Verifying Safety of a Token Coherence Implementation by Compositional Parametric Refinement

Sebastian Burckhardt
Rajeev Alur
Milo M.K. Martin

University of Pennsylvania
Verifying Token Coherence

- *Token Coherence* is a novel approach for multiprocessor memory systems that reconciles the diverging requirements of *performance / cost / verifiability*
Verifying Token Coherence

- Token Coherence is a novel approach for multiprocessor memory systems that reconciles the diverging requirements of performance / cost / verifiability.
- We develop a compositional method for verifying *Token Coherence Safety* and apply it to a detailed implementation:
  - we found 7 bugs
  - we demonstrate scalability of the method
Related Work

- K. L. McMillan. *A compositional rule for hardware design refinement*
  - compositional assume-guarantee refinement rules for synchronous process model
- Our abstract protocol resembles work by
  - G. Delzanno (state multisets for parametric cache coherence verification)
  - Arvind and X. W. Shen (term rewriting for functional hardware specifications)
  - G. Berry and G. Boudol (chemical abstract machine)
Background: Shared-Memory Multiprocessor

- Processors access the ‘same’ memory (reads and writes to memory addresses)
Coherent Memory

- Asynchronous interconnection network
- Processors cache local copies as needed
- Coherence Protocol prevents access to stale copies
Example: Coherence by “Single Writer, Multiple Reader”

P1, P2 have read-only copies

P3 wants to access and modify the block

P1

P2

P3
Example: Coherence by “Single Writer, Multiple Reader”

P1
C
N
D
3

P2
C

P3
C

wants to access and modify the block
Example: Coherence by “Single Writer, Multiple Reader”

has exclusive copy → can safely write
Performance, Cost, and Complexity

- **Performance**: prefer direct communication to centralized directory (lower cache-to-cache latency)
- **Cost**: prefer unordered interconnect (e.g. mesh topology)
- But this creates complexity, i.e. it’s hard to design a protocol that is correct and fast.
  - races need arbitration
  - unordered, asynchronous broadcast creates unreliable snapshots of the system state
Performance, Cost, and Complexity —

Token Coherence

- **Performance**: direct communication ✓
- **Cost**: unordered interconnect ✓
- **Complexity**: ✓
  - use token counting to guarantee safety.
    - there is a fixed number of tokens per cache block
    - require one token to read, all tokens to write
  - use timeouts to guarantee liveness.
    - after timeout, use slower, but reliable mechanism
Performance, Cost, and Complexity — Token Coherence

- **Performance**: direct communication
- **Cost**: unordered interconnect
- **Complexity**: ✓

- Use token counting to guarantee safety.
  - There is a fixed number of tokens per cache block
  - Require one token to read, all tokens to write
- Use timeouts to guarantee liveness.
  - After timeout, use slower, but reliable mechanism

Allows scalable formal verification
Tokens help to verify safety

• Our method is scalable because
  • it’s *compositional* --- each component is model-checked individually
  • it’s *parametric* --- safety is proved for an arbitrary number of caches

• How do we achieve that?
Tokens help to verify safety

- Our verification method is scalable because
  - it’s compositional --- each implementation component is model-checked individually
  - it’s parametric --- safety is proved for an arbitrary number of caches

- How do we achieve that?
  - we develop a formal model, the ‘abstract token protocol’ and prove that it is coherent
  - we check refinement of this abstract protocol in a compositional and inductive manner.
Token counting rules
(these are the original rules)

Basic Rules
• Fixed number of tokens per cache block
• Require at least one token for read access.
• Require all tokens for write access.

Extended (MOESI) Rules
• One of the tokens is “owner” token
• Owner token can be clean or dirty
  • dirty owner token indicates that writeback is needed
We take it one step further:

**Abstract Token Protocol**

- Model single cache block only
  - This is sufficient because the memory system is not required to be sequentially consistent. The only requirement is coherence.

