Detecting Atomicity Violations via Integrated Dynamic and Static Analysis

FRACTAL

April 25, 2009

Qichang Chen, Liqiang Wang
Department of Computer Science
University of Wyoming

Zijiang Yang
Department of Computer Science
Western Michigan University

Scott D. Stoller
Computer Science Department
Stony Brook University

Outline

- Introduction to atomicity
- Integrate dynamic and static analysis to detect atomicity violations
- Conflict-edge algorithm
- Experiments and conclusions

A Typical Concurrency Error: Atomicity Violation

Duplicate vector v1

Thread 1

lock(v1);
v2 = new Vector(v1.size());
unlock(v1);

lock(v1);
copy v1's elements to v2;
unlock(v1);

Thread 2

lock(v1);
Remove all elements in v1;
unlock(v1);

Error: space is allocated for v2,
but no elements are copied.

Informal Definition of Atomicity

- **Transaction**: an execution of a code block expected to be atomic.
- Given a program, a set of code blocks (includes transactions) is **atomic** if every concurrent execution of the program is **equivalent** to a serial execution (i.e., the transactions are executed without interruption by other threads).
- Two executions are **equivalent** if every two conflicting events (a read and a write to the same variable, or two writes to the same variable) appear in the same order.

Example: Atomicity Violation

- Transaction t1= $W_1(x)$ $W_2(x)$
- Transaction t2 = W(x).
- All feasible executions:

E1: $W_1(x)$ $W_2(x)$ $W(x)$ serial

E2: $W(x)$ $W_1(x)$ $W_2(x)$ serial

E3: $W_1(x)$ $W(x)$ $W_2(x)$ not serial

- E3 is not equivalent to E1 and E2,
- {t1, t2} has a potential atomicity violation.

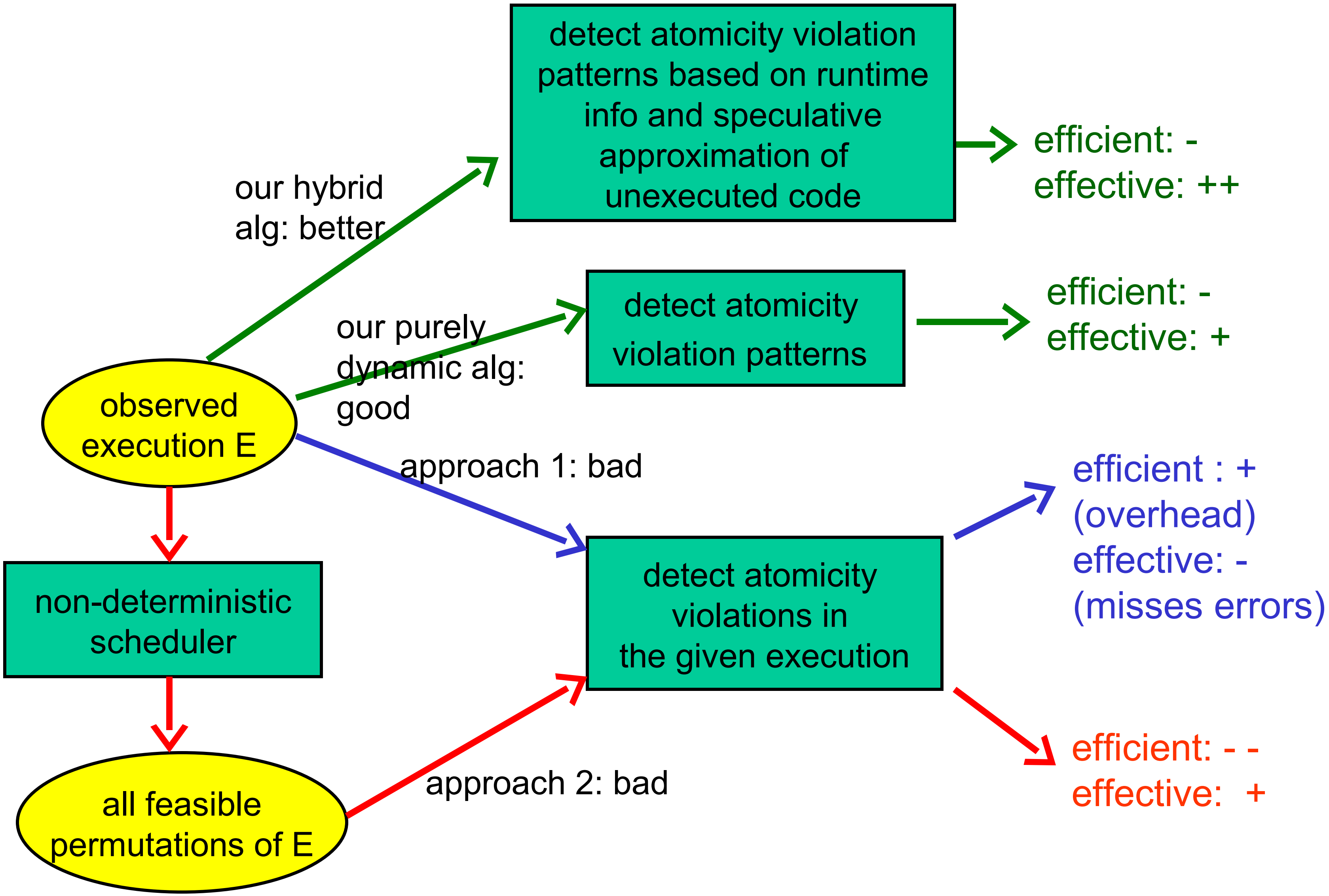
Outline

- Introduction to atomicity
- Integrate dynamic and static analysis to detect atomicity violations
- Conflict-edge algorithm
- Experiments and conclusions

