A Knowledge Based Analysis of Cache Coherence[⋆]

Kai Baukus[1] and Ron van der Meyden[2]

School of Computer Science and Engineering
University of New South Wales[{1,2}], and
National ICT Australia[2]
{*kbaukus,meyden*}*@cse.unsw.edu.au*

Abstract. This paper presents a case study of the application of the knowledge-based approach to concurrent systems specification, design and verification. A highly abstract solution to the cache coherence problem is first presented, in the form of a knowledge-based program, that formalises the intuitions underlying the MOESI [Sweazey & Smith, 1986] characterisation of cache coherency protocols. It is shown that any concrete implementation of this knowledge-based program, which relates a cache's actions to its knowledge about the status of other caches, is a correct solution of the cache coherence problem. Three existing protocols in the MOESI class are shown to be such implementations. The knowledge-based characterisation furthermore raises the question of whether these protocols are optimal in their use of information available to the caches. This question is investigated using by the model checker MCK, which is able to verify specifications in the logic of knowledge and time.

1 Introduction

Reasoning about knowledge [FHMV95] provides an approach to concurrent system specification, development, and verification that focuses on information flow in the system and how information relates to action. The approach promises a number of advantages: a high level of abstraction in specification, ease of verification and a methodology for the development of protocols that make optimal use of information. This paper presents a case study of the approach in a recently developed formulation, and uses for part of the effort a new tool that enables model checking specifications in the logic of knowledge and time [GM04].

The focus of the case study is cache coherence protocols for multi-processor systems. Numerous cache coherency solutions have been proposed [AB86], and the area has been a popular target for the application of model checking systems [CGH$^+$95]. Often, the literature on verification of such protocols has concentrated on verifying specific concrete protocols rather than a more abstract

⋆ Work supported by an Australian Research Council Discovery Grant. National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the ARC.

design level. One exception is the work of Sweazey and Smith [SS86], who have given an abstract characterisation of a class of cache coherency protocols in order to show that these protocols can be implemented on the FutureBus. They show that several of the protocols in the literature can be implemented as concrete instances of their general scheme. However, their presentation is informal and does not provide a rigorous relationship between their intuitions and the concrete transition tables by which they present the concrete protocols.

We show in this paper that the knowledge-based approach provides the means to formalise the abstract intuitions of Sweazey & Smith, enabling their relationship to the concrete protocols to be precisely defined and justified. We describe how this implementation relationship can be verified. Our characterisation clarifies some aspects of Sweazey and Smith's intuitions. Moreover, our formulation raises the interesting question of whether these concrete protocols make optimal use of the information generated in running them. We investigate this question using a newly developed tool, MCK, a model checker for the logic of knowledge and time [GM04]. In some cases, we find that the protocols do make optimal use of information, but we also identify ways in which they do not. In some cases, this is because our analysis takes into account aspects not considered by the protocol designers. In one case, however, the Synapse protocol, this is much less clear, and our analysis points to a way to optimize the protocol.

2 Framework: Syntax and Semantics

In this section we recall some definitions from the literature on reasoning about knowledge, and sketch the process notation we use in the paper. For more detailed exposition and motivation on reasoning about knowledge, we refer the reader to [FHMV95].

2.1 Reasoning about Knowledge

Consider a concurrent system comprised of a set of processes. Each process is associated with a set of variables that we call its *local variables*. A *global state* is an assignment of values to each of the local variables of each process. Executing the processes from some initial state produces a *run*, a mapping from natural numbers to global states. If r is a run and $n \in \mathbf{N}$ a time, then $r(n)$ denotes the global state at time n. A *point* is a pair (r, n) consisting of a run r and a time $n \in \mathbf{N}$.

A *system* is a set of runs. If *Prop* is a set of atomic propositions, an *interpretation* of *Prop* in a system $\mathcal{R}$ is a mapping π associating each point in $\mathcal{R}$ with the set of $p \in \mathit{Prop}$ it satisfies. An *interpreted system* is a pair $(\mathcal{R}, \pi)$ where $\mathcal{R}$ is a system and π is an interpretation for $\mathcal{R}$.

The logic of knowledge contains a modal operator K_i, intuitively representing the expression "process i knows that", as well as the usual boolean operators. If ϕ is a formula then so is $K_i\phi$, where i is a process. The semantics of the logic of knowledge is usually defined in interpreted systems as follows. First, each process

i is associated with a *local state* at each point (r, n) — this local state is denoted by $r_i(n)$, and intuitively captures all the information available to the process for determining what it knows at that time. Two points (r, n) and (r', n') are said to be *indistinguishable* to process i, written $(r, n) \sim_i (r', n')$, if $r_i(n) = r_i(n')$. Formulas are given semantics by means of a relation of satisfaction of a formula ϕ at a point (r, n) in an interpreted system $\mathcal{I}$, written $\mathcal{I}, (r, n) \models \phi$. If $\mathcal{I} = (\mathcal{R}, \pi)$, this relation is defined inductively by

1. $\mathcal{I}, (r, n) \models p$, for $p \in \mathit{Prop}$, if $p \in \pi(r, n)$,
2. $\mathcal{I}, (r, n) \models K_i\phi$, if $\mathcal{I}, (r', n') \models \phi$ for all points $(r', n') \sim_i (r, n)$ in $\mathcal{R}$,

and the obvious clauses for the boolean operators. Intuitively, this definition says that process i knows ϕ if ϕ holds at all points it is unable to distinguish from the current point. We write $\mathcal{I} \models \phi$ if ϕ is valid in the structure $\mathcal{I}$, i.e, $\mathcal{I}, (r, n) \models \phi$ for all points (r, n) in $\mathcal{I}$.

