A Case Study in Abstract Testing

Verifying C String Manipulation

Nurit Dor and Michael Rodeh

http://www.cs.tau.ac.il/~nurr

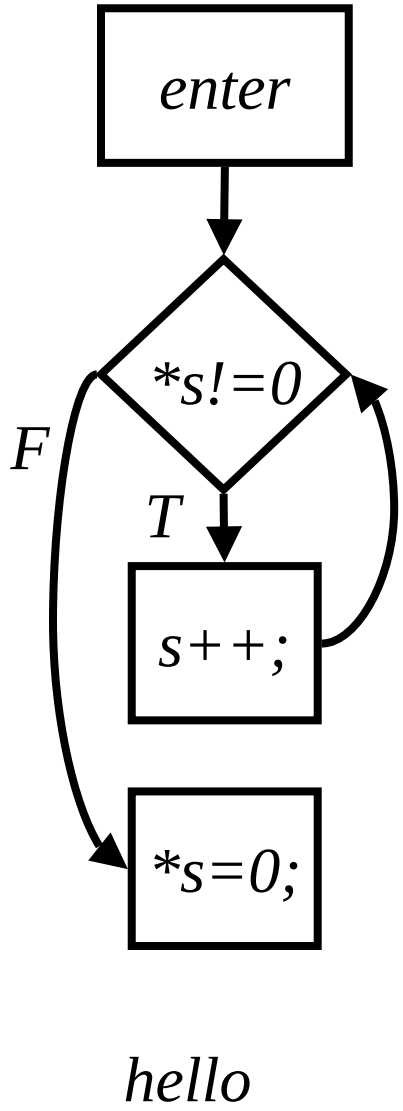
Mooly Sagiv

A Motivating Example

```
/* from web2c [strpascal.c] */
    void foo(char *s)
    {
        while ( *s != ‘ ‘ )
                s++;
        *s = 0;
    }
```

Unit Testing

```
void foo(char *s) {
   while ( *s != ‘ ‘ )
                s++;
        *s = 0;
    }
void main() {
    char temp[]=“hello world”;
    foo(temp);
    printf(“%s”, temp);
 }
```



Abstract Testing

```
void foo(char *s)
{
    while ( *s != ‘ ‘ )
        s++;
    *s = 0;
}

void main() {
    char temp[]=“hello-world”;
    foo(temp);
    printf(“%s”, temp);
}
```

Cleanness is potentially violated:
offset(s) ≥ alloc(base(s))

Abstract Testing

```
void foo(char *s)
{
    while ( *s != ‘ ‘ && *s != 0)
            s++;
    *s = 0;
}
```

Cleanness is potentially violated:
oftset(s) ≥alloc(base(s))

Abstract Testing

```
void foo(char *s)
requires string(s)
{
    while ( *s != ‘ ‘  && *s != 0)
            s++;
    *s = 0;
}
```

No cleanness violation can occur

Testing can only show the presence of errors

Plan

- ✓ Motivating example
- What is abstract testing (interpretation)
- CSSV: C Static String Verifier
  - Verifies cleanness of string manipulation
- Other abstract testing projects
  - 3VMC: 3 Valued Model Checker
    - Verifies Safety Properties in Concurrent Java Programs

Collecting Testing Procedure

1. Explore all concrete states arising at each program point
2. Check precondition of every statement on all states

- *Sound: every safety violation is detected*
- *Complete: every bug reported is real*

- *Very expensive*
- *Usually does not terminate*

Abstract Testing (Interpretation) Cousot and Cousot 1976

1. Explore all abstract descriptions of concrete states arising at each program point
2. Conservatively check precondition of every statement against abstract description
   - Err on the safe side

