다중 쓰레드 프로그램의 경쟁상태오류 검출기법 분류

KAIST Provable SW 연구실
홍신 , 김문주

Motivations (1/2)

- Many SW applications utilize multi-threaded programming techniques as multi-core hardware become widely spread.

- Writing correct multi-threaded programs is difficult.
  – Exponential number of execution scenarios
  – Detecting errors by assertion is not effective

  ➔Bug detection techniques specialized for concurrency errors

Motivations (2/2)

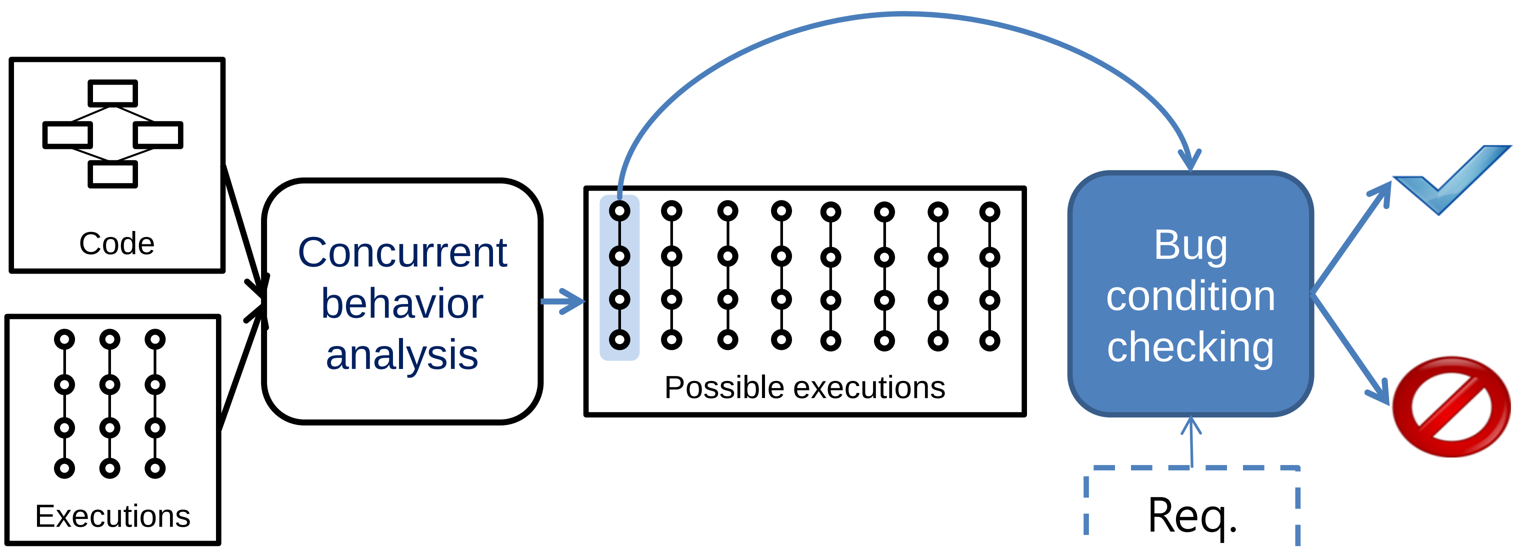
- Many techniques have evolved
  - Deadlock : [K. M. Chandy *et al.*, TOCS 1983], [R. Agarwal *et al.* , IBM J. 2010]
- However, for race bugs, techniques have used their own definitions and notations without any reliance on a common ground or platform.

	Bug name	Specification	Target program	Bug checking method
Kivati	Atomicity violation	User annotations	C program	Memory access pattern in run-time
LiteRace	Data race	N/A	x86 binary	Execution orders in run-time
Havelund *et al.*	High-level data race	Analyze code and infer specifications	Java	Relationship between synchronization and variables in run-time execution

→ Develop classification which clarify the relationship between techniques and provide a clear top-down view of race detection techniques

Approach (1/2)

- Provide a formal execution model and specify various bug conditions according to the model
  - Decoupling **what to detect** and **how to detect**
    – Concurrent behavior analyses: generate potential executions of a program by information from static/dynamic analyses [F. Chen *et al*. ICSE 2008]
    – **Bug condition checking**: examine a given execution is acceptable/erroneous

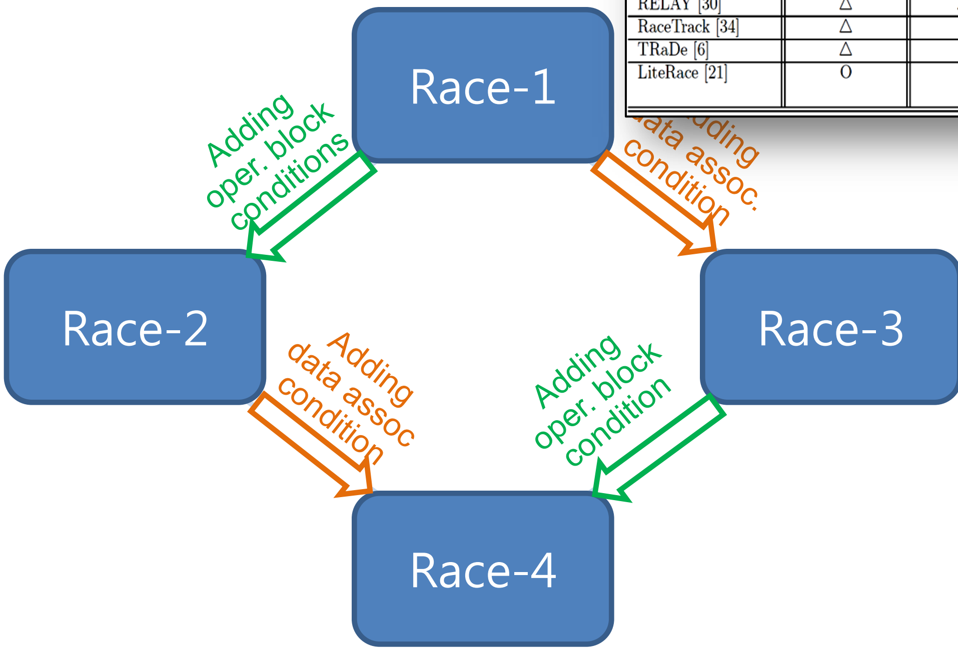


Approach (2/2)

- Classify bug conditions according to type of specifications
  - Operation block
  - Data association