The local state can be defined in a number of ways. One, which we call the *observational* interpretation of knowledge, defines $r_i(n)$ to be the restriction $r(n) \upharpoonright \mathit{Vars}_i$ of the global state $r(n)$ to the set Vars_i of local variables of process i. (We could furthermore restrict to a smaller set of variables if the question of what information is carried by that smaller set of variables is of interest). Another, the *asynchronous perfect recall* semantics inductively defines the local state as a sequence of assignments as follows: $r_i(0) = r(0) \upharpoonright \mathit{Vars}_i$ and $r_i(n+1) = r_i(n)$ if $r(n) \upharpoonright \mathit{Vars}_i = r(n+1) \upharpoonright \mathit{Vars}_i$ and $r_i(n+1) = r_i(n) \cdot (r(n+1) \upharpoonright \mathit{Vars}_i)$ otherwise. Intuitively, in this interpretation, process i remembers its complete history of observations except that due to asynchrony it is unable to distinguish stuttering equivalent sequences of observations.

The semantics above has formed the basis of much of the literature on the knowledge-based approach, but has been generalised [EMM98] in order to provide a framework that provides greater flexibility. The generalisation uses the notion of *sound local propositions.* We formulate this here as follows. Introduce a set of atomic propositions of the form $\mathbf{k}_i\phi$, where ϕ is a formula, with the semantics of these atomic propositions given by the *first* of the two clauses above. To retain some of the key aspects of the usual semantics of the knowledge operator, we impose the following constraints:

Locality: If $r_i(n) = r'_i(n')$, then $\mathbf{k}_i\phi \in \pi(r, n)$ iff $\mathbf{k}_i\phi \in \pi(r', n')$.
Soundness: $\mathcal{I} \models \mathbf{k}_i\phi \Rightarrow \phi$

Intuitively, locality says that the proposition $\mathbf{k}_i\phi$ is a function of process i's local state. Sound local propositions generalise the logic of knowledge: it can be seen that soundness and locality implies that $\mathcal{I} \models \mathbf{k}_i\phi \Rightarrow K_i\phi$, so if an interpretation of the extended set of atomic propositions also satisfies

Completeness: $\mathcal{I} \models K_i\phi \Rightarrow \mathbf{k}_i\phi$

then we have $\mathcal{I} \models \mathbf{k}_i\phi \Leftrightarrow K_i\phi$.

It has been noted [EMM98] that much of the work of knowledge formulas in the knowledge-based approach can be done with sound local propositions, and a formal refinement calculus embodying this generalisation of the approach is under development [EMM01,Eng02]. We use some ideas from this calculus in the present paper.

2.2 Process Notation

We will describe systems at a number of levels of abstraction. Most concretely, we use a process algebraic notation to describe the implementation level. Variables may be updated by assignments. We use standard nondeterministic programming constructs, and a parallelism operator $||$, the semantic details of which we omit for space reasons. The overall semantics is non-deterministic and asynchronous in the sense that at any time, up to one of the possible events is nondeterministically selected for execution. The choice is fair, in the sense that a continuously enabled event is eventually selected.

The communication mechanism in the notation is by a synchronous handshake between some set of processes. This may be a two-way handshake passing a value v from process A to B, which we represent by having A perform the event $B!v$ and having B perform $A?x$, writing the value v into $B's$ variable x. This handshake event is treated as causing a single transition in the global state of the system. In order to model communication on the bus, we also need a handshake involving more than two processes. Such handshake events also cause a single transition in the system state, and may change any of the local variables of the participating processes. We define the semantics of such handshakes by providing sequential code used by each of the participating processes to update its local variables: this code is interpreted to run atomically.

3 The Cache Coherence Problem

We study cache coherence protocols in the standard setting of several processors that access their data indirectly through caches, connected via a bus to each other and the main memory. The situation is depicted in Figure 1. The processors access memory locations via `read` and `write` requests. In a system without caches, the requests would be queued to access the main bus sequentially. Caches are fast memories introduced to speed up access to frequently used locations. The direct communication between processor and cache is much faster than the main bus serving all parties connected to it. When the requested data value exists as a *valid* copy in the cache, the cache answers the request immediately. Otherwise, the cache has to request the corresponding location via the main bus. A *cache coherence protocol* is a protocol run by the caches on the main bus in order to guarantee consistency between the copies of data values that they maintain.

A variety of different cache coherence protocols have been designed, that have been placed into a general framework by Sweazey and Smith [SS86], who identify the following attributes as being of relevance in the family of cache-coherence protocols they consider:

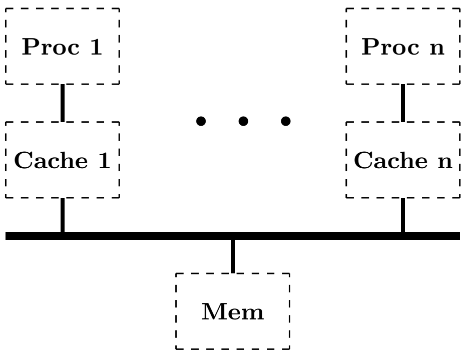


Fig. 1. Caches connected via a bus

1. **Ownership/Modification:** The owner of a memory line is responsible for the value of that line. When the owner is a cache, it is the owner just when it holds a modified value of the memory line.
2. **Exclusiveness:** Copies of the memory line may be held simultaneously by a number of caches. If it is held by just one, that cache holds it exclusively. A non-exclusive line is said to be *shared*.
3. **Validity:** The copy held by a cache may be valid (equal to the copy of the owner) or invalid.

