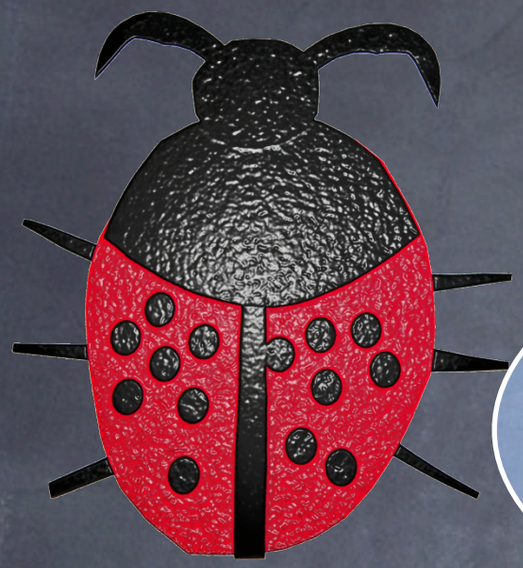
GPredict: Generic Predictive Concurrency Analysis

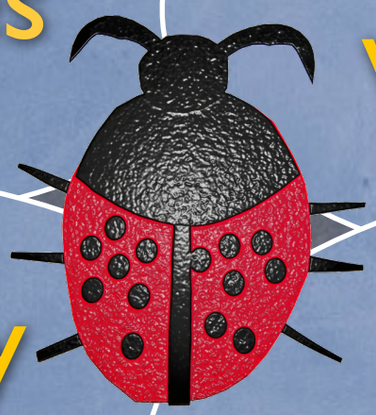
Jeff Huang, Qingzhou Luo, Grigore Rosu

Texas A&M University University of Illinois at Urbana-Champaign

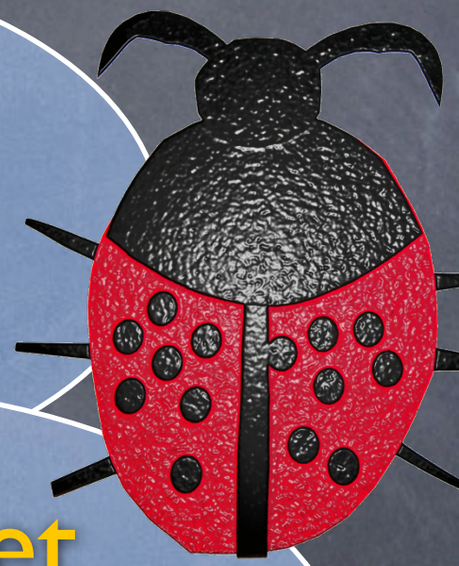
Concurrency Bugs



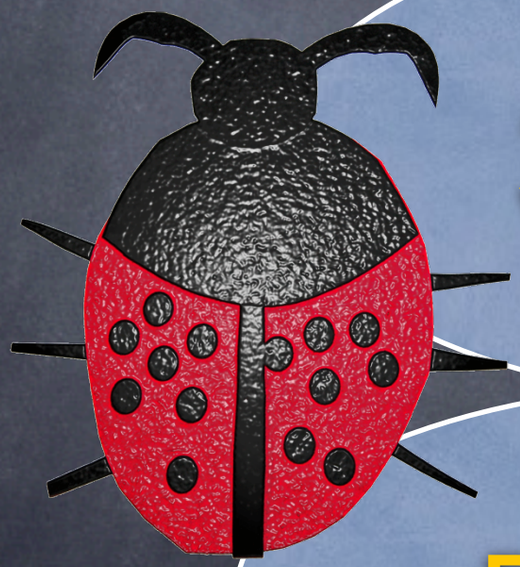
Data races



Order Violations



Atomicity Violations



Atomic-set Serializability Violations

Deadlocks

...