- Express **all** relevant state as tokens
  - use token $v$ to represent data value
  - use token $v$ to represent the memory value

- Express state of components as a token bag (multiset)
Token Transition System

System components and messages are abstractly modeled as token bags

- Bags can split (= send a message)
  ![Diagram of bags splitting]

- Bags can fuse (= receive a message)
  ![Diagram of bags fusing]

- Bags can perform local reactions
  ![Diagram of bags reacting]
Rules for a 3-token system

Rewrite Rules:

Read v

Write w

Memread

Memwrite

Copy

Drop

Tokens:

Regular Token
Clean Owner Token
Dirty Owner Token
Data Token
Memory Token

Invariants:

v must have Regular Token in same bag
D must have v in same bag
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token

wants to read
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token

v1 wants to read
Abstract Token Protocol: Example

v1 wants to read

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token

v1 wants to read v1
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token

wants to read

v1
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token
Abstract Token Protocol: Example

wants to read

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token
Abstract Token Protocol: Example

wants to read

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token

v1 wants to read
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token

V1 wants to read
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token

v1 wants to read
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token

Example tokens: v1 reads v1
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token

wants to modify
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token

wants to modify
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token

wants to modify
Abstract Token Protocol: Example

- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token

$v_1$ wants to modify
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token

v1 wants to modify
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token

C can now modify: adds 1
Abstract Token Protocol: Example

wants to modify

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token
Abstract Token Protocol: Example

wants to modify

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token
Abstract Token Protocol: Example

wants to modify

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token

v1

wants to modify

v2

D

C

Abstract Token Protocol: Example

can now modify

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token
Abstract Token Protocol: Example

wants to evict

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token
Abstract Token Protocol: Example

Tokens :
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token
Abstract Token Protocol: Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token
Abstract Token Protocol:
Example

Tokens:
- Regular Token
- Clean Owner Token
- Dirty Owner Token
- Data Token
- Memory Token
Rules for a 3-token system

Rewrite Rules:

Read v

v \rightarrow v

Write w

C/D v \rightarrow D w

Memread

v C \rightarrow v C v

Memwrite

w v D \rightarrow w C

Copy

v \rightarrow v v

Drop

v \rightarrow

Invariants:

v must have \text{Regular Token} in same bag

D must have \text{Data Token} in same bag

Tokens:

\text{Regular Token}

\text{Clean Owner Token}

\text{Dirty Owner Token}

\text{Data Token}

\text{Memory Token}
Rules for a 3-token system

Rewrite Rules:

Read v
Memread
Memwrite
Drop

Tokens:

Regular Token
Clean Owner Token
Dirty Owner Token
Data Token
Memory Token

We prove by hand that the abstract token protocol is coherent.

The proof is quite simple because token bags (multisets) are intuitive to reason about.
What’s next

- Formal spec of abstract token protocol ✓
  - Multisets
  - Rewrite rules
- Abstract token protocol is coherent ✓
  - Direct proof by hand

Next: we take an *actual implementation* and do a compositional parametric refinement verification.
The implementation...

- is fairly low-level
  - it’s not just about tokens – it contains many microarchitectural details, as it was used for performance simulations
  - it includes the secondary (liveness) protocol
- is too big to be efficiently verifiable by purely manual methods
- is coded in SLICC
  - specification language for implementing cache coherence
  - generates html-formatted overview tables
This is the overview table for the cache controller, generated from the SLICC code by the SLICC compiler.
SLICC code specifies action sequences to execute for a given state & trigger:

```plaintext
transition(NP, Ifetch, IS) {
    pp_allocateL1ICacheBlock;
    i_allocateTBE;
    a_issueRequest;
    uu_profileMiss;
    k_popMandatoryQueue;
}
```