These three attributes give 8 possible states, but ownership and exclusivity are not of interest for invalid memory lines, so [SS86] aggregate the four invalid cases into one, leaving five states, which they call M (exclusive modified), O (shared modified), S (shared unmodified), E (exclusive unmodified) and I (invalid). These states are updated as a result of bus communications, where in addition to the memory value, six different boolean signals are communicated across the bus: each is associated with an intuitive meaning, and the possible patterns of communication of these signals in each of the MOESI states and the resulting state updates are described in a set of tables. The tables allow for some choice of action in a number of states, and [SS86] show that a number of existing cache-coherency protocols can be modelled by appropriate choice of a subset of the MOESI states and a choice of the associated actions.

The MOESI states are not given a formal specification in [SS86], and the associated tables are justified only by some text that provides some informal motivation. We propose here a higher level view of the MOESI states and the associated protocols, by describing a knowledge-based program that captures the underlying reasons for correctness of the MOESI protocols. This will have the benefit not just of providing a clearer exposition of the protocol, but also of opening the way for an investigation of whether the existing protocols make optimal use of the information being communicated across the bus (something we investigate by model checking some particular cases) and permitting the exploration of alternative implementations that make greater use of the pattern of communication across the bus.

4 Knowledge-based Specification

We now give an abstract, knowledge-based specification of cache coherence. We consider a single memory line, which we model using a single bit. The abstract behaviour of the processors is to non-deterministically choose between modifying the value arbitrarily and reading/writing of the value back to memory, represented by the following term:

```
$P_i \equiv$ do $value := 0$
       [] $value := 1$
       [] $\mathcal{C}_i!\,\mathbf{write}(value);\ \mathcal{C}_i?\,\mathbf{ack}$
       [] $\mathcal{C}_i!\,\mathbf{read};\ \mathcal{C}_i?\,value$
       od
```

We denote the set of caches by `Cache`. Each cache $i \in$ `Cache` has associated with it the following variables:

1. a variable c_i denoting the value i has for the memory line. This might be $\bot$ if the cache does not have any value registered. It is convenient to denote the main memory by m and the value of the main memory by c_m, but this variable will never take the value $\bot$.
2. a variable $pending_i$ which has as value either the action that process i has requested to be performed (**read** or **write**(v), for $v \in \{0, 1\}$), or the value $\bot$, indicating that all requests have been processed.

At each moment of time, either one of the caches or the main memory is designated as the owner of the memory line; we represent the current owner by the variable o, with domain $\mathtt{Cache} \cup \{m\}$. Exactly how the owner is determined at each moment of time is a matter of implementation detail - we provide some examples of particular implementations later.

A cache's actions in response to its processors read and write requests depend on what the cache knows about the state of the system, in particular, what it knows about the three attributes identified above. We represent this knowledge by means of three atomic propositions: $\mathbf{k}_i(\mathtt{excl}_i)$, $\mathbf{k}_i(o = i)$ and $\mathbf{k}_i(c_i = c_o)$, which we will require to be given a sound interpretation local to cache i (see Section 2). The meaning of the formulas in these proposition names is as follows:

1. We write $\mathtt{excl}_i$ for $c_i \neq \bot \wedge \bigwedge_{j \in \mathtt{Cache} \setminus \{i\}} c_j = \bot$, i.e., cache i is the only cache with a copy of the line; thus $\mathbf{k}_i(\mathtt{excl}_i)$ intuitively says that cache i knows that it has an exclusive copy;
2. $o = i$ says that cache i is the owner, hence $\mathbf{k}_i(o = i)$ intuitively says that cache i knows that it is the owner;
3. $c_i = c_o$ says that cache i's copy of the data value is equal to the owner o's copy, hence $\mathbf{k}_i(c_i = c_o)$ intuitively says that cache i knows that its copy is valid.

