

JASMIN JAHIC, SIMON BARNER

jj542@cam.ac.uk

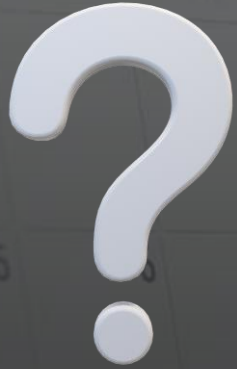
barner@fortiss.org

<https://jahic.github.io/hipeac2021>

09:30 - 13:00, 18.01.2021,  
BUDAPEST, HUNGARY

# HANDLING CONCURRENCY IN EMBEDDED SOFTWARE SYSTEMS FROM ARCHITECTURAL POINT OF VIEW: PART 3

# AGENDA



9:30

Session 1: Fundamental Issues with Concurrency in Embedded Software Systems from Architectural Point of View

10:30

10:45

Session 2: Modelling and DSE Methods for Mixed-Critical Software Systems using Multicore Architectures

11:45

12:00

Session 3: Synchronization in Concurrent Software is an Architectural Decision

13:00

## SESSION 3

12:00

Beyond the “problem with threads”

Synchronisation mechanisms

Concurrency bugs

Testing concurrent software - finding  
concurrency bugs

12:45

# SPEEDUP OF A SINGLE TASK

## Set#3

Core affinity  
Scheduling policy  
Interrupts

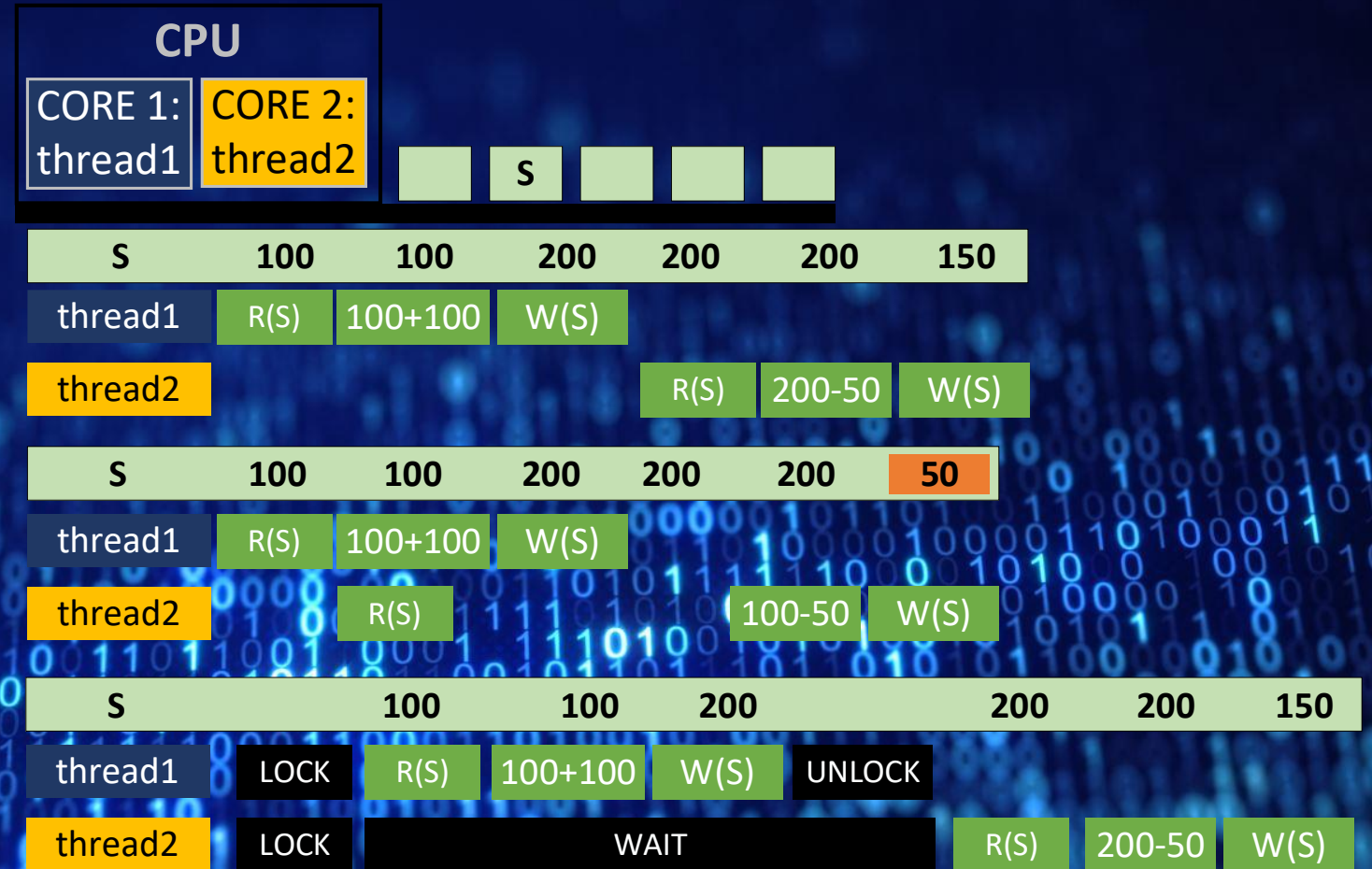
## Set#4

Ways and means to partition software -  
partitioning strategy  
Thread start-up time  
Synchronisation  
Liveness  
Concurrency bugs  
Bugs that exist on execution paths possible  
only because of concurrency

ID	006	Status	
Name	...	Owner	
Quality	Average case execution time – single task – partitioning – dependencies, shared memory	Stakeholders	
		Quantification	
Environment	Task is executing on a CPU	Execution time = $t$ ; #cores>1	
Stimulus	Partition the task into threads	#treads>1, set#4 params, set#3 params	
Response	Significantly reduced (by factor k) execution time	Execution time = $t/k$	



# CONCURRENCY BUG EXAMPLE



# THE PROBLEM WITH THREADS

[1] Edward A. Lee. 2006. *The Problem with Threads*. *Computer* 39, 5 (May 2006), 33–42. DOI:

<https://doi.org/10.1109/MC.2006.180>

[2] H. Sutter and J. Larus. *Software and the concurrency revolution*. *ACM Queue*, 3(7), 2005.

- “They (threads) discard the most essential and appealing properties of sequential computation: **understandability**, **predictability**, and **determinism**.” [1]
- “Nondeterminism should be explicitly and judiciously introduced where needed, rather than removed where not needed.” [1]
- “humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among even simple collections of partially ordered operations.” [2]

# THE PROBLEM WITH THREADS

[1] Edward A. Lee. 2006. *The Problem with Threads*. *Computer* 39, 5 (May 2006), 33–42. DOI:

<https://doi.org/10.1109/MC.2006.180>

[3] L. A. Stein. Challenging the computational metaphor: Implications for how we think. *Cybernetics and Systems*, 30(6), 1999

- Given a program and an initial state, any two programs  $p$  and  $p'$  (that compute the same function) can be compared. They are equivalent if they **halt** for the same initial states, and for such initial states, their **final state** is the same.
- Assume that  $p_1$  and  $p_2$  execute concurrently in a multithreaded fashion. Pair  $(p_1, p_2)$  is equivalent to  $(p'_1, p'_2)$  if **all interleavings** halt for the same initial state and yield the same final state.
- **BONUS**: we have to know about **all other threads** that might execute.
- #threads  $n$ , #instructions  $i$ ; #interleavings =  $2^{n*i}$
- "with threads, there is no useful theory of equivalence" [1]
- "achieving reliability and predictability using threads is essentially impossible for many applications" [1]
- "to replace the conventional metaphor a sequence of steps with the notion of a community of interacting entities" [3]

# ARCHITECTURAL PATTERNS AND ANTI-PATTERNS

*Software Architecture  
Measurement—Experiences from a  
Multinational Company, W. Wu, Y. Cai,  
R. Kazman et al., ECSA 2018*

- Patterns: Reuse previous knowledge
  - Problem; Solution; Advantages; Disadvantages
- Anti-patterns: Avoid, bad smells, technical debt
  - Design; Threshold; Severity
- Unstable dependency: A subsystem (component) that depends on other subsystems that are less stable, with a possible ripple effect of changes in the project.
- Cyclic dependency (CD): A subsystem (component) that is involved in a chain of relations that break the desirable acyclic nature of a subsystem dependency structure.
- Hotspot Patterns: Implicit Cross-Module Dependency, Cross-Module Cycle, and Cross Package Cycle.

