Software Model Checking

Nicholas Kidd, nkidd@purdue.edu
November 18, 2009

Resources Used

- Edmund Clarke's *course*
    - And his *Turing Award Lecture*
- Online tutorial papers/slides:
    - Clarke, Merz, Schmidt, Muller-Olm, Visser, Rushby, Jhala,...
- Others slides from Akash Lal

Outline

- Model checking in 20 minutes
- Software Model Checking
  ... what's the big deal

Temporal Logic Model Checking
[Clarke]

- Model checking is an *automatic verification technique* for finite-state concurrent systems
- Developed independently by Clarke and Emerson and by Queille and Sifakis in early 80's
- Specifications are written in *propositional temporal logic*
- Verification procedure is an *exhaustive search* of the state space

Model Checking of Hardware Systems

- By definition a *finite* system
- Verification successes
  - Cache-coherence protocols
- Important bugs
  - *Intel Pentium FDIV bug* ($500 million)
  - See [Clarke] for more

Advantages of Model Checking

[Clarke]

- No proofs
- Fast
- Counterexamples
- Partial specifications OK
- Logics can express *concurrency properties*

Mutual Exclusion

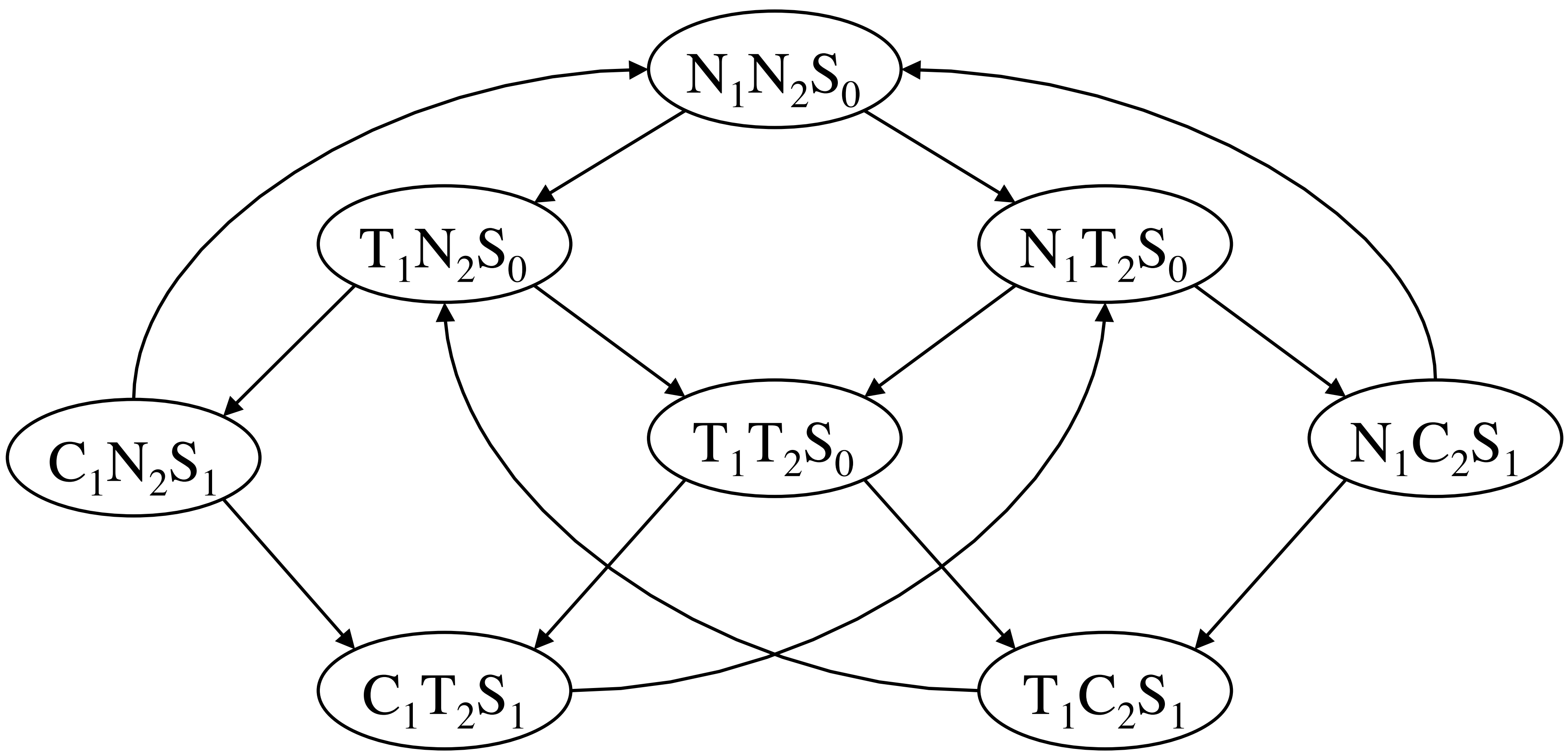
[Visser, ASE 2000]

- Two process mutual exclusion
- Each process has 3 states:
  - Non-critical (N), Trying (T), Critical (C)
- Semaphore is free (S_0) or taken (S_1)

$$
\begin{array}{lcl}
N_1 & \rightarrow & T_1 \\
T_1 \,\&\, S_0 & \rightarrow & C_1 \& S_1 \\
C_1 & \rightarrow & N_1 \& S_0
\end{array}
\quad || \quad
\begin{array}{lcl}
N_2 & \rightarrow & T_2 \\
T_2 \,\&\, S_0 & \rightarrow & C_2 \& S_1 \\
C_2 & \rightarrow & N_2 \& S_0
\end{array}
$$

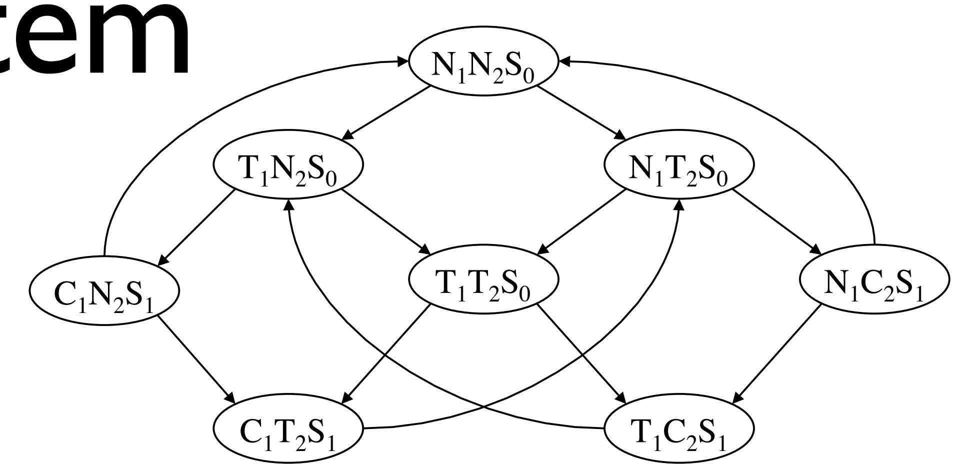
Mutual Exclusion

[Visser, ASE 2000]



Finite-State System

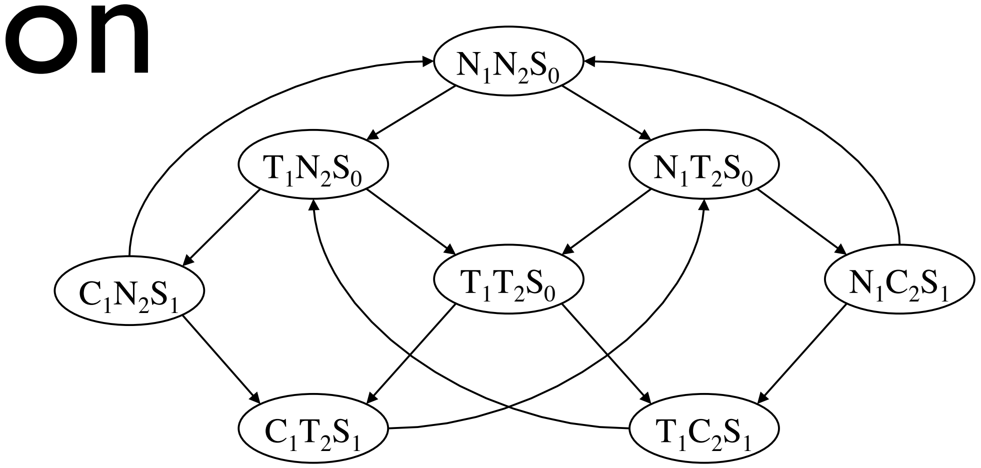
[Clarke]



- State-transition graph (*Kripke* structure)
  - Set of states S
  - Set of atomic propositions AP
  - Labeling function $L : S \to 2^{AP}$
  - Transition relation $R \subseteq S \times S$

Mutual Exclusion

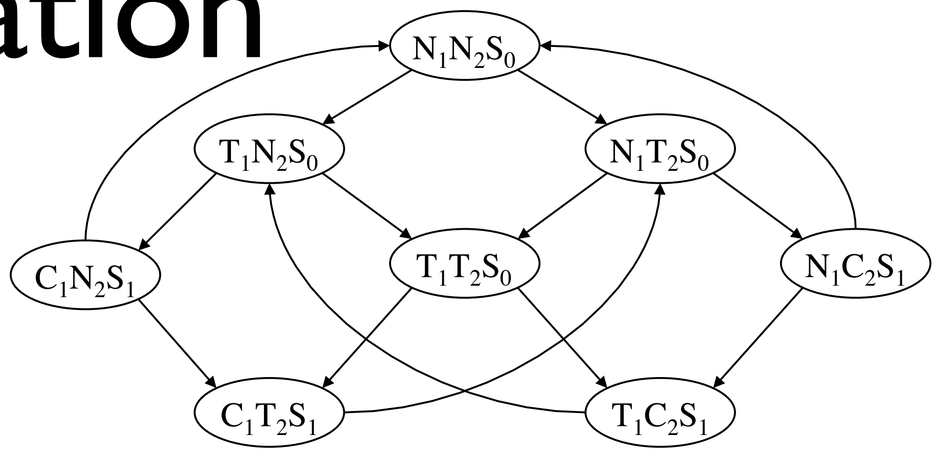
[Visser, ASE 2000]



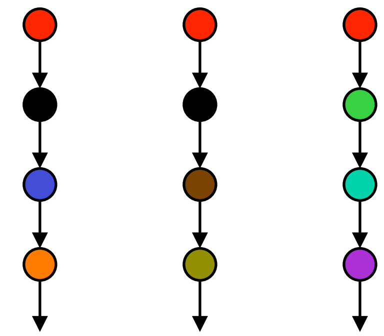
- *Goal*: Show that from any state, it is always possible to get to the initial state
- Need way to specify property of *execution traces*!
  - *Liveness* -- Something good eventually happens (infinitely often)
  - Safety -- Something bad does not happen

Property Specification

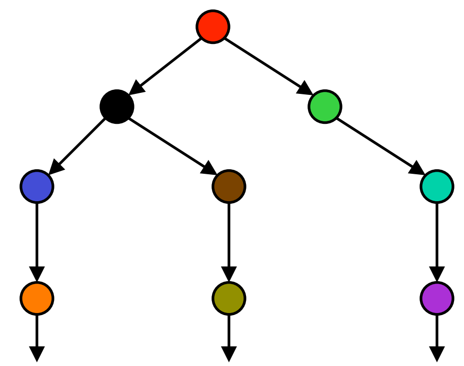
[Visser, ASE 2000]



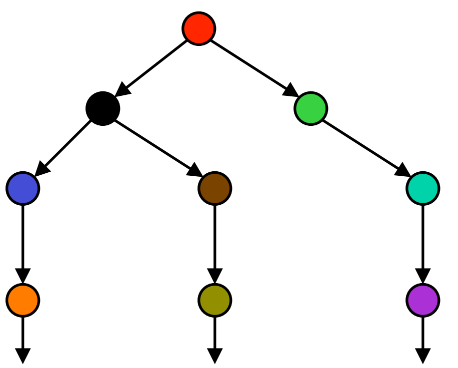
- *Temporal Logic*: Express properties of event orderings in time
- Linear Time (LTL)
  - Every moment has a unique successor
  - Infinite sequences

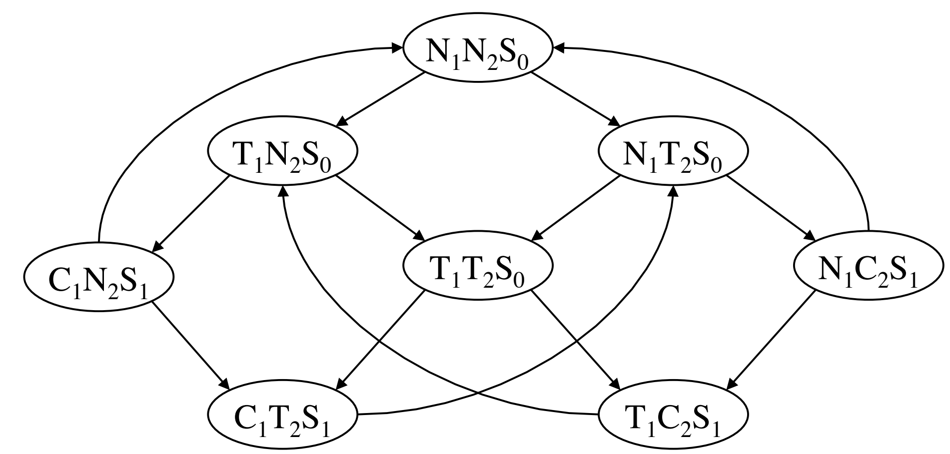


- Branching Time (CTL)
  - Every moment has >= 1 successors
  - Infinite tree



CTL
[Clark]

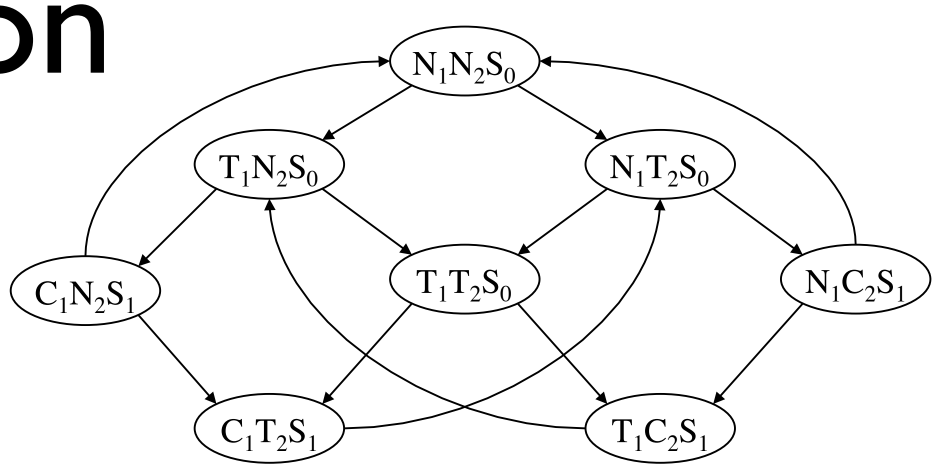




- Formula built from *path quantifiers* and *temporal operators*
- *Path quantifier*
  - A ≡ "On *All* paths" (all my children)
  - E ≡ "*Exists* some path" (>= 1 of my children)
- Temporal Operator
  - (X|F|G) p ≡ p holds (*neXt* | *Globally* | *Future*)