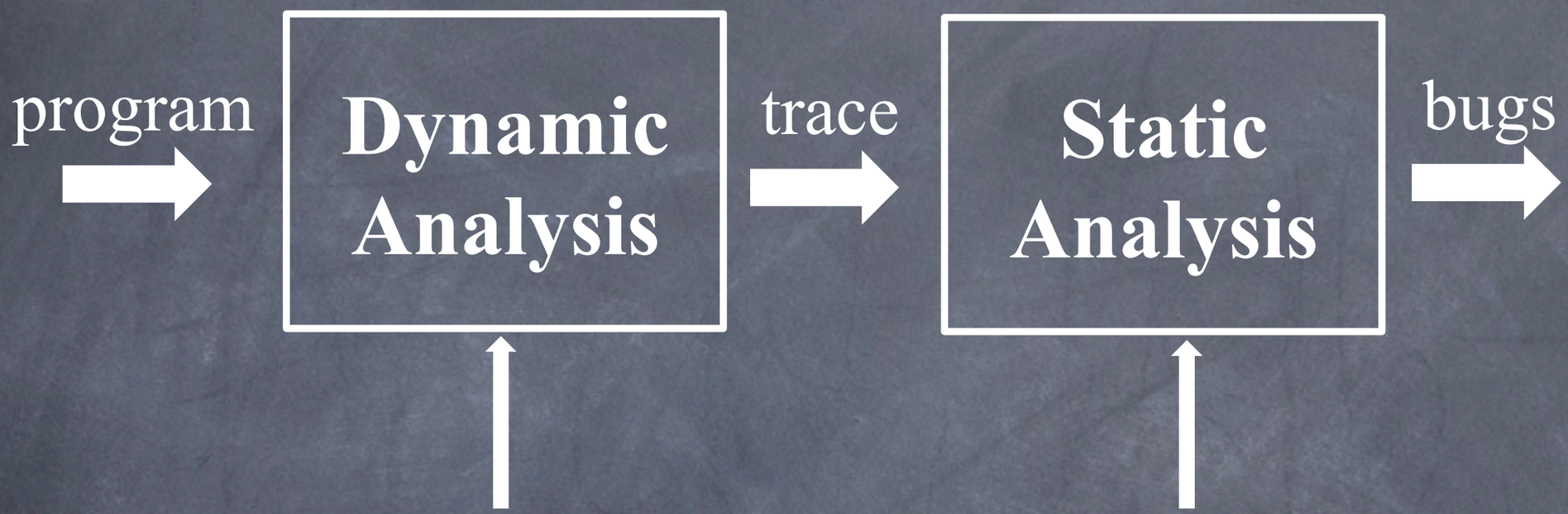
Real Problems Caused by Concurrency Bugs

