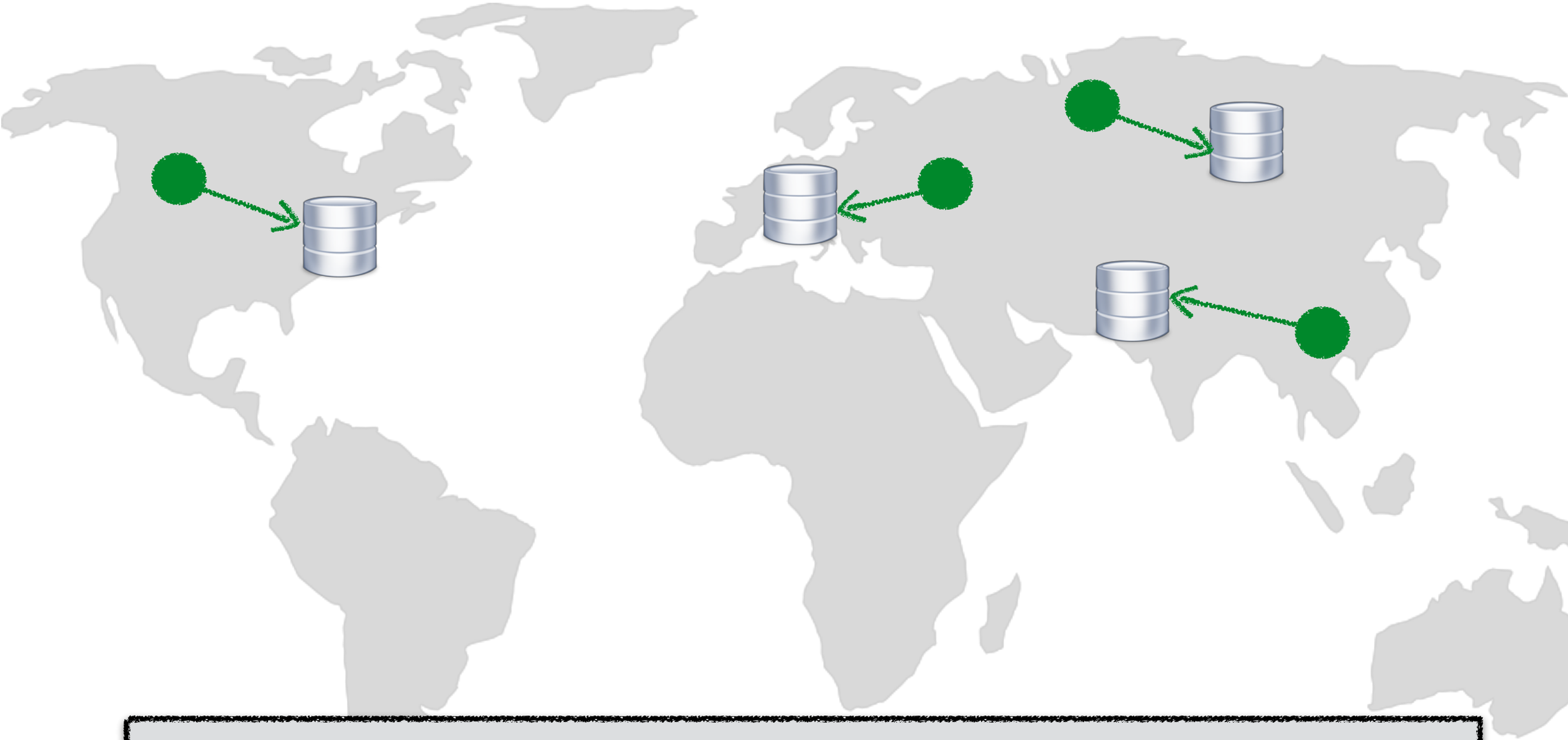
Building correct-by-design highly available cloud applications

Carla Ferreira
Universidade NOVA de Lisboa / NOVA Lincs

Joint work with
Valter Balegas, Nuno Preguiça, Sérgio Duarte (NOVA Lisboa)
Rodrigo Rodrigues (IST),
Marc Shapiro (Sorbonne), Mahsa Najafzadeh (Purdue)
Alexey Gostman (IMDEA), Hongseok Yang (KAIST)

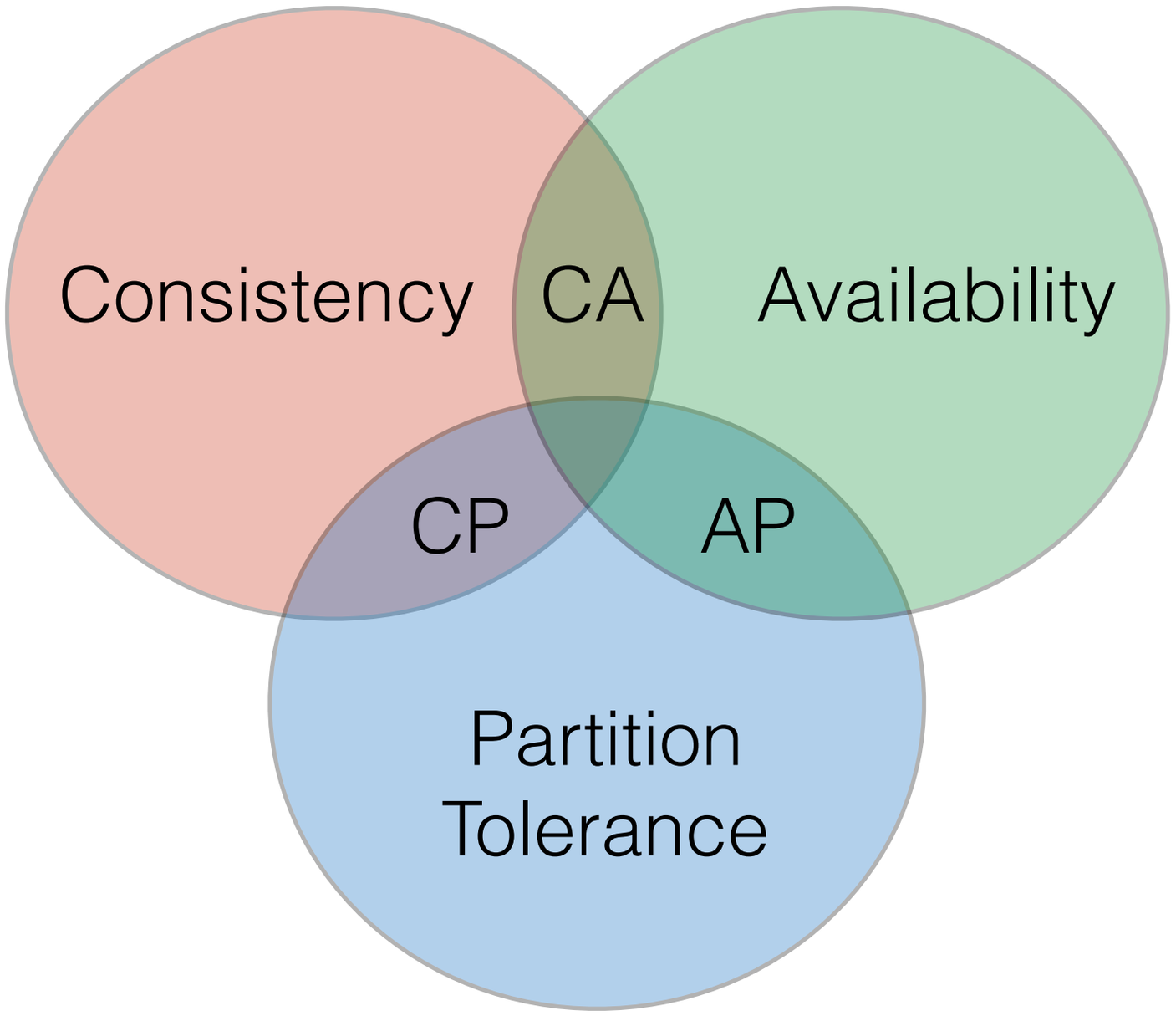
Shonan PL4DS Meeting, 27 - 30 May 2019

Geo-replication



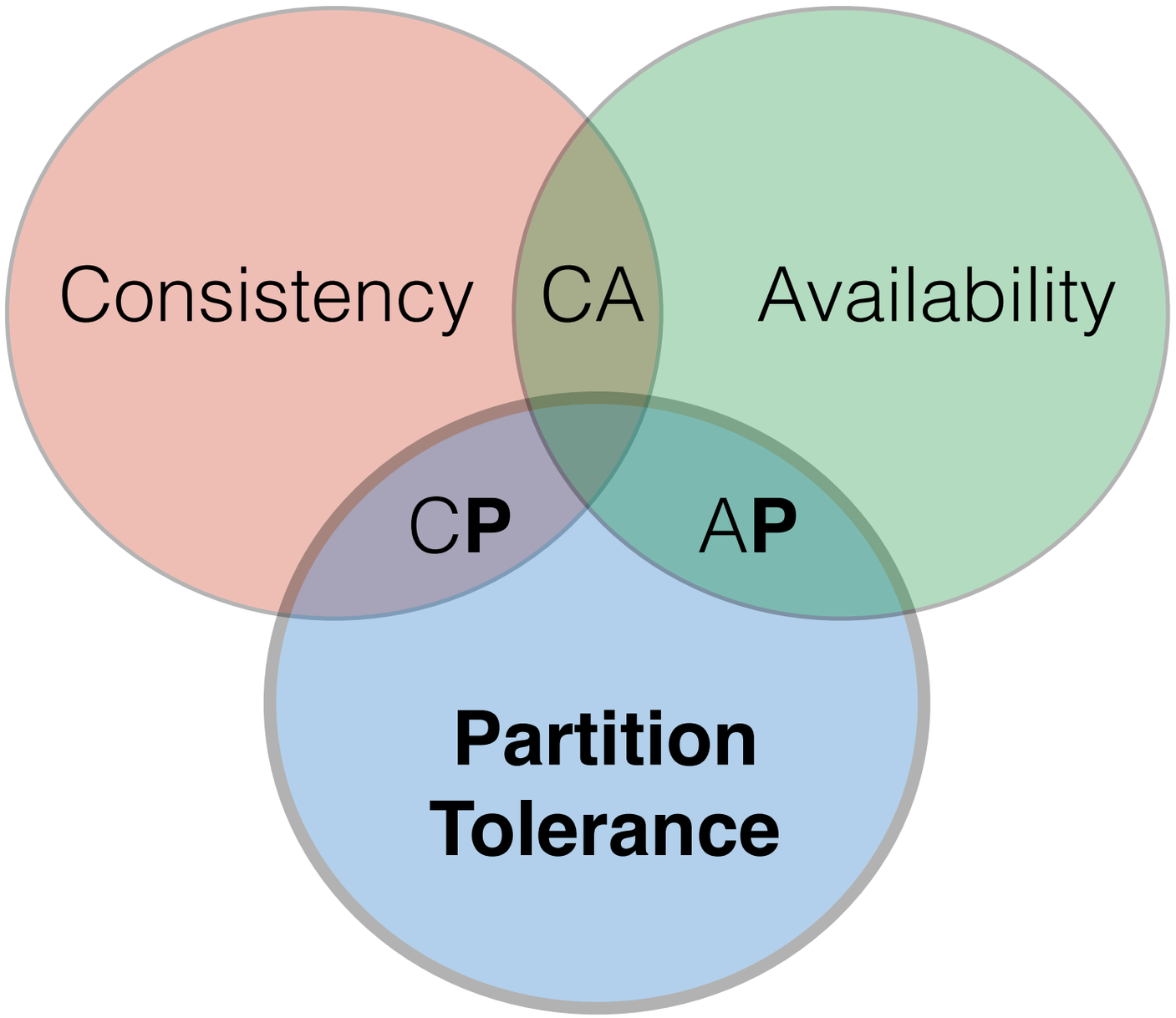
Provides fault tolerance and high availability.
Moving data closer to users improves response time.

CAP theorem



A distributed system can have only two out of the three guarantees [Brewer99]

CAP theorem



In distributed systems network partitions are unavoidable, so partition tolerance is mandatory.

Data consistency

Strong consistency

Eventual consistency

Weak consistency

- Reasoning about applications under weak consistency is fraught with technical subtleties
- Difficult to find the right balance between correctness, performance, and availability
- Requires a high-level of expertise from programmers

Weak consistency

- Reasoning about applications under weak consistency is fraught with technical subtleties
- Difficult to find the right balance between correctness, performance, and availability
    - Requires a high-level of expertise from programmers

- **Goal:** To provide tools and techniques to build correct-by-design distributed programs from correct sequential programs

CRDTs: distributed set

- Sequential specification
  - {true} add(e) { e ∈ S }
  - {true} rmv(e) { e ∉ S }
- Commutative (e ≠ f):
  - {true} add(e) || add(e) { e ∈ S }
  - {true} rmv(e) || rmv(e) { e ∉ S }
  - {true} add(e) || add(f) { e,f ∈ S }
  - {true} rmv(e) || rmv(f) { e, f ∉ S }
  - {true} add(e) || rmv(f) { e ∈ S, f ∉ S }
- Deterministic conflict resolution policy:
  - {true} add(e) || rmv(e) { e ∈ S } **Add-wins**

CRDTs: distributed set

- Sequential specification
  - {true} add(e) { e ∈ S }
  - {true} rmv(e) { e ∉ S }
- Commutative (e ≠ f):
  - {true} add(e) || add(e) { e ∈ S }
  - {true} rmv(e) || rmv(e) { e ∉ S }
  - {true} add(e) || add(f) { e,f ∈ S }
  - {true} rmv(e) || rmv(f) { e, f ∉ S }
  - {true} add(e) || rmv(f) { e ∈ S, f ∉ S }
- Deterministic conflict resolution policy:
  - {true} add(e) || rmv(e) { e ∉ S } **Remove-wins**

CRDTs in the wild

Convergence is not enough to ensure application invariants

https://pages.lip6.fr/syncfree

e-games platform

player
alice
bob
eve

Set CRDT

enrolled
(alice,apex)
(bob,battle)
(alice,battle)

Set CRDT

tournament
apex
battle

Set CRDT

Invariant: A player may only enroll in an existing tournament (referential integrity)

