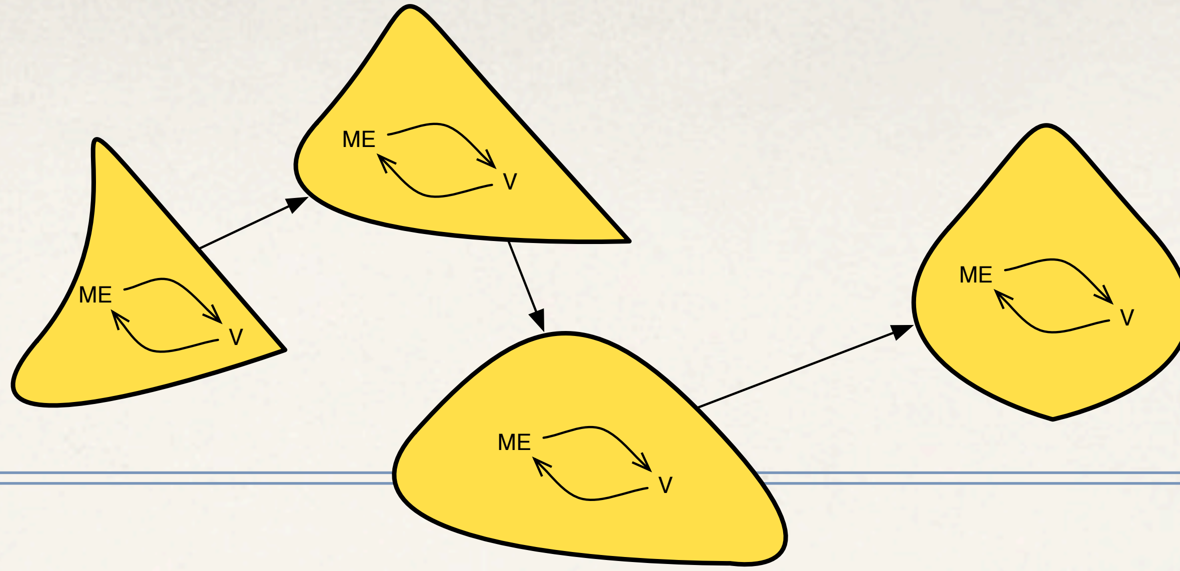


- The code may contain its own metamorphic engine ME
- The metamorphic engine can be used when engineering malware
- Metamorphic signature: is a language $\mathcal{L}$ of possible signatures generated by a metamorphic malware:

$$\sigma \in \mathcal{L} \quad \Rightarrow \quad \sigma \text{ is a possible signature}$$

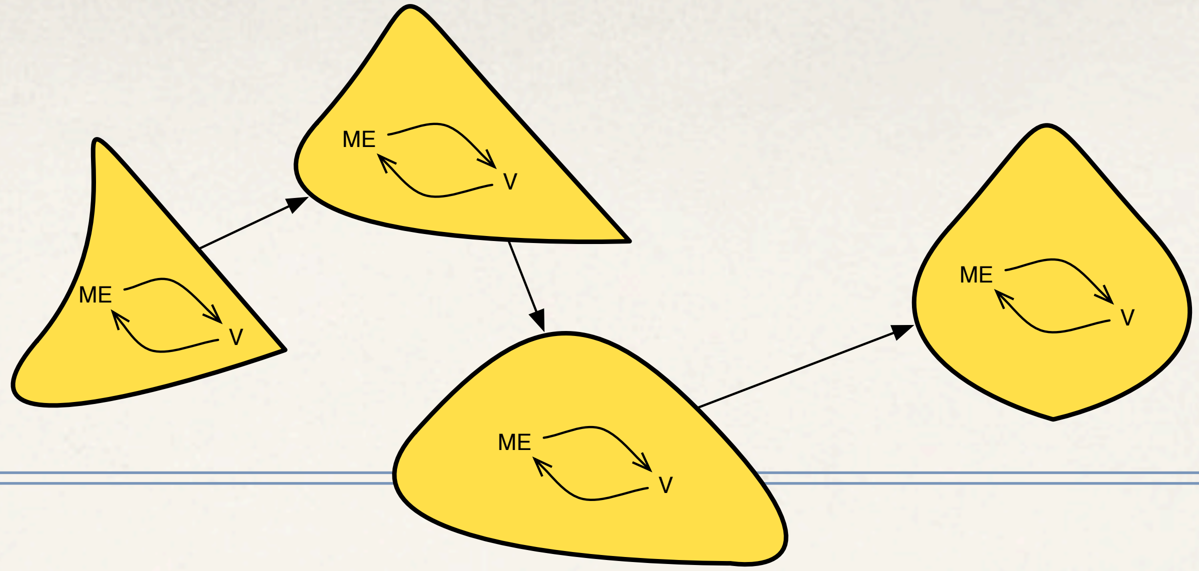
- Is there a way for extracting a metamorphic signatures?

Related works



- Specify some abstraction (CFG, instruction equivalence, rewrite rules towards normal form - *undo metamorphism*)
- [Dalla Preda et al POPL07, Filiol PWASET07, Zbitsky JCV 09, Bonfante et al JCV 09]
  - Existing semantics-based approach to malware detection are promising but they still rely on a priori knowledge of the metamorphic transformations used by malware writers
- Need to model the self-modifying behavior of a metamorphic malware without any a priori knowledge of the transformations it uses

Idea



- Idea: Extract $\mathcal{L}$ as a abstract interpretation of the metamorphic malware!

  Extracting metamorphic signatures is approximating malware semantics

  - data objects are code slices
  - abstraction acts on code structure *(code may be as complex as data!!)*
  - invariants on mutational code structure describe the metamorphic engine behavior!!
  - fix-point abstraction approximate invariants, i.e. generates metamorphic signatures....

