Proofs of Correctness of Cache-Coherence Protocols

Joseph Stoy[1], Xiaowei Shen[2], and Arvind[2]

[1] Oxford University Computing Laboratory
Oxford OX1 3QD, England
Joe.Stoy@comlab.ox.ac.uk
[2] Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge MA 02139, USA
xwshen, arvind@lcs.mit.edu

Abstract. We describe two proofs of correctness for Cachet, an adaptive cache-coherence protocol. Each proof demonstrates soundness (conformance to an abstract cache memory model CRF) and liveness. One proof is manual, based on a term-rewriting system definition; the other is machine-assisted, based on a TLA formulation and using PVS. A two-stage presentation of the protocol simplifies the treatment of soundness, in the design and in the proofs, by separating all liveness concerns. The TLA formulation demands precision about what aspects of the system's behavior are observable, bringing complication to some parts which were trivial in the manual proof. Handing a completed design over for independent verification is unlikely to be successful: the prover requires detailed insight into the design, and the designer must keep correctness concerns at the forefront of the design process.

1 Introduction: Memory Models and Protocols

Shared memory multiprocessor systems provide a global memory image so that processors running parallel programs can exchange information and synchronize with one another by accessing shared variables. In large-scale systems the physical memory is usually distributed across different sites to achieve better performance. Distributed Shared Memory (DSM) systems implement the shared memory abstraction with a large number of processors connected by a network, combining the scalability of network-based architectures with the convenience of shared memory programming. The technique known as caching allows shared variables to be replicated in multiple sites simultaneously to reduce memory access latency. DSM systems rely on cache-coherence protocols to ensure that each processor can observe the semantic effect of memory access operations performed by another processor.

A shared memory system implements a *memory model*, which defines the semantics of memory access instructions. An ideal memory model should allow efficient and scalable implementations while still having simple semantics for architects and compiler writers to reason about. Commit-Reconcile-Fences (CRF) [SAR99b] is a mechanism-oriented memory model intended for architects and compiler writers rather than for high-level parallel programming. It is intended to give architects great flexibility for efficient implementations, while giving compiler writers adequate control. It can be used to give precise descriptions of the memory behavior of many existing architectures; moreover, it can be efficiently implemented on these platforms. Conversely, if implemented in its own right, CRF provides a platform for their efficient implementations: thus upward and downward compatibility is obtained.

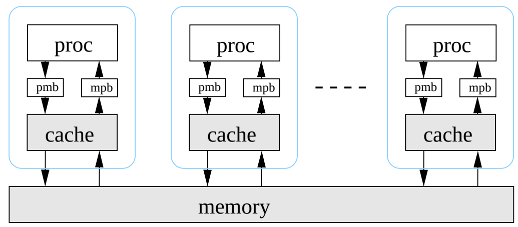
Caching and instruction reordering are ubiquitous features of modern computer systems and are necessary to achieve high performance. The design of cache-coherence protocols plays a crucial role in the construction of shared memory systems because of its profound impact on the overall performance and implementation complexity. Such protocols can be extremely complicated, especially in the presence of various optimizations. It often takes much more time to verify their correctness than to design them, and the problem of their verification has gained considerable attention in recent years [ABM93,Arc87,Bro90,PD95,PD96a] [PD96b,PD96c,PNAD95,SD95,HQR99,Del00]. Formal methods provide the only way to avoid subtle errors in sophisticated protocols.

This paper addresses the task of implementing CRF in its own right. As part of this task, we propose a cache-coherence protocol, Cachet [SAR99a,She00], which is adaptive in the sense that it can be tuned on the fly to behave efficiently under varying patterns of memory usage. This is a complex protocol; it is an amalgam of several micro-protocols, each intended for a different usage pattern. We show that the design of each micro-protocol, and Cachet itself, is simplified by taking it in two stages: "imperative" rules, which are sufficient to guarantee the protocol's soundness, are specified (and may be proved correct) before adding the "directive" rules, which are needed to ensure its liveness.

Even with this simplifying approach, however, the result is so complex that a formal correctness proof is desirable; and constructing such a proof with confidence calls for machine assistance. In this paper we compare two proof efforts for components of the Cachet protocol: one is manual, rooted in the term-rewriting methodology in which CRF and Cachet are described; the other is machine-assisted, using an implementation of Lamport's TLA [Lam94] in SRI's PVS [COR$^+$95]. The manual proof may be found in [She00]; the PVS version of TLA and the full machine-assisted proofs are available on the web [Sto].

1.1 The CRF Memory Model

The essence of memory models is the correspondence between each load instruction and the store instruction that supplies the data retrieved by the load. The memory model of uniprocessor systems is intuitive: a load operation returns the most recent value written to the address, and a store operation binds the value for subsequent load operations. In parallel systems, notions such as "the most recent value" can become ambiguous since multiple processors access memory



SYS	$\equiv$	Sys(MEM, SITEs)	*System*
SITEs	$\equiv$	{SITE}	*Set of Sites*
SITE	$\equiv$	Site(CACHE, PMB, MPB, PROC)	*Site*
CACHE	$\equiv$	{Cell(a,v,CSTATE)}	*Semantic Cache*
CSTATE	$\equiv$	Clean ‖ Dirty	*Cache State*
PMB	$\equiv$	[$\langle t$,INST$\rangle$]	*Processor-to-Memory Buffer*
MPB	$\equiv$	{$\langle t$,REPLY$\rangle$}	*Memory-to-Processor Buffer*
INST	$\equiv$	Load(a) ‖ Store(a,v)	*Transaction Instruction*
		‖ Commit(a) ‖ Reconcile(a)	
REPLY	$\equiv$	v ‖ Ack	*Reply*
MEM	$\equiv$	A $\rightarrow$V	*Main Memory*

Fig. 1. System Configuration of CRF (omitting fences)

concurrently. Surveys of some well-known memory models can be found elsewhere [AG96,KPS93].