Concurrent updates

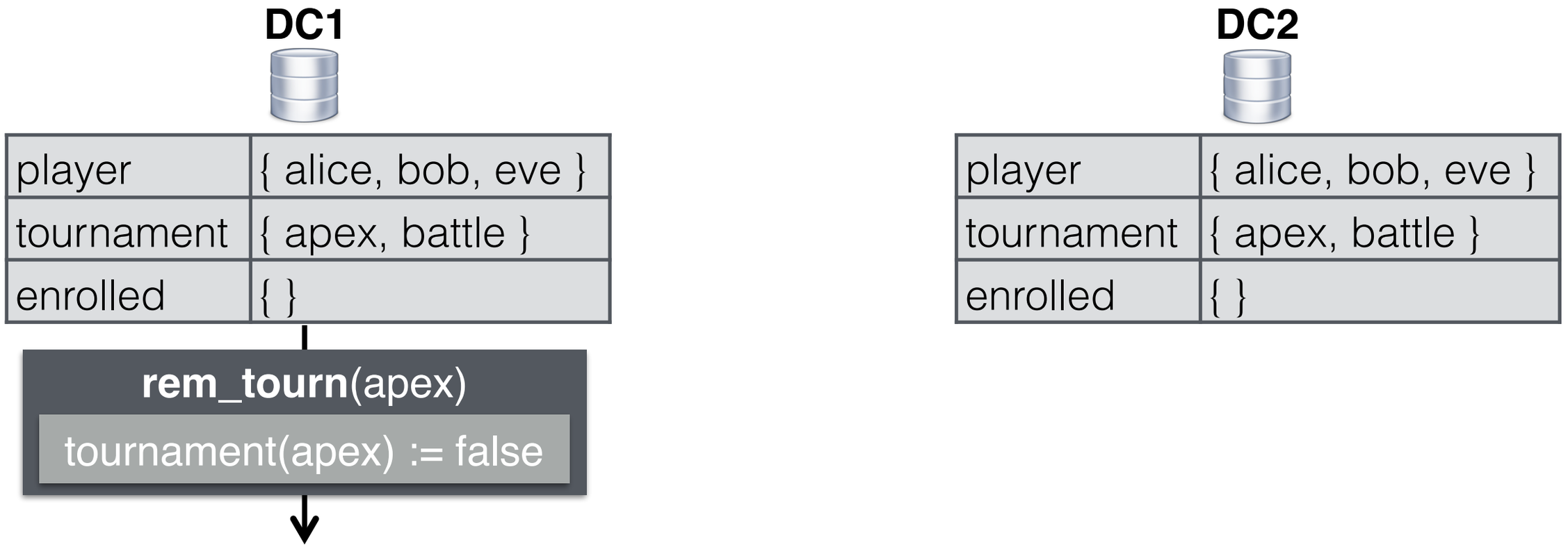
DC1

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ }

DC2

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ }

Concurrent updates



Concurrent updates

DC1

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ }

rem_tourn(apex)