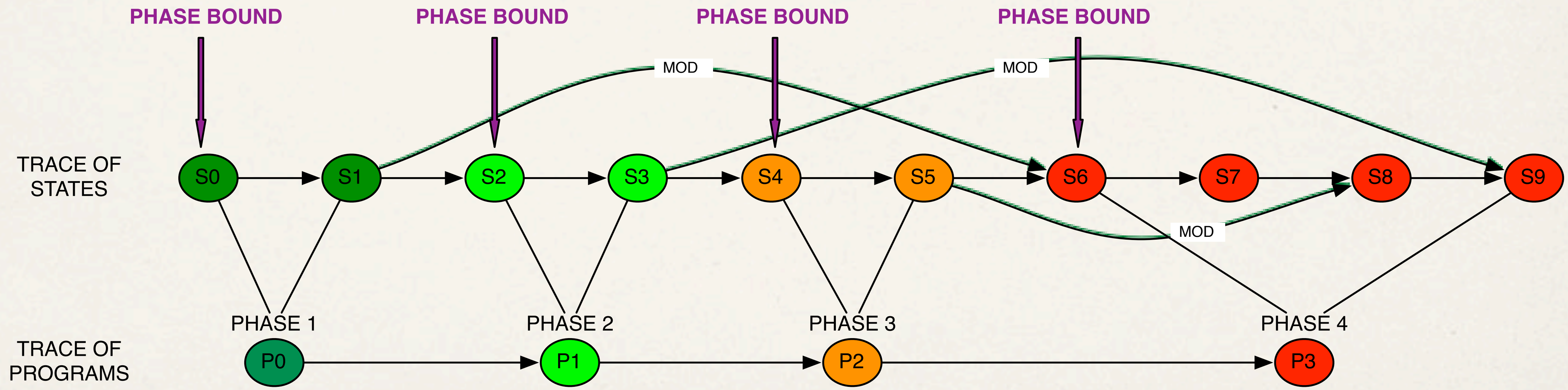
Modeling metamorphism

Phase semantics

- States: *no distinction between code and data* $\langle a, \mathtt{m}, \theta, \mathfrak{I} \rangle$ (entry point, memory, stack, input)

- Phase semantics: partition the trace of execution states into phases, each collecting the computation of a particular code variant

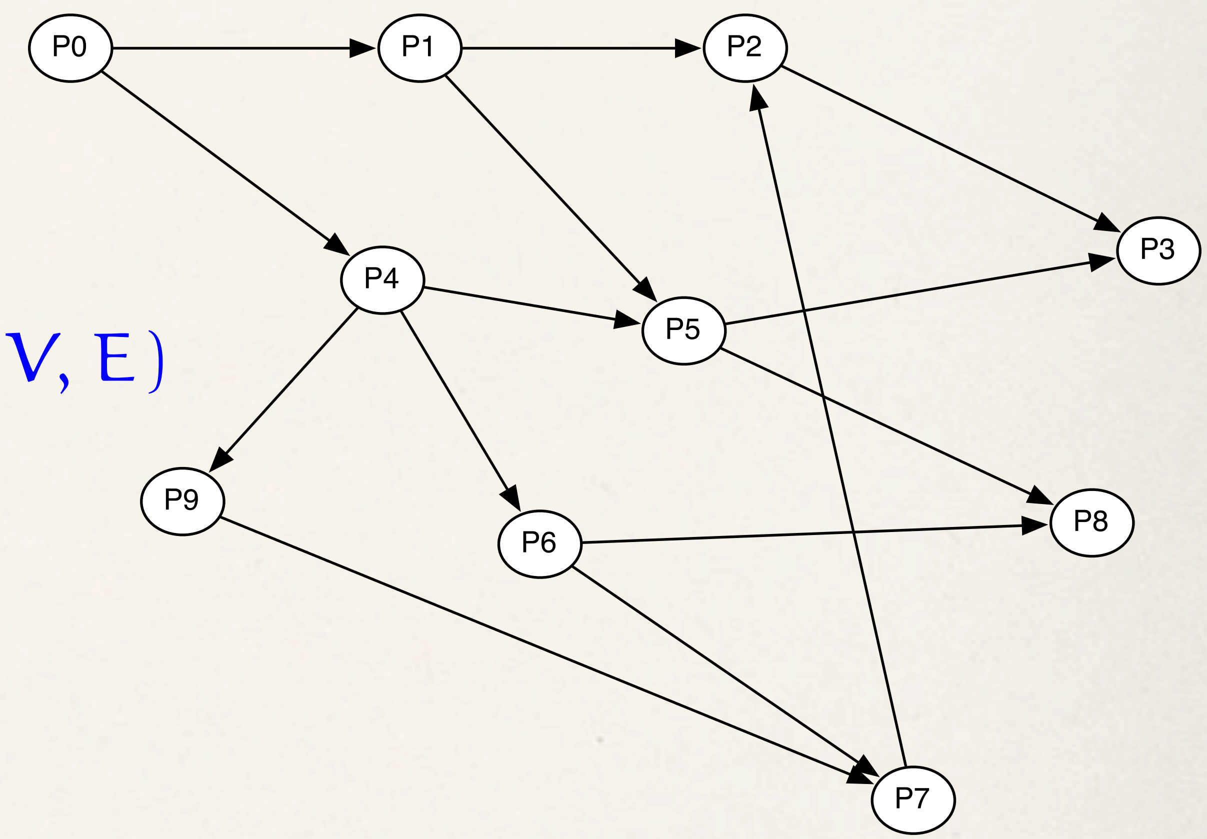
- Maximal trace semantics: $\mathbf{S}[\![\mathrm{P}]\!] = \mathit{lfp}\mathcal{F}_{\mathcal{T}}[\![\mathrm{P}]\!]$



$$\mathit{bound}(s) = \{s_0\} \cup \{s_i \mid \mathit{MOD}(s_{i-1}) \cap \{a_j \mid i \leq j \leq n\} \neq \emptyset\}$$

$$\mathit{phases}(s) = \{s_i \ldots s_j \mid s_i, s_{j+1} \in \mathit{bound}(s), \forall l \in [i+1, j] : s_l \notin \mathit{bound}(s)\}$$

Fix-point phase semantics

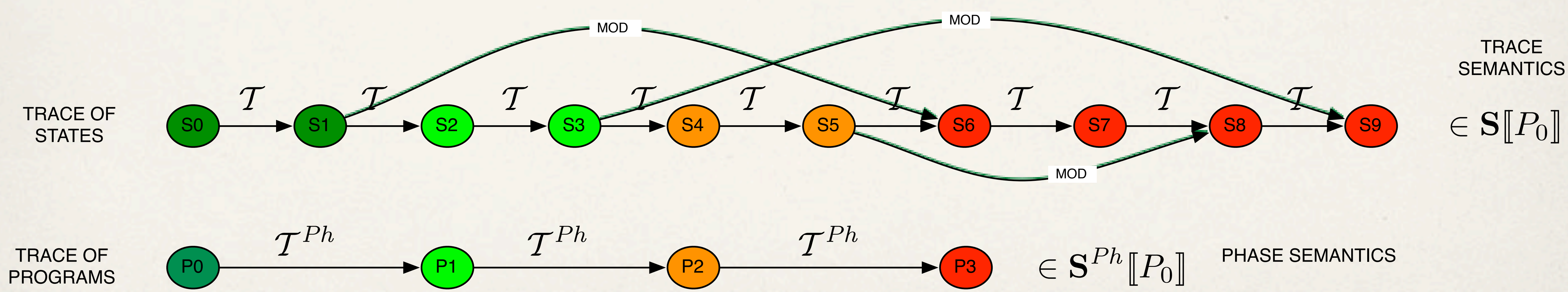


- Program evolution graph: $G[\![P_0]\!] = (V, E)$
  - Nodes = Phases
  - Edges = Phase transitions
- The phase semantics of a program P_0 is given by the set of all possible paths of its program evolution graph

$$\mathbf{S}^{Ph}[\![P_0]\!] = \{P_0 \ldots P_n \mid \forall i \in [0, n-1] : (P_i, P_{i+1}) \in E\}$$

