# Using behavioral abstractions for testing and validating classes

Diego Garbervetsky

Summer School in Leicester – 2019

(thanks S. Uchitel for bootstrapping material)

# Outline

- Introduction (Verification and Validation)
- Validation of Contract-based Specifications
- Validation of Executable Specifications (a.k.a code)
- Testing using Specs
- Summary and Conclusions

Enabledness Preserving Abstractions

# Day 1-2

- Building EPAs from code
  - Epas for C, C#/Java
  - Software Model Checkers
  - Limitations
- Examples

# What about code?

APIs, Classes & Protocols Implementations?

# A Stack...

```
typedef struct node{ int data; struct node *next;} node;
typedef struct stack{ int size; node* first;} stack;

stack* s;

int inv() { return s==NULL || s->size >= 0; }
int Stack() {
  s = (stack*) malloc(sizeof(stack));
  if (s == NULL) {
    printf("Memory not available\n"); return 0;
  }
  s->size = 0; s->first = NULL;
  return 1;
}

int pre_push() { return s!=NULL; }
int push(int e) {
  node *tmp = s->first;
  while (tmp->next != s->first)
    tmp = tmp->next;
  tmp->next = (node*) malloc(sizeof(node));
  if (tmp->next == NULL) {
    printf("Memory not available\n");
    s = NULL; return 0;
  }
  tmp->next->data = e;
  tmp->next->next = s->first;
  s->size++;
  return 1;
}

int pre_pop() { return s!=NULL && s->size > 0; }
int pop() {
  node* new_first = s->first->next;
  int ret = s->first->data;
  free(s->first);
  s->first = new_first;
  return ret;
}
```

# Testing, verification, validation...

```
typedef struct node{ int data; struct node *next;} node;
typedef struct stack{ int size; node* first;} stack;

stack* s;

int inv() { return s==NULL || s->size >= 0; }
int Stack() {
  s = (stack*) malloc(sizeof(stack));
  if (s == NULL) {
    printf("Memory not available\n"); return 0;
  }
  s->size = 0; s->first = NULL;
  return 1;
}

int pre_push() { return s!=NULL; }
int push(int e) {
  node *tmp = s->first;
  while (tmp->next != s->first)
    tmp = tmp->next;
  tmp->next = (node*) malloc(sizeof(node));
  if (tmp->next == NULL) {
    printf("Memory not available\n");
    s = NULL; return 0;
  }
  tmp->next->data = e;
  tmp->next->next = s->first;
  s->size++;
  return 1;
}

int pre_pop() { return s!=NULL && s->size > 0; }
int pop() {
  node* new_first = s->first->next;
  int ret = s->first->data;
  free(s->first);
  s->first = new_first;
  return ret;
}
```

To test this program...

- We need an oracle

To formally verify it...

- We need a formal specification

*How do we know if the oracle/spec corresponds to what we wanted to build?*

- This is a *validation* problem (again)

Let's try a brief code inspection...

# Code Inspection

```
typedef struct node{ int data; struct node *next;} node;
typedef struct stack{ int size; node* first;} stack;

stack* s;

int inv() { return s==NULL || s->size >= 0; }
int Stack() {
  s = (stack*) malloc(sizeof(stack));
  if (s == NULL) {
    printf("Memory not available\n"); return 0;
  }
  s->size = 0; s->first = NULL;
  return 1;
}

int pre_push() { return s!=NULL; }
int push(int e) {
  node *tmp = s->first;
  while (tmp->next != s->first)
    tmp = tmp->next;
  tmp->next = (node*) malloc(sizeof(node));
  if (tmp->next == NULL) {
    printf("Memory not available\n");
    s = NULL; return 0;
  }
  tmp->next->data = e;
  tmp->next->next = s->first;
  s->size++;
  return 1;
}

int pre_pop() { return s!=NULL && s->size > 0; }
int pop() {
  node* new_first = s->first->next;
  int ret = s->first->data;
  free(s->first);
  s->first = new_first;
  return ret;
}
```

→ Data structures (stack, node)

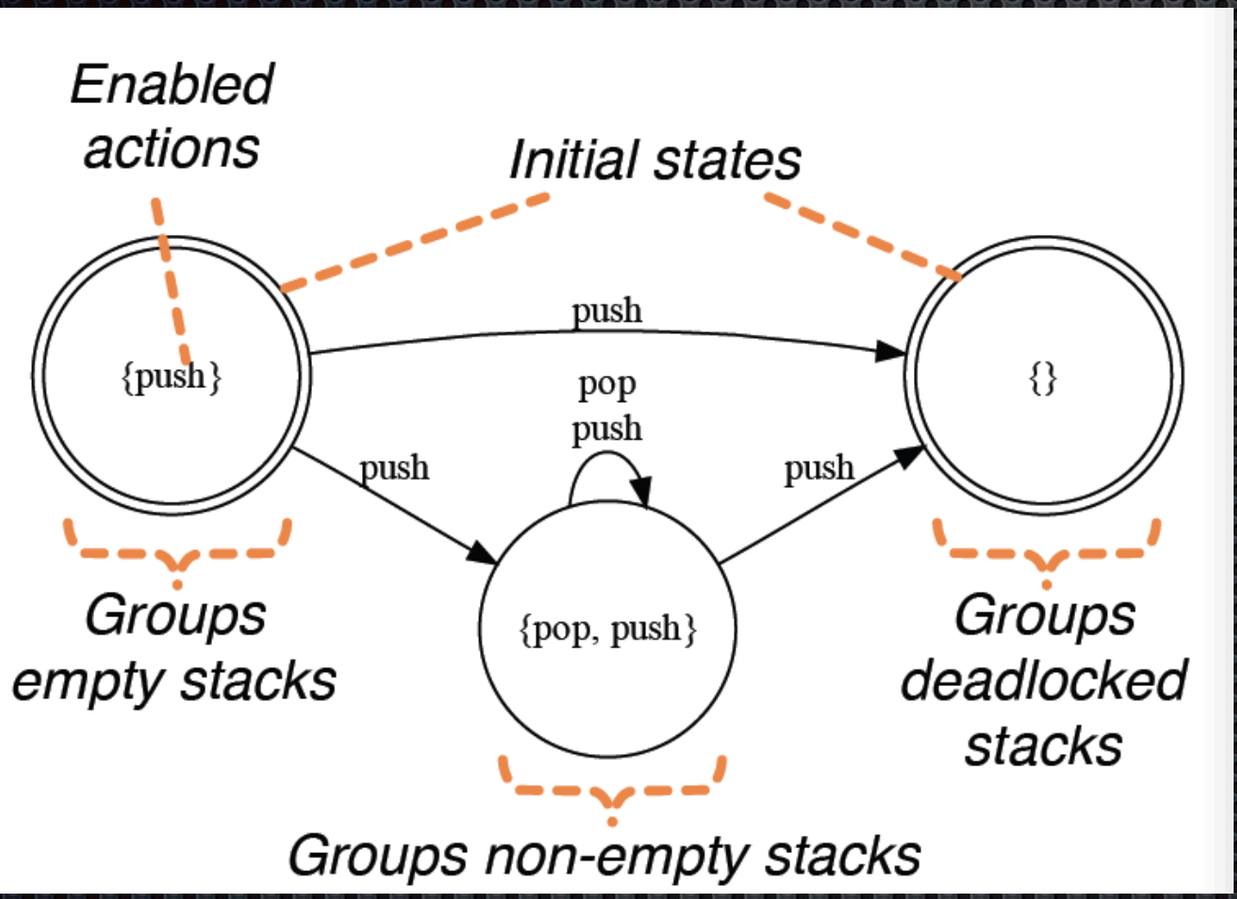
→ "Class" invariant

→ Initialization operation

→ Push operation (with requires clause)

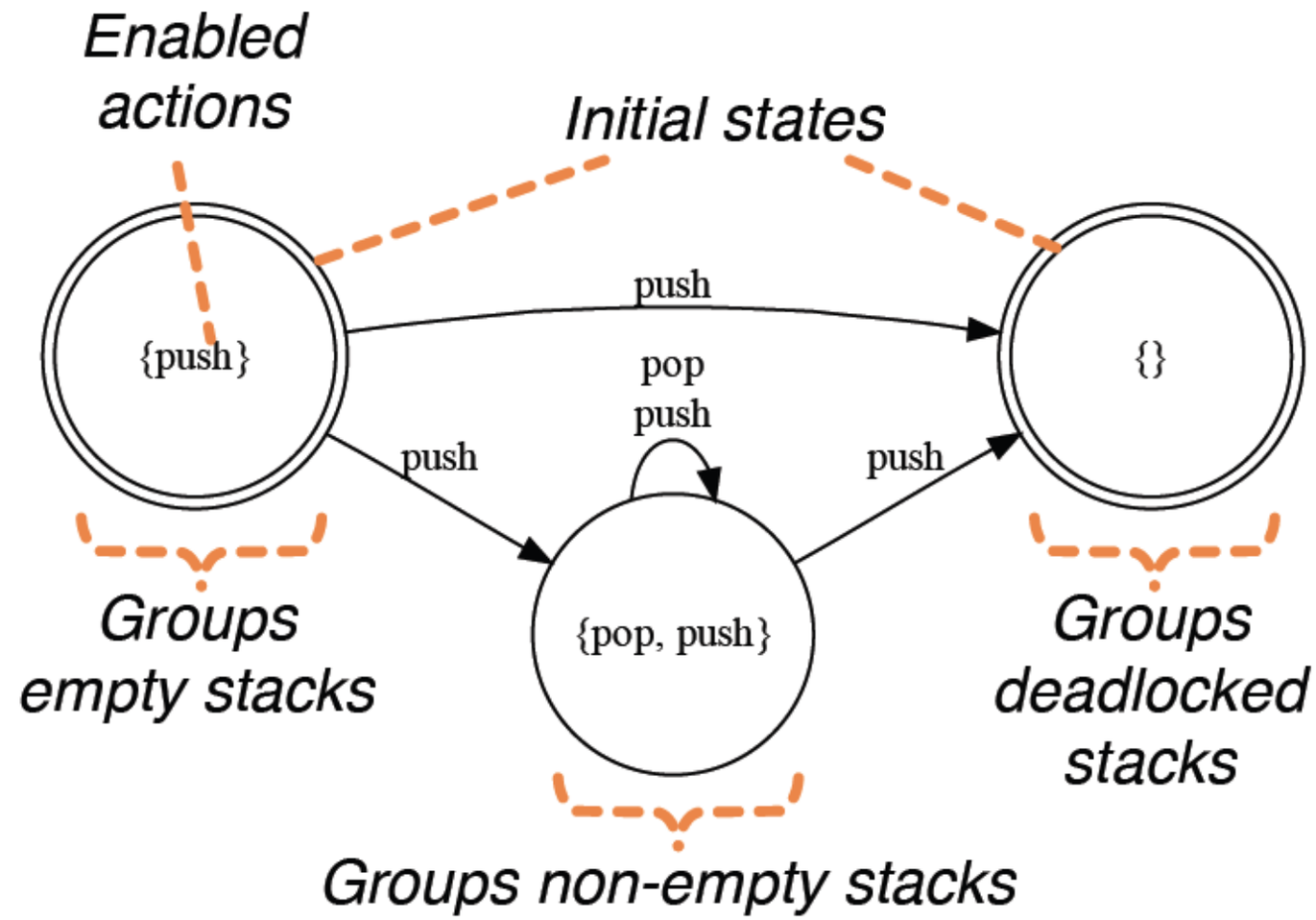
→ Pop operation (with requires clause)

# Enabledness Preserving Abstraction




```
typedef struct node{ int data; struct node *next;} node;
typedef struct stack{ int size; node* first;} stack;

stack* s;

int inv() { return s==NULL || s->size >= 0; }
int Stack() {
  s = (stack*) malloc(sizeof(stack));
  if (s == NULL) {
    printf("Memory not available\n"); return 0;
  }
  s->size = 0; s->first = NULL;
  return 1;
}

int pre_push() { return s!=NULL; }
int push(int e) {
  node *tmp = s->first;
  while (tmp->next != s->first)
    tmp = tmp->next;
  tmp->next = (node*) malloc(sizeof(node));
  if (tmp->next == NULL) {
    printf("Memory not available\n");
    s = NULL; return 0;
  }
  tmp->next->data = e;
  tmp->next->next = s->first;
  s->size++;
  return 1;
}

int pre_pop() { return s!=NULL && s->size > 0; }
int pop() {
  node* new_first = s->first->next;
  int ret = s->first->data;
  free(s->first);
  s->first = new_first;
  return ret;
}
```

# An Enabledness Preserving Abstraction of the Stack's Semantics



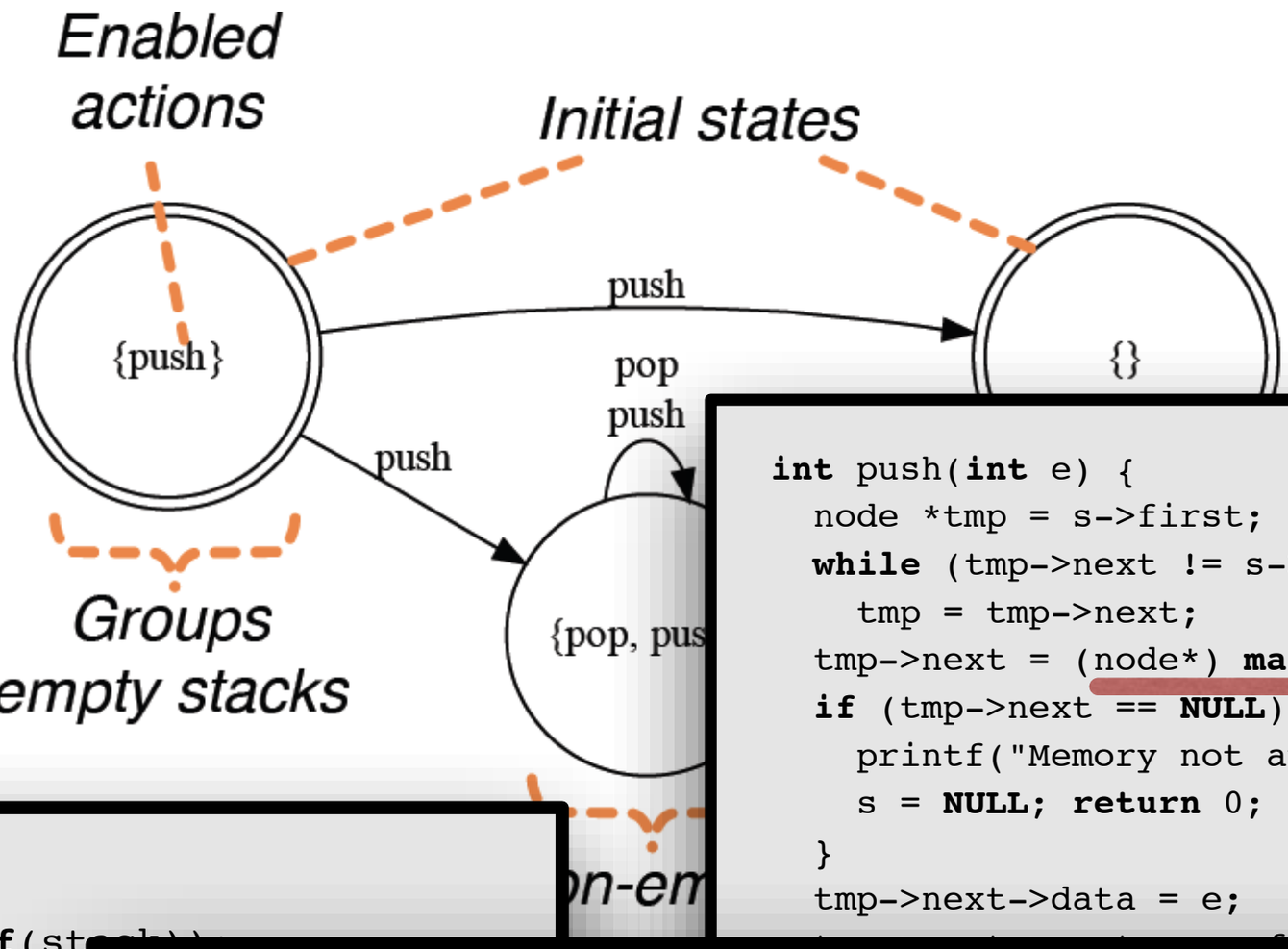

Why do we have two initial states?

Why is there a deadlock state?

State Abstraction + Parameter Abstraction

Stack uses **dynamic memory management**

# An Enabledness Preserving Abstraction of the Stack's Semantics

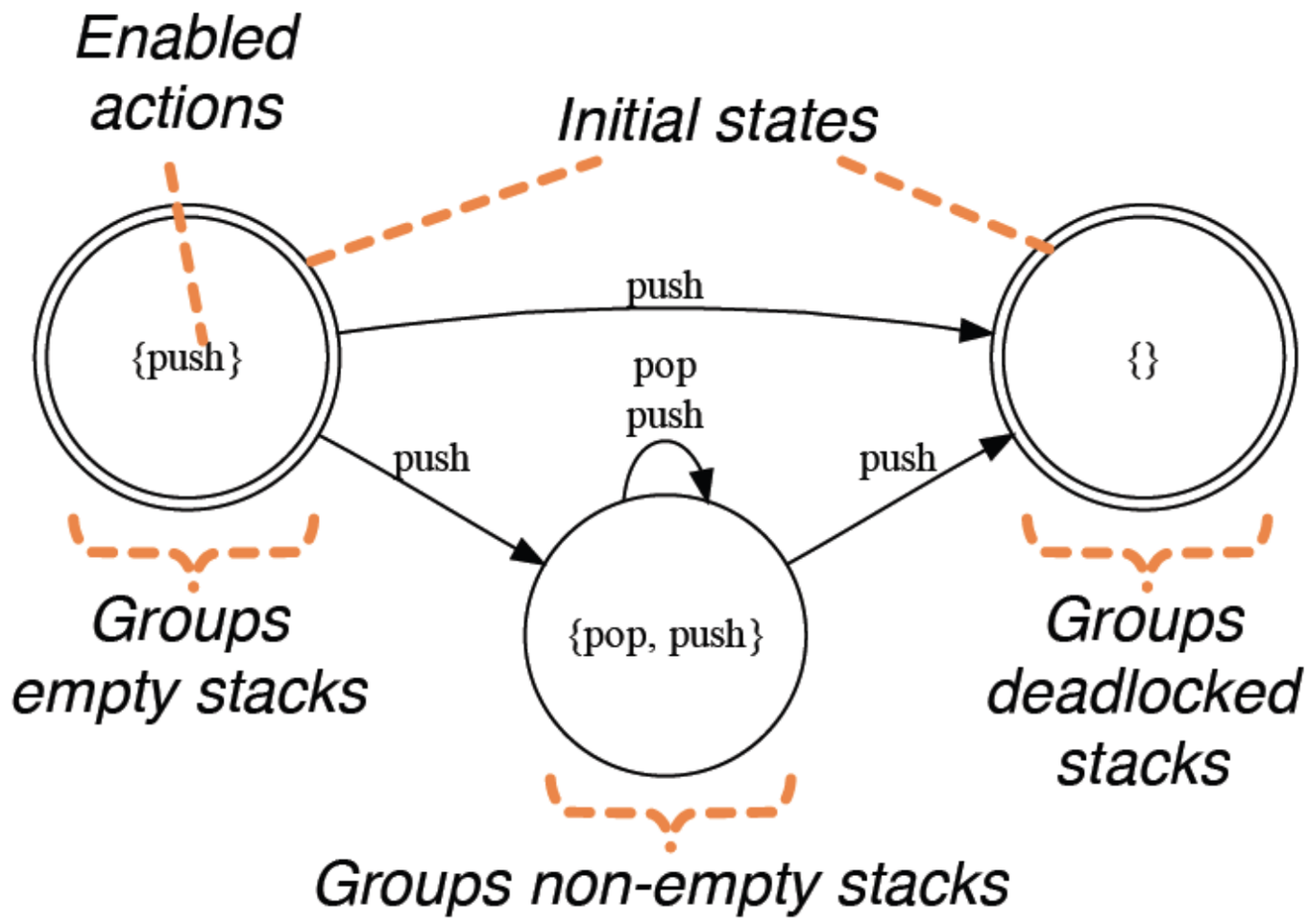



```
int push(int e) {
  node *tmp = s->first;
  while (tmp->next != s->first)
    tmp = tmp->next;
  tmp->next = (node*) malloc(sizeof(node));
  if (tmp->next == NULL) {
    printf("Memory not available\n");
    s = NULL; return 0;
  }
  tmp->next->data = e;
```

```
int Stack() {
  s = (stack*) malloc(sizeof(st
  if (s == NULL) {
    printf("Memory not availabl
  }
  s->size = 0; s->first = NULL;
  return 1;
}
```

```
int pre_push() { return s!=NULL; }
//...

int pre_pop() { return s!=NULL && s->size > 0; }
//...
```