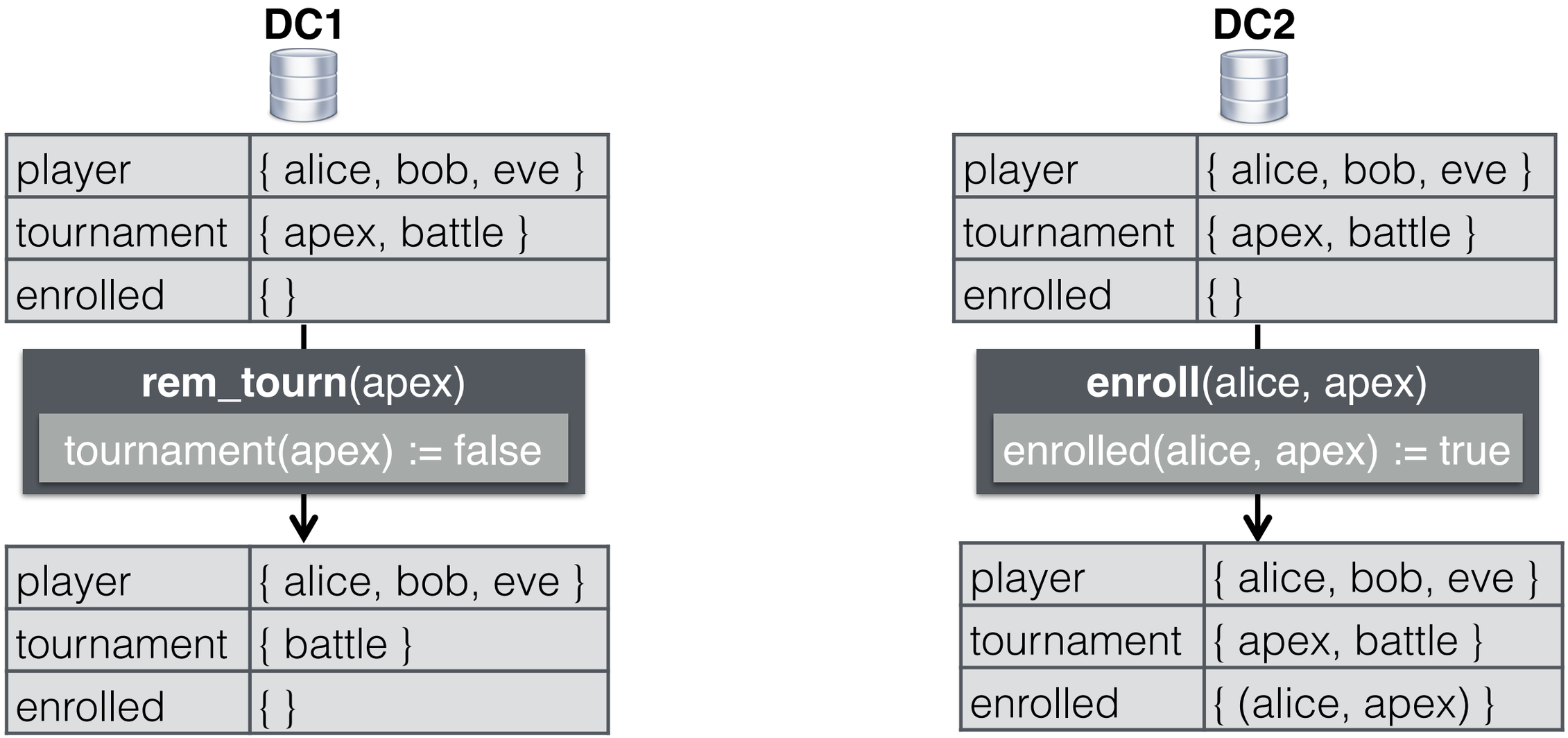
tournament(apex) := false

player	{ alice, bob, eve }
tournament	{ battle }
enrolled	{ }

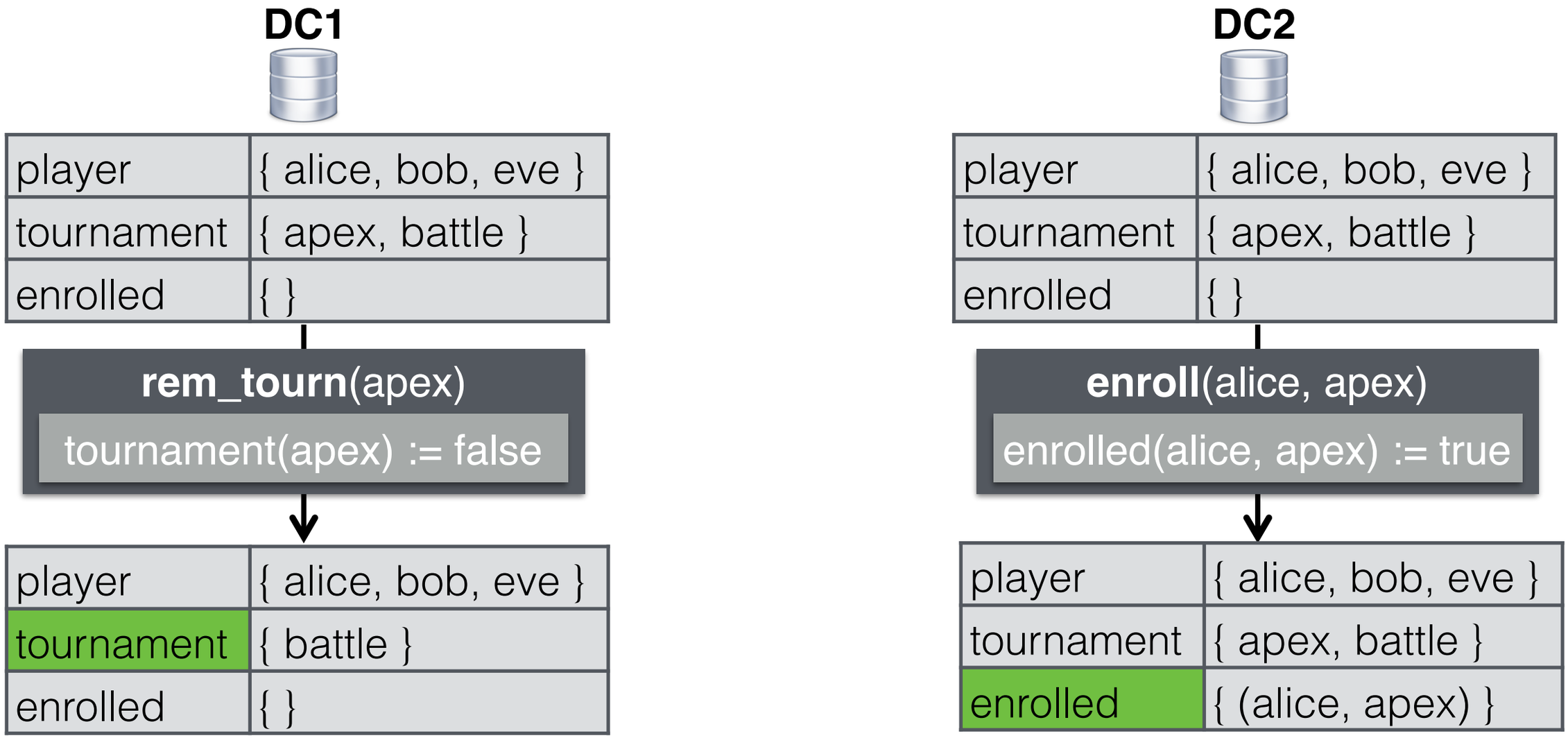
DC2

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ }

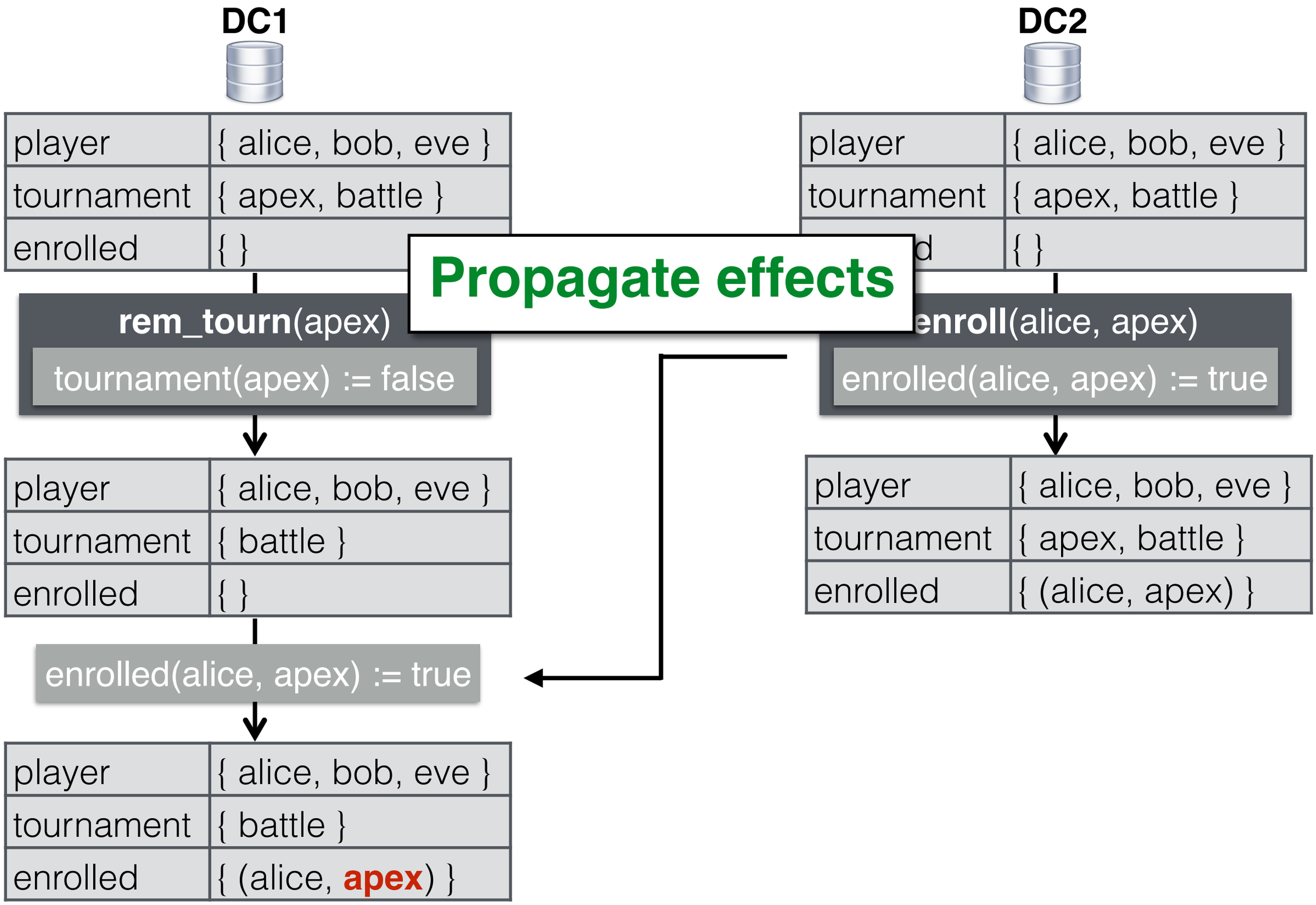
Concurrent updates



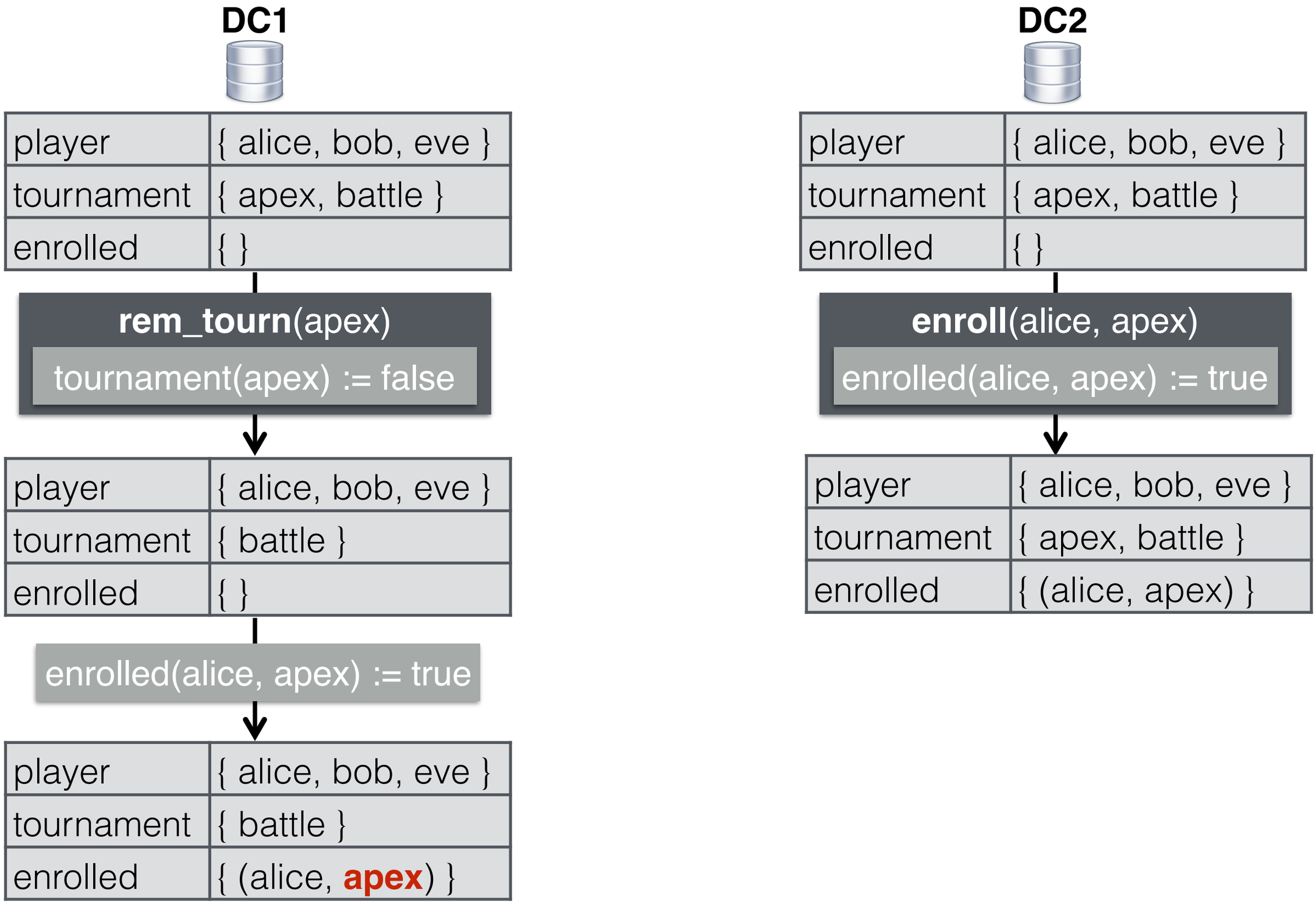
Concurrent updates



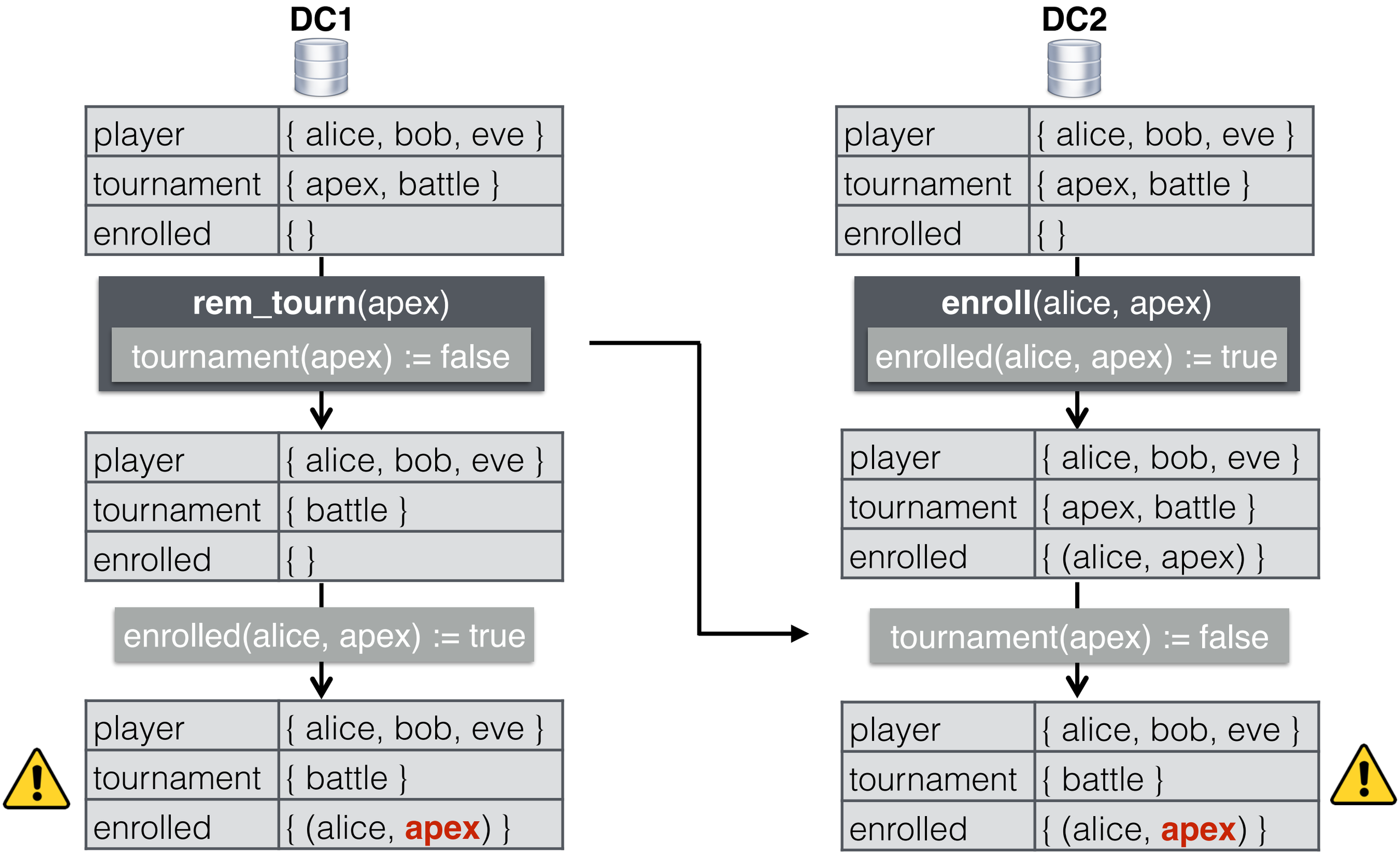
Concurrent updates



Concurrent updates



Concurrent updates



State converges,
but the invariant is broken

DC1

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ }

rem_tourn(apex)

tournament(apex) := false

player	{ alice, bob, eve }
tournament	{ battle }
enrolled	{ }

enrolled(alice, apex) := true

player	{ alice, bob, eve }
tournament	{ battle }
enrolled	{ (alice, **apex**) }

DC2

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ }

enroll(alice, apex)

enrolled(alice, apex) := true

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ (alice, apex) }

tournament(apex) := false

player	{ alice, bob, eve }
tournament	{ battle }
enrolled	{ (alice, **apex**) }

Invariants are global properties
replicas only have local information

DC1

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ }

rem_tourn(apex)

tournament(apex) := false

player	{ alice, bob, eve }
tournament	{ battle }
enrolled	{ }

enrolled(alice, apex) := true

player	{ alice, bob, eve }
tournament	{ battle }
enrolled	{ (alice, **apex**) }

DC2

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ }

enroll(alice, apex)

enrolled(alice, apex) := true

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ (alice, apex) }

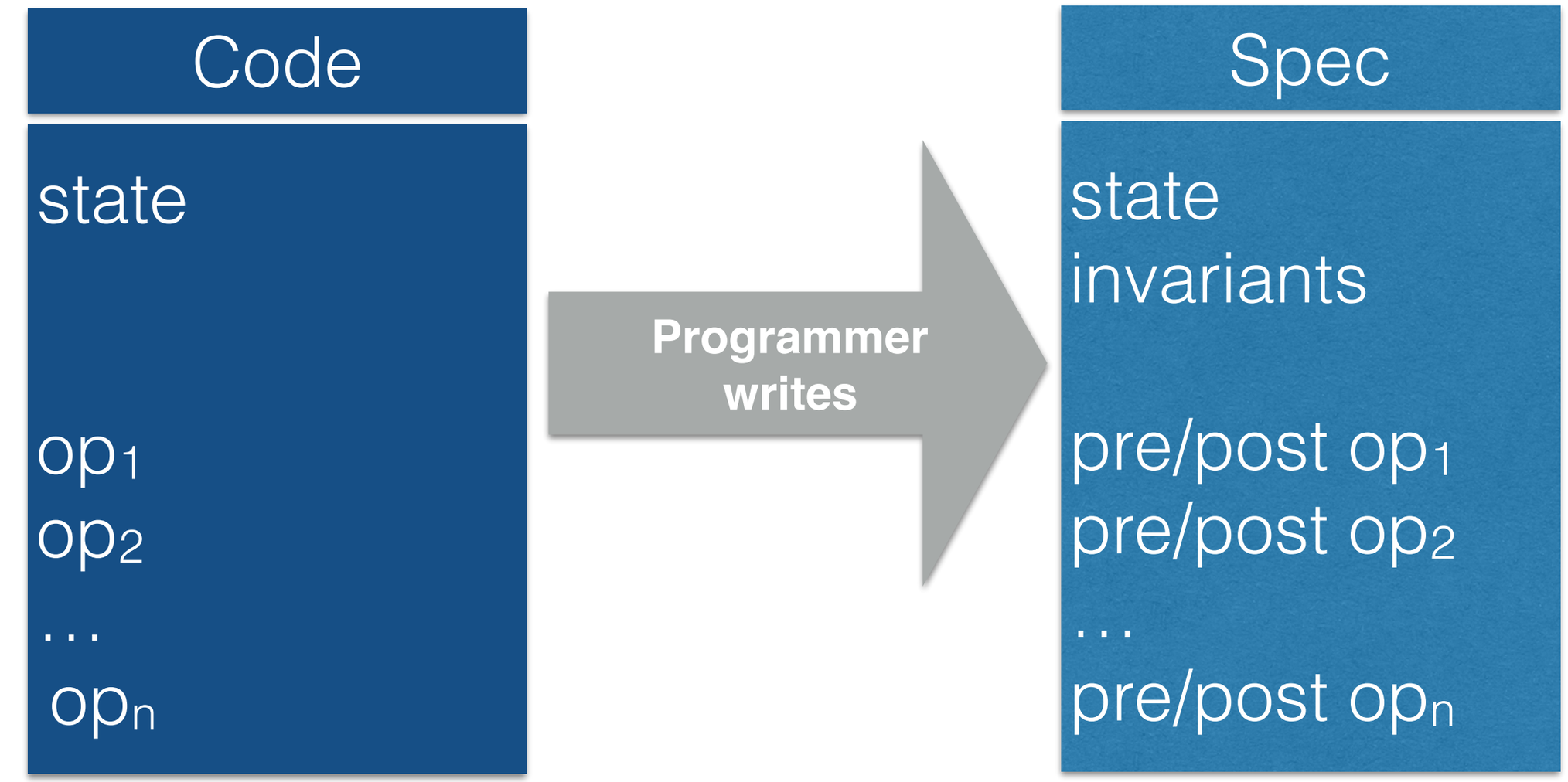
tournament(apex) := false

player	{ alice, bob, eve }
tournament	{ battle }
enrolled	{ (alice, **apex**) }

Techniques explored

- Invariant violation avoidance [EuroSys'15, POPL'16]
  - Prevent conflicting executions
  - Minimize synchronization
- Invariant preservation [VLDB'19]
  - Transform operations
  - Repair invariant when conflicting operations execute concurrently
  - Avoid synchronization altogether