👍 *Guaranteed to terminate*
👍 *Can be made efficient*

👍 *Sound: every safety violation is detected*

👎 *Incomplete: May produce false alarms*

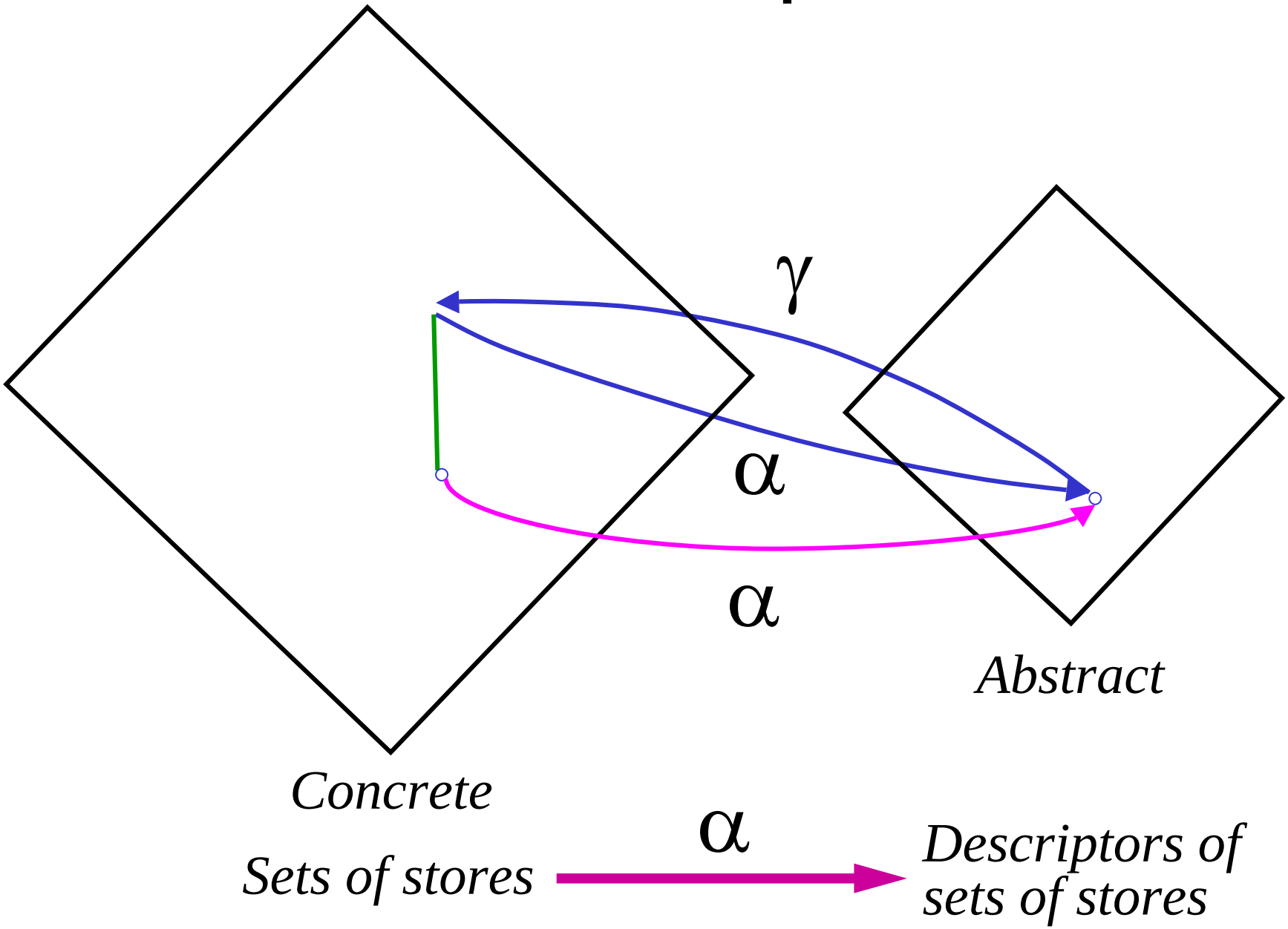
Even/Odd Abstract Testing

- Determine if an integer variable is even or odd at a given program point

Example Program

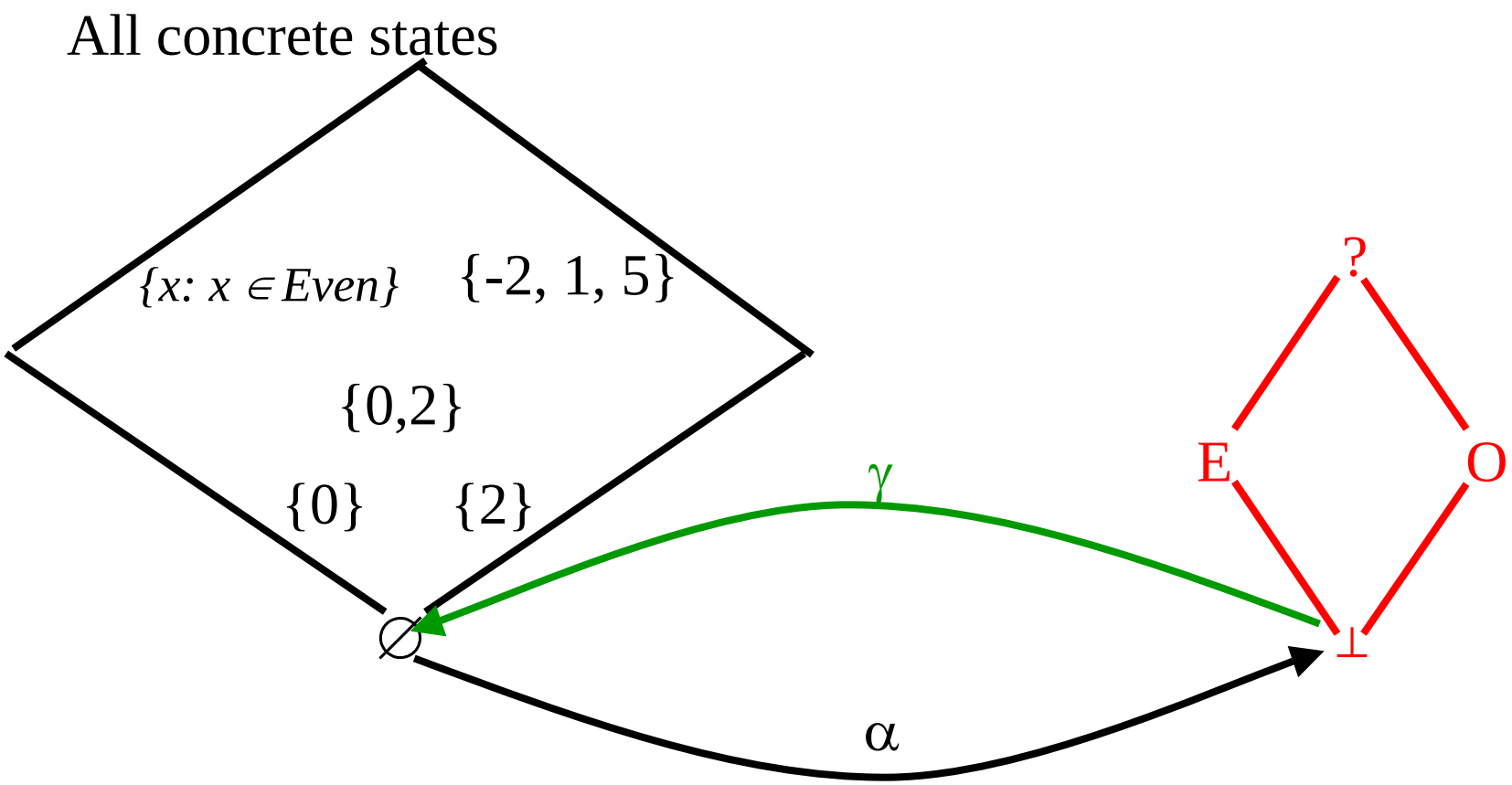
```
/* x=? */

while  (x !=1)  do {  /* x=? */
        if  (x %2) == 0
/* x=E */            { x := x / 2; }        /* x=? */
                 else
/* x=O */            { x := x * 3 + 1;      /* x=E */
                       assert (x %2 ==0); }
}
/* x=? */
```

