Verifying Safety of a Token Coherence Implementation by Parametric Compositional Refinement[⋆]

Sebastian Burckhardt, Rajeev Alur, and Milo M.K. Martin

Department of Computer Science
University of Pennsylvania
{sburckha,alur,milom}@cis.upenn.edu

Abstract. We combine compositional reasoning and reachability analysis to formally verify the safety of a recent cache coherence protocol. The protocol is a detailed implementation of *token coherence*, an approach that decouples correctness and performance. First, we present a formal and abstract specification that captures the safety substrate of token coherence, and highlights the symmetry in states of the cache controllers and contents of the messages they exchange. Then, we prove that this abstract specification is coherent, and check whether the implementation proposed by the protocol designers is a refinement of the abstract specification. Our refinement proof is parametric in the number of cache controllers, and is compositional as it reduces the refinement checks to individual controllers using a specialized form of assume-guarantee reasoning. The individual refinement obligations are discharged using refinement maps and reachability analysis. While the formal proof justifies the intuitive claim by the designers about the ease of verifiability of token coherence, we report on several bugs in the implementation, and accompanying modifications, that were missed by extensive prior simulations.

1 Introduction

Shared memory multiprocessors have become the most important architecture used for commercial and scientific workloads. Such systems use hardware cache coherence protocols to create the illusion of a single, shared memory without caches. These protocols are important factors of the overall system performance, and numerous optimizations contribute to their complexity. Since hard-to-cover race conditions elude simulations of the protocols, formal methods are often employed to verify their correctness.

Token Coherence is a new approach to cache coherence protocols that decouples correctness requirements from performance choices, claiming to improve both performance and verifiability [22]. Separate correctness mechanisms ensure safety and liveness. *Safety* is achieved by token counting: per memory location,

⋆ This research was partially supported by the NSF award CCR0306382, and a donation from Intel Corporation.

the number of tokens in the system is a global invariant. By requiring at least one token for read access and all tokens for write access, the protocol directly enforces a single-writer, multiple-reader policy. On the other hand, *Liveness* is achieved by persistent requests. This reliable, but slower protocol is used when the regular requests do not succeed within a timeout period. Persistent requests are required because the regular requests, while likely to complete quickly, do not guarantee eventual success.

In this work, we combine compositional verification and model checking to verify the safety of a detailed implementation of a token coherence protocol for an arbitrary number of caches. Our method takes advantage of the opportunities offered by the token coherence design. It proceeds in four steps.

1. We present a formal specification of the safety substrate of token coherence. This abstract protocol is based on rewrite rules and multisets, and expresses the symmetry between components and messages. It applies to arbitrary network topologies, cache numbers, and even cache hierarchies.
2. We prove manually that the abstract protocol is safe (i.e. coherent). The verification problem is thus reduced to checking that the implementation correctly refines the abstract protocol.
3. We prove that the refinement can be verified for each component individually, by replacing its context with an abstraction. We prove that this decomposition into local refinement obligations is sound, using a variant of assume-guarantee reasoning based on *contextual refinement*, and performing an induction on the number of caches.
4. We discharge the local refinement obligations with the conventional model checker Murφ [12, 11]. To obtain the models, we manually translate, abstract and annotate the implementation code. This procedure reduces the refinement checking to a reachability problem, which Murφ solves by enumerative state space search.

Even though the protocol implementation had been extensively simulated prior to this work, we discovered a few bugs, and were able to fix them quickly with the help of counterexamples produced by the model checker. The compositional refinement method proved to be effective in avoiding the state space explosion problem [16] which is commonly encountered in system-level models [28].

Because of the page limit, we had to omit most proofs. A more complete version of this article can be found online [7].

1.1 Related Work

Prior work on formal verification of cache coherence varies in (1) the protocol complexity and level of detail (2) the coverage achieved (safety, liveness, parametric systems) (3) the underlying tools (enumerative or symbolic model checkers, decision procedures, theorem provers), (4) reduction techniques (symmetry, abstraction, compositional verification), and (5) degree of automation.

We refer to Pong and Dubois [28] for a general survey, and to various illustrative efforts [23, 27, 14, 3].

Our proof methodology modifies and combines a variety of ideas in the formal verification literature. These include assume-guarantee reasoning for compositional verification (c.f. [1, 8, 2, 25]), structural induction for proving properties for arbitrary number of processes (c.f. [19, 9, 15, 13, 10, 4]), data abstraction (c.f. [32, 17]), use of term rewrite systems for hardware verification [5], and proving refinement using reachability analysis (c.f. [18]).

2 Process Model

In this section, we define the process model and introduce our assume-guarantee proof rules. We chose to define the process model from scratch, so to keep it concise and self-contained, and to obtain the desired combination of features. Except for the specialized definition of *contextual refinement*, all concepts (traces, composition, refinement) are standard and appear in many variations and combinations in the process algebra literature [29].