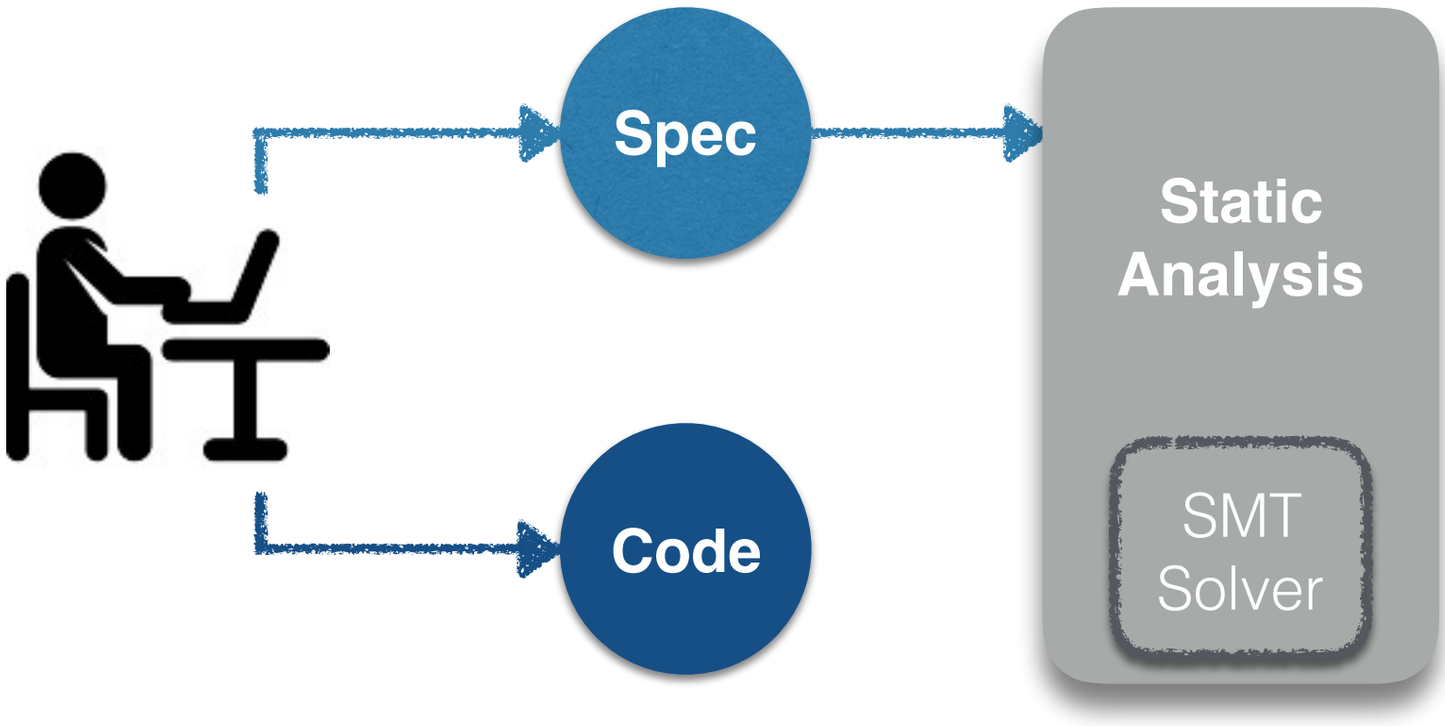
Methodology



Java, C, Ruby, …

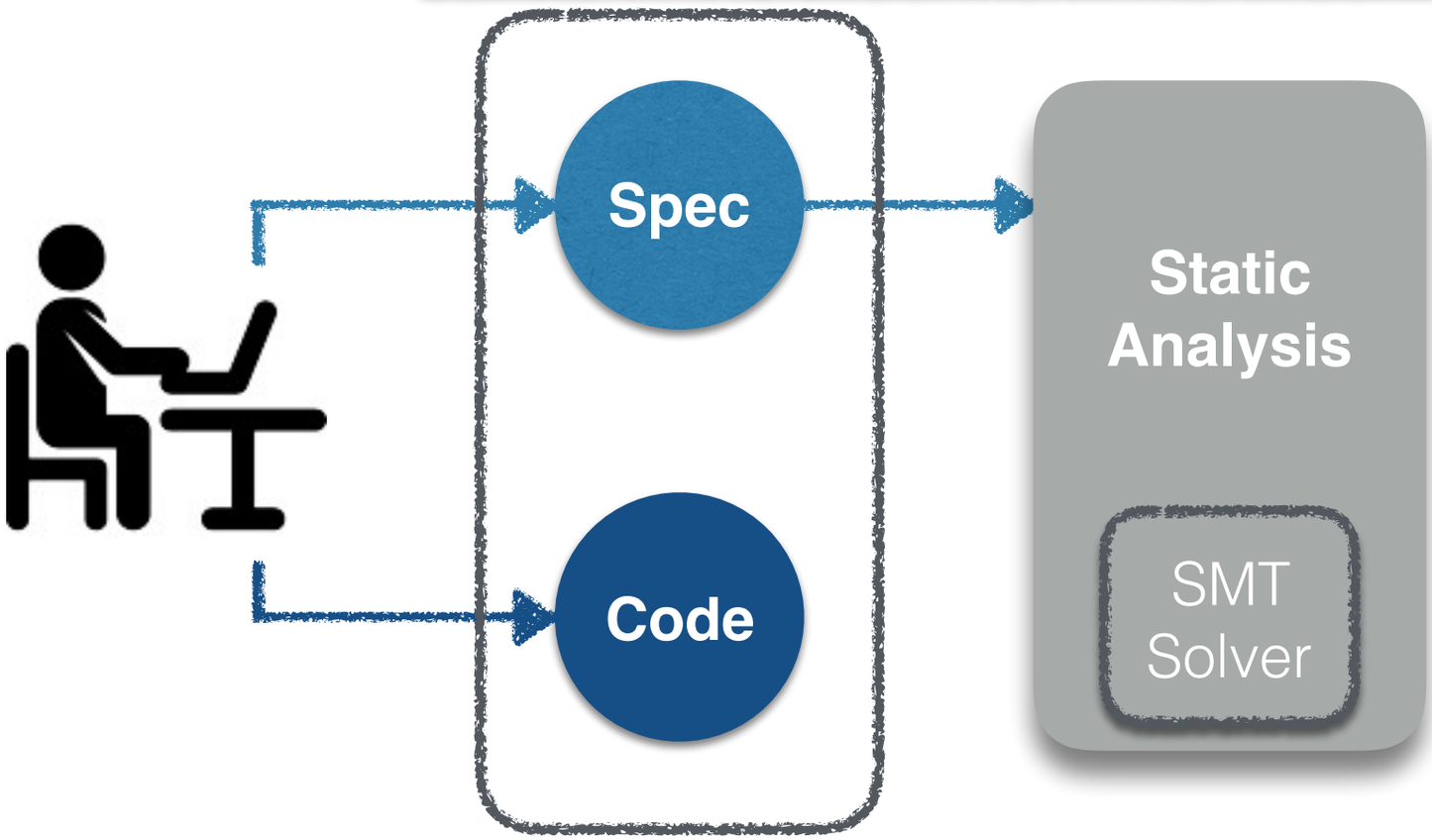
Boogie, Why3, …

Developer should focus on the application invariants



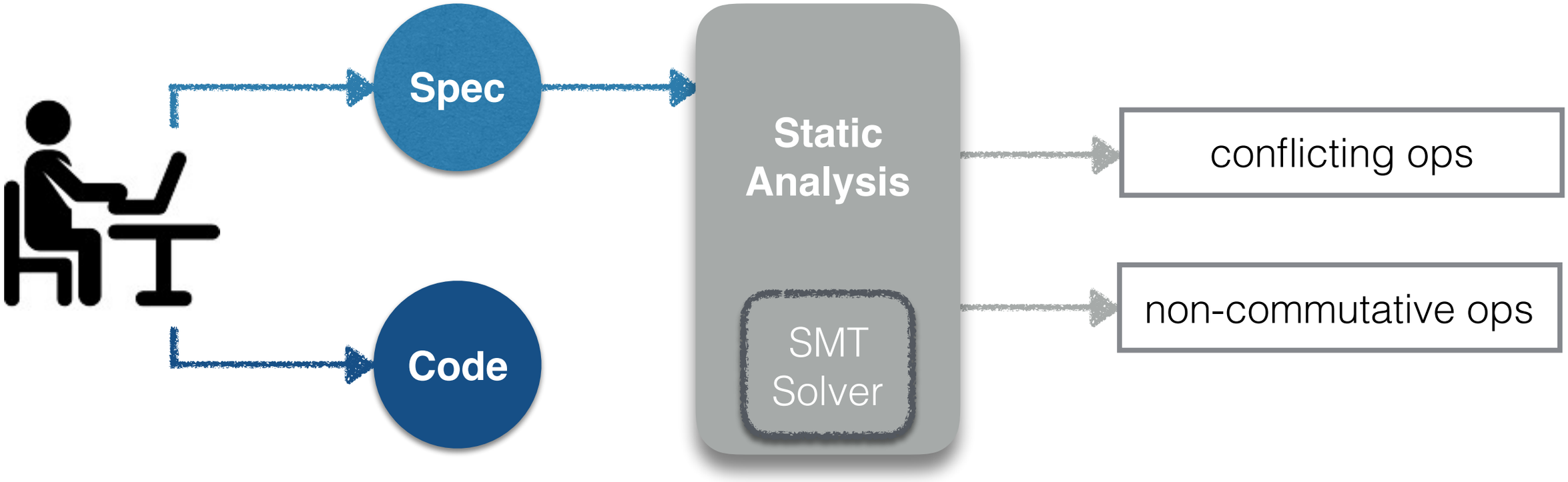
Developer should focus on the application invariants

Ignore concurrency problems raised by geo-replication

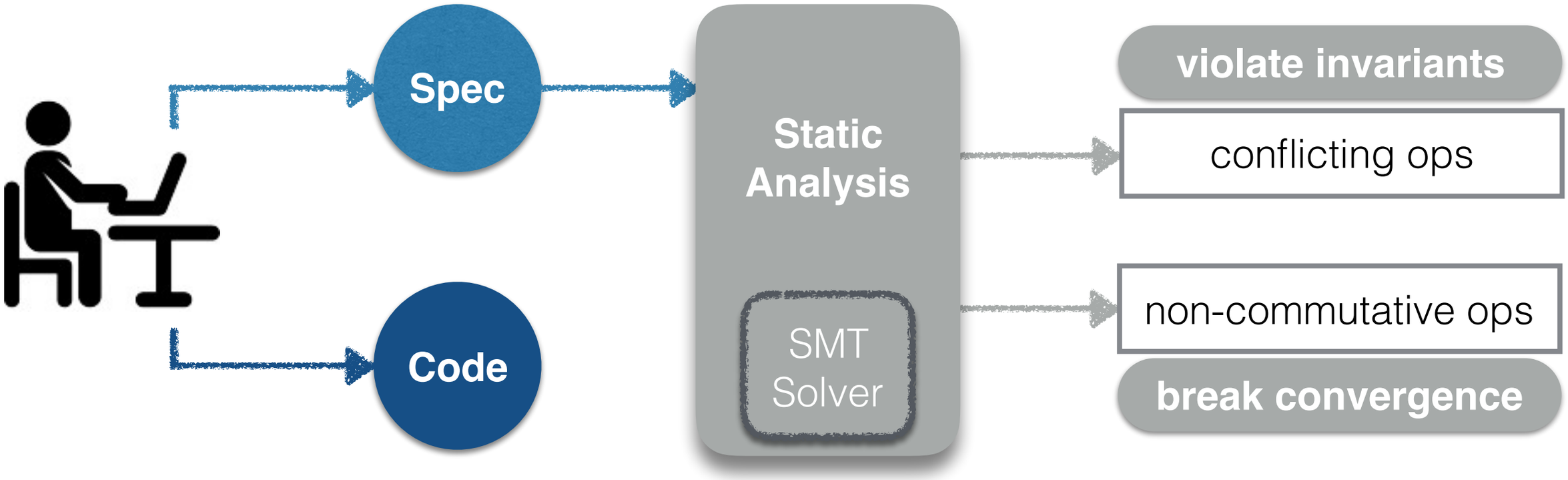


sequential code

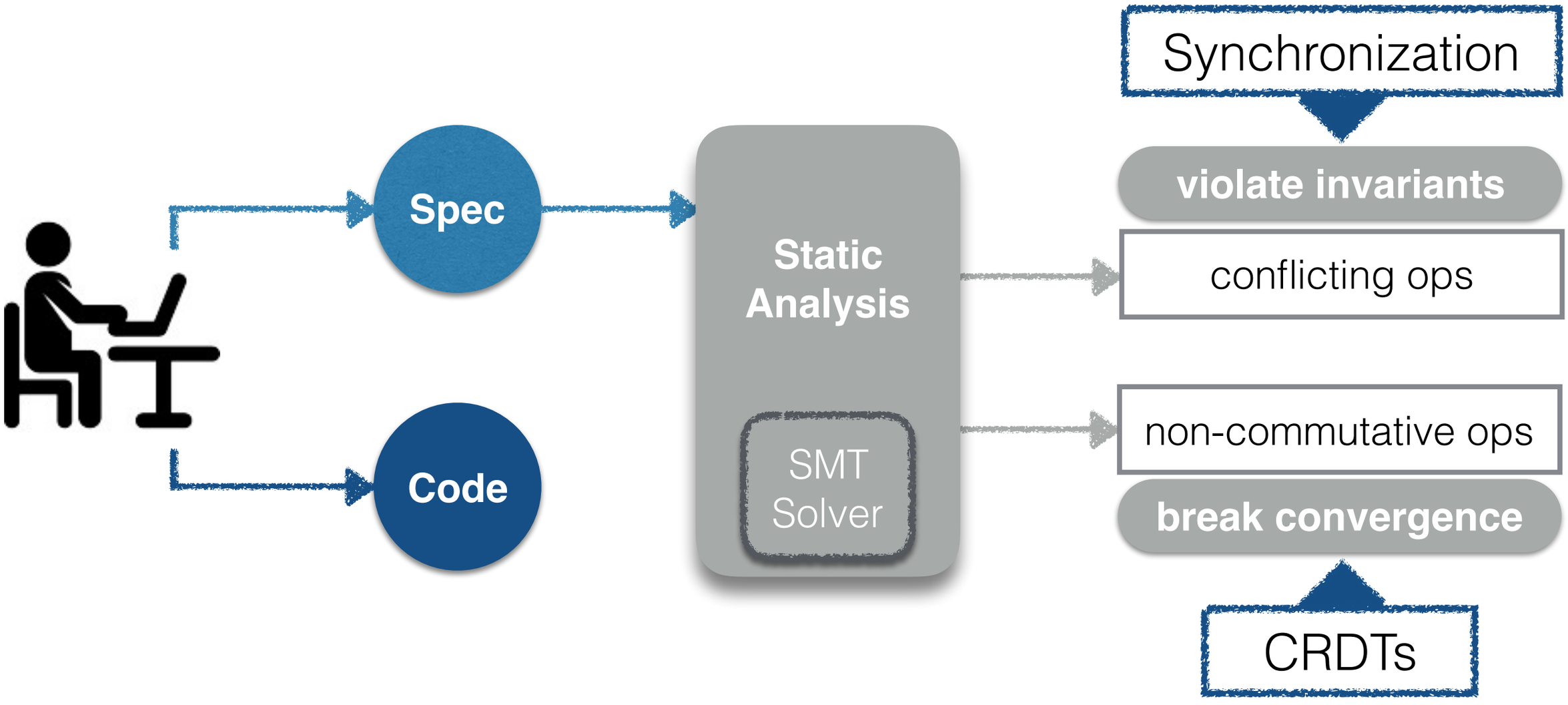
Detect problematic concurrent operations