One motivation underlying CRF is to eliminate the *modèle de l'année* aspect of many existing relaxed memory models while still permitting efficient implementations. It exposes both data replication and instruction reordering at the instruction set architecture level. The CRF model has a semantic notion of caches (referred to as "saches" when there is any danger of confusion with physical caches). Loads and stores are always performed directly on local caches. New instructions are provided to move data between cache and main memory whenever necessary: the Commit instruction ensures that a modified value in the cache is written back, while the Reconcile instruction ensures that a value which might be stale is purged from the cache. CRF also provides fine-grain fence instructions to control the re-ordering of memory-related instructions: they are irrelevant to protocol correctness, and are not treated further in this paper.

The CRF model permits aggressive cache-coherence protocols because no operation explicitly or implicitly involves more than one semantic cache. A novel feature of CRF is that many memory models can be expressed as restricted versions of CRF: programs written under those memory models can be translated into efficient CRF programs. Translations of programs written under memory

Processor Rules

CRF-Loadl Rule
$\text{Site}(\mathit{cache},\ \langle t,\text{Load}(a)\rangle{:}\mathit{pmb},\ \mathit{mpb},\ \mathit{proc}) \quad \mathit{if} \quad \text{Cell}(a,v,\text{-}) \in \mathit{cache}$
$\rightarrow \text{Site}(\mathit{cache},\ \mathit{pmb},\ \mathit{mpb}|\langle t,v\rangle,\ \mathit{proc})$

CRF-Storel Rule
$\text{Site}(\text{Cell}(a,\text{-},\text{-}) \mid \mathit{cache},\ \langle t,\text{Store}(a,v)\rangle{:}\mathit{pmb},\ \mathit{mpb},\ \mathit{proc})$
$\rightarrow \text{Site}(\text{Cell}(a,v,\text{Dirty}) \mid \mathit{cache},\ \mathit{pmb},\ \mathit{mpb}|\langle t,\text{Ack}\rangle,\ \mathit{proc})$

CRF-Commit Rule
$\text{Site}(\mathit{cache},\ \langle t,\text{Commit}(a)\rangle{:}\mathit{pmb},\ \mathit{mpb},\ \mathit{proc}) \quad \mathit{if} \quad \text{Cell}(a,\text{-},\text{Dirty}) \notin \mathit{cache}$
$\rightarrow \text{Site}(\mathit{cache},\ \mathit{pmb},\ \mathit{mpb}|\langle t,\text{Ack}\rangle,\ \mathit{proc})$

CRF-Reconcile Rule
$\text{Site}(\mathit{cache},\ \langle t,\text{Reconcile}(a)\rangle{:}\mathit{pmb},\ \mathit{mpb},\ \mathit{proc}) \quad \mathit{if} \quad \text{Cell}(a,\text{-},\text{Clean}) \notin \mathit{cache}$
$\rightarrow \text{Site}(\mathit{cache},\ \mathit{pmb},\ \mathit{mpb}|\langle t,\text{Ack}\rangle,\ \mathit{proc})$

Background Rules

CRF-Cache Rule
$\text{Sys}(\mathit{mem},\ \text{Site}(\mathit{cache},\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc}) \mid \mathit{sites}) \quad \mathit{if} \quad a \notin \mathit{cache}$
$\rightarrow \text{Sys}(\mathit{mem},\ \text{Site}(\text{Cell}(a,\mathit{mem}[a],\text{Clean}) \mid \mathit{cache},\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc}) \mid \mathit{sites})$

CRF-Writeback Rule
$\text{Sys}(\mathit{mem},\ \text{Site}(\text{Cell}(a,v,\text{Dirty}) \mid \mathit{cache},\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc}) \mid \mathit{sites})$
$\rightarrow \text{Sys}(\mathit{mem}[a{:=}v],\ \text{Site}(\text{Cell}(a,v,\text{Clean}) \mid \mathit{cache},\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc}) \mid \mathit{sites})$

CRF-Purge Rule
$\text{Site}(\text{Cell}(a,\text{-},\text{Clean}) \mid \mathit{cache},\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc})$
$\rightarrow \text{Site}(\mathit{cache},\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc})$

Fig. 2. The CRF Rules (omitting fences)

models such as sequential consistency and release consistency into CRF programs are straightforward.

Figure 1 shows the system configuration of the CRF model. We use `{SITE}` to indicate a set of sites, and $[< t, \texttt{INST} >]$ to indicate a list of items of the form $< t, \texttt{INST} >$ (each instruction is associated with a unique tag). Notation $\texttt{A} \rightarrow \texttt{V}$ denotes a function from addresses to values. Note that cache cells have two states, Clean and Dirty. The Clean state indicates that the value has not been modified since it was last cached or last written back; the Dirty state indicates that the value has been changed and has not been written back to the memory

Processor Rules				
Rule Name	**Instruction**	**Cstate**	**Action**	**Next Cstate**
CRF-Loadl	Load(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)
		Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)
CRF-Storel	Store(a,v)	Cell(a,-,Clean)	*retire*	Cell(a,v,Dirty)
		Cell(a,-,Dirty)	*retire*	Cell(a,v,Dirty)
CRF-Commit	Commit(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)
		$a \notin cache$	*retire*	$a \notin cache$
CRF-Reconcile	Reconcile(a)	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)
		$a \notin cache$	*retire*	$a \notin cache$

Background Rules				
Rule Name	**Cstate**	**Mstate**	**Next Cstate**	**Next Mstate**
CRF-Cache	$a \notin cache$	Cell(a,v)	Cell(a,v,Clean)	Cell(a,v)
CRF-Writeback	Cell(a,v,Dirty)	Cell(a,-)	Cell(a,v,Clean)	Cell(a,v)
CRF-Purge	Cell(a,-,Clean)	Cell(a,v)	$a \notin cache$	Cell(a,v)

Fig. 3. Summary of the CRF Rules

since then. Notice that different caches may have cells with the same address but different values.