Therac-25 3+ dead

$500M losses for investors

North American Blackout of 2003

Predictive Trace Analysis

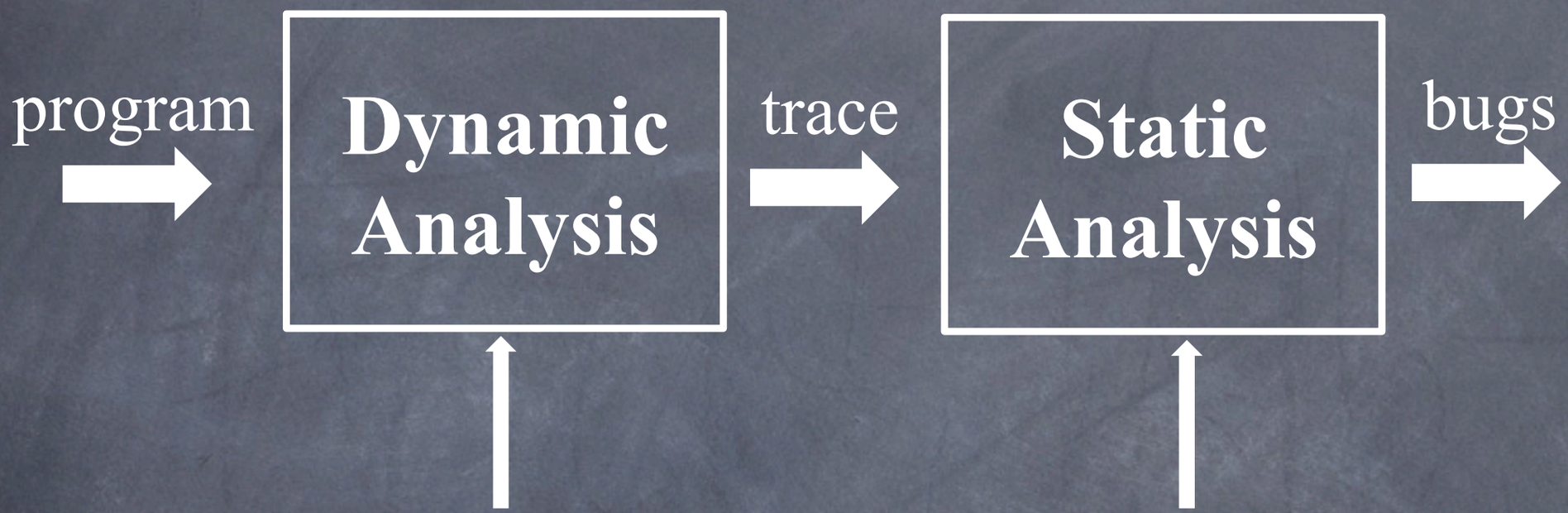


Collect shared data accesses & synchronization operations

Explore thread scheduling freedom

Chen et al. 2008
Wang et al. 2009
Huang et al. 2011
Huang et al. 2014

Predictive Trace Analysis



Collect shared data accesses & synchronization operations

Explore thread scheduling freedom

More precise than static analysis
More powerful than dynamic analysis

Chen et al. 2008
Wang et al. 2009
Huang et al. 2011
Huang et al. 2014

Predictive Trace Analysis

y is volatile

Thread T1	Thread T2
1: x=1	5: y=0
2: y=1	6: if(y==1)
3: if(x<0)	7: x=-1
4: Error	

Predictive Trace Analysis

y is volatile

Thread T1	Thread T2
1: x=1	5: y=0
2: y=1	6: if(y==1)
3: if(x<0)	7: x=-1
4: Error	

Race: (3,7)

Predictive Trace Analysis

y is volatile

Thread T1	Thread T2
1: x=1	5: y=0
2: y=1	6: if(y==1)
3: if(x<0)	7: x=-1
4: Error	

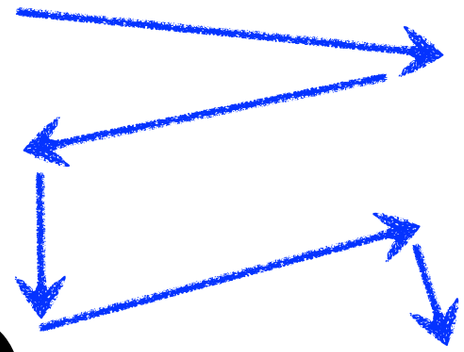
Race: (3,7)

Not race: (1,7)

Predictive Trace Analysis

schedule a: 1-5-2-3-6-7

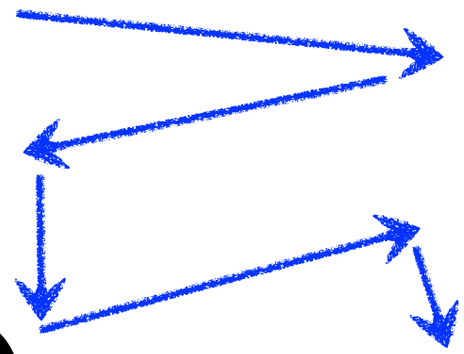
Thread T1	Thread T2
1: x=1	5: y=0
2: y=1	6: if(y==1)
3: if(x<0)	7: x=-1
4: Error	