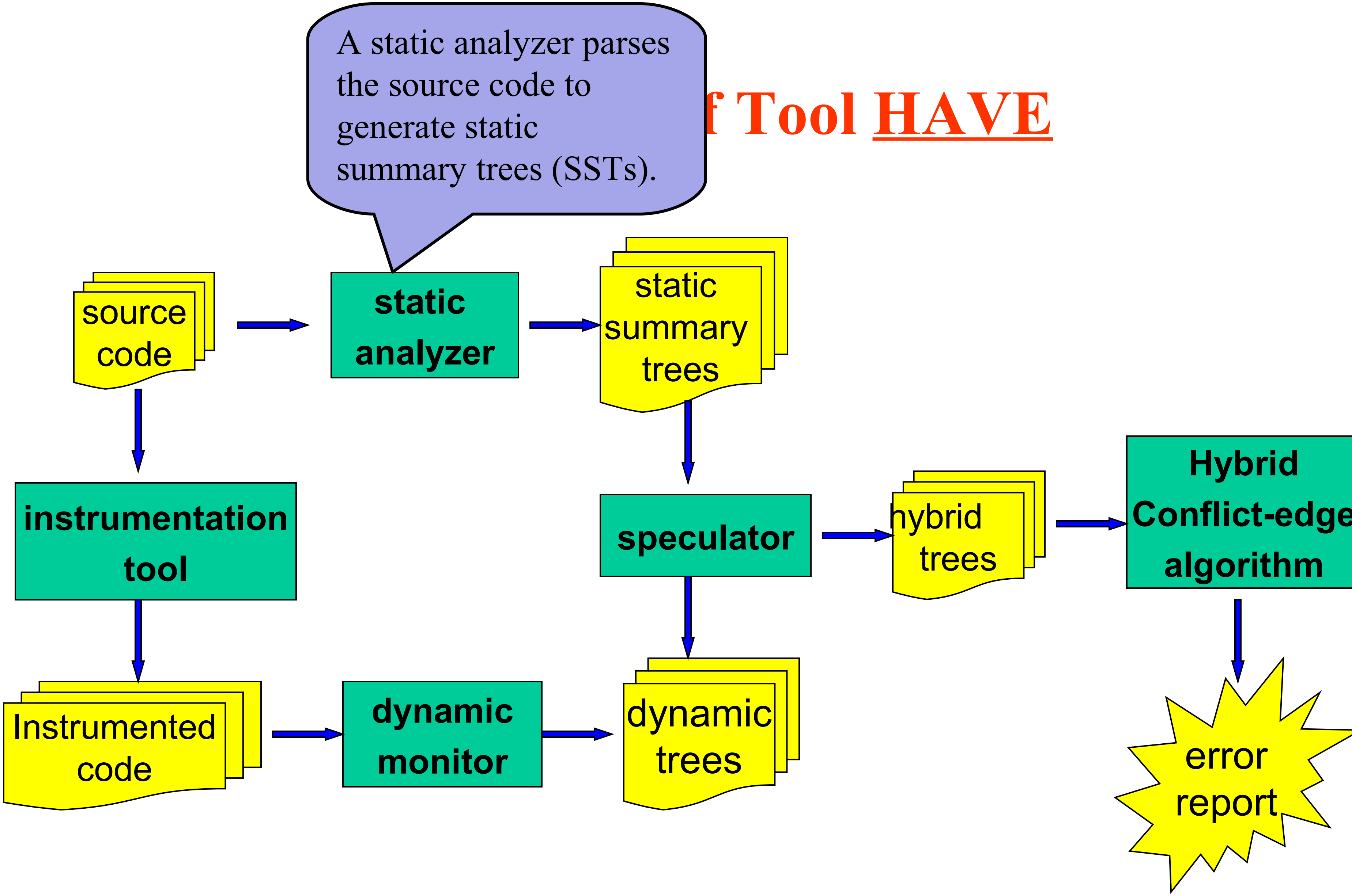
Approaches to Detect Atomicity Violations

- Purely static (Flanagan et al.)
  - Pros: all possible behaviors can be checked.
  - Cons: many false positives.
- Purely dynamic (Wang et al.; Flanagan et al.; Xu et al.; Lu et al.; Park et al.)
  - Pros: much fewer false positives.
  - Cons: cannot analyze unobserved behaviors.
- Integrated static and dynamic analysis (our approach)
  - Get some of the benefits of both approaches
  - Few false positives
  - Analyzes some unobserved behaviors

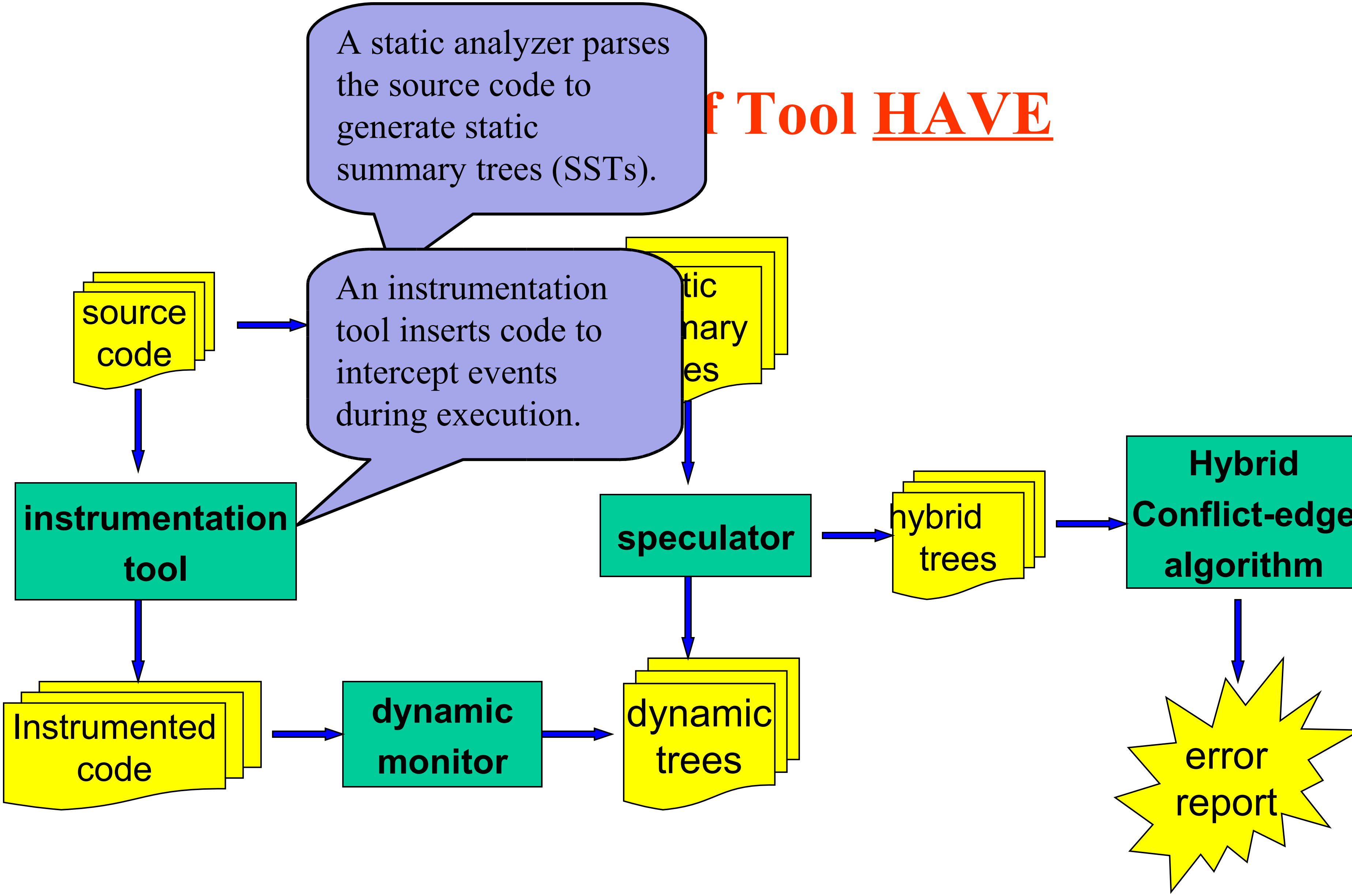
How to Detect Atomicity Violations?



