Testing Composable Specifications

Ken McMillan

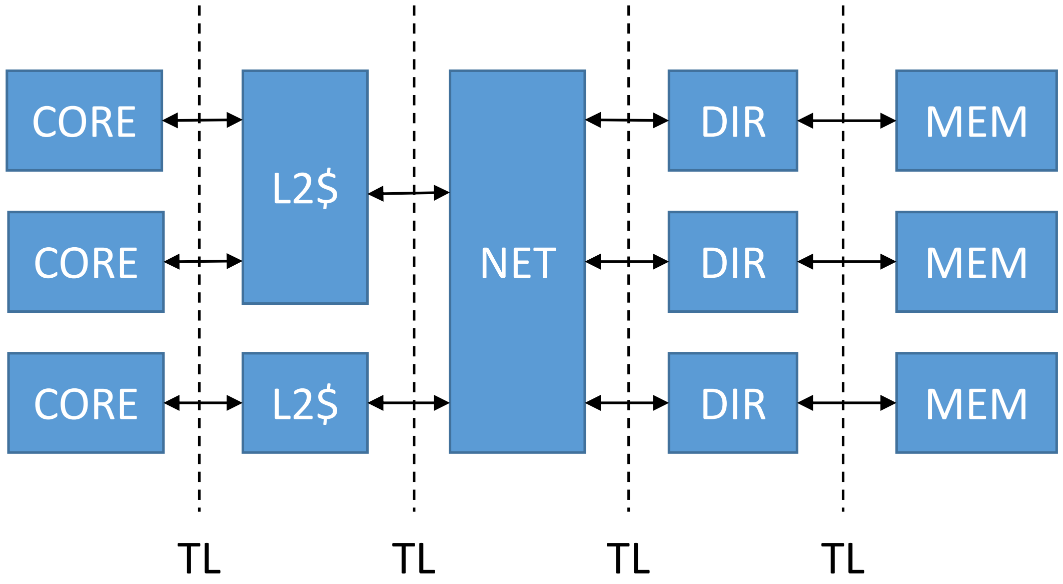
Microsoft Research

Case study

- TileLink is a protocol for implementing a coherent memory in a system-on-chip (SoC).
- Goal: a formal, modular specification of TileLink
  - Specify the protocol
  - Prove that it implements correct memory semantics
  - Rigorously test component implementations
  - Allow rapid configuration of SoC designs

TileLink system

- Hierarchy of memory system components for SoC using a common interface protocol.



Hierarchy implements *weakly consistent* memory model.

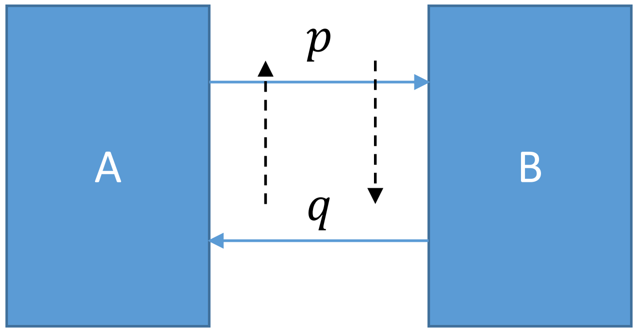
Modular verification

- General approach:
  - Write generic formal specifications of components
  - Verify components locally against specifications
  - Infer that systems of such components are correct
- *Composable* specifications:
  - Correctness of components implies correctness of system.
  - With a composable specification, we can assemble arbitrary configurations of components.

Some composable specifications are better than others, however…

Good composability

- Assume/guarantee specifications
  - A conjunction of temporal properties of interfaces
  - Assume/guarantee relationships



A: "$G\ (Hq \Rightarrow p)$"

B: "$G\ (Hp \Rightarrow q)$"

A||B: "$G(p \wedge q)$" composable!

This proof is checkable in P-time

We want our specifications to be composable "by construction".