Fix-point phase semantics

- Phase transition: $\mathcal{T}^{Ph}(\mathrm{P}_0) = \left\{ \mathrm{P}_i \;\middle|\; \begin{array}{l} s = s_0 \ldots s_i \ldots s_n \in \mathbf{S}[\![\mathrm{P}_0]\!], s_i \in \mathit{bound}(s), \\ \forall l \in [1, i-1] : s_l \notin \mathit{bound}(s) \end{array} \right\}$
  - Fix-point iteration: $\mathbf{S}^{Ph}[\![\mathrm{P}]\!] = \mathit{lfp}\mathcal{F}_{\mathcal{T}^{Ph}}[\![\mathrm{P}]\!]$



Correctness of phase semantics

$$\langle \wp(\Sigma^*), \subseteq \rangle \xleftarrow[\alpha_{Ph}]{\gamma_{Ph}} \langle \wp(\mathbf{P}^*), \subseteq \rangle$$

- Trace semantics and phase semantics are related by abstraction:
  - α_{Ph} keeps only phase bounds
- Locally incomplete.....
- Fix-point complete: $\alpha_{Ph}(\mathit{lfp}\mathcal{F}_{\mathcal{T}}[\![P_0]\!]) = \mathit{lfp}\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!]$

CONCRETE TEST FOR METAMORPHISM

$$P_0 \rightsquigarrow_{Ph} Q \quad \Leftrightarrow \quad \exists P_0, P_1, \ldots, P_n \in \mathbf{S}^{Ph}[\![P_0]\!], \exists i \in [0, n] : \; P_i = Q$$

no false positives, no false negatives

Abstracting metamorphism

Abstracting phases

- Need abstraction for approximating phases!!!

  design GC: $\langle \wp(\mathbf{P}^*), \subseteq \rangle \xleftrightarrow[\alpha_A]{\gamma_A} \langle A, \sqsubseteq_A \rangle$

  define the abstract transition relation $\mathcal{T}^A : A \to \wp(A)$

  define $\mathcal{F}_{\mathcal{T}^A}[\![P_0]\!] : A \to A$ whose fixpoint computation $\mathit{lfp}^{\sqsubseteq_A} \mathcal{F}_{\mathcal{T}^A}[\![P_0]\!] = \mathbf{S}^A[\![P_0]\!]$ corresponds to the abstract specification of the metamorphic behavior

  prove that $\mathbf{S}^A[\![P_0]\!]$ is a correct approximation of phase semantics $\mathbf{S}^{Ph}[\![P_0]\!]$, i.e.,
  $\alpha_A(\mathit{lfp}^{\subseteq} \mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!]) \sqsubseteq_A \mathit{lfp}^{\sqsubseteq_A} \mathcal{F}_{\mathcal{T}^A}[\![P_0]\!]$

ABSTRACT TEST FOR METAMORPHISM

$$P_0 \leadsto_A Q \quad \Leftrightarrow \quad \alpha_A(Q) \sqsubseteq_A \mathbf{S}^A[\![P_0]\!]$$