Figure 2 gives the term-rewriting rules for the CRF model (omitting all mention of fences). As usual for term-rewriting systems, a rule may be applied whenever there is a context matching its left-hand side; if more than one rule is applicable, the choice is non-deterministic. For example, a Commit instruction at the head of the processor-to-memory buffer *pmb* does not in itself imply that the CRF-Commit rule can be applied: the rule is not applicable if the relevant cache cell state is Dirty. In that case, however, the background CRF-Writeback rule is applicable; and when that rule has been applied, the CRF-Commit rule can then be used.

In the CRF specification, we use constructors '|' and ':' to add an element to a set and to prepend an element to a list. For example, the processor-to-memory buffer *pmb* can be thought of as an FIFO queue; this aspect is captured by the use of ':'. The notation $mem[a]$ refers to the content of memory location a, and $mem[a{:=}v]$ represents the memory with location a updated with value v.

Figure 3 shows the rules in summarized form. The tabular description are easily translated into formal TRS rules (cases that are not specified represent illegal or unreachable states). The complete definition of CRF can be found elsewhere [SAR99b,She00].

1.2 The Cachet Protocol

The Cachet protocol is a directory-based adaptive cache-coherence protocol to implement the CRF memory model in distributed shared memory systems. It is a seamless integration of several so-called micro-protocols (Base, Writer-Push and

Migratory), though each micro-protocol is functionally complete in itself. It provides both intra-protocol and inter-protocol adaptivity which can be exploited by appropriate heuristic mechanisms to achieve optimal performance under changing program behavior. Different micro-protocols can be used by different cache engines, and a cache can dynamically switch from one micro-protocol to another.

The CRF model allows a cache-coherence protocol to use any cache or memory in the memory hierarchy as the rendezvous for processors that access shared memory locations, provided that it maintains the same observable behavior. The micro-protocols differ in the actions they perform when committing dirty cells and reconciling clean ones.

Base: The most straightforward implementation simply uses the memory as the rendezvous. When a Commit instruction is executed for an address that is cached in the Dirty state, the data must be written back to the memory before the instruction can complete. A Reconcile instruction for an address cached in the Clean state requires that the data be purged from the cache before the instruction can complete. An attractive characteristic of Base is its simplicity: no extra state needs to be maintained at the memory side.

Writer-Push (WP): If load operations are far more frequent than store operations, it is desirable to allow a Reconcile instruction to complete even when the address is cached in the Clean state; then a subsequent load access to the address causes no cache miss. This implies, however, that when a Commit instruction is performed on a dirty cell, it cannot complete until any clean copies of the address are purged from all other caches. It can therefore be a lengthy process to commit an address that is cached in the Dirty state.

Migratory: When an address is used exclusively by one processor for a considerable time, it makes sense to give the cache exclusive ownership, so that all instructions on the address become local operations. This is reminiscent of the exclusive state in conventional invalidate-based protocols. The protocol ensures that an address can be stored in at most one cache at any time. A Commit instruction can then complete even when the address is cached in the Dirty state, and a Reconcile instruction can complete even when the address is cached in the Clean state. The exclusive ownership can migrate among different caches whenever necessary.

Different micro-protocols are optimized for different access patterns. The Base protocol is ideal when the location is randomly accessed by several processors and only necessary commit and reconcile operations are invoked. The WP protocol is appropriate when some processors are likely to read an address many times before any processor writes the address. The Migratory protocol fits well when one processor is likely to read or write an address many times before any other processor uses the address.

1.3 The Imperative-&-Directive Design Methodology

To simplify the process of designing protocols such as these, we have proposed a two-stage design methodology called Imperative-&-Directive, to separate soundness and liveness concerns. Soundness ensures that the system exhibits only legal behaviors permitted by the specification; liveness ensures that the system eventually performs actions which make progress. The first stage of the design involves only *imperative rules*: these specify actions which can affect the soundness of the system. The messages handled by these rules are known as *imperative messages*. The second stage of the design adds *directive messages*: these can be used to invoke imperative rules, but they are also manipulated by other rules known as *directive rules*. Imperative and directive rules are properly integrated to ensure both soundness and liveness. Directive rules do not change the soundness of a state; moreover, improper conditions for invoking imperative rules can cause deadlock or livelock but cannot affect soundness. It therefore suffices to verify the soundness of the system with respect to the imperative rules, rather than the integrated rules of the integrated protocol.

As an example, the WP protocol includes an imperative rule which allows a cache to purge a clean value, notifying the memory via an imperative Purged message. The imperative rule does not specify when this must be invoked to ensure the liveness of the system. When the memory requires that the cache perform a purge operation (to allow a writeback elsewhere to complete), it sends a directive PurgeReq message to the cache. The integrated protocol ensures both soundness and liveness by requiring that the cache respond appropriately once such a request is received.

We also make an entirely separate classification of the rules of the integrated protocol, dividing them into two disjoint sets: *mandatory rules* and *voluntary rules*. The distinction is that for liveness of the system it is essential that mandatory rules, if they become applicable, are sooner or later actually executed. Voluntary rules, on the other hand, have no such requirement and are provided purely for adaptivity and performance reasons: an enabled voluntary rule may be ignored forever without harm to the protocol's correctness (but possibly with considerable harm to the performance).

Mandatory rules, therefore, require some kind of fairness to ensure the liveness of the system. This can be expressed in terms of weak or strong fairness. *Weak fairness* means that if a mandatory rule remains applicable, it will eventually be applied. *Strong fairness* means that if a mandatory rule continually becomes applicable, it will eventually be applied. When we say a rule is weakly or strongly fair, we mean the application of the rule at each possible site is weakly or strongly fair.