Abstract Interpretation

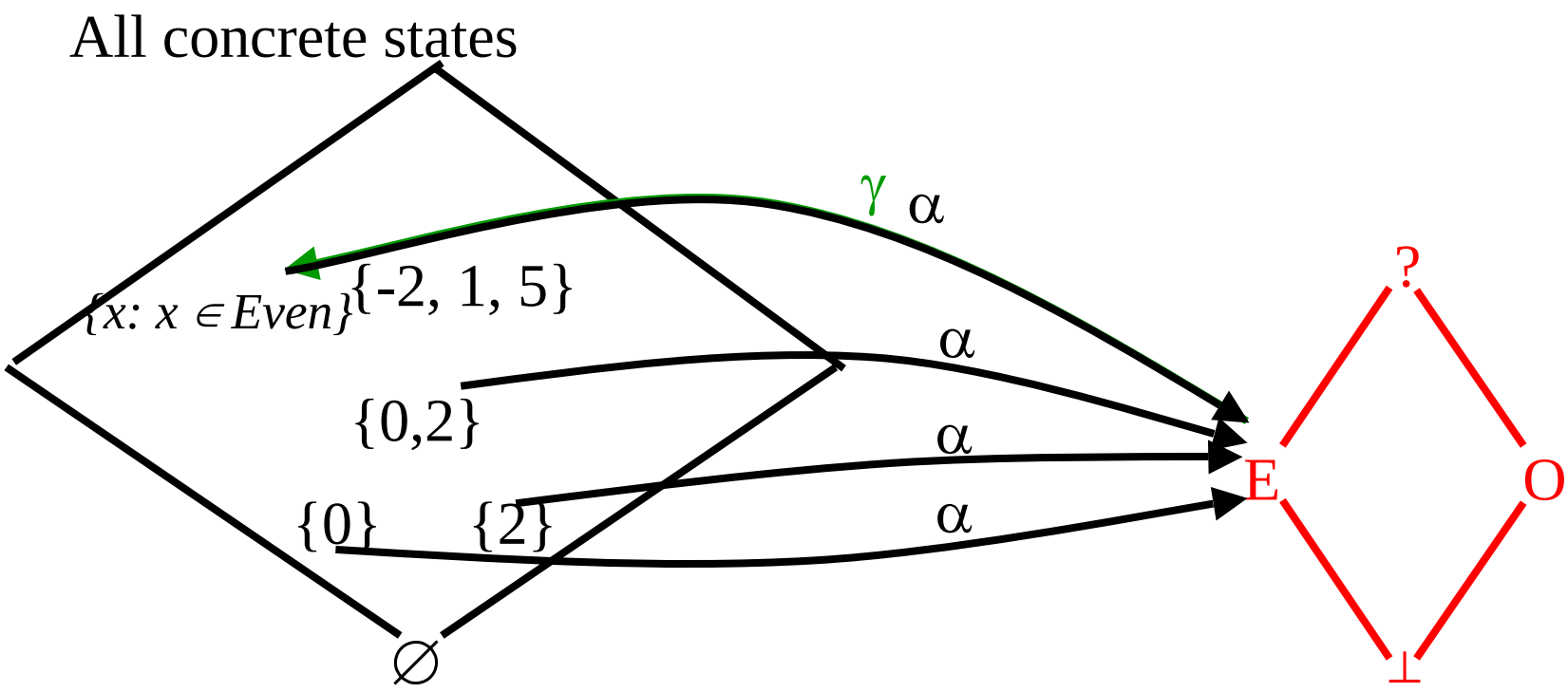


Sets of stores $\xrightarrow{\alpha}$ *Descriptors of sets of stores*

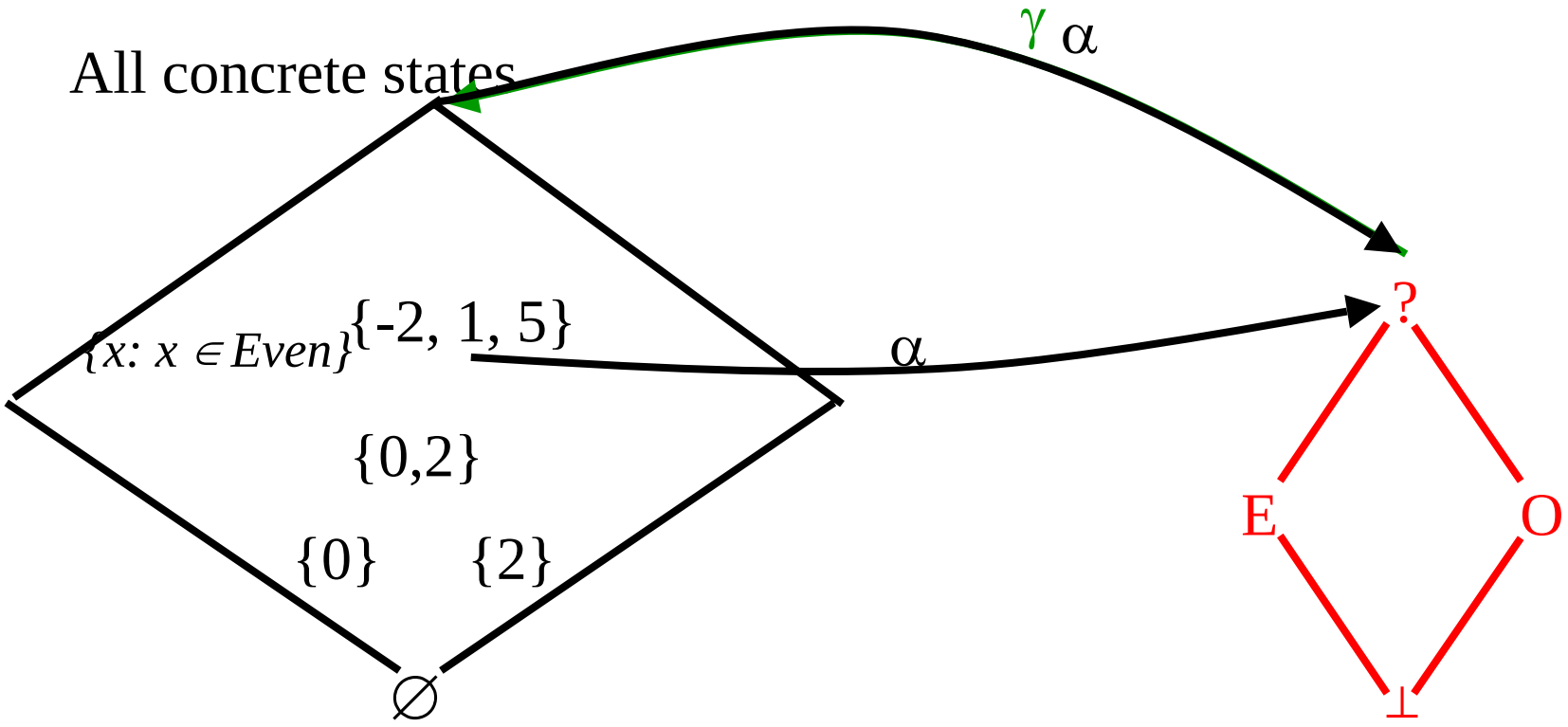
Odd/Even Abstract Interpretation



Odd/Even Abstract Interpretation



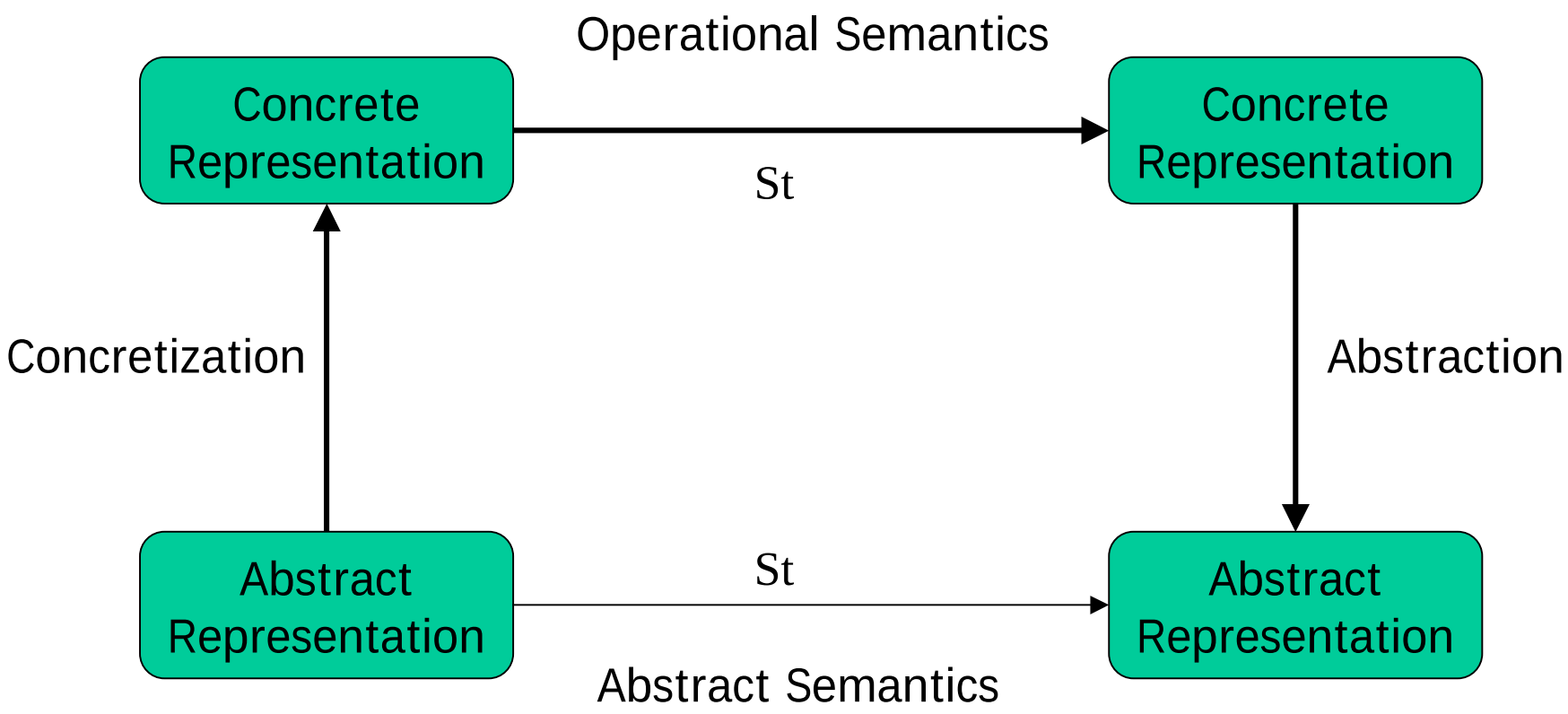
Odd/Even Abstract Interpretation



Example Program

```
while  (x !=1)   do {
        if   (x %2) == 0
                        { x := x / 2; }
                else
   /* x=O */            { x := x * 3 + 1;      /* x=E */
                          assert (x %2 ==0); }
}
```

(Best) Abstract Transformer



Concrete and Abstract Interpretation

+	0	1	2	3	…
0	0	1	2	3	…
1	1	2	3	4	…
2	2	3	4	5	…
3	3	4	5	6	…
□	□	□	□	□	

*	0	1	2	3	…
0	0	0	0	0	…
1	0	1	2	3	…
2	0	2	4	6	…
3	0	3	6	9	…
□	□	□	□	□	

$+'$	?	O	E
?	?	?	?
O	?	E	O
E	?	O	E

$*'$	?	O	E
?	?	?	E
O	?	O	E
E	E	E	E

Challenges in Abstract Testing

- Finding appropriate program semantics (runtime)
- Designing abstract representations
  - What to forget
  - What to remember
    - Summarize crucial information
  - Handling loops
  - Handling procedures
- Scalability
  - Large programs
  - Missing source code
- Minimal false alarms

Runtime vs. Abstract Testing

	Runtime	Abstract
Effectiveness	Missed Errors	False alarms
		Locate rare errors
Cost	Proportional to program's execution	Proportional to program's size

CSSV Goals

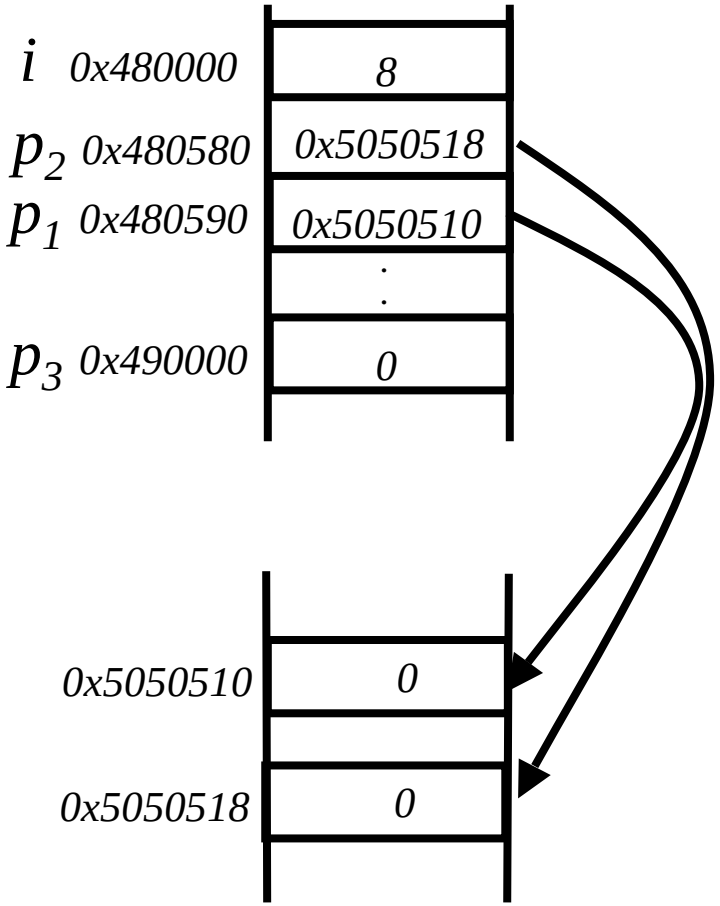
- Check cleanness
  - “Undefined ANSI-C semantics”
- Verify absence of
  - Out of bound references
  - Unsafe pointer arithmetic
  - References beyond null termination
  - Unsafe library calls

Standard C Semantics

p_1 =*calloc(10);*

$p_2 = p_1 + i;$

p_3= *p_2 ;