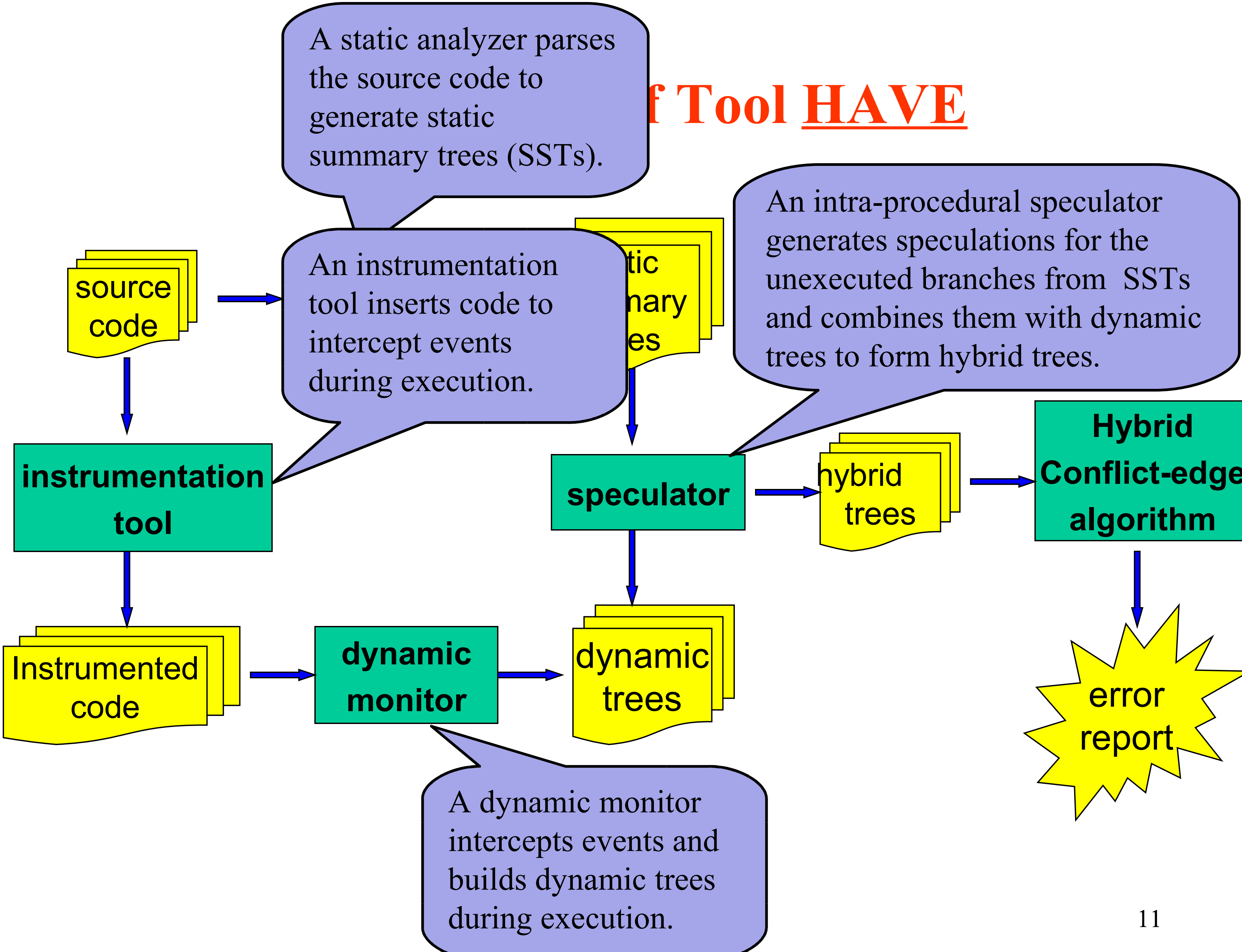
f Tool HAVE

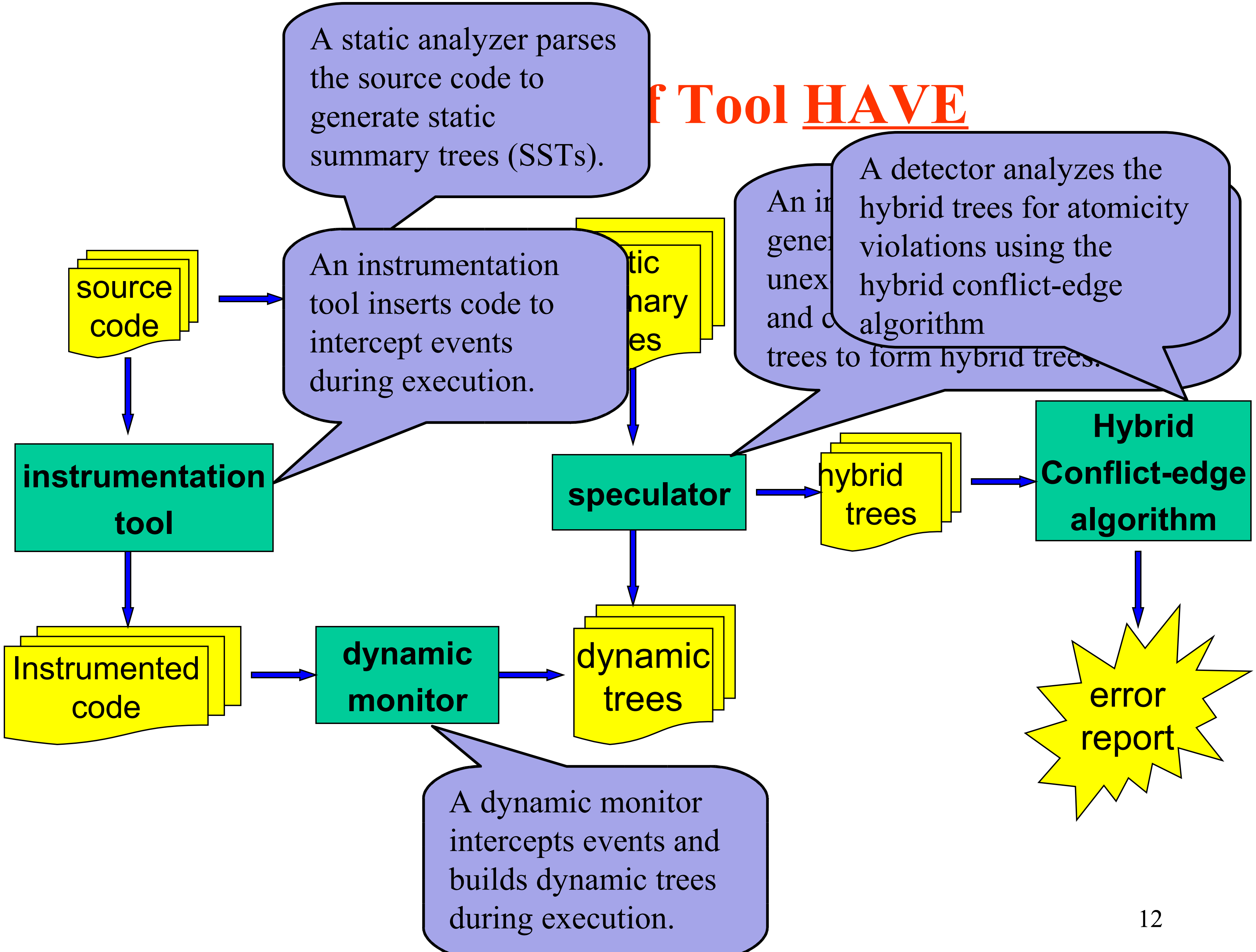

A static analyzer parses the source code to generate static summary trees (SSTs).
source code
static analyzer
static summary trees
instrumentation tool
speculator
hybrid trees
Hybrid Conflict-edge algorithm
Instrumented code
dynamic monitor
dynamic trees
error report

f Tool HAVE


A static analyzer parses the source code to generate static summary trees (SSTs).
An instrumentation tool inserts code to intercept events during execution.
source code
tic
nary
es
instrumentation tool
speculator
hybrid trees
Hybrid Conflict-edge algorithm
Instrumented code
dynamic monitor
dynamic trees
error report

f Tool HAVE


A static analyzer parses the source code to generate static summary trees (SSTs).
source code
static summary trees
An instrumentation tool inserts code to intercept events during execution.
An intra-procedural speculator generates speculations for the unexecuted branches from SSTs and combines them with dynamic trees to form hybrid trees.
instrumentation tool
speculator
hybrid trees
Hybrid Conflict-edge algorithm
Instrumented code
dynamic monitor
dynamic trees
error report
A dynamic monitor intercepts events and builds dynamic trees during execution.