Liveness is not handled by the TRS formalism itself, so needs some extra notation. Temporal logic provides the appropriate repertoire. For example, the "leads to" operator "$\leadsto$" is defined by $F \leadsto G \quad \equiv \quad \Box(F \Rightarrow \Diamond G)$, which asserts that whenever F is true, G will be true at some later time. Then our overall liveness criterion (that every processor request is eventually satisfied) may be written as

$$\langle t, - \rangle \in pmb \quad \leadsto \quad \langle t, - \rangle \in mpb.$$

A mandatory action is usually triggered by events such as an instruction from the processor or a message from the network. A voluntary action, in contrast, is enabled as long as the cache or memory cell is in some appropriate state.

Protocol	Imperative Rules	Integrated Rules
Base	15	27
WP	19	45
Migratory	16	36
Cachet	75	146

Fig. 4. The Number of Imperative and Integrated Rules

For example, the voluntary purge rule allows a cache to drop a clean copy at any time (for example because of more pressing demands on the cache's limited capacity), while the mandatory purge rule requires the same operation once a PurgeReq request is received.

Conventional cache-coherence protocols consist only of mandatory actions. In our view, an adaptive coherence protocol consists of three components: mandatory rules, voluntary rules and heuristic policies. Voluntary rules provide enormous adaptivity, which can be exploited by various heuristic policies. An entirely separate mechanism can use heuristic messages and heuristic states to help determine when one of the voluntary rules should be invoked at a given time. Different heuristic policies can result in different performance, but they cannot affect the soundness and liveness of the system, which are always guaranteed.

The Imperative-&-Directive methodology can dramatically simplify the design and verification of cache-coherence protocols. Protocols designed with this methodology are often easy to understand and modify. Figure 4 illustrates the number of imperative and integrated rules for Cachet and its micro-protocols. Although Cachet consists of 146 rewriting rules, only 75 basic imperative rules need be considered in the soundness proofs, including the proofs of many soundness-related invariants used in the liveness proof. To simplify protocol design and verification still further, protocol rules can be classified in yet another dimension, into *basic* and *composite* rules [She00]. The verification of both soundness and liveness may then be conducted with respect only to the basic rules. The Cachet protocol, for example, contains 60 basic imperative rules and 113 basic integrated rules.

1.4 The Writer-Push Protocol

Our main example is the WP protocol, which is designed to ensure that if an address is cached in the Clean state, the cache cell contains the same value as the memory cell. This is achieved by requiring that all clean copies of an address be purged before the memory cell can be modified. As the name "Writer-Push" suggests, the writer is responsible for informing potential readers to have their stale copies, if any, purged in time. A commit operation on a dirty cell can therefore be a lengthy process, since it cannot complete before clean copies of the address are purged from all other caches.

There are three stable cache states for each address, Invalid, Clean and Dirty. Each memory cell maintains a memory state, which can be C[*dir*] or T[*dir*,*sm*],

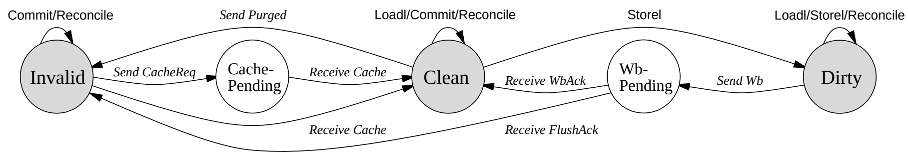


Fig. 5. Cache State Transitions of WP

where C and T stand for cached and transient, respectively. In the transient state, the directory *dir* contains identifiers of the cache sites in which the address is cached (the purpose of the suspended message buffer *sm* will be explained below).

There are five imperative messages, with the following informal meanings:

– Cache: the memory supplies a data copy to the cache.
– WbAck: the memory acknowledges a writeback operation and allows the cache to retain a clean copy.
– FlushAck: the memory acknowledges a writeback operation and requires the cache to purge the address.
– Purged: the cache informs the memory of a purge operation.
– Wb: the cache writes a dirty copy back to the memory.

The full WP protocol has in addition two transient cache states, WbPending and CachePending. The WbPending state means a writeback operation is being performed on the address, and the CachePending state means a cache copy is being requested for the address. There are two directive messages:

– PurgeReq: the memory requests the cache to purge its copy.
– CacheReq: the cache requests a data copy from the memory.

Figure 5 shows the cache state transitions of WP. A cache can purge a clean cell and inform the memory via a Purged message. It can also write the data of a dirty cell to the memory via a Wb message and set the cache state to WbPending, indicating that a writeback operation is being performed on the address. There are two possible acknowledgements for a writeback operation. If a writeback acknowledgement (WbAck) is received, the cache state becomes Clean; if a flush acknowledgement (FlushAck) is received, the cache state becomes Invalid (that is, the address is purged from the cache). When a cache receives a Cache message, it simply caches the data in the Clean state. A cache responds to a purge request on a clean cell by purging the clean data and sending a Purged message to the memory. If the cache copy is dirty, the dirty copy is forced to be written back via a Wb message.

Figure 6 summarizes the rules of the WP protocol. The cache engine and memory engine rules are categorized into mandatory and voluntary rules; the processor rules are all mandatory. A mandatory rule marked with 'SF' means the rule requires strong fairness to ensure the liveness of the system; otherwise it

Mandatory Processor Rules					
Instruction	**Cstate**	**Action**	**Next Cstate**		
Load(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	P1	SF
	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)	P2	SF
	$a \notin cache$	$\langle$CacheReq,$a\rangle \rightarrow$ H	Cell(a,-,CachePending)	P5	
Store(a,v)	Cell(a,-,Clean)	*retire*	Cell(a,v,Dirty)	P4	SF
	Cell(a,-,Dirty)	*retire*	Cell(a,v,Dirty)	P5	SF
	$a \notin cache$	$\langle$CacheReq,$a\rangle \rightarrow$ H	Cell(a,-,CachePending)	P6	
Commit(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	P7	SF
	Cell(a,v,Dirty)	$\langle$Wb,a,$v\rangle \rightarrow$ H	Cell(a,v,WbPending)	P8	
	$a \notin cache$	*retire*	$a \notin cache$	P9	SF
Reconcile(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	P10	SF
	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)	P11	SF
	$a \notin cache$	*retire*	$a \notin cache$	P12	SF