We relate these local propositions to the cache's actions by the following specification of the cache's behaviour, in the form of a guarded do loop that runs forever. The clauses of this do loop have been labelled for reference, e.g. the first is labelled "{Get request}". Intuitively, the treatment of the variable $pending_i$ ensures that this variable acts as a flag indicating whether there exists a pending request from the processor. Requests to read are cleared by the cache by executing the {Read Hit} or {Read Miss} clauses and requests to write are cleared by executing the clause {Write Local} or {Write Bus}. The memory line is cleared from the cache by executing the {Copy back} or {Flush} clause. The reason we have two clauses to achieve each effect is that one involves a bus event and the other can be achieved locally.

```
$\mathcal{C}_i \equiv$ do
        {Get request}
        $pending_i = \bot \rightarrow \mathcal{P}_i?pending_i$
    []  {Read Hit}
        $pending_i = \mathbf{read} \wedge \mathbf{k}_i(c_i = c_o)$          $\rightarrow \mathcal{P}_i!\, c_i;\ pending_i := \bot$
    []  {Read Miss}
        $pending_i = \mathbf{read} \wedge \neg\mathbf{k}_i(c_i = c_o)$       $\rightarrow [c_o = X,\ c_o = X \wedge K_i(c_i = c_o)]_B;$
                                                  $\mathcal{P}_i!\, c_i;\ pending_i := \bot$
    []  {Write Local}
        $pending_i = \mathbf{write}(v) \wedge \mathbf{k}_i(\mathtt{excl}_i)$     $\rightarrow [\text{TRUE}, c_i = v \wedge o = i]_L;$
                                                  $\mathcal{P}_i!\,\mathbf{ack};\ pending_i := \bot$
    []  {Write Bus}
        $pending_i = \mathbf{write}(v) \wedge \neg\mathbf{k}_i(\mathtt{excl}_i)$  $\rightarrow [\text{TRUE}, c_o = v \wedge c_i = v]_B;$
                                                  $\mathcal{P}_i!\,\mathbf{ack};\ pending_i := \bot$
    []  {Copy Back}
        $pending = \bot \wedge \mathbf{k}_i(o = i)$               $\rightarrow [c_i = X,\ i \neq o \wedge c_o = X \wedge$
                                                          $pending_i = \bot]_B$
    []  {Flush}
        $pending = \bot \wedge \neg\mathbf{k}_i(o = i)$             $\rightarrow [i \neq o,\ c_i = \bot \wedge pending_i = \bot]_L;$
    []  {Bus Observation}
        $[pending_i = X, pending_i = X]_B$
    od
```

This specification is in the spirit of knowledge-based programs in the sense of [FHMV95,FHMV97], in that it relates the cache's actions to its knowledge. However, it also draws on elements of the knowledge based refinement calculus under development [EMM98,MM00,EMM00,EMM01,Eng02]. One of these elements is the use of sound local propositions $\mathbf{k}_i\phi$ as guards, where the [FHMV95,FHMV97] knowledge-based programs would require use of sound and complete guards $K_i\phi$. As we will show in the next section, soundness of the guards suffices for correctness of the specification. This makes our specification more general. (The astute reader may have noticed that we do use the classical notion of knowledge in the {Read Miss} clause. We comment on the reasons for this later.)

Another difference from the knowledge-based programs of [FHMV95,FHMV97] is that in addition to having concrete actions in the clauses, we also make use of *specification statements* $[\phi, \psi]$. These are pre- and post-condition specifications: they stand for some code, to be determined, that when executed from a state satisfying ϕ terminates in a state satisfying ψ. (The subscripts B and L are explained below.) We give a formal definition below. The variable X is used in the specification statements as an implicitly universally quantified frame variable, which is required to assume the same value in the pre-condition and the post-condition. That is, the specification in this case is required to be satisfied for all values of X.

We place some further constraints on the code implementing the specification statements in the program:

`I1`. We assume that this code does not involve any communication actions between the cache and its processor, so that the only such events in the implementation are those already visible in the program above.
`I2`. Statements of the form $[\phi, \psi]_L$ must be implemented by a program that takes only actions *local* to the cache.
`I3`. Statements of the form $[\phi, \psi]_B$ must be implemented by a single bus event.
`I4`. The variable c_i cannot change from having value $\perp$ to having a value not equal to $\perp$ without a bus event occurring.
`I5`. During the execution of $[i \neq o,\ c_i = \perp \wedge \mathit{pending}_i = \perp]_L$, $i \neq o$ remains true.

The intuition for I5 is that the clause containing this statement is intended to flush the data value from the cache, and it does not make sense for the cache to acquire ownership in order to do so. (This condition is also required for technical reasons below.)

The {Bus Observation} clause is included to deal with the cache's observations (*snooping*, in the terminology of the literature) of bus events generated by other caches. The cache may update its own state as a result of observing such events. It is assumed that all caches participate in each bus event, either by causing it, or by observing it. Observers may also exchange information during the bus event, e.g., by declaring that they retain a copy of the memory line. There is a further requirement that each event observed on the bus be caused by the action of one of the caches on the bus. Formally, we capture this by the following conditions:

`BObs`. Each bus event in a run of the system corresponds to the simultaneous execution by each cache of a bus event specification statement $[\alpha, \beta]_B$. At least one of these is not the {Bus_Observation} specification statement.

We also need some constraints on the definition of ownership:

`O1`. The owner's value for the memory line is never $\perp$, i.e., $\Box c_o \neq \perp$
`O2`. It is always the case that the owner, if a cache, knows that it is the owner, i.e., $\bigwedge_{i \in \texttt{Cache}} \Box(o = i \Rightarrow \mathbf{k}_i(o = i))$
`O3`. If a cache is the owner, it remains the owner until a bus event occurs.

We note that `O3` permits that ownership can be transferred from the main memory to a cache without a bus event occurring. We later give an example where this occurs.

An *candidate implementation* of the above specification consists of the following:

1. A set of concrete processes C_i for each of the cache specifications $\mathcal{C}_i$, obtained by substituting specific code for the specification statements in the process terms $\mathcal{C}_i$, in accordance with constraints `I1-I3`. These concrete processes may make use of variables additional to those mentioned above, in order to maintain information about the state of other caches.
2. A concrete process $\mathcal{M}$ for the memory.
3. A definition of the semantics of any handshake actions used in these processes, defining which combinations may occur in a handshake, and what the effect is on the local states.
4. A definition of the local propositions $\mathbf{k}_i\phi$: these should depend only on the local variables of cache i in the implementation. These definitions should be substituted in the guards of $\mathcal{C}_i$ in constructing the implementation C_i.
5. A definition of the ownership as a function of global state.