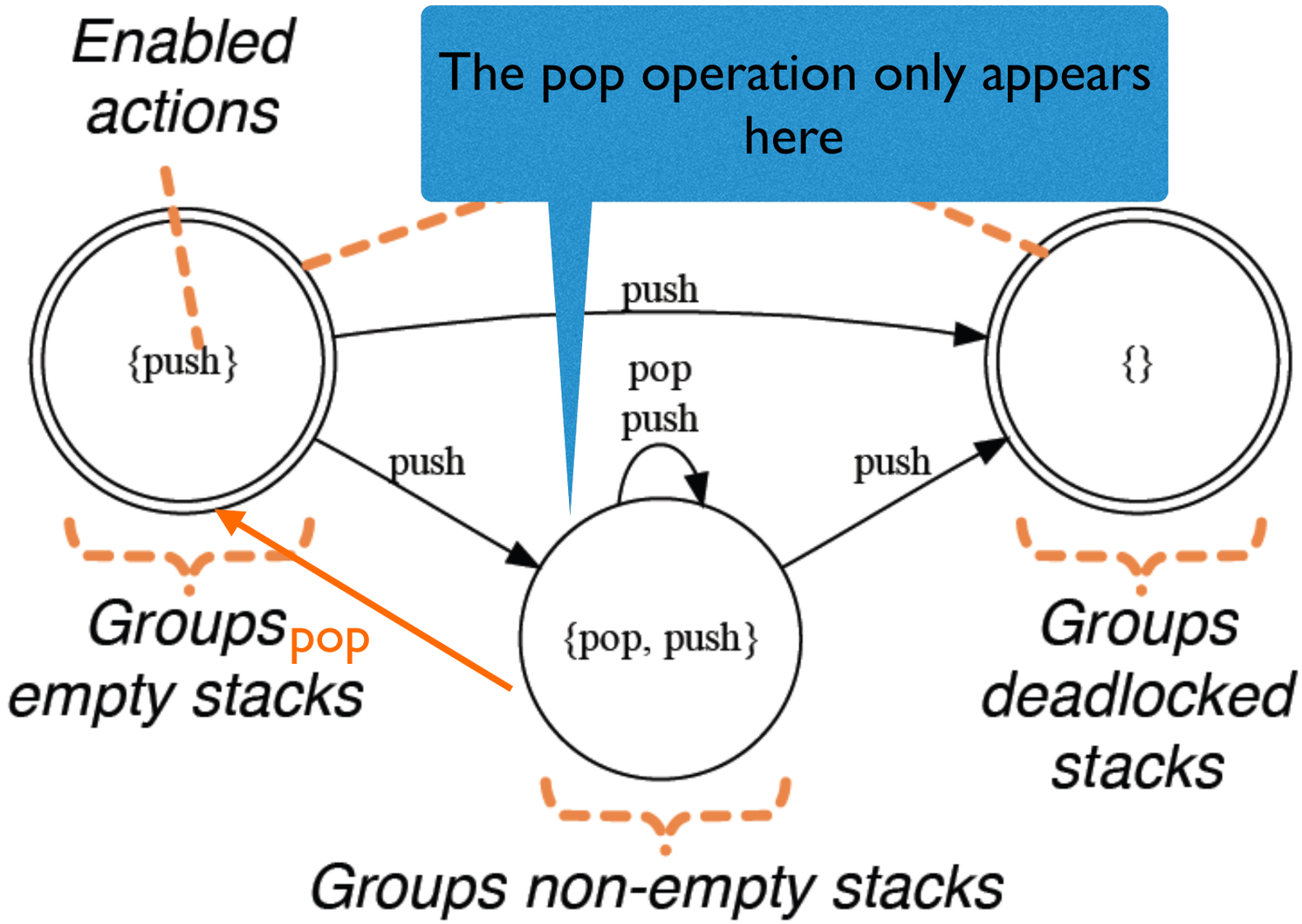
# An Enabledness Preserving Abstraction of the Stack's Semantics




Why isn't there a "full stack state"?

Because this is an infinite data structure

# An Enabledness Preserving Abstraction of the Stack's Semantics




Why can't I pop back to the initial state?

Something is wrong

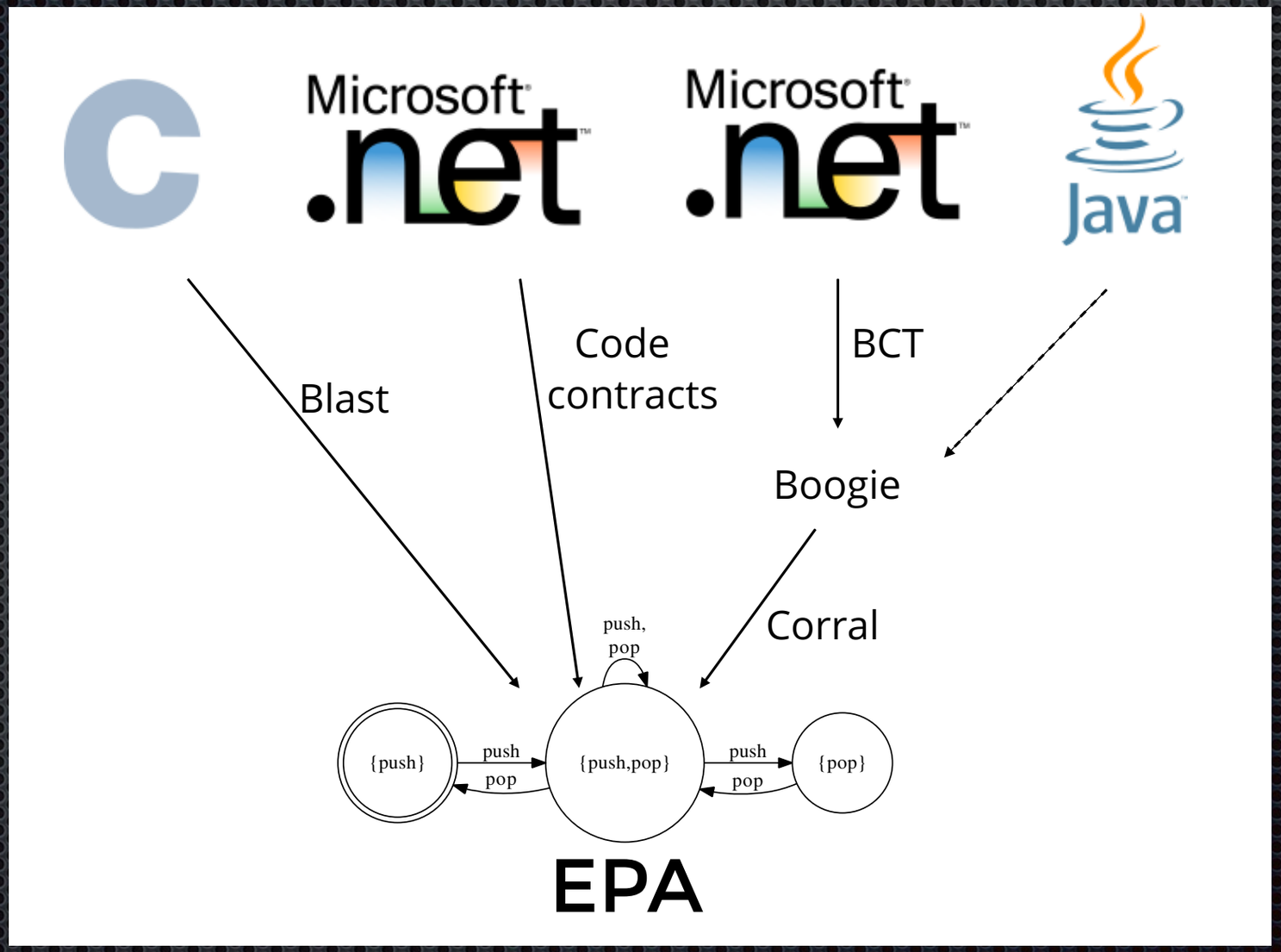
# An Enabledness Preserving Abstraction of the Stack's Semantics

```
typedef struct stack{ int size; node* first;} stack;

/*...*/

int pop() {
  node* new_first = s->first->next;
  int ret = s->first->data;
  free(s->first);
  s->first = new_first;
  return ret;
}
```

The pop operation does not update the size field...

And this is a **bug**!!

# Building EPAs for code

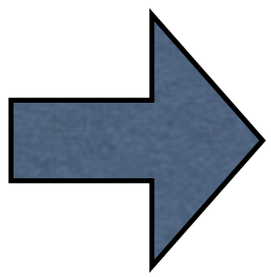


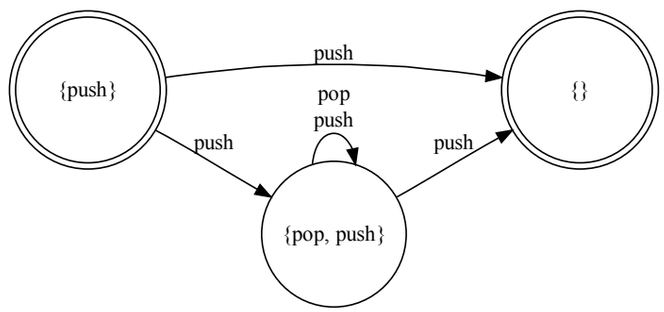
# Enabledness Preserving Abstractions (EPAs)

**Automatically and statically** construct EPAs

```
typedef struct node{ int data; struct node *next;} node;
typedef struct stack{ int size; node* first;} stack;

stack* s;

int inv() { return s==NULL || s->size >= 0; }
int Stack() {
  s = (stack*) malloc(sizeof(stack));
  if (s == NULL) {
    printf("Memory not available\n"); return 0;
  }
  s->size = 0; s->first = NULL;
  return 1;
}

int pre_push() { return s!=NULL; }
int push(int e) {
  node *tmp = s->first;
  while (tmp->next != s->first)
    tmp = tmp->next;
  tmp->next = (node*) malloc(sizeof(node));
  if (tmp->next == NULL) {
    printf("Memory not available\n");
    s = NULL; return 0;
  }
  tmp->next->data = e;
  tmp->next->next = s->first;
  s->size++;
  return 1;
}

int pre_pop() { return s!=NULL && s->size > 0; }
int pop() {
  node* new_first = s->first->next;
  int ret = s->first->data;
  free(s->first);
  s->first = new_first;
  return ret;
}
```

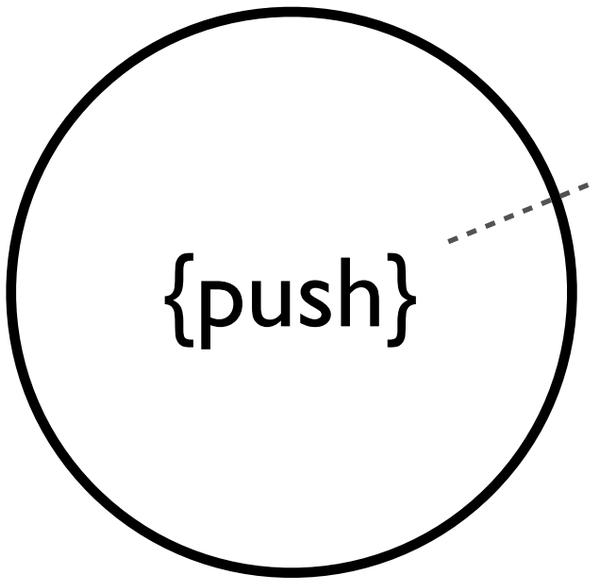
Source code
with
requires clauses






EPA

# State Invariants



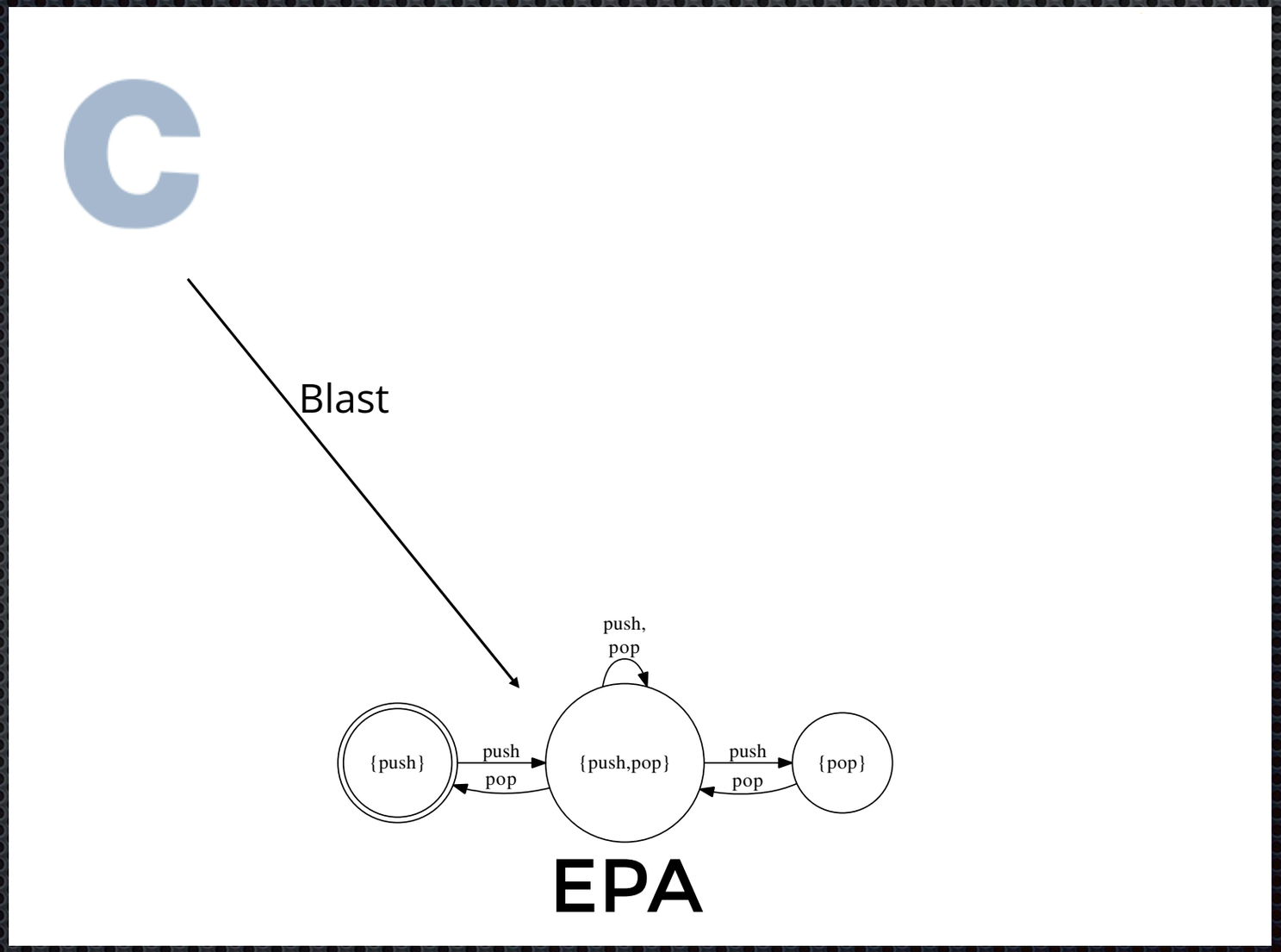

$$inv_{\{push\}}(s) \Leftrightarrow inv(s) \wedge R_{push}(s) \wedge \neg R_{pop}(s)$$

Now, preconditions and invariants can be code…

```
bool inv_push(stack* s) {
   return (s == NULL || s->size >= 0) &&
          (s != NULL) &&
          !(s != NULL && s->size > 0)
}
```

**We need something more powerful than a SMT solver…**

# Building EPAs using Blast



# Naïve Construction Algorithm

1. Construct the $2^{|\text{actions}|}$ abstract states
2. For each pair of abstract states S1 and S2
   - For each action a ∈ S1
     - If there exists a concrete instance s in S1 and concrete parameters for a that make s evolve into s' in S2
       - Add a transition $S1 \xrightarrow{a} S2$

Key step

SMT solver:

$$\exists c \,.\, Pred_{S1}(c) \wedge \exists p \,.\, R_a(c,p) \wedge Pred_{S2}(a(c,p))$$

# Solving the key step

The transition {push,pop} $\xrightarrow{\text{pop}}$ {push} needs to be added if:

there is a concrete instance s in {push,pop} such that pop makes s evolve into s' in {push}

```
void pushPop_to_push_via_pop(stack* s) {
  if (pred_pushPop(s)) {
    pop(s);
    if (pred_push(s))
      TARGET;
  }
}
```

If the TARGET statement is reachable by an execution...

# Code Reachability Queries

In these cases we add the transition

```
void pushPop_to_push_via_pop(stack* s) {
  if (pred_pushPop(s)) {
    pop(s);
    if (pred_push(s))
      TARGET;
  }
}
```

Program

Statement

TARGET

Software model checker

Reachable

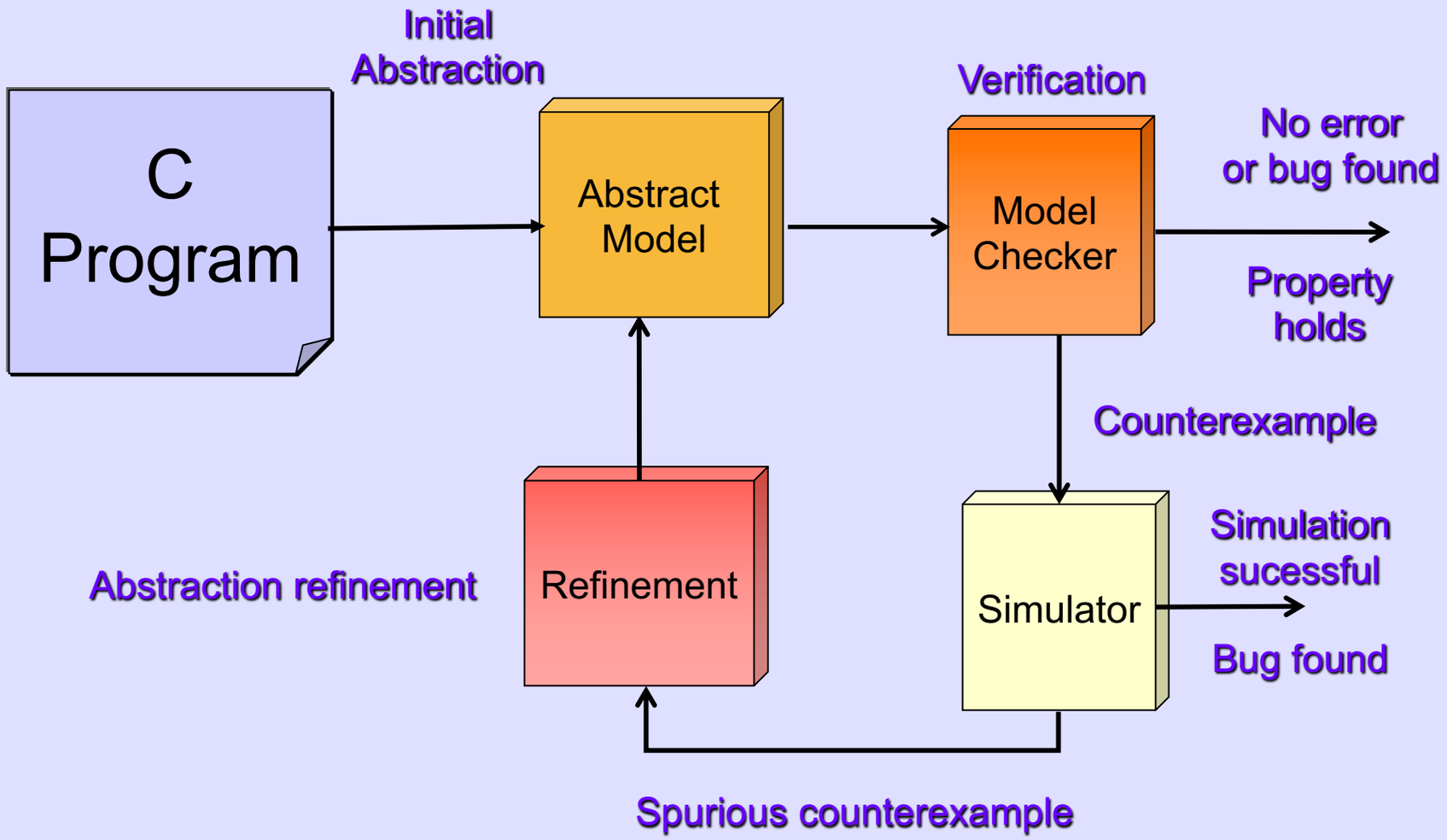
Unreachable

Unknown

# Software Model Checker Blast

## CEGAR

CounterExample-Guided Abstraction Refinement

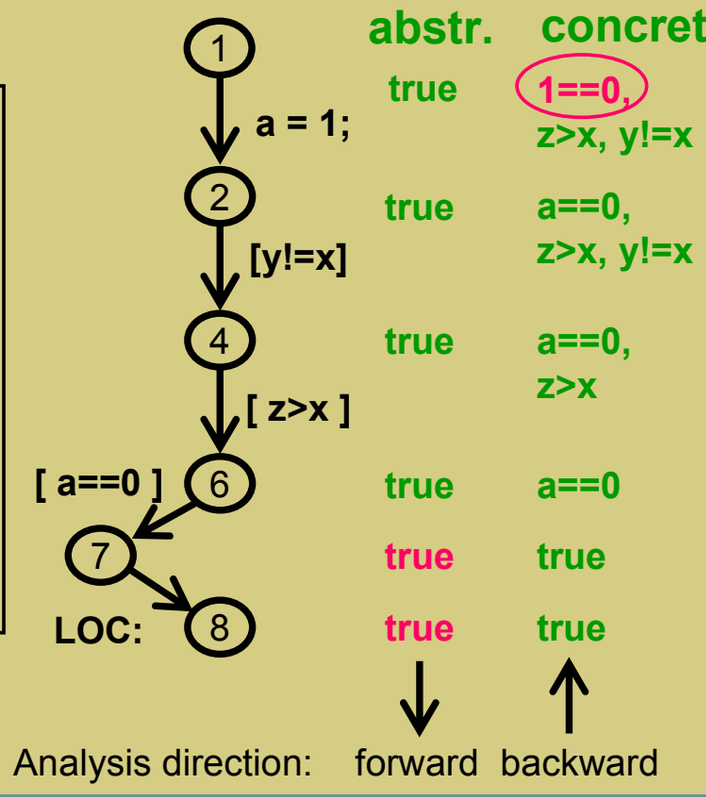


# Blast over simplified

- Determines whether target location is reachable
- CEGAR: Starts from control flow graph, refine predicate based on spurious counter examples