Mutual Exclusion

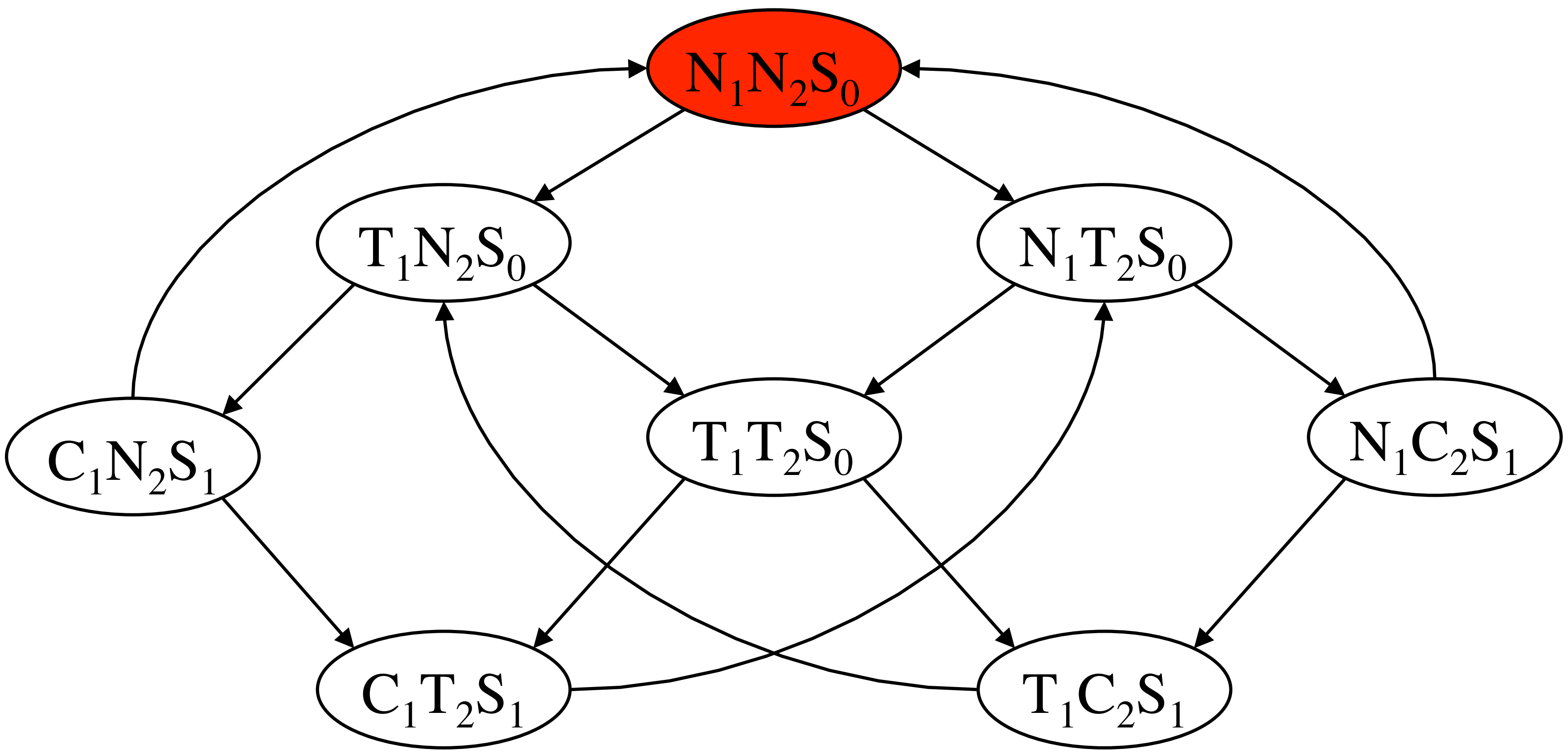
[Visser, ASE 2000]



- *Goal*: Show that from any state, it is always possible to get to the initial state
- AG EF (N_0 & N_1 & S_0)

Mutual Exclusion

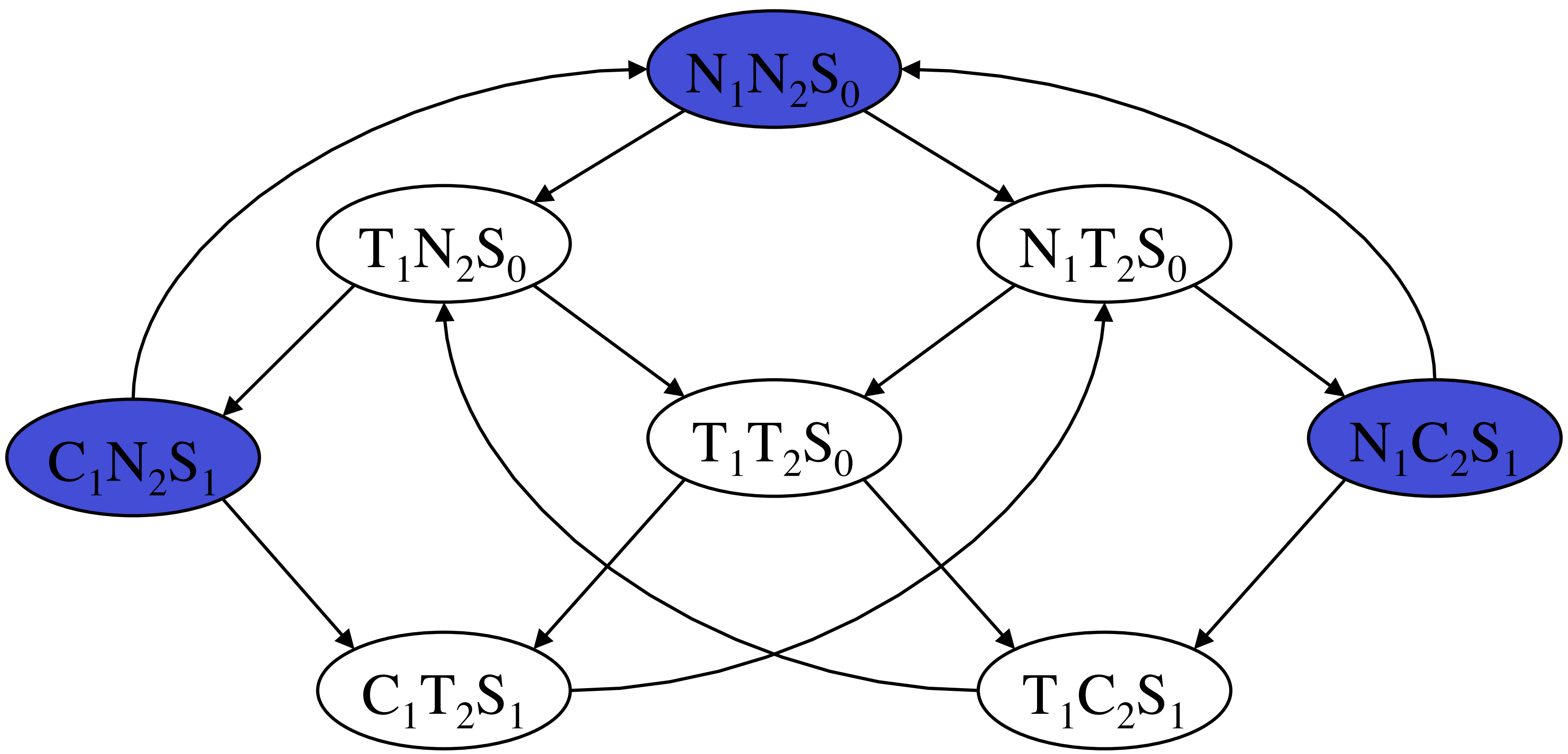
[Visser, ASE 2000]



$$K \models AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$$

Mutual Exclusion

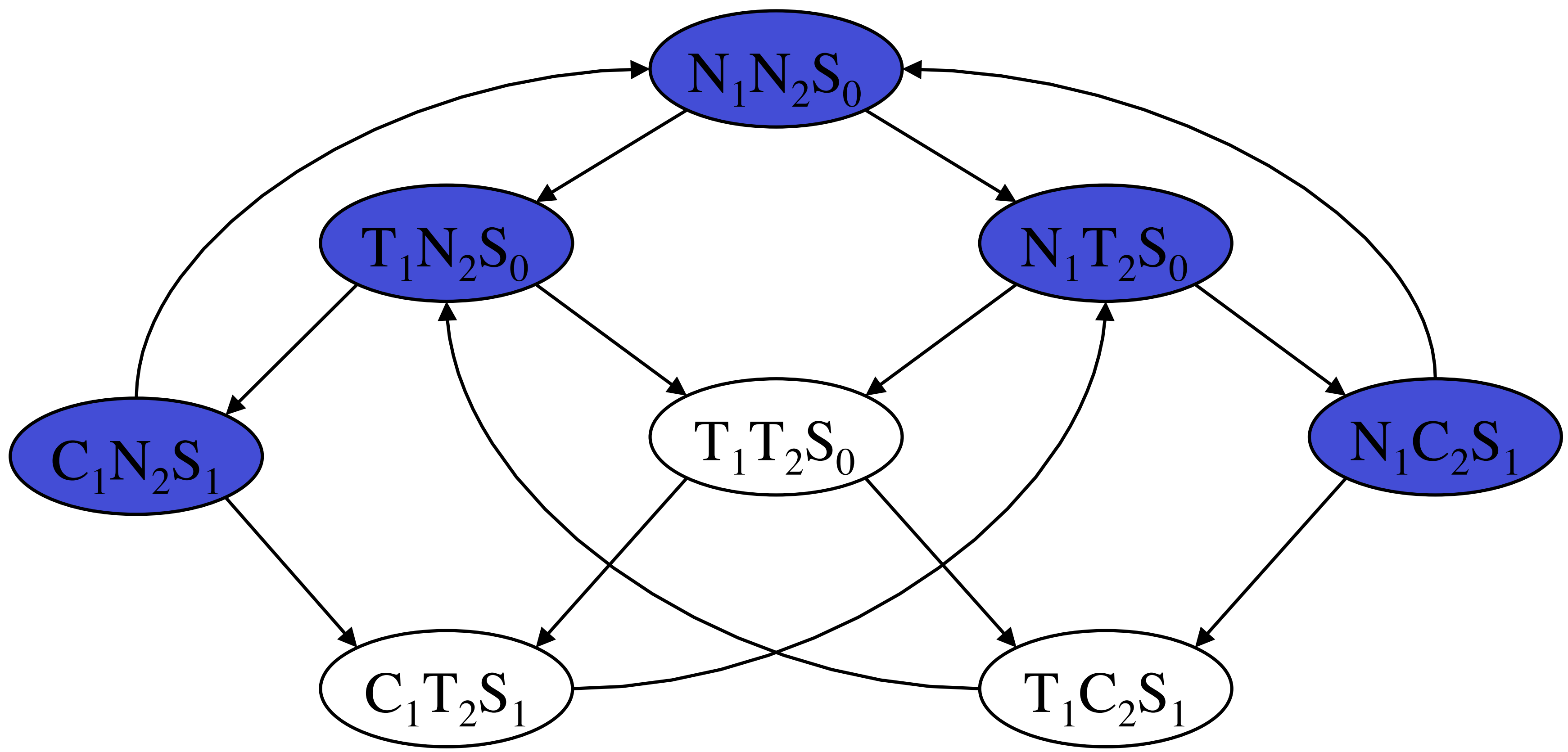
[Visser, ASE 2000]



$$K \models AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$$

Mutual Exclusion

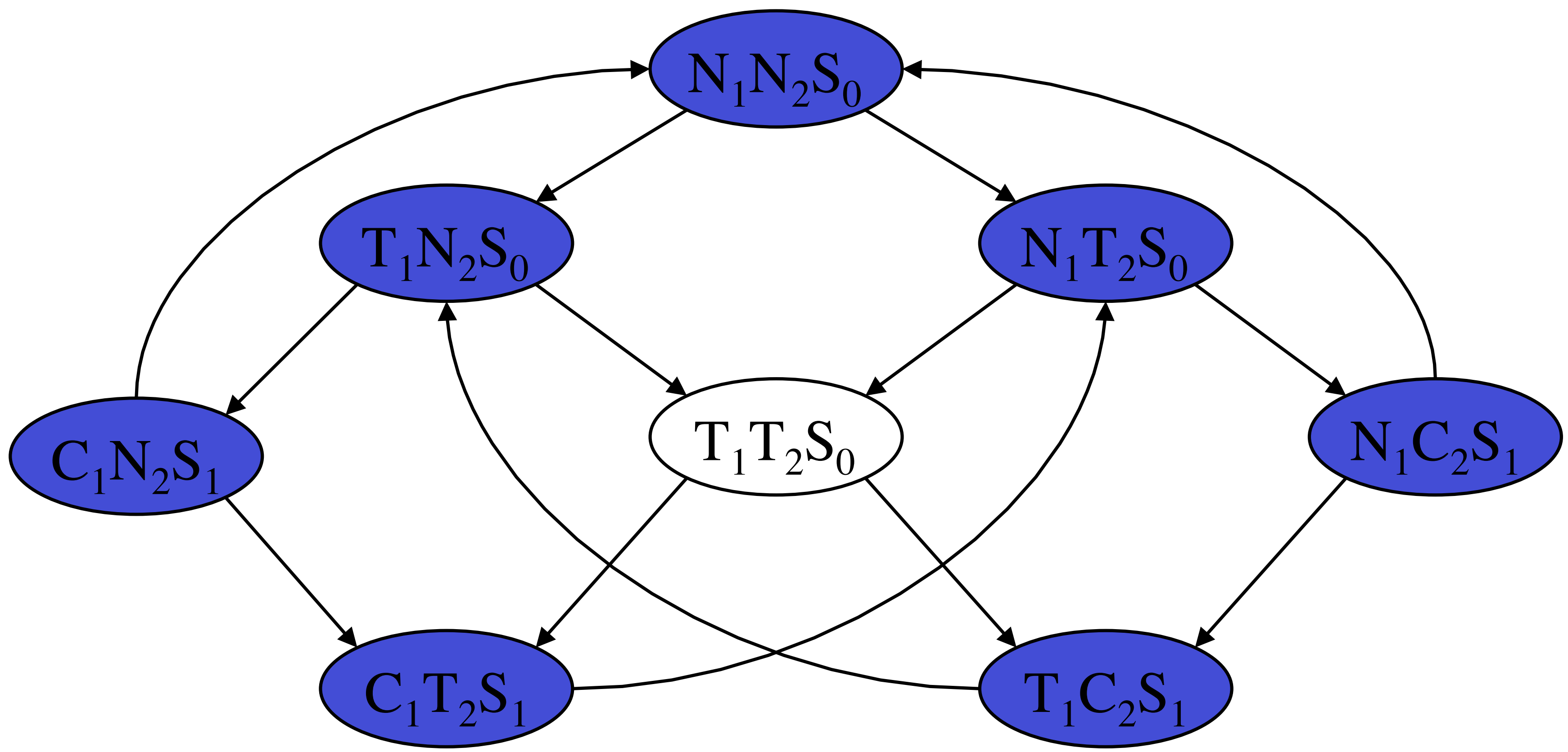
[Visser, ASE 2000]



