Software Model Checking Using Bogor

– a Modular and Extensible Model Checking Framework

3rd Estonian Summer School in Computer and System Science (ESSCaSS'04)

Slide Set 06: Bogor's Statespace Reductions

http://bogor.projects.cis.ksu.edu
http://www.cis.ksu.edu/~hatcliff/ESSCaSS04

John Hatcliff   Matthew B. Dwyer   Robby

SAnToS Laboratory, Kansas State University, USA

Support

US Army Research Office (ARO)
US National Science Foundation (NSF)
US Department of Defense
Advanced Research Projects Agency (DARPA)

Boeing
Honeywell Technology Center
IBM
Intel

Lockheed Martin
NASA Langley
Rockwell-Collins ATC
Sun Microsystems

Effective for OO Software

How do we take the well-known explicit-state model-checking algorithms and enhance them to be effective for working directly on software?

- How do we represent the state effectively?
- How do we reduce the number of paths/states explored?

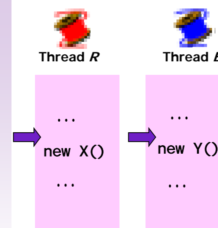
Heap Representation

Simple Idea

- Start with something like Spin
- Implement objects as records/structs
- Implement heap as an array of objects

...not a good idea!

Heap Representations



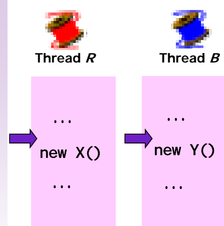
Two Possible Schedules

1. R goes first S_1 *2. B goes first*

Naïve Heap Representation

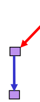
1. R goes first *2. B goes first*

Heap Representations



Two Possible Schedules

1. R goes first S_1 *2. B goes first*

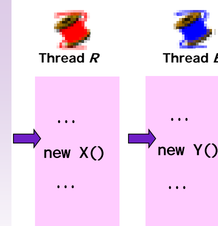


Naïve Heap Representation

X
Y

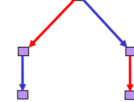
1. R goes first *2. B goes first*

Heap Representations



Two Possible Schedules

1. R goes first S_1 *2. B goes first*



Naïve Heap Representation

X	Y
Y	X

1. R goes first *2. B goes first*

Heap Representations

Thread *R* — Thread *B*

```
...
new X()
...
```

```
...
new Y()
...
```

Two Possible Schedules

1. *R goes first* — s_1 — 2. *B goes first*

We desire a single state here

But how do we design a representation that accomplishes this?

Naïve Heap Representation

X, Y — Observationally Equivalent $\cong$ — Y, X

1. *R goes first* — 2. *B goes first*

Assessment

Different thread interleavings may cause different positioning of heap objects. This will cause observationally equivalent heaps to be considered distinct states --- leading to tremendous state explosion.

garbage

Observationally Equivalent

$\cong$

garbage collection & canonical ordering on objects based on lexicographical order on field names in reachability chain

Canonical Heap (fully abstract)

Bogor’s Heap Representation

State

Heap

...transition may create new objects, garbage, etc.

...sort walks over heap, canonicalizes, and collects info

Canonical heap

Key Points...

...explicit heap representation

...after each transition, a topological sort gives heap objects a canonical order

...garbage is eliminated

...precise heap model

...precise alias information

...have access to all visited states (but, efficiently stored using collapse compression)

Heap Representation

- After each transition...
  - reachable heap objects are ordered
    - topological sort based on chains of field names
  - unreachable objects are discarded (garbage collection)
- State compression...
  - objects are held in an pool and are identified by bit patterns
  - state vector holds bit-vectors representing objects
  - means that object values can be shared across states
    - good! because in a typical transition, very little changes in the state
- Formalization of heap and thread symmetry...
  - presentation by Radu Iosif based on group theory

See *"State-space Reductions for Model-Checking Dynamic Software"* SoftMC 2003

Avoiding Equivalent Paths

Explosion of Paths

Many paths are equivalent in the sense that they cannot be distinguished by the property being checked.

Partial Order Reduction (POR)

Properties of Independent Transitions

For each state $s \in S$, *and for each* $(\alpha, \beta) \in I_s$:

– Preservation of Enabledness: *If* $\alpha, \beta \in enabled(s)$ *then* $\alpha \in enabled(\beta(s))$.