Example:

```
    int
    f (int x, int y, int z){
1:    int a = 1;
2:    if (y == x)
3:      y++;
4:    if (z <= x)
5:      y++;
6:    if (a == 0)
7:      LOC: // abort();
8:    return a;
    }
```



# Code Reachability Queries

In these cases we add the transition

```
void pushPop_to_push_via_pop(stack* s) {
  if (inv_pushPop(s)) {
    pop(s);
    if (inv_push(s))
      TARGET;
  }
}
```

Program

Statement

TARGET

Software model checker

Reachable

**Unreachable**

Unknown

Blast over approximates the state space
If Blast determined the target is unreachable we can be sure the transition is not possible

# Computing Transitions with Blast

Reachability queries

```
void pushPop_to_push_via_pop(stack* s) {
  if (pred_pushPop(s)) {
    pop(s);
    if (pred_push(s))
      TARGET;
  }
}
```

# A Detail....

How do we check the target is unreachable

```
void pushPop_to_push_via_pop(stack* s) {
  if (pred_pushPop(s)) {
     pop(s);
     if (pred_push(s))
        TARGET;
  }
}
```

$inv(s) \wedge (\exists e)(R_{push}(s,e)) \wedge \neg(\exists e)\ R_{pop}(s)$

How do you check that forall **e** push is not enabled?

# Dealing with Parameters

$$\text{inv}_{\{\text{push}\}}(s) \Leftrightarrow \text{inv}(s) \wedge (\exists e)(\text{Inv}_{\text{push}}(s, e)) \wedge \neg \text{Inv}_{\text{pop}}(s)$$

How can we compute this state predicate?

One possibility is to assume that the requires clause can be rewritten as:

$$R_{\text{push}}(s, elem) \Leftrightarrow \text{SR}_{\text{push}}(s) \ \wedge \text{PR}_{\text{push}}(elem)$$

Such that the following holds:

$$\exists elem \ \text{PR}_{\text{push}}(elem)$$

# Possible Solution: Split Requires Clauses

$$R_{\text{push}}(s, elem) \stackrel{\text{def}}{=} s \neq \text{NULL} \;\&\&\; elem > 0$$

$$\text{SR}_{\text{push}}(s) = s \neq \text{NULL} \qquad \text{PR}_{\text{push}}(elem) = elem > 0$$

With this splitting, we can now rewrite the state invariant

$$\text{inv}_{\{\text{push}\}}(s) \Leftrightarrow \text{inv}(s) \wedge (\exists e)(R_{\text{push}}(s,e)) \wedge \neg R_{\text{pop}}(s)$$

$$(\exists e)(R_{\text{push}}(s,e)) \rightarrow \text{SR}_{\text{push}}(s)$$

# Possible Solution: Split Requires Clauses

It is not always possible to do so:

$$R_{\text{push}}(s, elem) \stackrel{\text{def}}{=} s \neq \text{NULL} \;\&\&\; elem > s.\text{size}$$

$$\text{SR}_{\text{push}}(s) = ? \qquad \text{PR}_{\text{push}}(elem) = ?$$

# Splitting Requires Clauses

We can ask a weaker condition

The requires clause needs to be approximated:

$$R^{\uparrow}_{\text{push}}(c, elem) \Rightarrow R_{\text{push}}(c, elem)$$
$$R_{\text{push}}(c, elem) \Rightarrow R^{\downarrow}_{\text{push}}(c, elem)$$

And each approximation is then splitted:

$$R^{\uparrow}_{\text{push}}(c, elem) = \text{SR}^{\uparrow}_{\text{push}}(c) \ \wedge \text{PR}_{\text{push}}(elem)$$
$$R^{\downarrow}_{\text{push}}(c, elem) = \text{SR}^{\downarrow}_{\text{push}}(c) \ \wedge \text{PR}_{\text{push}}(elem)$$

# Splitting Requires Clauses

$$R^{\uparrow}_{\text{push}}(c, elem) \Rightarrow R_{\text{push}}(c, elem) \qquad R^{\uparrow}_{\text{push}}(c, elem) = \text{SR}^{\uparrow}_{\text{push}}(c) \ \wedge \text{PR}_{\text{push}}(elem)$$

$$R_{\text{push}}(c, elem) \Rightarrow R^{\downarrow}_{\text{push}}(c, elem) \qquad R^{\downarrow}_{\text{push}}(c, elem) = \text{SR}^{\downarrow}_{\text{push}}(c) \ \wedge \text{PR}_{\text{push}}(elem)$$

$$R_{\text{push}}(s, elem) \stackrel{\text{def}}{=} s \neq \text{NULL} \ \wedge \ elem > s.\text{size}$$

$$\text{SR}^{\uparrow}_{\text{push}}(s) = s \neq \text{NULL} \wedge s.\text{size} = 0$$

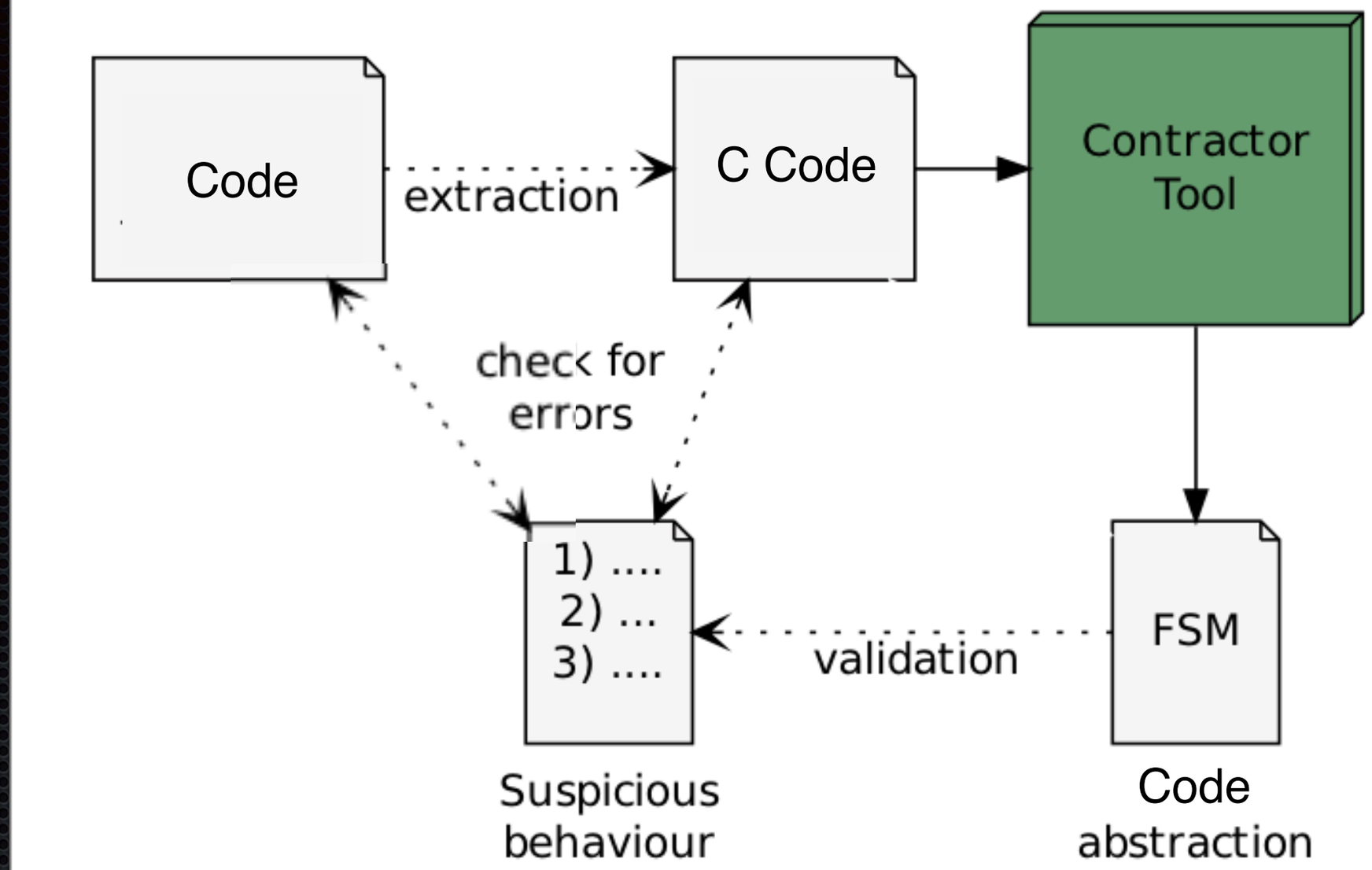
$$\text{PR}_{\text{push}}(elem) = elem > 0$$

$$\text{SR}^{\downarrow}_{\text{push}}(s) = s \neq \text{NULL}$$

Produces an even more over-approximated abstraction

# Basic Questions

- Are the abstractions useful for finding problems?
- Does the technique scale?




# Experimental Setup

(For code)

# Experimental Evaluation: Subjects

- Signature
- PipedOutputStream
- ListItr

→ JDK 1.4 implementation

- Socket
- Ristretto SMTP client
- JES SMTP server

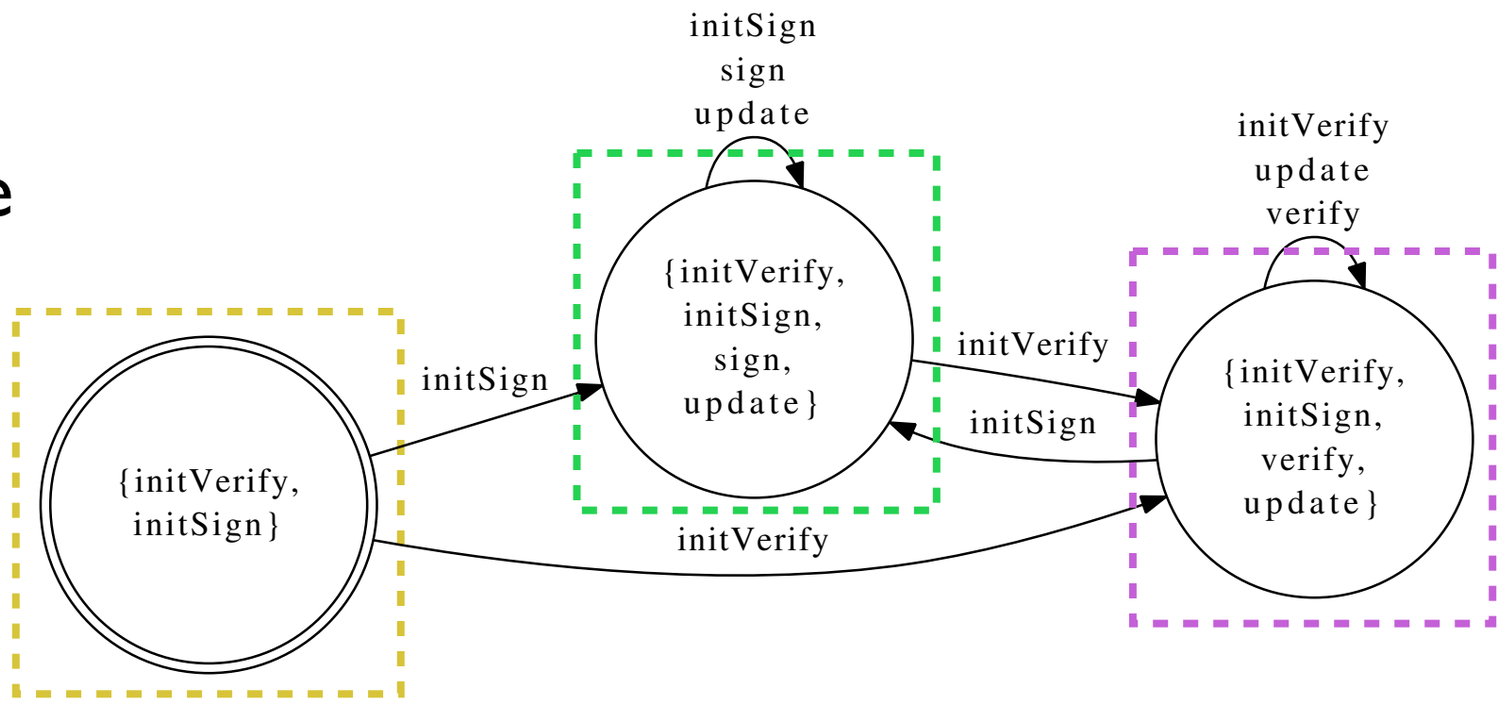
→ Open source mail client/server

- PCCRR protocol

→ SpecExplorer protocol model (C# code)

# Experimental Evaluation: Findings

## Java Signature

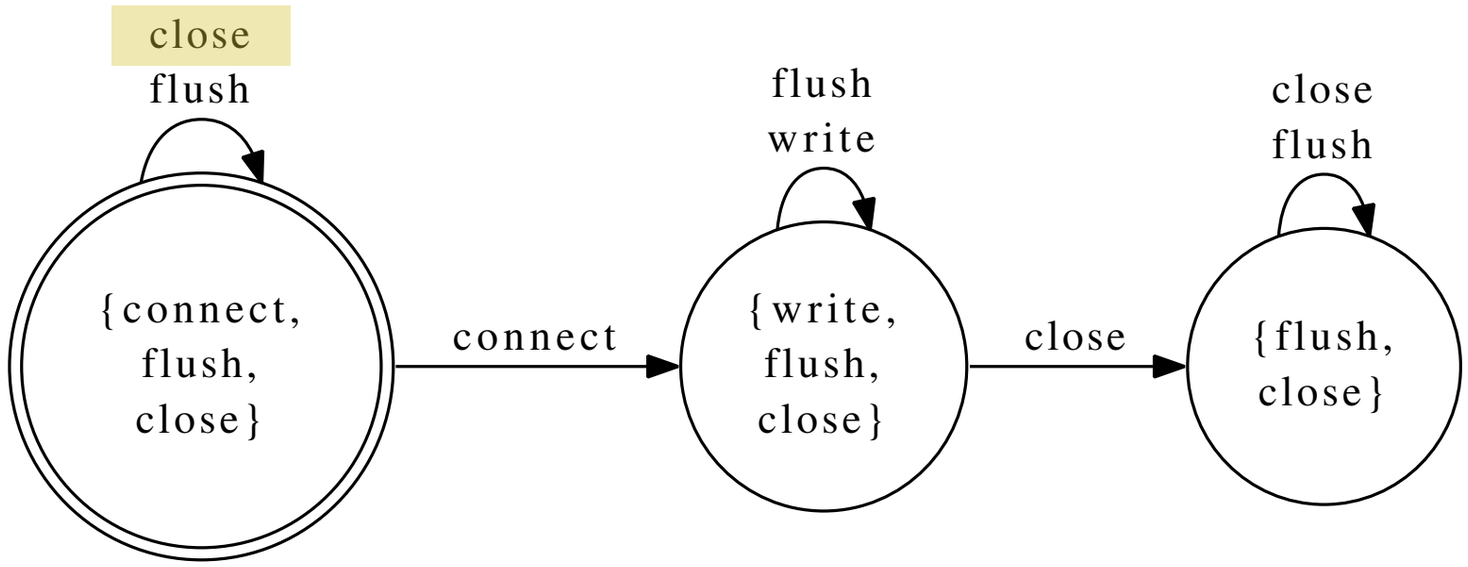



## Java official documentation:

| Field Summary | |
|---|---|
| protected static int | **SIGN**<br>Possible state value, signifying that this signature object has been initialized for signing. |
| protected int | **state**<br>Current state of this signature object. |
| protected static int | **UNINITIALIZED**<br>Possible state value, signifying that this signature object has not yet been initialized. |
| protected static int | **VERIFY**<br>Possible state value, signifying that this signature object has been initialized for verification. |

# Experimental Evaluation: Findings

## Java PipedOutputStream




## Java official documentation:

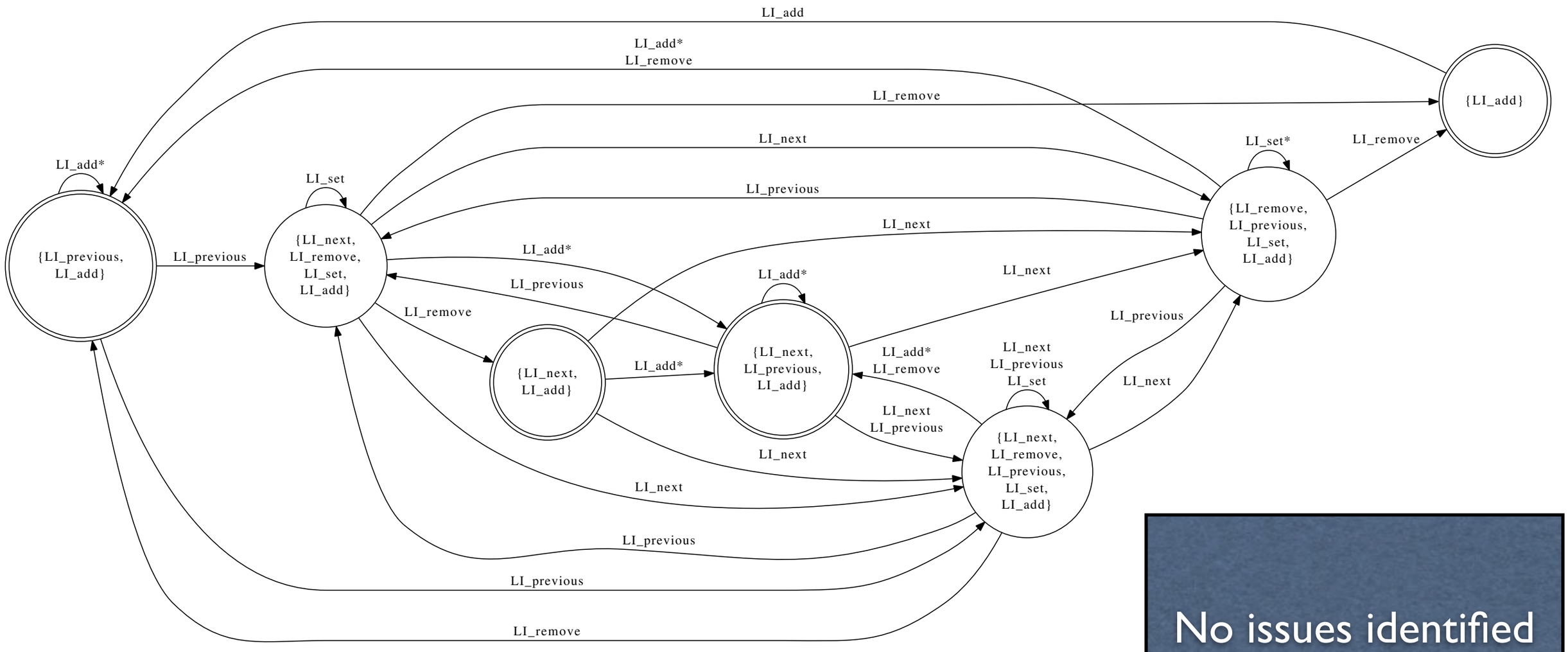
**close**

```
public void close()
            throws IOException
```

Closes this piped output stream and releases any system resources associated with this stream. This stream may no longer be used for writing bytes.