Race-1 techniques

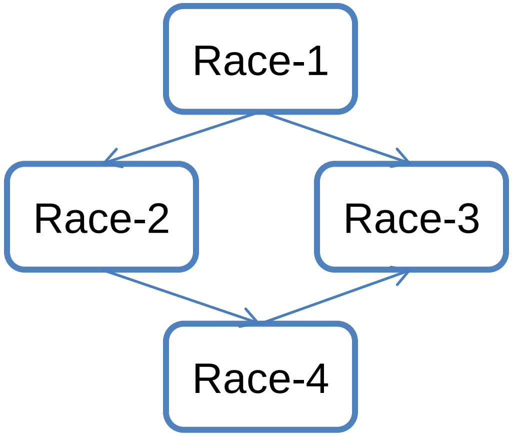
	(A1) $thread(p)$	(A2) $oprd(p)$	(A3) $conflict(p,q)$				(A4) $p \not\succ q$	
	precise thread info.	precise alias info.	W→R	R→W	W→W	R→R	lock	others
Choi et al. [5]	O	O	O	O	O	X	O	thread fork-join
Chord [22]	△	△	O	O	O	X	O	N/A
Eraser [27]	△	O	O	O	O	X	O	N/A
Hybrid Data Race Detection [24]	O	O	O	O	O	X	O	msg. passing
Racer [2]	O	O	O	O	O	X	O	N/A
RacerX [7]	△	△	O	O	O	X	O	N/A
RccJava [9]	△	△	O	O	O	O	O	N/A
RELAY [30]	△	△	O	O	O	X	O	N/A
RaceTrack [34]	△	O	O	O	O	X	O	thread fork-join
TRaDe [6]	△	O	O	O	O	X	O	N/A
LiteRace [21]	O	O	O	O	O	X	O	atomic operation, msg. passing



Contents

- Execution model for multithreaded program

- Four classes of race detection techniques
  - For each class,
    - bug example
    - bug conditions
    - techniques for checking conditions



- Implications for better race detections

Execution Model (1/5)

- An execution model of a target program P used for technique D is defined as

$$EM_P(D) = (\boxed{T,\ e,\ \triangleright},\ \boxed{B_{op},\ A_{data}})$$

Program behavior　　Requirements

- T : a finite set of threads
- e : an interleaved execution

  a finite sequence of operations $p_1, p_2, \ldots, p_n$ where $p_i \in Operation$

  - $thread(p) \in T$
  - $optr(p) \in Operator$

    $conflict(optr(p), optr(q))$ if the operators of p and q are commutable.

  - $oprd(p) \subseteq V_S$

Execution Model (2/5)

- An execution model of a target program P used for technique D is defined as

$$EM_P(D) = (T, e, \triangleright, B_{op}, A_{data})$$

- T : a finite set of threads
- e : an interleaved execution

  a finite sequence of operations $p_1, p_2, \ldots, p_n$ where $p_i \in Operation$

  - $thread(p) \in T$
  - $optr(p) \in Operator$

    $conflict(optr(p), optr(q))$ if the operators of p and q may not be commutable.
  - $oprd(p) \subseteq V_S$

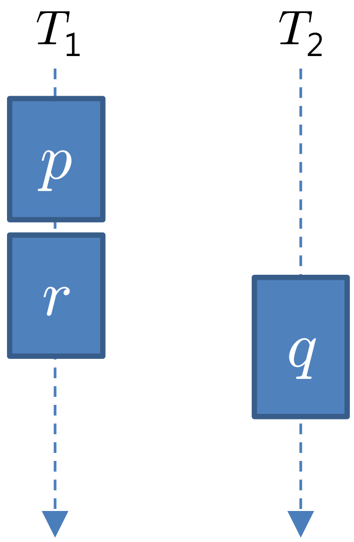
Execution Model (3/5)

- An execution model of a target program P used for technique D is defined as

$$EM_P(D) = (\, T, e, \triangleright, B_{op}, A_{data} \,)$$

- Operations in an execution are totally ordered by their start time.
- $\triangleright \subseteq Operation \times Operation$

  $(p, q) \in \triangleright$ if $t_s(p) < t_s(q)$ and $t_e(p) < t_e(q)$



$p \triangleright r$
$p \triangleright q$

$r \not\triangleright q$
$q \not\triangleright r$

Execution Model (4/5)

- An execution model of a target program P used for technique D is defined as

$$EM_P(D) = (T, e, \triangleright , \mathbf{B}_{op}, A_{data})$$

$$\mathbf{B}_{op} = \{b_1, b_2, ..., \} \text{ where } b_i : \mathit{Operation} \times \mathit{Operation}$$

An execution of an atomic code region corresponds to a sequence of operation, *operation blocks*.

$(p, q) \in b_i$ indicates that p and q are in the same operation block.

```
class BankAccount {
  int balance ;
  ...
  void withdraw(int amount) {
    if (getBalance() >= amount) {
      lock(m) ;
      balance = balance - amount ;
      unlock(m) ;
    }
  }
```

Execution Model (5/5)

- An execution model of a target program P used for technique D is defined as

$$EM_P(D) = (T, e, \triangleright, B_{op}, \boldsymbol{A}_{data})$$

$\boldsymbol{A}_{data}$: $V_S \times V_S$ where V_S is a set of shared variables

Frequently, variables in a composite data structure have dependencies and there exists relations/invariances on these variables.

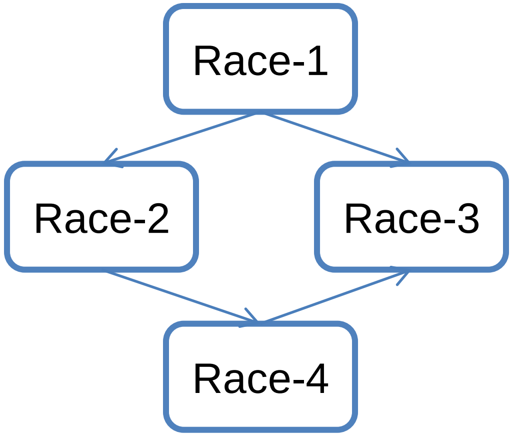
```
class BankAccount {
  int balance ;
  int debt ;
  /* invariant: (debt == 0    ∧ balance == 0) ∨
                (debt > 0     ∧ balance == 0) ∨
                (balance > 0 ∧ debt == 0) */
```