$$K \models AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$$

Mutual Exclusion

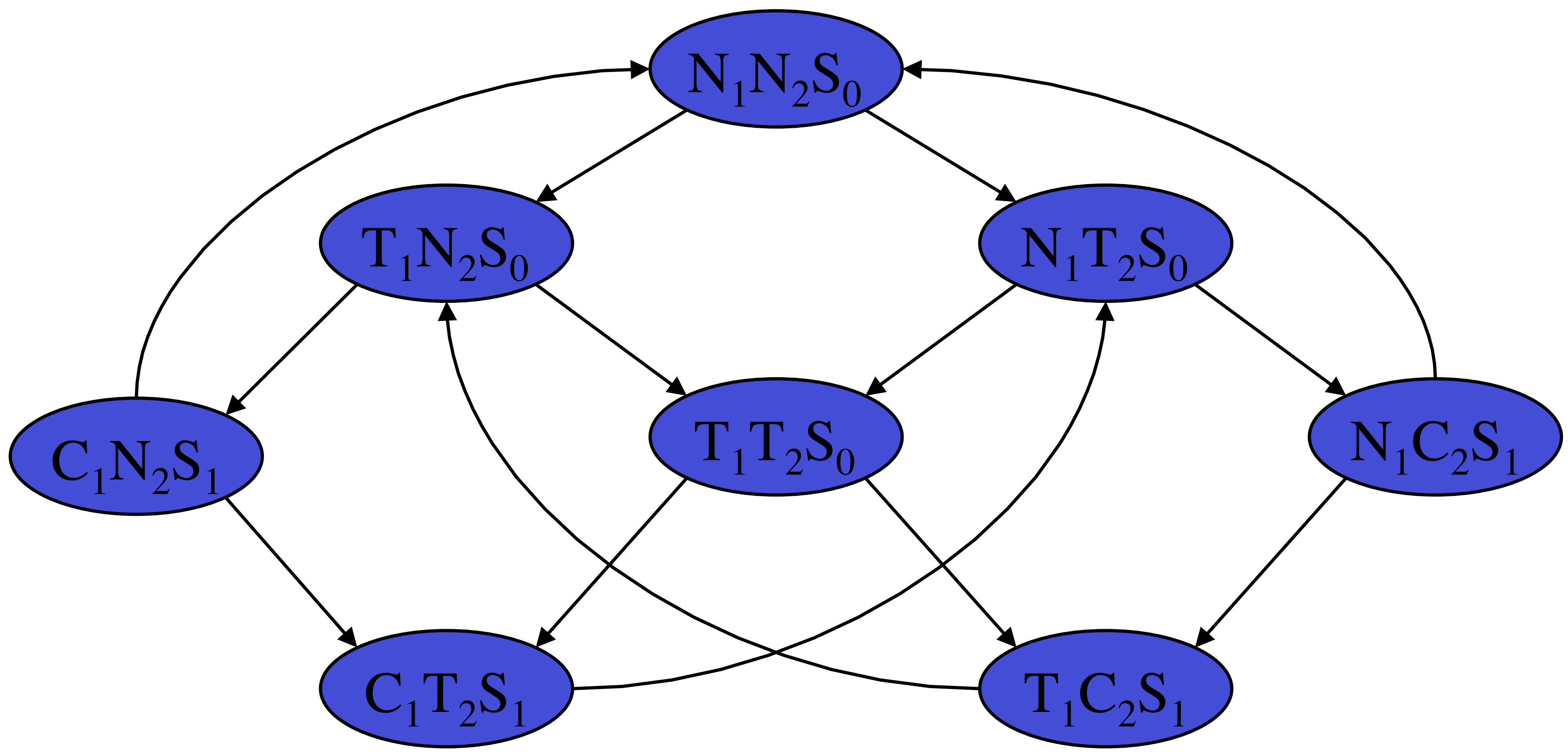
[Visser, ASE 2000]



$$K \models AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$$

Mutual Exclusion

[Visser, ASE 2000]

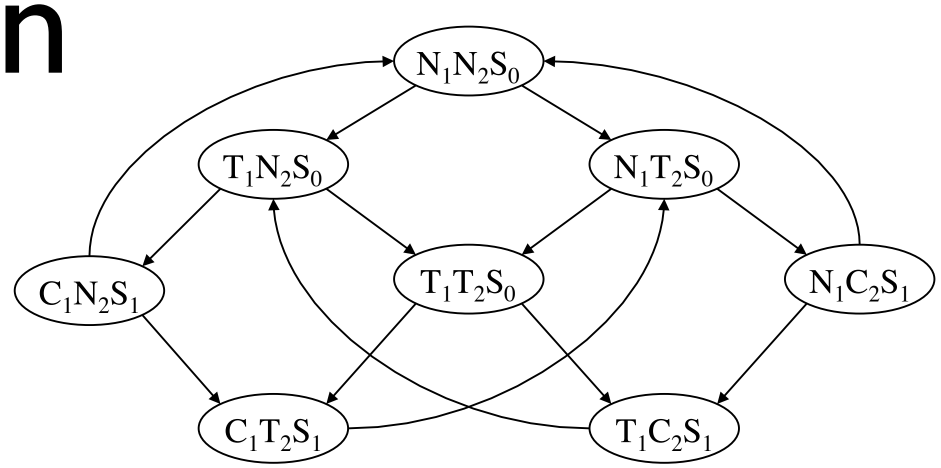


$$K \models AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$$

Property Verification
[Visser, ASE 2000]

- For Kripke structure $M = (S,R,L)$, find set of states satisfying TL formula f
  - $\{ s \in S \mid M,s \models f \}$
- Property verified if initial states $I \subseteq S$ are in the set
- If M,i $\not\models$f, for some i$\in$I, give witness trace (tree)
  - Also called counterexample

State Explosion



- n states
- m threads
- $O(n^m)$ is size of total state space

$N_1 \rightarrow T_1$		$N_2 \rightarrow T_2$
$T_1 \,\&\, S_0 \rightarrow C_1 \& S_1$	$\|\|$	$T_2 \,\&\, S_0 \rightarrow C_2 \& S_1$
$C_1 \rightarrow N_1 \& S_0$		$C_2 \rightarrow N_2 \& S_0$

Addressing State Explosion

- Symbolic Model Checking
  - Binary Decision Diagrams (BDDs)
- Partial order reduction
- Bounded Model Checking

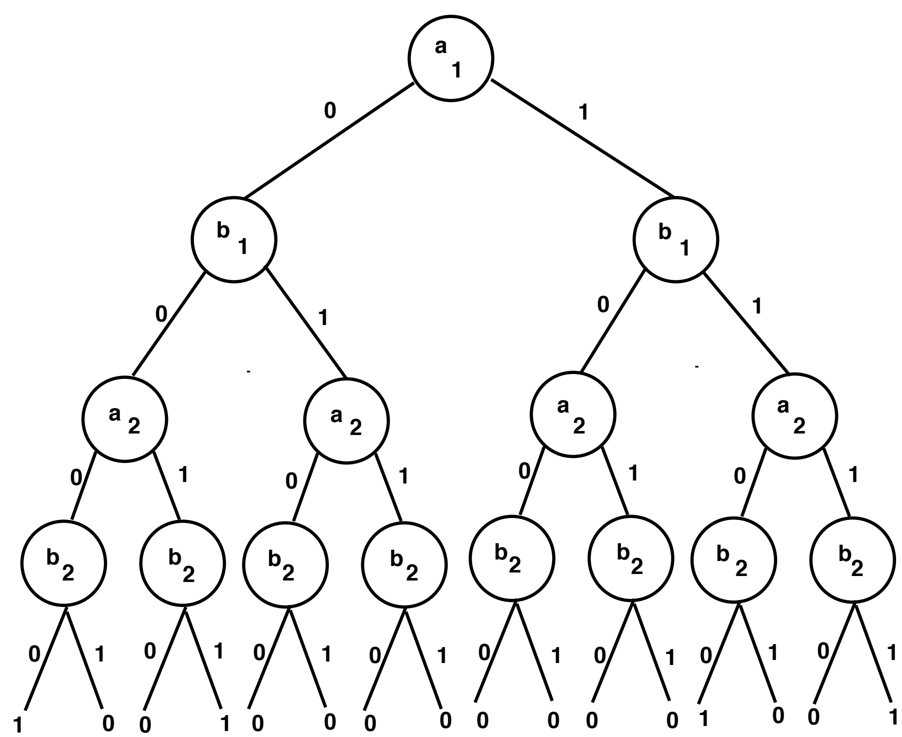
BDDs

Binary Decision Trees (Cont.)

A BDD for the two-bit comparator given by the formula

$$f(a_1, a_2, b_1, b_2) = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2),$$

is shown in the figure below:

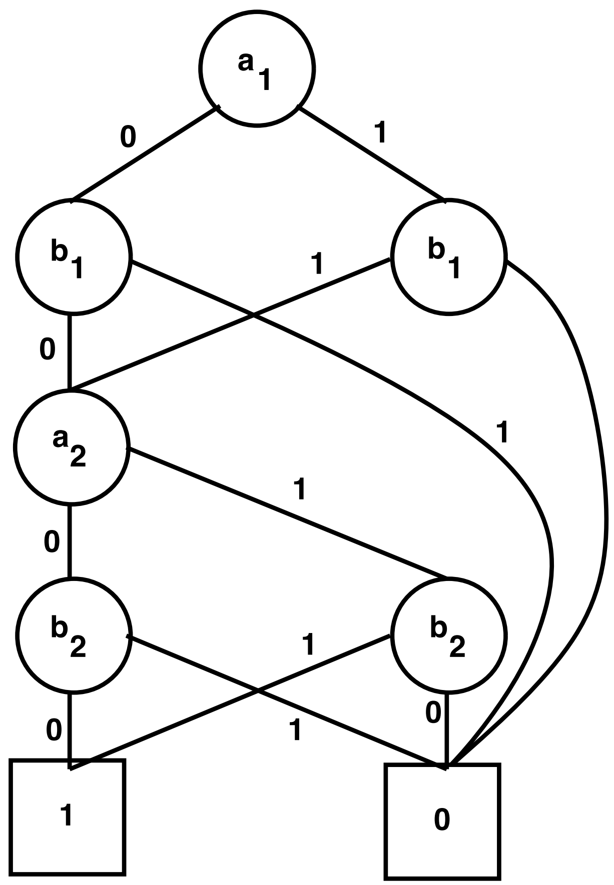


- http://www.cs.cmu.edu/afs/cs.cmu.edu/usr/emc/www/15817-f09/lecture1.pdf

BDDs

OBDD for Comparator Example

If we use the ordering $a_1 < b_1 < a_2 < b_2$ for the comparator function, we obtain the OBDD below:



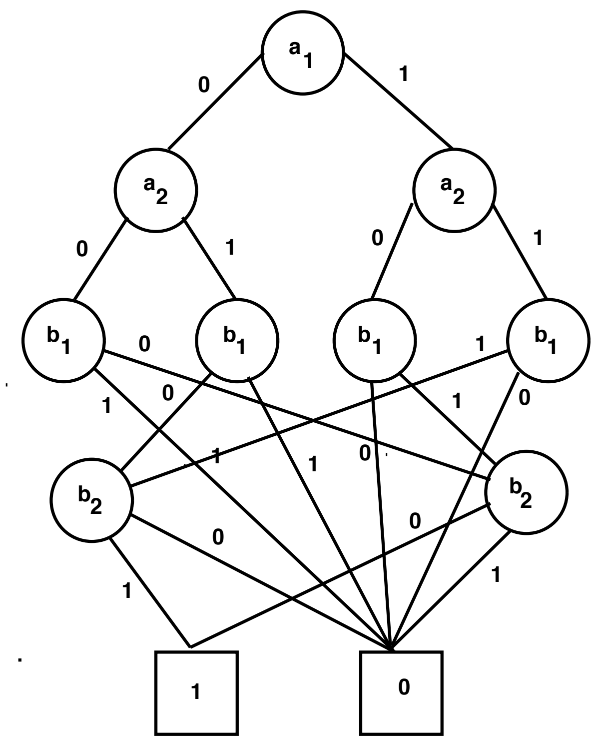
- http://www.cs.cmu.edu/afs/cs.cmu.edu/usr/emc/www/15817-f09/lecture1.pdf

BDDs

Variable Ordering Problem

The size of an OBDD depends critically on the variable ordering.

If we use the ordering $a_1 < a_2 < b_1 < b_2$ for the comparator function, we get the OBDD below:

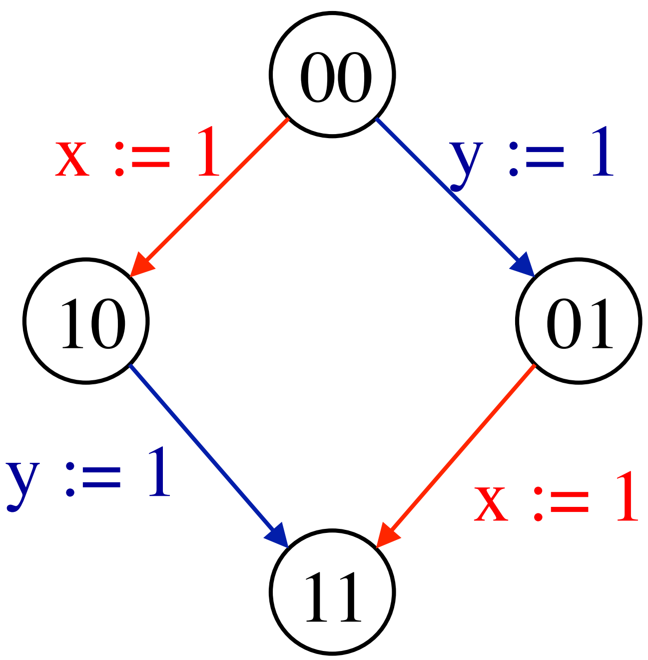


- http://www.cs.cmu.edu/afs/cs.cmu.edu/usr/emc/www/15817-f09/lecture1.pdf

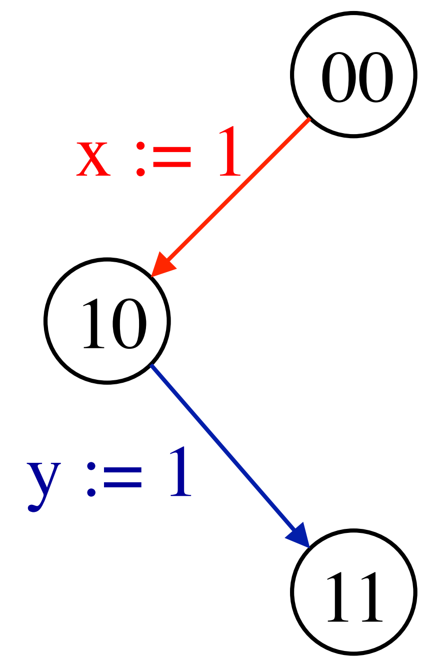
Partial Order Reduction

[Visser, ASE 2000]