no false negatives

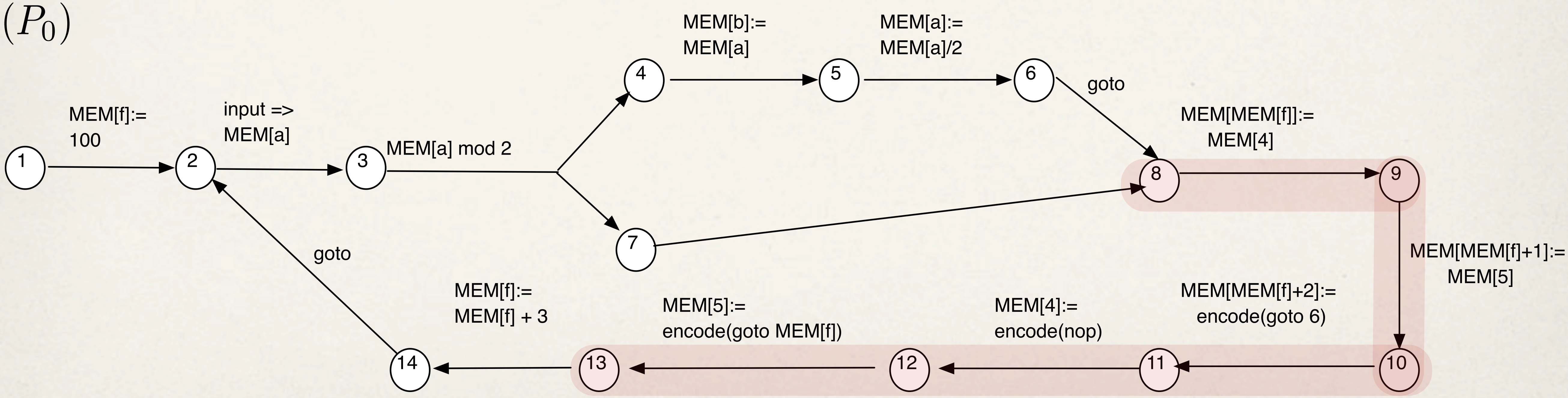
Phases as FSA

$\mathring{\alpha} : \mathbf{P} \longrightarrow \mathfrak{F}$

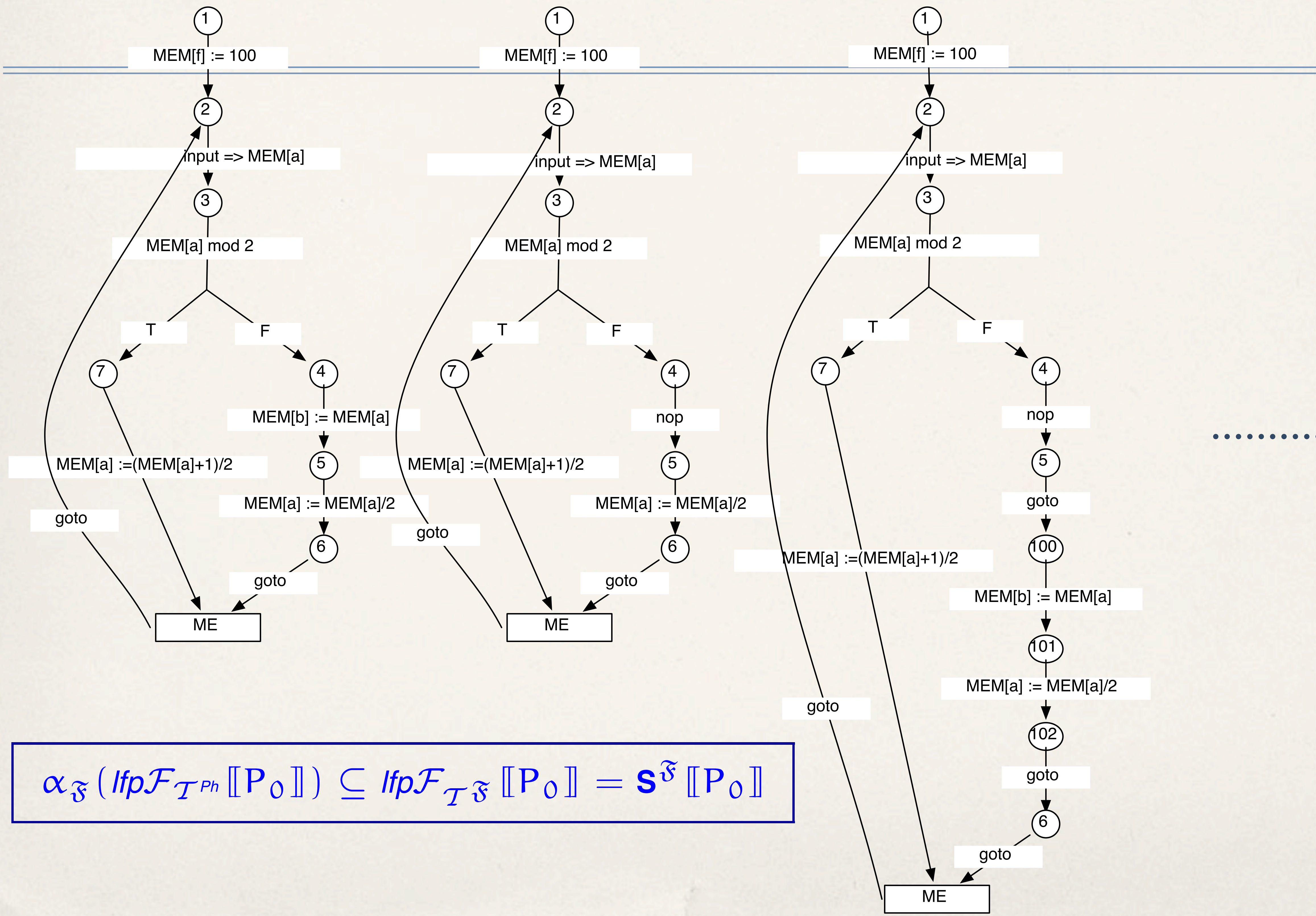
P_0

```
1:   MEM[f] := 100
2:   input ⇒ MEM[a]
3:   if (MEM[a] mod 2) goto 7
4:   MEM[b] := MEM[a]
5:   MEM[a] := MEM[a]/2
6:   goto 8
7:   MEM[a] := (MEM[a] + 1)/2
8:   MEM[MEM[f]] := MEM[4]
9:   MEM[MEM[f] + 1] := MEM[5]
10:  MEM[MEM[f] + 2] := encode(goto 6)
11:  MEM[4] := encode(nop)
12:  MEM[5] := encode(goto MEM[f])
13:  MEM[f] := MEM[f] + 3
14:  goto 2
```

$\mathring{\alpha}(P_0)$



Phase semantics as traces of FSA



$$\alpha_{\mathfrak{F}}\left(lfp\mathcal{F}_{\mathcal{T}^{Ph}}[\![\mathrm{P}_0]\!]\right) \subseteq lfp\mathcal{F}_{\mathcal{T}^{\mathfrak{F}}}[\![\mathrm{P}_0]\!] = \mathbf{S}^{\mathfrak{F}}[\![\mathrm{P}_0]\!]$$

Phase semantics as traces of FSA: $\mathbf{S}^{\sharp}[\![P_0]\!]$

- We need a static approximation of the Phase transfer function
  - Stack analysis: approximating the values on top of the stack
  - Memory analysis: approximating the values stored in memory
- We emulate the run of a phase generating a superset of FSA that may be generated (over approximation!)

$$\boxed{\mathbf{S}^{\mathfrak{F}}[\![P_0]\!] \subseteq \mathbf{S}^{\sharp}[\![P_0]\!]}$$

Widening phases: regular metamorphism

- Regular metamorphism: mutation constrained in a regular language of instructions

- Collapsing a (static) trace of FSA into a single FSA: widening

  - $\langle \mathfrak{F}/_{\equiv}, \sqsubseteq_{\mathfrak{F}} \rangle$ where $M_1 \sqsubseteq_{\mathfrak{F}} M_2 \Leftrightarrow L(M_1) \subseteq L(M_2)$

$$\mathbf{W}_0 = \mathring{\alpha}(P_0) \qquad \mathbf{W}_{i+1} = \mathbf{W}_i \nabla \mathcal{F}^{\Cup}_{\mathcal{T}^\sharp} [\![P_0]\!] (\mathbf{W}_i)$$

ABSTRACT TEST FOR METAMORPHISM on $\mathfrak{F}/_{\equiv}$

$$P_0 \rightsquigarrow_{\mathfrak{F}} Q \quad \Leftrightarrow \quad \mathring{\alpha}(Q) \sqsubseteq_{\mathfrak{F}} \mathbf{W}[\![P_0]\!]$$

no false negatives

Widening phases: regular metamorphism

- Let M_1 and M_2 be two FSA
- $R_n \subseteq Q_1 \times Q_2$ is a state relation