➔ (balance, debt) $\in A_{data}$

(debt, balance) $\in A_{data}$

Contents

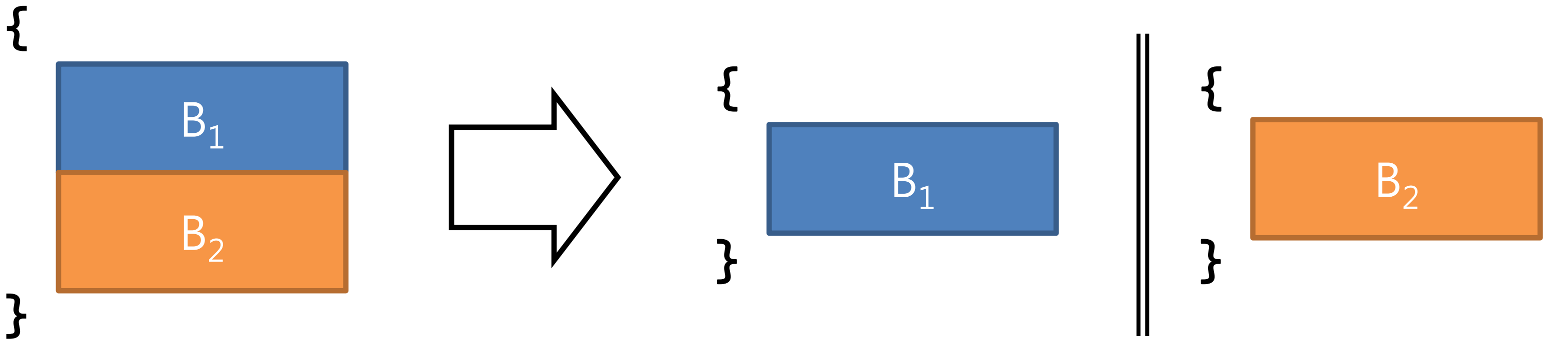
- Execution model for multithreaded program
- Four classes of race detection techniques
  - For each class,
    - bug example
    - bug conditions
    - techniques for checking conditions



- Implications for better race detections

Race-1: Data-race (1/6)

In parallelization,



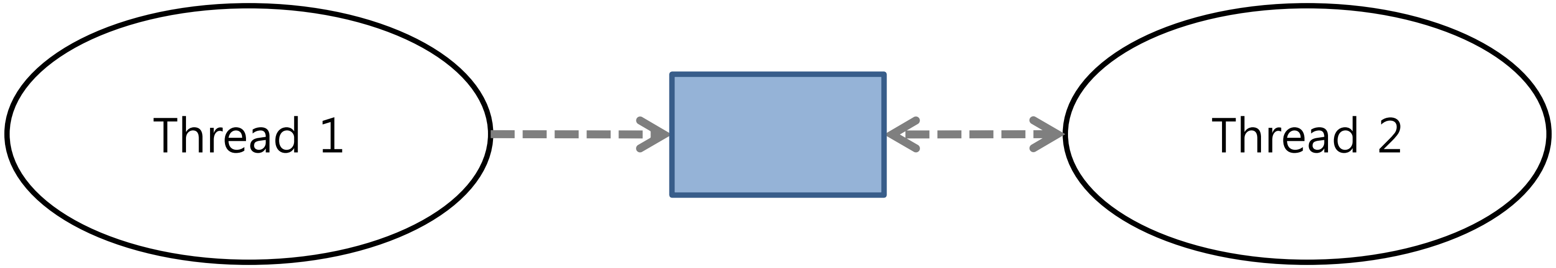
A sufficient condition for safe parallelization:

$MemoryRead(B_1) \cap MemoryWrite(B_2) = \emptyset$ and
$MemoryWrite(B_1) \cap MemoryRead(B_2) = \emptyset$

- In “What are race conditions?” [R. H. Netzer et al., LOPLAS 1992],
  a *data race* <a, b> over an execution exists if and only if
  (1) a data conflict exists in a program between a and b,
  (2) no temporal ordering between a and b.

Race-1: Data-race (2/6)

- In [Savage et al., SOSP 1997],
  a *data race* occurs when there exists two operations
  (1) executed by two concurrent threads,
  (2) access a shared variable
  (3) at least one access is write,
  (4) no explicit mechanism to coordinate their execution order



Race-1: Data-race (3/6)

- Buggy program code

```
class BankAccount_A {
  int balance;
  // balance should be non-negative
  void withdraw(int x) {
1:   if (balance >= x){
2:     balance = balance - x;}}
  ... }
```

- Error scenario

```
[ balance = 10 ]
--t1: withdraw(10)--          --t2: withdraw(10)--
                              1:if(balance >= 10)
1:if(balance>= 10)  p
                              2:balance = 10-10;  q
2:balance = 0 - 10;
```

The invariant is violated: `balance` becomes -10.

- A target program P has a *race-1 bug* if there is an execution σ such that σ has two operations p and q that satisfy the following conditions:

$$(A1) \quad thread(p) \neq thread(q)$$

$$(A2) \quad oprd(p) \cap oprd(q) \neq \emptyset$$

$$(A3) \quad conflict(optr(p), optr(q))$$

$$(A4) \quad p \not\triangleright q \wedge q \not\triangleright p$$

p and q are commutable?

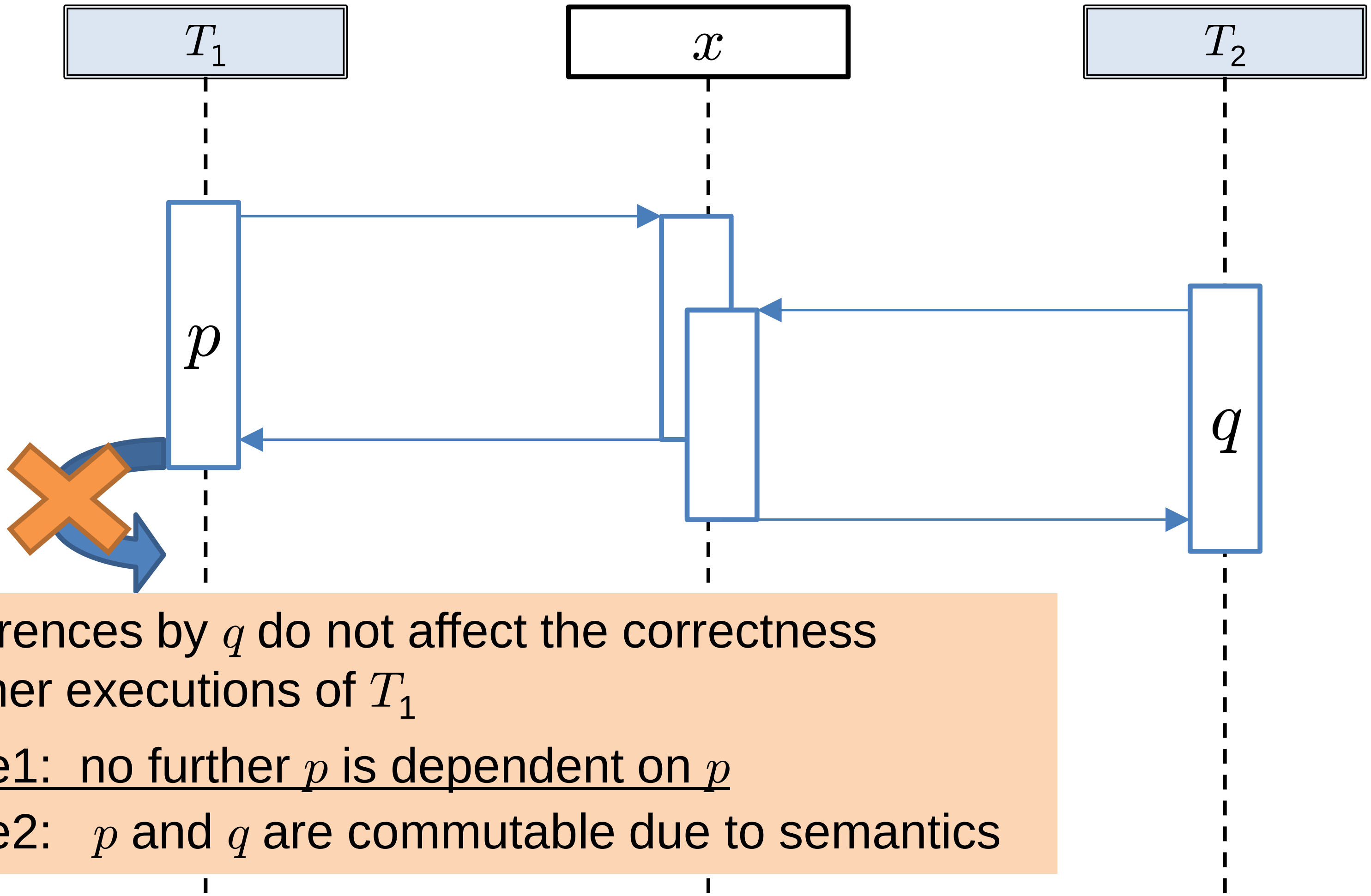
Race-1: Data-race (4/6)