x := 1 || y := 1 where initially x = y = 0



No Reductions



States Reduced

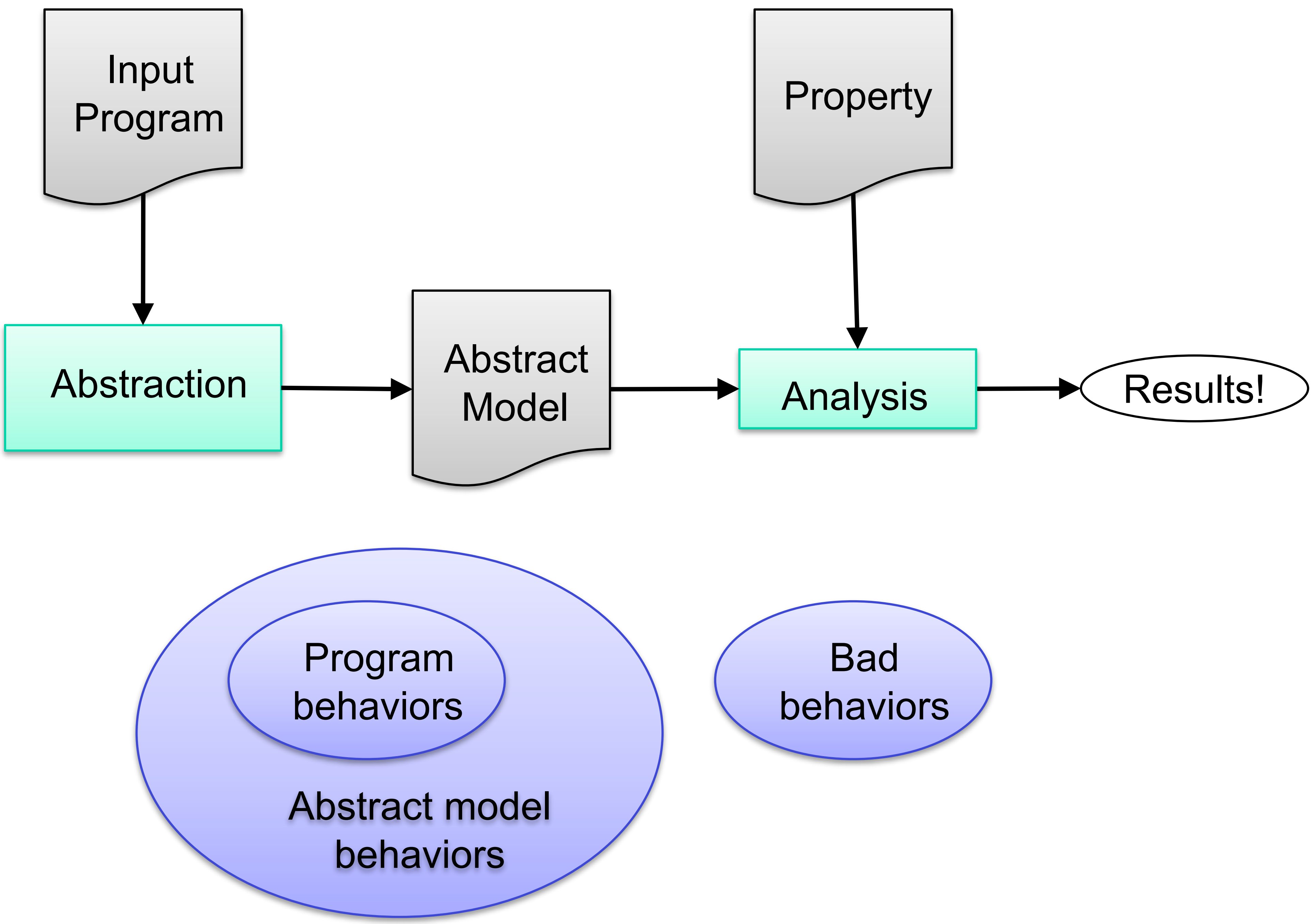
Bounded Model Checking

- Don't do *exhaustive* exploration of state space
- Search *bounded-depth* execution traces
- Encode as SAT to use fast solvers
- http://www.cprover.org/cbmc/

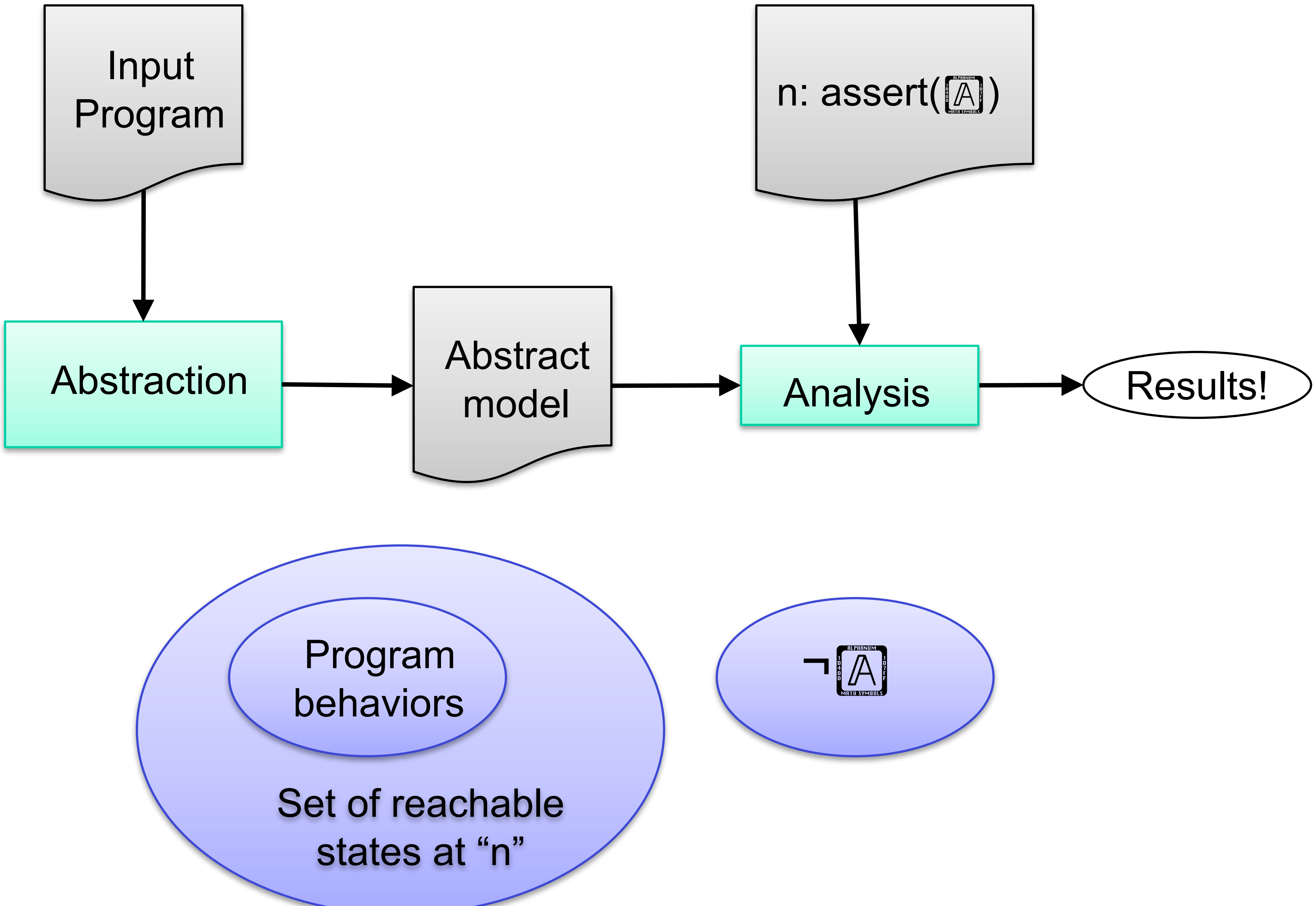
Software Model Checking

- *Idea*: apply same techniques to software verification (automatic, no proofs, counterexamples, ...)
- Programs are *infinite* state!
  - Memory allocation, recursion, memory values, input/output, ...
  - Use *abstraction*
  - Use *infinite* Kripke structures

Software Model Checking



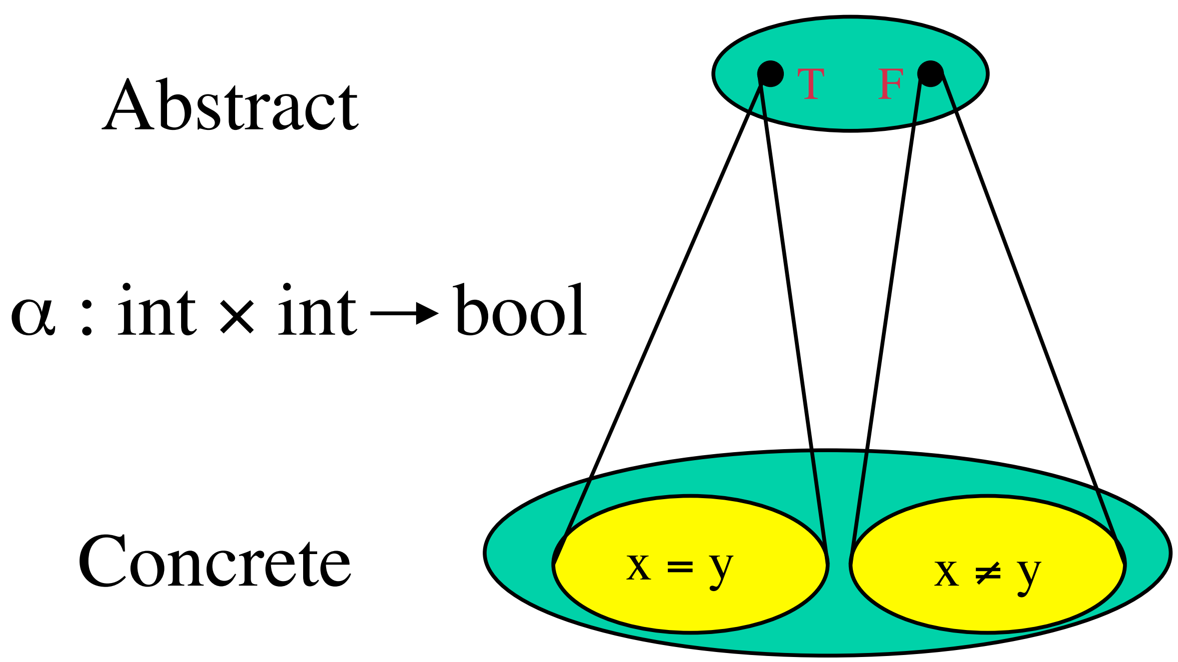
Software Model Checking

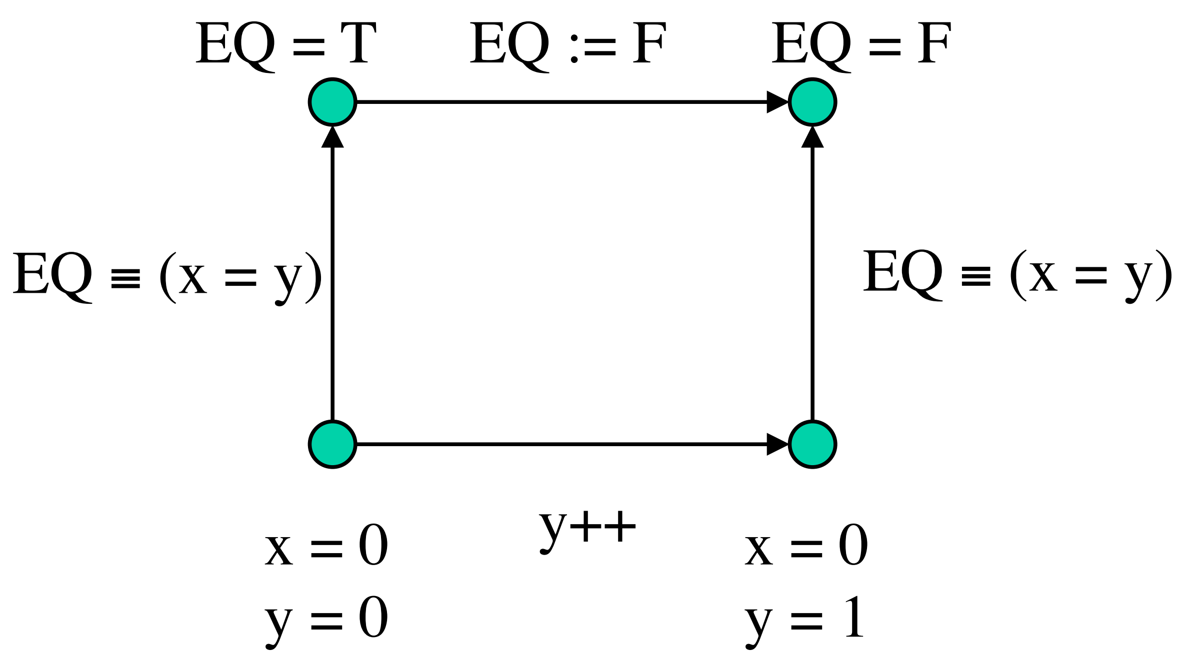


Predicate Abstraction

[Graf & Saidi] [Slide: Visser, ASE 2000]

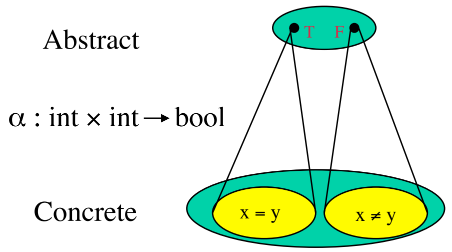
- Map *concrete* system to *abstract* system, whose states correspond to truth values of set of predicates