Voluntary C-engine Rules				
	Cstate	**Action**	**Next Cstate**	
	Cell(a,-,Clean)	$\langle$Purged,$a\rangle \rightarrow$ H	$a \notin cache$	VC1
	Cell(a,v,Dirty)	$\langle$Wb,a,$v\rangle \rightarrow$ H	Cell(a,v,WbPending)	VC2
	$a \notin cache$	$\langle$CacheReq,$a\rangle \rightarrow$ H	Cell(a,-,CachePending)	VC3

Mandatory C-engine Rules				
Msg from H	**Cstate**	**Action**	**Next Cstate**	
$\langle$Cache,a,$v\rangle$	$a \notin cache$		Cell(a,v,Clean)	MC1
	Cell(a,-,CachePending)		Cell(a,v,Clean)	MC2
$\langle$WbAck,$a\rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean)	MC3
$\langle$FlushAck,$a\rangle$	Cell(a,-,WbPending)		$a \notin cache$	MC4
$\langle$PurgeReq,$a\rangle$	Cell(a,-,Clean)	$\langle$Purged,$a\rangle \rightarrow$ H	$a \notin cache$	MC5
	Cell(a,v,Dirty)	$\langle$Wb,a,$v\rangle \rightarrow$ H	Cell(a,v,WbPending)	MC6
	Cell(a,v,WbPending)		Cell(a,v,WbPending)	MC7
	Cell(a,-,CachePending)		Cell(a,-,CachePending)	MC8
	$a \notin cache$		$a \notin cache$	MC9

Voluntary M-engine Rules				
	Mstate	**Action**	**Next Mstate**	
	Cell(a,v,C[dir]) ($id \notin dir$)	$\langle$Cache,a,$v\rangle \rightarrow id$	Cell(a,v,C[$id \mid dir$])	VM1
	Cell(a,v,C[dir]) ($dir \neq \epsilon$)	$\langle$PurgeReq,$a\rangle \rightarrow dir$	Cell(a,v,T[dir,ϵ])	VM2

Mandatory M-engine Rules					
Msg from id	**Mstate**	**Action**	**Next Mstate**		
$\langle$CacheReq,$a\rangle$	Cell(a,v,C[dir]) ($id \notin dir$)	$\langle$Cache,a,$v\rangle \rightarrow id$	Cell(a,v,C[$id \mid dir$])	MM1	SF
	Cell(a,v,C[dir]) ($id \in dir$)		Cell(a,v,C[dir])	MM2	
	Cell(a,v,T[dir,sm]) ($id \notin dir$)	*stall message*	Cell(a,v,T[dir,sm])	MM3	
	Cell(a,v,T[dir,sm]) ($id \in dir$)		Cell(a,v,T[dir,sm])	MM4	
$\langle$Wb,a,$v\rangle$	Cell(a,v_1,C[$id \mid dir$])	$\langle$PurgeReq,$a\rangle \rightarrow dir$	Cell(a,v_1,T[dir,(id,v)])	MM5	
	Cell(a,v_1,T[$id \mid dir$,sm])		Cell(a,v_1,T[dir,(id,v) $\mid sm$])	MM6	
$\langle$Purged,$a\rangle$	Cell(a,v,C[$id \mid dir$])		Cell(a,v,C[dir])	MM7	
	Cell(a,v,T[$id \mid dir$,sm])		Cell(a,v,T[dir,sm])	MM8	
	Cell(a,-,T[ϵ,(id,v) $\mid sm$])	$\langle$FlushAck,$a\rangle \rightarrow id$	Cell(a,v,T[ϵ,sm])	MM9	
	Cell(a,-,T[ϵ,(id,v)])	$\langle$WbAck,$a\rangle \rightarrow id$	Cell(a,v,C[id])	MM10	
	Cell(a,v,T[ϵ,ϵ])		Cell(a,v,C[ϵ])	MM11	

Fig. 6. The WP Protocol

requires only weak fairness. The notation '$msg \to dir$' means sending the message *msg* to the destinations in directory *dir*. The transient memory state T[*dir*,*sm*] is used for bookkeeping during a writeback operation: *dir* represents the cache sites which have not yet acknowledged the broadcast PurgeReq requests, and *sm* contains the suspended writeback message that the memory has received but has not yet acknowledged (only the source and the data need to be recorded).

2 The Manual Proof of Correctness

One way to show that an implementation is correct with respect to a specification is to show that one can simulate the other. In particular, every sequence of terms generated by the rewriting rules of the implementation ought to be compatible (with respect to some observation function) with some sequence that could be generated by the specification system. (Sometimes it is also possible to show the reverse simulation, but this is not necessary for the correctness of an implementation.)

Accordingly we prove the soundness of WP by showing that CRF can simulate WP. The first stage involves only the imperative messages (not the directives). Queues can be thought of as multi-sets, not ordered FIFO sequences, and messages may be selected from the queues non-deterministically; the soundness property will therefore not be compromised in the presence of specific reordering restrictions introduced later, such as FIFO message passing for each particular address. The second stage adds directives to the repertoire of messages, and operations to handle them. Accordingly, we first show soundness of the imperative subset, by proving that any imperative rule of WP can be simulated in CRF with respect to some particular abstraction function. The soundness of the complete protocol follows from the fact that all the other rules may be derived from the imperative subset.

We first define an abstraction function from WP to CRF. For WP terms in which all message queues are empty, it is straightforward to find the corresponding CRF terms: there is a one-to-one correspondence between these "drained" terms of WP and the terms of CRF. For WP terms that contain non-empty message queues, we apply a set of "draining" rules to extract all the messages from the queues. These rules are derived from a subset of the rules of the protocol, some of them in reverse: we use backward draining for Wb messages and forward draining for all other messages (forwarding draining of Wb messages would lead to non-deterministic drained terms when there are multiple writeback messages regarding the same address). Consequently, all the Cache, WbAck, FlushAck and Wb messages will be drained towards cache sites, while all the Purged messages will be drained towards the memory. The system of draining rules is strongly normalizing and terminating; so it is easy to use it to define an abstraction function.