A process is defined as the set of its traces, which are finite words over an alphabet Σ of events. Σ is considered fixed and common to all processes. We further partition $\Sigma = \Sigma_e \cup \Sigma_c$ into disjoint subclasses: Σ_e contains events that are visible to external observers of the system only, while Σ_c describes synchronous communication events. Matching events in Σ_c (e.g. sending and receiving of a message) are denoted σ and $\overline{\sigma}$.

Definition 2.1. *A* process *P over Σ is a non-empty prefix-closed language; i.e. $P \subset \Sigma^*$, $P \neq \emptyset$ and for all $u, v \in \Sigma^*$: $uv \in P \Rightarrow u \in P$. A process P* refines *a process Q, written $P \preccurlyeq Q$, iff $P \subset Q$. A process P is* closed *if $P \subset \Sigma_e^*$.*

The refinement relation $\preccurlyeq$ is a complete partial order on the processes. The bottom (silent) process $\{\epsilon\}$ has but one trace: the empty string. The top (universal) process Σ^* includes all possible traces.

When composing processes, we merge their traces by interleaving their events and hiding mutual communication.

Definition 2.2. *Let $u, v, w \in \Sigma^*$ be traces. We define the relation $u \mid v \vdash w$ (speak: u, v can combine to form w) by the following inference rules:*

$$\frac{}{\epsilon \mid \epsilon \vdash \epsilon}\ (\textsc{epsilon}) \qquad \frac{u \mid v \vdash w \qquad \sigma \in \Sigma_c}{u\sigma \mid v\overline{\sigma} \vdash w}\ (\textsc{communication})$$

$$\frac{u \mid v \vdash w \qquad \sigma \in \Sigma}{u\sigma \mid v \vdash w\sigma}\ (\textsc{l-event}) \qquad \frac{u \mid v \vdash w \qquad \sigma \in \Sigma}{u \mid v\sigma \vdash w\sigma}\ (\textsc{r-event})$$

Example 2.3. Let $\Sigma_e = \{a, b, c, d\}$ and $\Sigma_c = \{e, \overline{e}\}$. Then we have

$$ab \mid cd \vdash acbd \qquad ab \mid cd \vdash abcd \qquad ae \mid \overline{e}b \vdash ab \qquad ae \mid \overline{e}b \vdash ae\overline{e}b$$

but not $ae \mid \overline{e}b \vdash ba$.

Definition 2.4. *Let P, Q be processes. Then $P \mid Q \doteq \{w \in \Sigma^* \mid \exists u \in P : \exists v \in Q : u \mid v \ \vdash w\}$.*

Composition is commutative and associative. Composition does not restrict its components: for processes P, Q we always have $P \preccurlyeq P \mid Q$. This same style of communication is used by CCS [26].

Refinement is preserved by composition: if $P' \preccurlyeq P$, then $P' \mid Q \preccurlyeq P \mid Q$. We can use this fact to prove that a system implementation refines its specification

$$P' \mid Q' \preccurlyeq P \mid Q \tag{1}$$

from the simpler, local refinement conditions

$$P' \preccurlyeq P \quad \text{and} \quad Q' \preccurlyeq Q\ . \tag{2}$$

However, this method is not very powerful, because the refinements (2) do often not hold because of implicit assumptions on the context. Assume-guarantee reasoning remedies this shortcoming. We provide the context as an explicit subscript to the refinement relation, enabling us to conclude (1) from

$$P' \preccurlyeq_Q P \quad \text{and} \quad Q' \preccurlyeq_P Q\ . \tag{3}$$

Most process models used for compositional refinement of hardware [2, 24] can express the contextual refinement $P' \preccurlyeq_Q P$ directly as $P' \parallel Q \preccurlyeq P$ (using synchronous parallel composition). The same does not work in our context (as exemplified by the observation 5 below), so we use a direct definition instead.

Definition 2.5 (Contextual refinement). *Let P, P', C be processes. Then P' is said to refine P in context C, written $P' \preccurlyeq_C P$, iff for all traces $u \in P'$ the following condition holds: if there is a trace $v \in C$ such that $u \uparrow \Sigma_c = \overline{v \uparrow \Sigma_c}$ (i.e. the communication events in u, v match up), then $u \in P$.*

Intuitively, we require that all behaviors of P' that are actually possible within an environment that adheres to C are allowed by P.

The following observations provide insight about contextual refinement.

1. For any process C, $\preccurlyeq_C$ is a pre-order on processes.
2. If $P' \preccurlyeq_C P$, and $C' \preccurlyeq C$, then $P' \preccurlyeq_{C'} P$.
3. Refinement in a universal context corresponds to regular refinement: $P' \preccurlyeq_{\Sigma^*} P \Leftrightarrow P' \preccurlyeq P$.
4. Refinement in a silent context corresponds to refinement of closed processes: $P' \preccurlyeq_{\{\epsilon\}} P \Leftrightarrow (P' \cap \Sigma_e^*) \preccurlyeq (P \cap \Sigma_e^*)$
5. The refinement $P' \mid C \preccurlyeq_{\{\epsilon\}} P \mid C$ does not imply $P' \preccurlyeq_C P$, because the traces of $P' \mid C$ do not indicate what mutual communication takes place. However, the converse always holds.