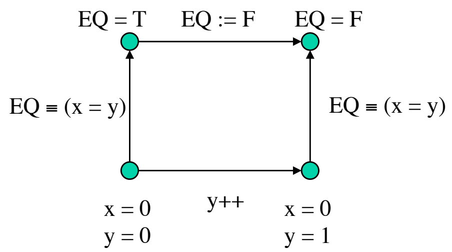




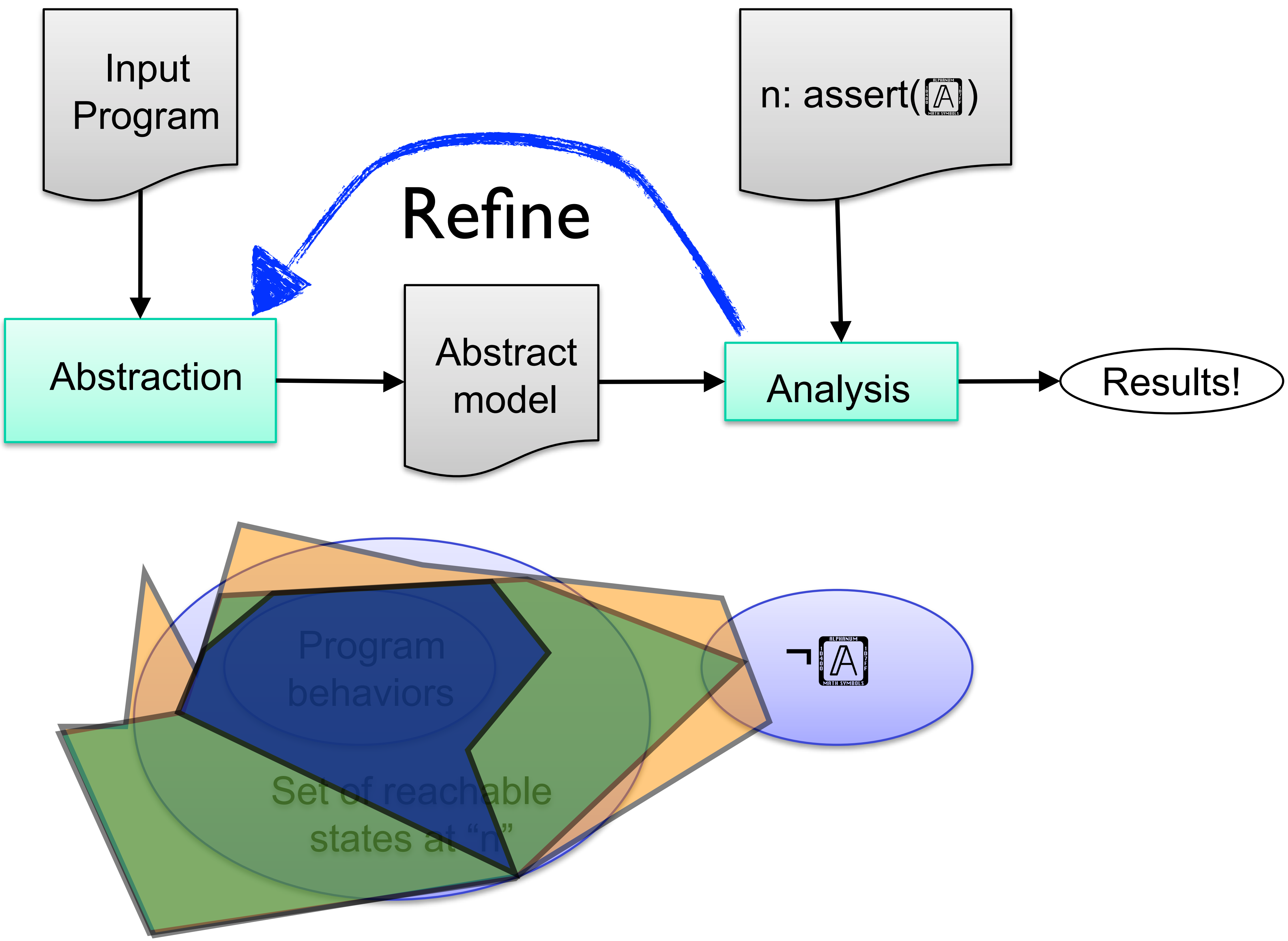
Predicate Abstraction
[Visser, ASE 2000]

- Map *concrete* system to *abstract* system, whose states correspond to truth values of set of predicates
- Adds *spurious* behaviors to abstract system
- Requires *verifying* counterexample validation if bug found





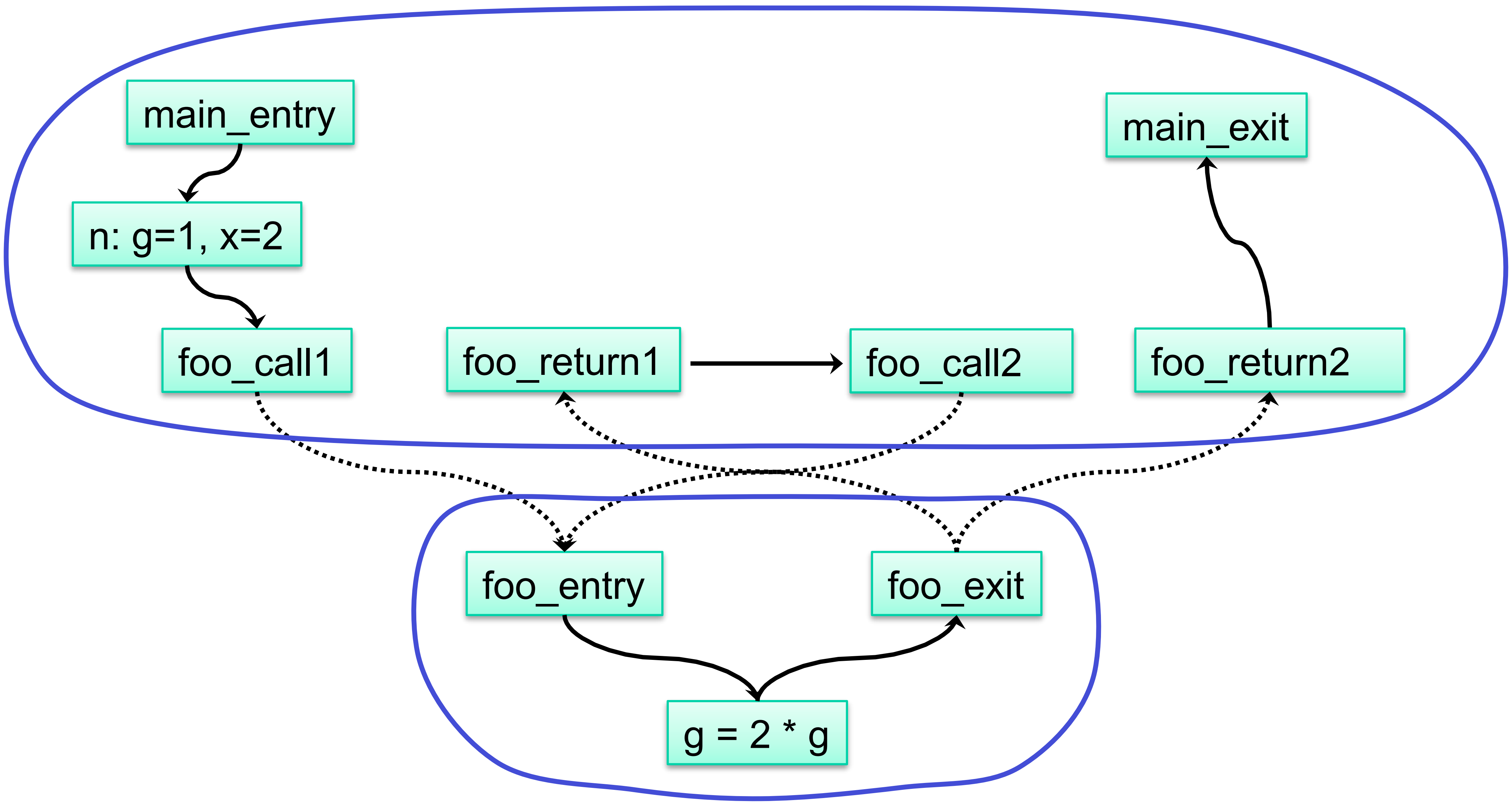
Software Model Checking



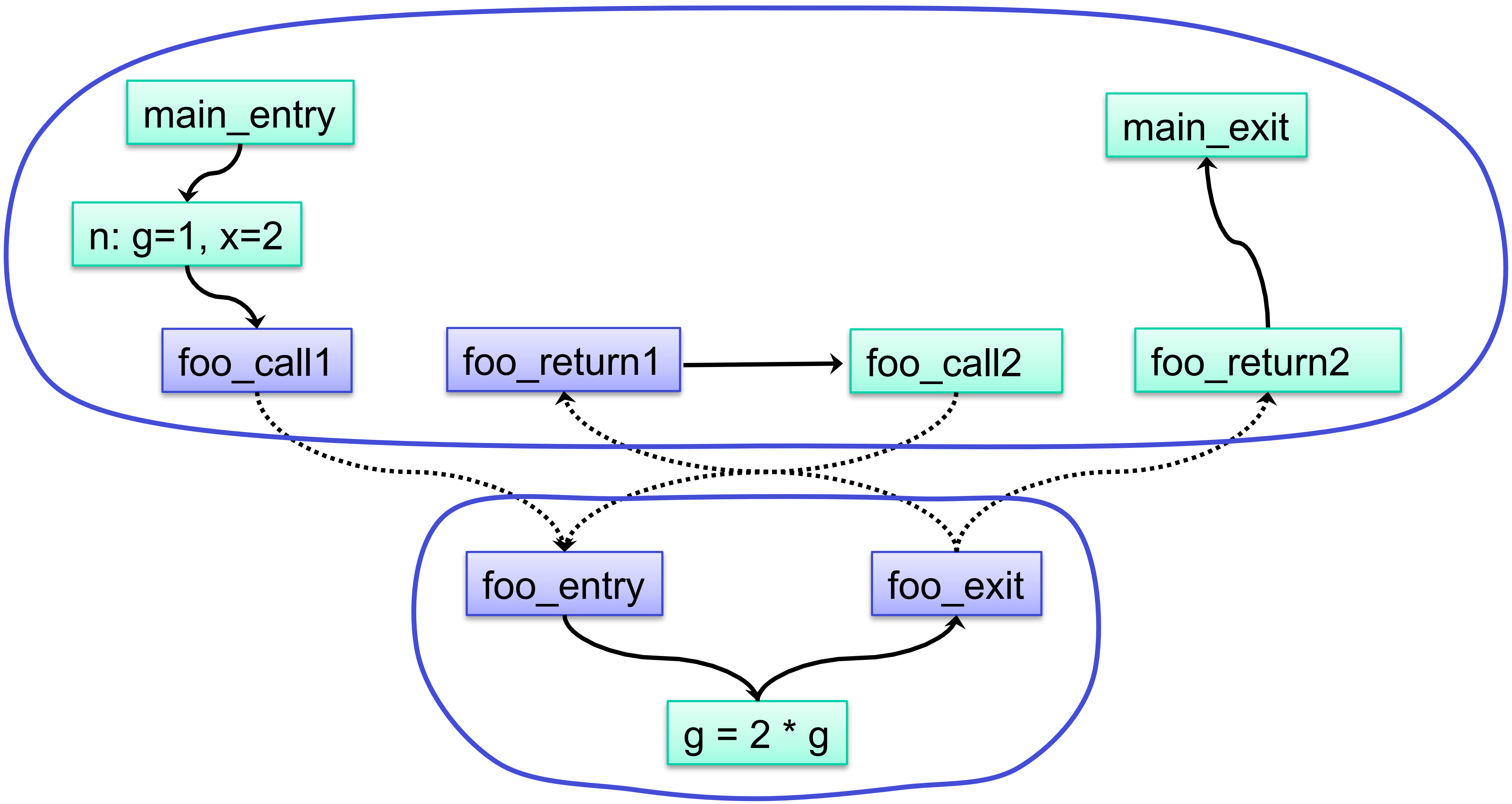
What about the stack

- Programs have a stack
    - Support component reuse and *modularity*
    - Support *recursion*
- *Context-sensitive analysis*
    - Precise with respect to matched calls and returns