Given a candidate implementation, let $S = M \parallel (P_1 \parallel C_1) \parallel \ldots \parallel (P_n \parallel C_n)$ be the parallel composition of all the components, and let $\mathcal{I}$ be the system consisting of the set of runs of S, with propositions interpreted in accordance with their interpretations in the candidate implementation. Then the candidate implementation is an *implementation* if the following hold in $\mathcal{I}$.

1. Constraints `I4-I5`, `O1-O3` and `BObs` are satisfied.
2. The specification constraints are satisfied: for any run r of $\mathcal{I}$, for any interval (m, n) such that the code implementing the statement $[\phi, \psi]$ runs over interval (m, n) in r, if $\mathcal{I}, (r, m) \models \phi$ then $\mathcal{I}, (r, n) \models \psi$. (Note that because the underlying semantics is asynchronous, actions of other processes may occur during the interval $[m, n]$.)

For the purposes of the above definition, the knowledge operator in {Read Miss} should be interpreted according to the observational semantics of knowledge.

We now show that any implementation of the knowledge-based specification solves the cache coherence problem, by establishing that a strong type of sequentiality holds if one observes the system at the level of bus events. Say that a *write event (of cache i)* is an event involving cache i, such that the next event involving cache i is $\mathcal{P}_i!\,\mathbf{ack}$. That is, a write event of a cache i is the final event of the implementation of one of the specification statements $[\textsc{True}, c_i = v \wedge o = i]_L$ or $[\textsc{True}, c_o = v \wedge c_i = v]_B$. We say that the value v is the *value written* in this write event. We call events of the form $\mathcal{P}_j!\,w$ where w is a memory line value, *read events*, with value w. The following result states that values read are always equal to the last value written. When an event corresponds to the transition between times n and $n + 1$, we say it occurs at time n.

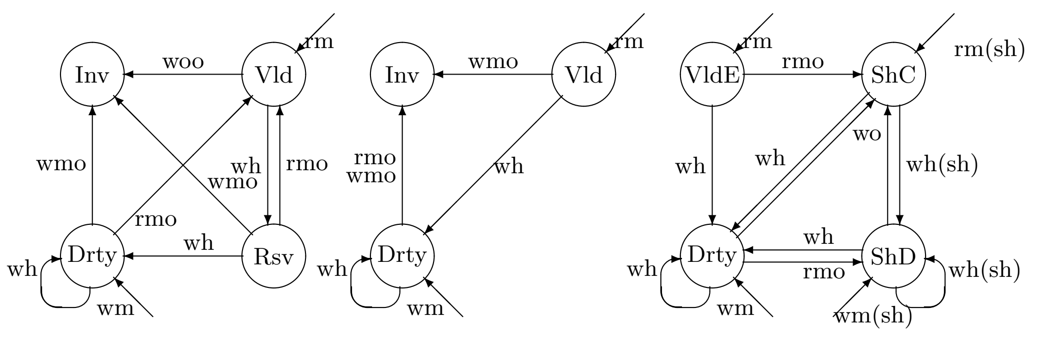
Lemma 1 (Sequentiality). *Let $\mathcal{I}$ be a system implementing the knowledge-based specification, and let r be a run of $\mathcal{I}$. If a write event with value v occurs at time m in r, a read event with value w occurs at time $n > m$, and no write event occurs at times p with $m < p \leq n$, then $v = w$.*

It is worth remarking that the specification statement for the {Read Miss} clause could have used the local proposition $\mathbf{k}_i(c_i = c_o)$ in place of $K_i(c_i = c_o)$, but this would be less general by validity of $\mathbf{k}_i(\phi) \Rightarrow K_i(\phi)$. In practice, however, $\mathbf{k}_i(c_i = c_o)$ could well be the postcondition established, and would have the benefit of matching the guard of the {Read Hit} statement. If one were to weaken $K_i(c_i = c_o)$ in the postcondition of the {Read Miss} clause to $c_i = c_o$, the sequentiality result would no longer hold, intuitively because there is then no guarantee that the owner does not perform a local write interleaved between execution of the bus event and the handshake $P_i!c_i$. However, a closely related result, stating that observations of reads and writes at the level of the processors are *linearisable* [HW90] could still be proved with this weakening.

5 Implementation of the Cache Protocols

One of the benefits of the knowledge-based approach is that it enables high level descriptions capturing the underlying reasons for correctness of a variety of protocols. We illustrate this by showing that three known cache coherence protocols are implementations of our knowledge-based program for cache coherence. We refer the reader to [AB86] for a detailed presentation of these protocols.

The first protocol is the so-called *Write-Once* protocol, which gets its name from the fact that the first write hit is performed as a write-through to the memory and gives the other caches the information to invalidate their copies. After this write-through, reads and writes can be performed locally, and the cache can also be flushed without a write-back if there has been no second write. See Figure 2(a) for a graphical representation. The cache line can be in four different states: `Invalid (Inv)`, `Valid (Vld)`, `Reserved (Rsv)`, or `Dirty (Drty)`. Writes and reads in state `Inv` lead to a *Write-Miss (wm)* resp. *Read-Miss (rm)*. In all other states the protocol reacts with a hit action, e.g., *Write-Hit (wh)* resp. *Read-Hit (rh)*. Other caches observe the miss actions, and take in response the transitions labelled *wmo* (write miss observed) *rmo* (read miss observed), and also observe the write-hit in state `Vld`, taking the transition *woo* (write once observed).