To avoid circularity in the assume-guarantee reasoning, we conservatively require that the specification processes can always engage in a subset of communication events $\Sigma_r \subset \Sigma_c$ that is sufficiently large, i.e. $\Sigma_r \cup \overline{\Sigma_r} = \Sigma_c$; in our case, we will take care of this requirement by having specification processes accept any message at any time[1]. We use the following definition to formalize this property of processes.

Definition 2.6. *Let P be a process over Σ, and $\Sigma_r \subset \Sigma$ be an event subset. P is called Σ_r-enabled iff $\forall u \in P : \forall \sigma \in \Sigma_r : u\sigma \in P$.*

We now give the two proof rules for compositional refinement. The first rule is simpler, but restricted to two components. The second rule is a generalization suited for induction.

Theorem 2.7. *Let P, P', Q, Q', C be processes over $\Sigma = \Sigma_e \cup \Sigma_c$. Let $\Sigma_r \subset \Sigma_c$ such that $\Sigma_r \cup \overline{\Sigma_r} = \Sigma_c$. Then the following proof rules are sound:*

$$\frac{P' \preccurlyeq_Q P \qquad P, Q \text{ are } \Sigma_r\text{-enabled} \qquad Q' \preccurlyeq_P Q}{P' \mid Q' \preccurlyeq_{\{\epsilon\}} P \mid Q}$$

$$\frac{P' \preccurlyeq_{Q|C} P \qquad P, Q \text{ are } \Sigma_r\text{-enabled} \qquad Q' \preccurlyeq_{P|C} Q}{P' \mid Q' \preccurlyeq_C P \mid Q}$$

For example, consider again the local refinement obligations (3). Suppose that the specification processes P, Q can receive messages at any time. We can then apply the first proof rule to conclude that $P' \mid Q'$ refines $P \mid Q$, if there is no external communication, i.e., there are no other components in the system.

3 Token Coherence

In this section, we introduce a formal specification of the safety substrate of token coherence. This abstract protocol is a generalization of the MOESI token counting rules in Martin's dissertation [20]. We then justify it's use as a specification, by proving that it is coherent, and with it any implementation that refines it.

3.1 Background: Cache Coherence

Cache coherence describes the contract between the memory system and the processor in a shared-memory multiprocessor. It is typically established at the granularity of a cache block. A memory system is cache coherent if for each block, writes are serialized, and reads get the value of the last write.

Definition 3.1. *Let V be the set of values of a fixed cache block, and $v_0 \in V$ the initial value. Let $\Sigma_{rw} = \{rd(v), wr(v) \mid v \in V\}$ be the alphabet of events,*

[1] If this is not true by default, we could extend the specification to generate a special error event if it receives an unexpected message.

describing accesses to the block by some processor. Then the coherent traces of the system are given by the following regular language over Σ_{rw} :

$$Coh = rd(v_0)^* \left(\bigcup_{v \in V} wr(v)\ rd(v)^* \right)^*$$

Token coherence, like many contemporary coherence protocols such as the popular MOESI protocol family [31], provides this strong form of coherence by enforcing a "single writer, multiple reader" policy[2].

3.2 The Abstract Protocol

In our abstract protocol, system components and messages are of the same type and treated completely symmetrically: both are represented by token bags. Token bags are finite multisets (or bags) over some set T of tokens, and may be required to satisfy some additional constraints (well-formedness). The tokens in the bag constitute the state of the component, or the contents of the message.

The state of the entire system is represented as yet another bag that encloses the token bags of the individual components and messages. The sending of a message is modeled as a division, where a bag separates into two bags, dividing its tokens. The receipt of a message, symmetrically, is modeled as a fusion of token bags. Change is expressed by local reactions: tokens within a bag can be consumed, produced or modified according to rewrite rules.

We give two preliminary definitions before proceeding to the definition of the abstract protocol.

Definition 3.2 (Multisets). *Let T be a set. Two words $u, v \in T^*$ are equivalent if one is a permutation of the other. The induced equivalence classes $\{[u] \mid u \in T^*\}$ are called finite multisets over T, or T-bags. Multiset union is defined as concatenation $[u] \smile [v] \doteq [uv]$. The set of all T-bags is denoted $\mathcal{M}(T)$. For $x \in \mathcal{M}(T)$, let $|x|$ denote the set of elements of T that occur in x.*

For example, for any $t_1, t_2 \in T$, all of the following denote the same T-bag: $[\ t_1^2\ t_2\ ] = [\ t_1\ t_1\ t_2\ ] = \{t_1t_1t_2, t_1t_2t_1, t_2t_1t_1\}$. The exponent is a convenient notation for repeated symbols, and often used with regular languages.