The liveness proof deals with the integrated protocol, and typically assumes that the queues satisfy FIFO ordering. We prove that whenever a processor initiates an instruction, there will be a later state in which that instruction has been retired. The proof involves showing that the appropriate messages are placed in the queues, that each messages makes progress towards the head of the queue, and that it is dealt with when it arrives at its destination. Each of these requires a fairness constraint on the relevant rules, to guarantee that they are eventually executed; arrival at the head of a queue also requires an induction. Matters are further complicated by the possibility of stalled messages, and transitory states of the main memory.

Many details of this proof are omitted; some arise similarly in the machine-assisted version and are treated below in Section 4. A complete description of the manual proof can be found elsewhere [She00].

3 Setting Up for the Machine-Assisted Verification

3.1 Choice of a Logical System

For systems as complicated as the CRF protocols, correctness proofs become too large to handle reliably by hand. It is therefore necessary to resort to mechanical assistance, and so to choose an appropriate tool. A mechanical tool requires more formality in the expression of a specification; and it is often convenient to adopt some existing formal system as a vehicle. The liveness parts of the CRF specifications find natural expression in terms of temporal logic, and it was therefore in this area that we looked for a formal system.

It takes a good deal of investment of effort to become proficient in the use of any substantial piece of mathematics; so it is worth choosing carefully among the possibilities before one starts. In the choice of a suitable system, we have adopted one engineered to concentrate on the areas in which most of our detailed work will be found. The "actions" of Lamport's Temporal Logic of Actions [Lam94] correspond closely with the transitions of our term-rewriting systems; and its temporal logic provisions seem to cope with the liveness and fairness constraints we need to handle, without burdening us with unnecessary complication.

3.2 Choice of a Tool

In the selection of a mechanical tool, too, a careful choice needs to be made. Lamport correctly points out that when verifying a design one spends most of the time in mundane checking of small print, based on simple arithmetic rather than abstruse logic. So we look for a tool which embodies efficient decision procedures in these areas; indeed, for this it is worth sacrificing the ability to define and to work with non-conventional logics (as will be seen, this trade-off arises in the present work). PVS [COR$^+$95] fits that particular bill. Moreover, PVS now contains a fairly rudimentary model-checker which may be of use in certain parts of our investigations in the future.

We have accordingly implemented [Sto] TLA in PVS. This may be thought of as analogous to implementing the algorithms of Linear Algebra in C, except that instead of developing the subroutines, with the aid of the compiler, we are proving the theorems, with the aid of the proof engine.

In the tabular form:

Rule Name	Instruction	Cstate	Action	Next Cstate
CRF-Loadl	Load(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)
		Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)

In TRS:

CRF-Loadl Rule
   Site(*cache*, $\langle t$,Load(a)$\rangle$:*pmb*, *mpb*, *proc*)   *if* Cell(a,v,-) $\in$ *cache*
$\rightarrow$ Site(*cache*, *pmb*, *mpb*$|\langle t$,$v\rangle$, *proc*)

CRF-Loadl′ Rule
   Site(*cache*, *interface*(Load,a,-))   *if* Cell(a,v,-) $\in$ *cache*
$\rightarrow$ Site(*cache*, *interface*(Ready,a,v)
)

In PVS:

```
CRF_Load(i) : action = LAMBDA s0, s1:

  ( s0`proc(i)`op = load AND
    full?(s0`cache(i)(a)) AND

    s1 = s0 WITH
    [`proc(i)`op  := ready,
     `proc(i)`val := val(s0`cache(i)(a))] )
   WHERE a = s0`proc(i)`adr
```

Fig. 7. The Load Rule in PVS

3.3 The Move to PVS

In the PVS version of our specification, since we are not concerned with instruction re-ordering, we found it convenient to simplify the processor interface: for this version it is just a $\langle$Op,a,$v\rangle$ triple, where Op is one of the CRF operations or Ready). To illustrate how the TRS version is transliterated into PVS, Figure 7 shows various definitions for the Load rule. First we repeat the tabular summary from Figure 3; then we give the TRS rule in full, followed by the version with the simplified interface; finally we give a PVS version (note that the backquote denotes field selection from a record).

It will be seen that there is a fairly obvious correspondence between TRS rules and their equivalents as TLA "actions" in PVS. Indeed, we are planning to automate the translation, so that the same source file can be used both as the basis for our verification proofs and as the starting point for hardware synthesis. Similarly, there is a close correspondence between a TLA behavior and the sequence of states arising in a TRS reduction.

Assertions about states also appear similar in the two systems. For example, the assertion which allows the Reconcile instruction to be a no-op in WP is shown

Assertion about TRS:

$\text{Cell}(a,v,\text{Clean}) \in \text{Cache}_i(s) \Rightarrow \text{Cell}(a,v,-) \in \text{Mem}(s)$

In PVS:

```
clean_w?(s`cache(i)(a)) => s`cache(i)(a)`val = s`mem(a)
```

Fig. 8. Assertions on States

For each site and address,
  The cache state is WbPending **iff**
    (1) there's a Wb in the home queue **or**
    (2) there's a WbAck or FlushAck in the site queue **or**
    (3) the memory state is transient, and a Wb is stalled there
  and never more than one of (1),(2),(3) is true
  and there's never more than one Wb in the home queue
  and there's never more than one Purged in the home queue
  and there's never more than one WbAck or FlushAck in the site queue
  and if the cache is Clean its value is the same as the memory's
  and the site is in the memory directories **iff**
    (a) there's a Cache command in the site queue **or**
    (b) the cache is valid **or**
    (c) the cache is WbPending and there's not a FlushAck in the site queue **or**
    (d) there's a Purged in the home queue
  and never more than one of (a),(b),(c),(d) is true
  and any Wb command in the home queue has the same value as the cache
  and any Cache command in the site queue has the same value as the memory
  and any stalled Wb has the same value as the cache
  and if there's a WbAck in the site queue, the cache value is the same as the memory's
  and none of the stalled messages in the home queue is an imperative message.
(Note that here "a site is in the memory directories" means it's in the directory either of the C state or the T state, **or** there's a stalled Wb message about it in the sm set of the T state.)