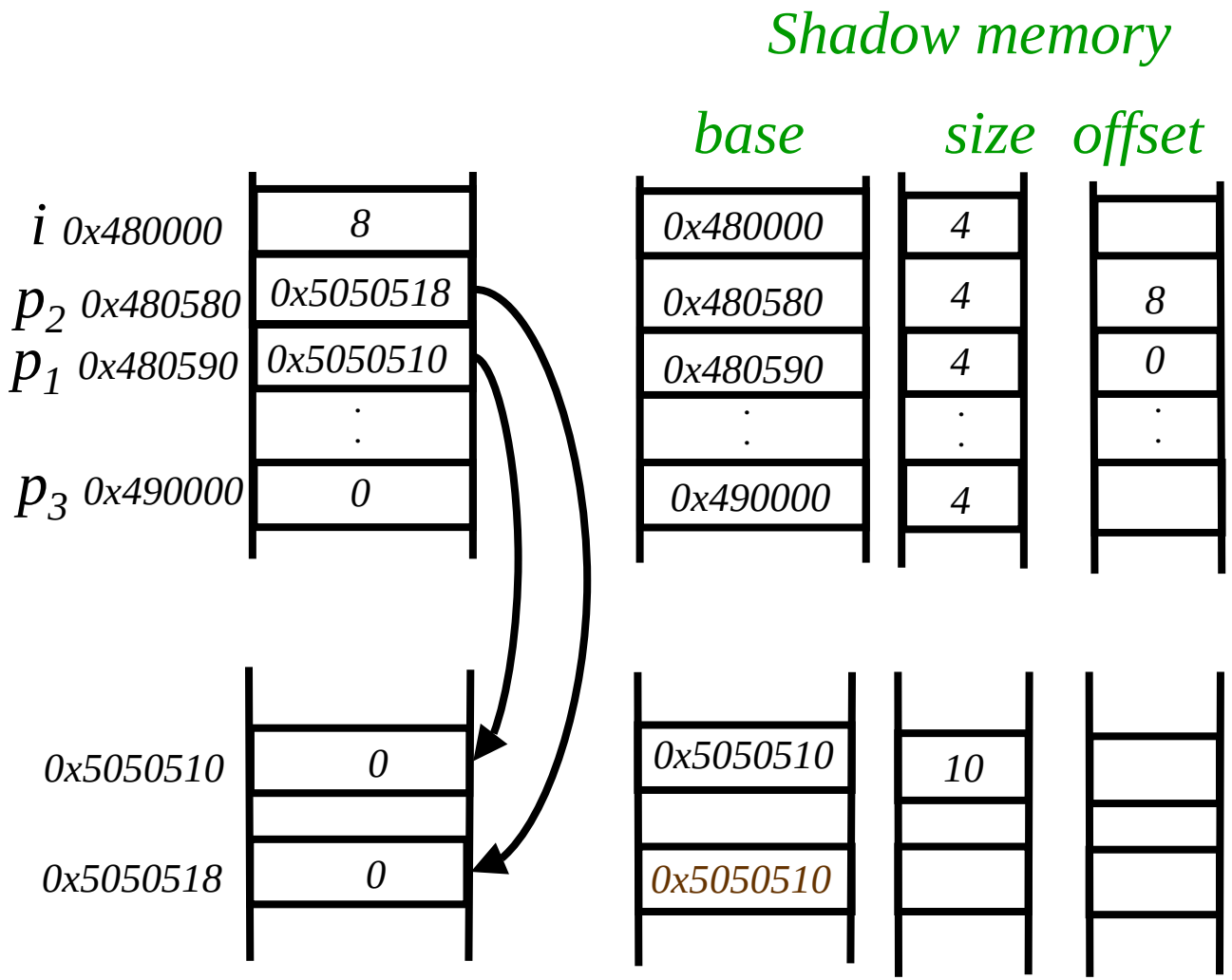


Instrumented Semantics

p_1 = *calloc(10);*

$p_2 = p_1 + i;$

$p_3 = *p_2;$



CSSV's Abstraction

- Ignore
  - Memory Addresses
  - Actual pointer values
- *Dynamic Size*
- *Dynamic Offsets*

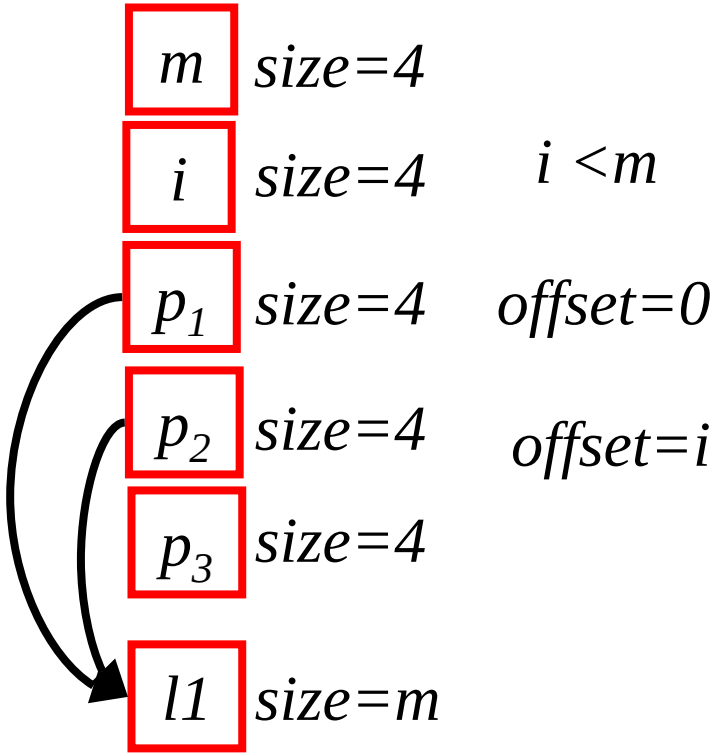
- Track
  - Allocation sites
  - Which pointer may point to memory addresses (allocation sites)
- *Fixed sizes*
- *Fixed offsets*
- *Fixed index variables*
- *Integer Relationships between offsets, sizes, and integer variables*

Abstract Semantics

$l1: p_1 = calloc(m);$

$p_2 = p_1 + i;$

$p_3 = *p_2;$



Scalability

- Cheap whole program pointer analysis
  - Flow insensitive
- Use modular analysis

Procedure Calls – Contracts

```
char* strcpy(char* dst, char *src)
    requires    ( string(src) ∧
                  alloc(dst) > len(src)
                )

    mod         len(dst), is_nullt(dst)
    ensures     ( len(dst) = = pre@len(src) ∧
                  return = = pre@dst
                )
```

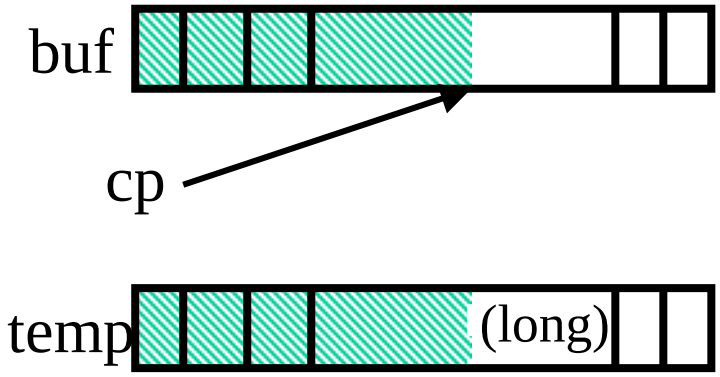
Contracts – insert_long()

```
/* insert_long.c */
#include "insert_long.h"
char buf[BUFSIZ];
char * insert_long (char *cp) {
   char temp[BUFSIZ];
   int i;
   for (i=0; &buf[i] < cp; ++i){
      temp[i] = buf[i];
      }
    strcpy (&temp[i],"(long)");
    strcpy (&temp[i + 6], cp);
    strcpy (buf, temp);
    return cp + 6;
}
```

```
char * insert_long(char *cp)
   requires( string(cp) ∧
            buf ≤ cp < buf  + BUFSIZ
            )
   mod cp.strlen
   ensures (
      len(cp) = = pre[len(cp) + 6]
            ∧
      return_value = = cp + 6 ;
            )
```

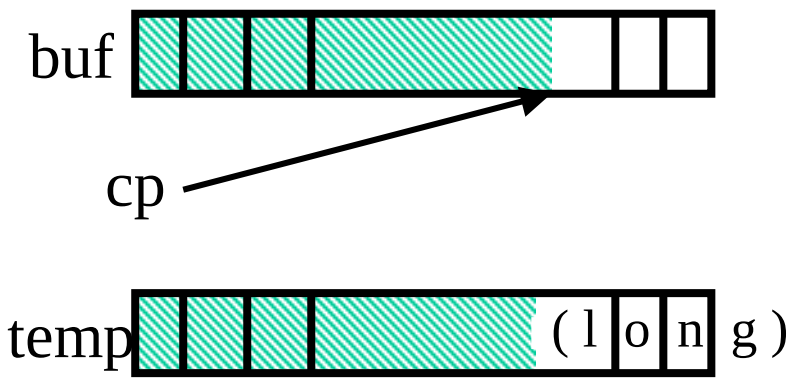
Abstract Testing insert_long()

```
/* from web2c [fixwrites.c] */
#define BUFSIZ 1024
char      buf[BUFSIZ];

char insert_long(char *cp)
{
      char    temp[BUFSIZ];
      ...
      for (i = 0; &buf[i] < cp ; ++i)
            temp[i] = buf[i];

      strcpy(&temp[i],"(long)");
      strcpy(&temp[i+6],cp);
      ...
```



Abstract Testing insert_long()

```
/* from web2c [fixwrites.c] */
#define BUFSIZ 1024
char      buf[BUFSIZ];

char insert_long(char *cp)
{
      char    temp[BUFSIZ];
      ...
      for (i = 0; &buf[i] < cp ; ++i)
            temp[i] = buf[i];

      strcpy(&temp[i],"(long)");
      strcpy(&temp[i+6],cp);
      ...
```

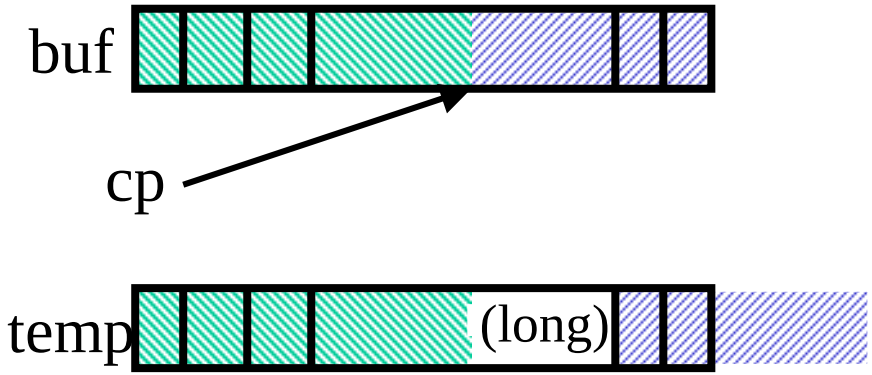


Cleanness is potentially violated:

$7 + \text{offset}(cp) \geq BUFSIZ$

Abstract Testing insert_long()

```
/* from web2c [fixwrites.c] */
#define BUFSIZ 1024
char      buf[BUFSIZ];

char insert_long(char *cp)
{
      char    temp[BUFSIZ];
      ...
      for (i = 0; &buf[i] < cp ; ++i)
            temp[i] = buf[i];

      strcpy(&temp[i],"(long)");
      strcpy(&temp[i+6],cp);
      ...
```



Cleanness is potentially violated:

offset(cp)+7 +len(cp) ≥ BUFSIZ

7 + offset (cp) < BUFSIZ

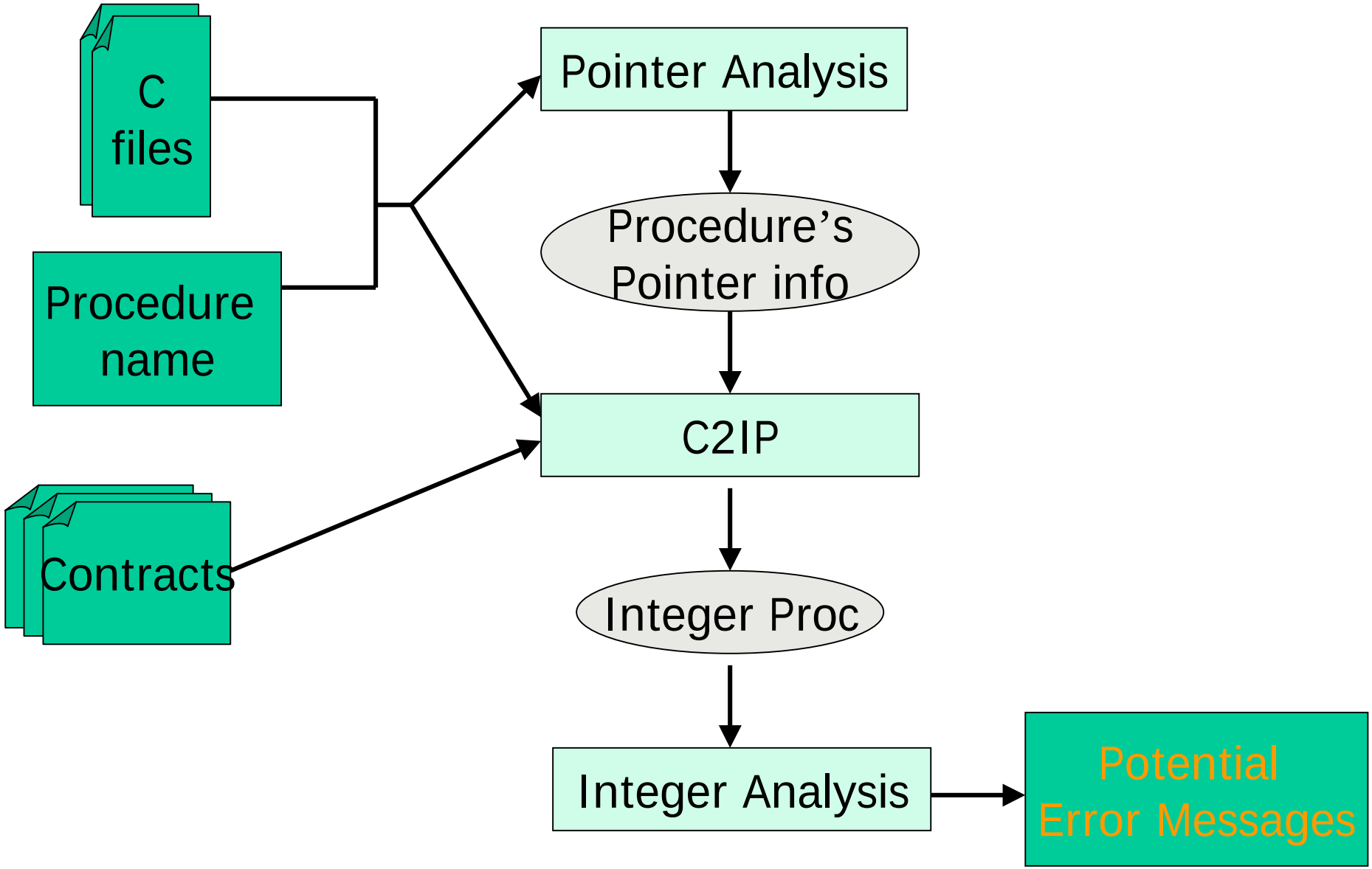
Advantages of Procedure Contracts

- Modular analysis
  - Not all the code is available
  - Enables more expensive analyses
- User control of the verification
  - Detect errors at point of logical error
  - Improve the precision of the analysis
  - Check additional properties
    - Beyond ANSI-C

Specification and Soundness

- All errors are detected
- Violation of procedure’s precondition
  - Call
- Violation of procedure's postcondition
  - Return
- Violation of statement’s precondition
  - ...a[i]...
- But what about false alarms?

CSSV – Technical overview



Implementation

- Handles full C
- Using:
  - ASToolKit [Microsoft]
  - GOLF [Microsoft – Manuvir Das]
  - New Polka [IMAG - Bertrand Jeannet]
- Main steps:
  - Simplifier (Greta Yorsh)
  - Pointer analysis
  - C2IP
  - Integer Analysis

Empirical Results

- Verified string library form Airbus with 6 false alarms
- Found 8 errors in another string intensive application with 2 false alarms

Other Abstract Testing Projects

- Polyspace
  - Compliance with ANSI C
- Absint
  - Worst case execution time
  - Memory consumption
- Microsoft Research
  - ESP
  - SLAM
  - Vault
  - Behave
- Berkley
  - CCURE
  - Cqual
- TAU
  - 3VMC Eran Yahav
    - POPL 01
    - ESOP 03

3VMC Goals

- Verification of concurrent Java programs
- Support the following
  - Interleaving concurrency-model
  - Dynamic allocation/deallocation of objects
  - Dynamic allocation/deallocation of threads

Java Concurrency

- *Threads and locks are just dynamically allocated objects*
- *synchronized* implements mutual exclusion
- *wait*, *notify* and *notifyAll* coordinate activities across threads

Java Concurrency Challenges

- Dynamic allocation
- Data and control are strongly related
  - Thread-scheduling info may require understanding of heap structure (e.g., scheduling queue)
  - Heap analysis requires information on thread scheduling

```
Thread t1 = new Thread();
Thread t2 = new Thread();
…
if (…) t = t1 else t = t2;
t.start();
```

Example - Mutual Exclusion

```
l_0: while (true) {
l_1:      synchronized(sharedLock) {
l_C:      // critical actions
l_2:      }
l_3: }
```

Two threads: $(pc_1, pc_2, lockAcquired_1, lockAcquired_2)$

- Allocate new lock ?
- Allocate new thread ?