(a) (b) (c)

Fig. 2. State graphs of the (a) Write-Once, (b) Synapse, and (c) Dragon protocols

The *Synapse* protocol on the other hand gets by with only three states as shown in Figure 2(b). There is no `Rsv` state, hence all miss actions of other caches result in invalidation of the cache line.

The *Dragon* protocol uses both a write-through and a copy-back approach. It needs 5 local states, but Figure 2(c) shows only 4: the invalid state is omitted. Once loaded, the value is always kept up-to-date until the line is flushed. If the cache holds the line exclusively, then all reads and writes are local. When other caches may also have a copy of the line, all activities go over the bus to update all of them. After supplying the line, the main memory is involved only when the owning cache flushes the line. The four possible states of a loaded line are: `Valid-Exclusive (VldE)`, `Shared-Clean (ShC)`, `Shared-Dirty (ShD)`, and `Dirty (Drty)`. Both dirty states imply ownership and copy-back responsibility.

The added *(sh)* at some labels indicates that other caches indicated (via the so-called shared-line) that they share the line. For the owning cache this means further updates need to be communicated via the bus. The protocol is written for special hardware that features such shared-lines and offers an inter-cache communication that is much faster than write-cycles involving the main memory.

In order to show that these protocols are implementations of the knowledge-based program, we define additional local variables required for the processes, and instantiate the various components in the knowledge-based program. By way of the additional variables, it suffices to associate a variable s_i with each cache $\mathcal{C}_i$. The variable s_i is used to indicate the state the protocol associates with the memory line in the cache. The following table shows how the propositions in the knowledge-based program are defined.

	$o = i$	$\mathbf{k}_i(o = i)$	$\mathbf{k}_i(c_i = c_o)$	$\mathbf{k}_i(\mathtt{excl}_i)$
Write-Once	$s_i = \mathtt{Drty}$	$s_i = \mathtt{Drty}$	$s_i \neq \mathtt{Inv}$	$s_i \in \{\mathtt{Rsv}, \mathtt{Drty}\}$
Synapse	$s_i = \mathtt{Drty}$	$s_i = \mathtt{Drty}$	$s_i \neq \mathtt{Inv}$	$s_i = \mathtt{Drty}$
Dragon	$s_i \in \{\mathtt{Drty}, \mathtt{ShD}\}$	$s_i \in \{\mathtt{Drty}, \mathtt{ShD}\}$	$s_i \neq \mathtt{Inv}$	$s_i \in \{\mathtt{VldE}, \mathtt{Drty}\}$

The implementation of the locally executed specification statements turns out to be the same in all the protocols: the {Flush} statement $[o \neq i, c_i = \perp \wedge \mathit{pending}_i = \perp]$ is implemented by $c_i := \perp; s_i := \mathtt{Inv}$, and the {Write Local} statement $[\textsc{True}, c_i = v \wedge o = i]$ is implemented by $c_i := v; s_i := \mathtt{Drty}$. Using the above table, it can be seen directly that, for each protocol, these implementations ensure that the postcondition is satisfied.[1]

The remaining statements requiring definition are the bus event specification statements. Recall that these execute as a handshake between all caches and the memory, with all but one of the caches executing the {Observe} statement. The following table describes, for each of the protocols, the atomic state transitions resulting from the {Write Bus} statement. One column describes the effects at the cache initiating the bus event, the other describes the transition at each of the caches observing the bus event.

[1] For the variable $\mathit{pending}_i$ we need to use the context in which the statement occurs.

$\{$Write Bus$\}$	$[\textsc{True}, c_o = v \land c_i = v]_B$	Observer j
Write-Once	$c_i := v$; if $s_i = \texttt{Inv}$ then $s_i := \texttt{Drty}$ else if $s_i = \texttt{Vld}$ then $\{c_m := v; s_i := \texttt{Rsv}\}$	$c_j := \bot; s_j = \texttt{Inv}$
Synapse	$c_i := v; s_i := \texttt{Drty}$	$c_j := \bot; s_j = \texttt{Inv}$
Dragon	$c_i := v$; *if* Sh *then* $s_i := \texttt{ShD}$ *else* $s_i := \texttt{Drty}$	*if* $s_j \neq \texttt{Inv}$ *then* $\{c_j := v; s_j = \texttt{ShC}\}$

In this table, Sh abbreviates $\exists j \neq i(s_j \neq \texttt{Inv})$, and corresponds to the SharedLine used by the caches to indicate that they have a copy of the memory line. The table for the $\{$Read Miss$\}$ statement is as follows.

$\{$Read Miss$\}$	$[c_o = X, c_o = X \land K_i(c_i = c_o)]_B$	Observer j
Write-Once	$c_i := c_o; s_i := \texttt{Vld}$	if $s_j = \texttt{Drty}$ then $c_m := c_j$; if $s_j \in \{\texttt{Rsv}, \texttt{Drty}\}$ then $s_j := \texttt{Vld}$
Synapse	$c_i := c_o; s_i := \texttt{Vld}$	if $s_j = \texttt{Drty}$ then $\{\ c_m := c_j; c_j := \bot; s_j = \texttt{Inv}\}$
Dragon	$c_i := c_o$; if Sh then $s_i := \texttt{ShC}$ else $s_i := \texttt{VldE}$	if $s_j \in \{\texttt{VldE}, \texttt{ShC}\}$ then $s_j := \texttt{ShC}$; if $s_j = \texttt{Drty}$ then $s_j := \texttt{ShD}$

In the case of the Synapse protocol, our modelling of the $\{$Read Miss$\}$ condenses two bus events into one: the actual protocol does not allow the memory line to be read from another cache: if some cache has a `Drty` copy, it must first write this back to memory, and the cache requesting a read must then reissue its request and obtain the line from the memory.