Memory semantics

Memory operations:

op(loc,kind,addr,data)

CPU

read
write
atomic

Happens-before relation on operations:

$$\text{happens-before}(op_1, op_2) \Leftrightarrow \text{loc}(op_1) = \text{loc}(op_2) \wedge \text{time}(op_1) < \text{time}(op_2)$$
$$\wedge\ (\text{addr}(op_1) = \text{addr}(op_2) \vee \text{atomic}(op_1) \vee \text{atomic}(op_2))$$

Consistency:

A sequence of ops is consistency if every read sees value of most recent write.

Weak consistency:

A set of operations is weakly consistent if there exists an ordering π s.t:

- π respects happens-before
- π is consistent

Problem

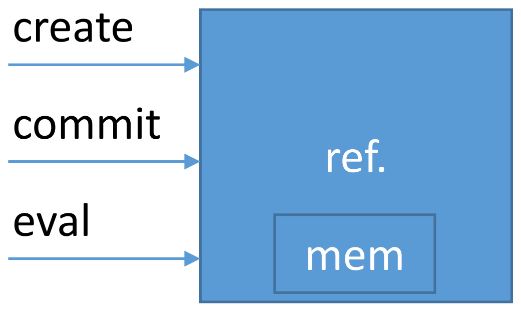
- How do you write a “good” composable specification for a system if its key property refers to all events in the system?

How do we witness the serialization π?

How do local operations fit into the global serialization?

Solution

- Add a “reference object”.
  - Constructs the witness for π.
  - Verifies consistency π as it is constructed



create : op × loc → stamp

commit : stamp → unit

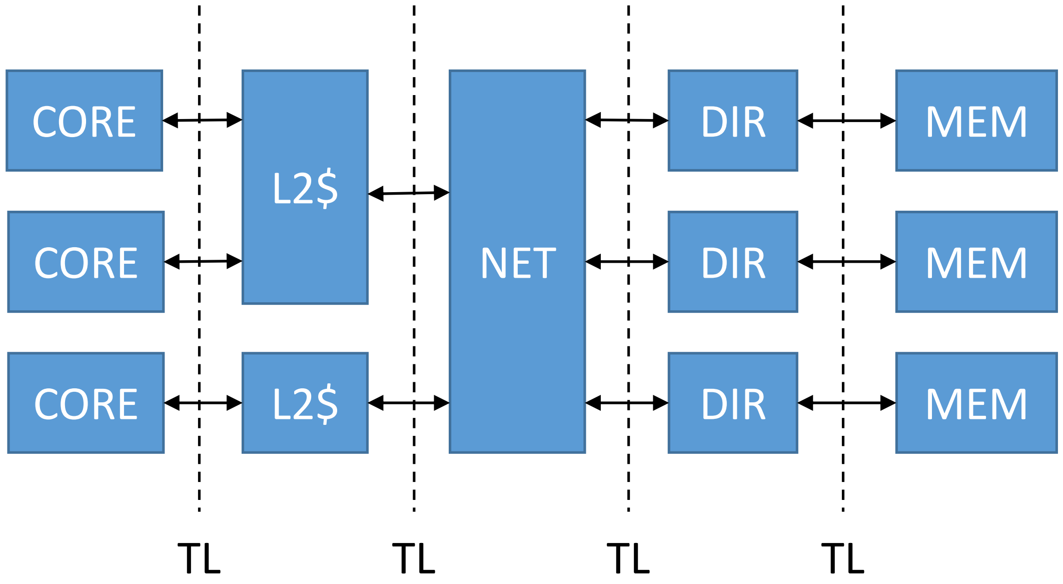
eval : stamp → value

commit(stamp):	*assumes*	happens-before(X,op(stamp)) ⇒ committed(X)
value = eval(stamp):	*assumes*	committed(stamp)
	guarantees	value = result(π,op(stamp))

These operations allow us to define the semantics of the system interfaces.