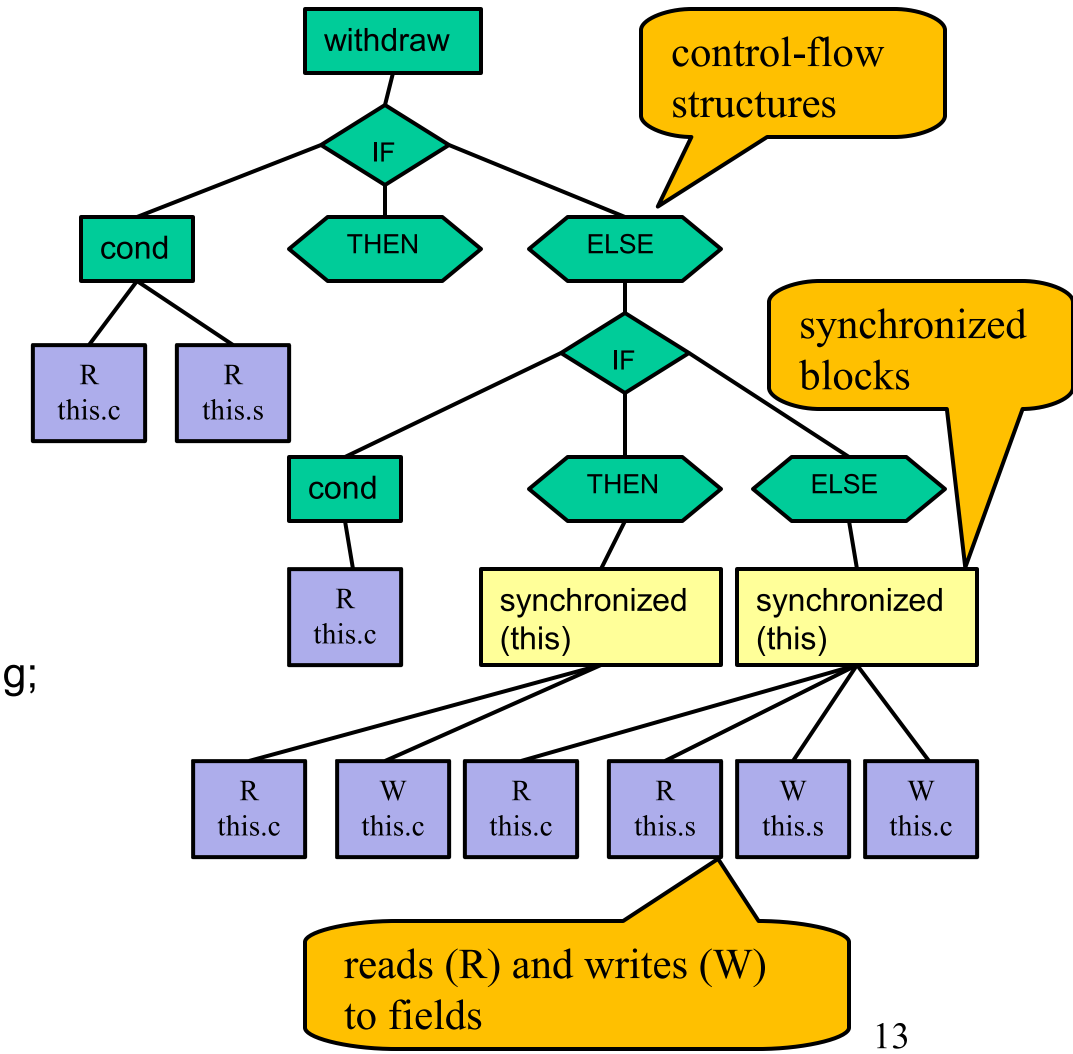

f Tool HAVE
A static analyzer parses the source code to generate static summary trees (SSTs).
source code
An instrumentation tool inserts code to intercept events during execution.
tic
nary
es
An i
gene
unex
and c
trees to form hybrid trees.
A detector analyzes the hybrid trees for atomicity violations using the hybrid conflict-edge algorithm
instrumentation tool
speculator
hybrid trees
Hybrid Conflict-edge algorithm
Instrumented code
dynamic monitor
dynamic trees
error report
A dynamic monitor intercepts events and builds dynamic trees during execution.

Static Analyzer

- The static analyzer parses source code to construct static summary trees.

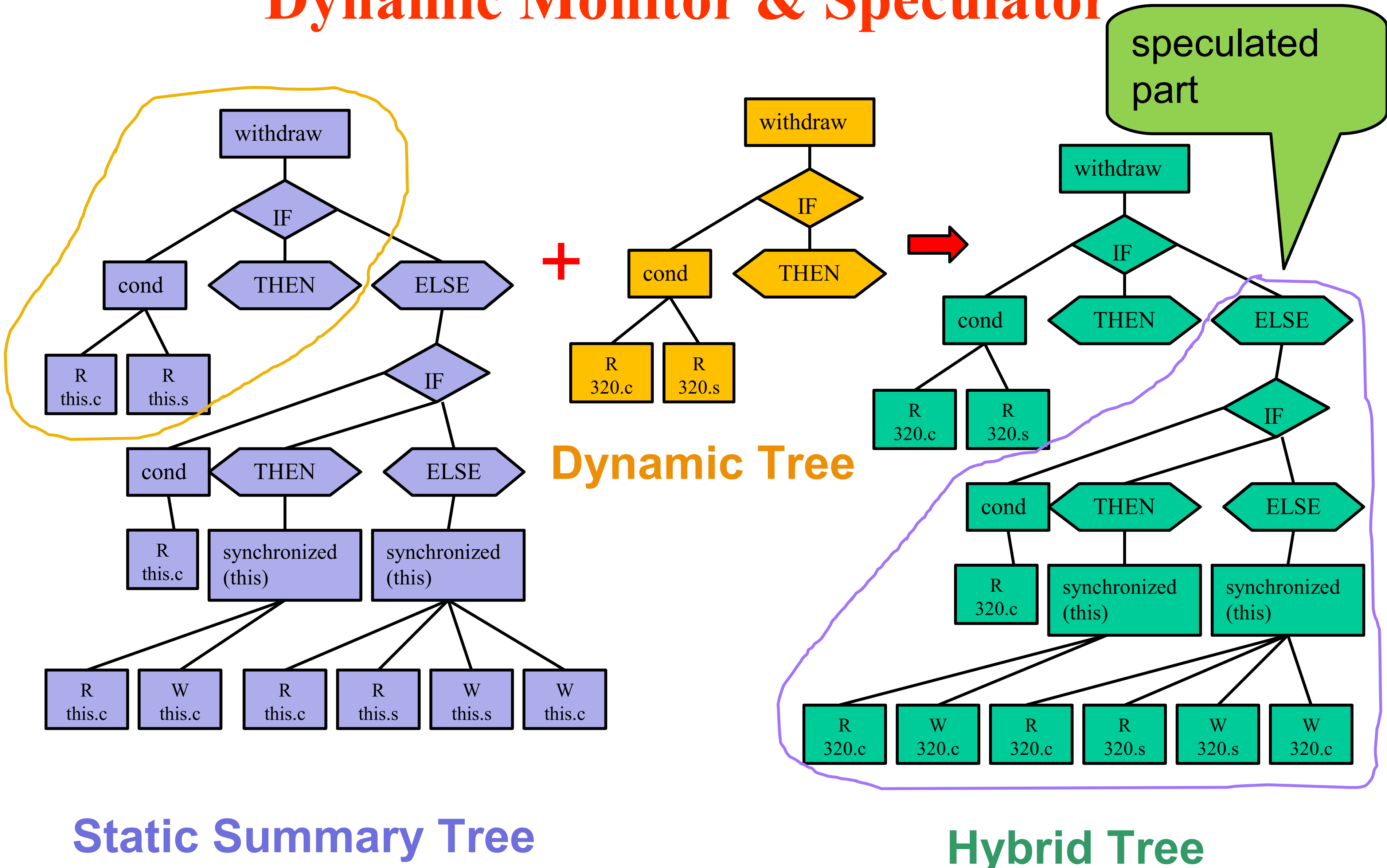
```
class Account {
   int checking, saving;

   public void withdraw(int w) {
      if ((this.checking + this.saving) < w)
         print("Insufficient balance");
      else if (this.checking >= w)
         synchronized(this)
            this.checking -= w;
      else
          synchronized(this) {
               this.saving -= w - this.checking;
               this.checking = 0;
          }
   }
}
```