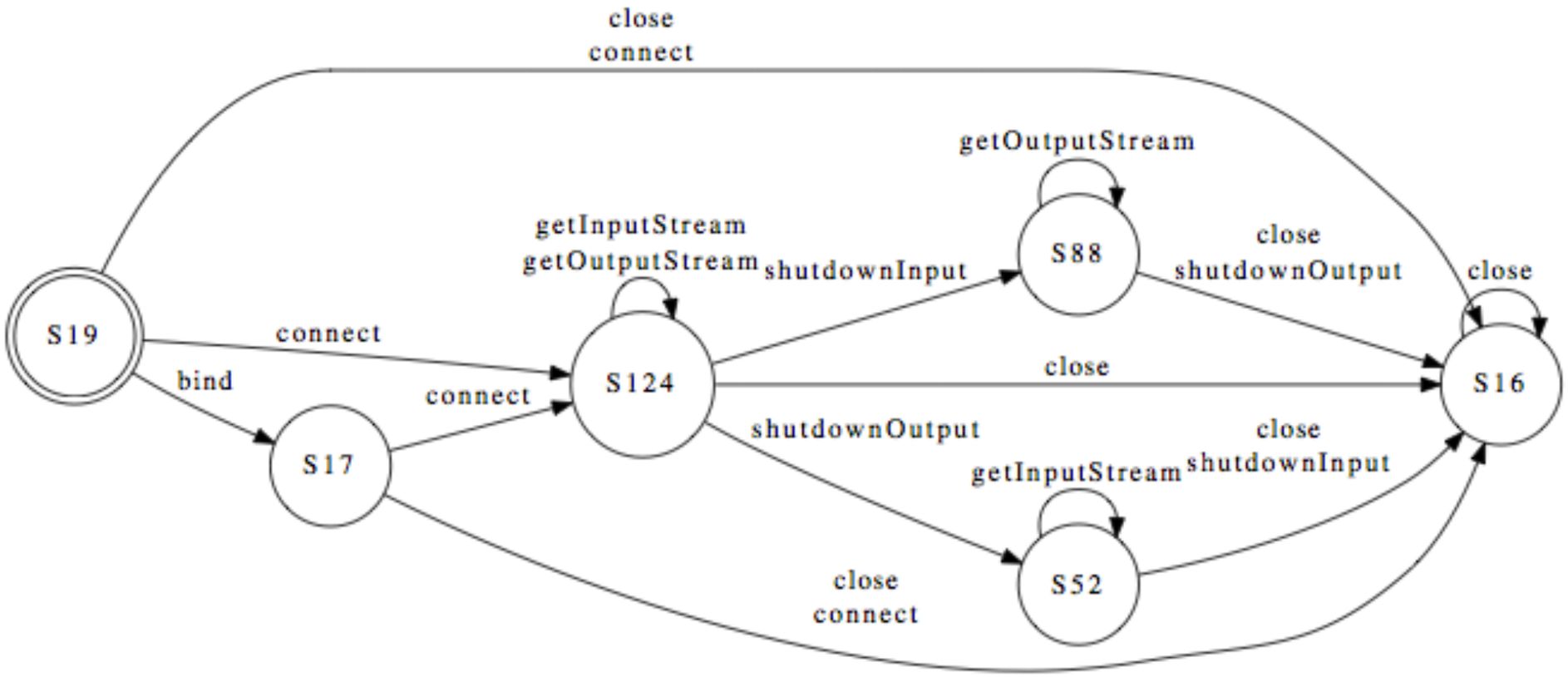
# Experimental Evaluation: Findings

## Java ListIterator




No issues identified

# Java Socket Implementation

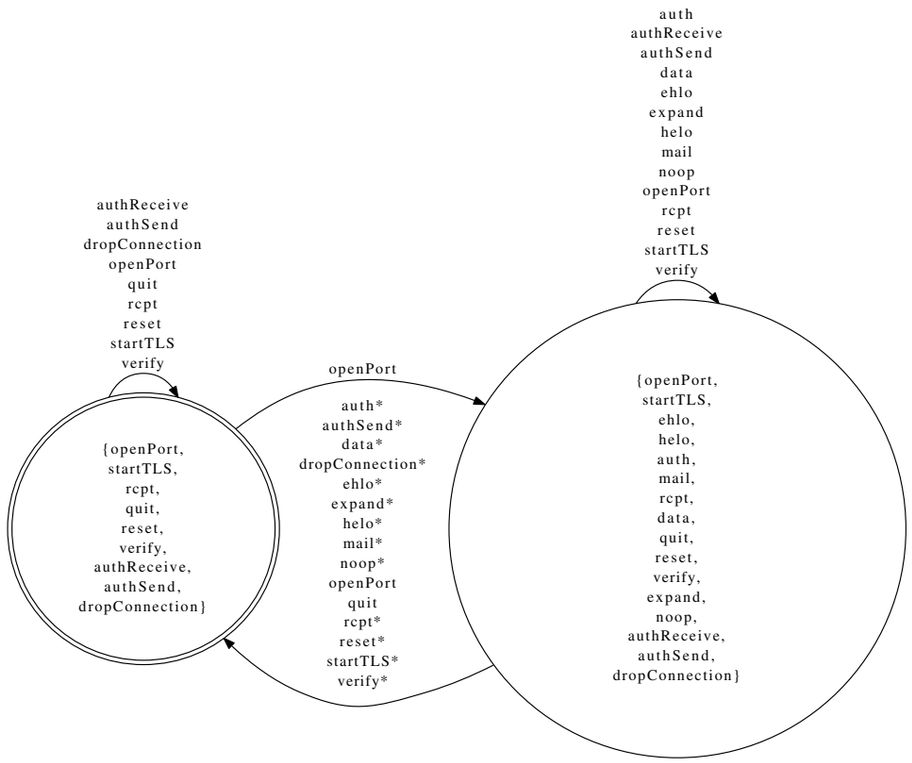


# SMTP in Java

Ristretto: Java Implementation that facilitates the construction of email clients

# Experimental Evaluation: Findings

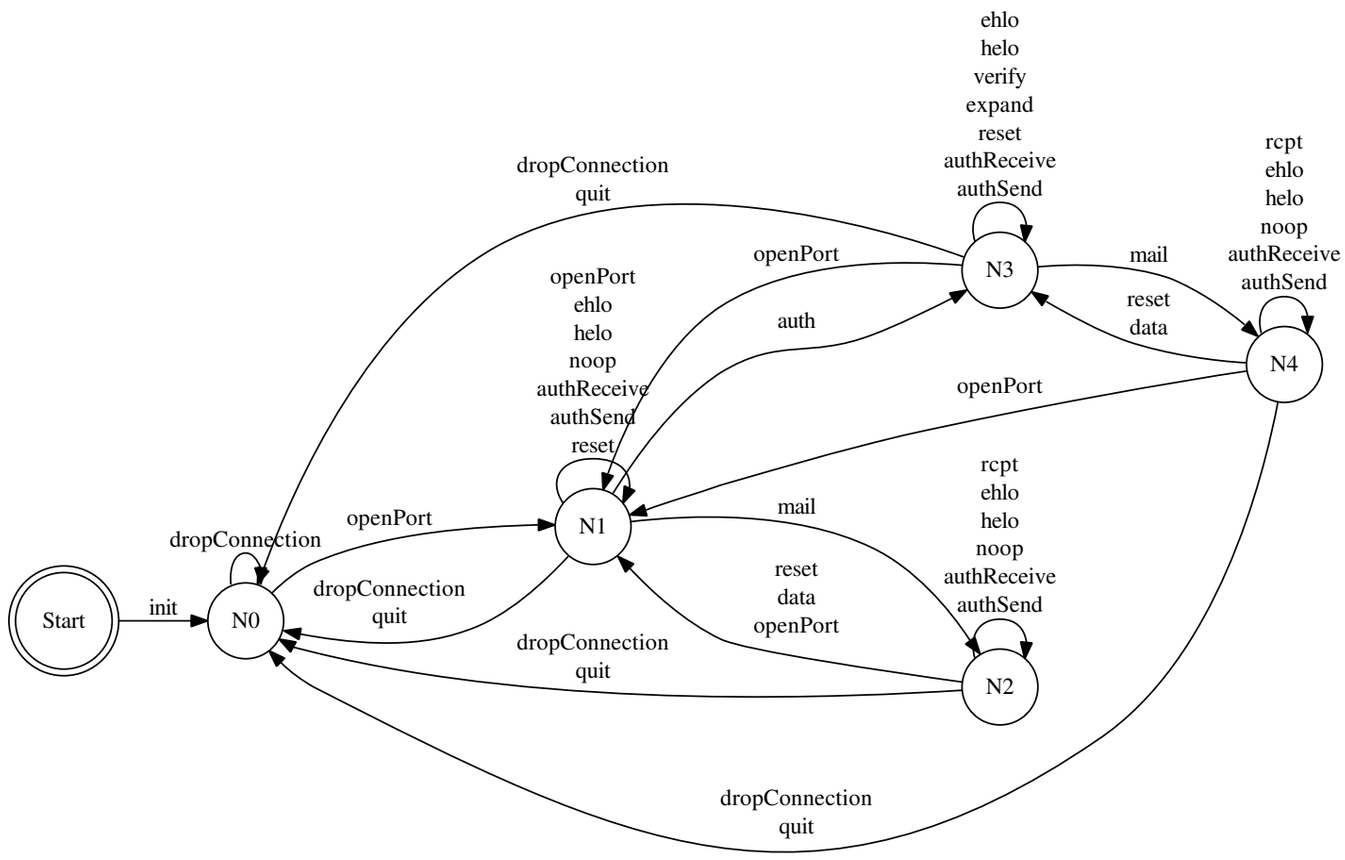
## Java SMTP client (SMTPProtocol)



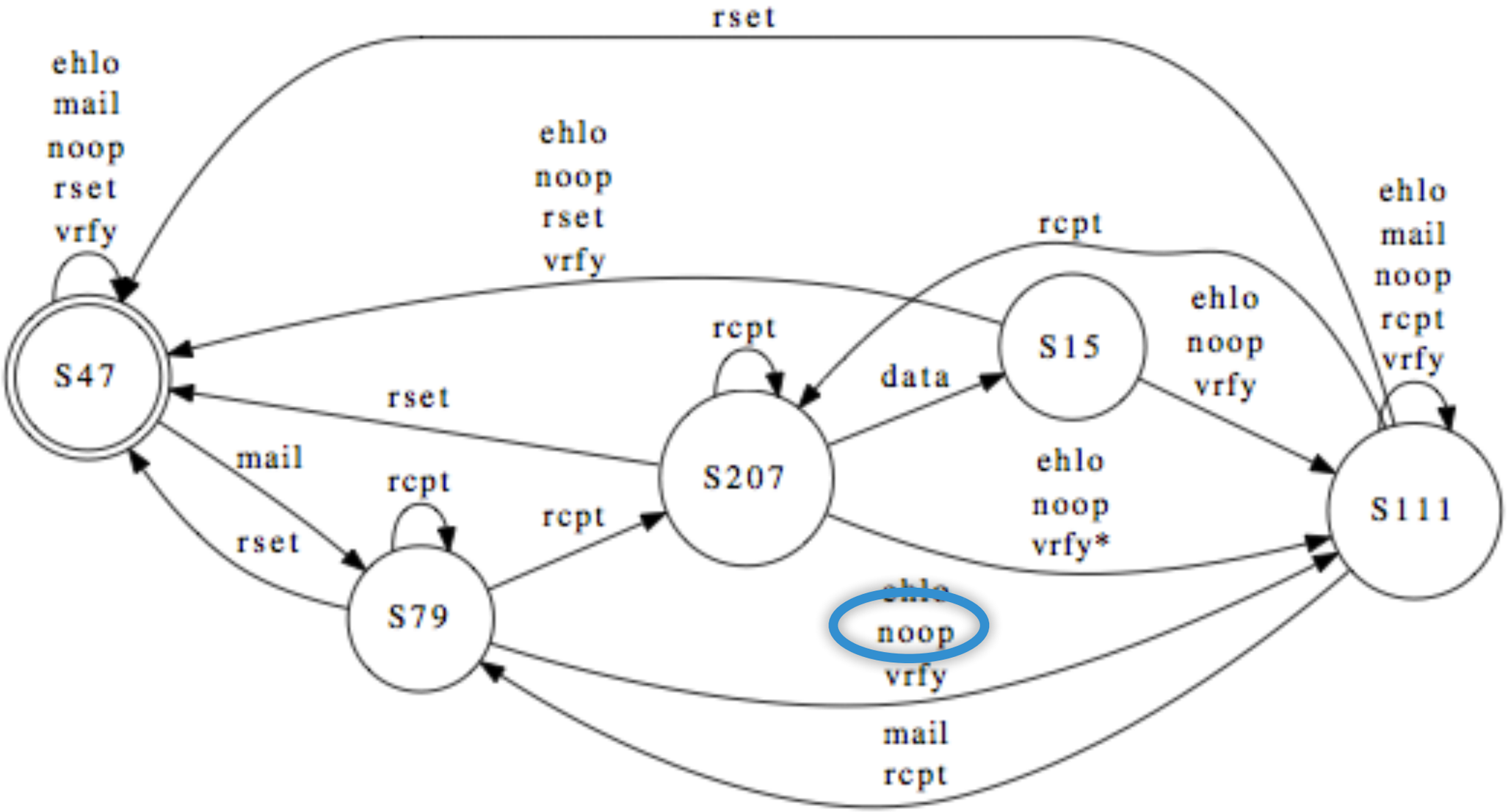

All message ordering restrictions checking is delegated to the SMTP server

The model was constructed using a particular server as environment

Abstraction according to [Dallmeier et al. - ISSTA 2010]



# Java SMTP Server Implementation

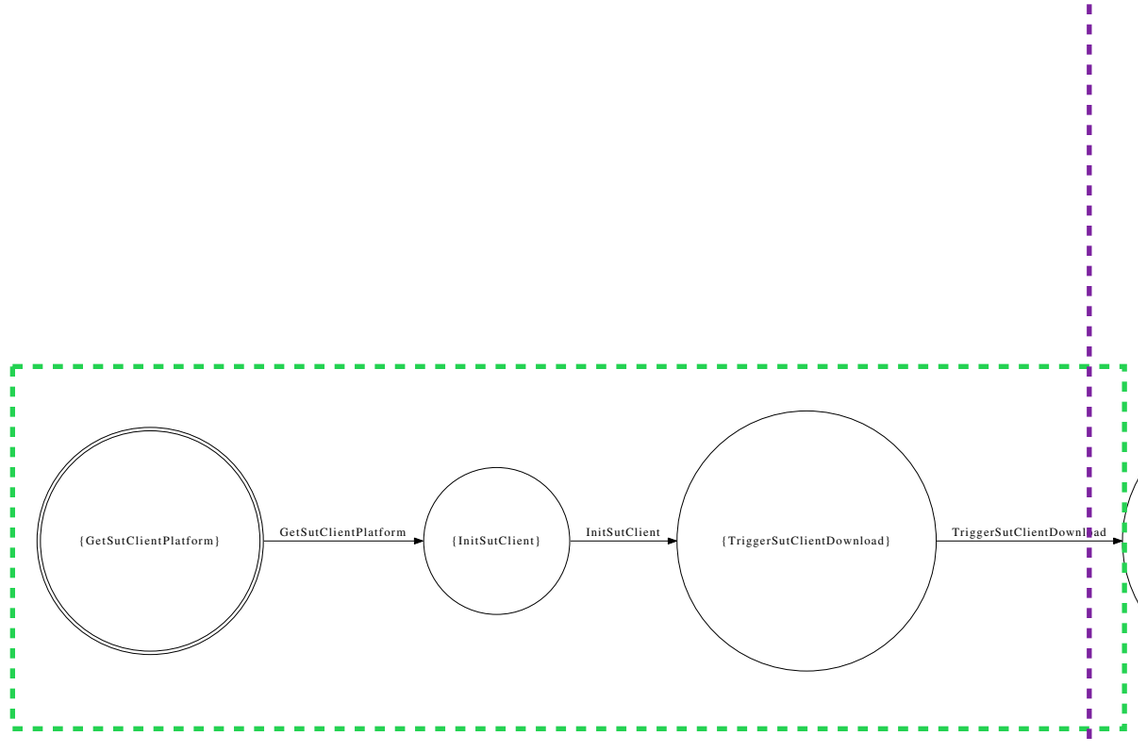


# Experimental Evaluation: Findings

Microsoft Peer Content Caching Retrieval Server Side Protocol

SpecExplorer model (C# code)

Requires clause for operation is too weak allowing execution of premature timeouts




Initialisation phase

Suspicious trap state

Almost unordered messages

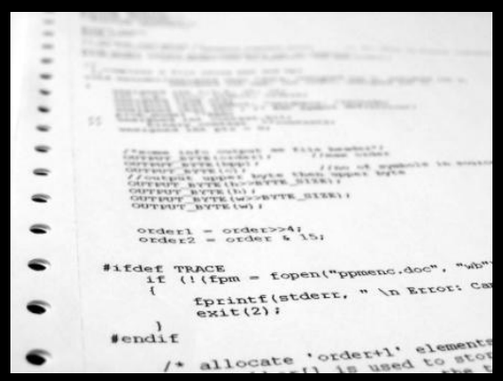
# Basic questions

- Are the abstractions useful for finding problems?
- Does the construction technique scale?