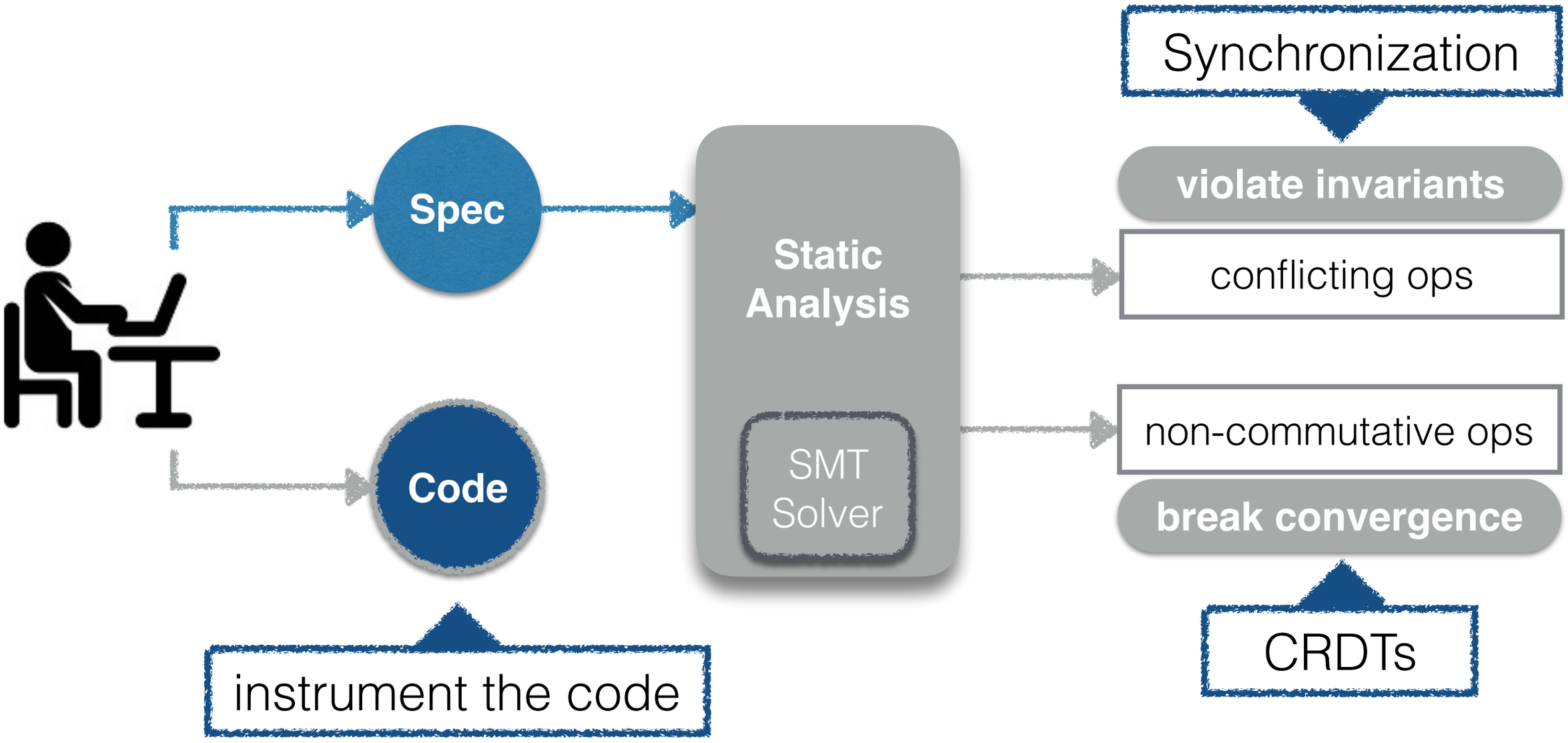
Detect problematic concurrent operations



Select mechanisms to make problematic operations safe

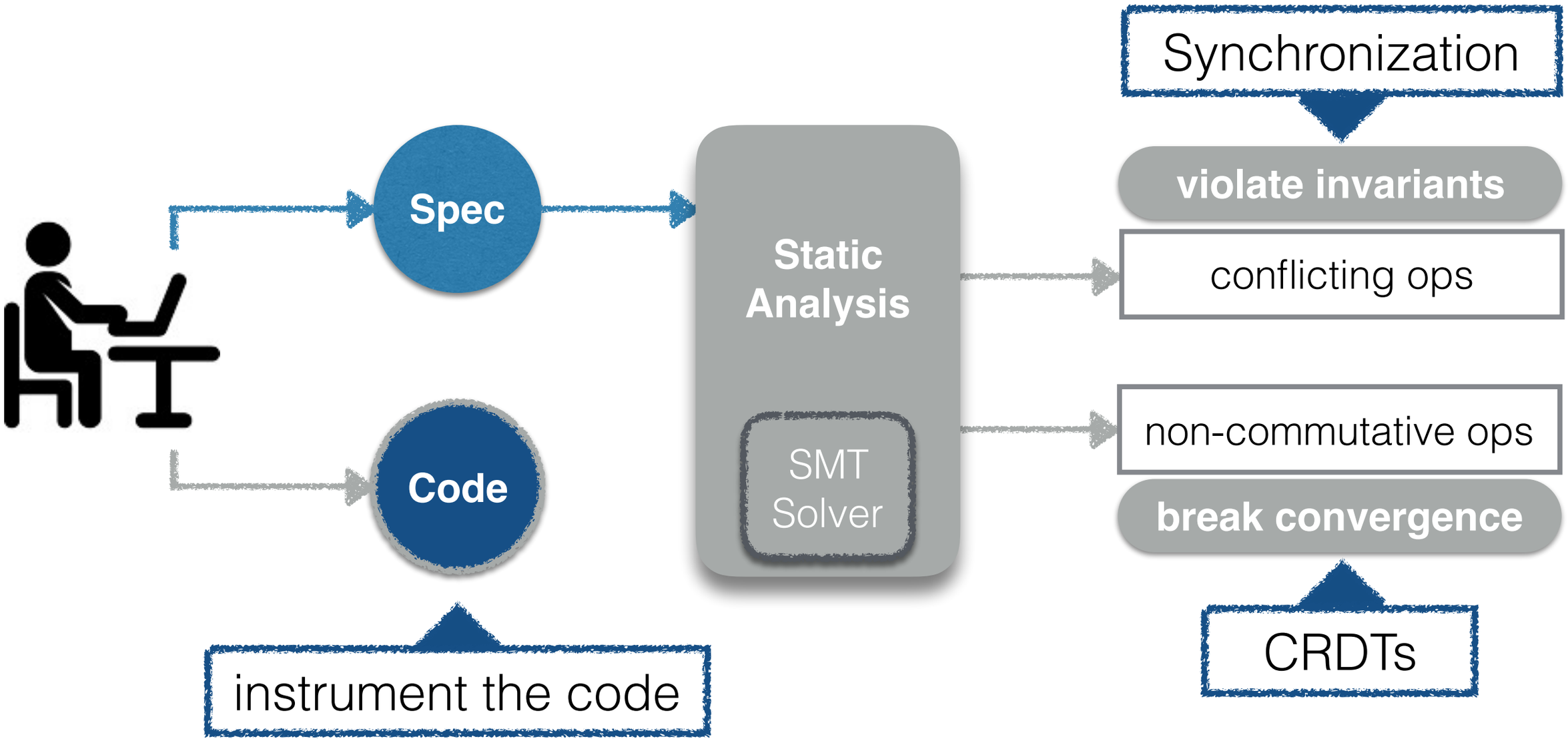


Select mechanisms to make problematic operations safe

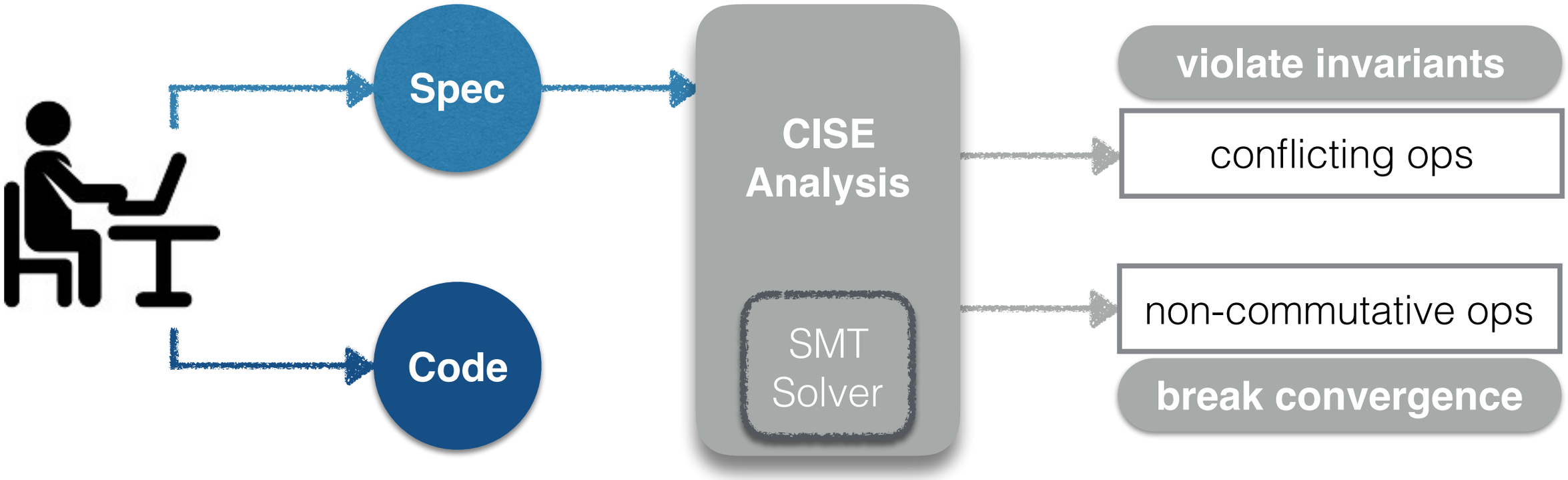


Select mechanisms

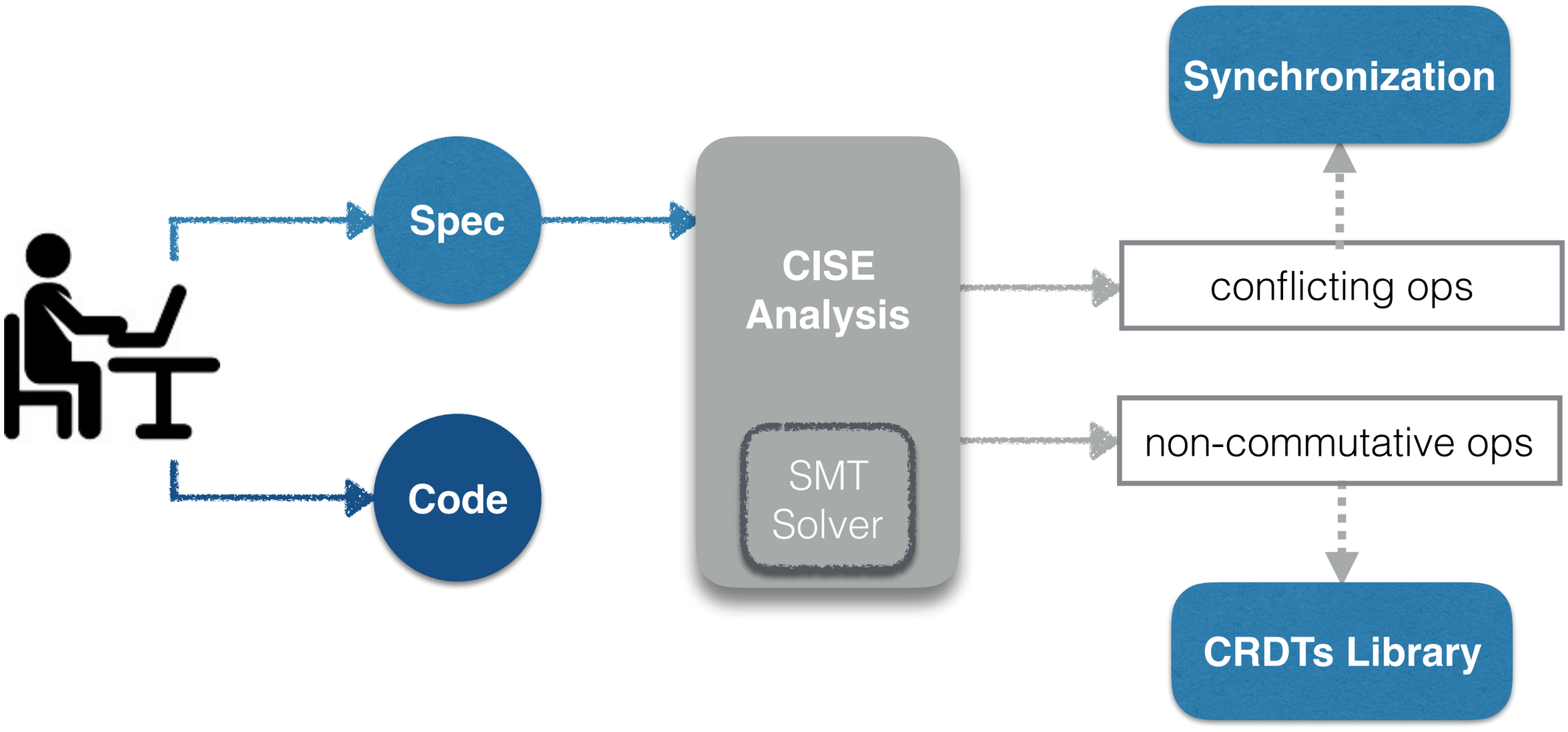
No insurance that the developer made the right choices