Predictive Trace Analysis

schedule a: 1-5-2-3-6-7

Thread T1	Thread T2
1: x=1	5: y=0
2: y=1	6: if(y==1)
3: if(x<0)	7: x=-1
4: Error	



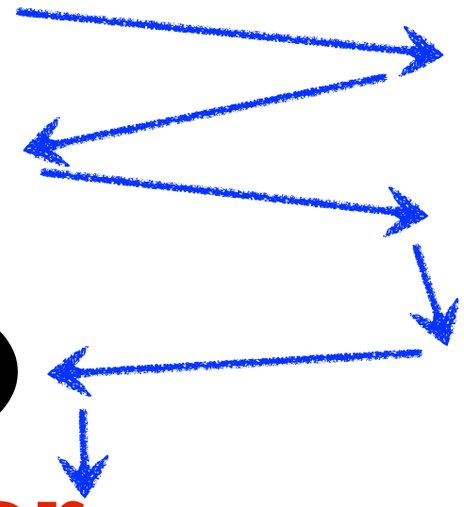
Race: (3,7)

Not race: (1,7)

Predictive Trace Analysis

schedule b: 1-5-2-6-7-3-4

Thread T1	Thread T2
1: x=1	5: y=0
2: y=1	6: if(y==1)
3: if(x<0)	7: x=-1
4: Error	



Race: (3,7)

Creating an execution to trigger the error

Huang et al. 2011, 2014

Concurrency Bugs

Data races

Order Violations

Atomicity

Atomic-set Violations

Deadlocks

...

All are low-level memory access errors!

High-level Property

Authenticate-before-use:

A method *authenticate* must be always called before a method *use* that uses a resource

High-level Property

Authenticate-before-use:

A method *authenticate* must be always called before a method *use* that uses a resource

No shared data!

High-level Property

Authenticate-before-use:

A method *authenticate* must be always called before a method *use* that uses a resource

No shared data!

Safe iterator:

A Java collection is not allowed to be modified when an iterator is accessing its elements

High-level Property

Authenticate-before-use:

A method *authenticate* must always be called

Existing low-level memory error-based detectors cannot detect these high-level property violations

A Java collection is not allowed to be modified when an iterator is accessing its elements

Contributions

A generic predictive trace analysis technique
to detect both high-level property violations and low-level memory errors

A specification language for defining concurrency property violations

An efficient constraint encoding approach based on thread local traces

Key Idea

Uniformly model property violations and thread schedules as simple ordering constraints and detect violations through constraint solving

Give an ORDER variable O to each critical event:

Authenticate-before-use

O_{auth} : order of the "authentication method call" event

O_{use} : order of a "resource use method call" event

$$O_{use} < O_{auth}$$

Key Idea

Uniformly model property violations and thread schedules as simple ordering constraints and solve them with an SMT solver

Give an ORDER variable O to each critical event:

Authenticate-before-use

O_{auth} : order of the "authentication method call" event

O_{use} : order of a "resource use method call" event

$$O_{use} < O_{auth}$$

Safe iterator

$$O_{create} < O_{update} < O_{next}$$

An Example

Collection<Item> c;
Item A, B; Iterator i_1, i_2;

T1	**T2**
1: c.add(A);	5: c.add(B);
2: start T2;	6: i_2=c.iterator();
3: i_1=c.iterator();	7: i_2.next()
4: i_1.next()	

O_i - order of event at line i

An Example

```
Collection<Item> c;
Item A, B;   Iterator i1, i2;

      T1                    T2

1:  c.add(A);
2:  start T2;                      5:  c.add(B);
3:  i1=c.iterator();               6:  i2=c.iterator();
4:  i1.next()                      7:  i2.next()
```

Oi - order of event at line i

Property violation constraint: $O_3 < O_5 < O_4$

An Example

```
Collection<Item> c;
Item A, B;   Iterator i1, i2;

        T1                      T2

1:  c.add(A);
2:  start T2;                5:  c.add(B);
3:  i1=c.iterator();         6:  i2=c.iterator();
4:  i1.next()                7:  i2.next()
```

O_i - order of event at line i

Property violation constraint:

$O_3 < O_5 < O_4$

Thread scheduling constraints:

$O_1 < O_2 < O_3 < O_4$

$O_5 < O_6 < O_7$

$O_2 < O_5$

An Example

```
Collection<Item> c;
Item A, B;   Iterator i1, i2;

T1                          T2
1: c.add(A);                5: c.add(B);
2: start T2;                6: i2=c.iterator();
3: i1=c.iterator();         7: i2.next()
4: i1.next()
```

Oi - order of event at line i

Property violation constraint:

$O_3 < O_5 < O_4$

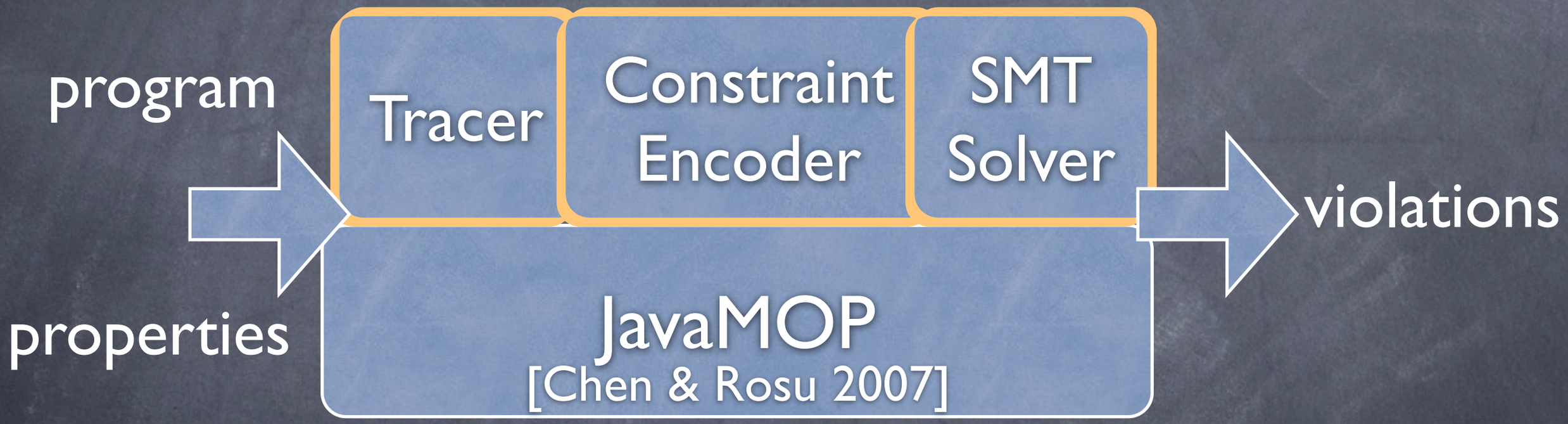
Thread scheduling constraints:

$O_1 < O_2 < O_3 < O_4$
$O_5 < O_6 < O_7$
$O_2 < O_5$

A property violation schedule:

1->2->3->5 ->4->6->7

GPredict Overview



GPredict Overview

Challenges

- Usability: how to define the property violations and easily use the technique in practice?
- Soundness & Completeness: we want to find as many as possible bugs with no false alarms
- Runtime Performance: reduce logging overhead as much as possible
- Scalability: how to efficiently solve the constraints?

Challenges

- Usability: how to define the property violations and

  RegExp patterns & JavaMOP extension

- Soundness & Completeness: we want to find as many as possible bugs with no false alarms

- Runtime Performance: reduce logging overhead as much as possible

- Scalability: how to efficiently solve the constraints?

Challenges

- Usability: how to define the property violations and

  RegExp patterns & JavaMOP extension

- Soundness & Completeness: we want to find as

  Parametric constraint encoding

- Runtime Performance: reduce logging overhead as much as possible

- Scalability: how to efficiently solve the constraints?

Challenges

- Usability: how to define the property violations and

  RegExp patterns & JavaMOP extension

- Soundness & Completeness: we want to find as

  Parametric constraint encoding

- Runtime Performance: reduce logging overhead as

  Constraint encoding with thread local traces

- Scalability: how to efficiently solve the constraints?