Definition 3.3 (Token Transition System). *A TTS is a tuple (T, B, I, Σ_e, W) where T is a set of tokens, $B \subset \mathcal{M}(T)$ defines the set of well-formed T-bags, $I \in \mathcal{M}(B)$ is the initial configuration, Σ_e is a set of local events, and $W \subset \Sigma_e \times \mathcal{M}(T) \times 2^T \times \mathcal{M}(T)$ is a set of rewrite rules.*

A rewrite rule $(a, x, H, y) \in W$ is denoted a: $x \underset{H}{\Longrightarrow} y$. It describes a reaction labeled a that can occur whenever all the tokens in x are together in a bag, and

[2] We are considering only the interface between the memory system and the processor here. Independently, the contract between the processor and the programmer may use weaker forms of coherence that involve temporal reordering of events, as specified by the memory model.

the bag does not contain any of the inhibiting tokens listed in H. When the reaction fires, the tokens x are replaced by the tokens y. If H is empty, we omit it from the notation.

A TTS defines a process over the alphabet $\Sigma = \Sigma_e \cup \Sigma_c$, with $\Sigma_c = \{snd(b), rcv(b) \mid b \in B\}$, with the traces $\{u \in \Sigma^* \mid \exists C \in \mathcal{M}(B) : I \xrightarrow{u} C\}$, where we define the transition relation $C \xrightarrow{u} C'$ with the inference rules[3] below.

$$\frac{}{C \xrightarrow{\epsilon} C}\ (\text{STUTTER}) \qquad \frac{C \xrightarrow{u} C' \qquad C' \xrightarrow{v} C''}{C \xrightarrow{uv} C''}\ (\text{TRANS})$$

$$\frac{x \smile y \in B}{[\,C\ x\ y\,] \xrightarrow{\epsilon} [\,C\ x \smile y\,]}\ (\text{FUSION}) \qquad \frac{}{[\,C\ x \smile y\,] \xrightarrow{\epsilon} [\,C\ x\ y\,]}\ (\text{DIVISION})$$

$$\frac{a\colon\ x \underset{H}{\Longrightarrow} y \qquad |z| \cap H = \emptyset \qquad y \smile z \in B}{[\,C\ x \smile z\,] \xrightarrow{a} [\,C\ y \smile z\,]}\ (\text{REACTION})$$

$$\frac{}{[\,C\ x\,] \xrightarrow{snd(x)} [\,C\,]}\ (\text{SEND}) \qquad \frac{}{[\,C\,] \xrightarrow{rcv(x)} [\,C\ x\,]}\ (\text{RECEIVE})$$

Token transition systems have a feel of concurrency much like a biological system where reactive substances are contained in cells that can undergo fusion and division. Chemical abstract machines [6] capture the same idea (with molecules, membranes, and solutions instead of tokens, bags, and configurations), but are also different in many ways (for example, they do not have fusion or division).

Definition 3.4 (The abstract protocol). *The safety substrate T_m (where m is the number of tokens, a fixed parameter) is a TTS (T, B, I, Σ_e, W) where*

- *T contains the following tokens:*
  R is a regular token as used by token coherence.
  $O(s)$ is a owner token in one of two states $s \in \{C, D\}$ (clean or dirty).
  $D(v)$ is an instance of the data, with value $v \in V$.
  $M(v)$ is a memory cell containing the value $v \in V$.
- *B is defined by imposing two conditions on a token bag $x \in \mathcal{M}(T)$:*
  - *if x contains data $D(v)$, then it must contain at least one regular token R or an owner token $O(s)$.*
  - *if x contains a dirty owner token $O(D)$, then it must contain data $D(v)$.*
- $I \doteq [\,[\,R^{m-1}\ O(C)\ M(v_0)\,]\,]$.
- $\Sigma_e \doteq \{rd(v), wr(v), memread, memwrite, copy, drop \mid v \in V\}$.
- *W consists of the rewrite rules shown in Fig. 1.*

Fig. 2 shows an example trajectory of the abstract protocol. Next, we explain the reaction rules and their interaction in some more detail.

[3] The variables in the rule templates range over the following domains: $u, v, w \in \Sigma^*$, $x, y, z \in B$, and $C, C', C'' \in \mathcal{M}(B)$. Furthermore, as a syntactic shortcut, we allow C, C', C'' to match several positions in a multiset of token bags: for example, $[\,C\ z\,]$ can match $[\,x\ y\ z\,]$ by setting $C = [\,x\ y\,]$.

Table 1. The reaction rules of the abstract protocol.

rd(v):	$[\,D(v)\,]$	$\Longrightarrow$	$[\,D(v)\,]$
wr(w):	$[\,R^{m-1}\;O(s)\;D(v)\,]$	$\underset{\{D(v)\}}{\Longrightarrow}$	$[\,R^{m-1}\;O(D)\;D(w)\,]$
memread:	$[\,M(v)\;O(C)\,]$	$\Longrightarrow$	$[\,M(v)\;O(C)\;D(v)\,]$
memwrite:	$[\,M(v)\;O(D)\;D(w)\,]$	$\Longrightarrow$	$[\,M(w)\;O(C)\;D(w)\,]$
copy:	$[\,D(v)\,]$	$\Longrightarrow$	$[\,D(v)\;D(v)\,]$
drop:	$[\,D(v)\,]$	$\Longrightarrow$	$[\;]$

Table 2. A short example trajectory of the abstract protocol, representing a system with a memory D and two caches C_1 and C_2. For clarification, token bags carry subscripts indicating the component that they represent. Those subscripts are *not* part of the abstract protocol.