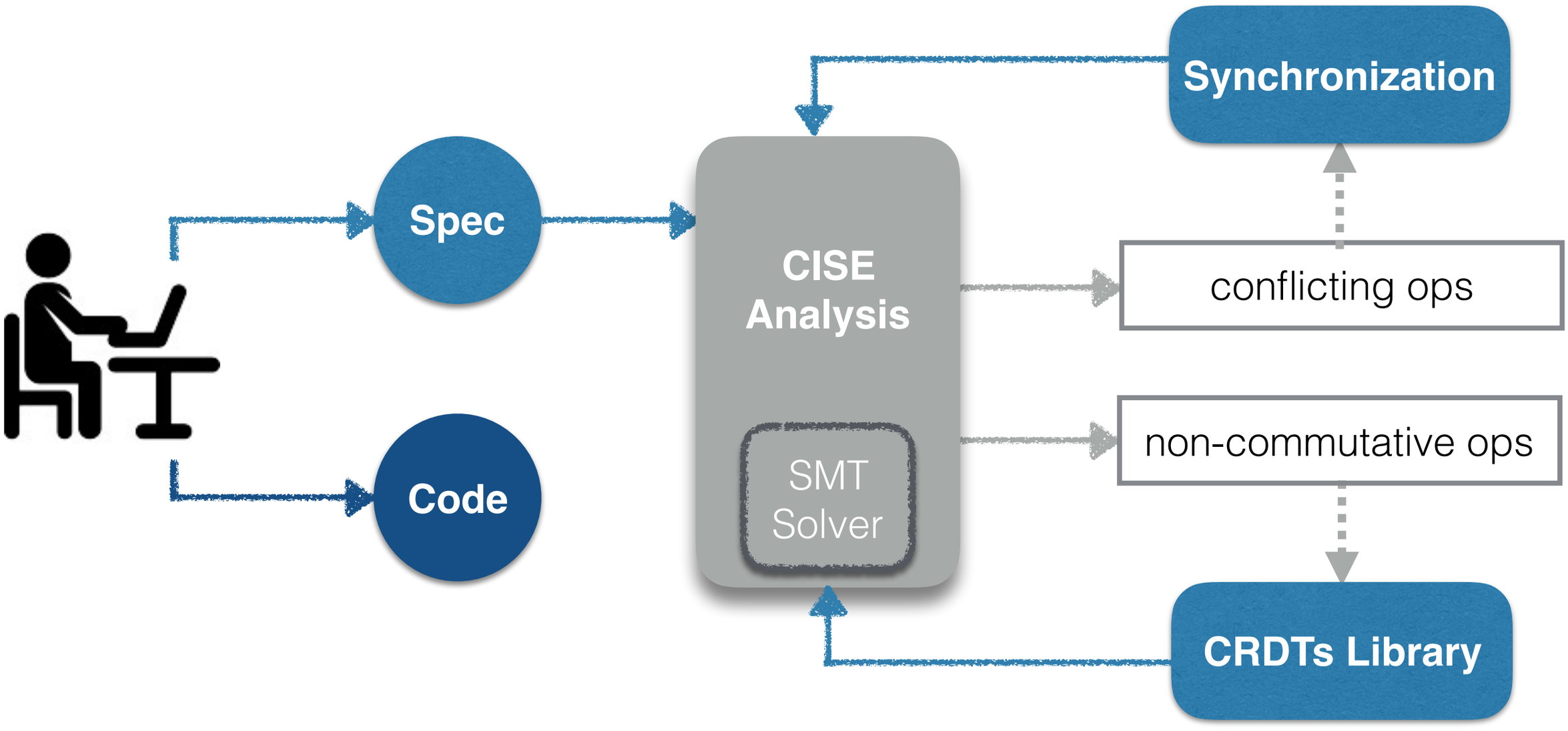
CISE: check the safety of synchronization choices



CISE: check the safety of synchronization choices

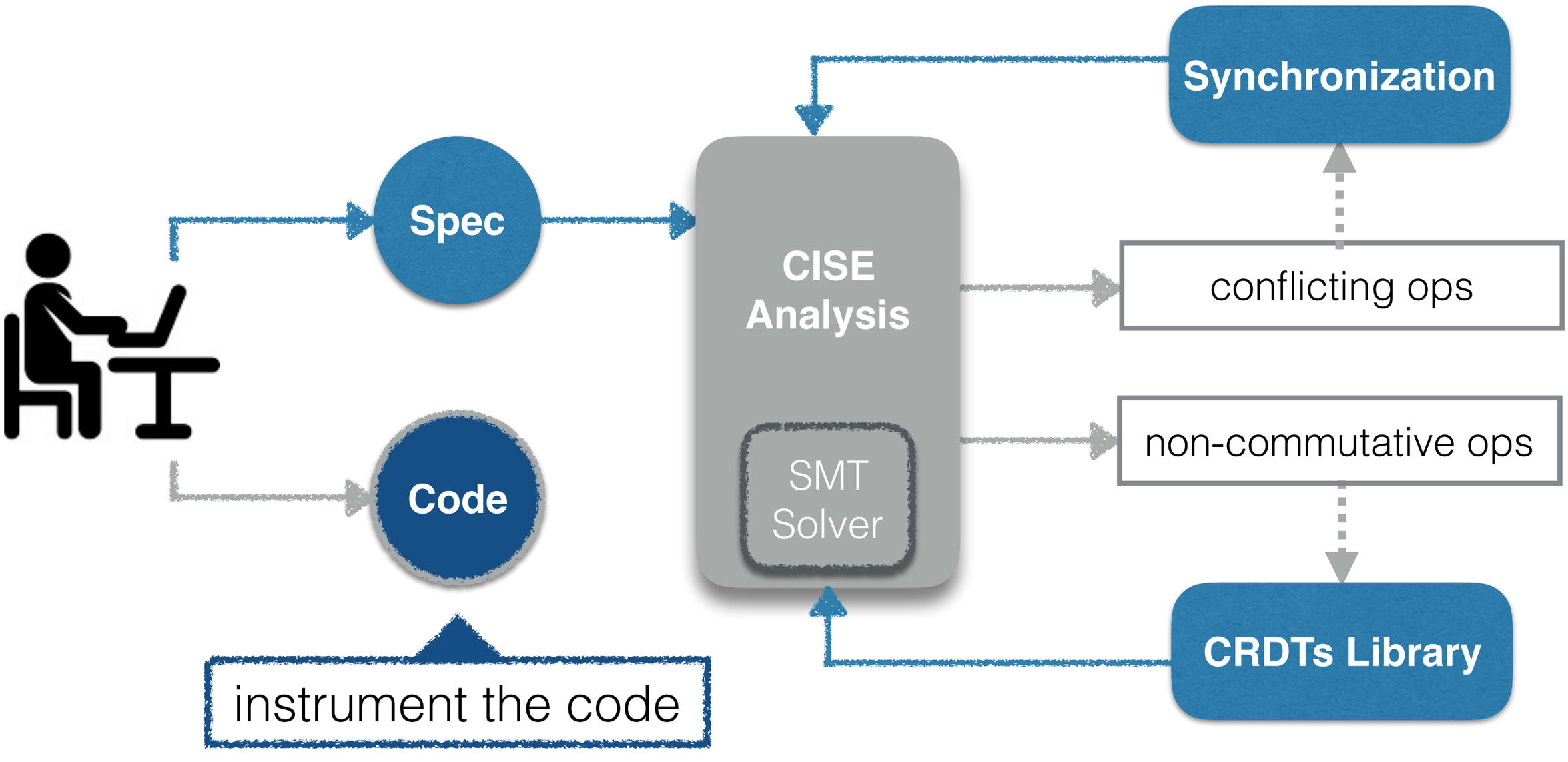


CISE: check the safety of synchronization choices

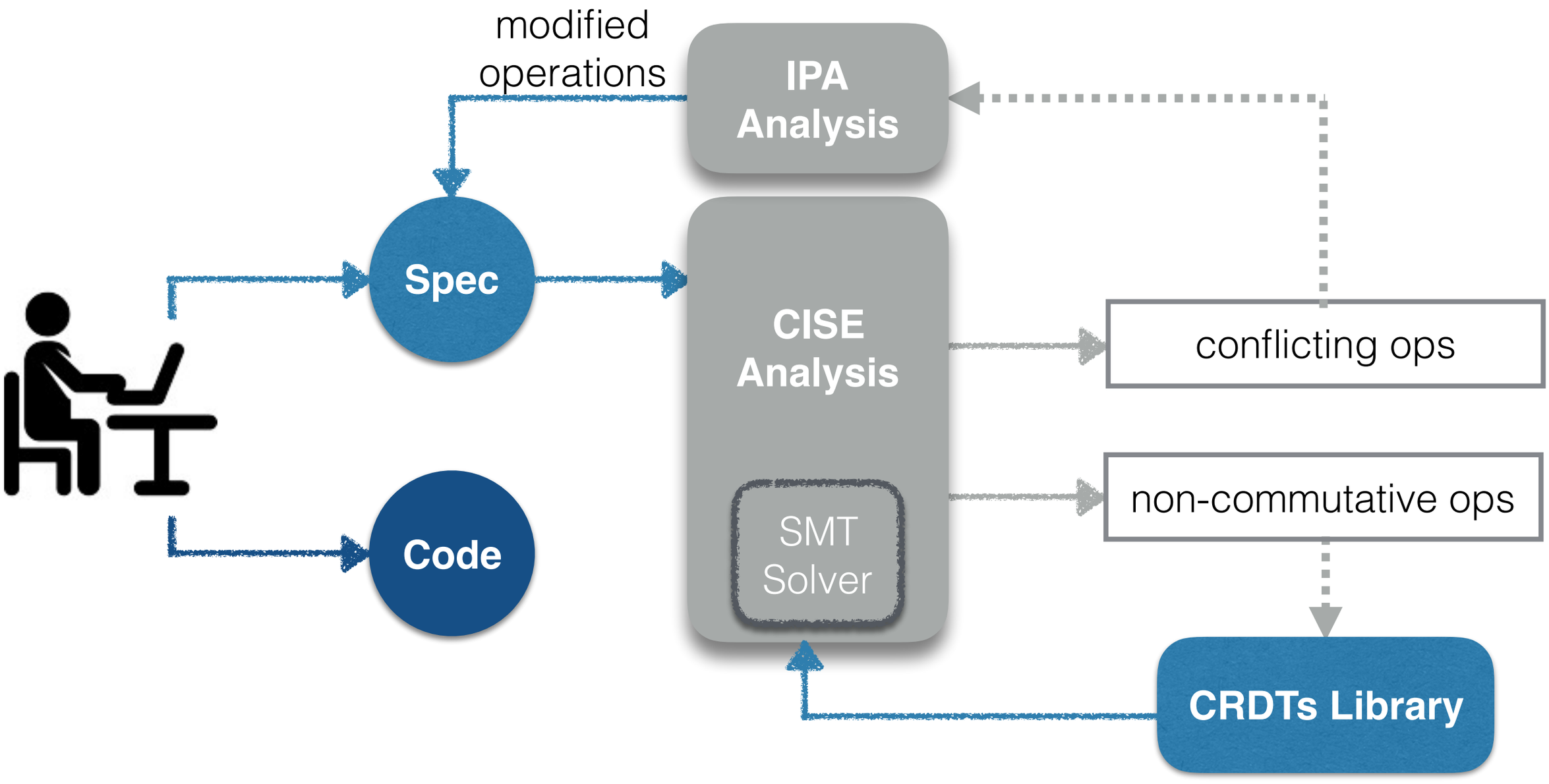


Correct-by-design* distributed application

*if the code satisfies the specification



Repairing invariants to avoid synchronization



DC1

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ }

rem_tourn(apex)

tournament(apex) := false

player	{ alice, bob, eve }
tournament	{ battle }
enrolled	{ }

DC2

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ }

enroll(alice, apex)

enrolled(alice, apex) := true

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ (alice, apex) }

Instead of repairing after the invariant violation preemptively add a repair to the operation

DC1

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ }

rem_tourn(apex)

tournament(apex) := false

player	{ alice, bob, eve }
tournament	{ battle }
enrolled	{ }

DC2

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ }

enroll(alice, apex)

enrolled(alice, apex) := true

tournament(apex) := true

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ (alice, apex) }

Instead of repairing after the invariant violation preemptively add a repair to the operation

DC1

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ }

rem_tourn(apex)

tournament(apex) := false

player	{ alice, bob, eve }
tournament	{ battle }
enrolled	{ }

DC2

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ }

enroll(alice, apex)

enrolled(alice, apex) := true

tournament(apex) := true

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ (alice, apex) }

Instead of repairing after the invariant violation preemptively add a repair to the operation