Fig. 9. The Invariant for WP

in both forms in Figure 8. These would naturally form part of an "invariant assertion" to be proved true for all states in a behavior (which in TLA would find expression as a temporal formula under the "always" operator $\Box$). Since the truth of the assertion in Figure 8 depends on the correct messages being sent at appropriate times, the complete assertion to be proved is much more complicated: the invariant used for the PVS proof is shown, somewhat informally, in Figure 9. In each system the main part of the proof of invariance is a large case analysis, showing that the truth of the assertion is preserved by each of the rules.

The liveness assertions, too, are fairly similar: they are shown in Figure 10. In the PVS notation, the type conversion between predicates on states and on behaviors is supplied automatically by the system.

About the TRS version:

$\mathrm{Proc}_{id}(\sigma) = \langle \mathrm{Op}, -, - \rangle \quad \rightsquigarrow \quad \mathrm{Proc}_{id}(\sigma) = \langle \mathrm{Ready}, -, - \rangle$

In PVS:

```
LET op_is_ready(i) : state_pred =
   LAMBDA s: s`proc(i)`op = ready
IN
LAMBDA b : FORALL i : ( []<>(op_is_ready(i)) )(b)
```

Fig. 10. The Liveness Assertions for WP

3.4 Structure of a TLA Specification in PVS

PVS is a strongly typed system, so we must begin by setting out the structure of the state (corresponding to the structure of a complete TRS term). The state contains components for the entire universe with which we shall be concerned: the abstract specification and all its implementations. Of course, not all components will be relevant to each specification; a specification's footprint sets out what is relevant and what is not. In the case of CRF, the fields are `proc` (an array of processors, one per site), `mem` (the main memory), and `cache` (an array of caches). The only relevant parts of a processor are its interface with the memory system; and in our simplified version this means that each element in the `proc` array is a triple, `<op, adr, val>`, where `op` is one of the CRF operations (omitting fences) or Ready.

A TLA specification does not usually attempt to regulate the entire state. For one thing, it needs to leave some freedom to the implementer. For example, CRF includes atomic transitions which write back data from cache to memory; this is implemented using queues, and the operation is no longer atomic. TLA allows us to be precise about which parts of the state are constrained by the specification, and which parts are of no concern. (In our example only the processor interfaces will be constrained by the specification; the implementation may use any convenient mechanism to achieve a satisfactory behavior of those interfaces.)

We accordingly say that each specification has a "footprint", which consists just of those parts of the state which it is constraining. If the footprint components in some sequence of states (some behavior) satisfy a specification, changes in other components are irrelevant.

A TLA specification is normally constructed from three principal components. Each component is formally an assertion about behaviors, but each of them has a different thrust. The first specifies the initial state (the first element in the behavior sequence. Each processor interface is ready, and each cache is empty for every address. The obvious definition is:

```
Init_crf : state_pred = LAMBDA s:
```

```
(FORALL i : s`proc(i)`op = ready) AND
(FORALL i,a : empty?(s`cache(i)(a)))
```

In fact we prefer to define the predicate as a function on the footprint rather than on the whole state. Doing this systematically makes it much simpler to prove that the whole specification has the required footprint, which is necessary later. The type checker automatically inserts the necessary (predefined) type conversions. So the definition becomes

```
Init_crf : state_pred = LAMBDA fp:
 (FORALL i : fp`proc(i)`op = ready) AND
 (FORALL i,a : empty?(fp`cache(i)(a)))
```

The second component specifies the permissible transitions. As we have seen, this part is closely related to the TRS rules: each rules has a corresponding PVS definition. The definitions for the Load rule were given in Figure 7; in the PVS version the first group of terms gives the precondition for the transition, and the second specifies the change. Here again we actually give the assertion in terms of the footprint, using `fp0` as the starting footprint and `fp1` the resulting one instead of `s0` and `s1`. The rules are grouped together, using appropriate predefined operators, into a composite action, say `CRF_N`, which specifies that any one of the component transitions may occur.

Since TLA specifications relate only to a particular footprint, and we do not wish in any way to constrain transitions elsewhere in the state, TLA introduces the notion of stuttering. A stutter occurs when two successive states are identical, or (from the point of view of a particular specification) when the footprint in the two states is unchanged. Only stutter-independent specification formulae are allowed by TLA: that is to say, a behavior acceptable to the formula must remain so if stuttering transitions are added or removed. (In TLA itself this is ensured by syntactic restrictions on the grammar of the formula notation; in our PVS version we have to check it semantically when necessary.) Lamport uses the notation $\Box[CRF\_N]_{c\_fp}$, which we write as `alSQUARE(CRF_N, c_fp)`, to assert that each state in a behavior is related to the next either by a transition allowed by `CRF_N`, or by a transition which stutters on the footprint c_fp. This is the second component of our specification.

The third component specifies the "liveness" requirement. Since the component that specifies the transition rules allows the possibility of continuous stuttering, we must specify the requirement that something actually happens. In general, the third component is intended to specify some global requirements of the behavior, without contradicting anything specified in the previous two components. In this case we wish to assert that any operation initiated by any processor interface eventually completes; the form of this assertion was described above, and we use it to define the formula `CRF_fair`. Notice that we do not impose any requirement that any operation should ever commence: that is up to the processor, and so is no part of the memory specification.