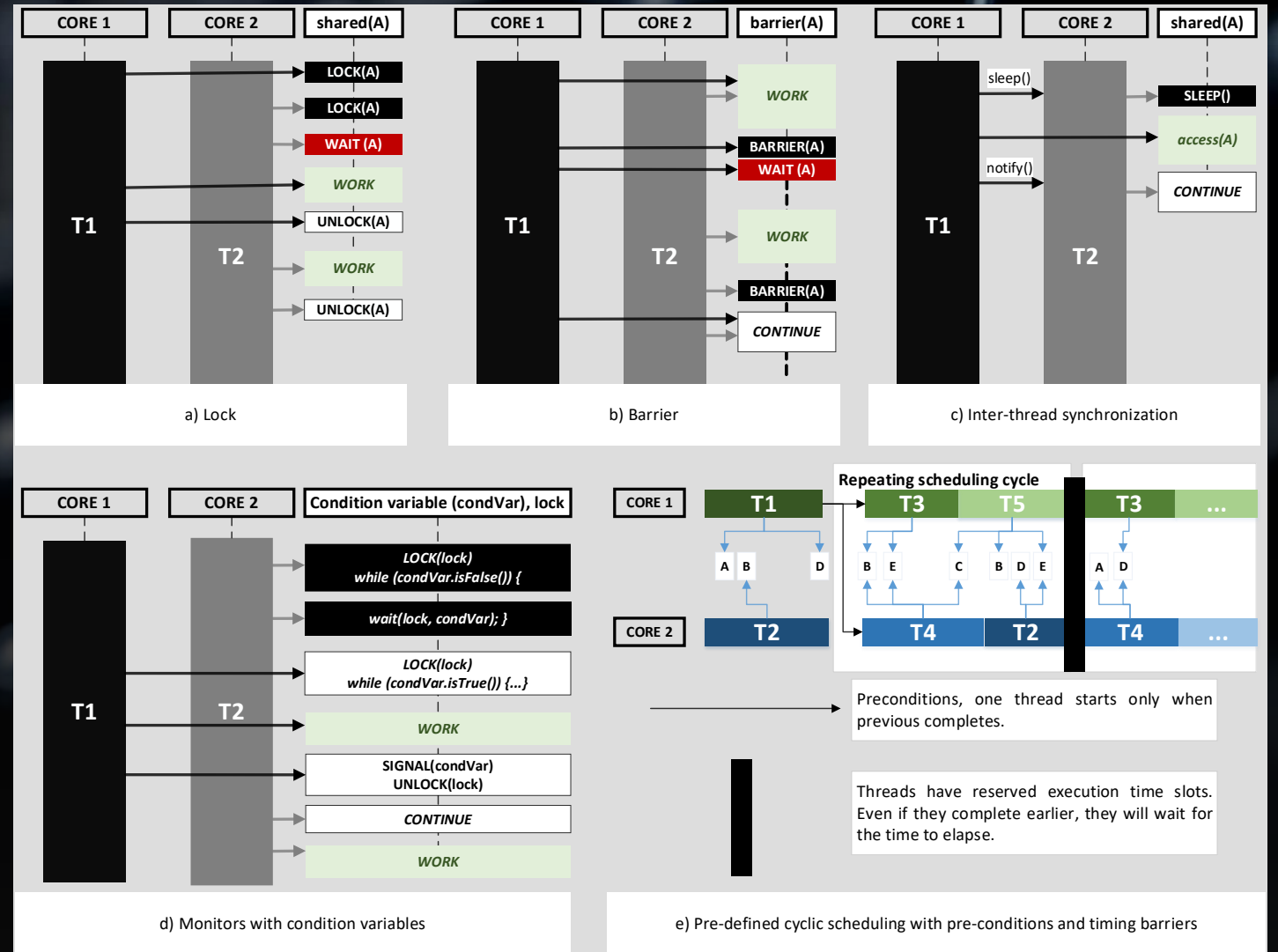


# THREADS FROM SOFTWARE ARCHITECTURE PERSPECTIVE

[1] J. Jahić, V. Kumar, P. O. Antonino and G. Wirrer, "Testing the Implementation of Concurrent AUTOSAR Drivers Against Architecture Decisions," 2019 IEEE International Conference on Software Architecture (ICSA), Hamburg, Germany, 2019, pp. 171-180, doi: 10.1109/ICSA.2019.00026.

- Cohesion: the degree to which the elements inside a module belong (logically) together
- Coupling: A measurement of interdependence between components.
  - Data coupling, control coupling, temporal coupling...
  - Example: Does one component need to understand what is happening in other component in order to use it?
- Usual goals: High cohesion and low coupling
- Threads are implicitly coupled:
  - Directly: One thread needs to know how all other threads access shared resources
  - Indirectly: Shared hardware
- The problem with threads: **there are no interfaces for accessing shared memory** – not transparent at all [1]

# SYNCHRONISATION MECHANISMS





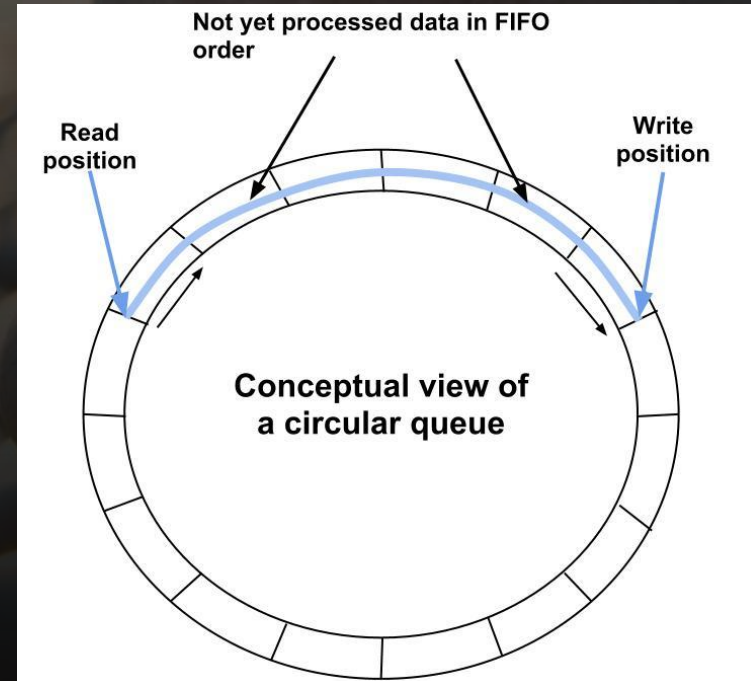
# LOGICAL EXECUTION TIME (LET) SCHEDULING

*Kirsch C.M., Sokolova A. (2012) The Logical Execution Time Paradigm. In: Chakraborty S., Eberspächer J. (eds) Advances in Real-Time Systems. Springer, Berlin, Heidelberg.  
[https://doi.org/10.1007/978-3-642-24349-3\\_5](https://doi.org/10.1007/978-3-642-24349-3_5)*

- Taming concurrency non-determinism
- Program reads input in zero time
- Program executes
- Program writes output in zero time
- Execution time = LET
- “if the program completes execution before the deadline, writing output is delayed until the deadline, i.e., until the LET has elapsed”
- The LET deadline is an upper bound, but also a lower bound, at least, logically.
- “In the LET model, using a faster machine does therefore not result in (logically) faster program execution but only in decreased machine utilization”

# NON-BLOCKING SYNCHRONIZATION

- Atomic Compare and Swap (CAS)
- Load Linked, Store Conditional (LL/SC)
- Data structures:
  - Ring
  - Buffer
  - Queue

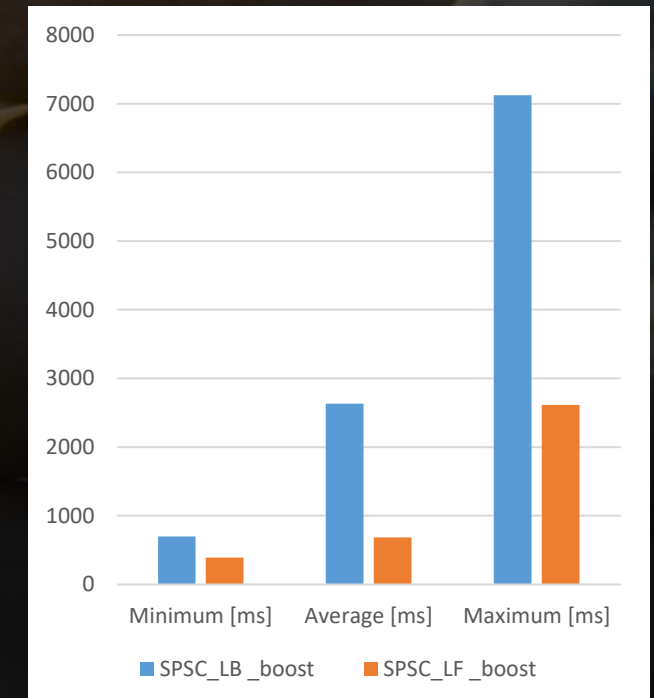
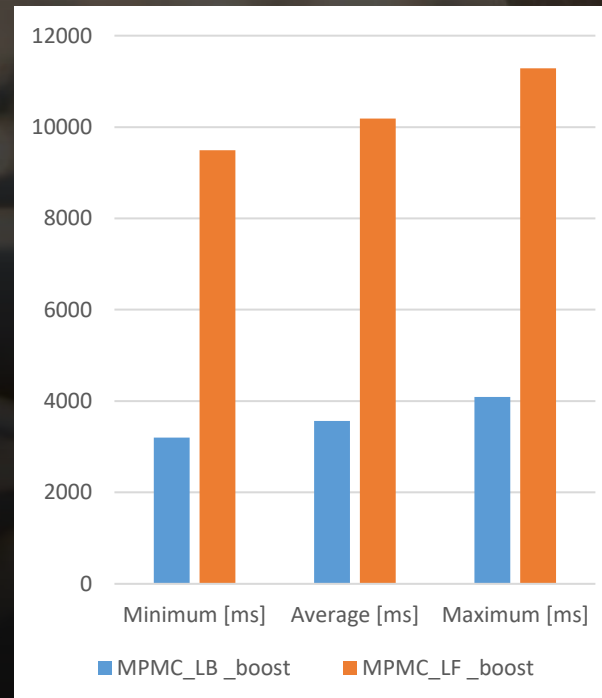