Not illustrated: assignments to reference variables.

Dynamic Monitor & Speculator

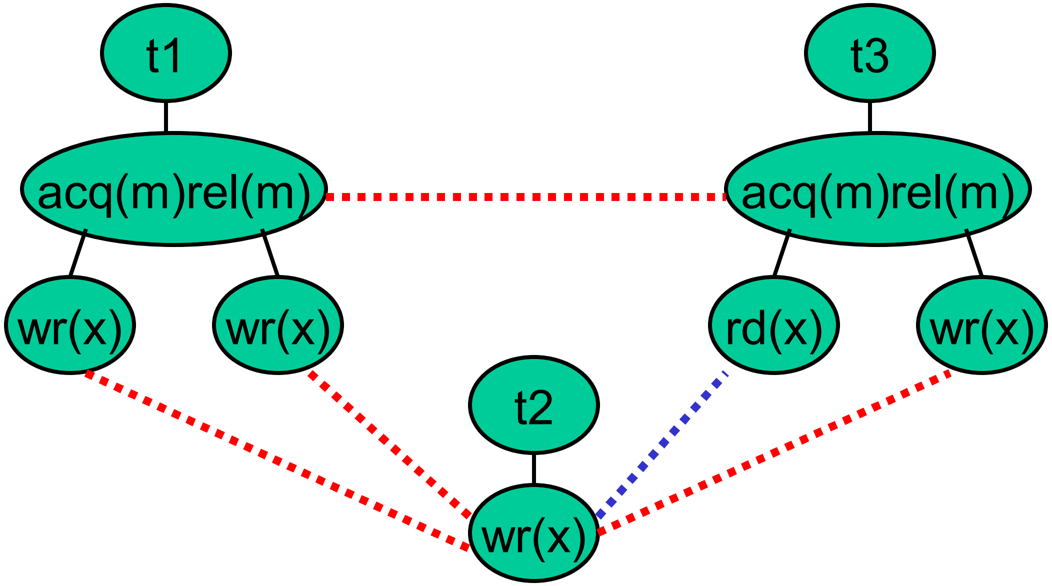


Outline

- Introduction to atomicity
- Integrate dynamic and static analysis to detect atomicity violations
- Conflict-edge algorithm
- Experiments and conclusions

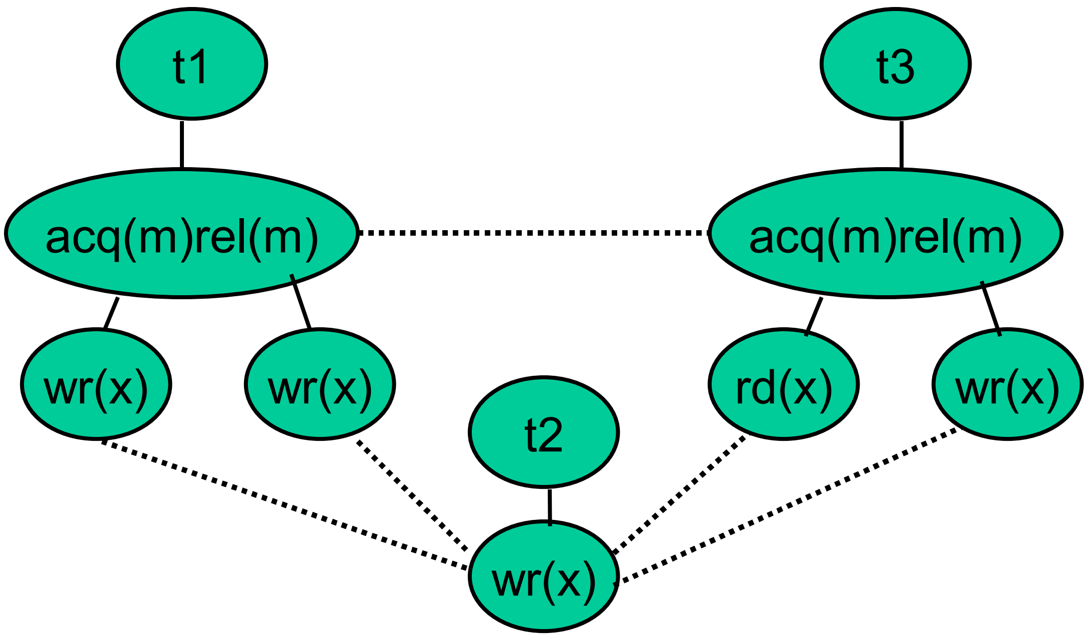
Add Inter-Edges Between Hybrid Trees

- Add **inter-edges** (also called **conflict edges**) between hybrid trees:
  - wr(x) – rd(x): the read event can read the value written by the write.
    - “can” means “in some feasible permutation of the execution”.
  - wr(x) – wr(x): both write to the same variable.
- If a common lock is held at both events, connect the lock nodes instead of the access nodes, because they cannot occur concurrently.