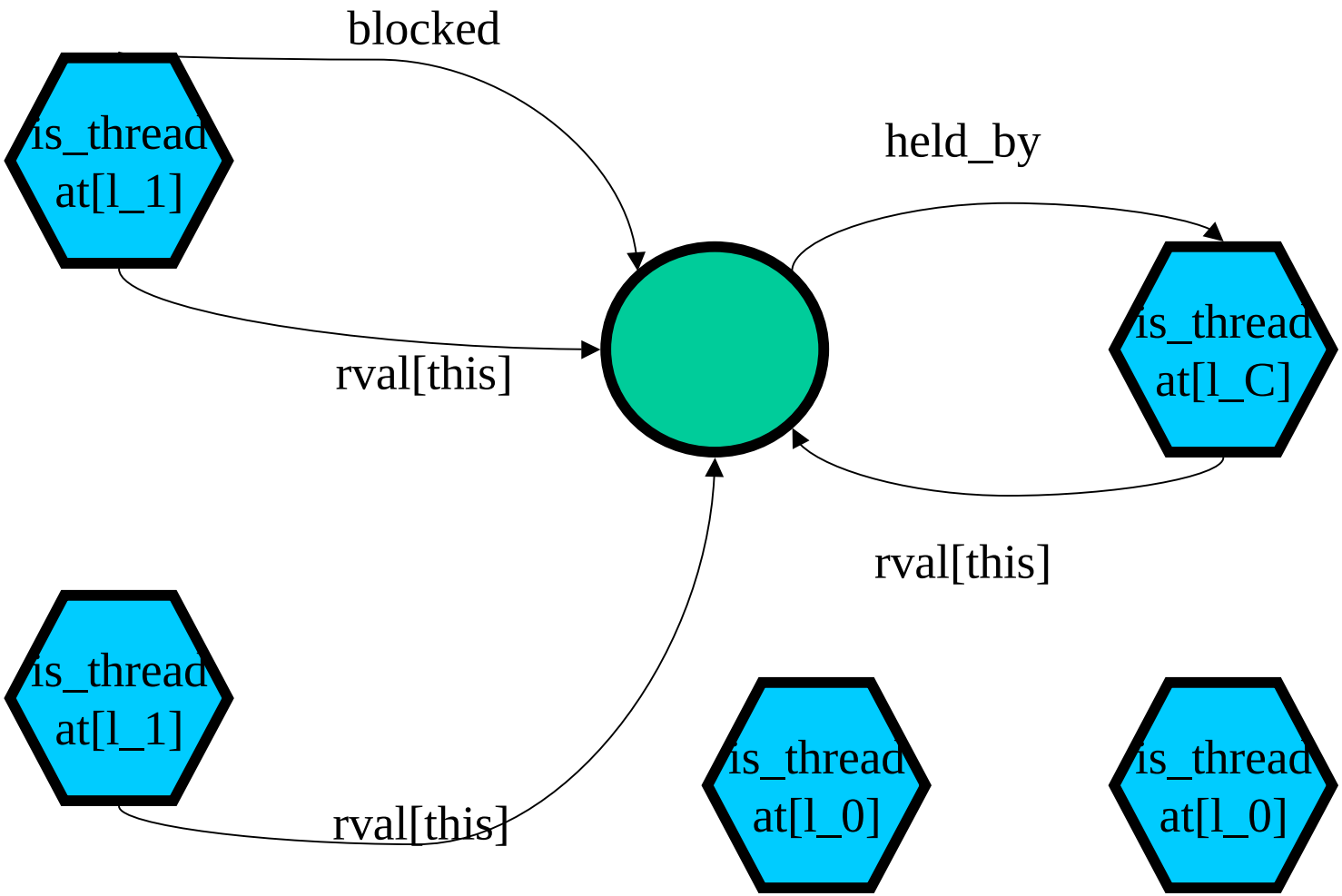
Configurations

- A program configuration encodes:
  - global store
  - program-location of every thread
  - status of locks and threads
- First-order logical structures used to represent program configurations
- Operational semantics expressed using Formulas
  - Interleaving model of concurrency

Configurations

- Predicates model properties of interest
  - is_thread(t)
  - { at[lab](t) : lab $\in$ Labels }
  - { rval[fld](o_1,o_2) : fld $\in$ Fields }
  - held_by(l,t)
  - blocked(t,l)
  - waiting(t,l)
- Can use the framework with different predicates

Concrete Configurations



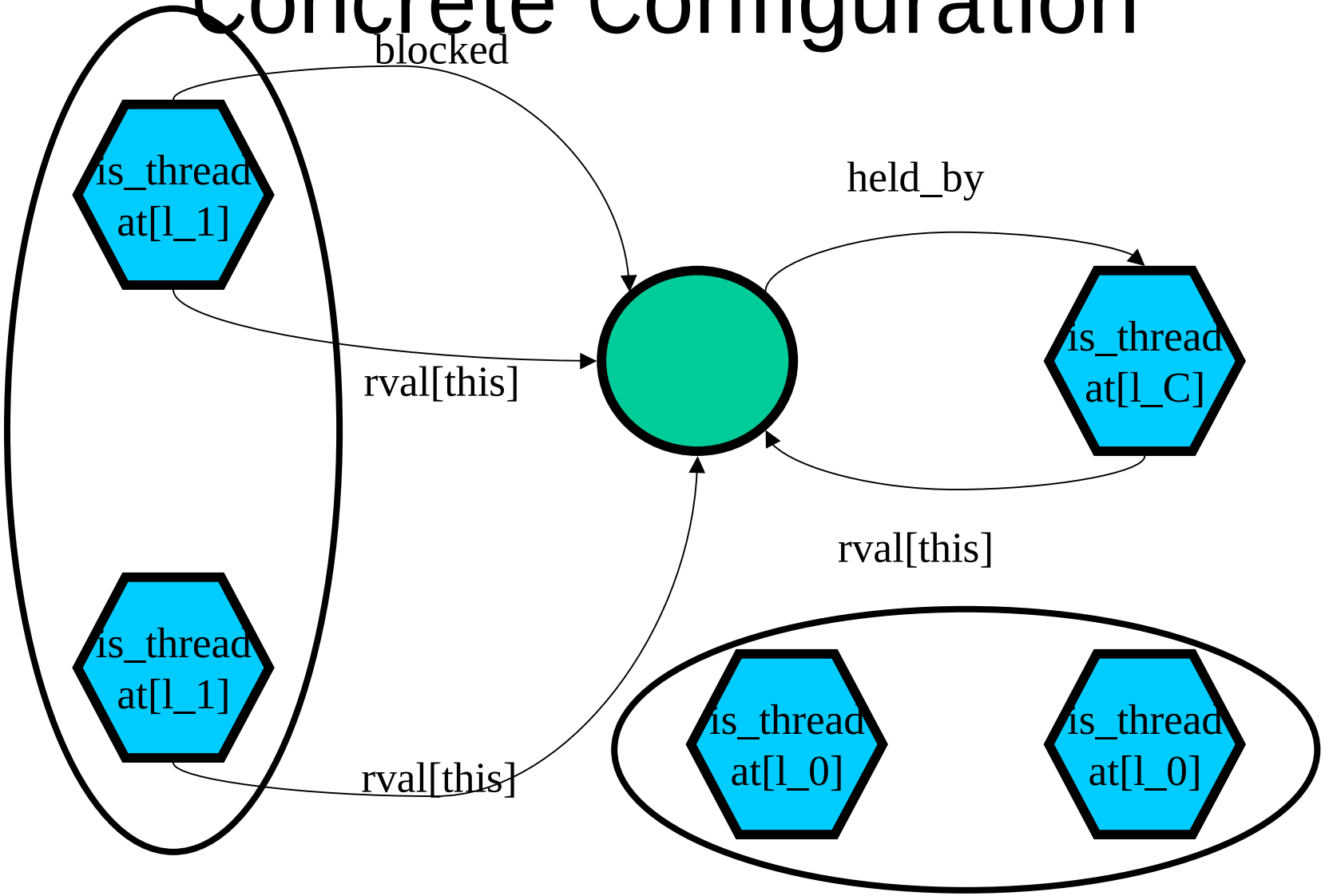
Configurations

- Program control-flow is not separately represented
- Program location for each thread is encoded inside the configuration
  - $\{ \text{at}[\text{lab}](t) : \text{lab} \in \text{Labels} \}$

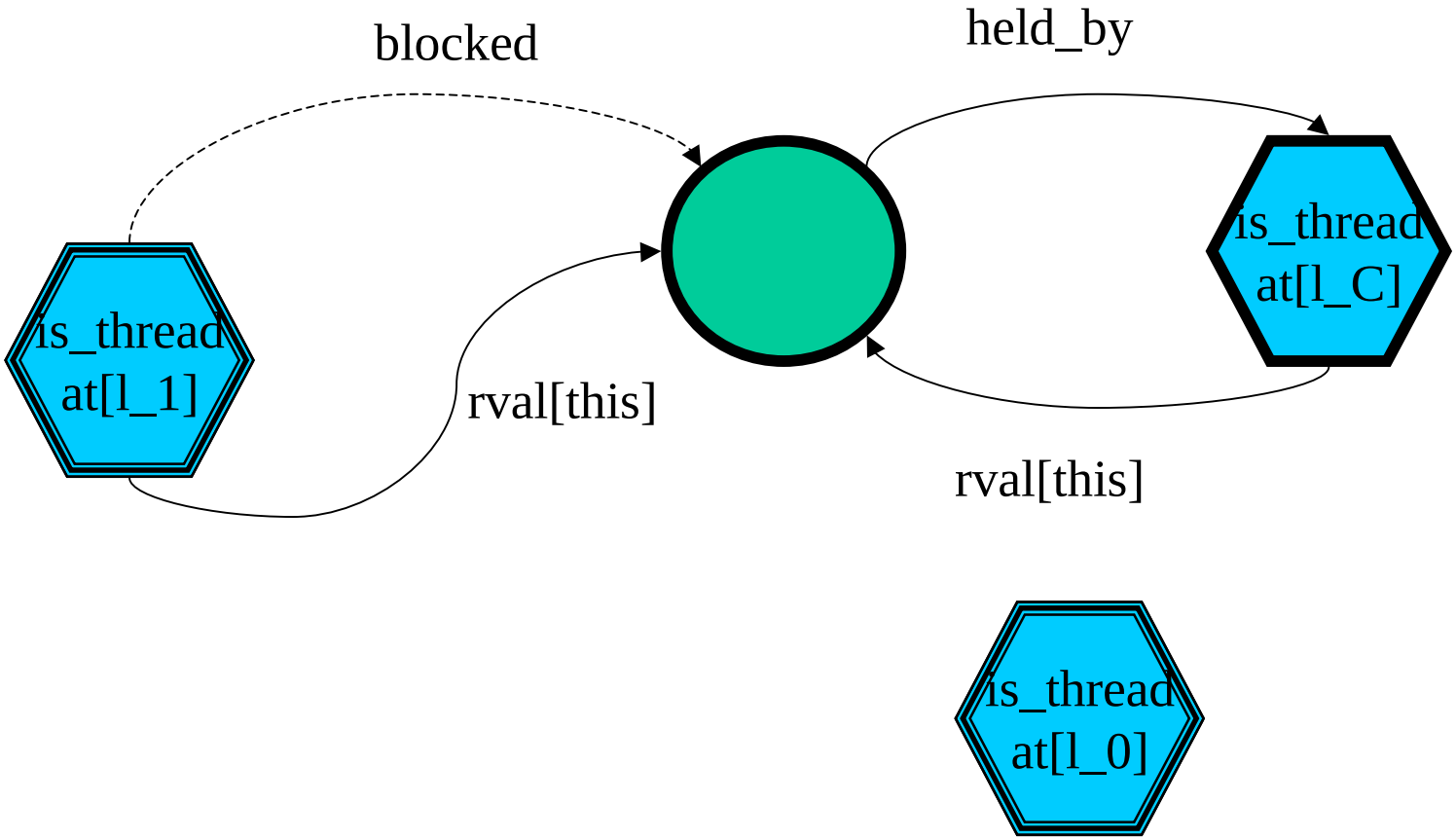
Abstract Configurations

- First-order 3-valued logical structures are used to represent abstract program configurations
- 3-valued logic
  - 1 = true
  - 0 = false
  - 1/2 = unknown
  - A join semi-lattice, $0 \sqcup 1 = 1/2$