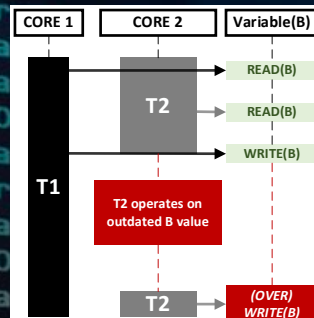
# NON-BLOCKING SYNCHRONIZATION

*J. Jahić, K. Ali, M. Chatrangoon, and N. Jahani. 2019. (Dis)Advantages of Lock-free Synchronization Mechanisms for Multicore Embedded Systems. International Conference on Parallel Processing: Workshops (ICPP 2019) DOI: <https://doi.org/10.1145/3339186.3339191>*

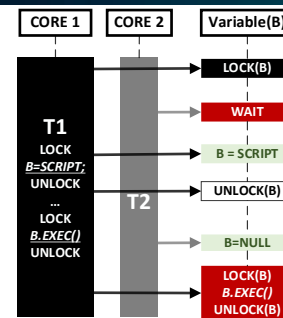
- Simple code (<https://github.com/KhuraamAli/JAM-Benchmark>)
- Multi-Producer/Multi-Consumer pattern (**MPMC**)
  - Producers = 4; Consumers = 4;
- Single-Producer/Single-Consumer Ring Buffer (**SPSC**)
- Each test executed 1000 times
- Lock based (**LB**): `std::mutex.lock(); std::mutex.unlock();`
- Lock free (**LF**): `boost::lockfree::queue`



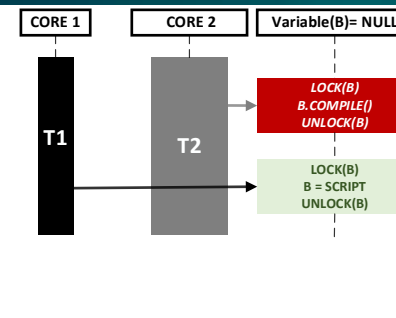
# CONCURRENCY BUGS



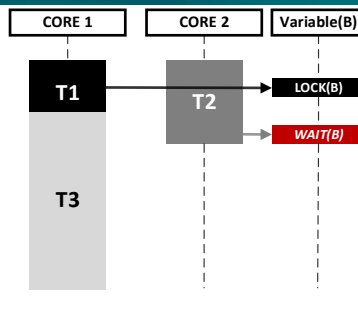
a) Data races



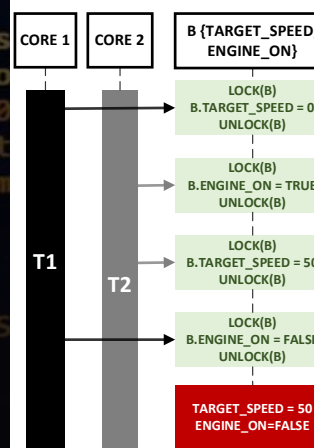
b) Atomicity violation (race free) - Programmers expect that some code regions will execute atomically.



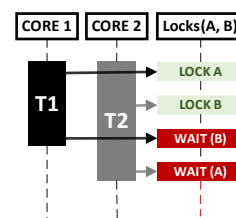
c) Order violation bugs - occur when the intended order between two operations (e.g., initialization is expected to execute before compile) is flipped. It is expected that T1 will first initialize variable B.



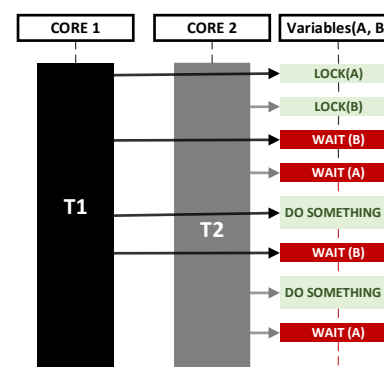
d) Priority inversion – Priorities ( $T_2 > T_3 > T_1$ ). T2 can make T1 to release the lock on lock on B. However, T1 has been already preempted by T3. Now T2 waits on T3 and T1, but it has a higher priority then both.



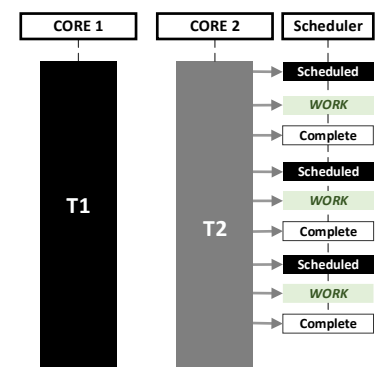
e) Multivariable concurrency bugs – logical inconsistency. B is a structure.



f) Deadlock



g) Livelock - states of the threads involved in the livelock constantly change with regard to one another. However, none is progressing.



h) Starvation - Thread T2 has a higher priority. The scheduler never schedules the T1 for execution.

# ATOMICITY VIOLATION EXAMPLE

```
... for object to mirror...
mirror_mod.mirror_object = ...
operation == "MIRROR_X":
mirror_mod.use_x = True
mirror_mod.use_y = False
mirror_mod.use_z = False
operation == "MIRROR_Y":
mirror_mod.use_x = False
mirror_mod.use_y = True
mirror_mod.use_z = False
operation == "MIRROR_Z":
mirror_mod.use_x = False
mirror_mod.use_y = False
mirror_mod.use_z = True
```

• t1:

• ...

• lock();

• `object=new O();`

• unlock()

• ...

• ...

• lock();

• `object.method1();`

• unlock()

• t2:

• ...

• lock();

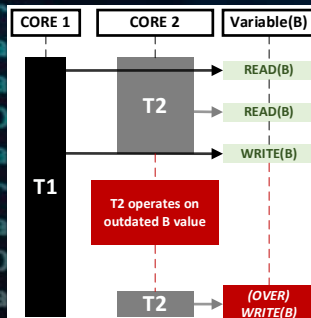
• `object=NULL;`

• unlock()

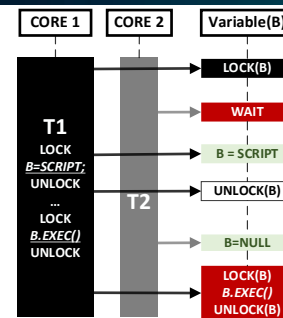
• ...



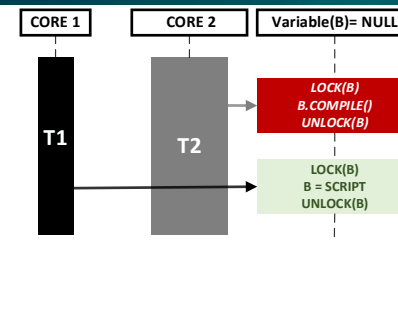
# CONCURRENCY BUGS



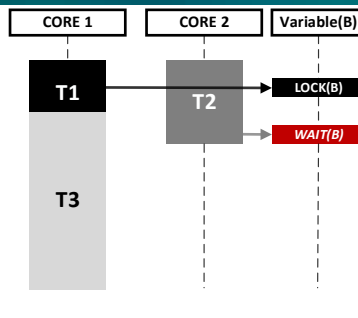
a) Data races



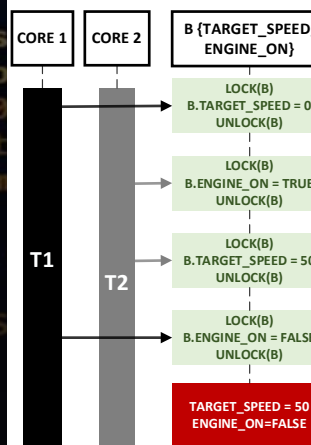
b) Atomicity violation (race free) - Programmers expect that some code regions will execute atomically.



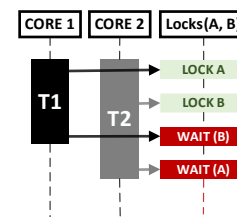
c) Order violation bugs - occur when the intended order between two operations (e.g., initialization is expected to execute before compile) is flipped. It is expected that T1 will first initialize variable B.