- Race-1 Detection Techniques

	(A1) $thread(p) \neq thread(q)$	(A2) $oprd(p) \cap oprd(q) \neq \emptyset$	(A3) $conflict(optr(p), optr(q))$				(A4) $p \not\triangleright q$
			W-R	R-W	W-W	R-R	
Choi *et al.*	concrete thread id.	concrete mem. addr.	O	O	O	X	tracking lock acq/rel, **thread fork/join**
Eraser	check shared or non-shared	concrete mem. addr.	O	O	O	X	tracking lock acq/rel
Hybrid data race detection	concrete thread id.	concrete mem. addr.	O	O	O	X	tracking lock acq/rel, **message send/receive**
Racer	concrete thread id.	concrete mem. addr.	O	O	O	X	tracking lock acq/rel
RaceTrack	check shared or non-shared	concrete mem. addr.	O	O	O	X	tracking lock acq/rel, **thread fork/join**
TRaDe	check shared or non-shared	concrete mem. addr.	O	O	O	X	tracking lock acq/rel
LiteRace	concrete thread id.	concrete mem. addr.	O	O	O	X	tracking lock acq/rel, **message passing, atomic instructions**
Chord	static approx.	static alias analysis	O	O	O	X	tracking lock acq/rel
RacerX	static approx.	static alias analysis	O	O	O	X	tracking lock acq/rel
RELAY	static approx.	static alias analysis	O	O	O	X	tracking lock acq/rel
RccJava	static approx.	static alias analysis	O	O	O	O	tracking lock acq/rel

Race-1: Data-race

- Limitations – false positive



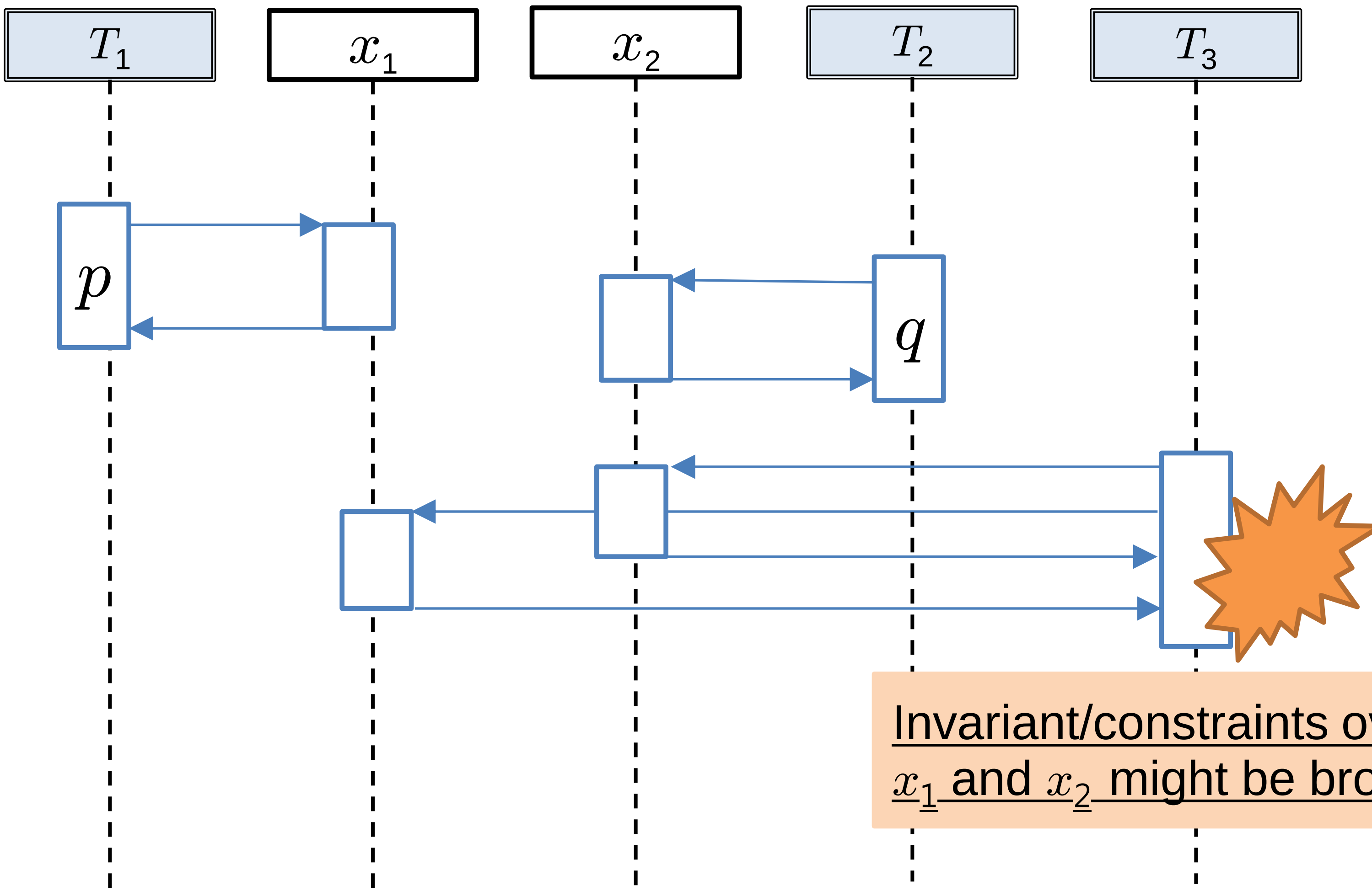
Interferences by q do not affect the correctness of further executions of T_1

<u>Case1: no further p is dependent on p</u>

Case2: p and q are commutable due to semantics

Race-1: Data-race (6/6)

- Limitations – false negative



Invariant/constraints over x_1 and x_2 might be broken

Race-2: Atomic Block Violation (1/5)

- Race-2 techniques check whether or not an ***operation block*** can interfere with another thread.