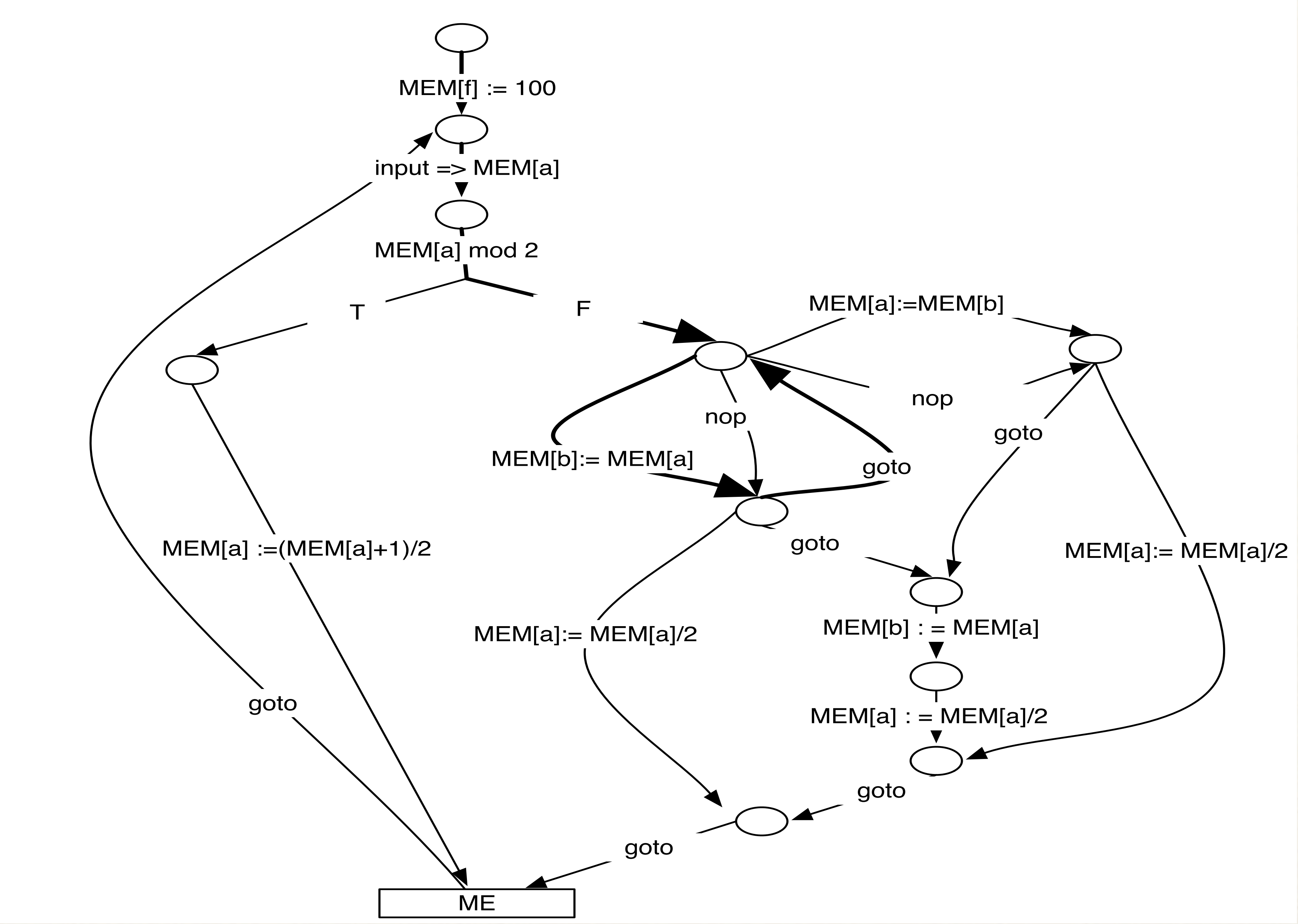
$(q_1, q_2) \in R_n$ if q_1 and q_2 recognize the same language of strings of length n

$$q \equiv_R q' \quad \text{iff} \quad \exists r \in Q_1 : (r, q) \in R_n \text{ and } (r, q') \in R_n$$

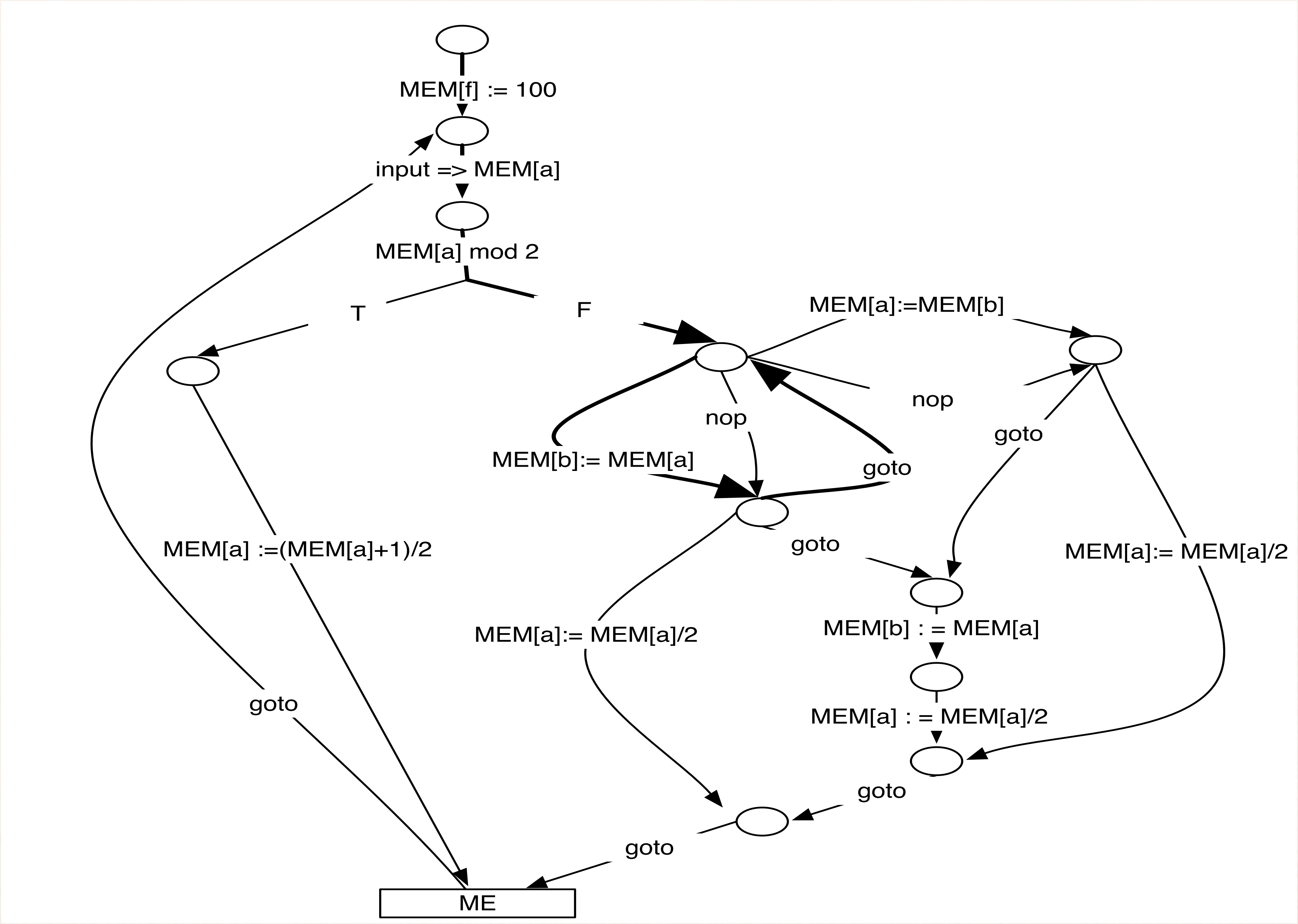
$$M_1 \triangledown M_2 = M_2 / \equiv_R$$