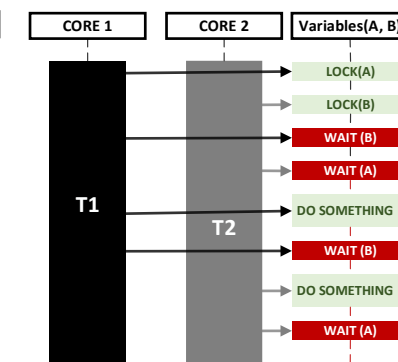
d) Priority inversion – Priorities ( $T_2 > T_3 > T_1$ ). T2 can make T1 to release the lock on lock on B. However, T1 has been already preempted by T3. Now T2 waits on T3 and T1, but it has a higher priority then both.



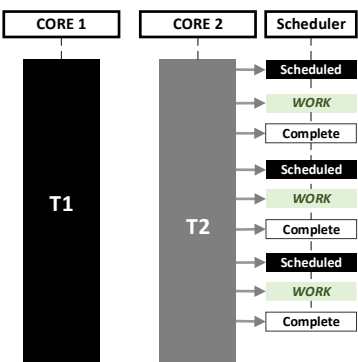
e) Multivariable concurrency bugs – logical inconsistency. B is a structure.



f) Deadlock



g) Livelock - states of the threads involved in the livelock constantly change with regard to one another. However, none is progressing.



h) Starvation - Thread T2 has a higher priority. The scheduler never schedules the T1 for execution.

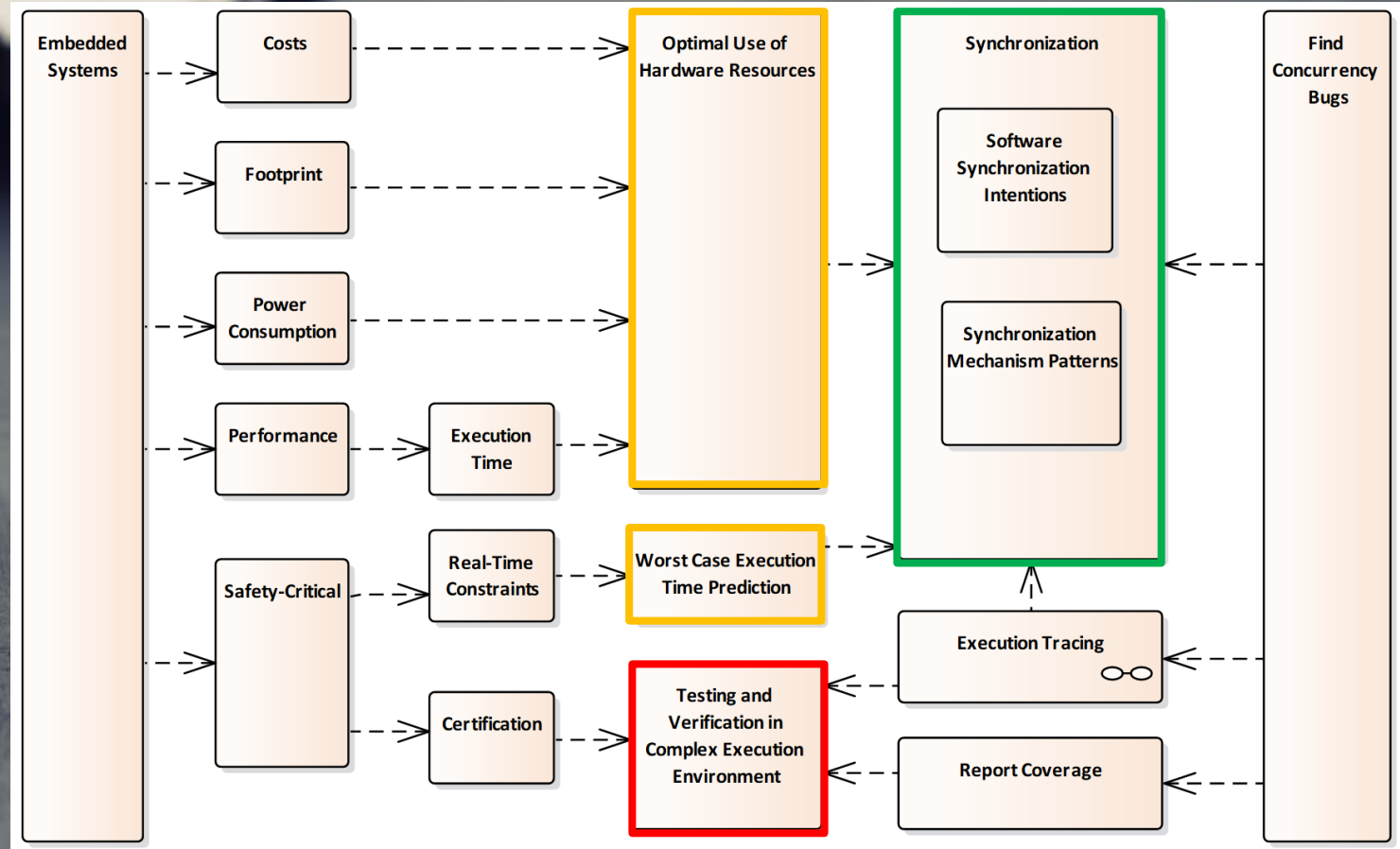


# BUGS CAUSED BY CONCURRENCY

*a - shared variable*

- t1:
  - lock(); a=10; unlock();
  - lock();
  - if(a!=10) { b=9; }
  - else { b=10; }
  - unlock();
  - ...
  - lock();
  - if(a==10) { c=1/(b-9); }
  - unlock();

- t2:
  - lock(); a=10; unlock();



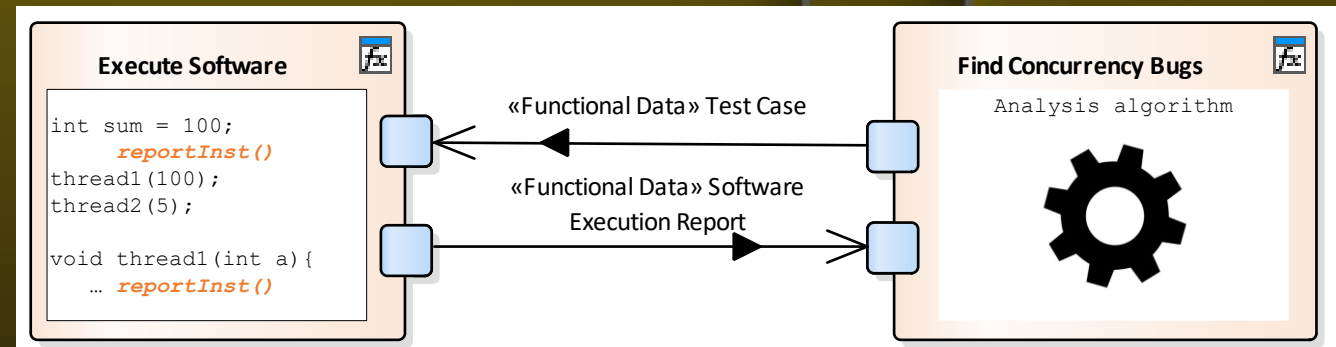
# SYNCHRONIZATION MECHANISMS AS ARCHITECTURAL DECISIONS

# FINDING CONCURRENCY BUGS

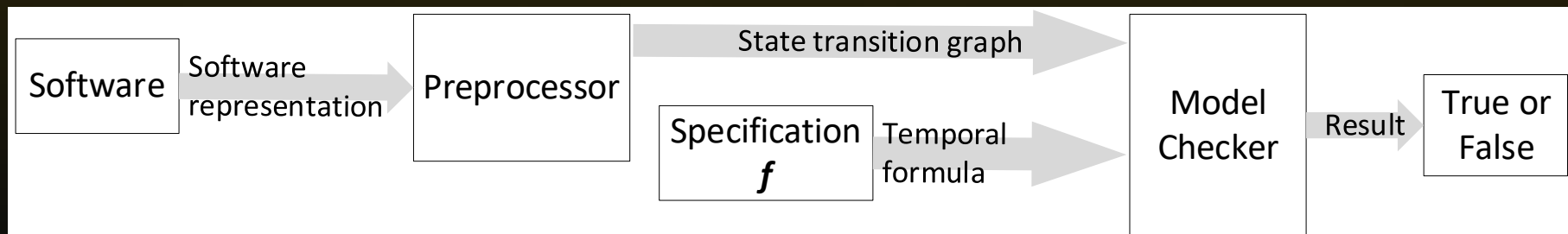
- Violation of synchronisation (atomicity) intentions
- Memory shared between threads
- Synchronisation mechanisms used by threads
- How to know developers' intentions?