Challenges

- Usability: how to define the property violations and

  RegExp patterns & JavaMOP extension

- Soundness & Completeness: we want to find as

  Parametric constraint encoding

- Runtime Performance: reduce logging overhead as

  Constraint encoding with thread local traces

- Scalability: how to efficiently solve the constraints?

  Existing high performance solvers: Z3 & Yices

Property Specification Syntax

<*GPredict Specification*>	::=	<*Property Name*> "(" <*Parameters*> ")" "{" <*Event*>* <*Pattern*> "}"
<*Event*>	::=	"event" <*Id*> <*AspectJ AdviceSpec*> ":" <*AspectJ Pointcut*>
<*Pattern*>	::=	"pattern :" (<*RegExp*> "||")*<*RegExp*>
<*Property Name*>	::=	<*Identifier*>
<*Parameters*>	::=	(<*Type*> <*Identifier*>)+
<*Id*>	::=	<*Identifier*>
<*AspectJ AdviceSpec*>	::=	AspectJ AdviceSpec syntax
<*AspectJ Pointcut*>	::=	AspectJ Pointcut syntax
<*Thread*>	::=	<*Identifier*>
<*AtomRegion*>	::=	<*Identifier*>
<*Identifier*>	::=	Java Identifier syntax
<*Begin*>	::=	"<" <*AtomRegion*>
<*End*>	::=	">" <*AtomRegion*>
<*RegExp*>	::=	Regular expression over {<*Id*>, <*Id*> "(" <*Thread*>, <*Begin*> | <*End*> ")"}

Gray color: new syntax introduced for concurrency properties

Support RegExp patterns, syntax based on JavaMOP & AspectJ

Property Specification Syntax

<*GPredict Specification*> ::= <*Property Name*> "(" <*Parameters*> ")" "{" <*Event*>* <*Pattern*> "}"

<*RegExp*> ::= Regular expression over {<*Id*>, <*Id*> "(" <*Thread*>,<*Begin*> | <*End*> ")"}

```
DataRace (Object o) {
    event read before(Object o):
            get(* s) && target(o);
    event write before(Object o):
            set(* s) && target(o);

    pattern: read(t1) || write(t2)
}
```

Gray color: new syntax introduced for concurrency properties

Support RegExp patterns, syntax based on JavaMOP & AspectJ

Property Specification Syntax

```
Unsafelterator (Collection c, Iterator i) {
    event create after(Collection c) returning(Iterator i) :
            call(Iterator Collection+.iterator()) && target(c);
    event update after(Collection c) :
            (call(* Collection+.remove*(..))
            || call(* Collection+.add*(..)) ) && target(c);
    event next before(Iterator i) :
            call(* Iterator.next()) && target(i);
    pattern: create next* update+ next
}
```

Gray color: new syntax introduced for concurrency properties

Support RegExp patterns, syntax based on JavaMOP & AspectJ

Property Specification Syntax

In a database application, a table must be created before any update or query on the table, and all operations must finish before closing the database connection

```
UnsafeDatabaseAccess(Connection conn, String table) {

    event open after() returning(Connection conn):
        call(Connection DriverManager.getConnection(String));
    event close before(Connection conn):
        call(void Connection.close())&&target(conn);
    event create before(Connection conn, String table):
        call(* createTable(Connection conn, String table))&&args(conn,table);
    event delete before(Connection conn, String table):
        call(* deleteTable(Connection conn, String table))&&args(conn,table);
    event update before(String table, String sql):
        call(boolean Statement.execute(String))
        &&args(sql)&&if(sql.contains(table));
```

pattern : !(open create update delete close)

Parametric Encoding

Binding property parameters to concrete object instances

Provided by JavaMOP

Caveat: an identifier may match with multiple events

Unsafelterator pattern: create next* update+ next

Item A, B; Iterator i_1, i_2;