In [Savage et al., SOSP 1997], *From a previous slide for race-1*

a *data race* occurs when there exists two operations

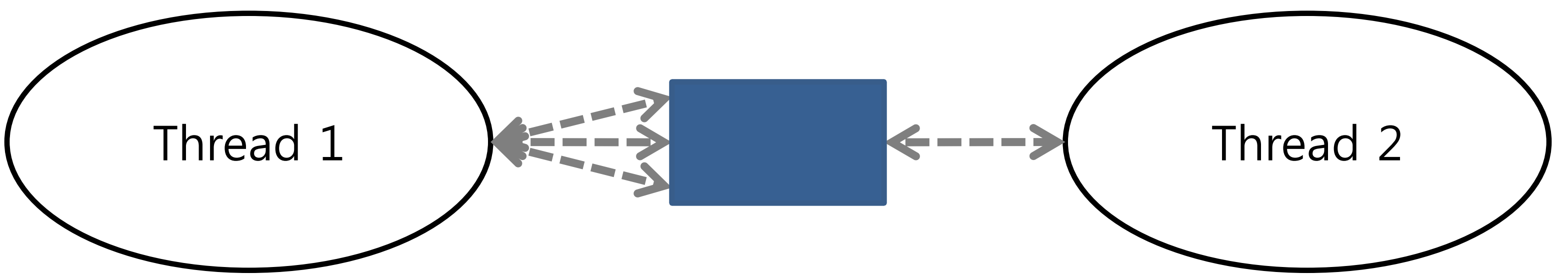
(1) executed by two concurrent threads,

(2) access a shared variable

(3) **at least one access is write**,

(4) no explicit mechanism to coordinate their execution order

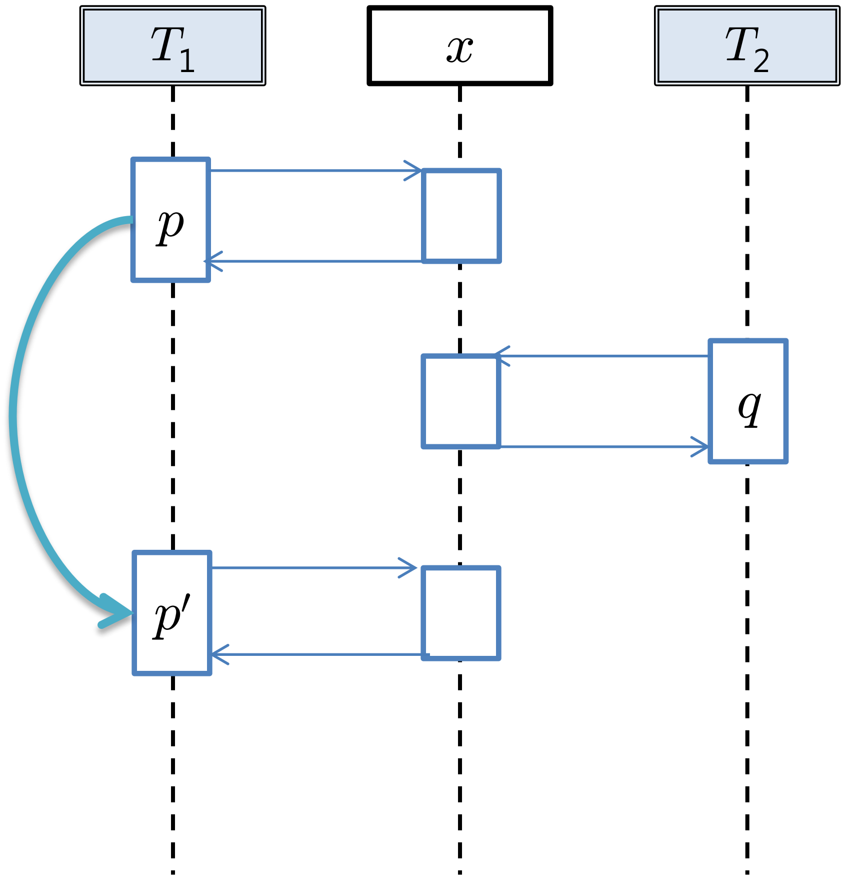
a sequence of accesses (operation block)



Race-2: Atomic Block Violation (2/5)

- A target program P has a *race-2 bug* if there is an execution σ such that σ has three operations p, p', and q that satisfy the following conditions:

$(B1)$	$thread(p) \neq thread(q)$
$(B2)$	$oprd(p) \cap oprd(q) \neq \varnothing$
$(B3)$	$conflict(optr(p), optr(q))$
$(B4)$	$\exists\, b_i \in B_{op}.(\,(p, p') \in b_i\,)$
$(B5)$	$oprd(p) \cap oprd(p') \neq \varnothing$
$(B6)$	$conflict(optr(p), optr(p'))$
$(B7)$	$oprd(p') \cap oprd(q) \neq \varnothing$
$(B8)$	$conflict(optr(p'), optr(q))$
$(B9)$	$q \not\rhd p \quad \wedge \quad p' \not\rhd q$



Race-2: Atomic Block Violation (3/5)

- Race-2 bug example:

- Buggy program code

```
class BankAccount_B {
  Lock m;
  int balance;
  // balance should be non-negative
  // balance should be synchronized by m

  int getBalance() {
    int tmp;
1:  lock(m);
2:  tmp = balance ;
3:  unlock(m);
4:  return tmp; }

  void withdraw(int x) {
    /*@atomic region begins*/
11: if (getBalance() >= x) {
12:   lock(m);
13:   balance = balance - x ;
14:   unlock(m); }
    /*@atomic region ends*/
    ...
```

- Race-2 error scenario

```
[ balance = 10 ]
  --t1: withdraw(10)--          --t2 : withdraw(10)--
operation block b_i             ...
11:if(getBalance()>=10)
 ↳ 1:lock(m);
   2:tmp = balance;    p
   3:unlock(m);
   4:return tmp;
                                12: lock(m);
                                13: balance=10-10;   q
                                14: unlock(m);
12: lock(m);
13: balance=0-10;      p'
14: unlock(m);
```

The invariant is violated: balance becomes -10.