- It is a widening if on finite alphabet: approximate instruction terms!

Widening phases: regular metamorphism

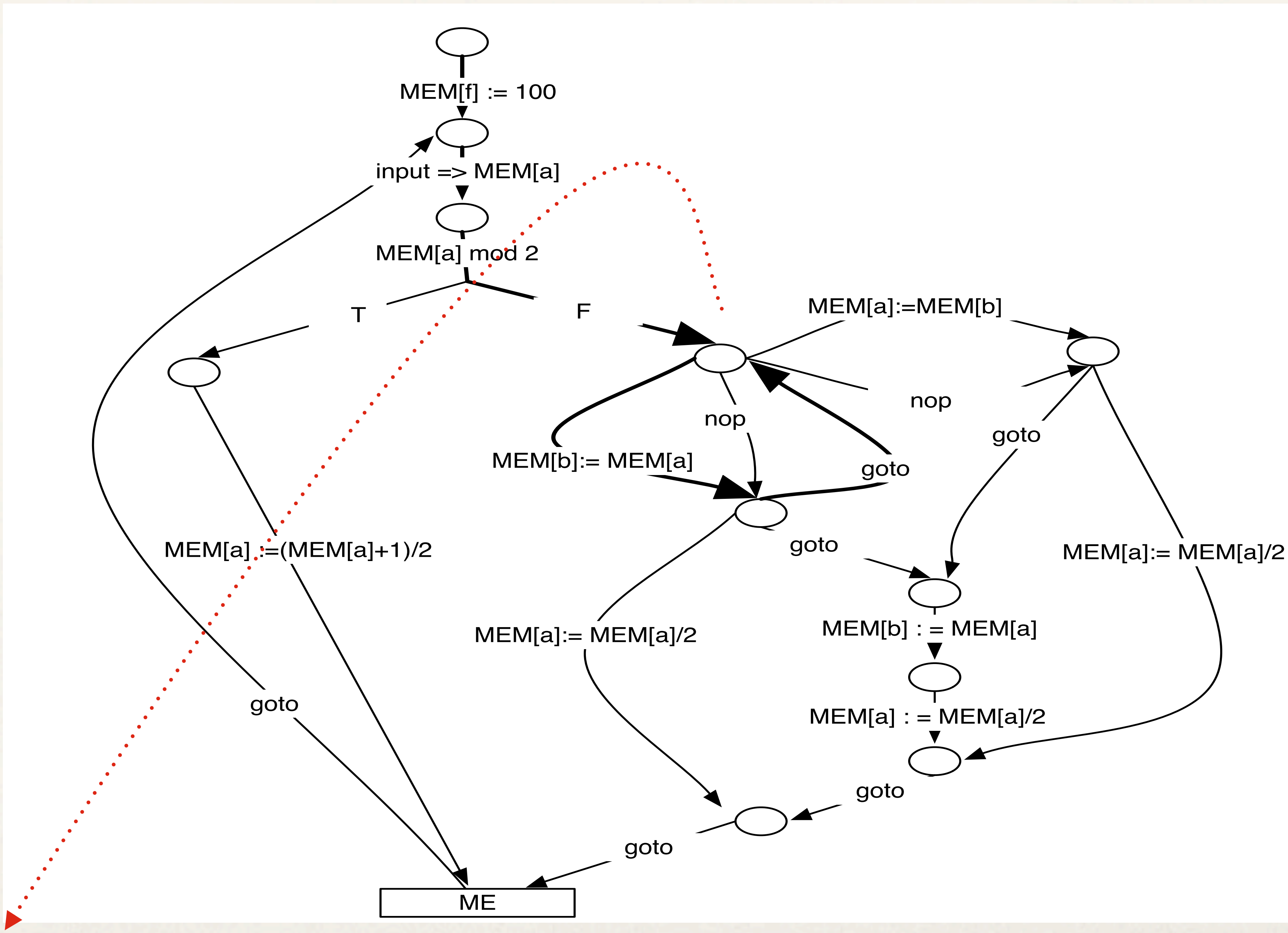


Widening phases: regular metamorphism



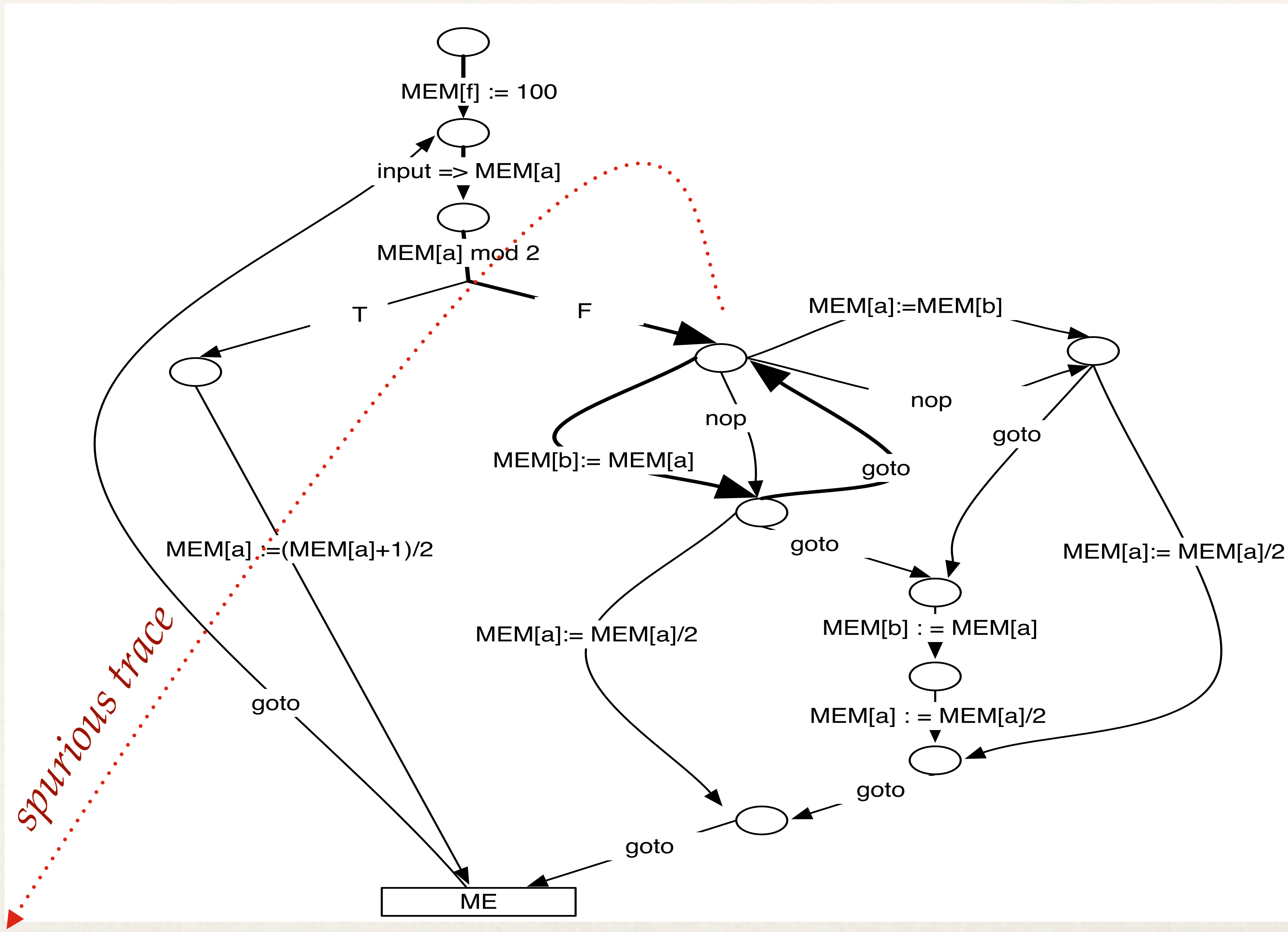
`MEM[f]:=100;input=>MEM[a];MEM[a] mod 2 = 0; MEM[b]:=MEM[a]; goto; MEM[b]:=MEM[a]; goto;...`