# FINDING CONCURRENCY BUGS

- Static analysis
- Dynamic analysis (runtime monitoring)



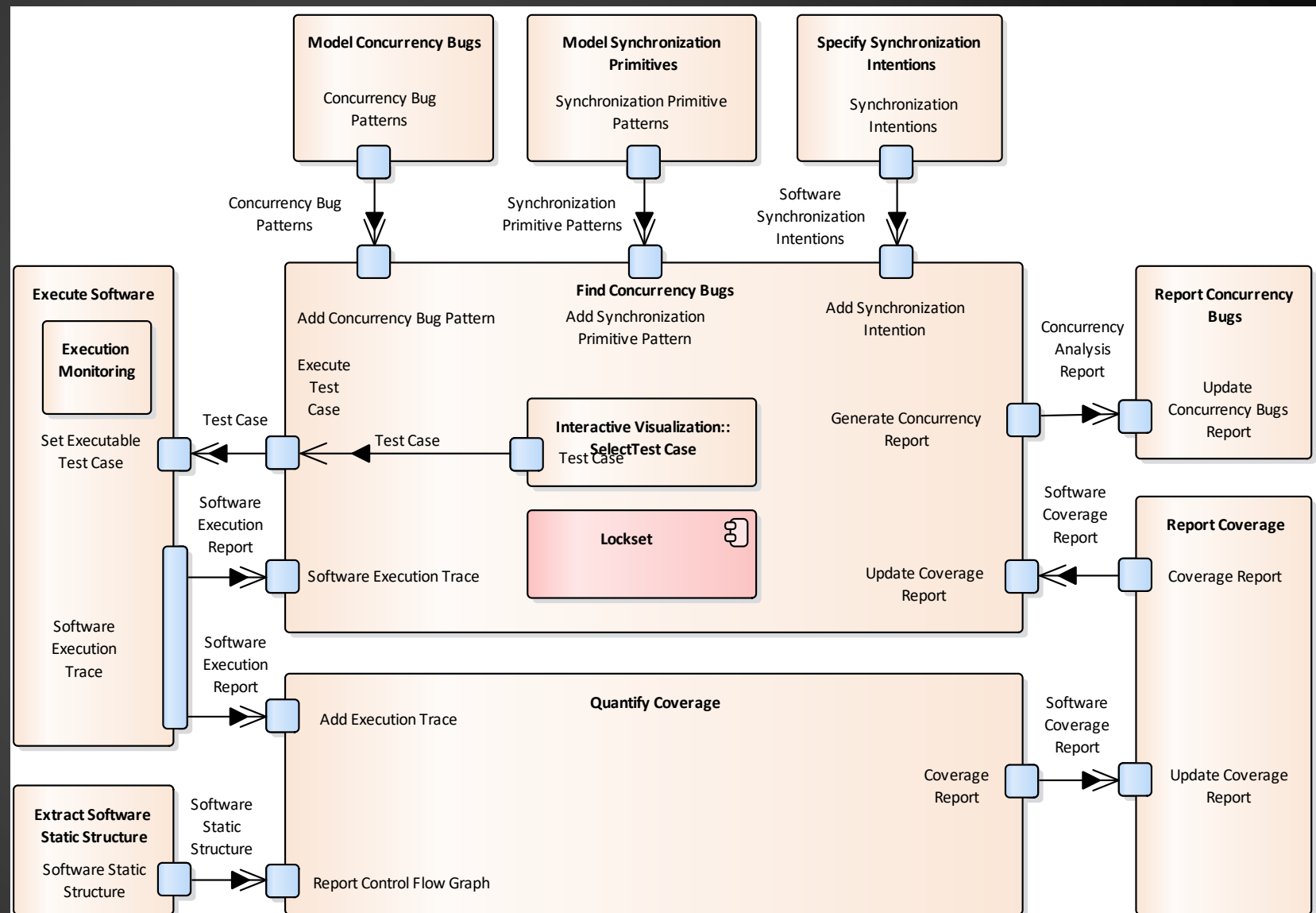
- Testing
- Model checking





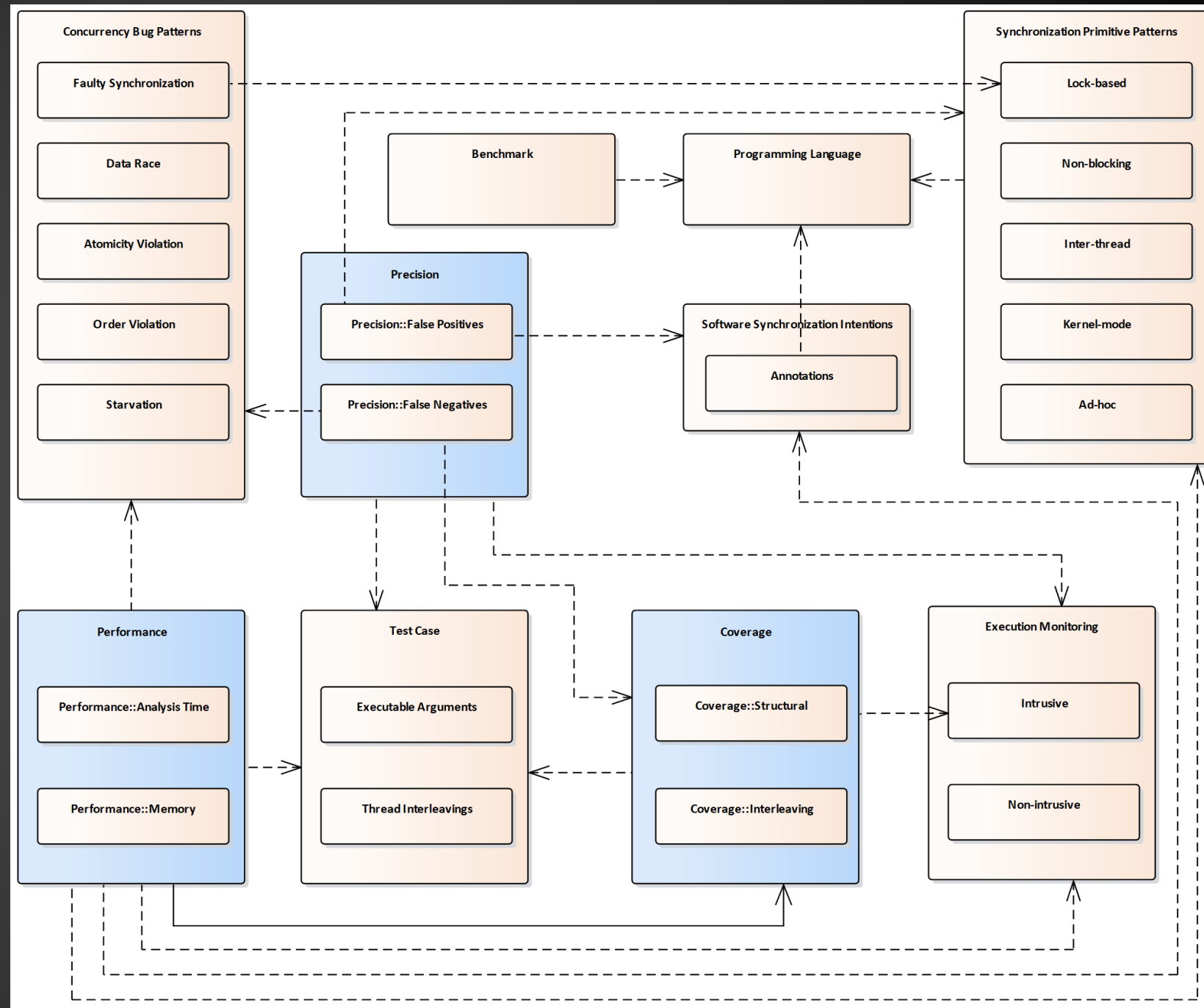
# EXECUTION MONITORING

*Jahić J., Bauer T., Kuhn T., Wehn N., Antonino P.O. (2020) FERA: A Framework for Critical Assessment of Execution Monitoring Based Approaches for Finding Concurrency Bugs. In: Arai K., Kapoor S., Bhatia R. (eds) Intelligent Computing. SAI 2020. Advances in Intelligent Systems and Computing, vol 1228. Springer, Cham. [https://doi.org/10.1007/978-3-030-52249-0\\_5](https://doi.org/10.1007/978-3-030-52249-0_5)*