SLICC code defines each action:

```plaintext
action(d_sendDataWithAllTokens, "d") {
    peek(requestNetwork_in, RequestMsg) {
        enqueue(responseNetwork_out, ResponseMsg) {
            out_msg.Address := address;
            out_msg.Type := DATA_OWNER;
            out_msg.Sender := id;
            out_msg(SenderMachine := MachineType:Directory;
            out_msg.Destination.add(in_msg.Requestor);
            out_msg.DestMachine := MachineType:L1Cache;
            assert(directory[in_msg.Address].Tokens > 0);
            out_msg.Tokens := directory[in_msg.Address].Tokens;
            out_msg.DataBlk := directory[in_msg.Address].DataBlk;
            out_msg.Dirty := false;
            out_msg.MessageSize := MessageSizeType:Response_Data;
            directory[in_msg.Address].Tokens := 0;
        }
    }
}
```
Why we need compositionality

- Verifying refinement of the entire system is beyond the practical limit of model checking
  - too many states for explicit model checker
  - BDDs too complex for symbolic model checker
Verify components individually

C refines D

D refines C

N refines

4 3 4

D C
But... there are some issues

- Components make assumptions on the context
  - if the context does not preserve token numbers, the implementation is not required to refine the specification.
- Solution: define contextual refinement
Contextual Refinement

- Say: Q is specification, P is implementation

- classic refinement:
  \( P \prec Q \) means
  any behavior of P is a possible behavior of Q

- contextual refinement:
  \( P \preceq_C Q \) means
  any behavior that P can exhibit in context C is a behavior that Q can exhibit in context C
Compositional Proof

If each system component refines the abstract protocol $T$ in context $T$:

\[
C \preceq_T T \quad \text{and} \quad D \preceq_T T \quad \text{and} \quad N \preceq_T T
\]

then the entire system refines the abstract protocol:

\[
C \mid C \mid C ... \mid C \mid N \mid D \preceq_T T
\]

Proof:

- uses assume-guarantee proof rules inductively
- exploits the deep symmetries of abstract protocol $T$
Formalism
(read the paper for more detail on this.)

- a process model based on finite traces
  - process = set of traces
  - interleaved concurrency
  - synchronous communication (CCS-style)
- contextual refinement
  - we use a direct definition for this process model
  - can handle open and closed refinement
- assume-guarantee proof rules
  - can be applied inductively
  - allow us to use context abstraction
Visualization:
Abstracting the context

- messages and nodes look alike
- abstract protocol is self-similar: $T | T | ... | T = T$
  means that context can be modeled as single bag
What’s next?

- We discharge the local obligations

\[
\begin{align*}
C & \not\leq_T T \\
D & \not\leq_T T \\
N & \not\leq_T T
\end{align*}
\]

by model checking the implementation
Model checking in Murphi

- Do translation & abstraction of SLICC
  - remove non-relevant state
  - introduces non-determinism
  - done by hand
- Model the abstract token coherence
  - Model is used both for spec. and context
  - use finite representations of bags
- Need correspondence between impl. and spec.
  - We use annotations to guide the model checker
    (eliminates existential quantifier from refinement problem)
Found 7 bugs

• Some debug-only code was not properly marked as such

• In four instances, state was updated incorrectly for the case where the last token was sent
  • was not hit by sim because the ratio tokens vs. nodes was large in all sim configurations

• Dirty bit was updated too often
  • causes superfluous writebacks – is a performance bug
  • would not have been found had we checked for coherence only, rather than the stronger refinement obligation

• Once the above were fixed, the dirty bit was no longer updated often enough
Results

- Found 7 bugs
  - note that implementation had been subjected to extensive simulation
- Model checking was very resource-efficient
  - well within reach of Murphi, with room to spare
  - indicates that the method scales to lower-level implementations
Contributions

• We perform a formal verification of a system-level implementation of token coherence

• We present a general, formal specification of the safety substrate of token coherence

• We demonstrate that the token abstraction makes the verification more scalable
Future Work

• We would like to verify liveness.
  • not a priori clear if same framework is beneficial.

• We would like to apply token formalisms to other protocols.
  • try to retrofit on existing protocols (e.g. define virtual tokens)

• We would like to automate the procedure.
  • Translation seems automatable
  • Apply algorithms for rewrite logic
  • Eliminate or reduce annotations
• Visit my homepage for
  • an extended version of the paper
  • this presentation in ppt format
  • additional resources (e.g. SLICC code)

• Thank you for your attention.