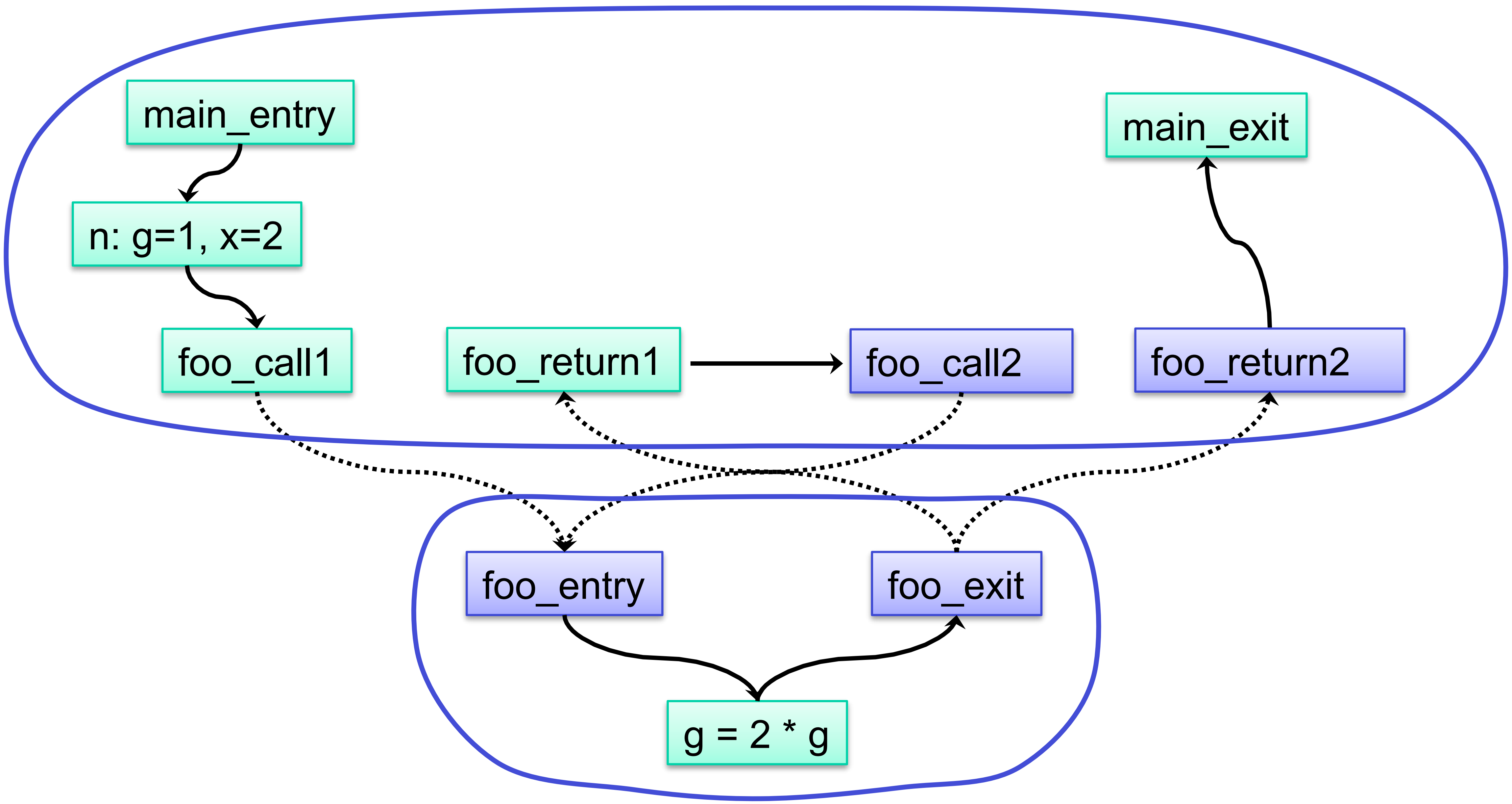
Matched Calls and Returns



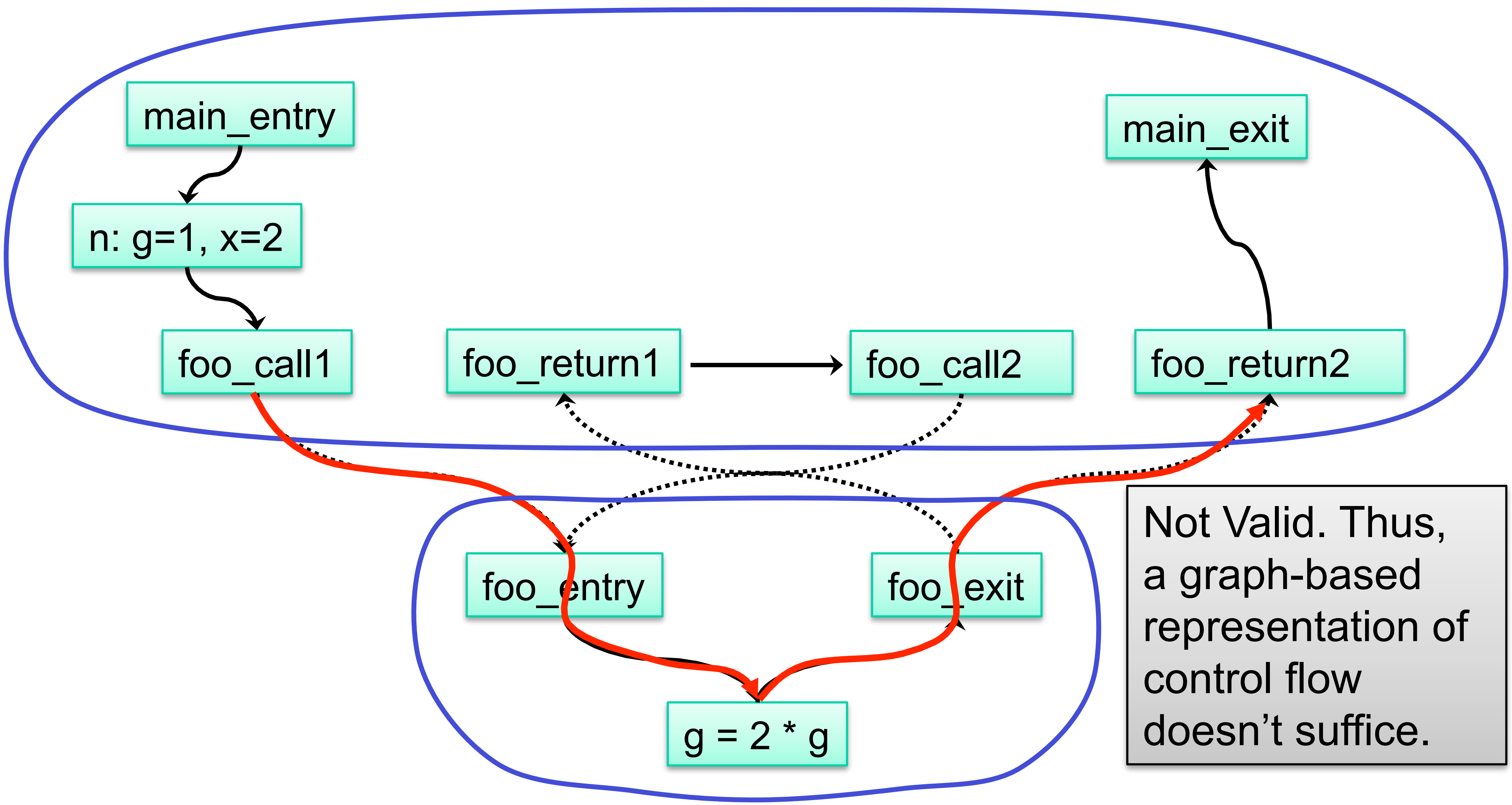
Matched Calls and Returns



Matched Calls and Returns

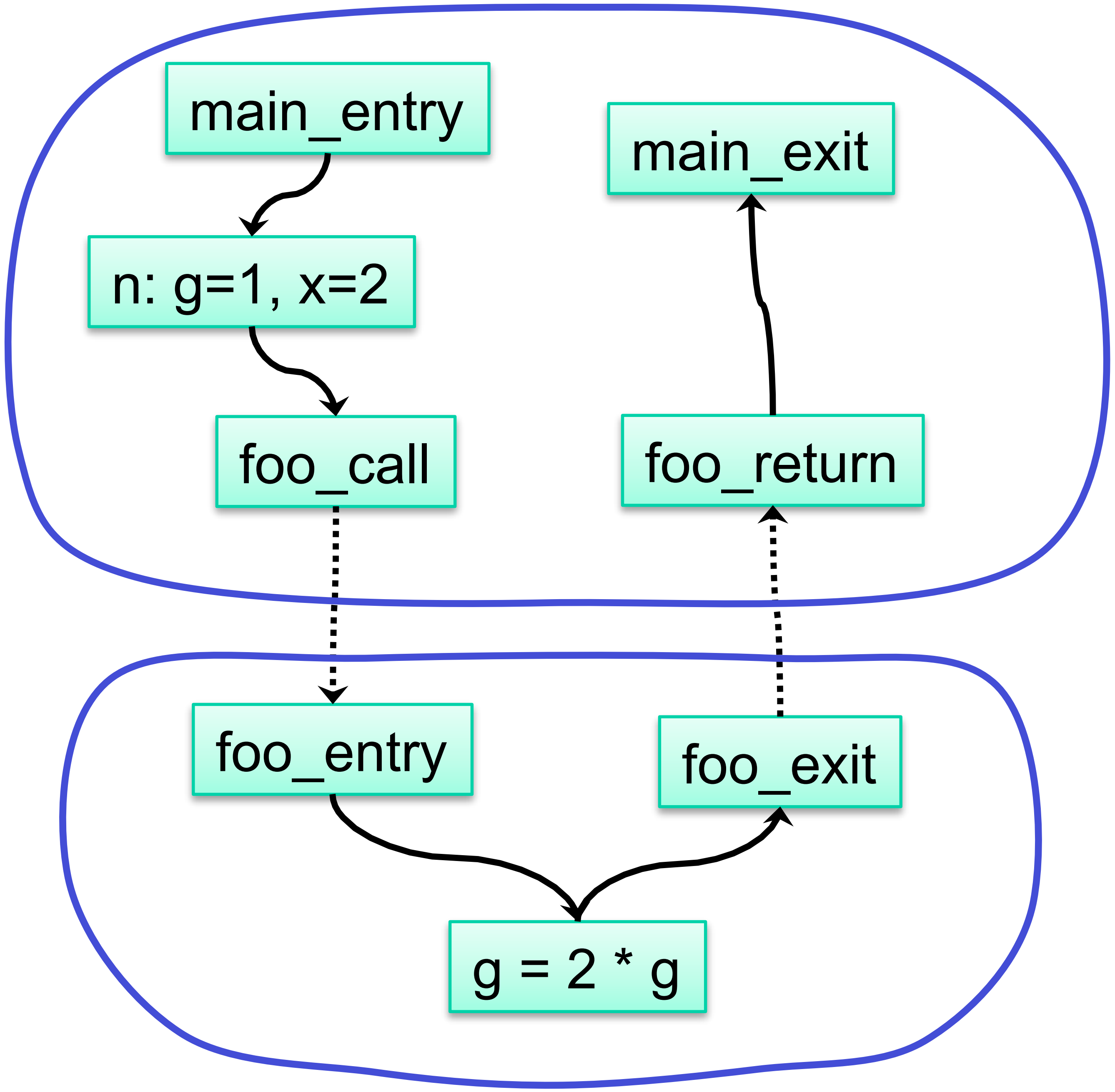


(Un)Matched Calls and Returns

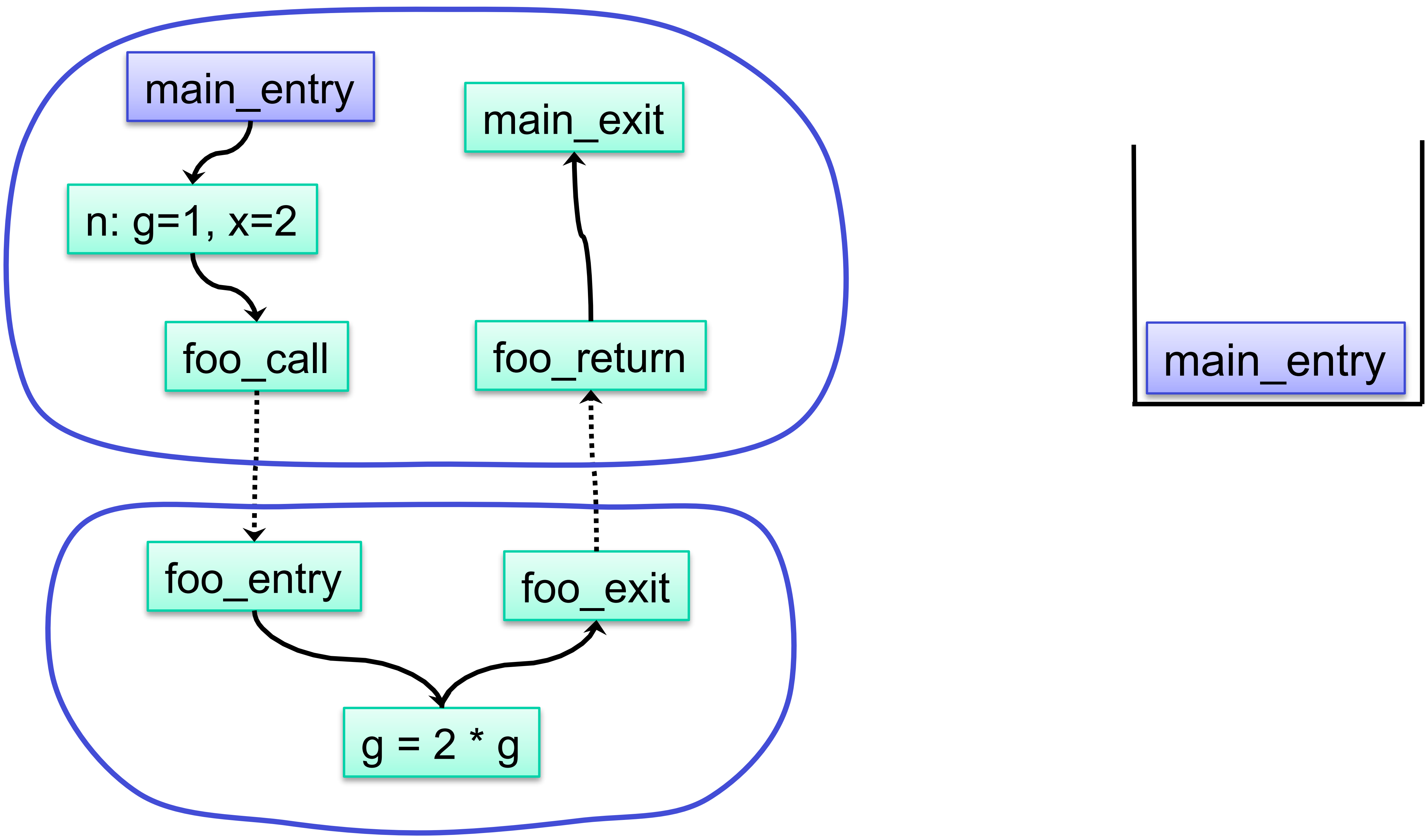


Not Valid. Thus, a graph-based representation of control flow doesn't suffice.