Description	System trajectory
initial state	$[\,[\,M(v_0)\;O(C)\;R^{m-1}\,]_D\;[\;]_{C_1}\;[\;]_{C_2}\,]$
C_1 requests M	(requests are abstracted away)
D responds	
— read memory data	$\xrightarrow{memread}[\,[\,M(v_0)\;D(v_0)\;O(C)\;R^{m-1}\,]_D\;[\;]_{C_1}\;[\;]_{C_2}\,]$
— send data w/ tokens	$\xrightarrow{\epsilon}[\,[\;M(v_0)\,]_D\;[\,D(v_0)\;O(C)\;R^{m-1}\,]\;[\;]_{C_1}\;[\;]_{C_2}\,]$
C_1 receives response	$\xrightarrow{\epsilon}[\,[\,M(v_0)\,]_D\;[\,D(v_0)\;O(C)\;R^{m-1}\,]_{C_1}\;[\;]_{C_2}\,]$
C_1 writes value v_1	$\xrightarrow{wr(v_1)}[\,[\,M(v_0)\,]_D\;[\,D(v_1)\;O(D)\;R^{m-1}\,]_{C_1}\;[\;]_{C_2}\,]$
C_2 requests S	(requests are abstracted away)
C_1 responds	
— copy data	$\xrightarrow{copy}[\,[\,M(v_0)\,]_D\;[\,D(v_1)\;D(v_1)\;O(D)\;R^{m-1}\,]_{C_1}\;[\;]_{C_2}\,]$
— send data w/ token	$\xrightarrow{\epsilon}[\,[\,M(v_0)\,]_D\;[\,D(v_1)\;O(D)\;R^{m-2}\,]_{C_1}\;[\,D(v_1)\;R\,]\;[\;]_{C_2}\,]$

$rd(v)$ reads a value from a data instance (it can be applied at any time, and does not modify the state). $wr(w)$ modifies a data token, and can only be applied if all m tokens (one owner token and $m-1$ regular tokens) are present, and no other data copies are in the same bag (which guarantees that the data token being modified is the only one in the system).

To guarantee proper writebacks of modified data, a special owner token is used. The owner token records the clean/dirty state, i.e. whether the memory value is stale. When modifying data, the owner token is set to dirty. When the memory writes back the data (*memwrite*), the owner token is cleaned. *memread* loads data from the memory only if there is a clean owner token, and thereby avoids reading stale data.

The rules *copy* and *drop* imply that data instances $D(v)$ can be freely copied or destroyed, subject only to the restriction enforced by B that all bags are well-formed – for example, whoever has the dirty owner token must keep at least one data instance.

We can now prove that the abstract protocol is coherent.

Theorem 3.5. *The closed system $T_m \cap \Sigma_e^*$ is coherent:*

$$(T_m \cap \Sigma_e^*) \uparrow \Sigma_{rw} \subset Coh$$

To prove this, verify that (1) all of the following invariants hold in the initial state I and (2) prove (by induction on derivations) that if the invariants hold for a state C, they hold for any state C' such that $C \xrightarrow{u} C'$ for some $u \in \Sigma_e^*$.

1. The number of regular tokens R in the system is $m - 1$.
2. There is always exactly one owner token $O(s)$.
3. There is always exactly one memory cell $M(v)$.
4. All data instances $D(v)$ have the same values.
5. If the owner token is clean, any data instances present have the same value as the memory cell.
6. If there is a data token, it contains the value of the last write. Otherwise, the memory does.

Together, these invariants guarantee that all data instances $D(v)$ are always up-to-date; therefore, reads get the correct value which implies coherence.

All state is modeled by tokens, and there is no distinction between components and messages. This symmetry points out interesting design directions. For example, we consider the memory cell $M(v)$ to be stationary. However, the formal token rules do not impose this restriction and and could be used as an implementation guideline for a system with home migration.

4 Implementation

In this section, we describe how we verified the safety of a detailed implementation of token coherence for an arbitrary number of caches. We describe how we used compositional verification to deal with the parametric character, and how we employed abstraction to handle the fine level of detail. We conclude with a list of discovered bugs.