Widening phases: regular metamorphism



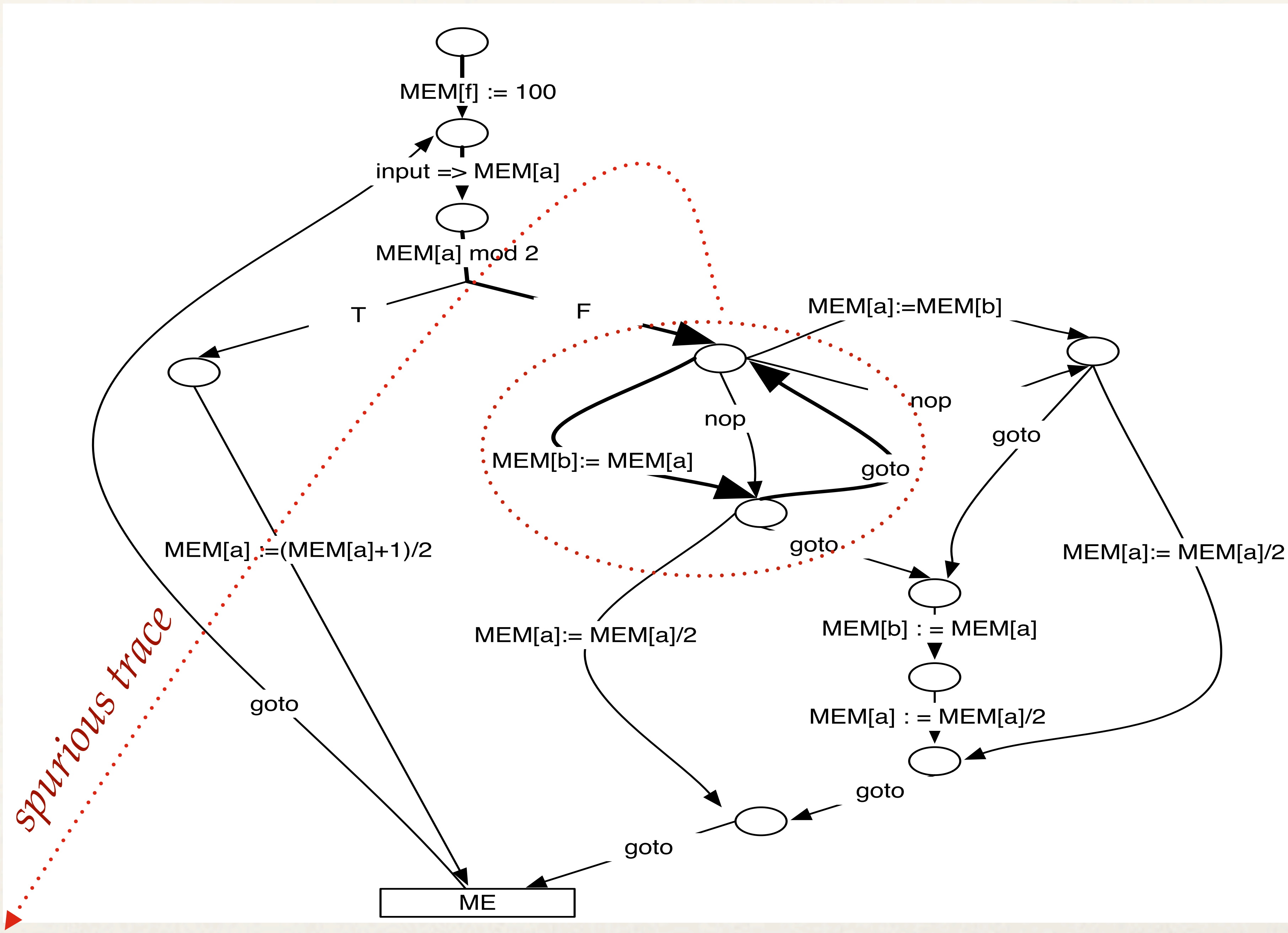

```
MEM[f]:=100;input=>MEM[a];MEM[a] mod 2 = 0; MEM[b]:=MEM[a]; goto; MEM[b]:=MEM[a]; goto;...
```

Widening phases: regular metamorphism



MEM[f]:=100;input=>MEM[a];MEM[a] mod 2 = 0; MEM[b]:=MEM[a]; goto; MEM[b]:=MEM[a]; goto;...