Our Previous Approach: Commit-Node Algorithm

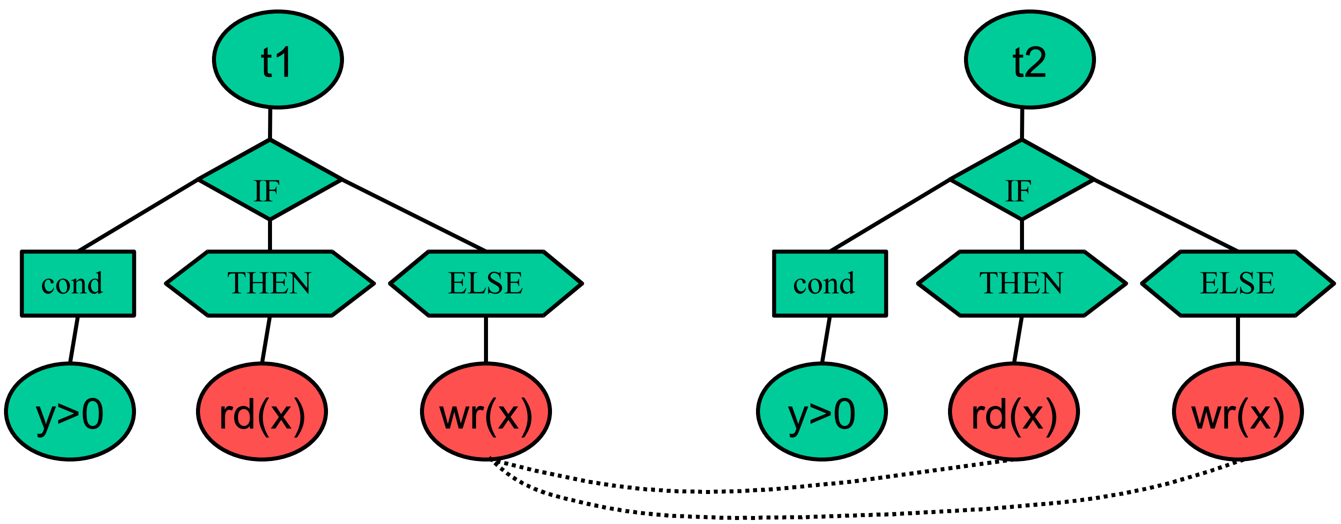
- **Communication node**: end-point of an inter-edge.
- **Commit node**: a communication node without communication node descendants.



The commit-node algorithm appeared in [Wang and Stoller, PPoPP 2006].

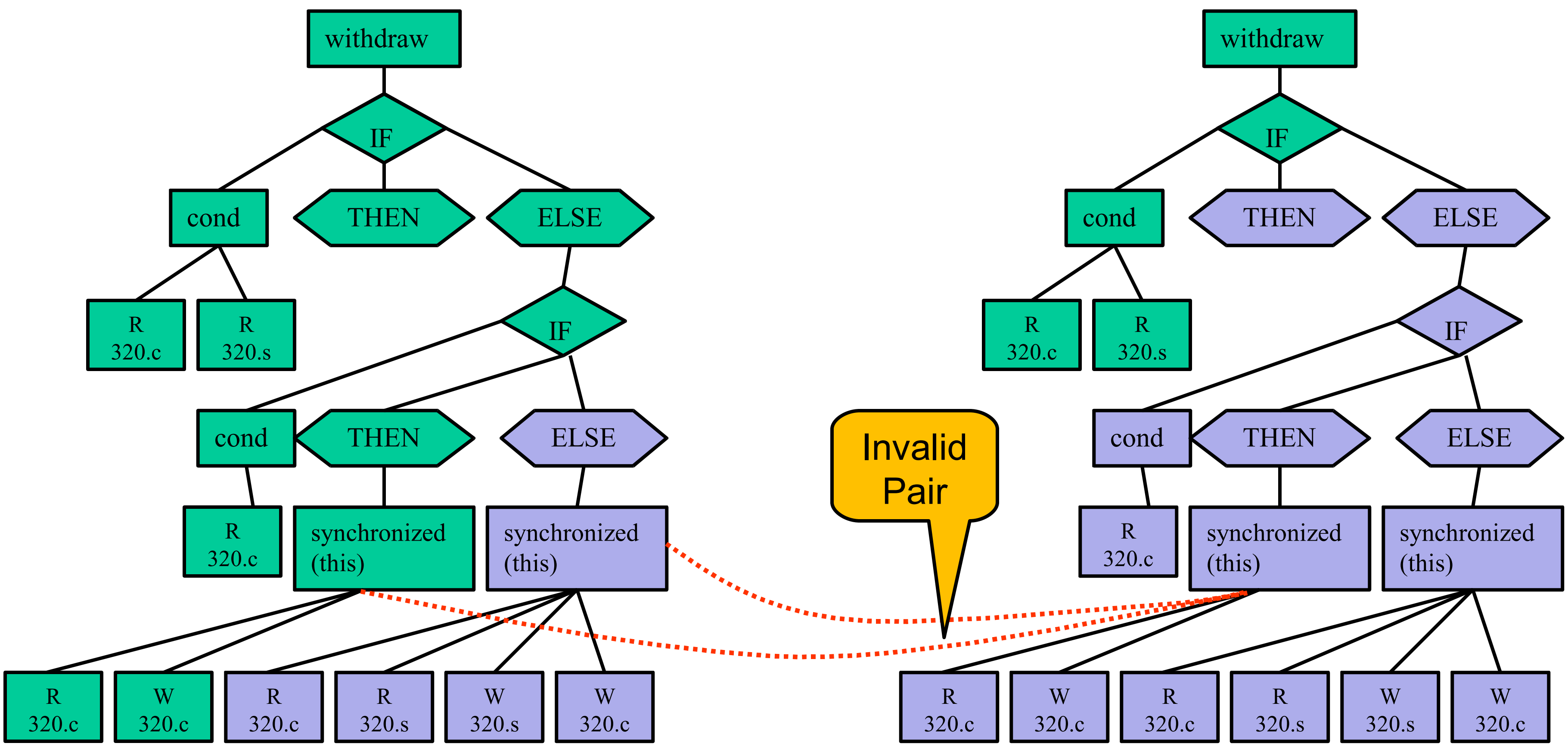
The Commit-Node Algorithm May Give False Alarms for Hybrid Trees.

- Motivation:
  - All inter-edges between dynamic trees (i.e., generated by purely dynamic analysis) can coexist in the same execution.
  - But this is not the case for inter-edges between hybrid trees.