4.1 The Protocol Implementation

The protocol implementation was developed by Martin et al. for architecture research on token coherence [20], and was extensively simulated prior to this

	Request-Excl.	Request-Shared	Lockdown	Unlockdown	Data Owner	Data Shared	Ack	Ack Owner	Exclusive Compl.	Owned Compl.	Shared Compl.
O	d b j /NO	d j /NO	dd l /L	▨	▨	f q k	f q k	▨	v i	w i	x i
NO	a b j	c j	a l / L	▨	m f p k /O	f q k	f q k	n f p k /O	v i	w i	x i
L	j	j	l	l / NO	r k	r k	r k	s k	v i	w i	x i

Fig. 1. The SLICC table for the memory controller. Rows show controller states, columns show events, and cells show transitions. For example, consider the upper left box. It states that if a Request-Exclusive message arrives while the controller is in state O, the actions d, b and j are executed in sequence, and the controller transitions to the NO state. Shaded cells indicate that an event is not expected to occur in the given state.

work. It consists of finite state machines (FSM) for the cache and memory controllers, augmented with message passing capabilities. The FSMs are specified using the domain-specific language SLICC (Specification Language for Implementing Cache Coherence) developed by Martin et al.

The FSMs include all necessary transient states that arise due to the asynchronous nature of the protocol. The memory and cache controller amount to 600 and 1800 lines of SLICC code, respectively, a scale on which purely manual analysis methods are impractical, in particular because these low-level specifications are usually changed over time.

The SLICC compiler generates (1) executables for the simulation environment and (2) summary tables containing the control states, events and transitions in a human-readable table format[4].

Fig. 1 shows the summary table for the memory controller, with its 3 states and 11 events. Note that some parts of the state, such as the number of tokens, or the actual data values, are stored in variables that are not visible in the summary table.

Due to lack of space, we can not reproduce the summary table for the cache controller (17 states and 20 events), and we can not explain further the meaning of the states and events. The complete SLICC code and interactive HTML-tables are online [21], along with implementations of three other cache coherence protocols.

4.2 Parametric Compositional Refinement Proof

Consider the system S'_n consisting of n caches C', a directory controller D' (which is attached to the memory, and sometimes called memory controller), and

[4] More about the table format can be found in Sorin et al. [30].

a interconnection network N'. We consistently use primes for implementation processes to distinguish them from specification processes:

$$S'_n \doteq \underbrace{C' \mid C' \mid \cdots \mid C'}_{n} \mid N' \mid D' \tag{4}$$

In the beginning, the memory holds all tokens. We define local specification processes as token transition systems:

$$\begin{aligned} D &\doteq T_m = (T, B, I, \Sigma_e, W) \\ C &\doteq (T, B, [\,[\ \ ]\,], \Sigma_e, W) \\ N &\doteq (T, B, [\,[\ \ ]\,], \Sigma_e, W) \end{aligned}$$

Since a token transition system already models all possible distributions of the state, no new behavior arises when it is composed:

$$C \mid D = D \qquad C \mid C = C$$

We now state the central result which (together with Theorem 3.5) allows us to verify the implementation components D', C' and N' individually, each within an abstracted context rather than a fully instantiated system.

Theorem 4.1. *If the implementation processes satisfy the local refinement obligations*

$$D' \preccurlyeq_C D \qquad C' \preccurlyeq_D C \qquad N' \preccurlyeq_D C$$

then for all $n \in \mathbb{N}$, we have $S'_n \preccurlyeq_{\{\epsilon\}} T_m$, i.e., the system refines the formal token coherence protocol.

The proof uses induction and the proof rules (Theorem 2.7).

4.3 Discharging the Obligations

To discharge the remaining obligations, we used manual translation, abstraction, and annotation, and the explicit model checker Murφ [12, 11]. The following steps give an overview of the method.

1. *Obtain models D', C' for the memory and cache controller implementations.* This step involves translating the SLICC code to Murφ, instrumenting it with the read/write events relevant for coherence, and abstracting both the state space and the message format. Fig. 2 shows snippets of translated code. The SLICC instructions that fell prey to the abstraction are in `slanted` face. For example, only a single cache block is modeled, therefore the code dealing with addresses is abstracted away. Also, message source and destination fields are irrelevant due to the deep symmetry of formal token coherence. Furthermore, two data values are sufficient[5].

[5] Restricting the set of values is justified by the *data-independence* [32], which implies that we can freely substitute values in the traces.

2. *Obtain good encodings for the specification/environment processes D, C.* We can take advantage (1) of the global system invariants established earlier and (2) of the fact that fusion and division are not observable. For example, the flattening map $[\ b_1\ b_2\ \dots\ b_k\ ] \mapsto b_1 \smile b_2 \dots \smile b_k$ provides a canonical representative state. This means that *a single T-bag, rather than a multiset of T-bags, is sufficient to model the context.* The models we obtain this way are compact and contribute much to the state-space economy of our approach.
3. *Annotate the transitions of the implementation with matching specification transitions, and provide refinement maps.* For each transition of the implementation process, the annotations specify a sequence of transitions of the specification process. Fig. 2 shows such annotations in uppercase. The refinement maps are functions that map a controller state to its corresponding token bag.
4. *Run the model checker Murφ separately for the two relevant obligations*[6] *$D' \preccurlyeq_C D$ and $C' \preccurlyeq_D C$.*

   Proposition 4.3 listed below describes how the contextual refinement is discharged. The state enumeration performed by the model checker effectively constructs and verifies the relation R, which describes the reachable states of the implementation process I within the abstract context C. The annotations provided by the user eliminate the need for existential quantification. The model checker also validates the assertions present in the implementation code.