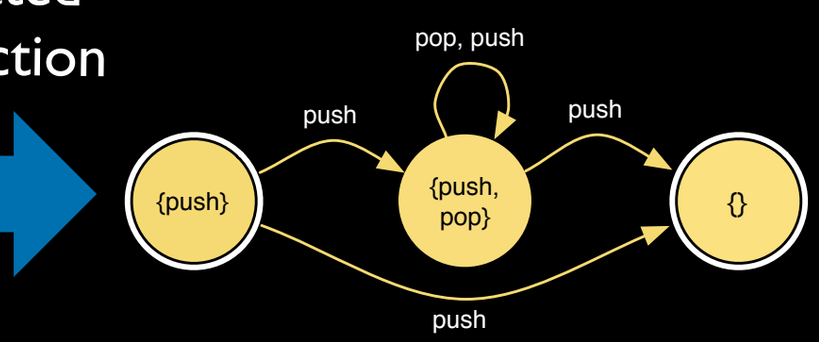
# Case Studies (Code)

| Name | Actions | Time (s) | States | Transitions | Query Certainty | Issues Found |
|---|---|---|---|---|---|---|
| List (Buggy) | 3 | 6 | 3 | 7 | 100% | 1 |
| List (Patched) | 3 | 7 | 3 | 8 | 100% | 0 |
| PipedOutputStream | 4 | 8 | 3 | 8 | 100% | 2 |
| Signature | 5 | 13 | 3 | 10 | 100% | 0 |
| ListItr | 5 | 314 | 7 | 32 | 97.5% | 1 |
| Socket (1st) | 8 | 2531 | 9 | 38 | 100% | 0 |
| Socket (2nd) | 8 | 1472 | 6 | 19 | 100% | 0 |
| SMTP Server | 9 | 1642 | 5 | 24 | 99.8% | 2 |
| PCCR (1st) | 12 | 628 | 16 | 34 | 100% | 1 |
| PCCR (2nd) | 12 | 398 | 10 | 14 | 100% | 0 |
| SMTP Client | 16 | 288 | 2 | 39 | 96.7% | ∞ |

# A Side effect...

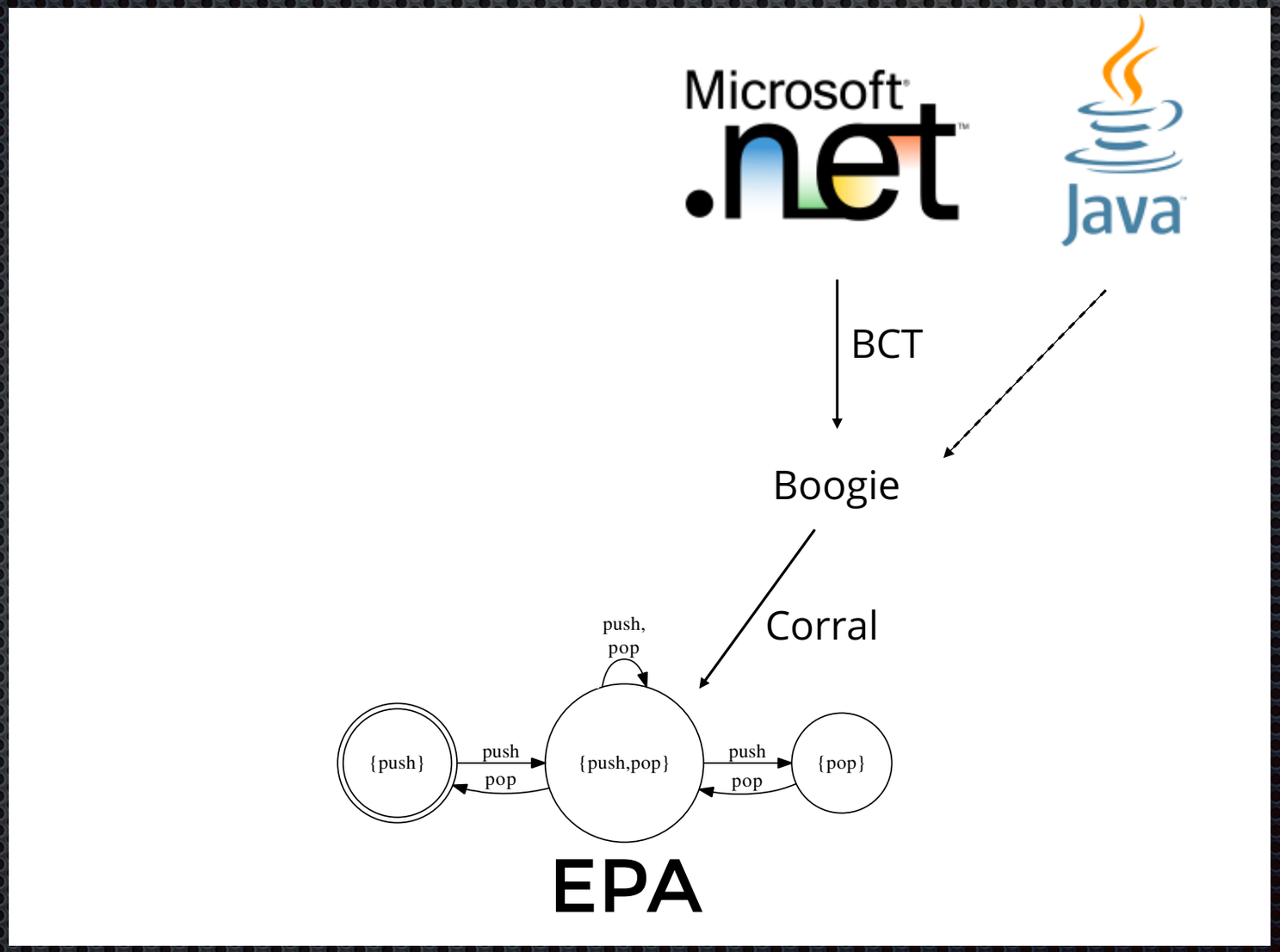


Automated Construction



## Automated Documentation Generation

# Generating EPAs using Boogie



Spec#

VCC (C)

HAVOC (C)

Dafny

Chalice

Eiffel (AveProofs)

Java BML

Region Logic

Corral

Poiro

Forró

QED

C B Analyze

STORM C)

Boogie x86

…

Boogie

inference

SymDiff

Simplify

SMT Lib

Z3

SMT Lib 2

Isabelle/ HOL

# Boogie language overview

## Mathematical features

- `type T`
- `const x…`
- `function f…`
- `axiom E`

## Imperative features

- `var y…`
- `procedure P… …` *spec…*
- `implementation P…` { *…body…* }

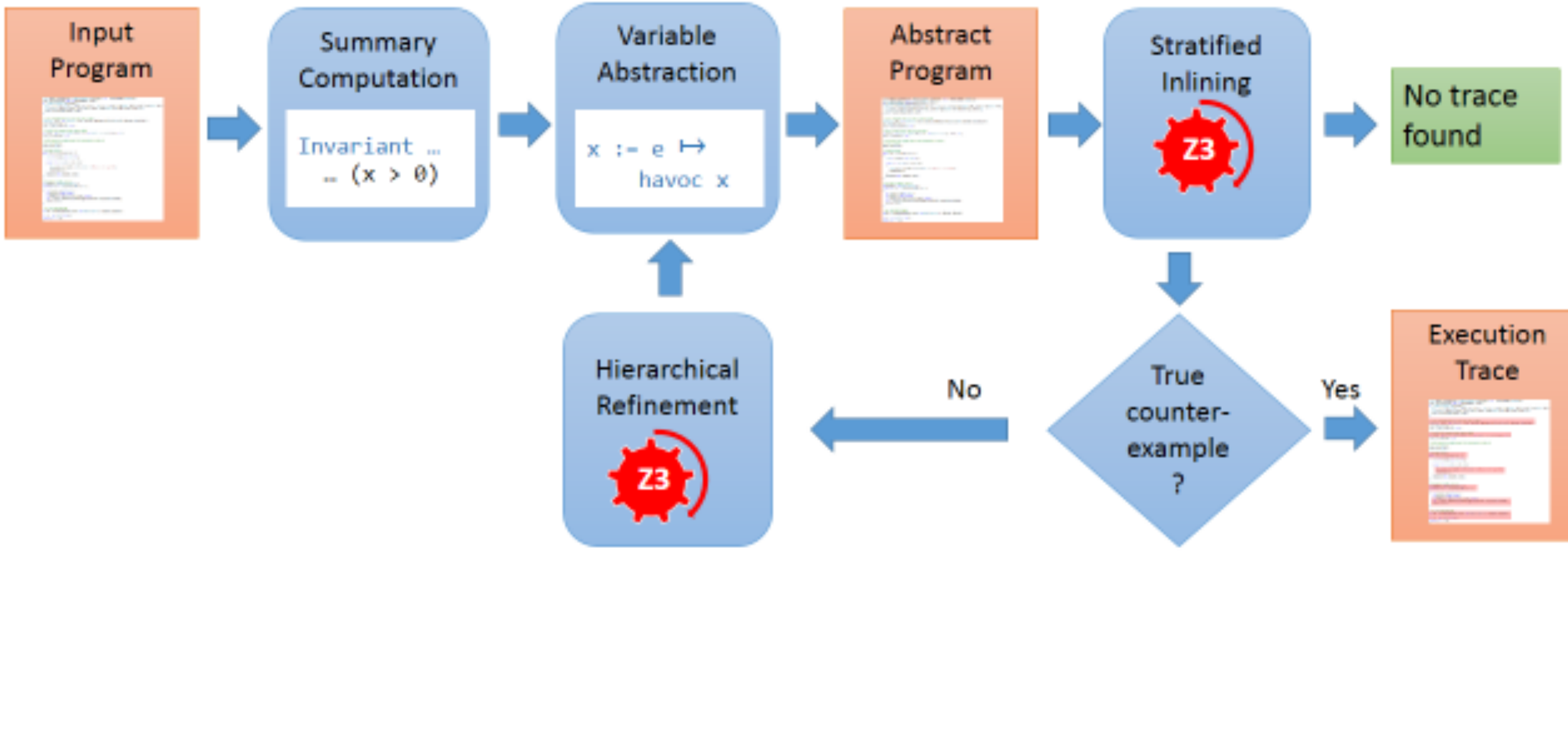
# Boogie statements

- `x := E`
- `a[i] := E`
- `havoc x`
- `assert E`
- `assume E`
- `;`
- `if`
- `while`
- `break`
- `label:`
- `goto A, B`

# Corral

- A whole-program analysis tool for **Boogie** programs.
- Goal-directed symbolic search techniques to **find assertion violations.**
- 3 possible results
  - True bug,
  - No bugs,
  - Recursion Bound Reached

# Corral Architecture



# Corral vs Blast

- Blast = **CEGAR** approach
  - Abstraction refinement => **Over approximation**
    - **Unreach** means Target is not reachable
    - Reach means Target is reachable
    - Unknown means Target may be reachable
- Corral = **Hybrid** approach
  - Inlining + CEGAR => **Under approximation**
    - No bug: assertion not violated (not reachable)
    - **True bug:** assert violation (reachable)
    - reached bound K (target not found so far… => may be reachable)

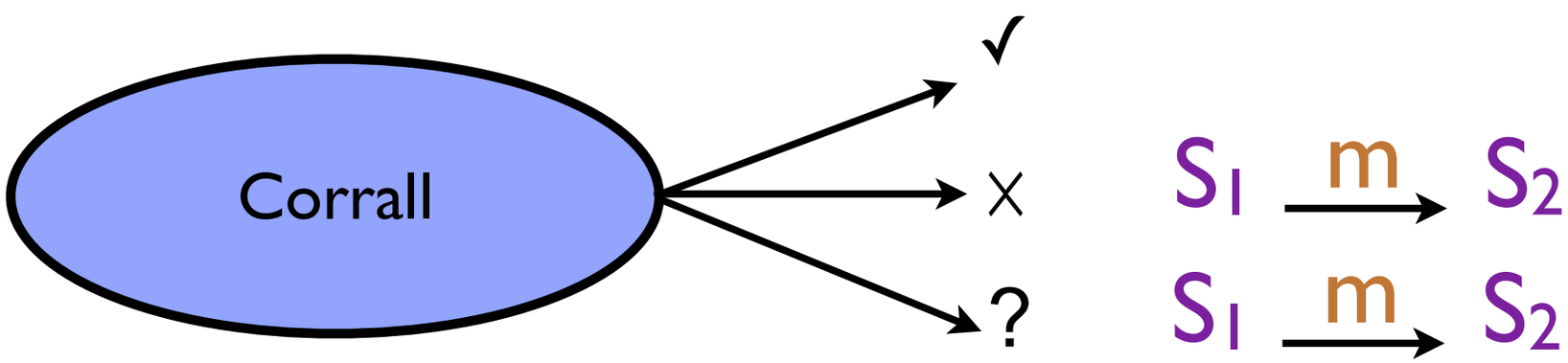
# Query transition in Corral

To see if an abstract state $S_1$ evolves via m to $S_2$ we try to verify the following method

```
procedure FeasibleTransition(S1: State, m: Action,  S2: State)
{
    assume pred(S1);
    call m();
    assert !pred(S2);
}
```

Bug means assert failure => **exist some state** from **S1** such that after executing **m** reach an state in **S2** =>

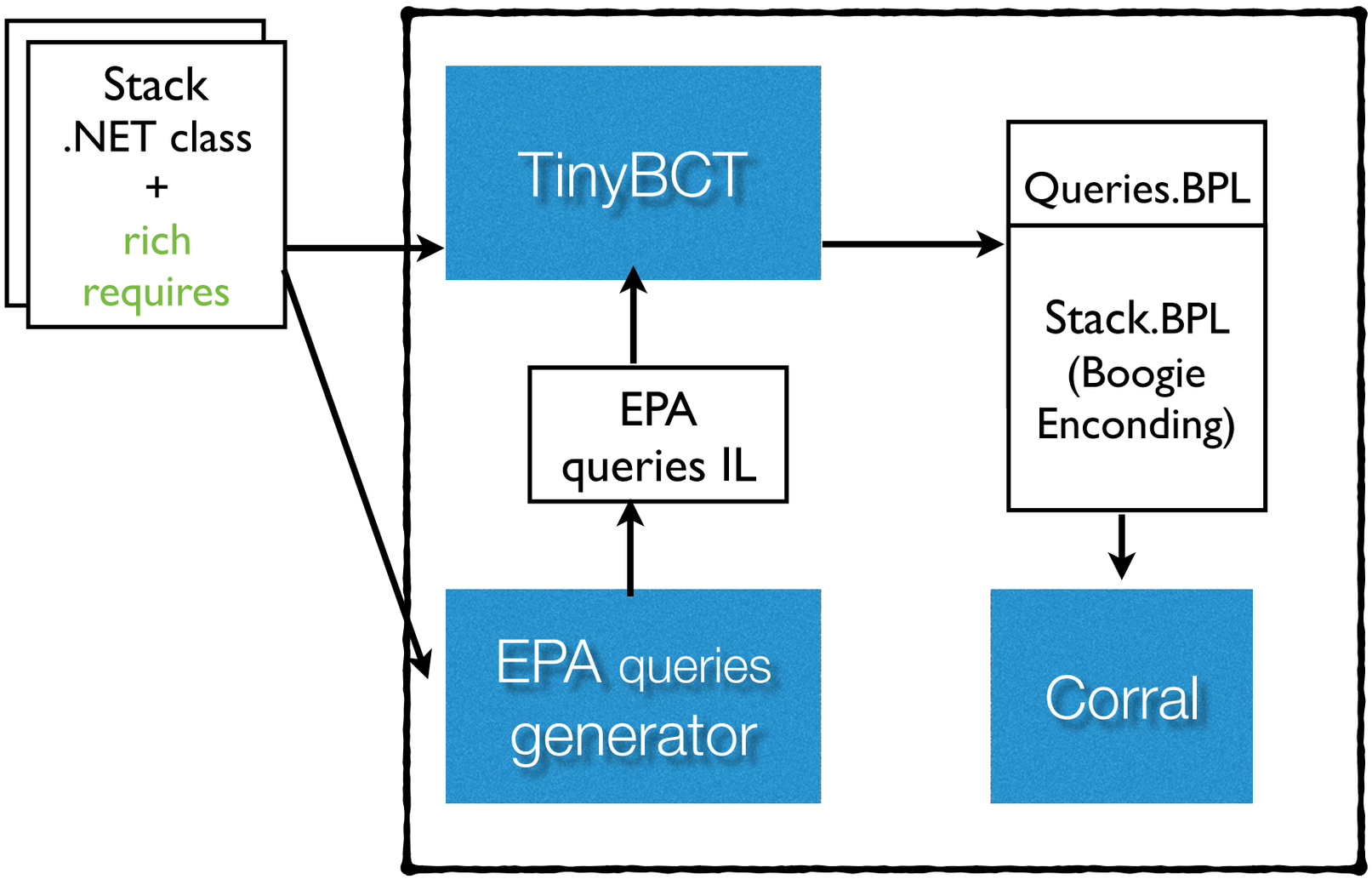
$S_1 \xrightarrow{m} S_2$

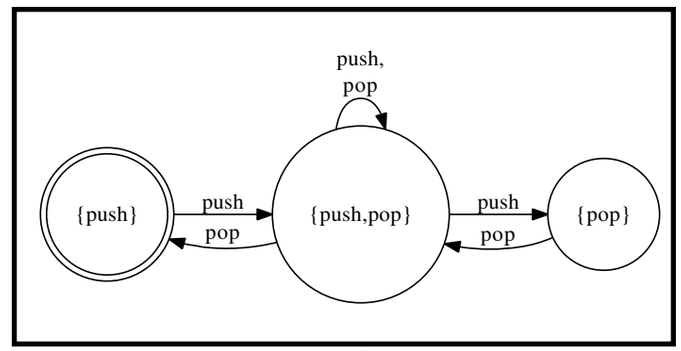


# (Tiny)BCT

- A translator from .NET code + Boogie implemented in LAFHIS (with some help of SRI+Microsoft)
- Support all .NET features
  - Structs, passing by ref, async calls, delegates, polymorphic method invocation
- TinyBCT + Corral = Software Model Checker for .NET :-)

# contractor.NET



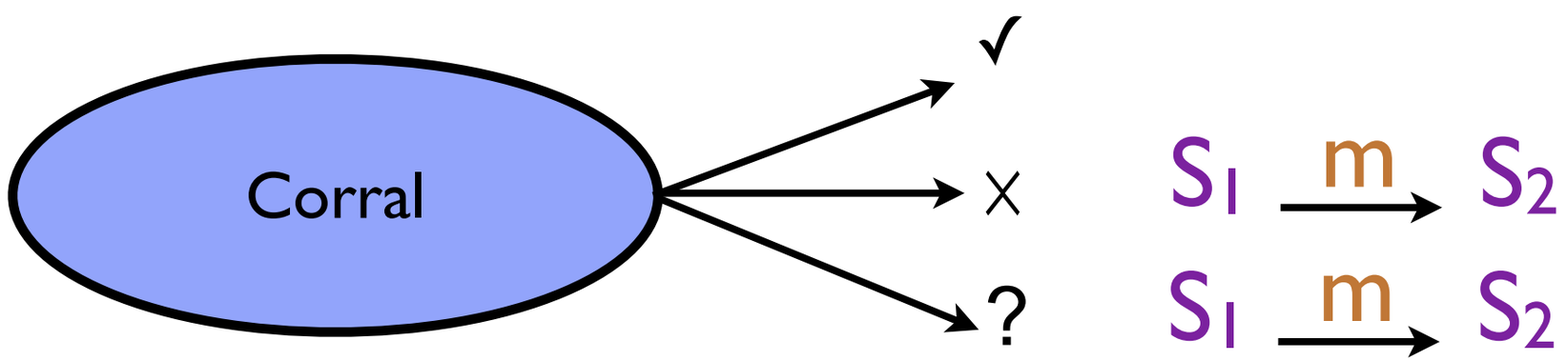
# Query transition in Corral

To see if an abstract state{push,pop} evolves via push to {push} we try to verify the following method

```
procedure pushPop_push_push(int e1)
{
    assume pred_push_pop();
    call push(e1);
    assert !pred_push();
}
```

Bug means assert failure => **exist some state** where **push** and **pop** are enabled, such that exist a parameter **e1** enabling **push** and after executing **push** at the resulting state only **push** is enabled

$$pred_{\{push\}}(s) \Leftrightarrow inv(s) \wedge (\exists e)(R_{push}(s, e)) \wedge \neg R_{pop}(s)$$



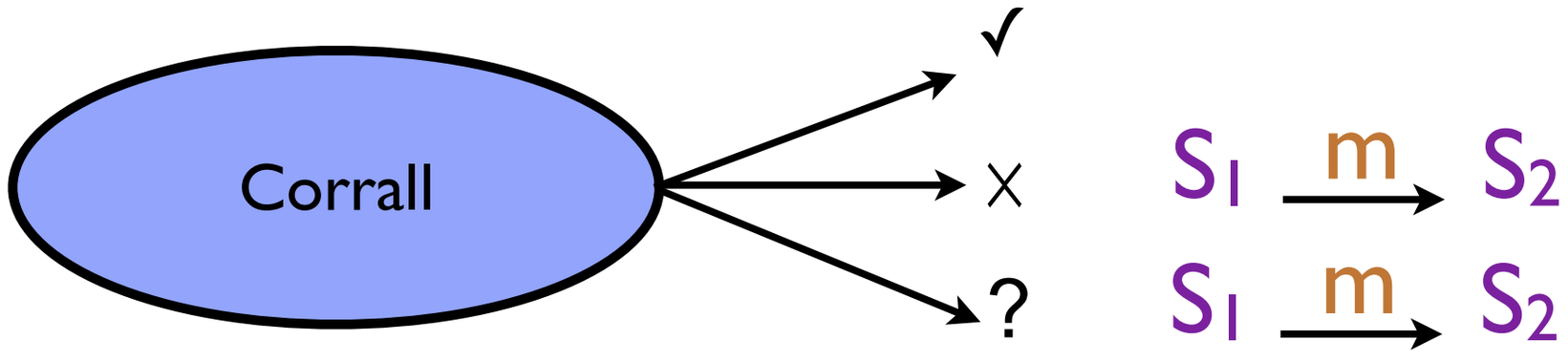
# Query transition in Corral

To see if an abstract state{push,pop} evolves via push to {push} we try to verify the following method

```
procedure pushPop_push_push(int e1, int e2)
{
    assume pred(this) && pre_push(this, e1) && pre_pop(this);
    call this.push(e1);
    assert !(pred(this)&& pre_push(this, e2) && !pre_pop(this));
}
```

$$inv_{\{push\}}(s) \Leftrightarrow inv(s) \wedge (\exists e)(R_{push}(s, e)) \wedge \neg R_{pop}(s)$$

Assertion fails when after executing **push** (i.e., a new state) there exist **e2** such that inv(this)&& pre_push(this, e2) && !pre_pop(this)



# Again problems with parameters

To see if an abstract state{push,pop} evolves via push to {pop} we try to verify the following method

```
procedure pushPop_push_pop(int e1, int e2)
{
    assume pred_push_pop();
    call this.push(e1);
    assert !pred_pop();
}
```

$$Pred_{\{pop\}}(s) \Leftrightarrow inv(s) \wedge !(\exists e)(R_{push}(s, e)) \wedge R_{pop}(s)$$

Assertion fails when after executing **push** (i.e., a new state) there **is no parameter e2** such that pre_push(this, e2) and …

Corral cannot check all values por **e2…**
Again: parameter splitting!