Repair should have no observable effect, if no conflict occurs

DC1

player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ }

rem_tourn(apex)

tournament(apex) := false

player	{ alice, bob, eve }
tournament	{ battle }
enrolled	{ }

DC2

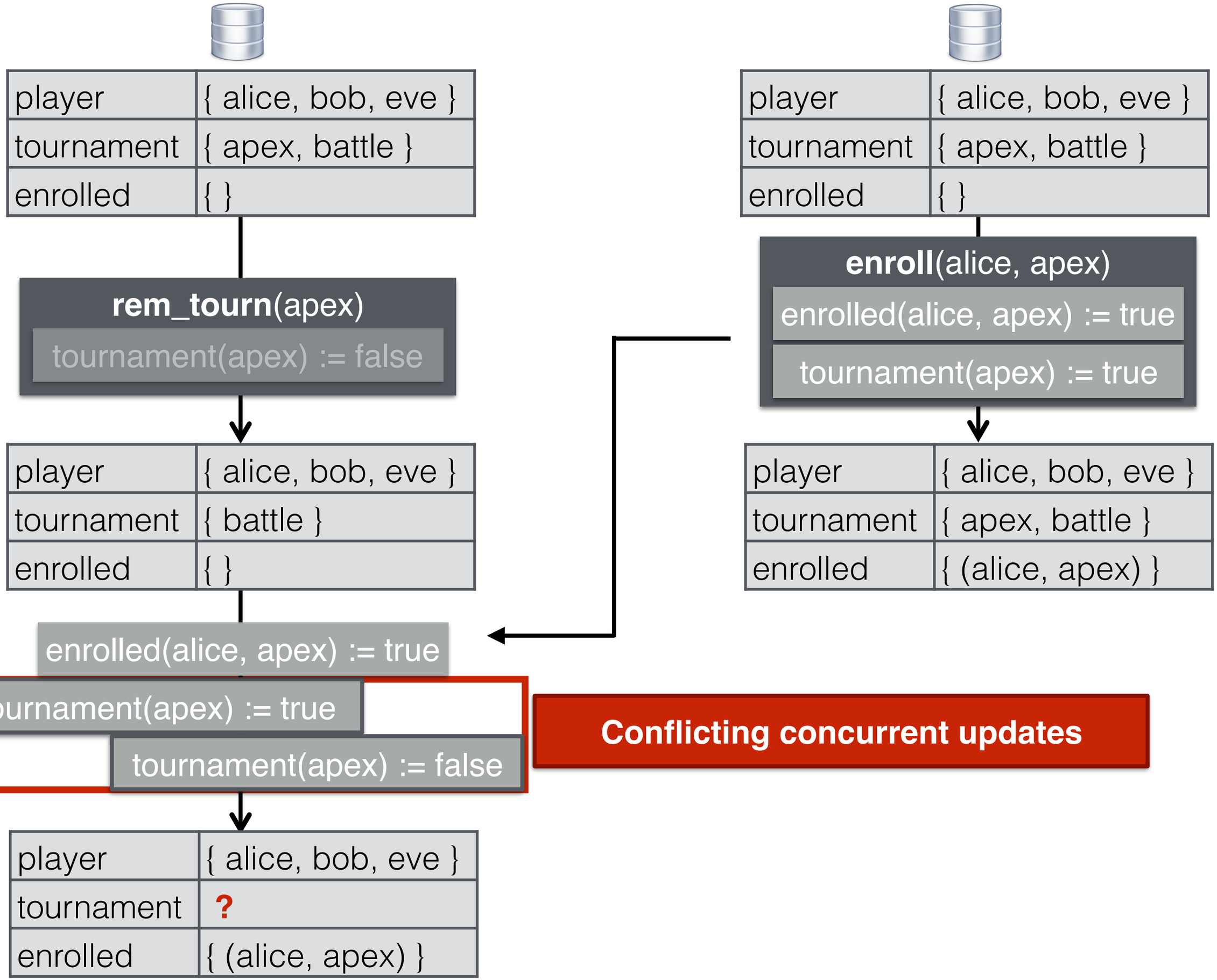
player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ }

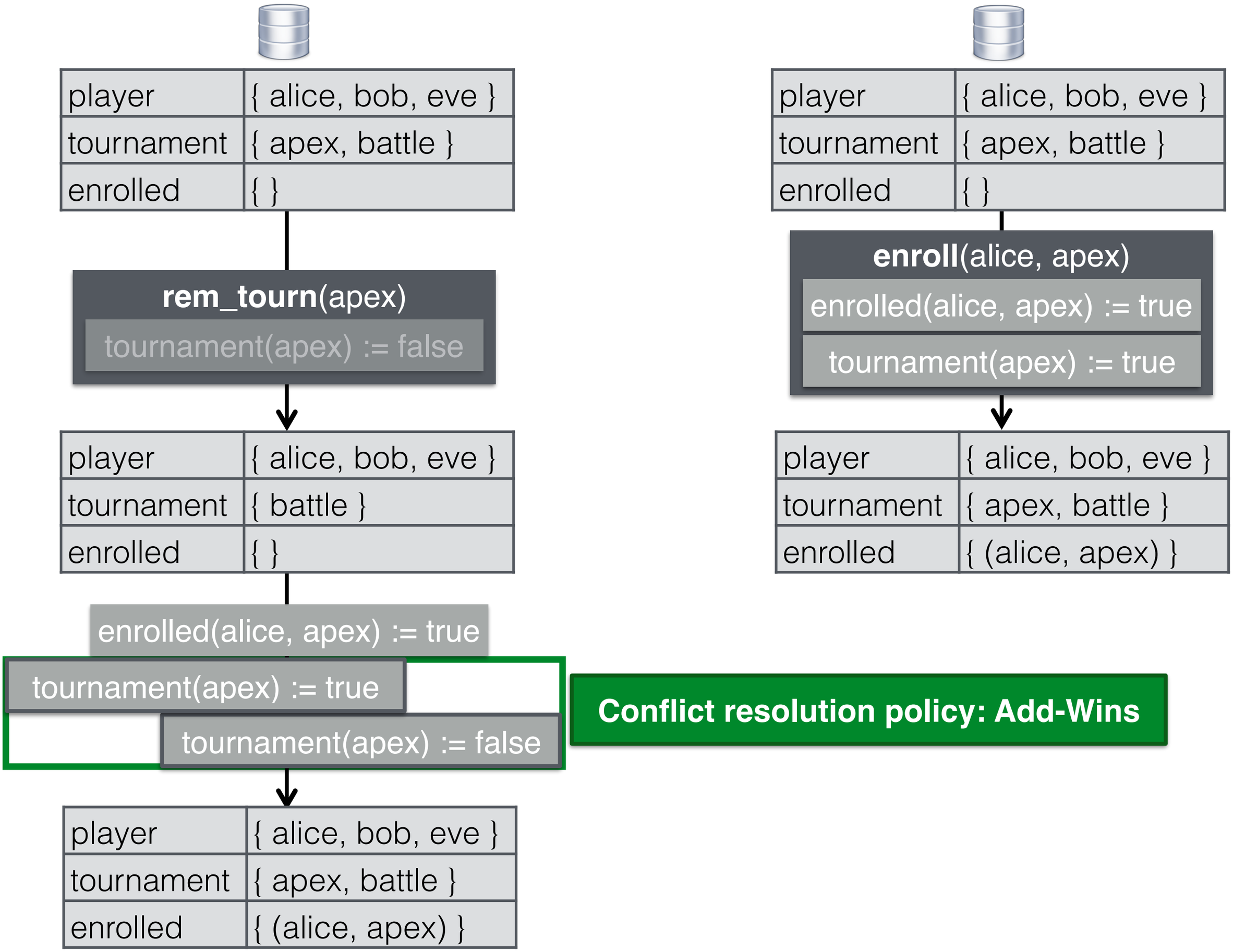
enroll(alice, apex)

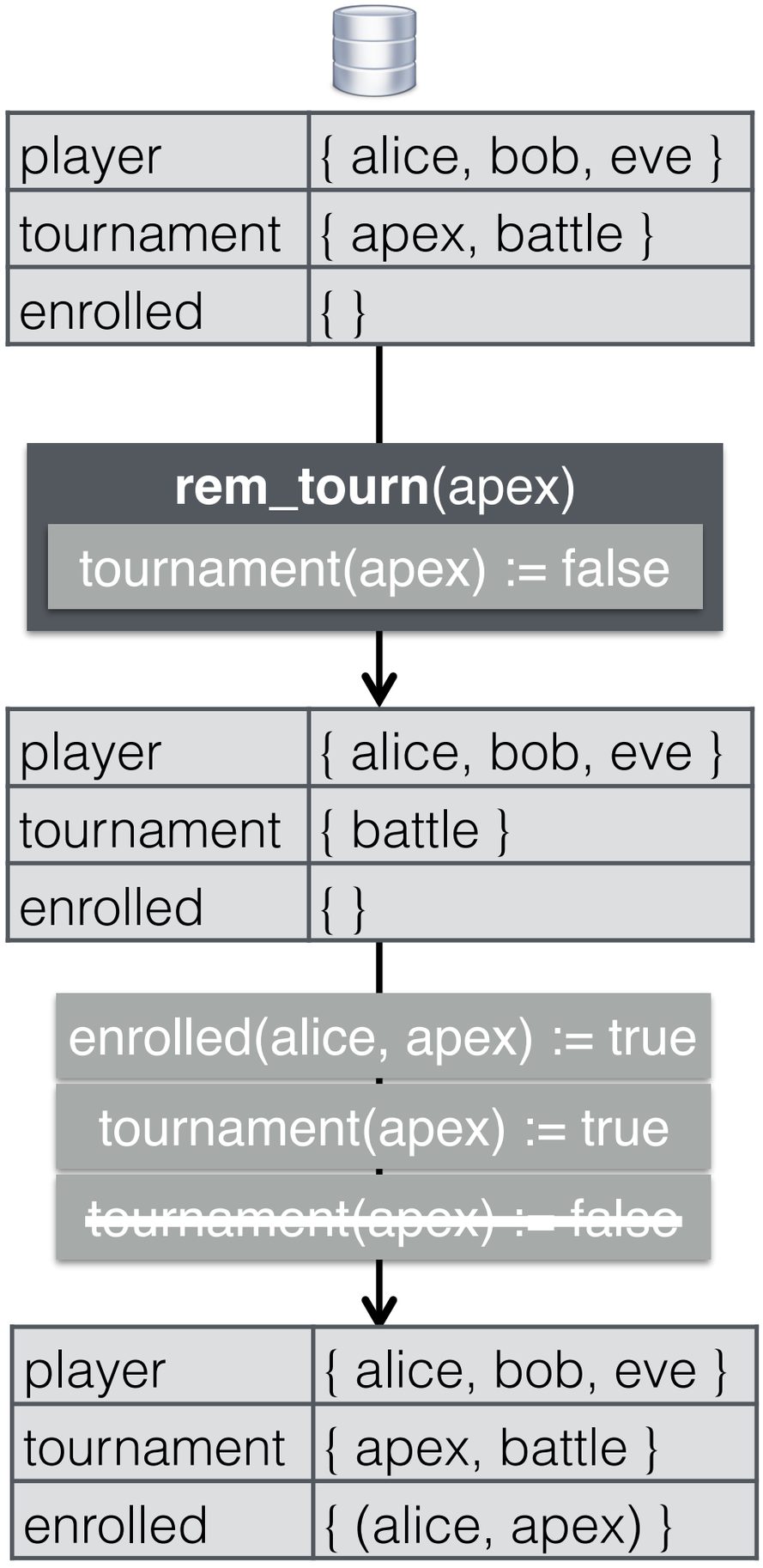
enrolled(alice, apex) := true

tournament(apex) := true

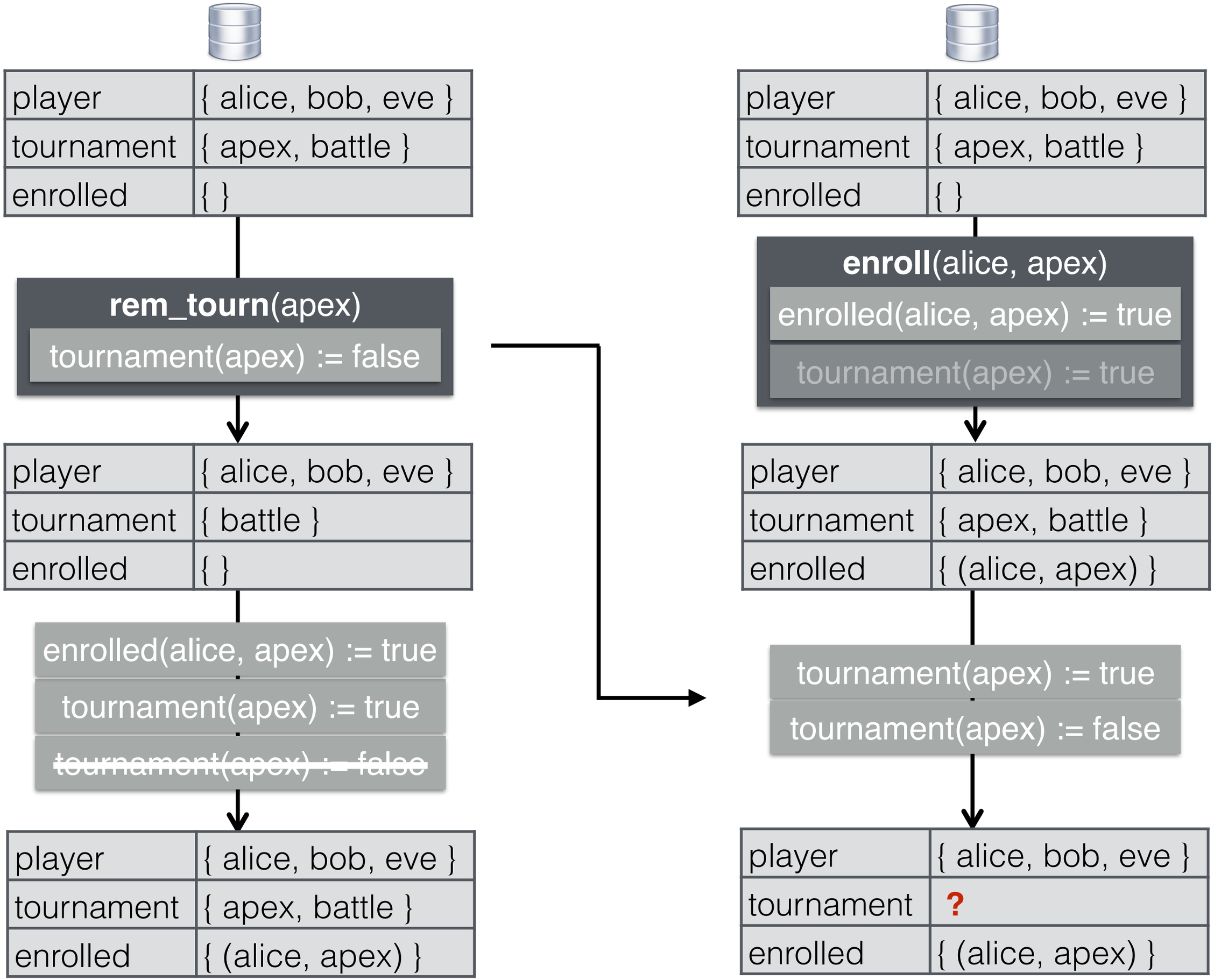
player	{ alice, bob, eve }
tournament	{ apex, battle }
enrolled	{ (alice, apex) }