In case of the $\{$Copyback$\}$ statement, the implementation in all the protocols is by $c_m := c_i; c_i := \bot; s_i := \texttt{Inv}$ at the cache performing the copyback, and no action at the observers. We remark that this implementation satisfies a stronger postcondition than is required by our knowledge-based specification, which does not require that $c_i = \bot$ after a copyback. This suggests the existence of protocols that separate copyback from flush operations. We return to this point later. Some other opportunities for alternate implementations are apparent: e.g., the specification permits the copyback to transfer ownership to another cache rather than the memory.

Another point needs to be noted concerning our modelling: by representing the memory line as a single variable, we do not distinguish between reads and writes of blocks versus single memory locations. Some changes would be required to capture this: for example, in the write-once protocol, a `Drty` cache observing a write must write the block back to memory, but the writing cache then need only write through the particular memory location being written.

6 Correctness of the Implementations

In order to verify that a concrete program is a correct implementation of the knowledge based-program, we need to check that each of the specification statements is satisfied by the code implementing that statement, that the test for knowledge are interpreted by sound conditions for knowledge, and that the other constraints noted are satisfied. This verification could be carried out by any standard means, e.g., theorem proving or model checking technology, and once carried out, implies, by the correctness theorem, that the implementation satisfies the correctness condition for cache coherence. (We can, of course, also verify this condition directly on the concrete code.)

We have conducted the verification by model checking using the model checker MCK [GM04]. In addition to providing standard temporal logic model checking capabilities, MCK enables model checking formulas of the logic of knowledge and time.[2] This makes it possible to check not just for correctness but also to determine if protocols are making optimal use of the information encoded in the local states. We illustrate our methodology in this section.

6.1 Global consistency

As an example of the global consistency properties we check, consider the requirement that there be a unique owner at all times. In the Write-Once protocol, we wish to define cache i to be the owner if $\mathit{status}_i = \mathit{Drty}$ and the owner to be the main memory otherwise. To check that this definition makes sense, we first model check the following formula $\Box(\mathit{status}_0 \neq \mathit{Drty} \vee \mathit{status}_1 \neq \mathit{Drty})$. This implies that the definition of ownership assigns a unique owner in all circumstances. Model checking other properties of this kind also provides a useful means to detect a variety of basic implementation errors.

6.2 Soundness

Although it may appear to involve model checking knowledge, verifying soundness of a concrete interpretation of the knowledge formulas can be carried out using only temporal logic model checking technology. For example, to obtain the Synapse protocol we interpret the local proposition $\mathbf{k}_i(\mathtt{excl}_i)$ by the test $s_i = \mathtt{Drty}$. Thus, soundness of this interpretation amounts to the claim $\Box(s_i = \mathtt{Drty} \Rightarrow (c_i \neq \bot \wedge \bigwedge_{j\neq i} c_j = \bot))$, which is a pure temporal logic formula.

6.3 Completeness

The most significant novelty of our methodology concerns cases where we model check knowledge formulas using the capabilities MCK provides for this purpose. In particular, for some simple configurations of the 3 protocols, we have checked the following:

`K1` A {Read Miss} establishes the postcondition $K_i(c_i = c_o)$, with respect to the observational (and hence also the perfect recall) interpretation of knowledge.

[2] We extended MCK to handle the asynchronous perfect recall semantics for knowledge in order to do so - the details will be reported elsewhere.

`K2` Does the formula $\Box(\mathbf{k}_i(\phi) \Leftrightarrow K_i\phi)$ hold for each of the local propositions $\mathbf{k}_i\phi$ in the specification, with respect to the asynchronous perfect recall semantics of knowledge?

The variables observable to the caches for these results included variables recording information potentially available to the caches in bus event, such as the transition being performed by the cache initiating the bus event. See the discussion above for the significance of property `K1`. A positive answer for Property `K2` shows that the protocols are *optimal* in their use of information, in the sense that the variables s_i store *all* the information available to the caches that they could use to determine whether the formulas ϕ hold. The answer to the question turns out to be yes in all three protocols when the formula ϕ is $o = i$. This is to be expected, since for each protocol, $\mathbf{k}(o = i)$ and $o = i$ are identical and i-local, from which soundness and completeness can easily be seen to follow.

However, when the formula ϕ is $\mathtt{excl}_i$, we find that the i-local interpretation for $\mathbf{k}_i(\phi)$ is sound but *not* complete in the Write-Once and Synapse protocols. A scenario showing incompleteness is where the first action in a run is for a cache to issue a {Read Miss}. Upon receipt of the value, the cache will be in state `Vld`, but will know from a log of its observations that no other cache has a copy, since it would have observed a read or write attempt by any other cache. To some extent this negative result is to be expected, since the Write-Once and Synapse protocols make no attempt to maintain exclusiveness information. The Dragon protocol *does* maintain information about whether the value is shared, and our model checking results indicate that whenever the Dragon cache could know that it has an exclusive copy from a complete log of its observations of bus events, it already knows this fact from just its Dragon state.