Widening phases: regular metamorphism

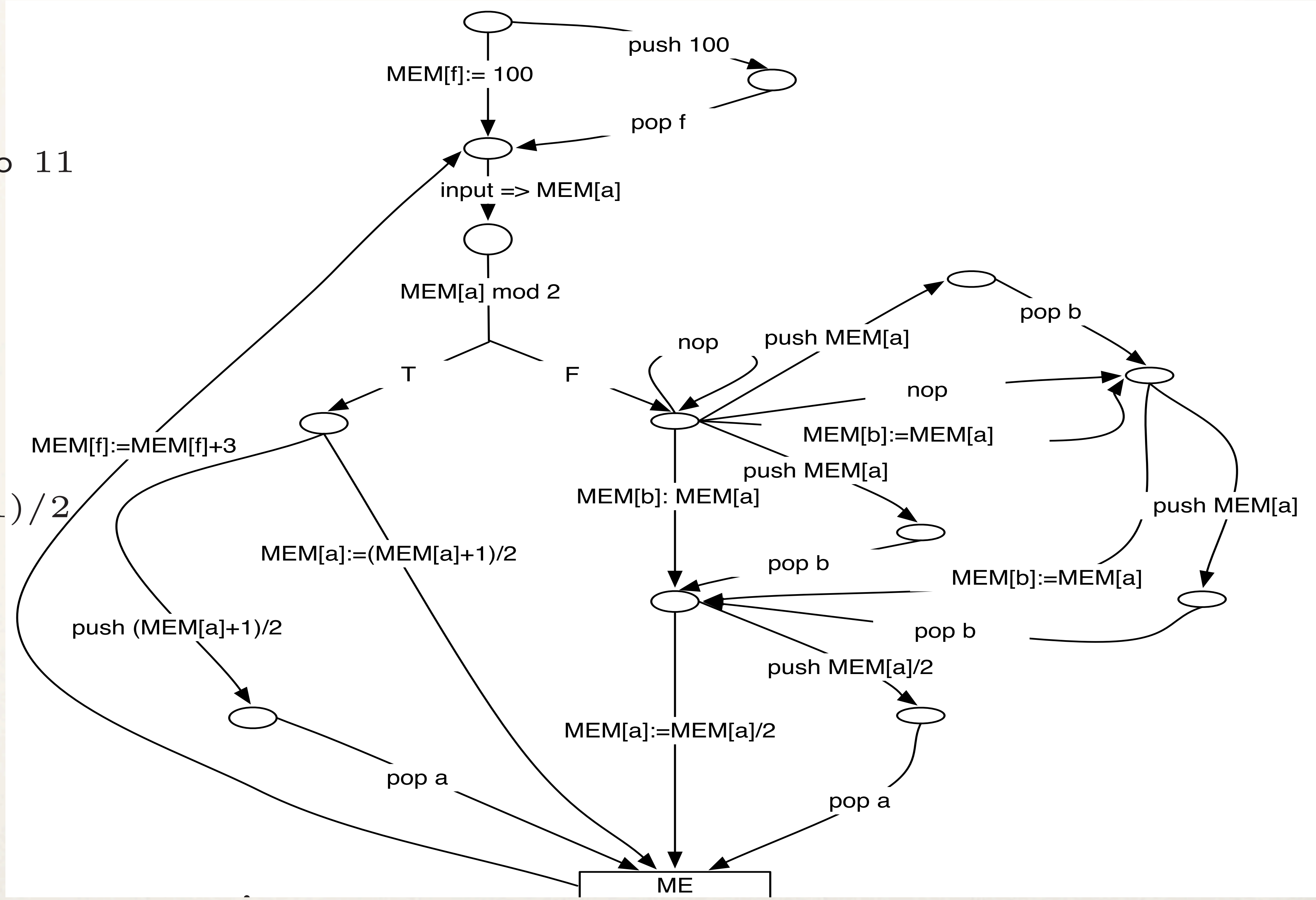


`MEM[f]:=100;input=>MEM[a];MEM[a] mod 2 = 0; MEM[b]:=MEM[a]; goto; MEM[b]:=MEM[a]; goto;...`

Example: code permutation + substitution

P_0^+ :

1 : goto 8
2 : if ($\mathtt{MEM}[a] \bmod 2$) goto 11
3 : nop
4 : goto 100
5 : push $\mathtt{MEM}[a]/2$
6 : pop a
7 : goto 12
8 : $\mathtt{MEM}[f] := 100$
9 : input $\Rightarrow \mathtt{MEM}[a]$
10 : goto 2
11 : $\mathtt{MEM}[a] := (\mathtt{MEM}[a] + 1)/2$
12 : ME
13 : goto 9
100 : push $\mathtt{MEM}[a]$
101 : pop b
102 : goto 5



Conclusions

What we have done!

- What we have:
  - A formal model of metamorphic code by Phase semantics
  - A method for approximating the Phase semantics
  - A computable approximation of regular metamorphism
- The approach:
  - requires no a priori knowledge about the metamorphic engine
  - is parametric on several abstractions (instructions, phases, metamorphism...)
  - is likely for refinement (grammars, constraints etc...)
  - suitable for semi-automatic malware analysis: *generation-test-refine*

What is missing?

- An adequate experimental evaluation (beyond toy examples....)
  - Pro: most malware implement relatively simple metamorphic engines (mostly regular) to foil syntactic signature checking
  - Con: hacking can easily foil any abstraction
- A practical solution: behavioral monitoring + FSA abstraction + widening
- More advanced abstractions: e.g., *context free metamorphism & grammar widening*
- The paper is a preliminary approach to a truly hard problem!
  - Next steps: experimental evaluation of regular metamorphism analysis, approximate behavioral monitoring.

Thanks!