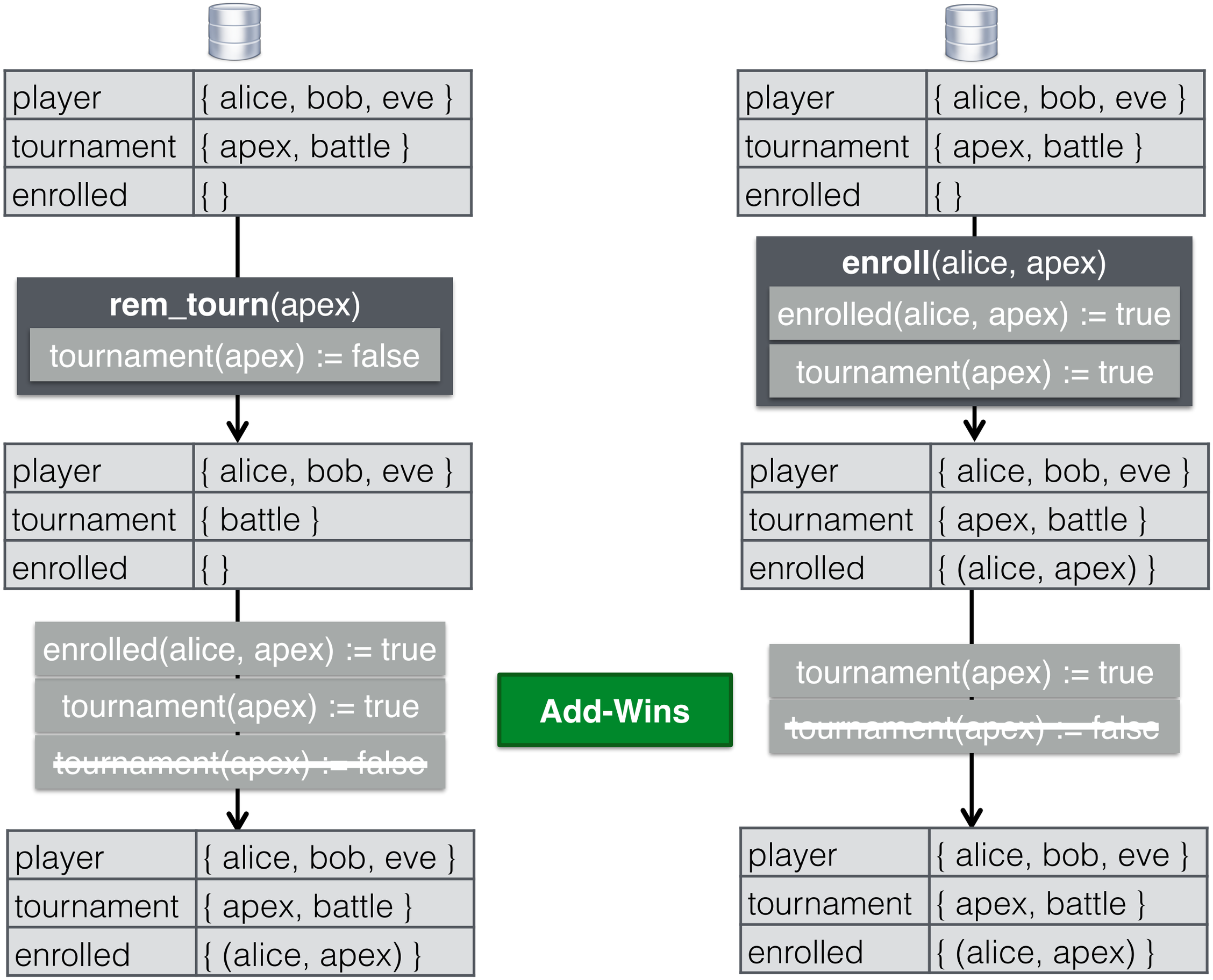

player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ }
rem_tourn(apex)
tournament(apex) := false
player
{ alice, bob, eve }
tournament
{ battle }
enrolled
{ }
enrolled(alice, apex) := true
tournament(apex) := true
tournament(apex) := false
player
{ alice, bob, eve }
tournament
?
enrolled
{ (alice, apex) }
enroll(alice, apex)
enrolled(alice, apex) := true
tournament(apex) := true
player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ (alice, apex) }
Conflicting concurrent updates

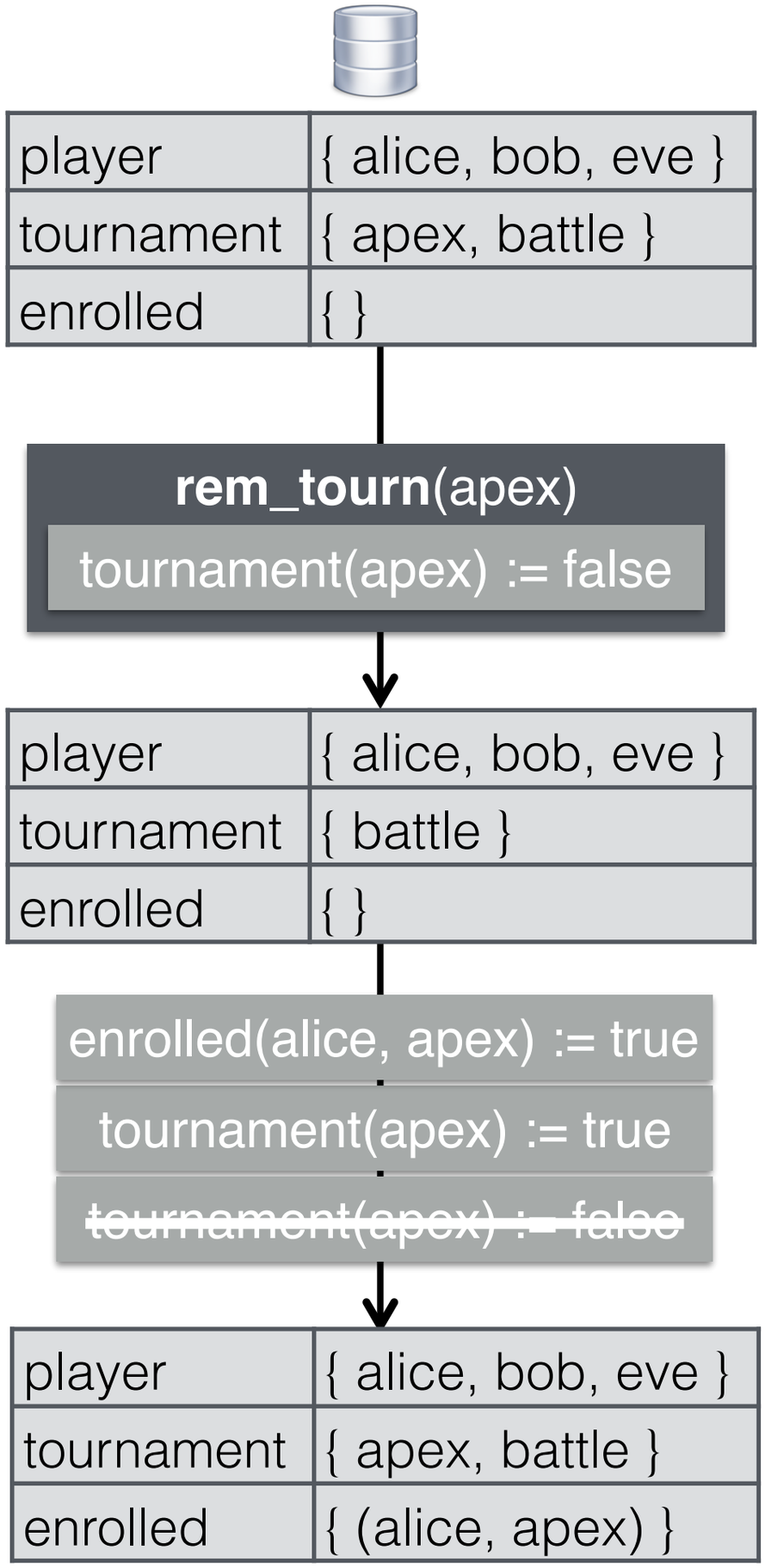

player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ }
rem_tourn(apex)
tournament(apex) := false
player
{ alice, bob, eve }
tournament
{ battle }
enrolled
{ }
enrolled(alice, apex) := true
tournament(apex) := true
tournament(apex) := false
player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ (alice, apex) }
enroll(alice, apex)
enrolled(alice, apex) := true
tournament(apex) := true
player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ (alice, apex) }
Conflict resolution policy: Add-Wins

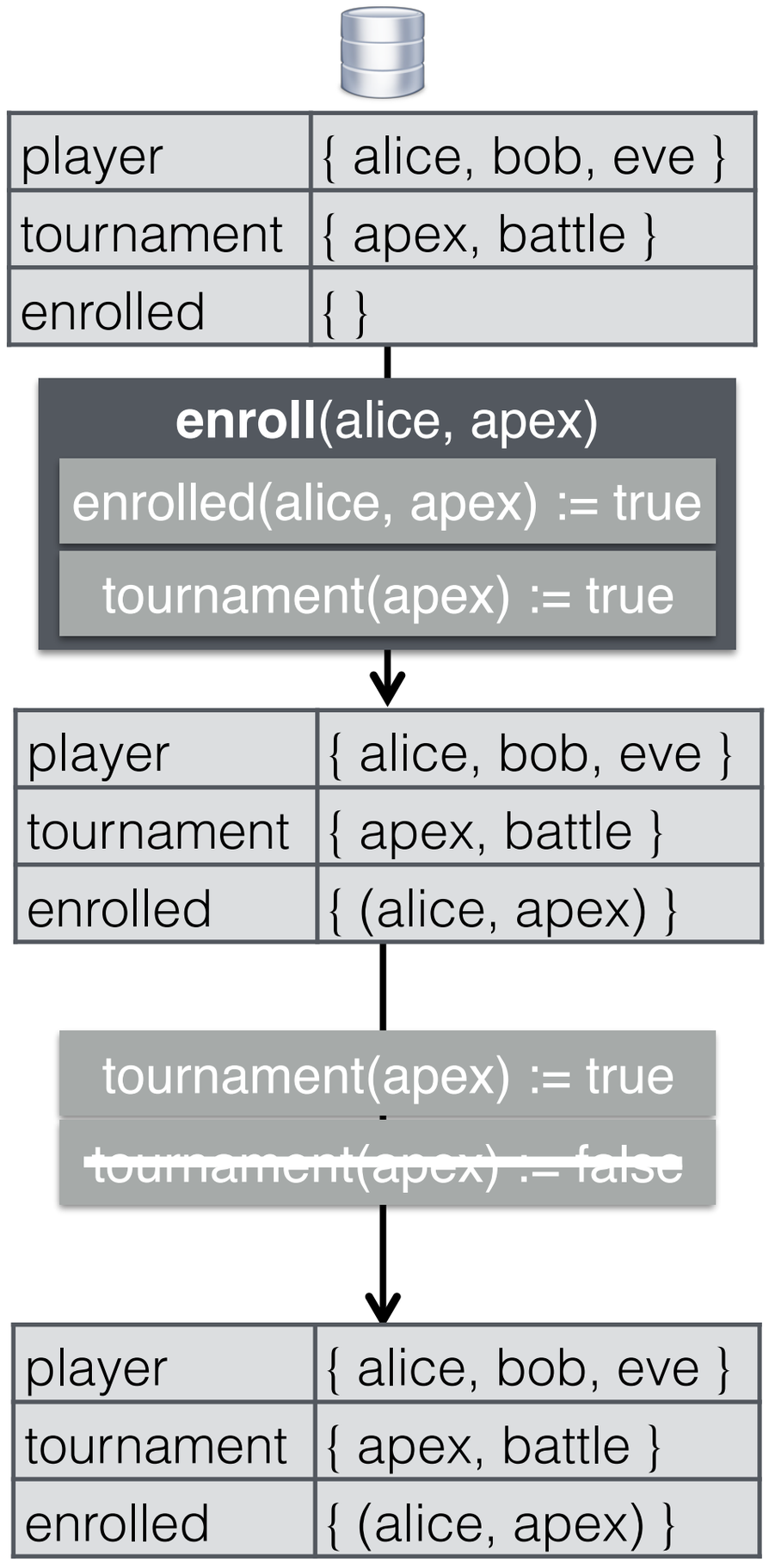


Conflict resolution policy: Add-Wins


player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ }
rem_tourn(apex)
tournament(apex) := false
player
{ alice, bob, eve }
tournament
{ battle }
enrolled
{ }
enrolled(alice, apex) := true
tournament(apex) := true
tournament(apex) := false
player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ (alice, apex) }
player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ }
enroll(alice, apex)
enrolled(alice, apex) := true
tournament(apex) := true
player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ (alice, apex) }
tournament(apex) := true
tournament(apex) := false
player
{ alice, bob, eve }
tournament
?
enrolled
{ (alice, apex) }


player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ }
rem_tourn(apex)
tournament(apex) := false
player
{ alice, bob, eve }
tournament
{ battle }
enrolled
{ }
enrolled(alice, apex) := true
tournament(apex) := true
tournament(apex) := false
player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ (alice, apex) }
Add-Wins
player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ }
enroll(alice, apex)
enrolled(alice, apex) := true
tournament(apex) := true
player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ (alice, apex) }
tournament(apex) := true
tournament(apex) := false
player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ (alice, apex) }


player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ }
rem_tourn(apex)
tournament(apex) := false
player
{ alice, bob, eve }
tournament
{ battle }
enrolled
{ }
enrolled(alice, apex) := true
tournament(apex) := true
tournament(apex) := false
player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ (alice, apex) }


player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ }
enroll(alice, apex)
enrolled(alice, apex) := true
tournament(apex) := true
player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ (alice, apex) }
tournament(apex) := true
tournament(apex) := false
player
{ alice, bob, eve }
tournament
{ apex, battle }
enrolled
{ (alice, apex) }

Future work

- Applications are no longer monoliths, now we have loosely-coupled microservices
  - Each microservice might have its own database
  - Intra and inter-database invariants
  - Cross-service operations
  - Reasoning has to be compositional
- Push-button verification (no specification effort)
  - Extract spec from code
    - Good results for Ruby on Rails [Bocic+ ICSE'14, Bocic+ ICSE'17]

References

[Balegas+ VLDB'19]
IPA: invariant-preserving applications for weakly consistent replicated databases. Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça

[Gotsman+ POPL'16]
'Cause I'm strong enough: Reasoning about consistency choices in distributed systems. Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, Marc Shapiro

[Balegas+ EuroSys'15]
Putting consistency back into eventual consistency. Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, Marc Shapiro