Definition 4.2. *For a labeled transition system $(Q, q0, \Sigma \cup \{\epsilon\}, \delta)$, states $q_1, q_2 \in Q$ and a word $v \in \Sigma^*$ we define: $q \overset{v}{\Longrightarrow} q'$ iff there exists a $k \geq 0$ and a sequence of transitions $q_0 \xrightarrow{v_1} q_1 \xrightarrow{v_2} \dots \xrightarrow{v_k} q_k$ such that $q_0 = q$, $q_k = q'$ and $v_1 v_2 \dots v_k = v$ (where $v_1 v_2 \dots v_k = \epsilon$ for $k = 0$).*

Proposition 4.3. *Let I, S and C be processes defined by the trace sets of the labeled transition systems $L_i \doteq (Q_i, q0_i, \Sigma \cup \{\epsilon\}, \delta_i)$ with $i \in \{I, S, C\}$. Let $\phi : Q_I \to Q_S$ be a function (the refinement map). If $R \subset Q_I \times Q_C$ is a relation with the properties (R1)–(R4) listed below, then $I \preccurlyeq_C S$.*

(R1) $(q0_I, q0_C) \in R$, *and* $\phi(q0_I) = q0_S$

(R2) *If* $(q_I, q_C) \in R$ *and* $q_C \xrightarrow{u} q'_C$ *for some* $u \in \Sigma_e \cup \{\epsilon\}$, *then* $(q_I, q'_C) \in R$.

(R3) *If* $(q_I, q_C) \in R$ *and* $q_I \xrightarrow{u} q'_I$ *for some* $u \in \Sigma_e \cup \{\epsilon\}$, *then* $(q'_I, q_C) \in R$ *and* $\phi(q_I) \overset{u}{\Longrightarrow} \phi(q'_I)$.

(R4) *If* $(q_I, q_C) \in R$ *and* $q_I \xrightarrow{\sigma} q'_I$ *and* $q_C \xrightarrow{\overline{\sigma}} q'_C$ *for some* $\sigma \in \Sigma_c$, *then* $(q'_I, q'_C) \in R$ *and* $\phi(q_I) \overset{\sigma}{\Longrightarrow} \phi(q'_I)$.

The full Murφ code is available online [7].

[6] Theorem 4.1 lists three obligations, but we skip $N' \preccurlyeq_D C$ because it reduces to checking the reliablity of the network, which is trivial at the given abstraction level.

```
rule "get Request-Excl in O state"
  (I_DirectoryState = state_O)
==>
begin
    d_sendDataWithAllTokens();
    I_DirectoryState := state_NO;
endrule;

procedure d_sendDataWithAllTokens();
var
  out_msg: I_message;
begin
  out_msg.RType := DATA_OWNER;
  if !(I_Tokens > 0) then
    error "d: assertion failed. ";
  endif;
  out_msg.Tokens := I_Tokens;
  out_msg.DataBlk := I_DataBlk;
  out_msg.Dirty := false;
  I_Tokens := 0;
  EVENT_MEMLOAD();
  EVENT_SEND(out_msg);
  EVENT_DROP();
end;
```

```
transition(O, RequestExcl, NO) {
  d_sendDataWithAllTokens;
  b_forwardToSharers;
  j_popIncomingRequestQueue;
}

action(d_sendDataWithAllTokens, "d") {
  peek(requestNetwork_in, RequestMsg) {
    enqueue(responseNetwork_out, ResponseMsg) {
      out_msg.Address := address;
      out_msg.Type := CoherenceResponseType: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[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[address].Tokens := 0;
}
```

Fig. 2. The murphi code (top) is obtained from the SLICC code (bottom).

4.4 Results

The translation required about two days of work. This estimate assumes familiarity with token coherence, and some knowledge of the implementation. We found several bugs of varying severity, all of which were missed by prior random simulation tests similar to those described by Wood et. al. [33]. Seven changes were needed to eliminate all failures (not counting mistakes in the verification model):

1. The implementation included assertions that do not hold in the general system. Although they were mostly accompanied by a disclaimer like "remove this for general implementation", the latter was missing in one case.
2. The implementation was incorrect for the case where a node has only one token remaining and answers a Request-Shared. This situation was not encountered by simulation, probably because the number of tokens always exceeded the number of simulated nodes. We fixed the implementation, which involved adding another state to the finite state control.
3. Persistent-Request-Shared messages (which are issued if the regular Request-Shared is not answered within a timeout period) suffered from the same problem, and we applied the same fix.
4. The implementation copied the dirty bit from incoming messages even if they did not contain the owner token. Although this does not compromise coherence, it can lead to suboptimal performance due to superfluous writebacks. This performance bug would have gone undetected had we only checked for coherence, rather than for refinement of the abstract protocol.
5. After fixing bug 4, a previously masked bug surfaced: the dirty bit was no longer being updated if a node with data received a dirty owner token.
6. Two shaded boxes (i.e. transitions that are specified to be unreachable) were actually reachable. This turned out to be yet another instance of the same kind of problem as in bug 2.
7. Finally, another (last) instance of bug 2 was found and fixed.