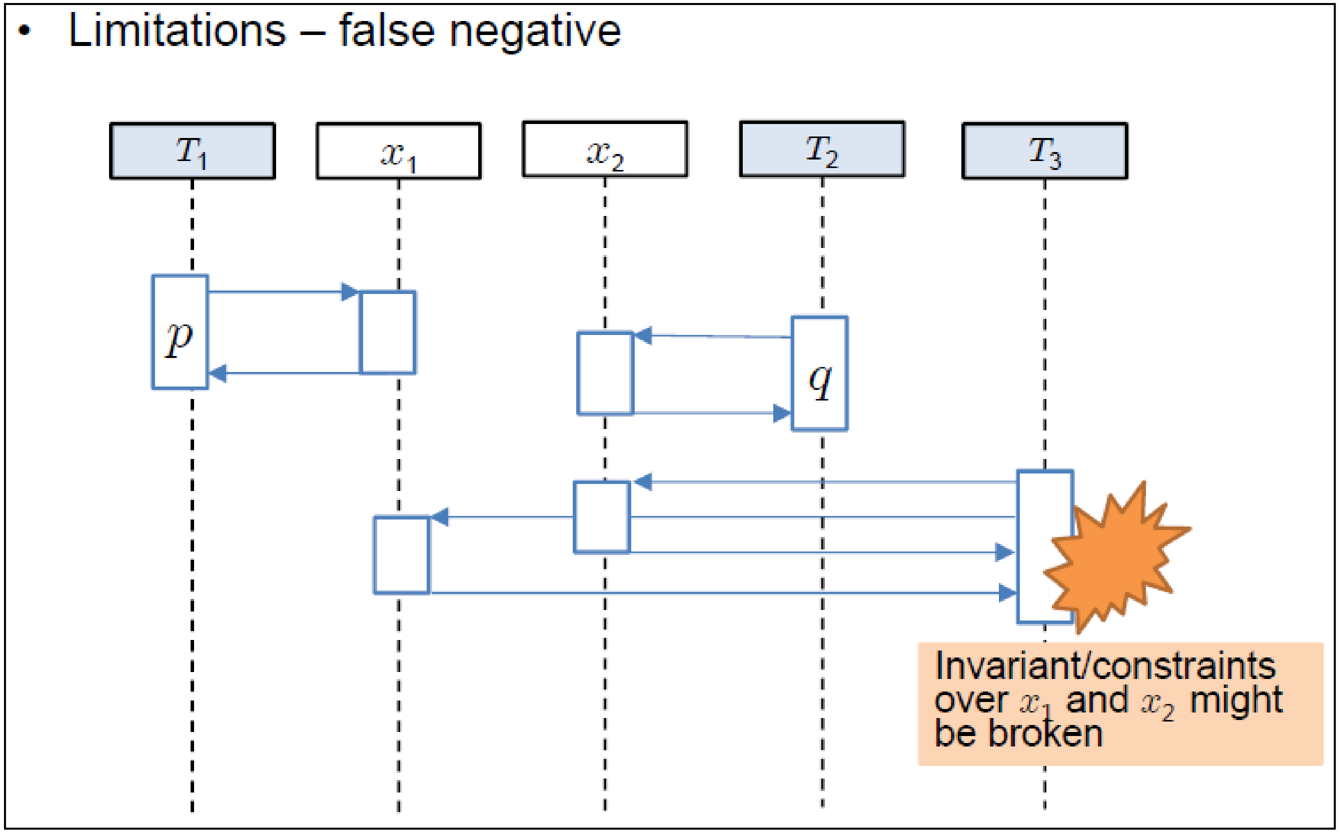
Race-2: Atomic Block Violation (4/5)

- Race-2 Detection Techniques

	(B1) $thread()$	(B2, B4,B7) $oprd()$	(B6) $conflict(optr(p), optr(q))$ where $thread(p) \neq thread(q)$				(B3, B8) $conflict(optr(p), optr(q))$ where $thread(p) = thread(q)$				(B9) $\not\triangleright$
			W-R	R-W	W-W	R-R	W-R	R-W	W-W	R-R	
Atomic-Aid	concrete thread id.	concrete mem. addr.	O	O	Cond.	X	O	O	O	O	tracking lock/unlock
AtomRace	concrete thread id.	concrete mem. addr.	O	O	Cond.	X	O	O	O	O	tracking lock/unlock
AVIO	concrete thread id.	concrete mem. addr.	O	O	Cond.	X	O	O	O	O	total execution order
Block-based algorithm	concrete thread id.	concrete mem. addr.	O	O	Cond.	X	O	O	O	O	tracking lock/unlock, message passing
Commit-node	concrete thread id.	concrete mem. addr.	O	O	O	X	O	O	O	O	tracking lock/unlock message passing
HAVE	concrete thread id.	concrete mem. addr.	O	O	O	X	O	O	O	O	tracking lock/unlock message passing.
Kivati	concrete thread id.	concrete mem. addr.	O	O	Cond.	X	O	O	O	O	total execution order
SVD	concrete thread id.	concrete mem. addr.	O	O	O	O	X	X	O	O	total execution order
PENELOPE	concrete thread id.	concrete mem. addr.	O	O	O	X	O	O	O	O	tracking lock/unlock
Velodrome	concrete thread id.	concrete mem. addr.	O	O	O	X	O	O	O	O	tracking lock/unlcok, message passing
Atomizer	static apprx.	alias analysis	O	O	O	O	O	O	O	O	tracking lock/unlock

Race-2: Atomic Block Violation (5/5)

- Limitation: false-negative



Race-3: Data Assoc. Violation (1/4)

- A unit of data can be located in two or more distinct variables.
- Race-3 detection techniques look for inconsistent updates of associated variables. [K. Havelund VVEIS03, S. Lu SOSP07, F. Tip ICSE08]

From a previous slide for Race-1

In [Savage et al., SOSP 1997],

a *data race* occurs when there exists two operations

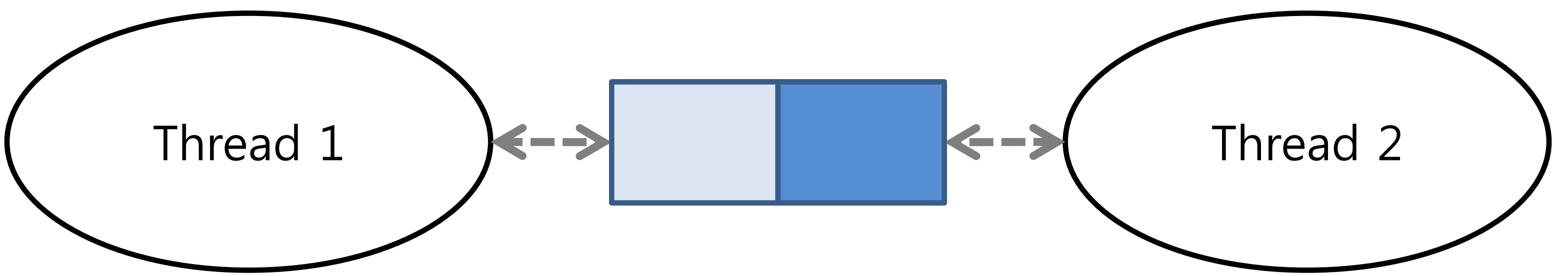
(1) executed by **two concurrent threads**,