– Commutativity: *If* $\alpha, \beta \in enabled(s)$ *then* $\alpha(\beta(s)) = \beta(\alpha(s))$.

s, s_1, s_2, r, α, β

transitions commute

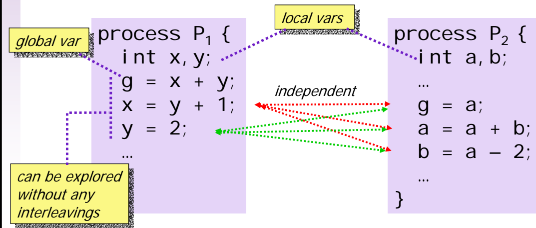
α *still enabled here*

Intuition

If property to be checked, doesn’t make observations about α, β then we only need to explore one of the paths

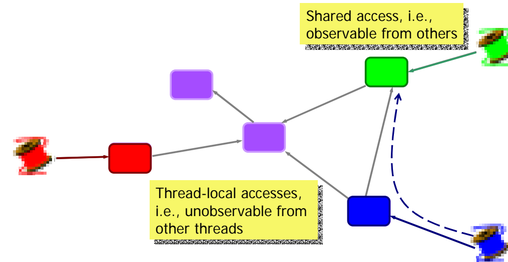
Classic PO Reductions

- Usually based on syntactic inspection of the transitions (approximation)
  - e.g., accesses to local variables



Dynamic Object-Oriented Software

Most data is heap-allocated, but it may still be local:



Dynamic Object-Oriented Software

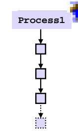
Most data is heap-allocated, but it may still be local:

- Local data corresponds to *thread-local objects* – objects that are accessible by a single thread only.
- *Thread-local transitions* are transitions that do not access non-thread-local objects.
  - analogous to transitions that only access local variables
- Thread-local transitions do not interfere with transitions from other threads – hence they should be considered independent.

Example

```
class Process extends Thread {
  Node head;
  public void run() {
    head = new Node(0);
    Node temp = head;
    for (int i = 1; i < 10; i++) {
      temp.next = new Node(i);
      temp = temp.next;
    }
    while (head != null) {
      head.x++;
      head = head.next;
    }
  }
}
```

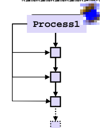
Create a linked-list of heap allocated nodes



Example

```
class Process extends Thread {
  Node head;
  public void run() {
    head = new Node(0);
    Node temp = head;
    for (int i = 1; i < 10; i++) {
      temp.next = new Node(i);
      temp = temp.next;
    }
    while (head != null) {
      head.x++;
      head = head.next;
    }
  }
}
```

Traverse list of nodes



Example

```
class Process extends Thread {
  Node head;
  public void run() {
    head = new Node(0);
    Node temp = head;
    for (int i = 1; i < 10; i++) {
      temp.next = new Node(i);
      temp = temp.next;
    }
    while (head != null) {
      head.x++;
      head = head.next;
    }
  }
}
```

How many of these transitions have to be interleaved with other threads?

…none! they are all thread local

Static Approach

- Static escape analysis can be used to detect thread-local objects (and thus, thread-local transitions)
- We implemented a modified version of Ruf's escape analysis for Java
  - concludes for the previous program that all transitions in the thread body do not access escaping objects
- However, there exists many other opportunities for thread-local-based reductions in typical programs
  - insight: an object can thread-local in some parts of the program but visible by more than one thread in others.

Example Revisited

```
class Process extends Thread {
  Node head;

  public void run() {
    head = new Node(0);
    Node temp = head;
    for (int i = 1; i < 10; i++) {
      temp.next = new Node(i);
      temp = temp.next;
    }

    while (head != null) {
      head.x++;
      head = head.next;
    }
  }
}
```

shift list creation code from 'run' into constructor

```
class Process extends Thread {
  Node head;

  public Process {
    head = new Node(0);
    Node temp = head;
    for (int i = 1; i<10; i++) {
      temp.next = new Node(i);
      temp = temp.next;
  }}

  public void run() {
    while (head != null) {
      head.x++;
      head = head.next;
    }
  }
}
```

Example Revisited -- Assessment

Static escape analysis says these objects are escaping and, thus, are not thread-local.

List of nodes are accessed by thread that creates Process here

List of nodes are accessed by Process thread here

```
class Process extends Thread {
  Node head;

  public Process {
    head = new Node(0);
    Node temp = head;
    for (int i = 1; i<10; i++) {
      temp.next = new Node(i);
      temp = temp.next;
  }}

  public void run() {
    while (head != null) {
      head.x++;
      head = head.next;
    }
  }
}
```

...but the list here is not visible to the creating thread!

Limitations of Static Approach

- To be classified as *thread-local*, conventional static escape analysis requires that an object *o* be accessible only by the same thread *t* throughout *o*'s entire lifetime.
- But note...
  - may have an object *o* that is thread-local for only part of its lifetime
  - may have an object *o* that is thread-local to a thread t_1 for one part of its lifetime and thread-local to a thread t_2 for another part of its lifetime.
  - may have a transition that is thread-local on some executions but not on others.