TileLink system

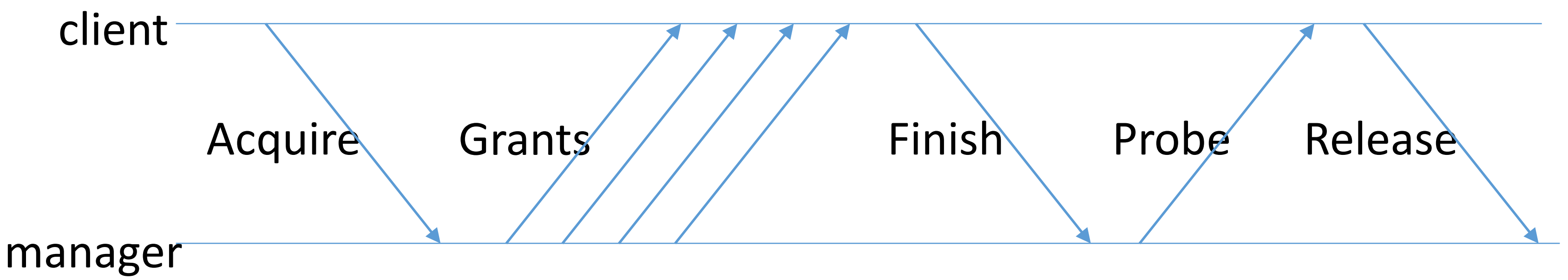
- Hierarchy of memory system components for SoC using a common interface protocol.



TileLink interface protocol

- Protocol messages implement
  - Coherent requests (MESI)
  - Invalidation
  - Ordered, non-coherent operations
- Interface has two roles:
  - Client ≈ processor
  - Manager ≈ memory

Typical transaction flow at interface



Writing a “good” composable spec

- Specification has two parts:
  - Temporal properties of interface
  - Assume/guarantee relationships between properties
- Interface properties of two types:
  - Interface protocol properties
  - Semantic properties, relative to reference object

Semantic interface properties

These properties refer to the reference object to define ordering and data values at the interface.

- Manager-side properties
  - M[1]: Data in cached *Grant* must match ref.mem.
  - M[2]: If uncached resp. then committed(stamp)
  - M[3]: If uncached resp. then data = eval(stamp)
- Client-side properties
  - C[1]: Data in cached Release must match ref.mem.
  - C[2]: If uncached req. then happens-before(X,stamp) implies requested(X).
  - C[3]: If uncached resp. then data = eval(stamp)

Commitment properties

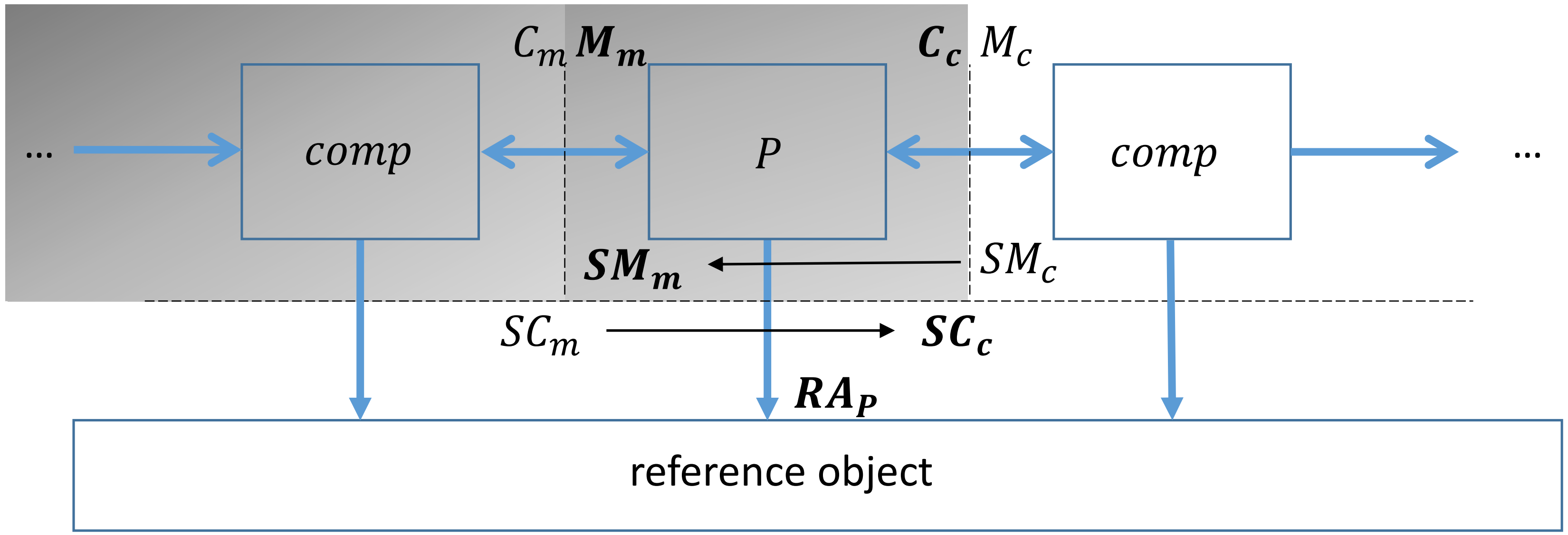
The coherence state determines what commitments are allowed on either side of the interface. This is the function of coherence.