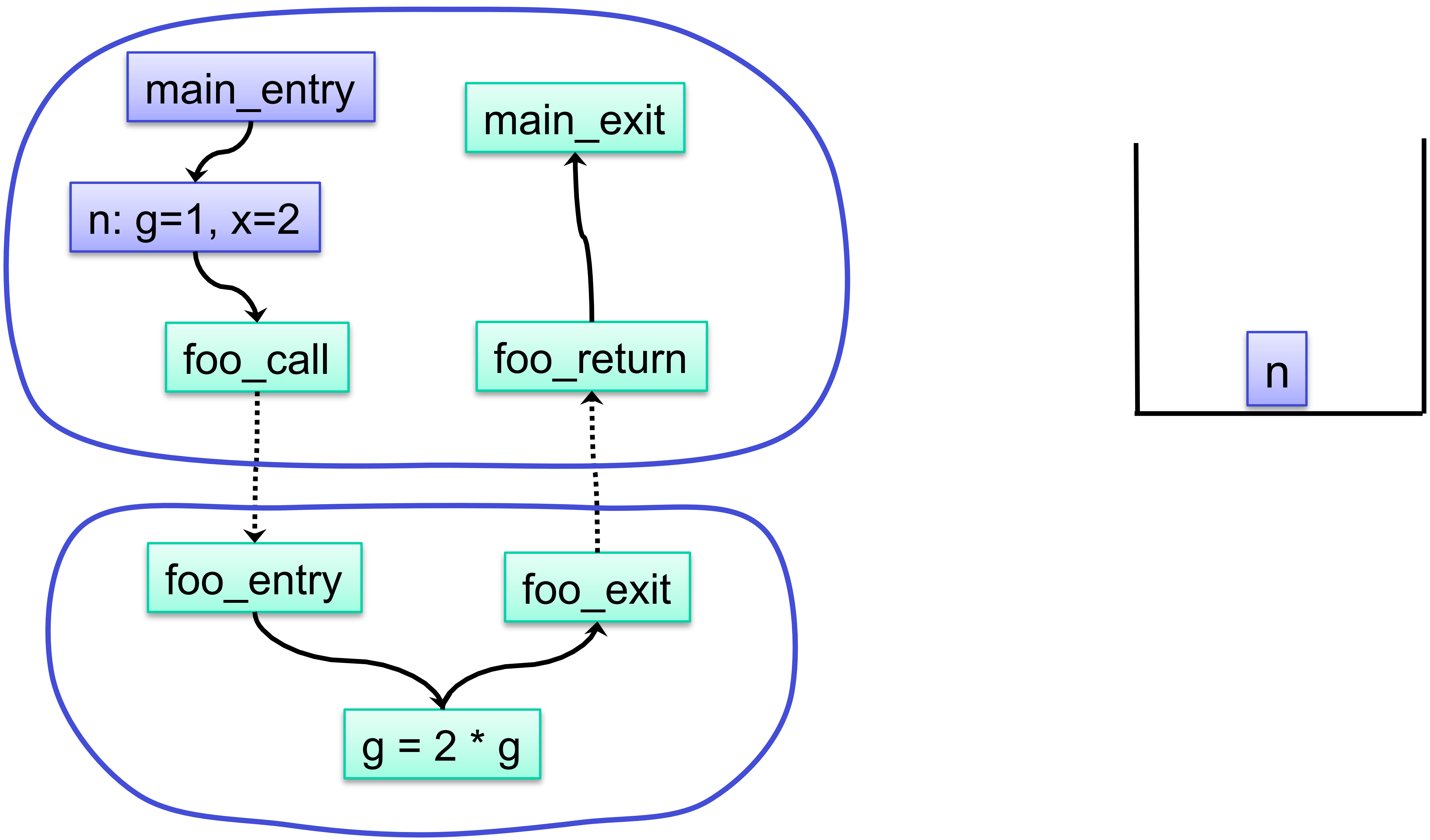
Modeling the Stack



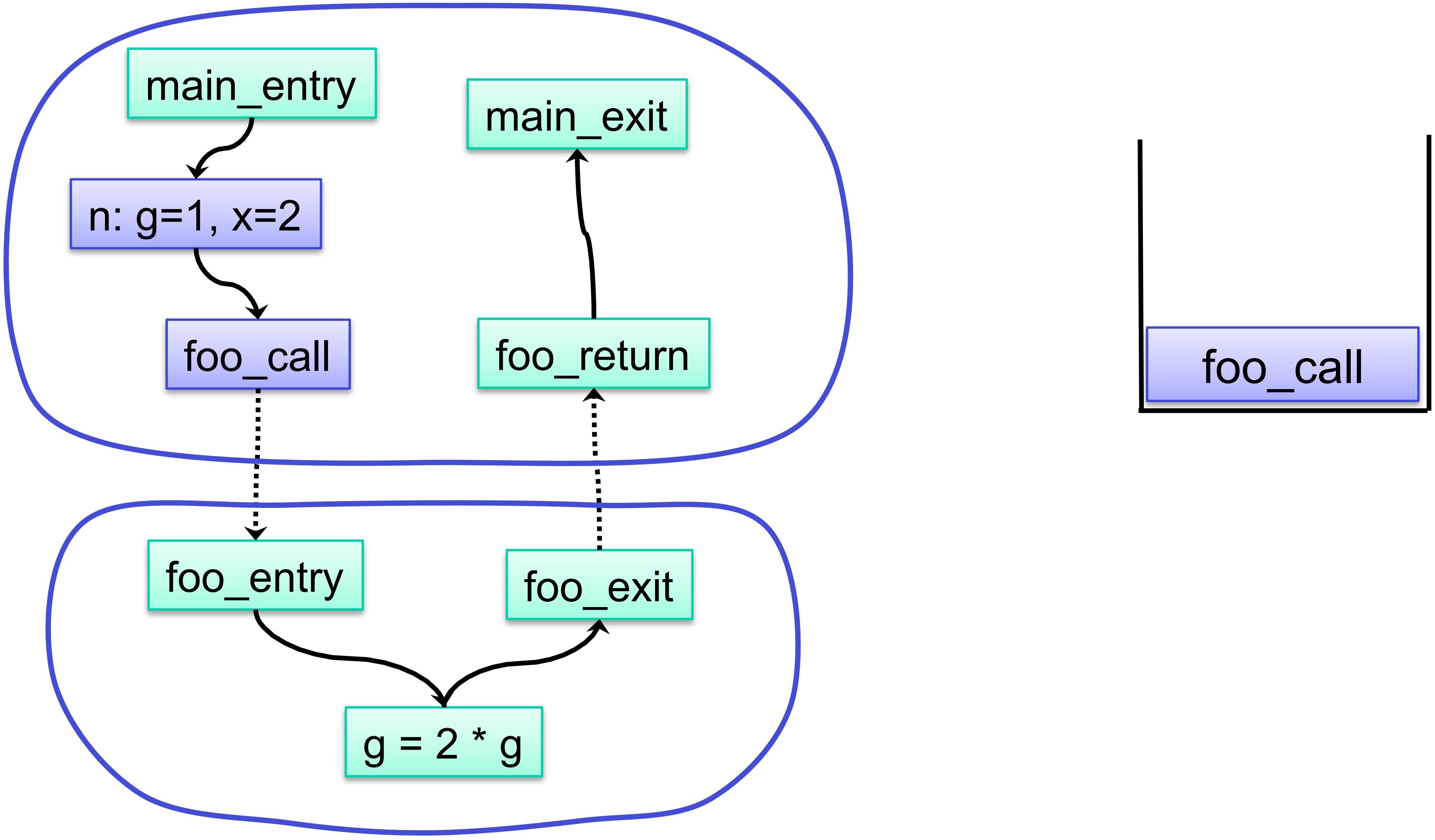
Modeling the Stack



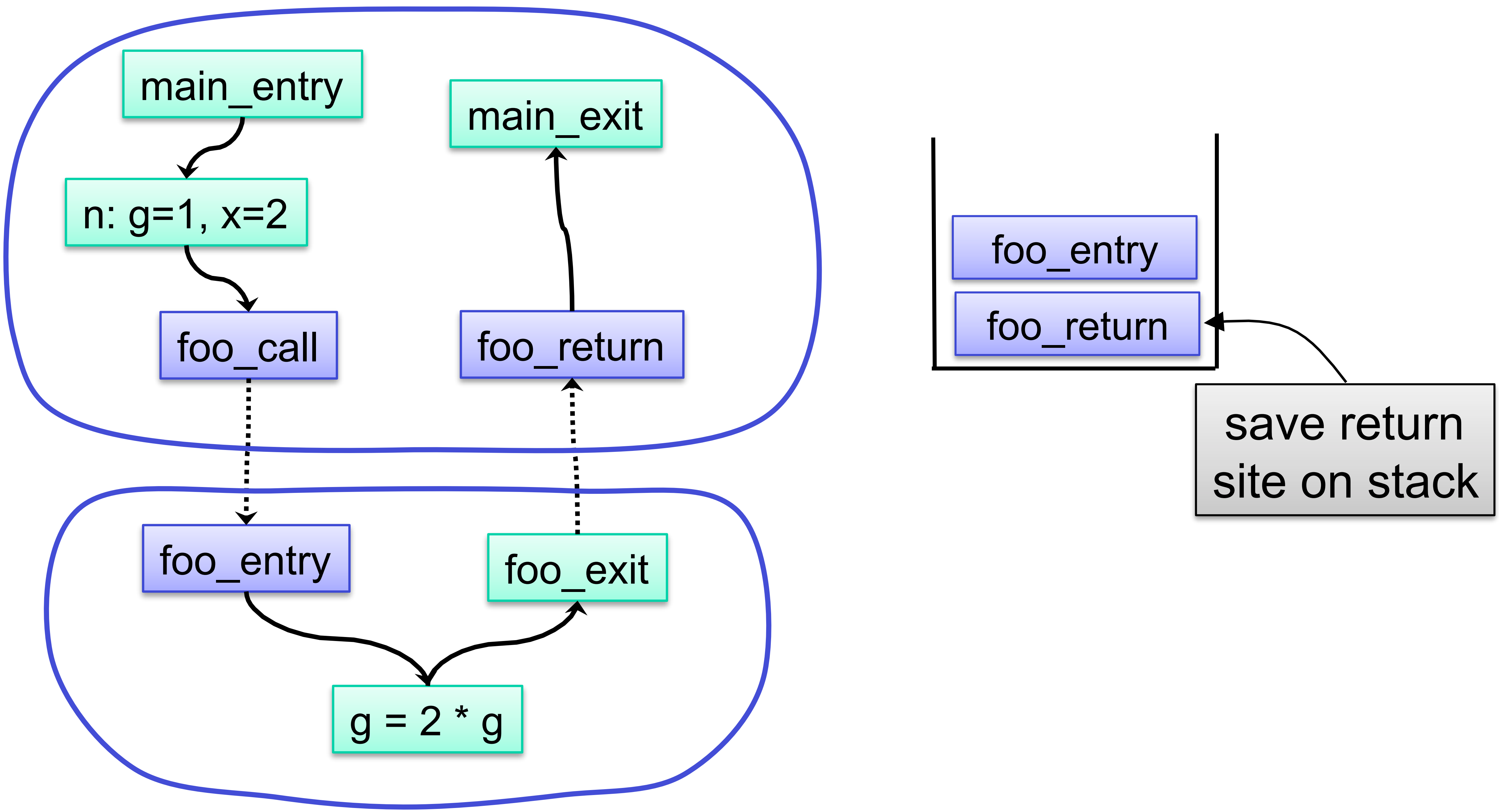
Modeling the Stack



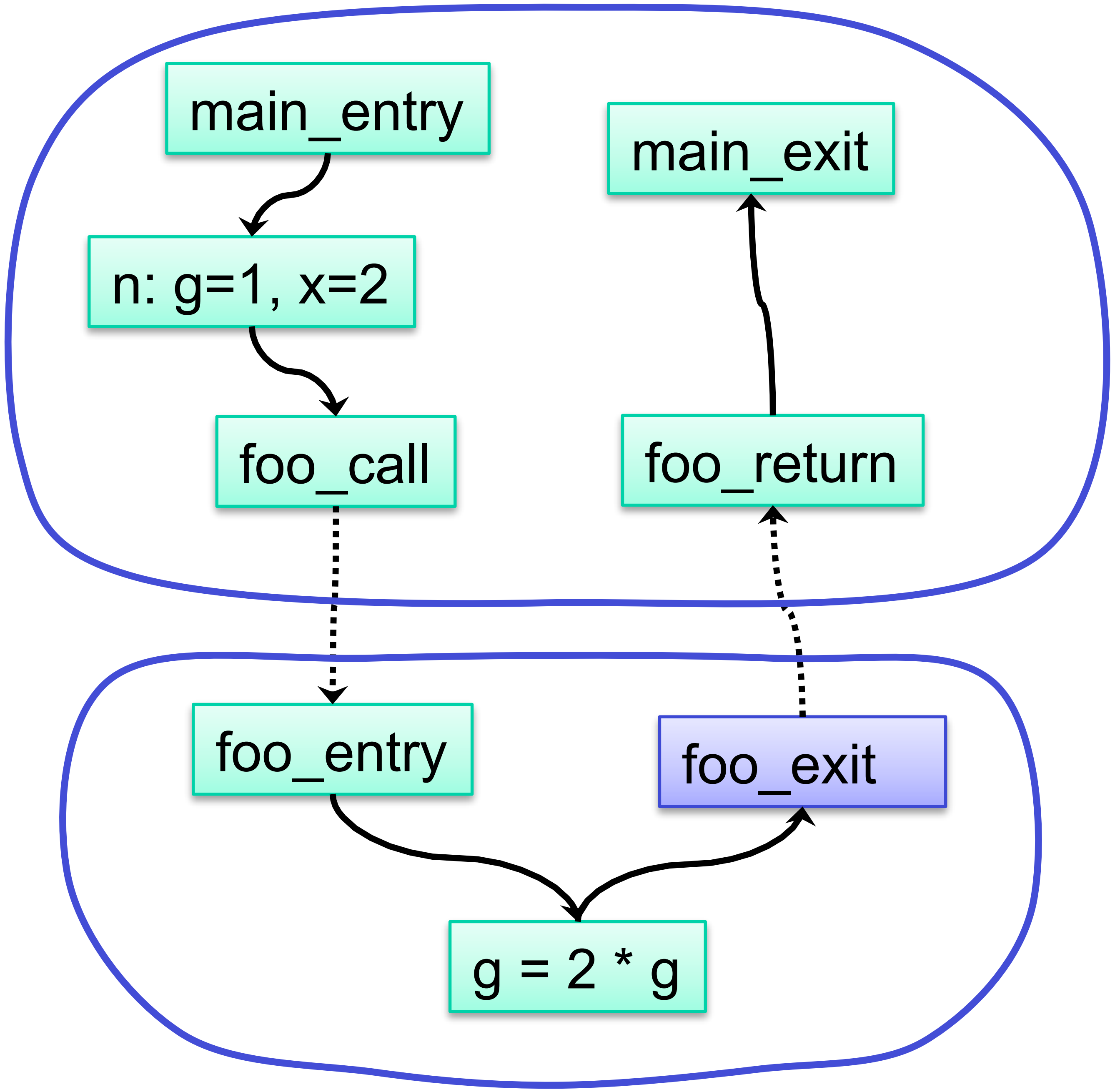
Modeling the Stack

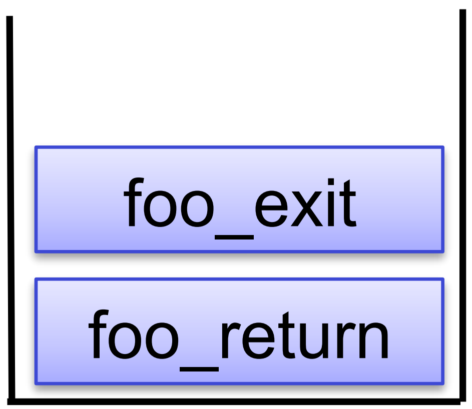


Modeling the Stack

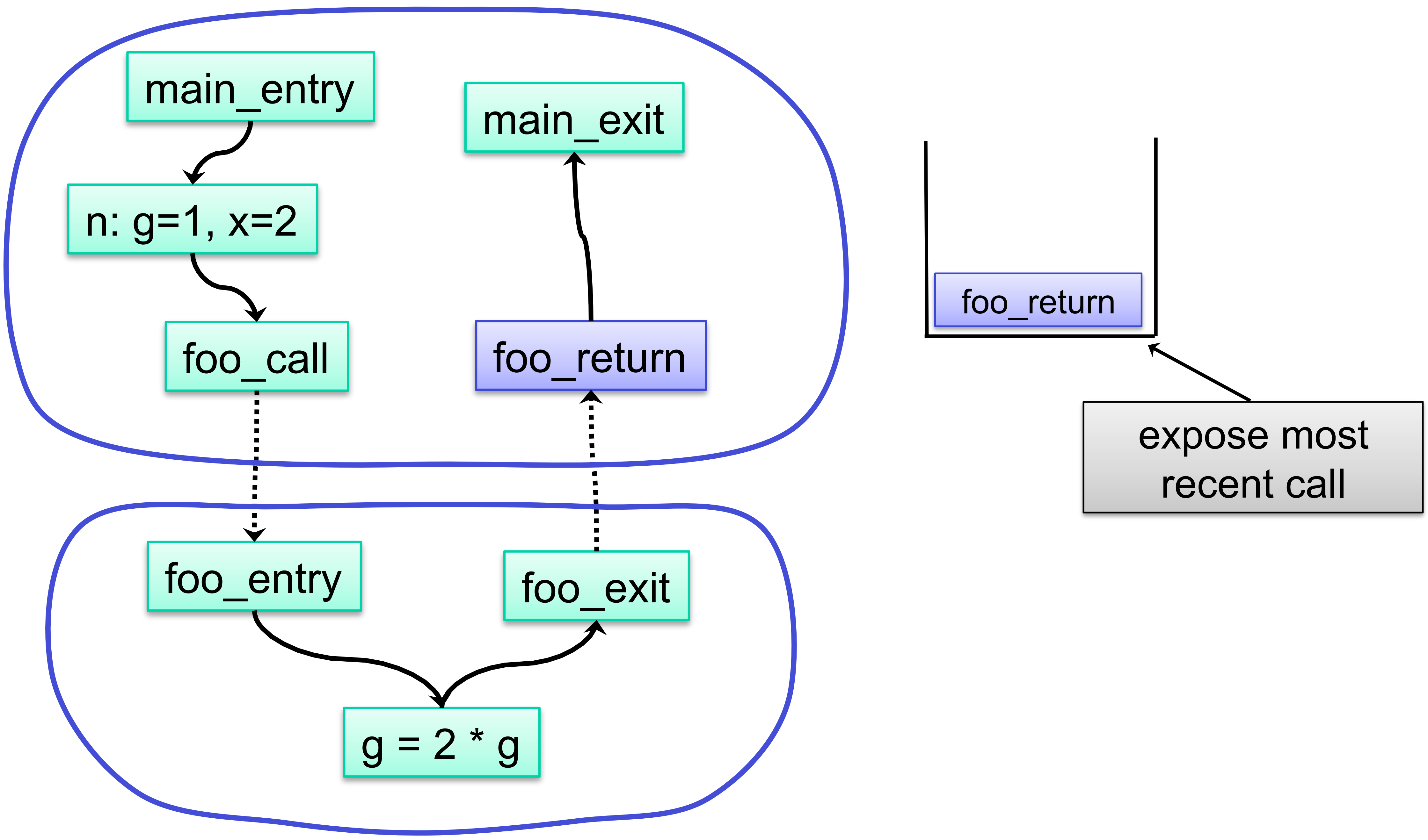


Modeling the Stack

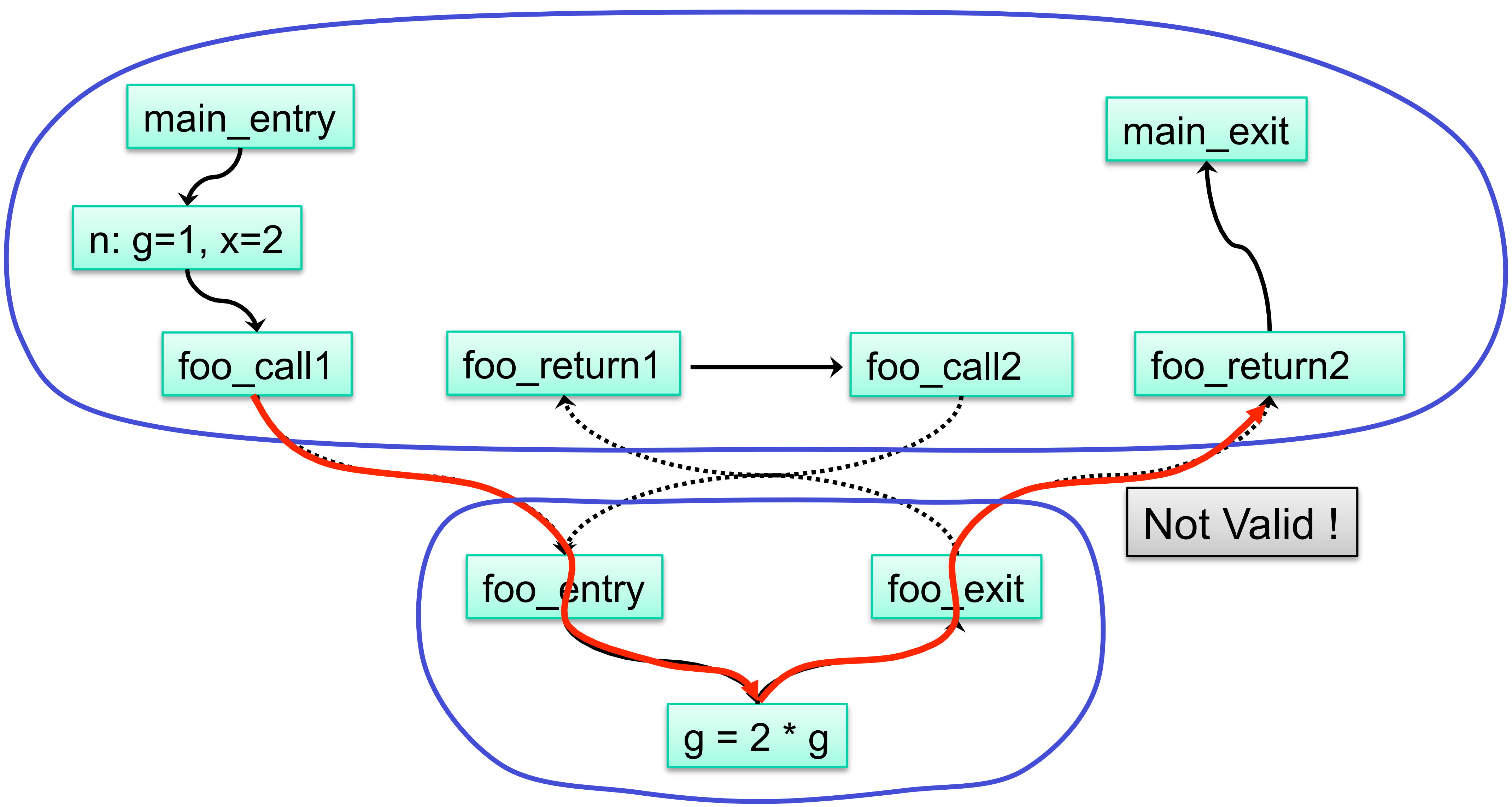




Modeling the Stack



Modeling the Stack



Pushdown Systems

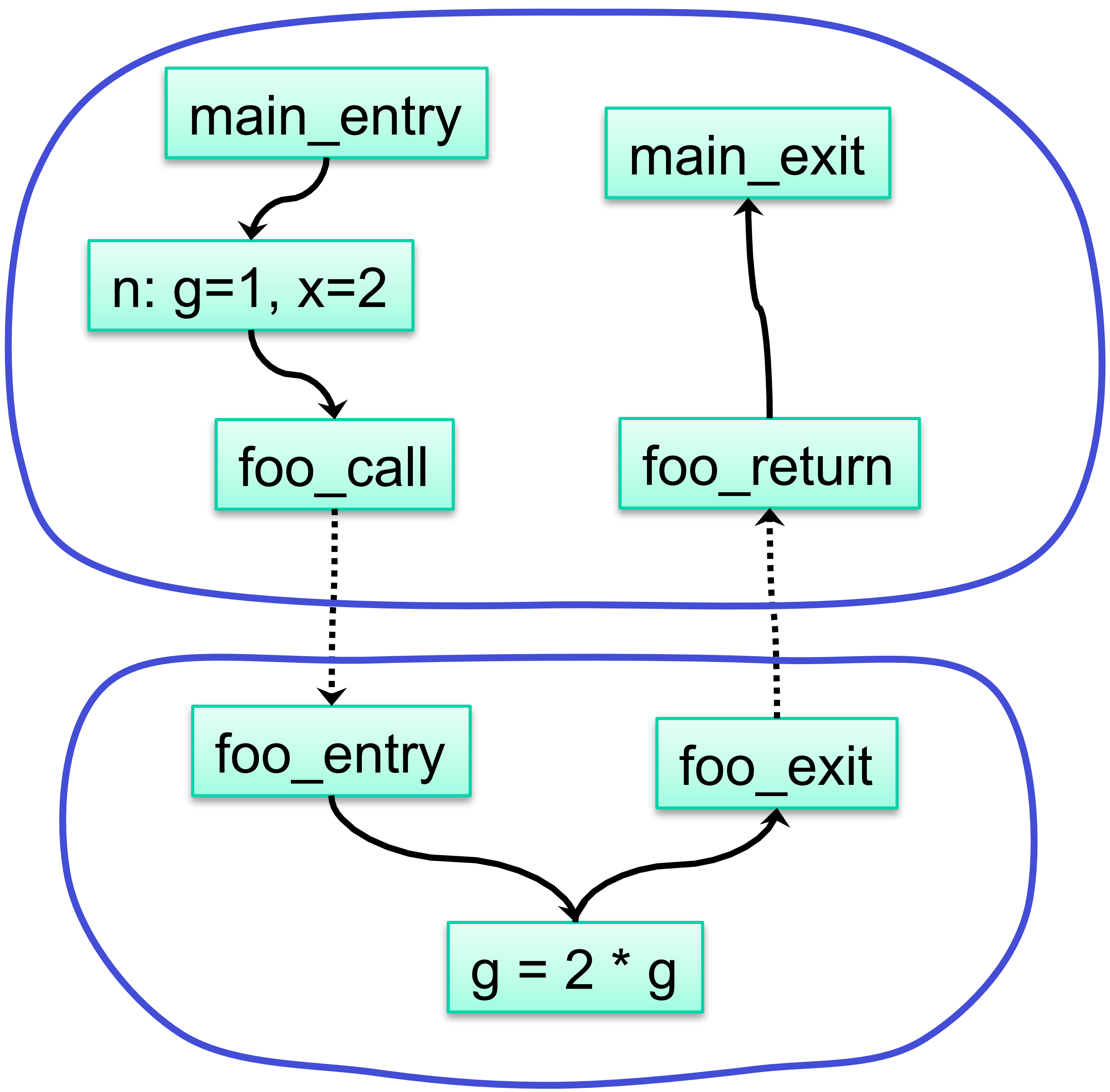
- Finite-state machine with a stack (P, Γ, Δ)
  - $P = \{p,q,\ldots\}$ is a finite set of states
  - $\Gamma = \{y,y',y''\}$ is a finite set of stack symbols
  - Δ: finite set of rules $\subseteq (P \times \Gamma \times P \times \Gamma^*)$
    - if $(p,y,p',u') \in \Delta$,
      then $(p, y \cdot u) \Rightarrow (p', u' \cdot u)$

      for $u \in \Gamma^*$

Pushdown Systems

ICFG	PDS	r∈Δ
Step	Step	(p,y,p',y')
Call	Push	(p,y,p',y',y'')
Return	Pop	(p,y,p')

Pushdown Systems



(p, main_entry, p, n)
(p, foo_call, p, foo_entry, foo_return)
(p, foo_exit, p,)

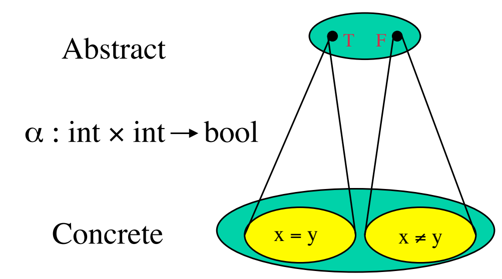
Pushdown Systems

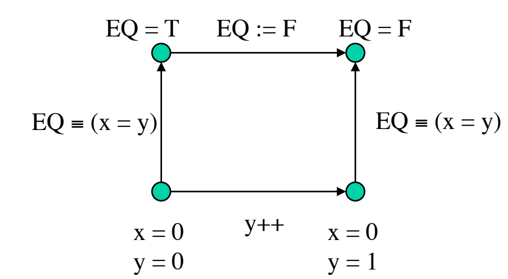
- Finite-state machine with a stack (P, Γ, Δ)
- Pushdown Model Checking algorithms for
  - LTL, CTL, CTL*
  - LTL is polynomial!
- Come to the PL Seminar next semester for more info :-)

SLAM (or now SDV)

[Ball & Rajamani]

- Predicate Abstraction defines a PDS (P,Γ,Δ)
    - Global predicates in P
    - Local predicates and PC in Γ
    - Δ encodes the transition relation
- Check LTL properties on abstract system





SLAM Example

[Ball & Rajamani]

```
      numUnits : int;
      level : int;
      void getUnit() {
[1]     canEnter: bool := F;
[2]     if (numUnits = 0) {
[3]       if (level > 10) {
[4]         NewUnit();
[5]         numUnits := 1;
[6]         canEnter := T;
          }
        } else
[7]         canEnter := T;

[8]     if (canEnter)
[9]       if (numUnits = 0)
[10]        assert(F);
          else
[11]        gotUnit();
      }
```

SLAM Example

```
      numUnits : int;
      level : int;
      void getUnit() {
[1]     canEnter: bool := F;
[2]     if (numUnits = 0) {
[3]       if (level > 10) {
[4]         NewUnit();
[5]         numUnits := 1;
[6]         canEnter := T;
          }
        } else
[7]         canEnter := T;

[8]     if (canEnter)
[9]       if (numUnits = 0)
[10]        assert(F);
          else
[11]        gotUnit();
      }
```

```
      void getUnit() {
[1]     ...
[2]     if (?) {
[3]       if (?) {
[4]         ...
[5]         ...
[6]         ...
          }
        } else
[7]         ...

[8]     if (?)
[9]       if (?)
[10]        ...
          else
[11]        ...
      }
```

SLAM Example

```
       numUnits : int;
       level : int;
       void getUnit() {
[1]      canEnter: bool := F;
[2]      if (numUnits = 0) {
[3]        if (level > 10) {
[4]          NewUnit();
[5]          numUnits := 1;
[6]          canEnter := T;
           }
         } else
[7]          canEnter := T;

[8]      if (canEnter)