# Parameter splitting in Corral

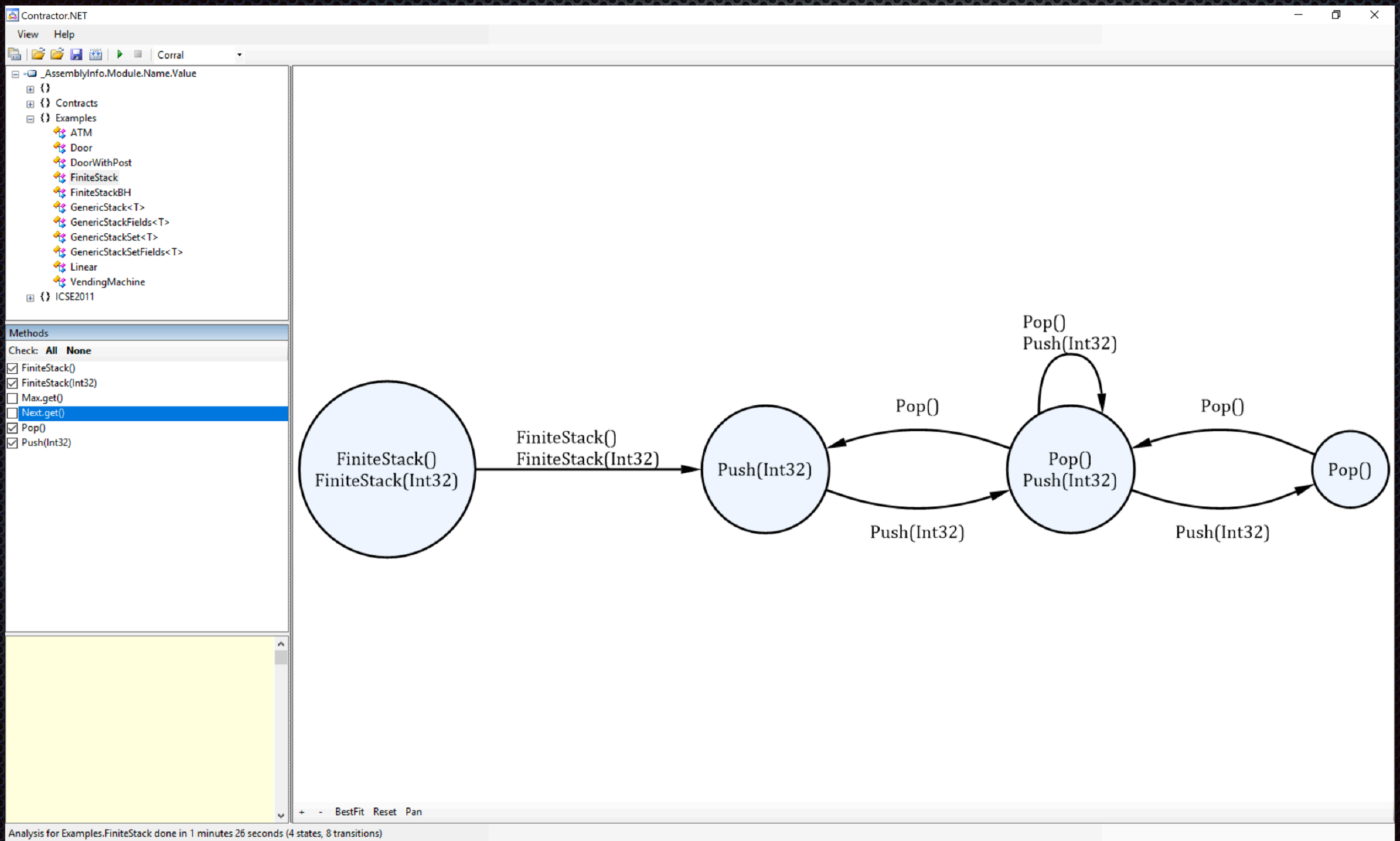
Given SR_push, PR_push

```
procedure pushPop_push_pop(int e1, int e2)
{
    assume pred_push_pop();
    call this.push(e1);
    assume PR_push(e2);
    assert !(inv(this)&& !SR_push(this) && SR_pop(this));
}
```

$$Pred_{\{pop\}}(s) \Longleftrightarrow inv(s) \wedge !(\exists e)(PR_{push}(e) \wedge SR_{push}(s)) \wedge SR_{pop}(s)$$

$$Pred_{\{pop\}}(s) \Longleftrightarrow inv(s) \wedge !SR_{push}(s)) \wedge SR_{pop}(s)$$

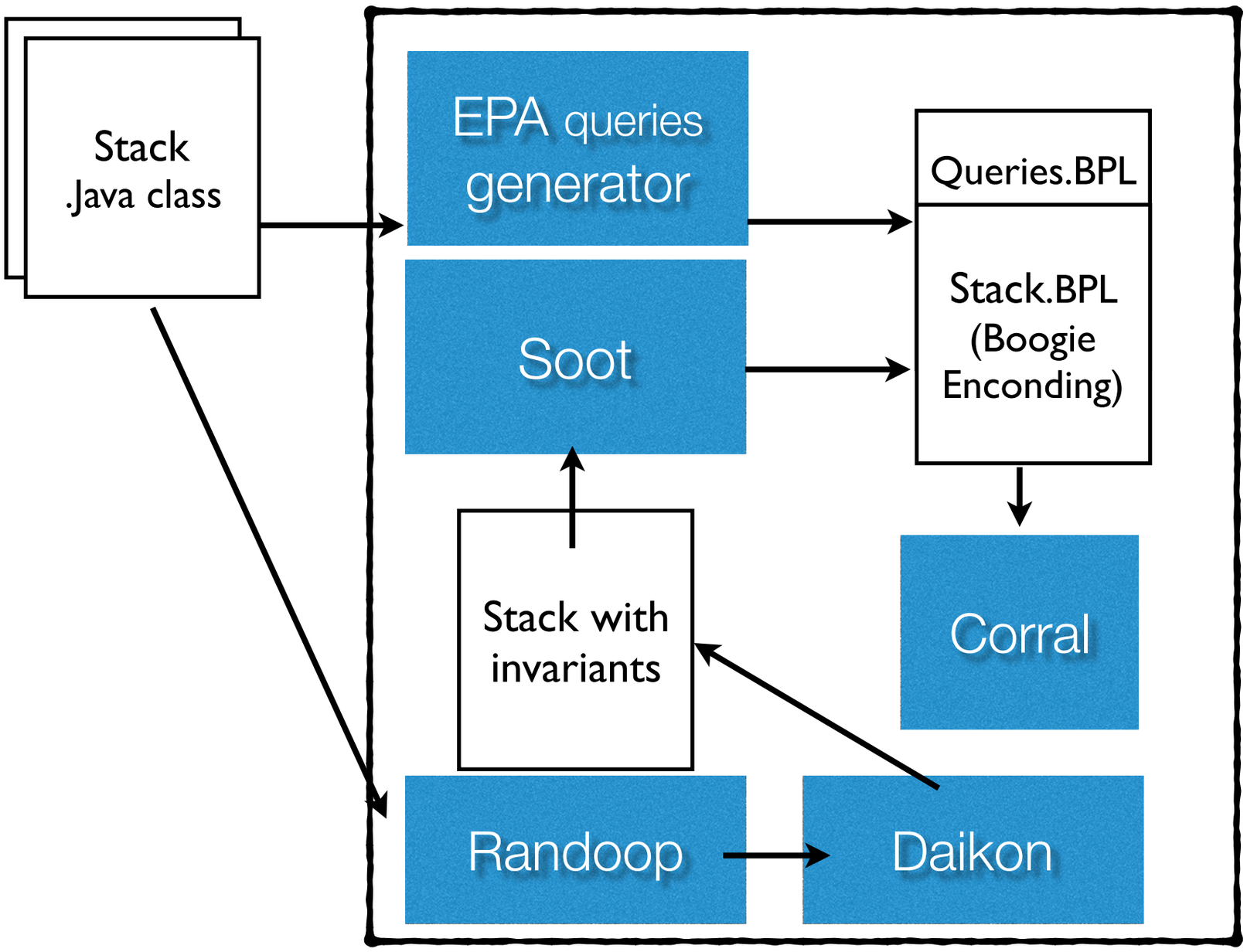
# Contractor.NET

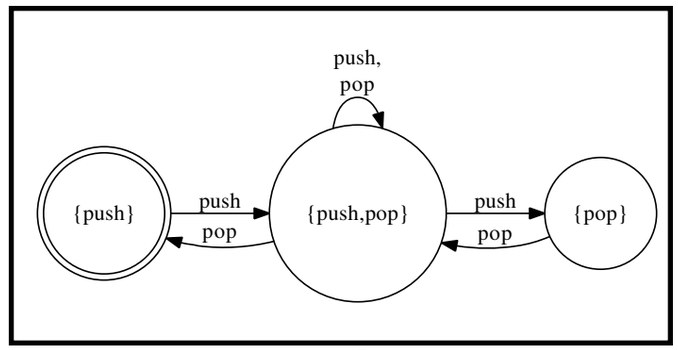


# Limitations

- Loops/recursion
  - Corral analyzes loops until a certain bound
  - vs Blast that over-approximates the loops and may find loop invariants
- Class invariants
- Preconditions
  - Do we really need actual preconditions?

# ContractorJ

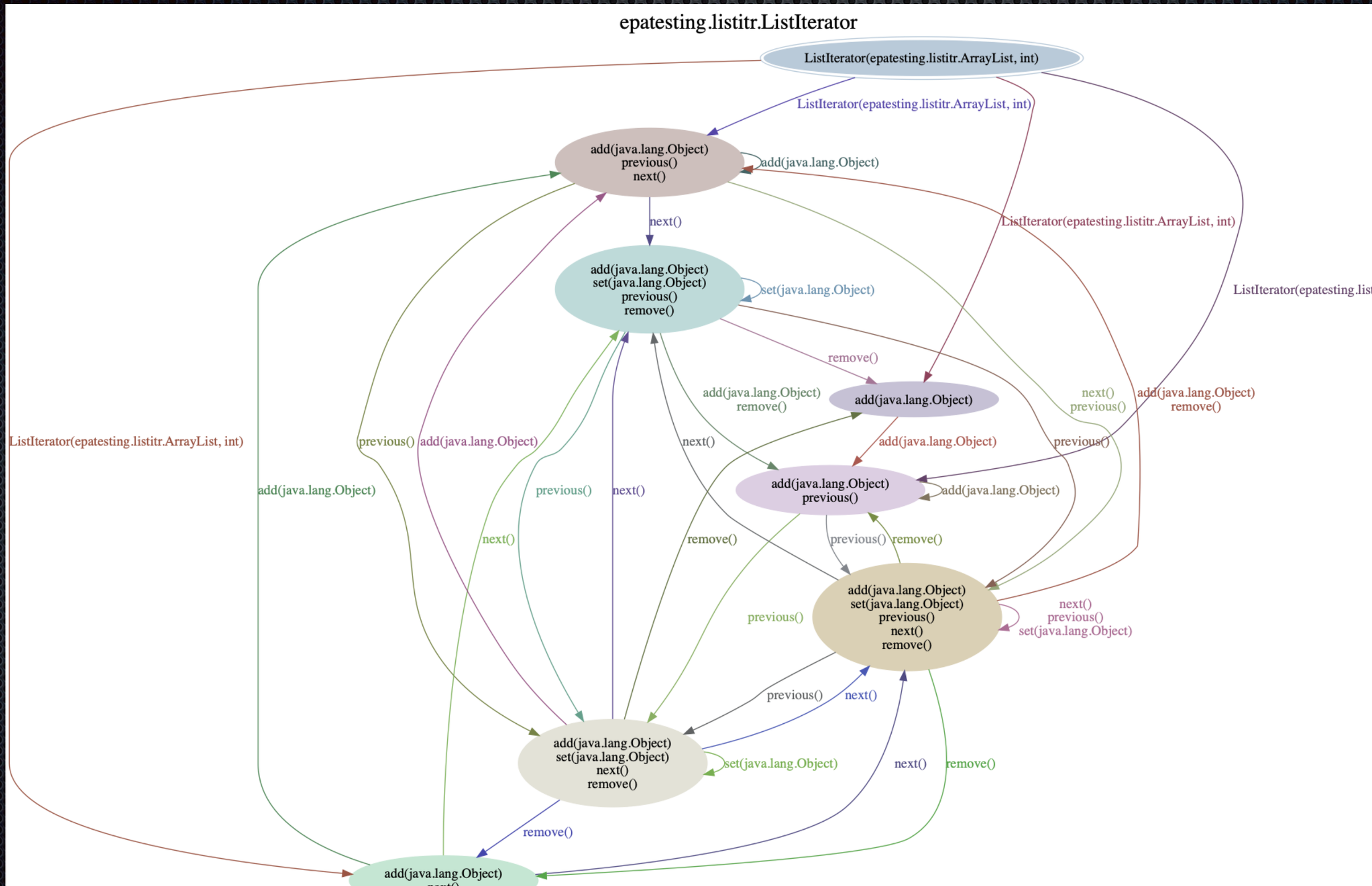


# ContractorJ

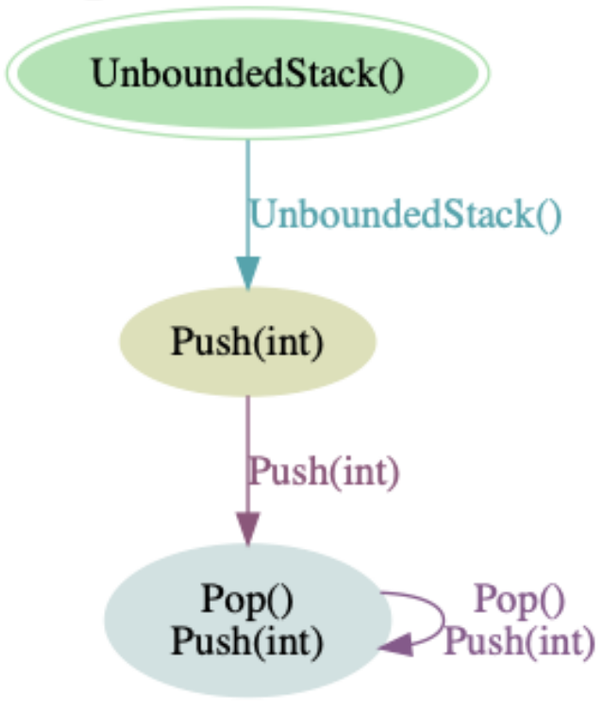
- Support Java classes
- PEPA: EPA with intended preconditions
- Model exceptions
- Automatic inference of class invariants