As expected, the compositional approach heavily reduced the number of searched states. This kept computational requirements low, in particular considering that the results are valid for an arbitrary number of caches. The measurements in Fig. 3 were carried out on a 300MHz Pentium III ThinkPad.

5 Conclusions and Future Work

We make three main contributions. First, we formally verified the safety of a system-level implementation of token coherence, for an arbitrary number of

# tokens	component	# states	# transitions	time
4	memory controller	92	1692	0.3s
8	memory controller	188	5876	0.6s
32	memory controller	764	83396	7.49s
4	cache controller	700	23454	1.4s
8	cache controller	1308	76446	4.6s
32	cache controller	4956	1012638	65.2s

Fig. 3. Computational requirements for the model checking.

caches. Second, we developed a general and formal specification of the safety substrate of token coherence, and prove its correctness. Third, we demonstrated that token coherence's "design for verification" approach indeed facilitates the verification as claimed.

Future work may address the following open issues. First, the methodology does not currently address liveness. Second, other protocols or concurrent computations may benefit from the high-level abstraction expressed by token transition systems, and offer opportunities for compositional refinement along the same lines. Third, much room for automation remains: for example, we could attempt to integrate theorem provers with the SLICC compiler.

References

1. M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–535, May 1995.
2. R. Alur and T. A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 207. IEEE Computer Society, 1996.
3. R. Alur and B. Wang. Verifying network protocol implementations by symbolic refinement checking. In *Proceedings of the 13th International Conference on Computer-Aided Verification*, 2001.
4. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck. Parameterized verification with automatically computed inductive assertions. In *Computer Aided Verification*, pages 221–234, 2001.
5. Arvind and X. W. Shen. Using term rewriting systems to design and verify processors. *IEEE Micro*, 19(3):36–46, /1999.
6. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.
7. S. Burckhardt et al. Verifying safety of a token coherence implementation by parametric compositional refinement: Extended version. `http://www.seas.upenn.edu/~sburckha/token/`, 2004.
8. K. M. Chandy and J. Misra. *Parallel program design: a foundation.* Addison-Wesley Longman Publishing Co., Inc., 1988.
9. E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. *ACM Trans. Program. Lang. Syst.*, 19(5):726–750, 1997.
10. G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Computer Aided Verification*, pages 53–68, 2000.
11. D. L. Dill. The murphi verification system. In *Proceedings of the 8th International Conference on Computer Aided Verification*, pages 390–393. Springer-Verlag, 1996.
12. D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design*, pages 522–525, 1992.
13. E. A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *Conference on Automated Deduction*, pages 236–254, 2000.
14. S. M. German. Formal design of cache memory protocols in IBM. *Formal Methods in System Design*, 22(2):133–141, 2003.
15. S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.

16. G. J. Holzmann. Algorithms for automated protocol verification. *AT&T Tech. J.*, Jan./Feb. 1990.
17. Y. Kesten and A. Pnueli. Control and data abstraction: The cornerstones of practical formal verification. *International Journal on Software Tools for Technology Transfer*, 2(4):328–342, 2000.
18. R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.
19. R. P. Kurshan and K. McMillan. A structural induction theorem for processes. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 239–247. ACM Press, 1989.
20. M. M. K. Martin. *Token Coherence*. PhD thesis, University of Wisconsin-Madison, 2003.
21. M. M. K. Martin et al. Protocol specifications and tables for four comparable MOESI coherence protocols: Token coherence, directory, snooping, and hammer. `http://www.cs.wisc.edu/multifacet/theses/milo_martin_phd/`, 2003.
22. M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: decoupling performance and correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 182–193. ACM Press, 2003.
23. K. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–51, Tokyo, Japan, 1991.
24. K. L. McMillan. A compositional rule for hardware design refinement. In *Proceedings of the 9th International Conference on Computer-Aided Verification*, pages 24–35, June 1997.
25. K. L. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 110–121. Springer, 1998.
26. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
27. S. Park and D. L. Dill. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 288–296. ACM Press, 1996.
28. F. Pong and M. Dubois. Verification techniques for cache coherence protocols. *ACM Computing Surveys*, 29(1):82–126, 1997.
29. A. Ponse, S. A. Smolka, and J. A. Bergstra. *Handbook of Process Algebra*. Elsevier Science Inc., 2001.
30. D. J. Sorin, M. Plakal, A. E. Condon, M. D. Hill, M. M. K. Martin, and D. A. Wood. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):556–578, 2002.
31. P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414–423. IEEE Computer Society Press, 1986.
32. P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 184–193. ACM Press, 1986.
33. D. A. Wood, G. A. Gibson, and R. H. Katz. Verifying a multiprocessor cache controller using random test generation. *IEEE Design & Test*, 7(4):13–25, 1990.