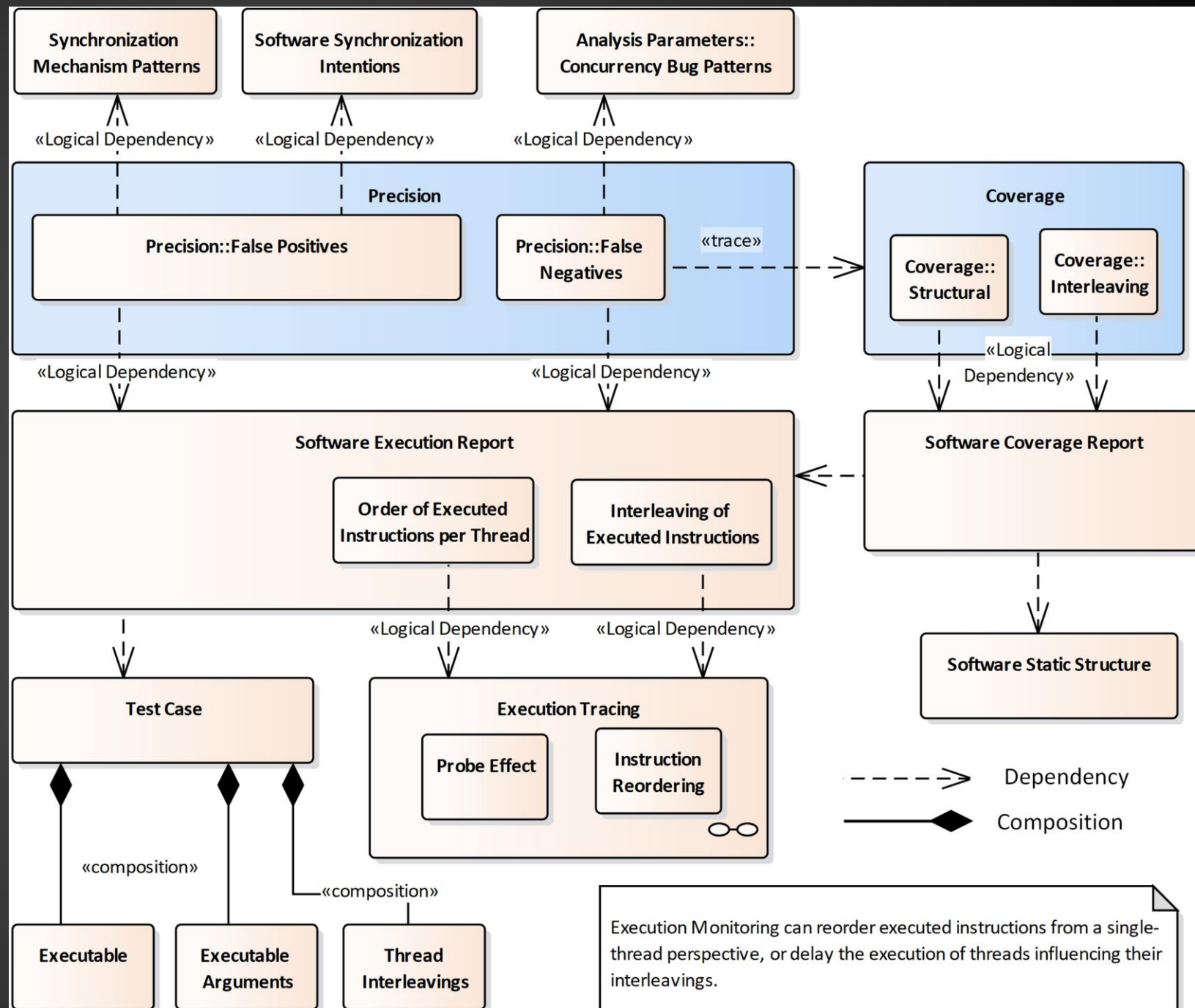
# EXECUTION MONITORING

Jahić J., Bauer T., Kuhn T., Wehn N., Antonino P.O. (2020) FERA: A Framework for Critical Assessment of Execution Monitoring Based Approaches for Finding Concurrency Bugs. In: Arai K., Kapoor S., Bhatia R. (eds) *Intelligent Computing. SAI 2020. Advances in Intelligent Systems and Computing*, vol 1228. Springer, Cham. [https://doi.org/10.1007/978-3-030-52249-0\\_5](https://doi.org/10.1007/978-3-030-52249-0_5)



# EXECUTION MONITORING: PRECISION

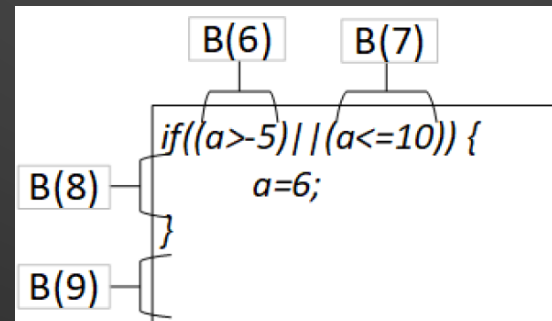
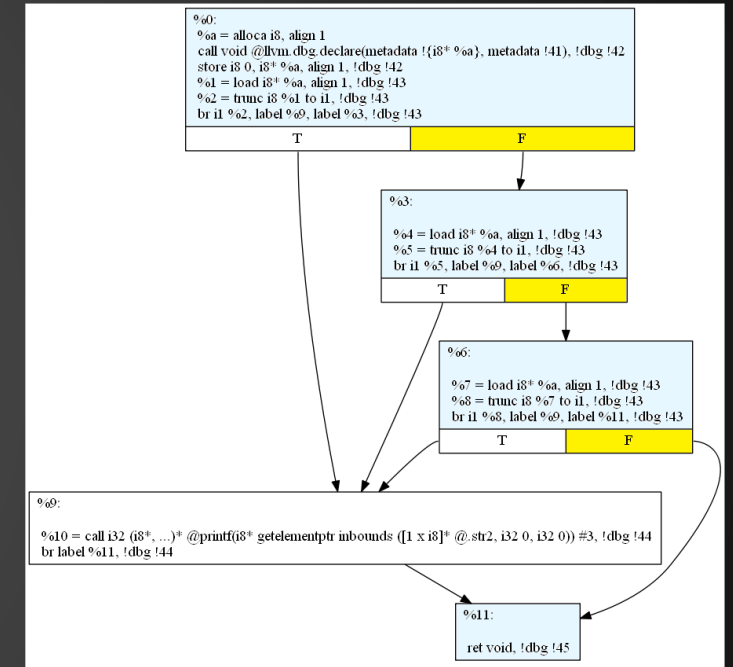
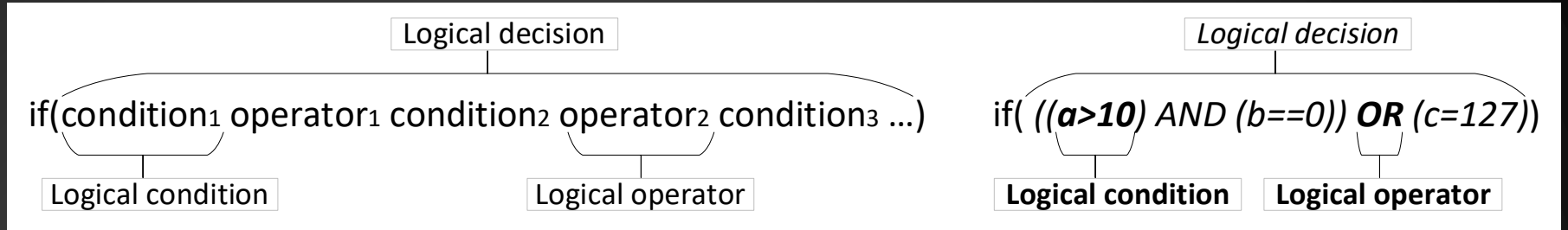
Jahić J., Bauer T., Kuhn T., Wehn N., Antonino P.O. (2020) FERA: A Framework for Critical Assessment of Execution Monitoring Based Approaches for Finding Concurrency Bugs. In: Arai K., Kapoor S., Bhatia R. (eds) *Intelligent Computing. SAI 2020. Advances in Intelligent Systems and Computing*, vol 1228. Springer, Cham. [https://doi.org/10.1007/978-3-030-52249-0\\_5](https://doi.org/10.1007/978-3-030-52249-0_5)



# CODE COVERAGE METRICS

Hayhurst Kelly J., Veerhusen Dan S., Chilenski John J., and Rierson Leanna K. 2001. *A Practical Tutorial on Modified Condition/Decision Coverage*. Technical Report. NASA Langley Technical Report Server