# List Iterator con contractorJ

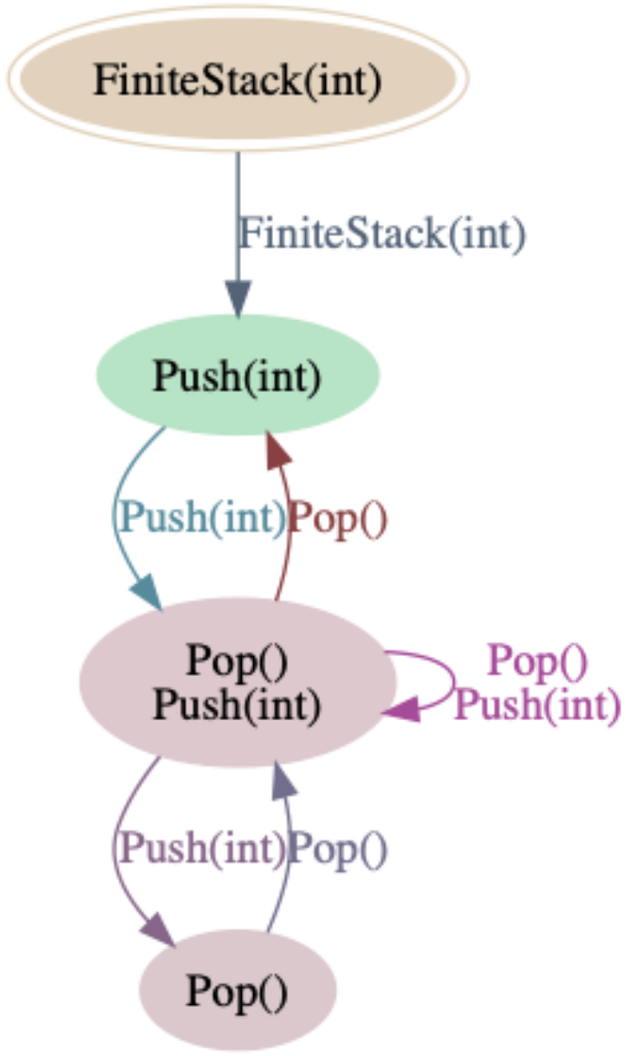


# Stacks

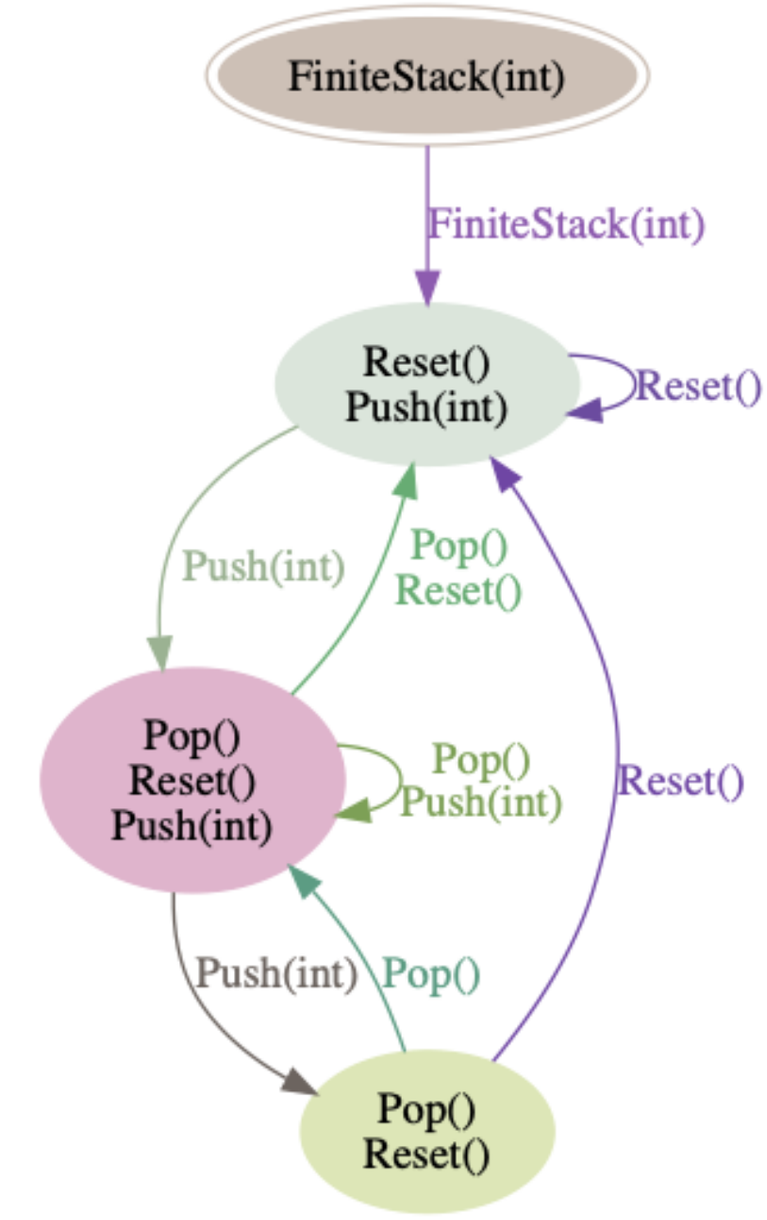
examples.UnboundedStack



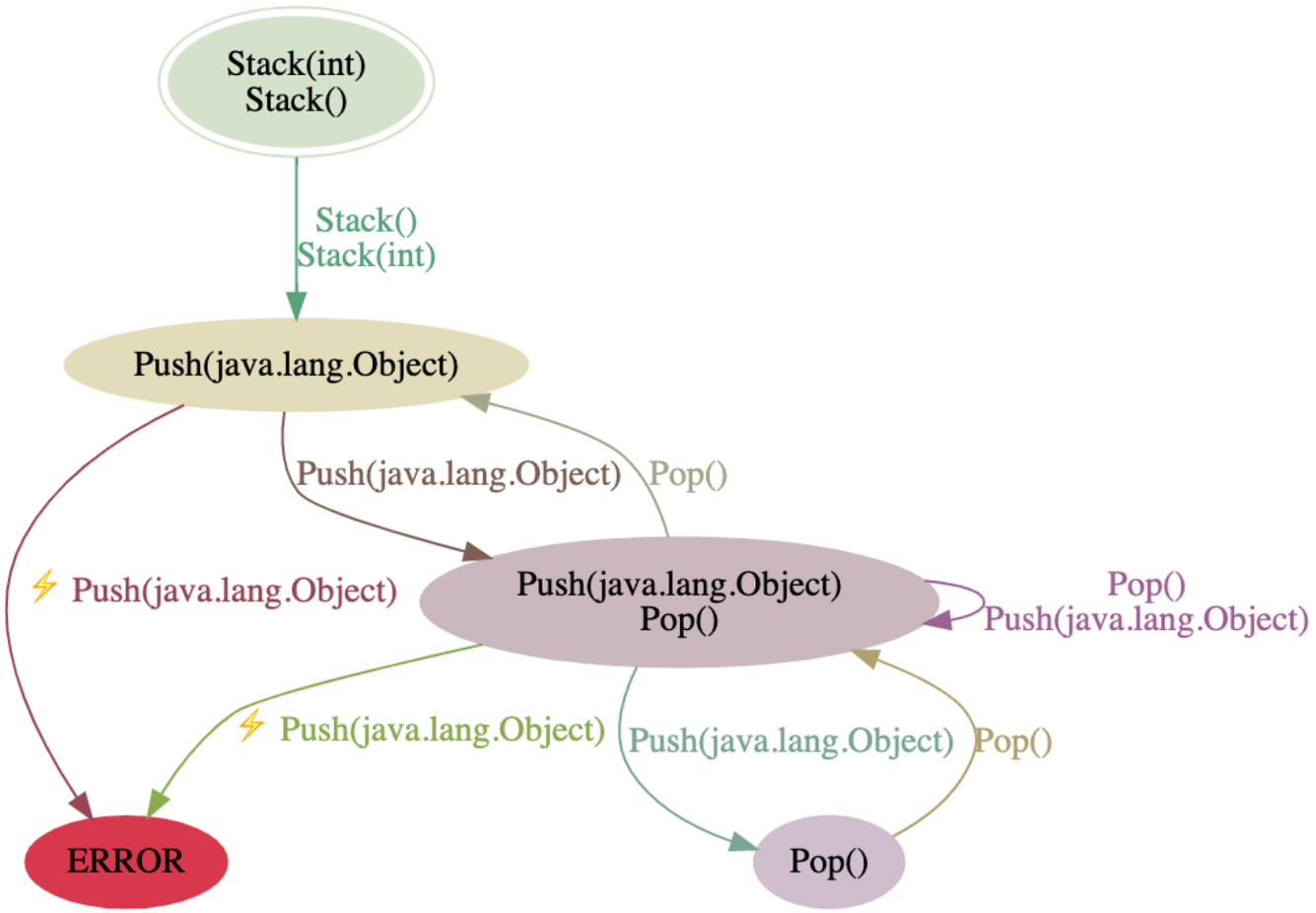

epatesting.stack.FiniteStack




epatesting.stack.FiniteStack

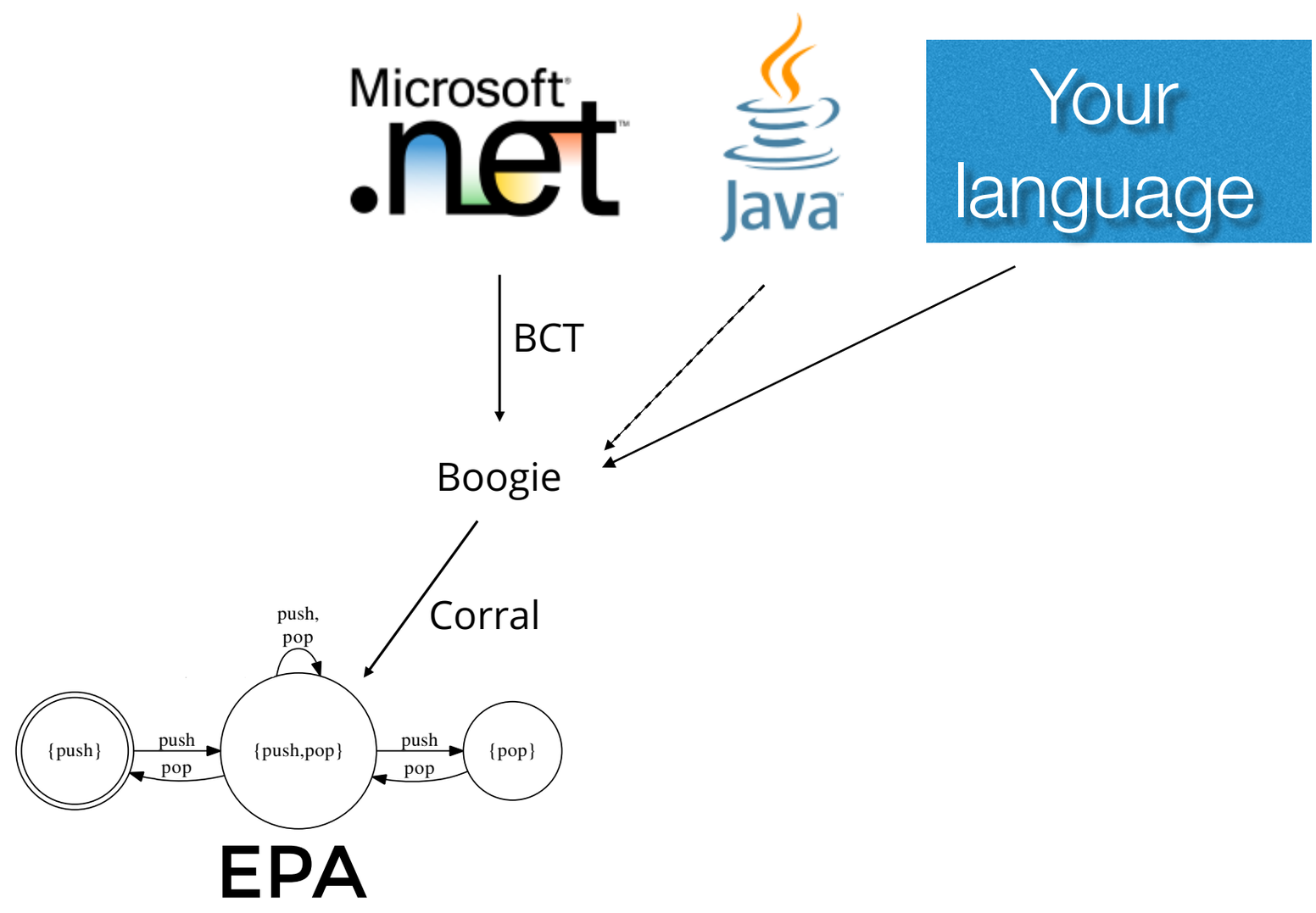


# Exceptions in PEPA

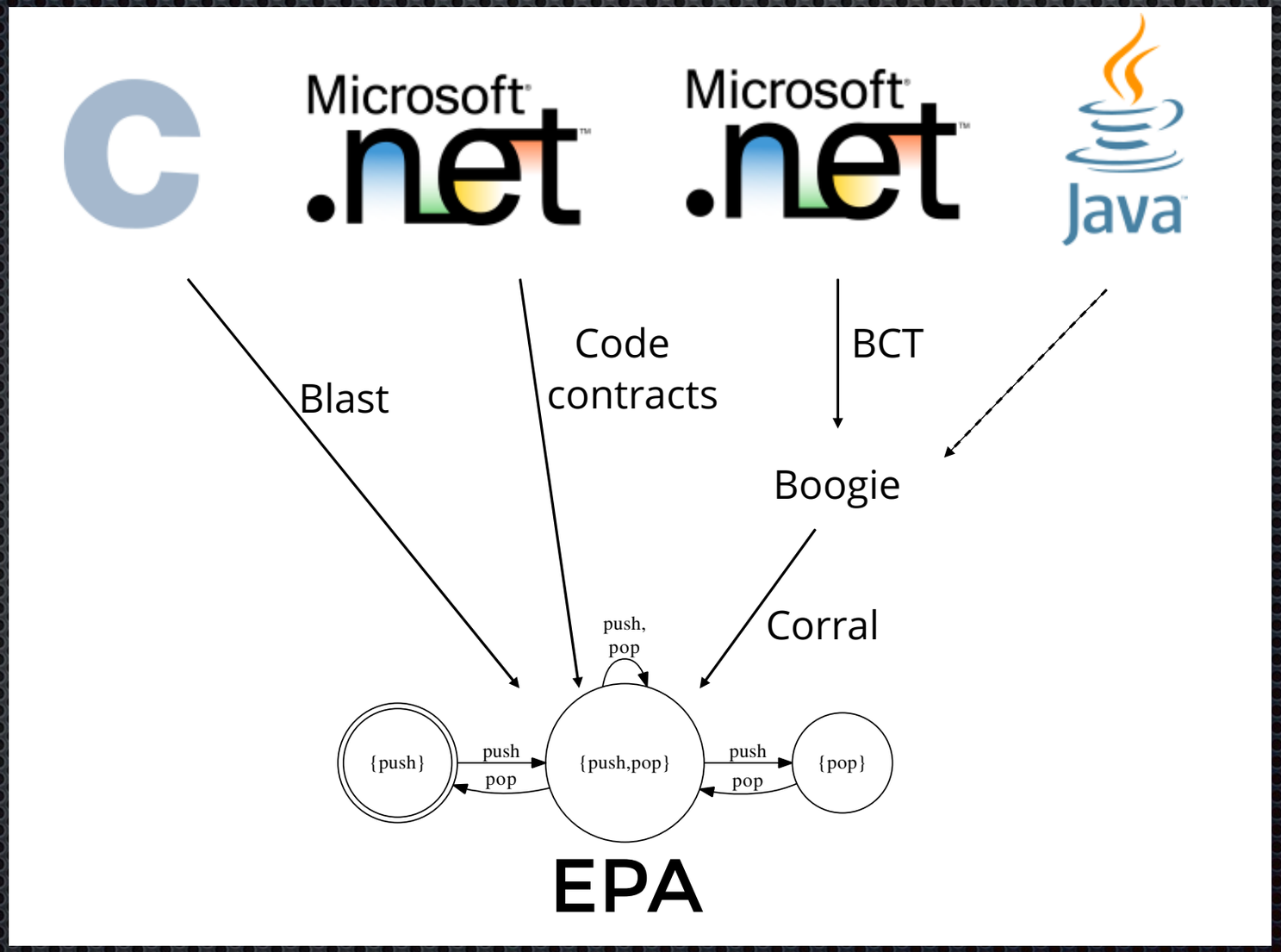



```
public void Push(Object o) {
 if (o == null) {
  // breaks invariant!
  size = capacity + 1;
  throw new NullPointerException();
  }
  /* ... */
}
```

# The Boogie approach

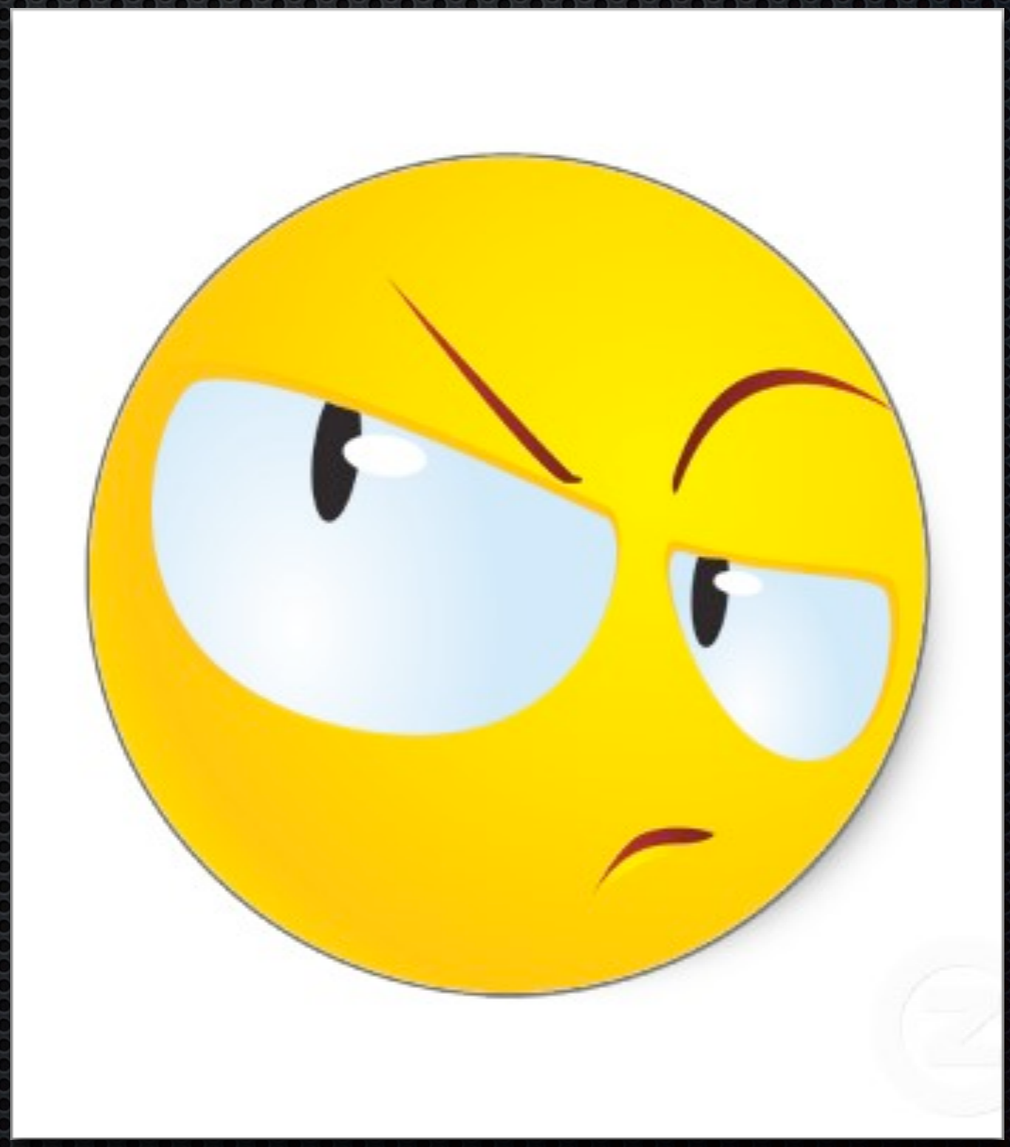


# Building EPAs for code



Ok, you found real faults....

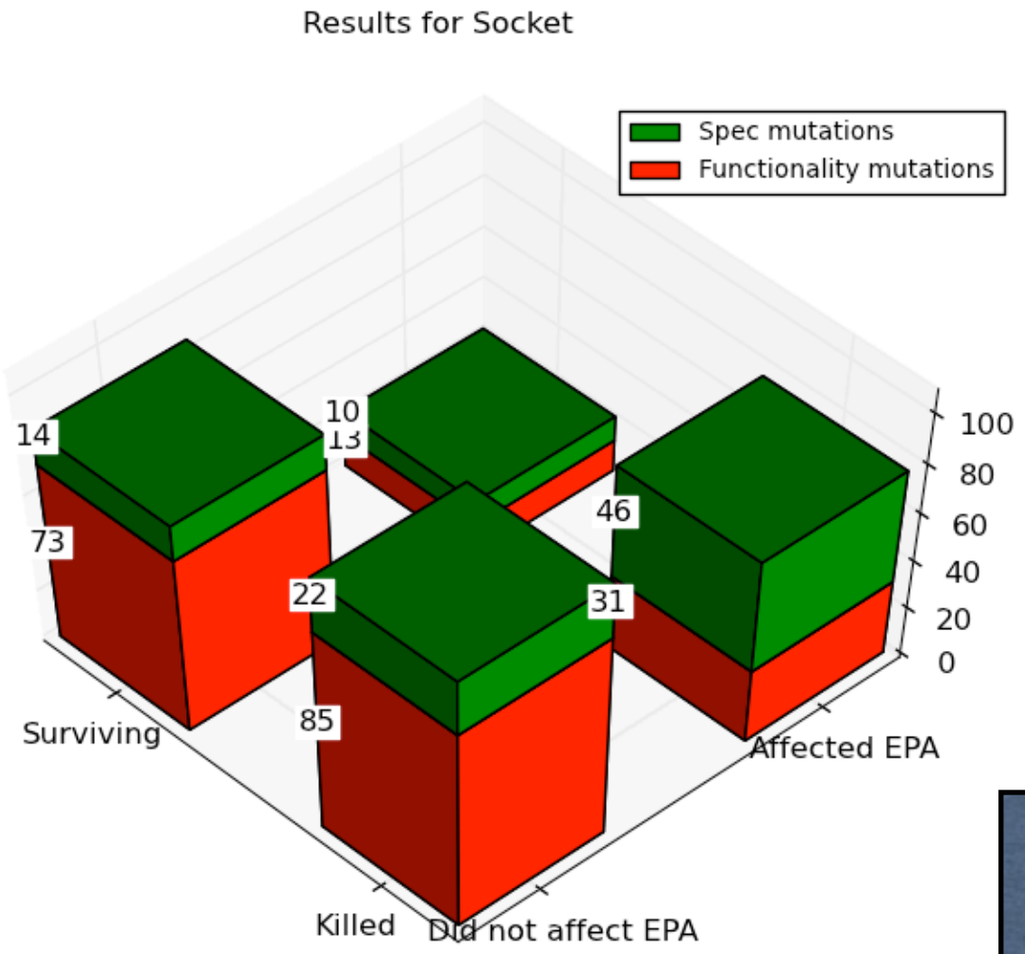
...but how many did you miss?