(2) access **a shared variable**

(3) **at least one access is write**,

(4) **no explicit mechanism to coordinate their execution order**

memory area,
set of variables
(associated variables)



Race-3: Data Assoc. Violation (2/4)

- Race-3 bug condition:

(C1) $thread(p) \neq thread(q)$

(C2) $\exists v_1, v_2 \in V_S.(v_1 \in oprd(p) \land v_2 \in oprd(q) \land (v_1, v_2) \in A_{data})$

(C3) $conflict(optr(p), optr(q))$

(C4) $p \not\triangleright q \land q \not\triangleright p$

Race-3: Data Assoc. Violation (3/4)

- Example

```
class BankAccount_C {
  int  balance ;
  int  debt ;
  /* Invariant:
    (balance == 0 ∧ debt == 0) ∨
    (debt > 0 → balance == 0) ∨
    (balance > 0 → debt == 0)   */
```

➔ (balance, debt) $\in A_{data}$ ∧
(debt, balance) $\in A_{data}$

```
    Lock  m_balance ;
    Lock  m_debt ;
```

```
[ balance=0, debt=10 ]
 --t1: deposit(20)--                 --t2: withdraw(5)--
                                     21:if(getBalance()==0)
                                      ...
11:lock(m_debt);
12:if(0 < 10 && 10 <=20)
13:tmp = 20-10;
14:dept=0;
15:unlock(m_debt);
16:lock(m_balance);                  22:lock(m_debt);
17:balance = 10;   p                 23:debt = 0+ 5;   q
```

The invariant is violated:
balance is 10 and debt is 5

Race-3: Data Assoc. Violation (4/4)

- Race-3 detection techniques

	(C1) $thread()$	(C2) A_{data}				(C3) $conflict()$	(C4) $\not\triangleright$
		Type	transi-tive	symm-etric	Source		
MultiRace	Concrete thread id.	$[\ a_1, a_2, \ldots, a_n\ ]$	O	O	User annotation	W-R, R-W, W-W	tracking lock/unlock
MUVI-Eraser	Heuristics	$<(a_1, t_1), (a_2, t_2)>$ where $t_1, t_2 \in \{rd, wr, rd\&wr\}$	X	X	Inferring from executions	W-R, R-W, W-W	tracking lock/unlock
Object data race detection	Static apprx.	$\{\ a_1, a_2, \ldots, a_n\ \}$	O	O	User annotation	W-R, R-W, W-W	tracking lock/unlock

Race-4 (1/4)

- Race-4 techniques utilize both operation block and data association together to reduce false positives and false negatives.

In [Savage et al., SOSP 1997],

From a previous slide for Race-1

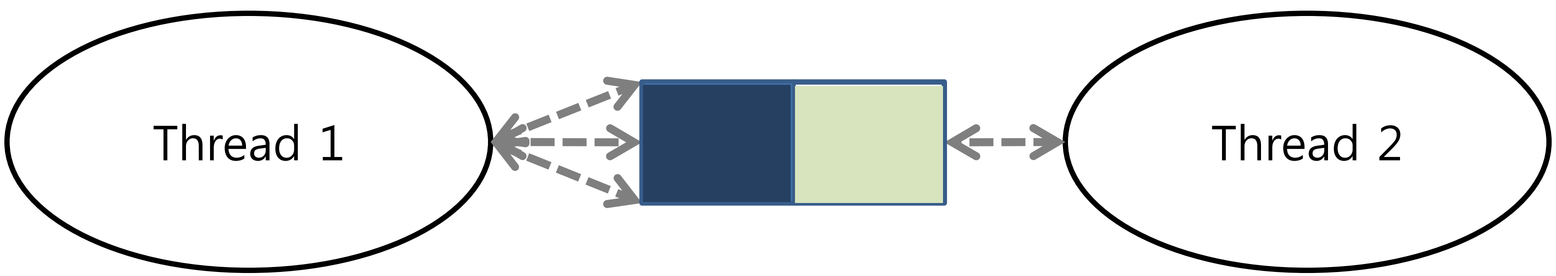
a *data race* occurs when there exists two operations

(1) executed by **two concurrent threads**,

(2) access **a shared variable**

(3) **at least one access is write**,

(4) **no explicit mechanism to coordinate their execution order**



Race-4 (2/4)

- Race-4 bug conditions:

Race-B conditions

Race-C conditions

$(D1)$	$thread(p) \neq thread(q)$
$(D2)$	$\exists v_1, v_2 \in V_S.(v_1 \in oprd(p) \ \wedge \ v_2 \in oprd(q) \ \wedge \ (v_1, v_2) \in A_{data})$
$(D3)$	$conflict(optr(p), optr(q))$
$(D4)$	$\exists v_3, v_4 \in V_S.(v_3 \in oprd(p') \ \wedge \ v_4 \in oprd(q) \ \wedge \ (v_3, v_4) \in A_{data})$
$(D5)$	$conflict(optr(p'), optr(q))$
$(D6)$	$\exists \ b_i \in B_{op}.(\ (p,p') \in b_i \ )$
$(D7)$	$\exists v_5, v_6 \in V_S.(v_5 \in oprd(p') \ \wedge \ v_6 \in oprd(q) \ \wedge \ (v_5, v_6) \in A_{data})$
$(D8)$	$conflict(optr(p), optr(p'))$
$(D9)$	$q \not\rhd p \ \ \wedge \ \ p' \not\rhd q$

Race-4 (3/4)

- Buggy program

```
class BankAccount_D {
    Lock m;
    int balance, debt;
    /*(balance, debt)∈ A_data
      (debt, balance)∈ A_data */

    int getBalance(int x) {
      int tmp;
 1:   lock(m);
 2:   tmp = balance;
 3:   unlock(m);
 4:   return tmp;}

    int withdraw(int x) {
11:   if (getBalance() == 0) {        b1
12:     lock(m);
13:     debt = debt + x;
14:     unlock(m);}}

    int deposit(int x) {
21:   lock(m);
22:   if (balance == 0) {
23:     if (x > debt) {
24:       balance = x - debt;
25:       debt = 0; }
26:       ...
27:   unlock(m); }
```

- Error execution scenario

```
[balance=0, debt=10 ]

  --t1: withdraw(5)--           --t2: deposit(20)--
11:if(getBalance()== 0)
  1: lock(m);
  2: tmp = 0;        p
  3: unlock(m);
                                21:lock(m);
                                22:if(0 == 0)
                                23:if(20 > 10);
                                24:balance= 20-10;   q
                                25:debt = 0;
                                26: …
                                27:unlock(m);
12:lock(m);
13:debt = debt+5;    p'
14:unlock(m);
```