- Statement
- Condition
- Decision
- MC/DC



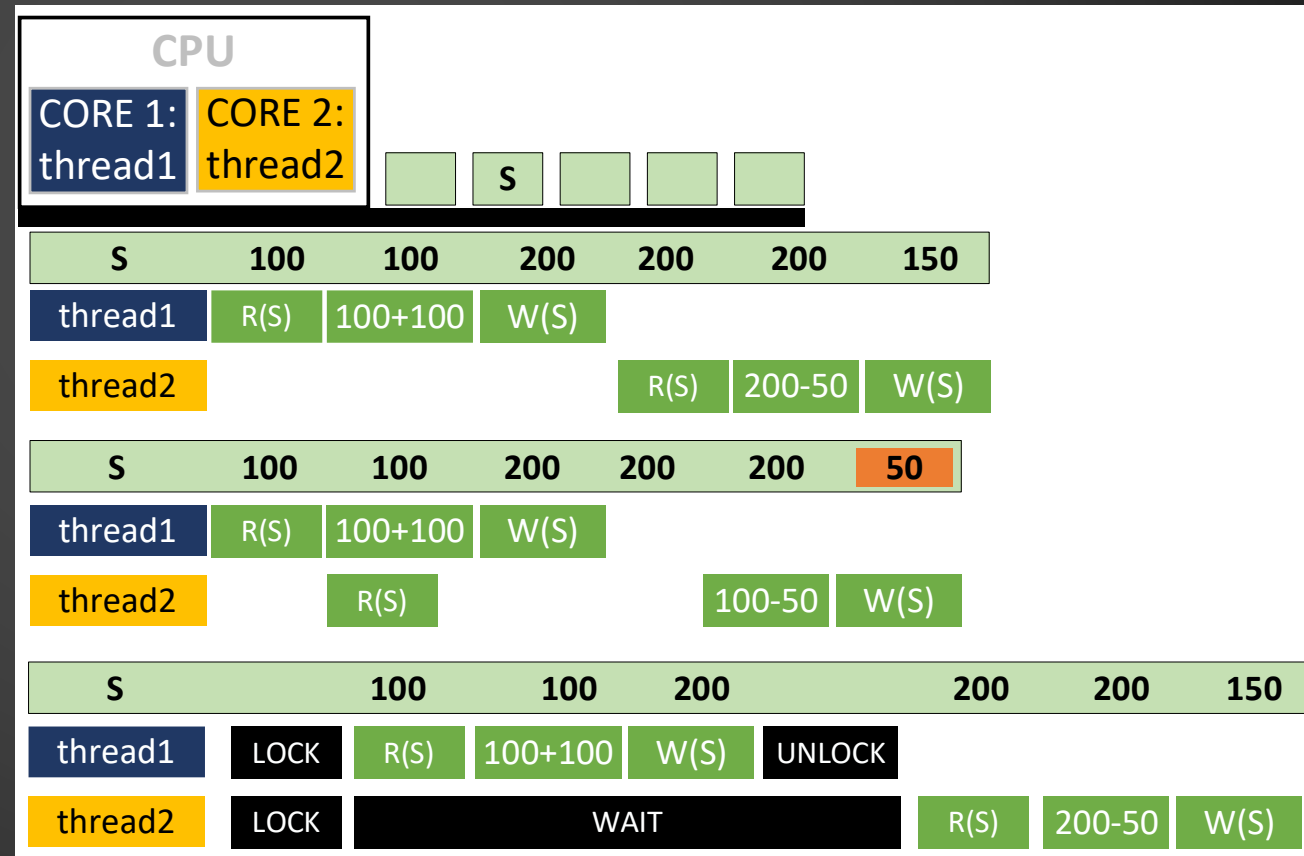
Decision START-Line(14) END-Line(19):

B(6>false	B(7>false	B(9>false	- NOT EXECUTED!
B(6>false	B(7>true	B(8>true	- NOT EXECUTED!
B(6>true	B(7>false	B(8>true	- DONE!



# CODE COVERAGE OF INTERLEAVINGS ???

- Random delays
- Targeted interleavings -> targeted delays



t1:LOCK

t2:LOCK

t1:R(100)

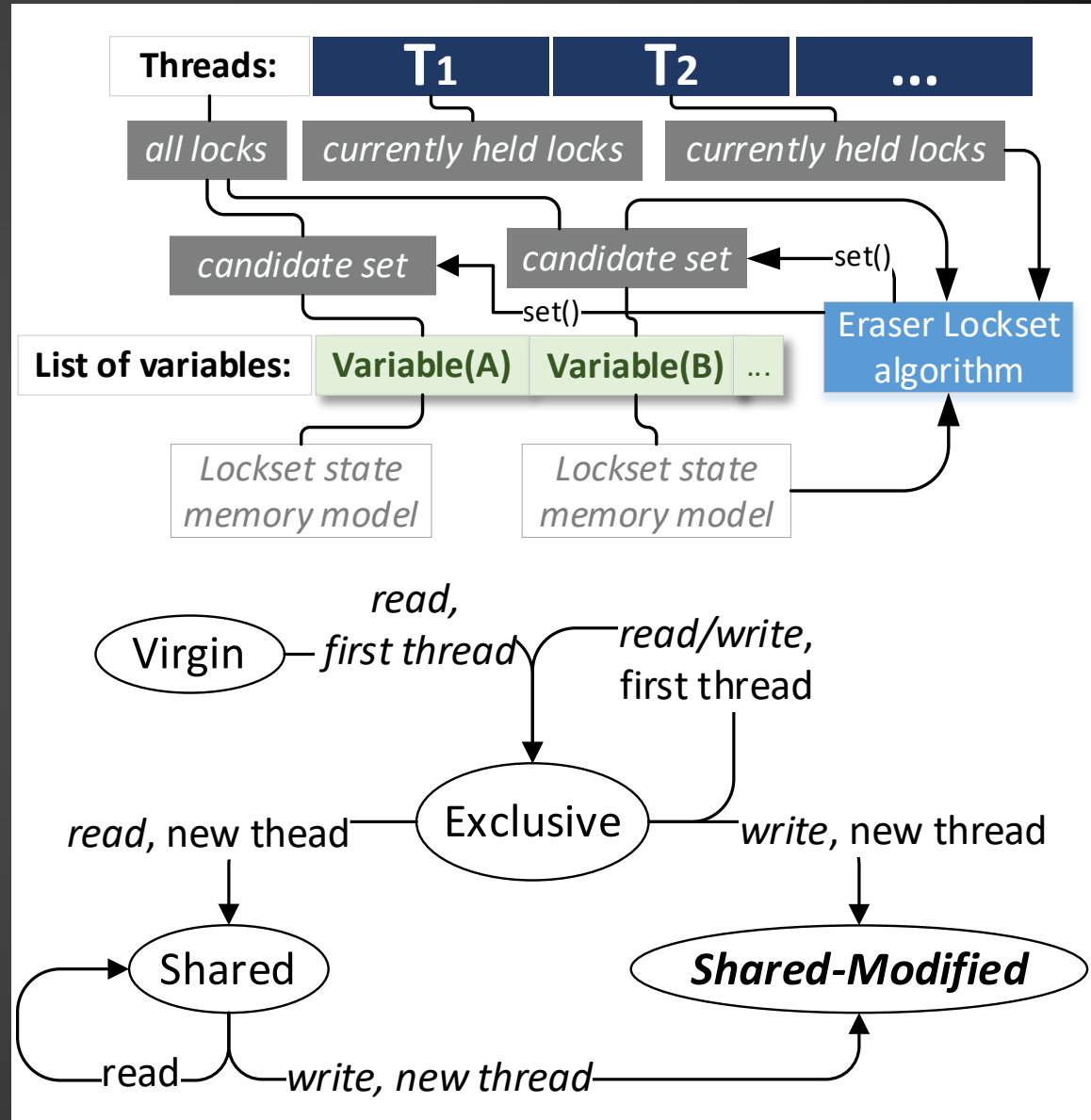
t1:100+100

t1:W(200)

t1:UNLOCK

t2:R(200)

# ERASER LOCKSET ALGORITHM



# ATOMICITY VIOLATION EXAMPLE

```
... for object to mirror...
mirror_mod.mirror_object = ...

operation == "MIRROR_X":
    mirror_mod.use_x = True
    mirror_mod.use_y = False
    mirror_mod.use_z = False
    operation == "MIRROR_Y":
        mirror_mod.use_x = False
        mirror_mod.use_y = True
        mirror_mod.use_z = False
    operation == "MIRROR_Z":
        mirror_mod.use_x = False
        mirror_mod.use_y = False
        mirror_mod.use_z = True
```

• t1:

• ...

• lock();

• `object=new O();`

• unlock()

• ...

• ...

• lock();

• `object.method1();`

• unlock()

• t2:

• ...

• lock();

• `object=NULL;`

• unlock()

• ...

# PROPER SYNCHRONISATION

Shan Lu, Joseph Tucek, Feng Qin, and  
Yuanyuan Zhou. 2006. AVIO: detecting  
atomicity violations via access  
interleaving invariants. *SIGPLAN Not.*  
41, 11 (November 2006), 37–48.  
DOI:[https://doi.org/10.1145/1168918.](https://doi.org/10.1145/1168918.1168864)  
1168864

```
• t1:
• ...
• lock();
• account-=a;
• unlock()
• ...
• lock();
• account-=c;
• unlock()
```

```
• t2:
• ...
• lock();
• account+=b;
• unlock()
• ...
```



# FINDING CONCURRENCY BUGS: LOCKING (LB) AND NON- BLOCKING SYNCHRONIZATION (LF)

*J. Jahić, K. Ali, M. Chatrangoon, and N. Jahani. 2019. (Dis)Advantages of Lock-free Synchronization Mechanisms for Multicore Embedded Systems. International Conference on Parallel Processing: Workshops (ICPP 2019) DOI: <https://doi.org/10.1145/3339186.3339191>*

- Simple code (<https://github.com/KhuraamAli/JAM-Benchmark>)
- MultiProducer/Multi-Consumer pattern (**MPMC**)
- SingleProducer/Multiple-Consumer Ring Buffer (**SPMCR**)
- Sum Counter (**SC**)
- - crash

Applic ation	#thre ads	Helgrind		ThreadSanitizer	
		#reported bugs	#false positives	#reported bugs	#false positives
MPMC_ LB	8	0-1	0-1	1	1
MPMC_ LF	8	-	-	9-11	9-11
SPMCR _LF	4	1156-1230	1156-1230	1	1
SPMCR _LB	4	0	0	0	0
SC_LB	4	-	-	0	0
SC_LF	4	0	0	0	0

# CHALLENGES WITH FINDING CONCURRENCY BUGS

*J. Jahić, V. Kumar, P. O. Antonino and  
G. Wirrer, "Testing the  
Implementation of Concurrent  
AUTOSAR Drivers Against  
Architecture Decisions," 2019 IEEE  
International Conference on  
Software Architecture (ICSA),  
Hamburg, Germany, 2019, pp. 171-  
180, doi: 10.1109/ICSA.2019.00026.*

- Unknown:
  - Shared memory locations
  - Used synchronisation
- Testing: to prove presence of bugs
- Static analysis: to prove absence of bugs
- Find violations of sequential intentions, BUT
  - How to detect/learn the intentions?