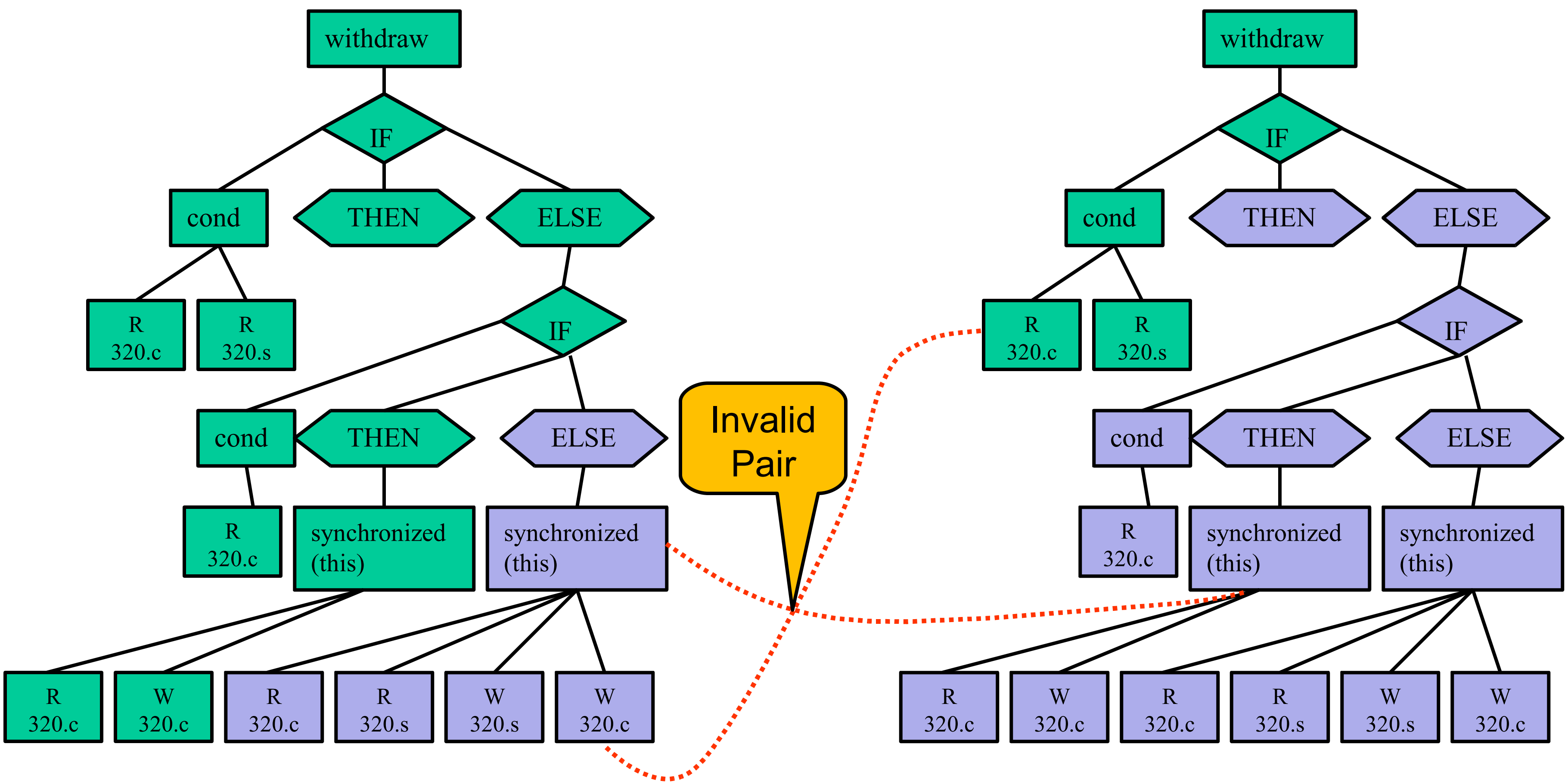
Valid Inter-Edge Pair

- A pair (e, e') of inter-edges is valid for hybrid tree *t* if
  - e and e' are compatible, and
  - e is not an ancestor of e' in *t*, vice versa, and
  - e and e' are incident on different nodes of *t*



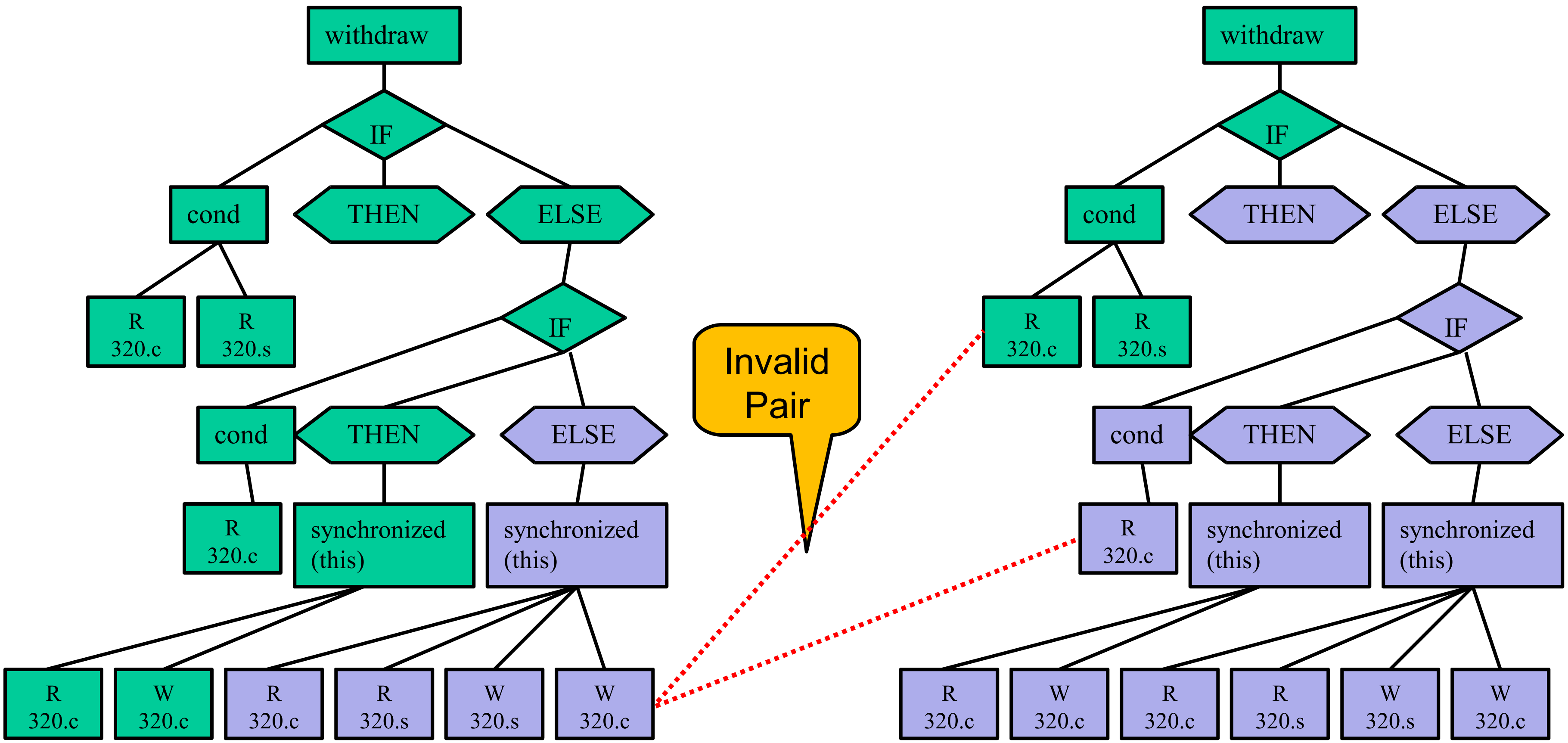
Valid Inter-Edge Pair

- A pair (e, e') of inter-edges is valid for hybrid tree *t* if
  - e and e' are compatible, and
  - e is not an ancestor of e' in *t*, vice versa, and
  - e and e' are incident on different nodes of *t*