The invariant is violated:
`balance` is 10 and `debt` is 15

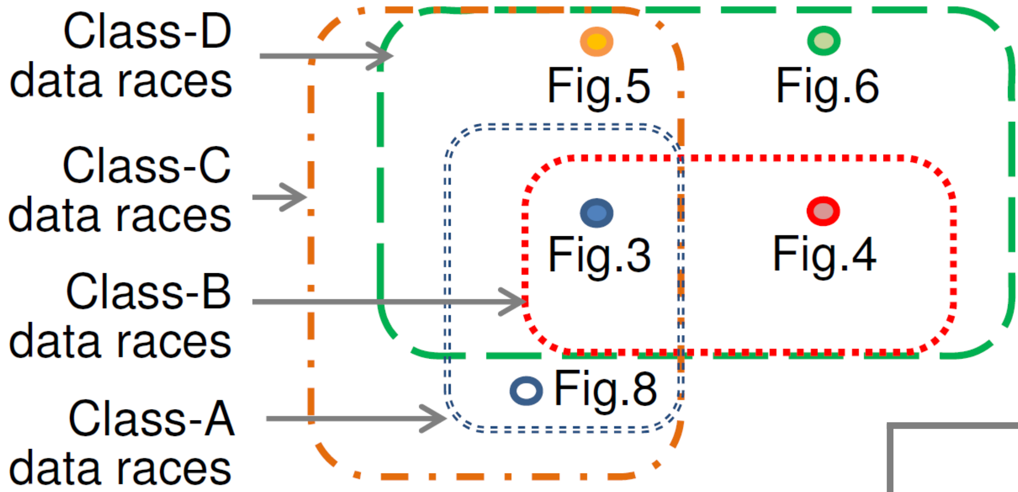
Race-4 (4/4)

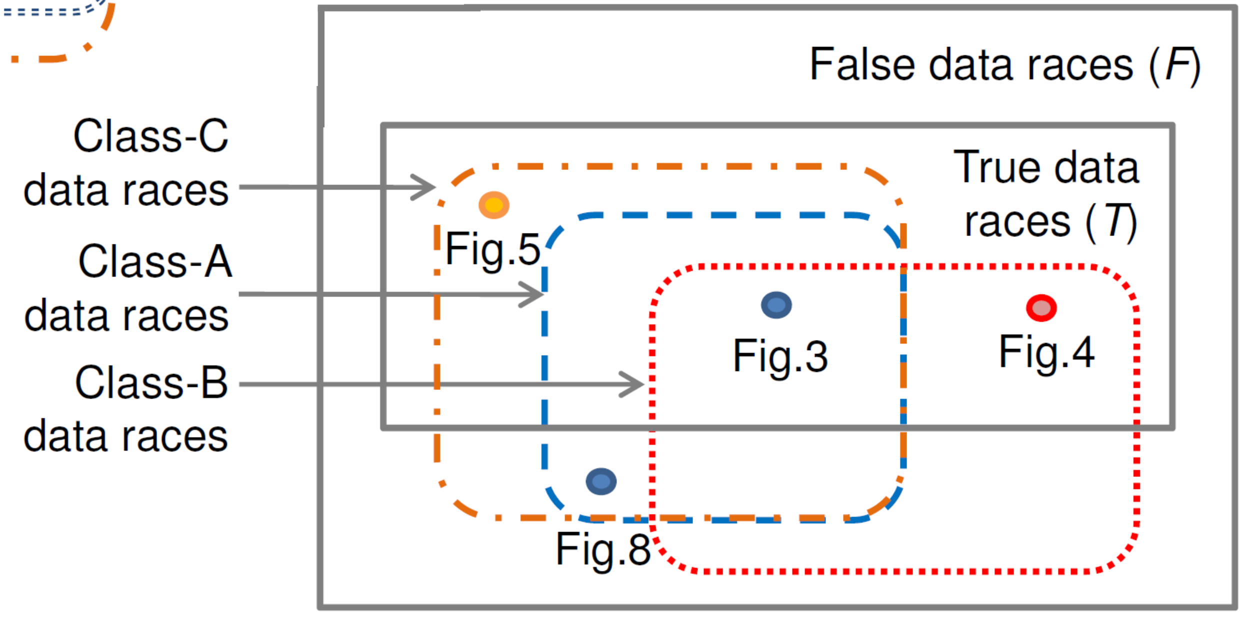
- Race-4 detection techniques

	(D1) $thread()$	(D2, D4, D7) A_{data}			(D3,D5) $conflict()$ where $thread(p) \neq thread(q)$	(D6,D8) $conflict()$ where $thread(p) \neq thread(q)$	(D9) $\not\triangleright$
		Type	transi-tive	symm-etric			
Atomic-Set serializability	Concrete thread id.	$< a_1, a_2 >$	O	O	W-R, R-W, W-W	W-R, R-W, W-W, R-R	total execution order
ColorSafe	Concrete thread id.	$< a_1, a_2 >$	O	O	W-R, R-W, W-W	W-R, R-W, W-W, R-R	total execution order
Method-consistency	Static apprx.	$<(a_1, t_1), (a_2, t_2)>$ where $t_1, t_2 \in \{read, update\}$	O	O	W-R, R-W, W-W, R-R	W-R, R-W, W-W, R-R	tracking lock/unlock
MUVI-AVIO	Concrete thread id.	$<(a_1, t_1), (a_2, t_2)>$ where $t_1, t_2 \in \{rd, wr, rd\&wr\}$	X	X	W-R, R-W, W-W(cond),	W-R, R-W, W-W, R-R	total execution order
View-consistency	Concrete thread id.	$<(a_1, t_1), (a_2, t_2)>$	O	O	W-R, R-W, W-W, R-R	W-R, R-W, W-W, R-R	tracking lock/unlock

Implications to Better Race Detection (1/2)

- Relations in four class of race detections





Implications to Better Race Detection (2/2)

- Static analyses can be applied for much precise race detections
  - Only few work use static analyses for inferring/checking operation block and data associations

Theorem 1. Let $G^{du} = (G, \Sigma, D, U)$ be a def/use graph with $G = (N, E)$, and let $u, v \in N$. If u is semantically dependent on v and this semantic dependence is finitely demonstrated, then u is syntactically dependent on v.[2]

Theorem 1 is significant because it shows that, given appropriate definitions of control and data dependence, syntactic dependence is a necessary condition for (finitely demonstrated) semantic dependence. Thus, the theorem provides justification for algorithms that use syntactic dependence to approximate semantic dependence. We refer to this desirable relationship between syntactic and semantic dependence as the "*syntactic–semantic* relationship".

M. J. Harrold, G. Rothermel, and S. Sinha. "Computation of Interprocedural Control Dependence" Proceedings of the ACM International Symposium on Software Testing and Analysis, March 1998.