When the formula ϕ is $c_o = c_i$, things are more subtle. In this case, the interpretations we gave for $\mathbf{k}_i(\phi)$ turn out to be sound and complete for all three protocols as we have described them. However, this conclusion hinges on the fact that we perform the assignment $c_i := \bot$ whenever we perform $s_i := \mathit{Inv}$. An alternate interpretation of the protocols would rely on the Flush operation to perform $c_i := \bot$, and execute just $s_i := \mathit{Inv}$ at other times.[3] This allows us to ask the question: are there situations where a cache could know that a value that it has kept is valid, even though the protocol has marked it as invalid?

Our model checking experiments discovered that there is indeed such a situation. In the Synapse protocol, note that a cache in state `Drty`, on observing a {Read Miss}, writes the value back to memory and goes to state `Inv`. In fact, it could know that the value it has remains valid, since any subsequent write would still have to involve a bus transaction. This suggests a change to the Synapse protocol, in which the *rmo* transition from `Drty` in Figure 2(b) goes to `Vld` instead of `Inv`. This observation has been previously made in the literature ([Han93], p. 175), but the point we wish to make is that model checking knowledge provides an automated way to discover such lack of optimality in

[3] As we noted above, the knowledge-based specification allows us to distinguish between a copyback and a flush.

protocols.[4] If we make the proposed change to the Synapse protocol, we then find that the interpretation for $\mathbf{k}_i(c_o = c_i)$ is both sound and complete.

7 Conclusion

The literature on the knowledge-based approach to the design and analysis of distributed systems provides comparatively few examples from which to generalise in development of the approach. Our results in this paper provide a new case study, which illuminates a number of key points. In particular, we see here an example where correctness of an implementation of a knowledge-based program requires only soundness of the knowledge test, and can be separated from the issue of optimality in use of information, which corresponds to completeness.

We have also clarified Sweazey and Smiths analysis of cache coherence, by giving it a highly abstract, knowledge-theoretic presentation, that highlights a number of subtleties and reveals some design alternatives for the development of new cache coherence protocols. We remark that there has been a previous analysis of cache coherence using epistemic logic [MWS94]. The key differences to our work is that it considers a somewhat different class of protocols for network rather than hardware settings, and uses a logic of belief rather than a logic of knowledge. Neiger [Nei95] has also considered, from a knowledge theoretic perspective, distributed shared memory satisfying weaker properties than cache coherence.

Verification of cache coherence protocols has previously been approached by refinement, e.g. [BDG+94]. What distinguishes our approach is that it allows us to automatically discover situations where protocols are not optimal in their use of information. It also opens up avenues for synthesis of implementations. For example, after providing implementations of the specification statements, we could try to determine sound and complete interpretations of the knowledge conditions by automated synthesis. We leave this as a question for future work.

References

[AB86] James Archibald and Jean-Loup Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems (TOCS)*, 4(4):273–298, 1986.

[BDG+94] E. Brinksma, J. Davies, R. Gerth, S. Graf, W. Janssen, B. Jonsson, S.Katz, G. Lowe, M. Poel, A. Pnueli, C. Rump, and J. Zwiers. Verifying sequentially consistent memory. Computing Science Reports 94-44, Eindhoven University of Technology, 1994.

[CGH+95] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L McMillan, and L.A. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6:217–232, 1995.

[4] How to modify a protocol in such a situation is an interesting but much more subtle question, that we leave for the future.

[EMM98] Kai Engelhardt, Ron van der Meyden, and Yoram Moses. Knowledge and the logic of local propositions. In Itzhak Gilboa, editor, *Theoretical Aspects of Rationality and Knowledge, Proceedings of the Seventh Conference (TARK 1998)*, pages 29–41. Morgan Kaufmann, July 1998.

[EMM00] Kai Engelhardt, Ron van der Meyden, and Yoram Moses. A program refinement framework supporting reasoning about knowledge and time. In *Foundations of Software Science and Computation Structures*, pages 114–129, 2000.

[EMM01] Kai Engelhardt, Ron van der Meyden, and Yoram Moses. A refinement theory that supports reasoning about knowledge and time for synchronous agents. In *Proceedings LPAR 2001*, pages 125–141, 2001.

[Eng02] Kai Engelhardt. Towards a refinement theory that supports reasoning about knowledge and time for multiple agents. In John Derrick, Eerke Boiten, Jim Woodcock, and Joakim von Wright, editors, *Electronic Notes in Theoretical Computer Science*, volume 70. Elsevier, 2002.

[FHMV95] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge.* MIT Press, 1995.

[FHMV97] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. Knowledge-based programs. *Distributed Computing*, 10(4):199–225, 1997.

[GM04] Peter Gammie and Ron van der Meyden. MCK — Model-checking the logic of knowledge. In *Proc. Computer Aided Verification: 16th International Conference, CAV*, pages 479 – 483. Springer LNCS No. 3114, 2004.

[Han93] J. Handy. *The Cache Memory Book.* Academic Press, 1993.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[MM00] Ron van der Meyden and Yoram Moses. On refinement and temporal annotations. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium, FTRTFT 2000 Pune, India, September 20–22, Proceedings*, pages 185–201, 2000.

[MWS94] Lily B. Mummert, Jeannette M. Wing, and M. Satyanarayana. Using belief to reason about cache coherence. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 71–80, 1994.

[Nei95] Gil Neiger. Simplifying the design of knowledge-based algorithms using knowledge consistency. *Information and Computation*, 119(2):283–293, 1995.

[SS86] 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.