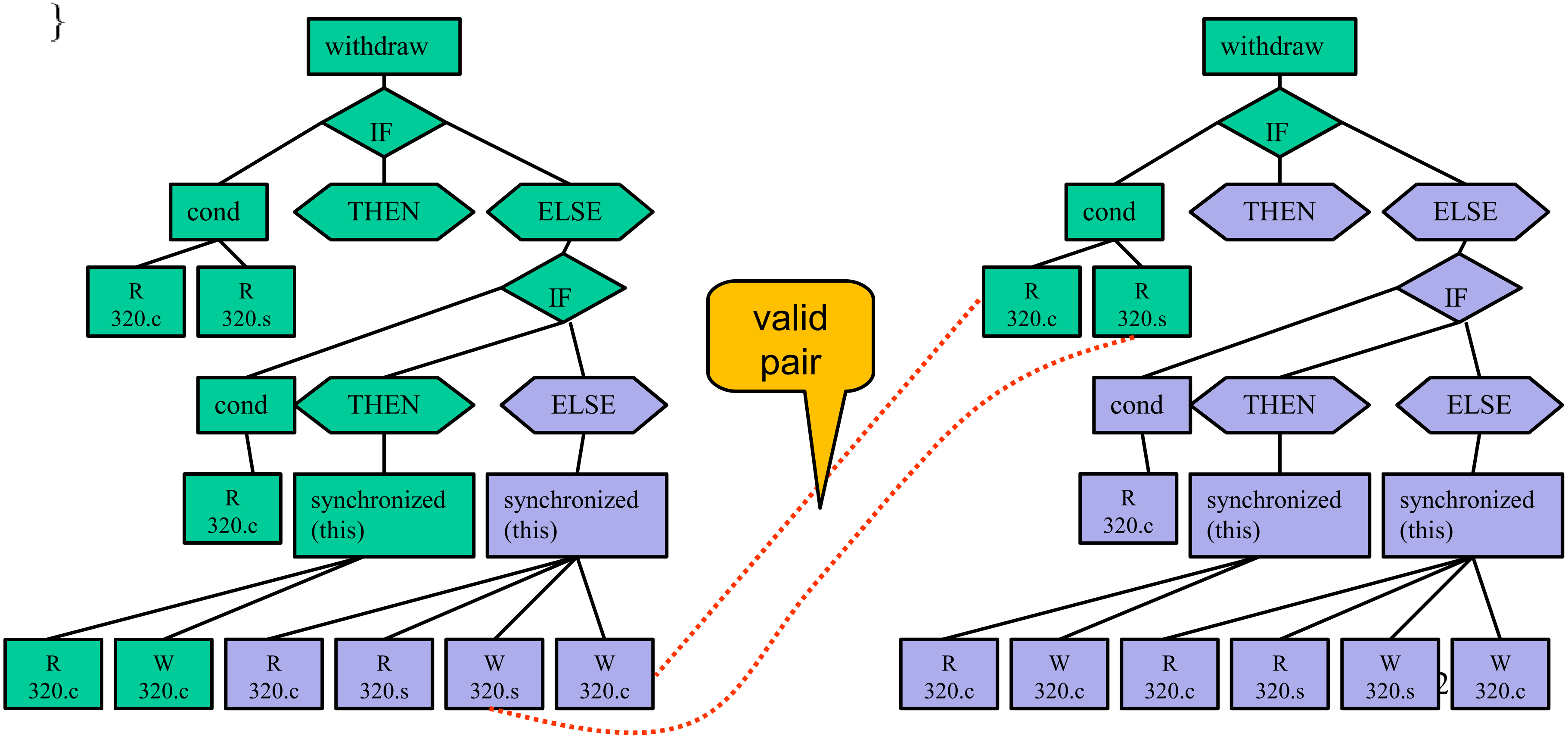
Valid Inter-Edge Pair

- A pair (e, e’) of inter-edges is valid for hybrid tree *t* if
  - e and e’ are compatible, and
  - e is not an ancestor of e’ in *t,* vice versa, and
  - e and e’ are incident on different nodes of *t*



Conflict-Edge Algorithm

CheckAtomicityViolations() {
  for each transactional hybrid tree t **do**
    for each valid inter-edge pair (e, e') of t **do**
      if only two hybrid trees including t are connected by e and e' **then**
        report a potential atomicity violation involving e and e'.
      else
        if $\exists$ a valid cycle c of inter-edges containing e and e' **then**
          report a potential atomicity violation involving c.
}



Outline

- Introduction to atomicity
- Integrate dynamic and static analysis to detect atomicity violations
- Conflict-Edge Algorithm
- Experiments and conclusions

Summary of Experiments

- Evaluated on 9 benchmarks totaling 284 KLOC
  - Includes Apache Tomcat and Jigsaw (web server from W3C)
- **Heuristic:** public or synchronized methods are expected to be atomic.

	Purely dynamic approach [Wang & Stoller, PPoPP 2006]	Hybrid approach
Accuracy	11 bugs (non-atomic transaction) involving 90 locations in source code 16 benign alarms 0 false alarms	13 bugs (non-atomic transaction) involving 145 locations in source code 16 benign alarms 0 false alarms
Performance (overhead)	Average 3.6x slowdown (except for the program TSP, with 35x slowdown)	Average 16.5x slowdown (except for the program TSP, with 167x slowdown)

A Possible Bug Found in TSP (parallel traveling salesman algorithm)

Thread: 11

```
static void set_best(int best, int[] path) {
        ....
        synchronized (TspSolver.MinLock) {
                if (best < TspSolver.MinTourLen) {
                        ...
                        TspSolver.MinTourLen = best;
                        ...
                }
        }
}
```

executed

```
static void set_best(int best, int[] path) {
        ....
        synchronized (TspSolver.MinLock) {
                if (best < TspSolver.MinTourLen) {
                        ...
                        TspSolver.MinTourLen = best;
                        ...
                }
        }
}
```

executed

```
[TspSolver.set_best(TspSolver.java)
synchronized@6658066@line:path=n/a@object:TspSolver.MinLock
TspSolver.set_best(TspSolver.java:1745)
TspSolver.visit_nodes(TspSolver.java:2674)
TspSolver.visit_nodes(TspSolver.java:2739)
TspSolver.visit_nodes(TspSolver.java:2739)
TspSolver.visit_nodes(TspSolver.java:2739)
TspSolver.visit_nodes(TspSolver.java:2739)
TspSolver.visit_nodes(TspSolver.java:2739)
TspSolver.visit_nodes(TspSolver.java:2739)
TspSolver.visit_nodes(TspSolver.java:2739)
TspSolver.visit_nodes(TspSolver.java:2739)
TspSolver.visit_nodes(TspSolver.java:2739)
TspSolver.visit_nodes(TspSolver.java:2739)
TspSolver.visit_nodes(TspSolver.java:2739)
TspSolver.recursive_solve(TspSolver.java:2504)
TspSolver.Worker(TspSolver.java:2791)
TspSolver.run(TspSolver.java:2518)]
```

Thread: 10

speculation

```
static void split_tour(int curr_ind) {
        ...
        synchronized (TspSolver.TourLock) {
                ...
                if (curr.last != Tsp.TspSize - 1) {
                        ...
                        for (i = 0; i < Tsp.TspSize; i++) {
                                ...
                                t3 = curr.lower_bound + wt <=
TspSolver.MinTourLen;
                                        ...
                        }
                }
        }
}
```

```
[TspSolver.split_tour(TspSolver.java:2224)
TspSolver.find_solvable_tour(TspSolver.java:1135)
TspSolver.get_tour(TspSolver.java)
synchronized@10736847@line:path=n/a@object:TspSolver.TourLock
TspSolver.get_tour(TspSolver.java:1287)
TspSolver.Worker(TspSolver.java:2784)
TspSolver.run(TspSolver.java:2518)]
```

Bugs Found in Jigsaw and Tomcat

- Jigsaw
  - The method `perform` in `httpd.java` has multiple atomicity violations regarding several fields, such as `LRUNode.next` and `ResourceStoreImpl.resources`.
- Tomcat
  - Potential atomicity violations are found on the fields *`StringCache.accessCount`* and *`StringCache.hitCount`* in the method *`toString(ByteChunk bc)`* of *`StringCache.java`*.
  - We classify this atomicity violation as a bug, because it may cause the statistics to be inaccurate, even though this inaccuracy does not cause other incorrect behavior.
- These bugs are not detected by the purely dynamic approach, because some of the field accesses are in speculatively executed branches.

Conclusions and Future Work

- Conclusions
  - We developed a new approach to enhance dynamic analysis with results from static analysis.
  - It improves code coverage and hence effectiveness at finding errors (atomicity violations).
    - Care is needed to avoid false alarms due to incompatible speculative branches.
  - In experiments, our new algorithm scales almost as well as our purely dynamic algorithm.
- Future work
  - Extend the static analysis to be inter-procedural.
  - Design more optimizations to reduce overhead.