[9]        if (numUnits = 0)
[10]         assert(F);
           else
[11]         gotUnit();
       }
```

```
       nU0: bool;
//     nU0 = T iff (numUnits = 0)
       void getUnit() {
[1]      ...
[2]      if (nU0) {
[3]        if (?) {
[4]          ...
[5]          nU0 := F;
[6]          ...
           }
         } else
[7]          ...

[8]      if (?)
[9]        if (nU0)
[10]         ...
           else
[11]         ...
       }
```

SLAM Example

```
     numUnits : int;
     level : int;
     void getUnit() {
[1]    canEnter: bool := F;
[2]    if (numUnits = 0) {
[3]      if (level > 10) {
[4]        NewUnit();
[5]        numUnits := 1;
[6]        canEnter := T;
         }
       } else
[7]        canEnter := T;

[8]    if (canEnter)
[9]      if (numUnits = 0)
[10]       assert(F);
         else
[11]       gotUnit();
     }
```

```
     nU0: bool;

     void getUnit() {
[1]    cE: bool := F;
[2]    if (nU0) {
[3]      if (?) {
[4]        ...
[5]        nU0 := F;
[6]        cE := T;
         }
       } else
[7]        cE := T;

[8]    if (cE)
[9]      if (nU0)
[10]       ...
         else
[11]       ...
     }
```

SLAM Example

```
     numUnits : int;
     level : int;
     void getUnit() {
[1]    canEnter: bool := F;
[2]    if (numUnits = 0) {
[3]      if (level > 10) {
[4]        NewUnit();
[5]        numUnits := 1;
[6]        canEnter := T;
       }
     } else
[7]      canEnter := T;

[8]    if (canEnter)
[9]      if (numUnits = 0)
[10]       assert(F);
       else
[11]       gotUnit();
     }
```

```
     void getUnit() {
[1]    ...
[2]    if (?) {
[3]      if (?) {
[4]        ...
[5]        ...
[6]        ...
         }
     } else
[7]        ...

[8]    if (?)
[9]      if (?)
[10]       ...
         else
[11]       ...
     }
```

```
     nU0: bool;
//   nU0 = T iff (numUnits = 0)
     void getUnit() {
[1]    ...
[2]    if (nU0) {
[3]      if (?) {
[4]        ...
[5]        nU0 := F;
[6]        ...
         }
     } else
[7]        ...

[8]    if (?)
[9]      if (nU0)
[10]       ...
         else
[11]       ...
     }
```

```
     nU0: bool;

     void getUnit() {
[1]    cE: bool := F;
[2]    if (nU0) {
[3]      if (?) {
[4]        ...
[5]        nU0 := F;
[6]        cE := T;
         }
     } else
[7]        cE := T;

[8]    if (cE)
[9]      if (nU0)
[10]       ...
         else
[11]       ...
     }
```

What about Concurrency?

- Easy to combine finite-state systems (cross product)
- Software abstractions can still be infinite-state systems
  - PDSs are finite-state machine *plus a stack!*
  - Two-stacks can simulate a turing machine ...

What about Concurrency?

- Easy to combine finite-state systems (cross product)
- Software abstractions are still infinite-state systems
  - Finite-state machine *plus a stack!*
  - Two-stacks can simulate a turing machine ...
  - Reachability is *undecidable* in general!

What about Concurrency?

- Must bound *global* communication
  - Transfer of global state (think volatile in Java)
  - Happens at context switch (interleaved semantics)
  - Happens at message pass (MPI)