A Dynamic Framework

- To allow flexible classification of objects...
  - we will carry out escape analysis dynamically (on-the-fly) and allow the classification of objects as thread-local to change over their lifetime
- To allow flexible classification of transitions...
  - we will relax the conventional notion of independence relation to allow *state-sensitive independence* -- i.e., a pair of transitions can be independent in some states, but dependent in others

Ample Sets POR Framework

```
DFS(s)
 workSet := ample(s)
 while workSet is not empty
   let α ∈ workSet
   workSet := workSet - {α}
   s' := α(s)
   if not (s' ∈ seen) then
      seen := seen UNION {s'}
      pushStack(s')
      DFS(s')
      popStack()
end DFS
```

Main point: only explore transitions in *ample set*

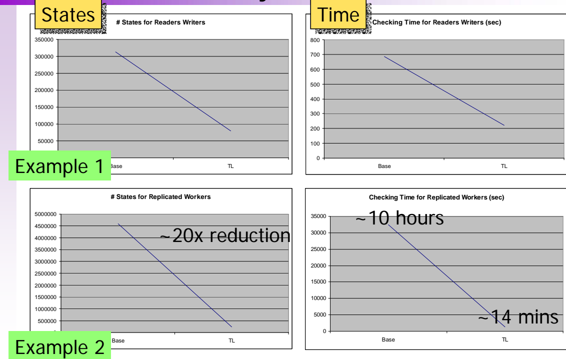
In our setting, we will construct ample(s) as follows:

pick a thread t such that all its transitions at the current state s are thread-local, and let ample(s) be this set of transitions.

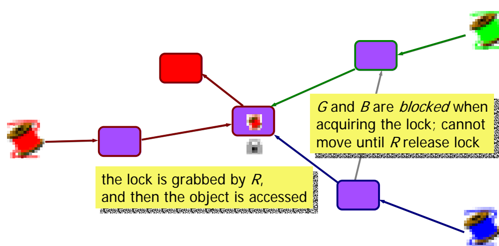
Assessment

- Replace a single fixed independence relation, I, with a family of independence relations, I_s, indexed by state s
  - This allows (α,β) to be independent in some states, but not in others.
- Correctness
  - Property preservation proofs for LTL-X
- Performance
  - Overhead for calculating I_s is small
  - Reduction is very large

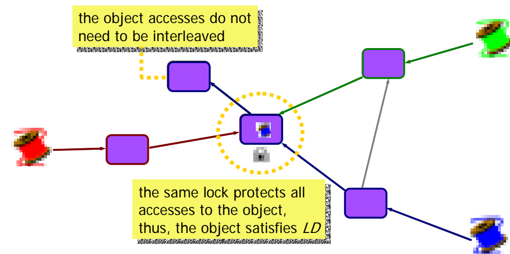
Thread Locality: Performace



Leveraging Locking Information



Leveraging Locking [Stoller '00]



Leveraging Locking Information

- Starting with the Stoller approach, we examined the locking patterns in a number of examples
- Added scans for patterns of valid locking to our "independence detector"
- Interesting combinations…
  - often an object is not locked during initialization (but it is usually thread-local then), then it becomes shared but lock-protected after the constructor completes

How Well Does It Work?

Base Bogor + Syntactic Local Variable POR

Thread-local Reduction (20x smaller)

Replicated Workers	trans	25557571	4376590	2076823	2535696	2661545
Threads: 4	states	4594721	1344215	827763	231219	231219
Locations: 509	time	9:0:54	1:12:35	0:39:48	0:21:47	0:21:10
Max. Objects: 44	mem	434.99	117.38	71.21	24.99	23.34

Thread-local + Locking Reduction (125x smaller)

Replicated Workers	trans	994007	994007	2908147		806863
Threads: 4	states	49354	49354	200947		36724
Locations: 509	time	0:4:8	0:5:6	0:16:39		0:3:52
Max. Objects: 44	mem	6.52	7.12	21.87		6.41

Thread-local + Locking + Read-only 4567 (1006x smaller)

>7000x reduction over old Spin-based version of Bandera!

How Hard To Implement?

- Dynamic Escape Analysis for POR
  - (150 LOC, 20x)
- Dynamic atomicity
  - (5 LOC, 5x)
- Extension of Stoller's Locking-discipline
  - (100 LOC, 20x)

Researchers at NASA Ames have now incorporated these reduction strategies into their JPF model-checker.

See "*Exploiting Object Escape and Locking Information in Partial Order Reduction for Concurrent Object-Oriented Programs*". FMSD 2004