- Client-side commitments:
  - SC[1]: Read may be committed on client side only if interface has *shared* or *exclusive* permissions.
  - SC[2]: Write may be committed on client side only if interface has *exclusive* permissions.
- Manager-side properties
  - SM[1]: Read may be committed on manager side only if interface has *shared* or *invalid* permissions.
  - SM[2]: Write may be committed on manager side only if interface has *invalid* permissions.

Note: “client side” means *any* component left of the interface.

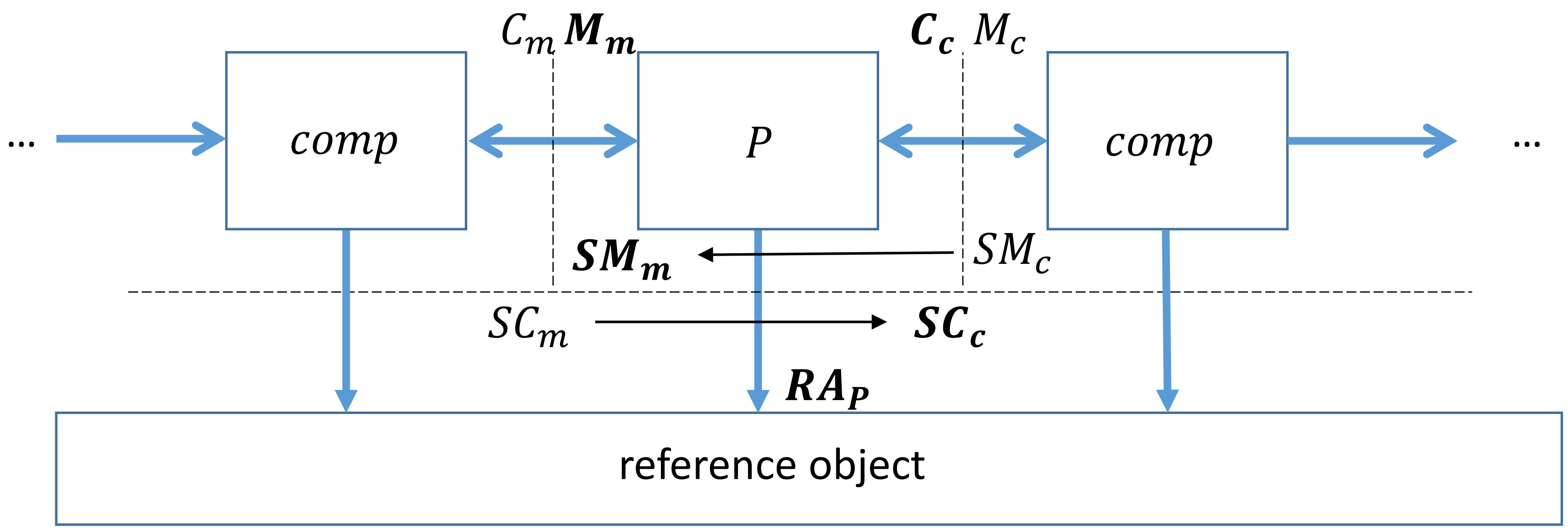
Assume/guarantee relationships

- An L2 cache has TileLink interfaces on processor side and memory side.



Assume/guarantee relationships

- An L2 cache has TileLink interfaces on processor side and memory side.