# TOOLS FOR FINDING CONCURRENCY BUGS

- Many and few
  - Many prototypes
  - Few available, semi-mature tools
- Helgrind - [www.valgrind.org/](http://www.valgrind.org/)
- ThreadSanitizer - <https://clang.llvm.org/docs/ThreadSanitizer.html>
- ...
- Execution tracing:
  - PIN - <https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html>
  - DynamoRIO - <https://dynamorio.org/>

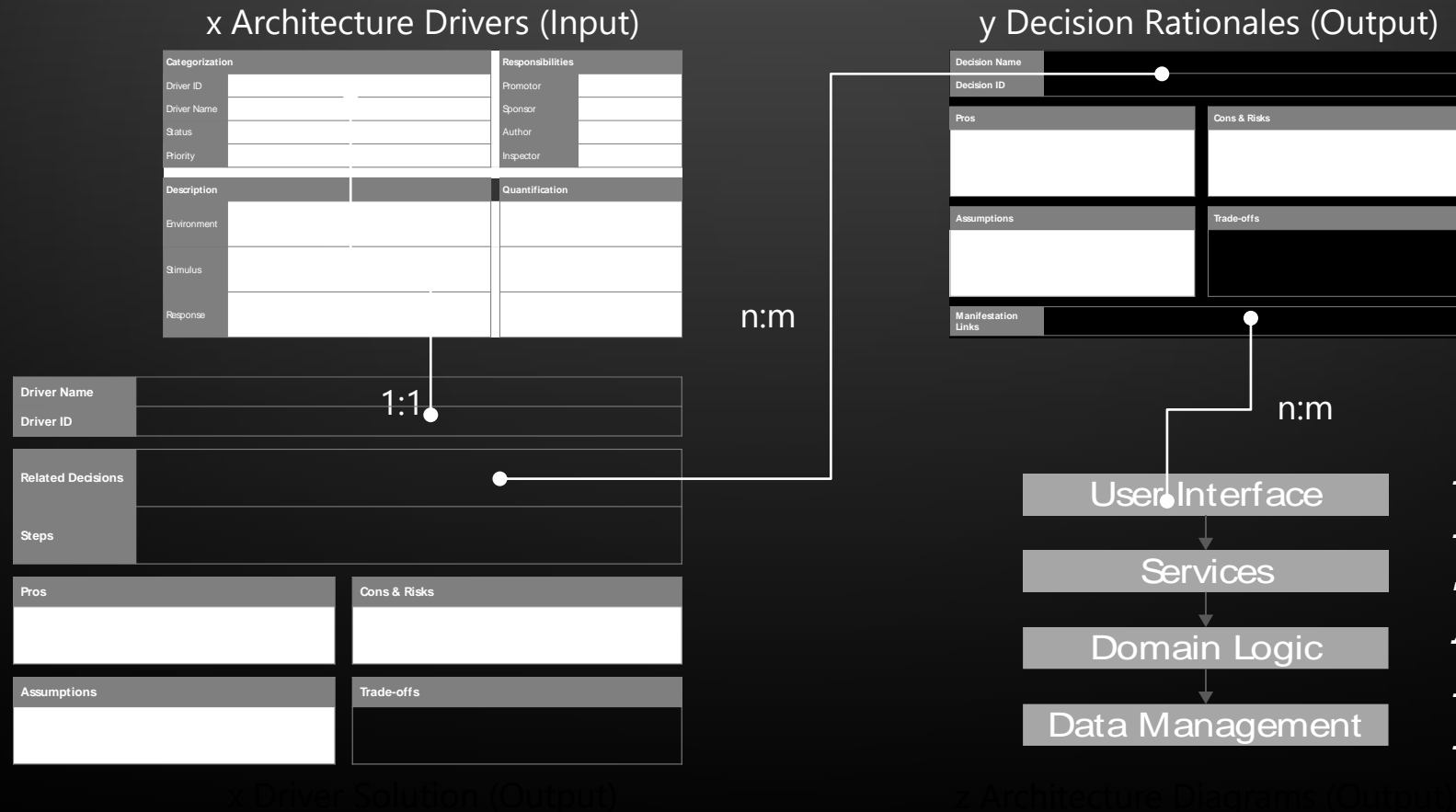
# INDUSTRIAL EXPERIENCE WITH TESTING TOOLS

*Continuous Testing Approach for  
Finding Data Races in Linux-based  
Industrial Embedded Systems,  
Volkan Doganci, 2020; TU  
Kaiserslautern & Siemens*

- Changes that tools introduce to software:
  - Change CMake and Make files
  - Works only with some compilers and their specific flags
  - Limited to 64bit software
  - Sometimes necessary to change source code
  - Almost always intrusive execution
  - No code coverage quantification
  - False positives (user-defined synchronisation)
- Tool chain is usually a design decision in embedded systems - no changes allowed.



# FROM DRIVERS TO SOLUTIONS



*Pragmatic  
Evaluation of  
Software  
Architectures, Jens  
Knodel and Matthias  
Naab, 2016*

# SOLUTION ADEQUACY CHECK

---

Strength, Weakness,  
Opportunities, and  
Threats (SWOT)  
analysis.

Architecture Trade-  
off Analysis Method  
(ATAM).

Rapid Architecture  
Evaluation (RATE)  
method.

# SWOT ANALYSIS

---

	Helpful <i>to achieving the objective</i>	Harmful <i>to achieving the objective</i>
Internal origin <i>attributes of the organisation</i>	S  <i>Strengths</i>	W  <i>Weaknesses</i>
External origin <i>attributes of the environment</i>	O  <i>Opportunities</i>	T  <i>Threats</i>

# ATAM

---

- Presentation of ATAM, business goals, and proposed architecture for addressing business goals;
- Investigation and analysis of system's quality properties, including analysis of trade-offs;
- Testing of the system's quality properties, with test case scenarios, for uncovering additional risks, sensitivity points, and trade-off points;
- Reporting of the findings from the previous steps

*The architecture tradeoff analysis method, Rick Kazman et al., 1998*



# RATE

- 
- Driver Integrity Check (DIC) - reveal unclear architecture drivers, and formulate them systematically using architecture scenarios.
    - Requirements, architecture documentation, stakeholders, and evaluators.
  - Solution Adequacy Check (SAC) - if architectural solutions at hand are adequate for the architecture drivers,
    - Confidence in the adequacy, following the same procedure as DIC.
    - Quantification of architectural decisions

Rationale (Pros, Advantages)	Assumptions & Risks (Constraints)
...	...
Scaling Factors	Trade-offs
...	...

*Pragmatic  
Evaluation of  
Software  
Architectures, Jens  
Knodel and Matthias  
Naab, 2016*

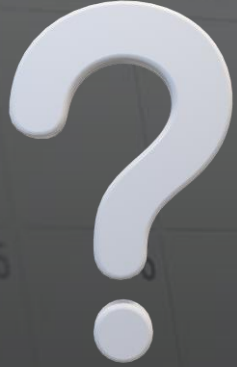
# SUMMARY

- Drivers:
  - Execution time
  - Redundancy (availability, reliability)
  - Power consumption
- Choosing multicores, concurrency, and multithreading to fulfil drivers is a complex decision:
  - Too many implications
  - Too many uncertainties
- Hard to predict the outcome
- Hard to program the design
  - Problem with threads: no interfaces – implicit coupling
- Hard to test it
  - What are assumptions about sequential executions?

- [jj542@cam.ac.uk](mailto:jj542@cam.ac.uk)
- [barner@fortiss.org](mailto:barner@fortiss.org)

# COOPERATION AND FURTHER COMMUNICATION

# QUESTIONS



9:30

Session 1: Fundamental Issues with Concurrency in Embedded Software Systems from Architectural Point of View

10:30

10:45

Session 2: Modelling and DSE Methods for Mixed-Critical Software Systems using Multicore Architectures

11:45

12:00

Session 3: Synchronization in Concurrent Software is an Architectural Decision

13:00