Concrete Configuration



Abstract Configuration



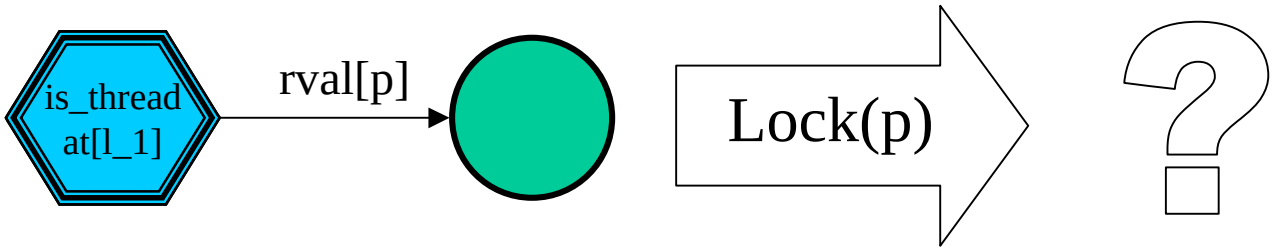
Canonic Abstraction

- Merge all nodes with the same unary predicate values into a single summary node
- Join predicate values
- Convert a configuration of arbitrary size into a 3-valued abstract configuration of bounded size

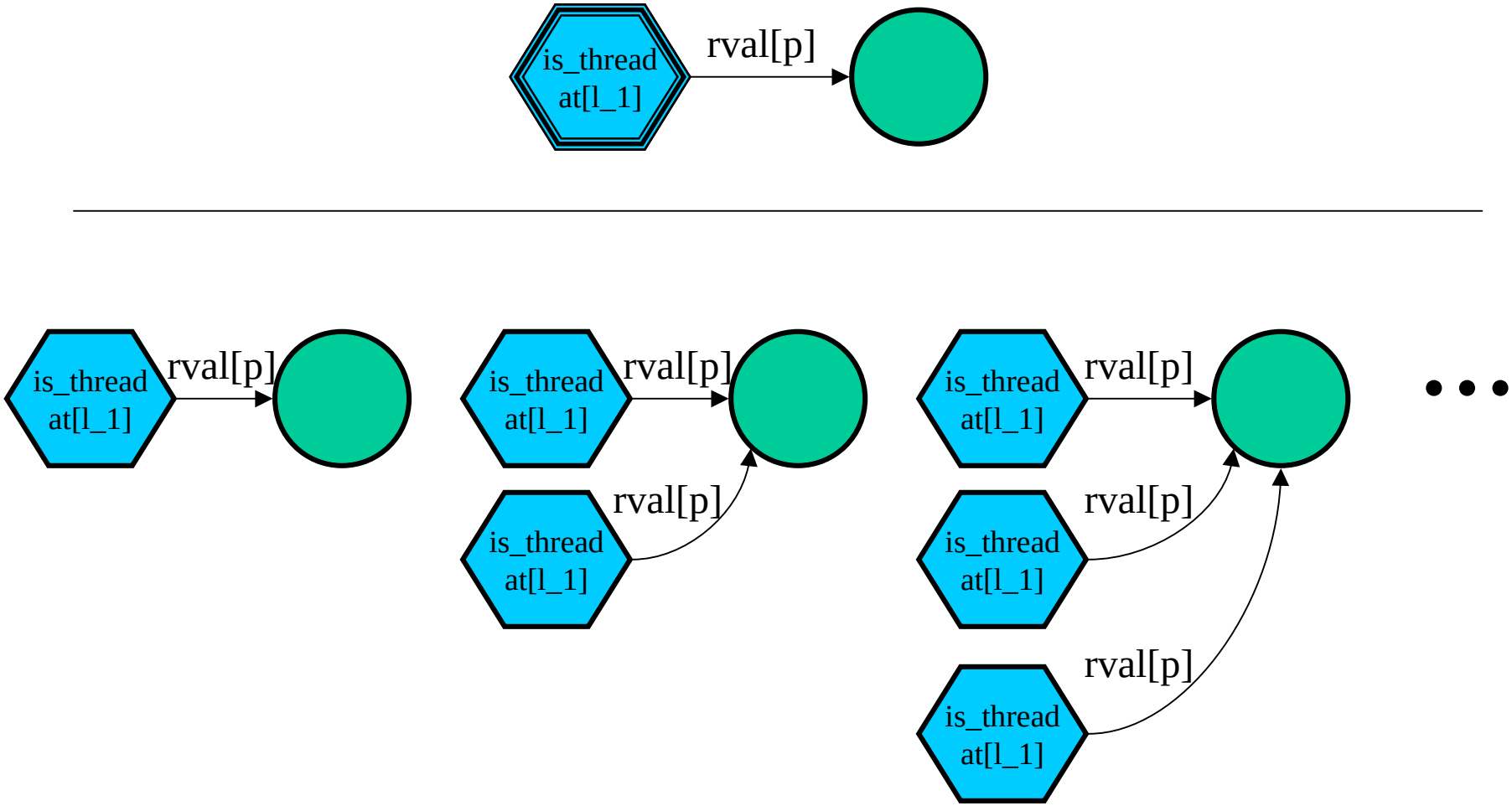
Unbounded Number of Threads

- Exploit state-space symmetry
- Previous work defined symmetry between process names (indices)
- Thread location = thread property
- Canonic names = symmetry between properties
- Anonymous thread names