T1	**T2**
1: c.add(A);	5: c.add(B);
2: start T2;	6: i_2=c.iterator();
3: i_1=c.iterator();	7: i_2.next()
4: i_1.next()	

Parametric Encoding

Binding property parameters to concrete object instances

Provided by JavaMOP

Caveat: an identifier may match with multiple events

Unsafelterator pattern: create next* update+ next

Item A, B; Iterator i_1, i_2;

T1	**T2**
1: c.add(A);	5: c.add(B);
2: start T2;	6: i_2=c.iterator();
3: i_1=c.iterator();	7: i_2.next()
4: i_1.next()	

$O_3 < O_1 < O_4$

$O_1 < O_2 < O_3 < O_4$

$O_5 < O_6 < O_7$

$O_2 < O_5$

Parametric Encoding

Binding property parameters to concrete object instances

Provided by JavaMOP

Caveat: an identifier may match with multiple events

Unsafelterator pattern: create next* update+ next

Item A, B; Iterator i1, i2;

Solution:

Enumerate all matching events and disjunct constraints

T2

$O_3 < O_1 < O_4 \vee O_3 < O_5 < O_4$

$O_1 < O_2 < O_3 < O_4$

$O_5 < O_6 < O_7$

$O_2 < O_5$

7: i2.next()

4: i1.next()

Thread Local Encoding

Predictive trace analysis often requires a global trace

But, logging a globally-ordered trace of events is expensive

Thread Local Encoding

Predictive trace analysis often requires a global trace

But, logging a globally-ordered trace of events is expensive

New encoding based on thread local traces, much cheaper to log

Thread Local Encoding

Predictive trace analysis often requires a global trace

But, logging a globally-ordered trace of events is expensive

New encoding based on thread local traces, much cheaper to log

Thread scheduling constraints

$$\Phi = \Phi_{mhb} \wedge \Phi_{sync} \wedge \Phi_{rw}$$

Must happens-before constraints (Φ_{mhb})

Synchronization Constraints (Φ_{sync})

Read-write constraints (Φ_{rw})

Thread Local Encoding

Predictive trace analysis often requires a global trace

But, logging a globally-ordered trace of events is expensive

New encoding based on thread local traces, much cheaper to log

Thread scheduling constraints

$$\Phi = \Phi_{mhb} \wedge \Phi_{sync} \wedge \Phi_{rw}$$

Must happens-befor

Synchronization Cor

Read-write constr

All take thread local traces as input

Results

Program	Property	#Threads	#Events	#Violations	Time
JFreeChart	UnsafeIterator	21	230	20	0.85s
H2	UnsafeDBAccess	5	126	16	0.67s
Derby1	NullPointerDeref	3	12K	5	4.7s
Derby2	CheckThenAct	3	21K	4	8.5s
Jigsaw	DataRace	12	17K	29	17.6s
StringBuffer	AtomicityViolation	3	64	2	0.4s
JDK-Logger	Deadlock	2	570	1	2.6s

Predicted 77 violations in seven benchmarks with real bugs

More description of the results:

http://parasol.tamu.edu/~jeff/gpredict/

Runtime Performance

Program	#Threads	Base Time	Global	Local (speedup)
Avrora	4	1.8s	2m20s	2m6s (10%)
Batik	2	2.1s	2m28s	1m2s (58%)
Xalan	11	1.7s	6m14s	1m9s (82%)
Lusearch	10	1.4s	15m17s	23m16s (78%)
Sunflow	25	1s	19m48s	11m28s (42%)

Runtime overhead reduced by 54% on average

Summary

- Generic predictive trace analysis for concurrent programs base on constraint solving
  - Predict both high-level property violations and low-level memory access errors
- Predictive runtime verification
  - A concurrency property specification language
  - Constraint encoding with thread local traces -- significant runtime performance improvement

GPredict: Generic Predictive Concurrency Analysis

Jeff Huang, Qingzhou Luo, Grigore Rosu

Texas A&M University University of Illinois at Urbana- Champaign

Thank You & Questions?