P,R: $C_m^-, M_c^- \rightarrow C_c, M_m$
P,R: $SC_m, C_m^-, M_c^- \rightarrow SC_c$
P,R: $SM_c, C_m^-, M_c^- \rightarrow SM_m$
P,R: $C_m^-, M_c^-, SM_c^-, SC_m^- \rightarrow RA_P$

$$\overline{}$$
$$G\ RA_p$$

Checking this proof is a purely syntactic operation

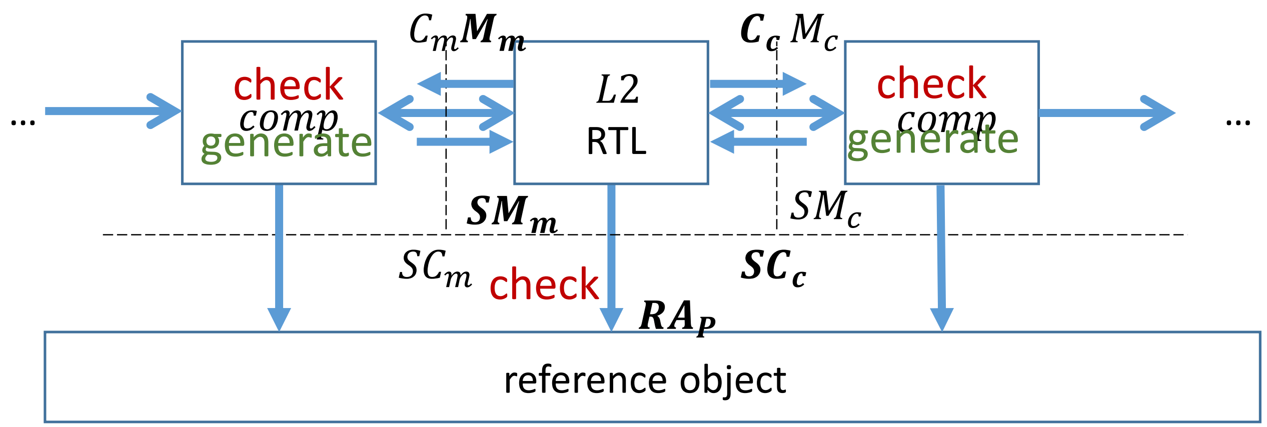
Formal proofs

- We can now formally verify components in isolation against their assume/guarantee specifications:
  - Reording buffer
  - Hierarchical cache
  - Processor, memory, etc.
- These are simple abstract component models, intended to show that the specification has the intended implementations.
  - Show key property that protocol is insensitive to message re-ordering.
  - In the process, specification was corrected.

Because our assume/guarantee specification is composable, we know that hierarchies built from these components implement a weakly consistent shared memory.

Compositional testing

- From an assume/guarantee specification, we can automatically generate a test environment.



- Tested two RTL level components with randomized generation using Z3:
    - L2 cache bank
    - Snooping hub

Testing results

- Compositional testing revealed currency errors in the RTL in under 1s (< 100 cycles)
  - Unit testing provides much greater flexibility in covering internal corner cases
  - Randomized specification-based testing reduces bias
- Latent bugs
  - Most bugs could not be stimulated in integration test
  - Latent bugs affect re-usability
- Importance of composability
  - All system-level errors exposed to unit testing
  - Gain confidence that components can be assembled into arbitrary configuration.

Conclusion

- Good composable specification is such that:
  - Correct component imply correct system
  - The proof of this is efficiently checkable
- Global properties (such as memory consistency)
  - Reference object + temporal assume/guarantee
  - Allows local specification of interface semantics
- Composable TileLink interface spec provides:
  - Documentation of the interface
  - Ability to reason formally about specification
  - Efficient and rigorous test to find latent bugs

Composable specifications provide a way to formal verification experts to provide value in an environment where most engineers do not have formal skills.

Specification as a social process

- The specification develops over time in collaboration with the system architects.
  - Ambiguities in informal specs must be resolved.
  - Initial formal spec almost certainly does not reflect designers intention.
  - Mismatch with implementation may indicate properties should be strengthened or weakened for efficiency.
- Over time the formal spec becomes a valuable document.
  - Encapsulates design knowledge.
  - Allows rigorous testing and verification.