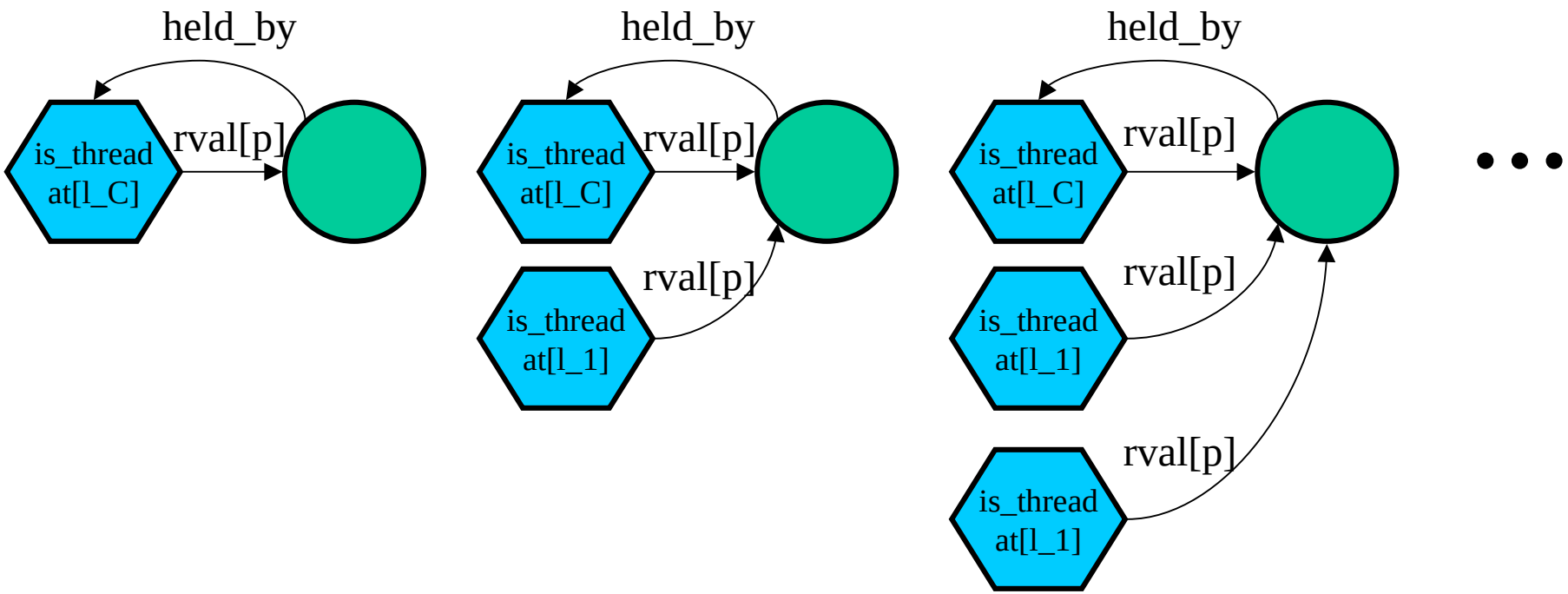
Abstract Transformers



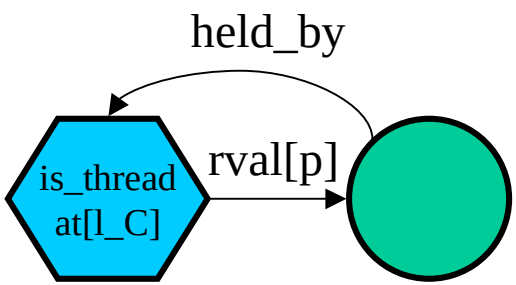
Concretization

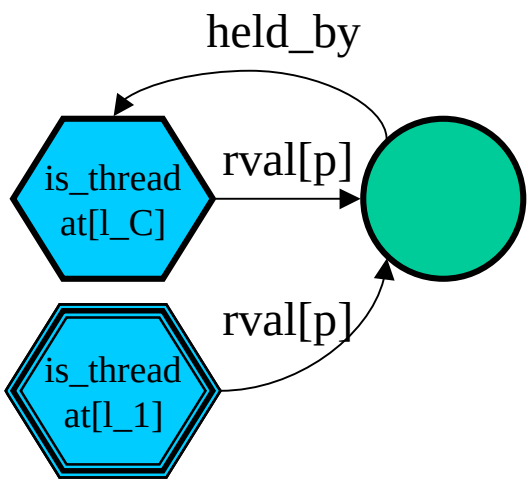


After Action

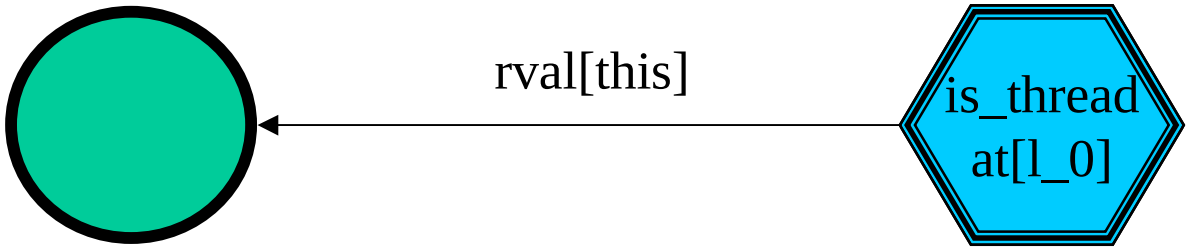


After Abstraction



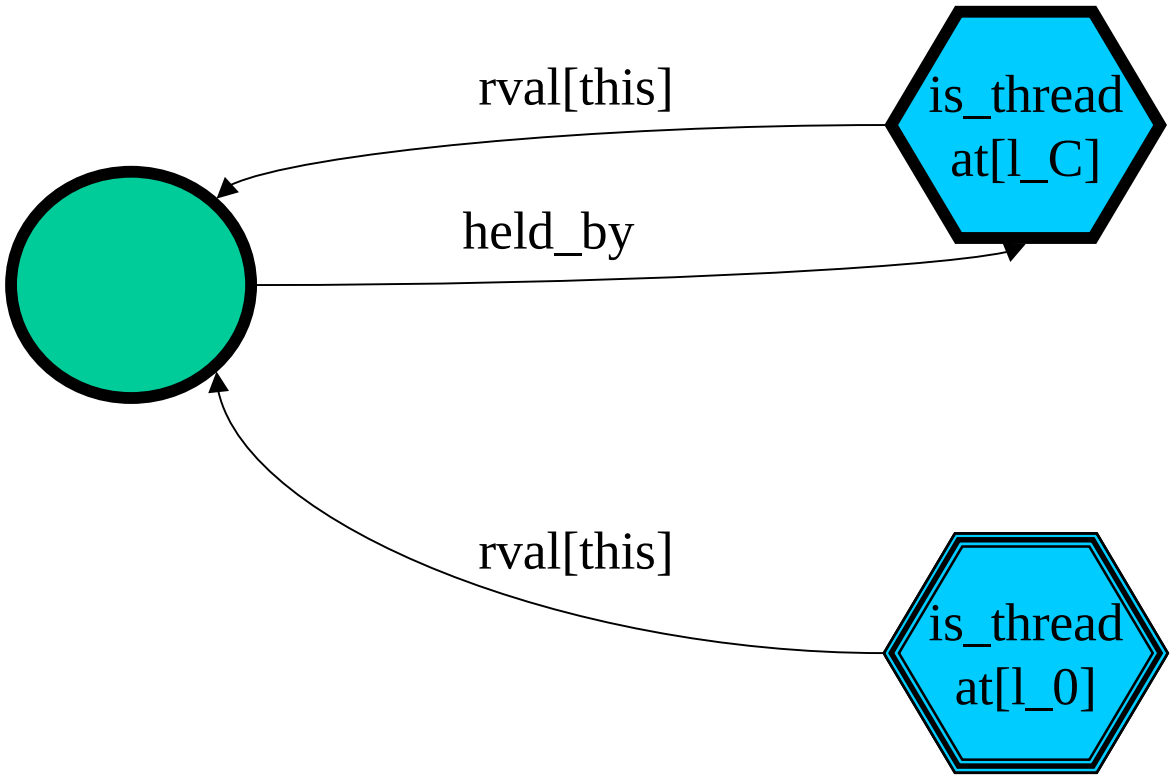


Example - Mutual Exclusion



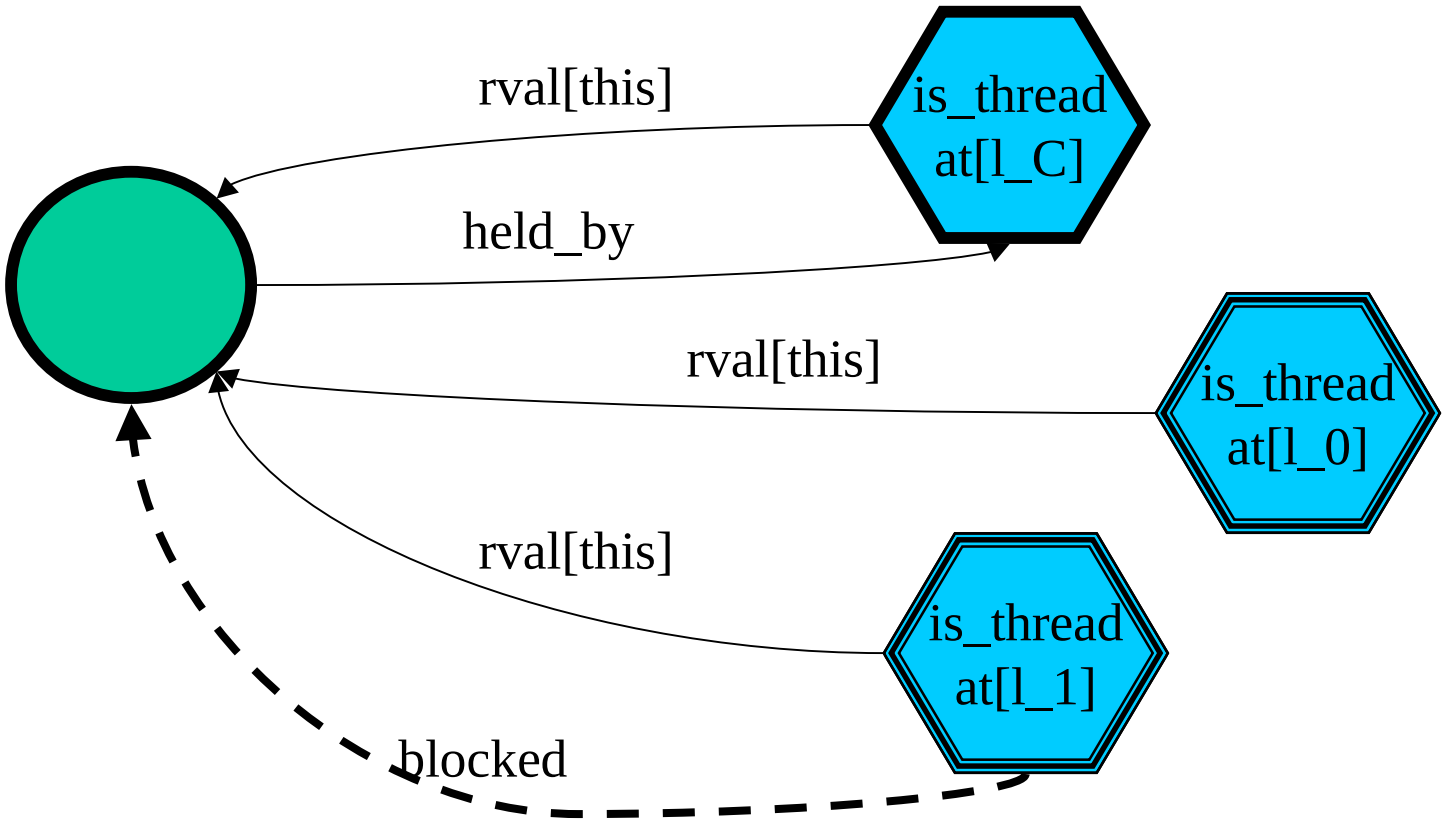
Initial configuration

Example - Mutual Exclusion



A thread enters the critical section

Example - Mutual Exclusion



Other threads may be blocked or just beginning execution

Safety Properties Revisited

- RW interference
- WW interference
- Total deadlock
- Nested monitors
- Illegal thread interactions

Prototype Implementation

- 3VMC (TVLA)
- Used to verify several small example programs
  - The apprentice challenge [Moore]
  - concurrent stack
  - queue
  - two-lock queue [PODC96]
  - dining philosophers

Conclusion

- Ambitious sound analyses
  - Appropriate operational semantics
  - Powerful abstractions
- Very few false alarms
- Scaling is an issue
  - staged analyses
  - modular analysis
  - encapsulation
  - property-guided