# Sensitivity Analysis

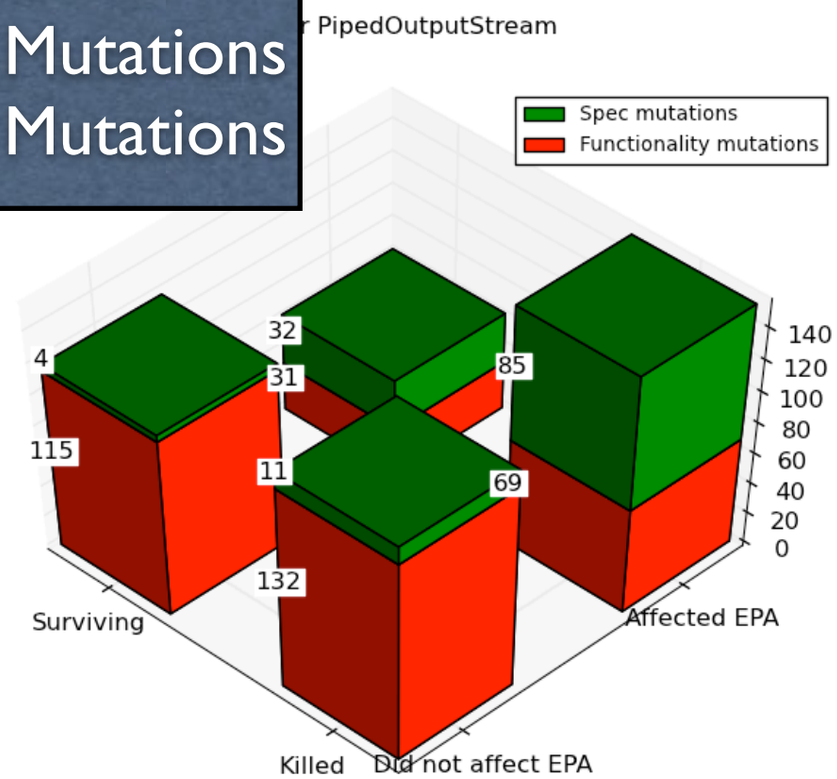
## based on Mutants

90% of Spec Mutations
56% of Functional Mutations

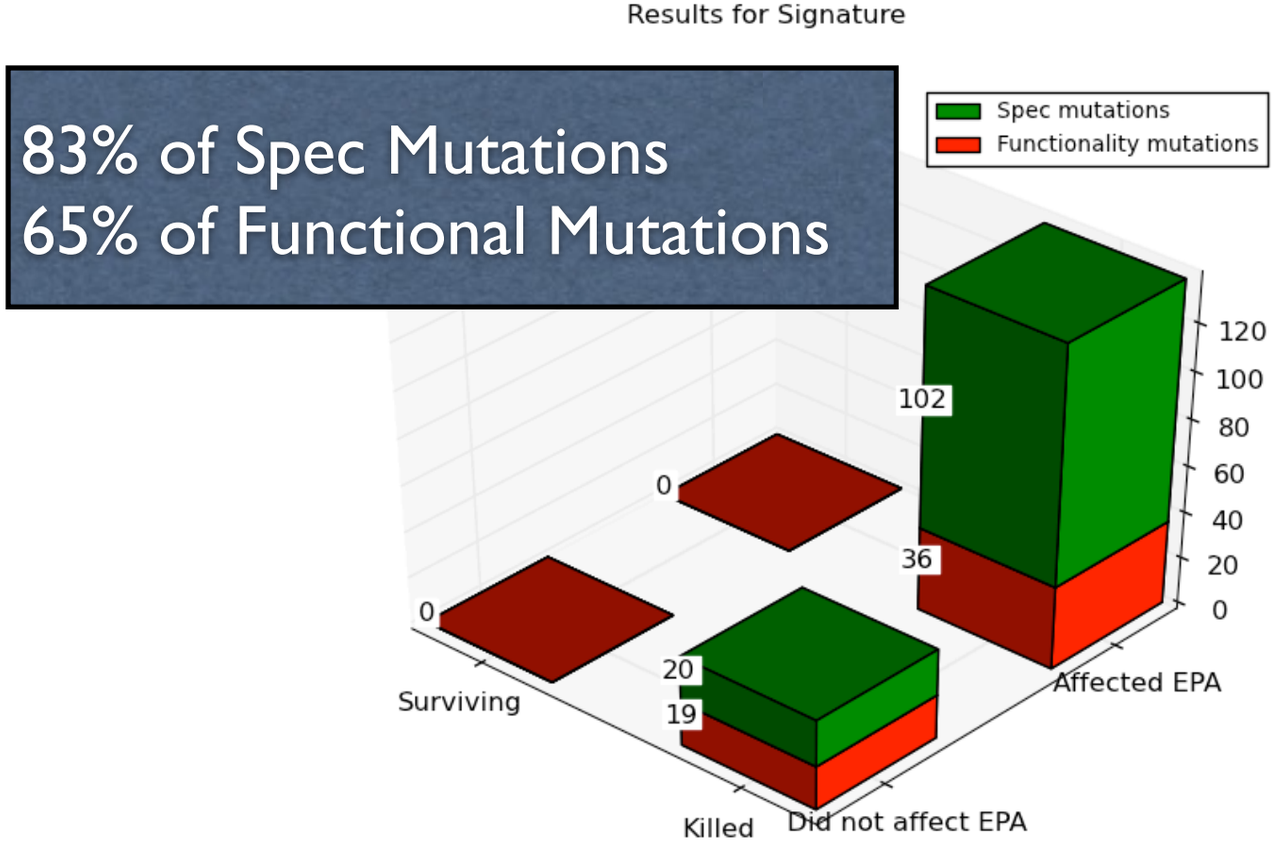



67% of Spec Mutations
26% of Functional Mutations

88% of Spec Mutations
34% of Functional Mutations




83% of Spec Mutations
65% of Functional Mutations



# Enabled Preserving Abstractions

- Can be constructed automatically and statically from code
- Evidence that they provide a “good” precision/size compromise
- Applied to programs with rich interaction protocols

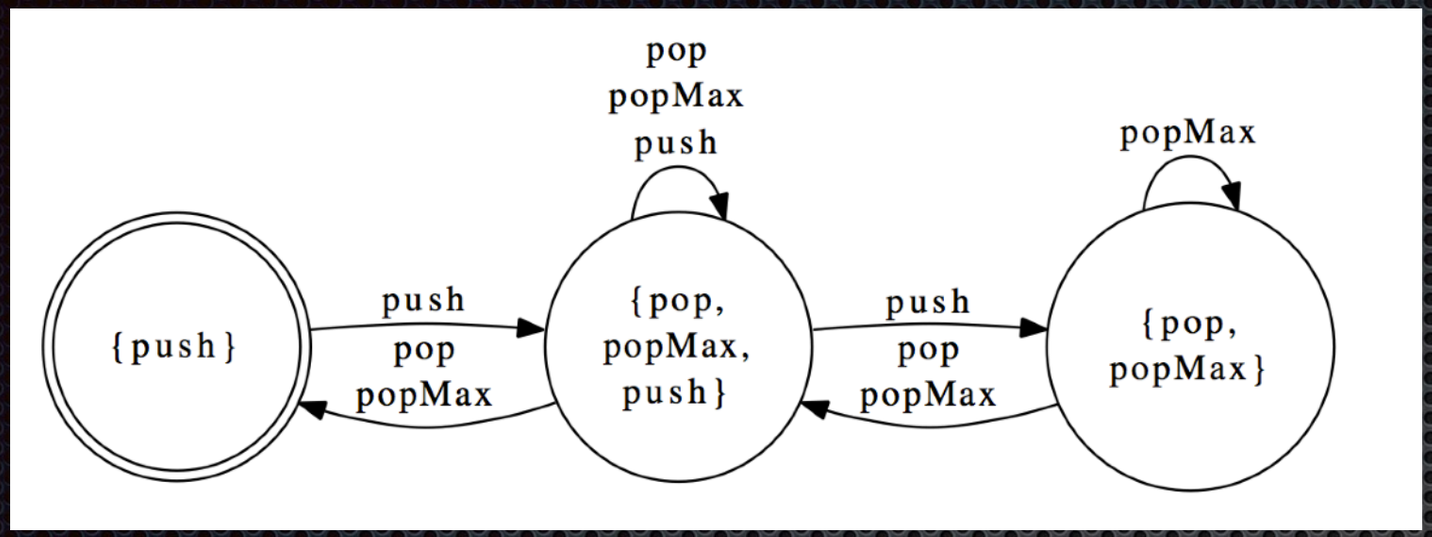
- Limitations
  - Requires clauses needed
  - Invariant clauses needed
  - Parameter splitting needed

What abstractions are “good” for supporting validation of complex behaviour?

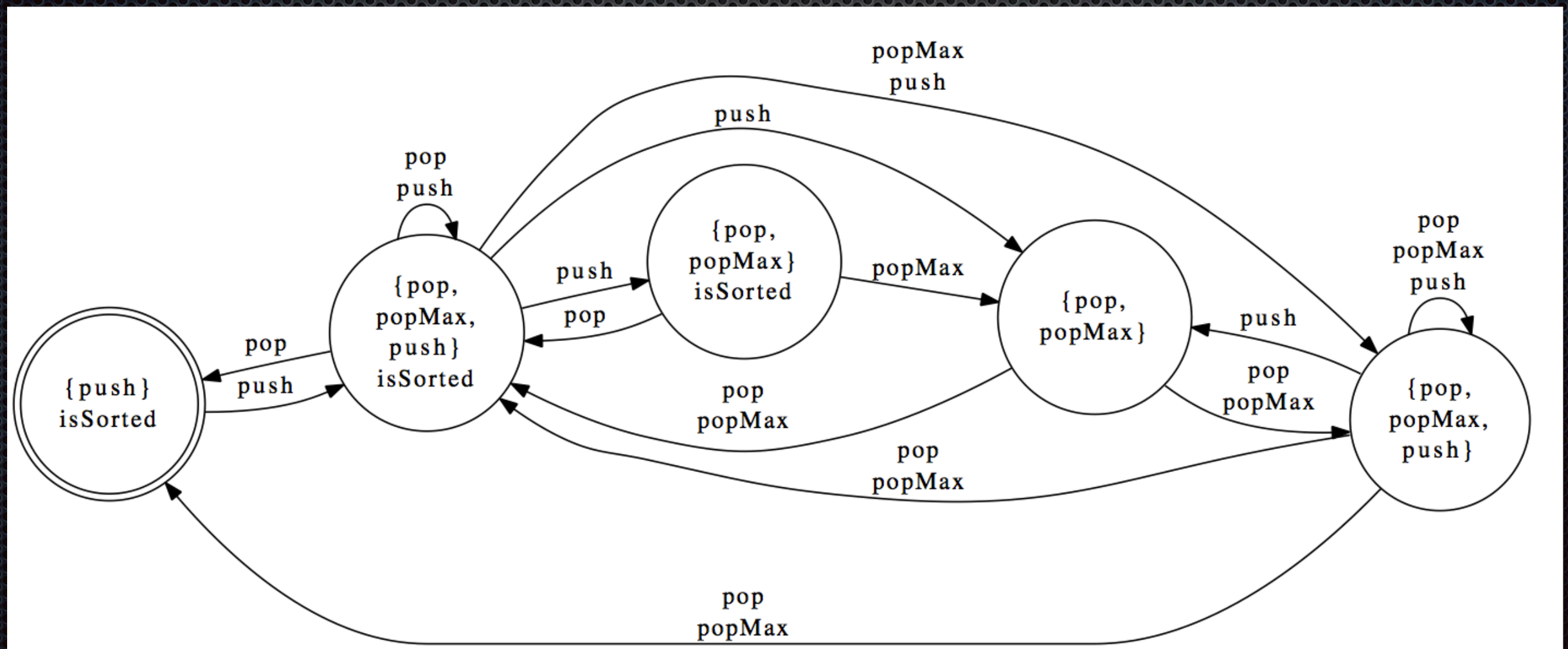
*Enabledness Preserving Abstractions*

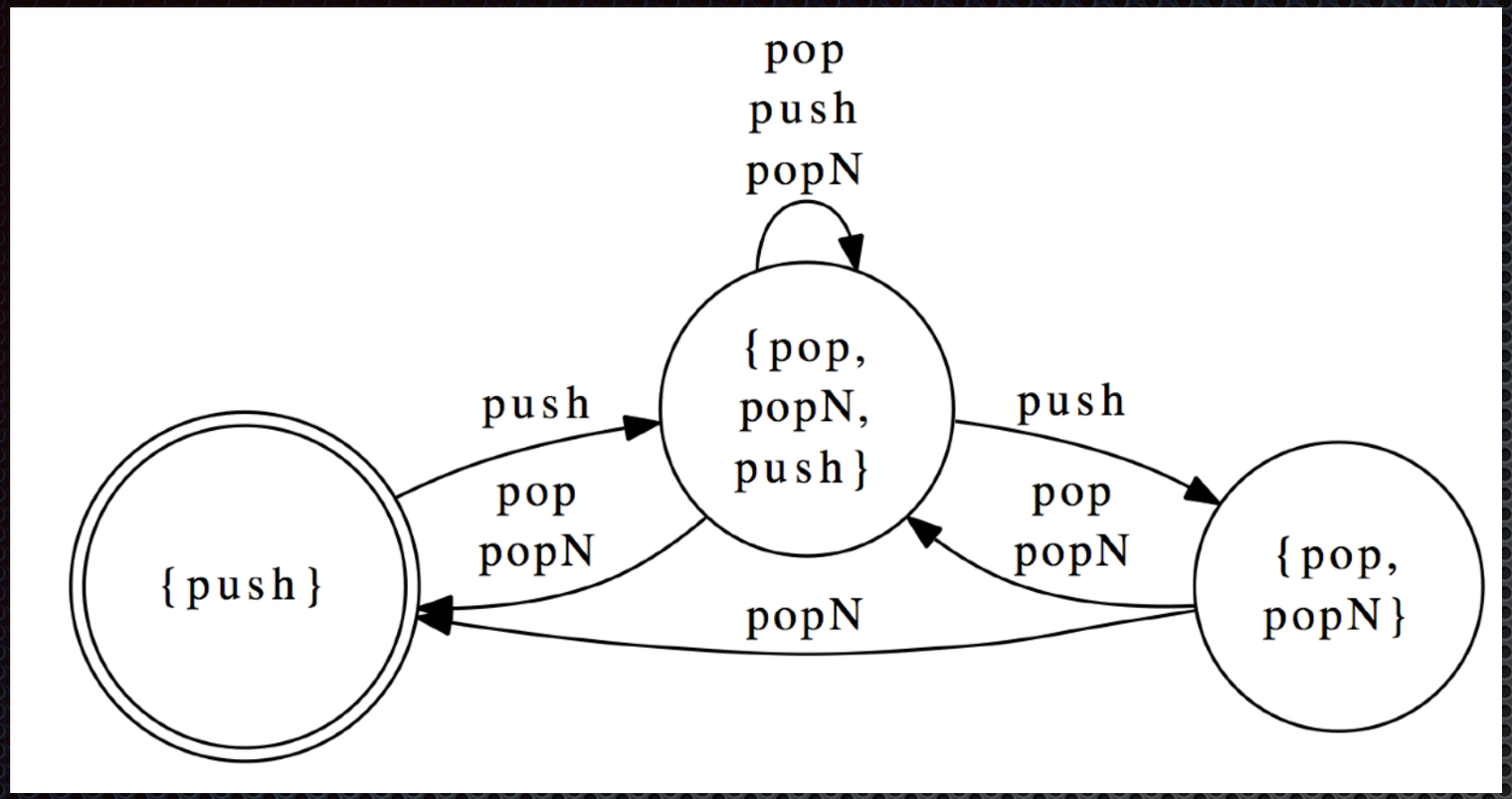
Is there anything better? or complementary?

*Sure! Zoom facility? Return values?*

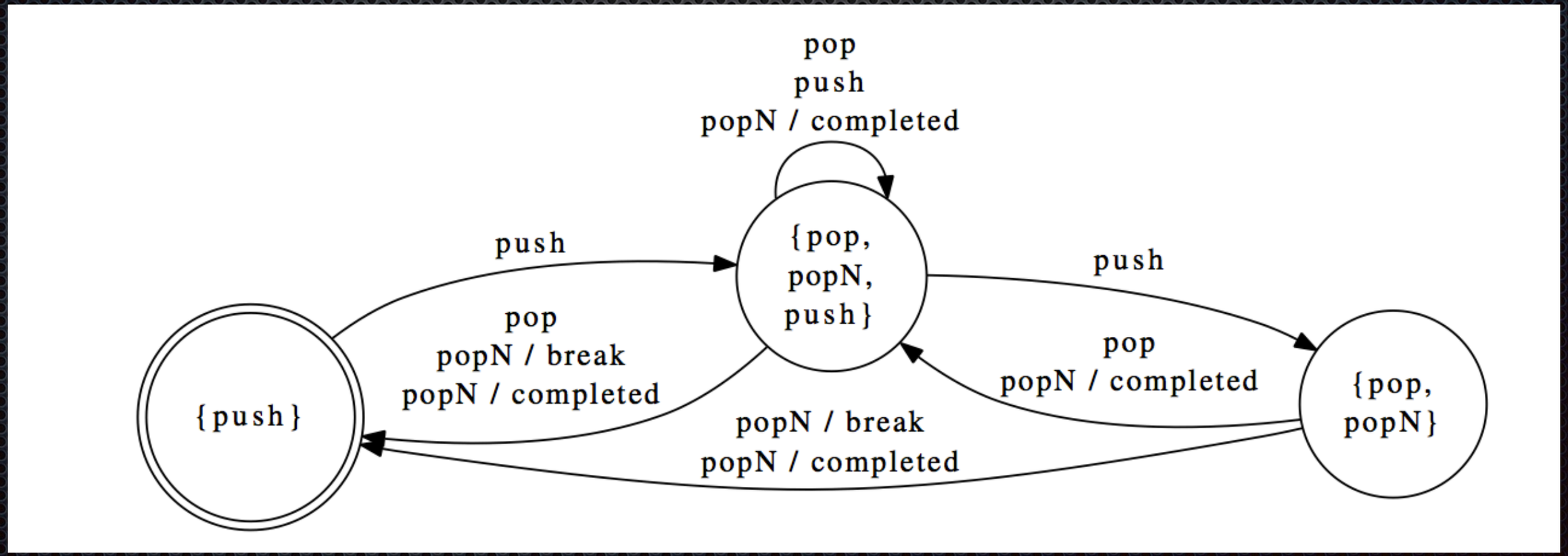


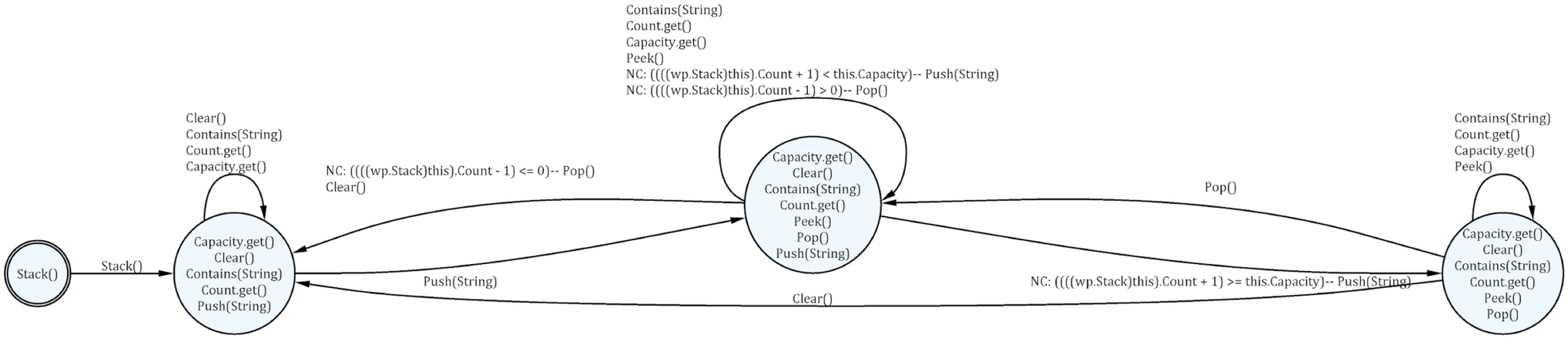

# Refined Stack with “Dummy Operations”



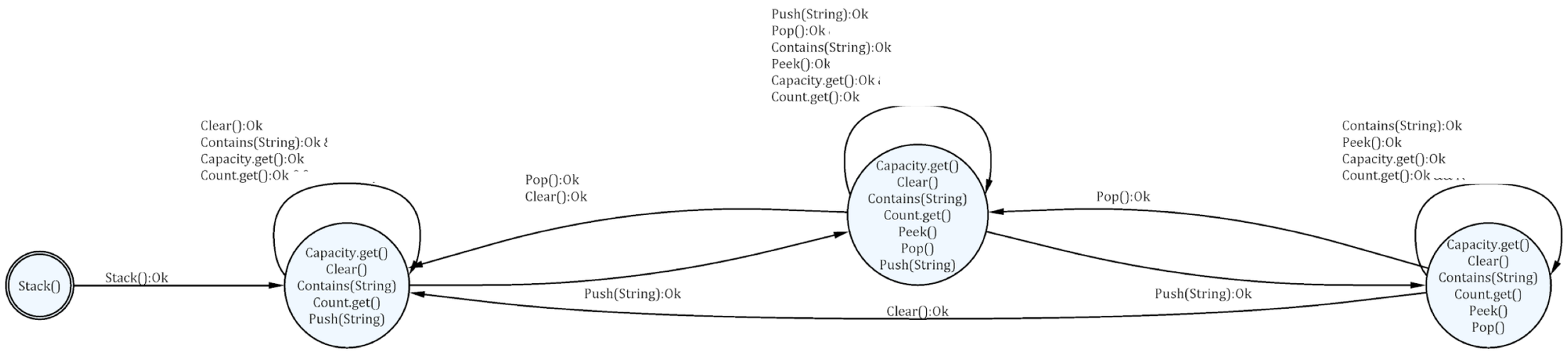



Refined Stack with Return Values






# Enabledness-Preserving Abstractions with Input-Conditions (EPA-I)




# Enabledness-Preserving Abstractions with Output-Conditions (EPA-O)