The three components are combined into the formula `Crf`:

```
Crf : temporal_formula =
 Init_crf AND  alSQUARE(CRF_N, c_fp) AND CRF_fair
```

This formula has a footprint which includes the processor interface, the main memory and the caches. Finally, therefore, we must indicate that we are attempting to specify only the behavior of the processor interface (`proc`). We use a notation very like an existential quantifier:

```
CRF : temporal_formula = EXISTSV(mem_cache, Crf).
```

Here, `mem_cache` is a variable consisting of the `mem` and the `cache` components of the state. The `EXISTSV` operator, applied to a behavior, asserts that there is a sequence of values for this variable with which the given behavior, or a behavior stutter-equivalent to the given behavior, can be updated, so as to give a behavior acceptable to `Crf`. Thus CRF is also stutter-independent, and constrains only the behavior of the processor interface: its footprint is merely the `proc` component of the state. The formal definition of `EXISTSV` is discussed by Lamport [Lam94]; the PVS definition is given at [Sto].

`CRF` is our final specification. Notice that the liveness component refers only to the externally visible interface—we find this convenient, as it allows the component to appear unchanged in the specification of the implementation. We later replace this component in an implementation by assertions about the fairness of some of the operations, and prove that they are sufficient to guarantee the original requirement.

3.5 Structure of a TLA-based Proof

An assertion of correctness is of the form

```
ASSERT(
 EXISTSV(mem_cache_queues, Cbase)
           => EXISTSV(mem_cache, Crf) )
```

The main part of a proof of this assertion is the construction of an "abstraction function". This is a function from the state as manipulated by the implementation (by `Cbase` in our example) to the kind of state acceptable to the abstract specification (`Crf`). More precisely, since it is not allowed to change the externally visible parts of the footprint, it is a substitution for the `mem` and `cache` components. Then we prove that, under this substitution, a state acceptable to `Cbase` is acceptable to `Crf`, i.e. that

```
ASSERT( Cbase => subst(Cbase_bar, mem_cache)(Crf) )
```

where `Cbase_bar` is the state-function giving the value to be substituted for the `mem_cache` variable.

In all but fairly trivial cases (such as our "derived rules" example below), we shall need some extra properties of the state to prove the correctness of this assertion. For example, we may need to prove that values waiting in queue entries remain faithful copies of values in memory, or that queues contain no duplicate entries. So another important part of the proof is the construction and verification of the appropriate "invariant" properties. It may be noted that some investigations of the correctness of protocols amount merely to the demonstration of the invariant; in our approach, invariants are aids to proving the validity of the corresponding implementation.

Once we have proved the substitution assertion above, we may infer (just as with ordinary existential quantification)

```
ASSERT( Cbase => EXISTSV(mem_cache, Crf) )
```

and finally, provided that `EXISTSV(mem_cache, Crf)` is independent of the variable `mem_cache_queues`, and that various of the formulae are stutter-independent, we infer

```
ASSERT( EXISTSV(mem_cache_queues, Cbase)
                    => EXISTSV(mem_cache, Crf) )
```

as required. The independence criterion is satisfied, partly because `Crf` does not involve the queues at all, and partly because for any formula `F`,

```
EXISTSV(mem_cache, F)
```

is independent of `mem_cache`. This final part of the proof is usually fairly formulaic: all the real work is in the previous sections.

In our application, the liveness component of the specification remains unchanged in the implementation, unaffected by the substitution; if this had not been the case, there are TLA rules for dealing with this much more complicated situation. See below for the treatment of liveness in this application.

3.6 Example: Derived Rules

As a simple example of this approach, we consider the validity of some "derived rules" in CRF. For example, CRF requires that an address be cached before a store operation overwrites the value; but clearly this is in some sense equivalent to a single operation which establishes a "dirty" value in a cache which did not previously contain that address. So there are two specifications, one (`CRF2`) containing the extra store-on-empty rule and the other (`CRF`) not: in what sense are they equivalent? If we consider the raw behavior, of the `mem`, `cache` and `proc` footprint, they are *not* equivalent: `Crf2` can do in a single transition what necessarily takes two in `Crf`. If we consider the behavior of the "quantified" version, however, in which the changes in `mem` and `cache` are invisible, the difference becomes merely an extra stutter (in this case, *before* the transition that affects `proc`); and we have agreed that behaviors which differ only in stuttering are to be considered equivalent.

To prove this equivalence we must show that each version implies the other. It is easy to show that every behavior acceptable to `CRF` is also acceptable to

```
svquiescent : action =
 (LAMBDA s0,s1: null?(s0`stut) AND null?(s1`stut))

setsv(i, sts) : action =
 (LAMBDA s0,s1:
   null?(s0`stut) AND s1 = s0 WITH [stut := inuse(sts, i)])

resetsv(i, sts) : action =
 (LAMBDA s0,s1: s0`stut = inuse(sts, i) AND null?(s1`stut))

CRF_Ns : action = CRF_N AND svquiescent

dummyCRF_soe : action =
  LAMBDA ss: EXISTS i :
     enabled(CRF_Store_on_empty(i)) AND setsv(i, store_on_empty_flag)

CRF_soe : action =
  LAMBDA ss: EXISTS i :
     Crf_Store_on_empty(i) AND resetsv(i, store_on_empty_flag)

CRF_N2s : action = CRF_Ns OR dummyCRF_soe OR CRF_soe
```

Fig. 11. Part of the "Derived Rule" Specification

```
Crf_bar(s) : sfnval =
  CASES s`stut OF
   nil : mcs(s`mem, s`cache, s`stut),
   inuse(sts, i) :
    IF    sts = store_on_empty_flag THEN
     mcs(s`mem,
         s`cache WITH [(i)(a) := cell(clean, s`mem(a))],
         s`stut)
          WHERE a = s`proc(i)`adr
    ELSE mcs(s`mem, s`cache, s`stut)
     ENDIF
  ENDCASES
```

Fig. 12. The Abstraction Function

`CRF2`, since `CRF2`'s rules are a superset of `CRF`'s. The argument for the other direction proceeds in two stages. First, we define yet another system, `Crf2s`, based on `Crf2` and involving a new state variable (let us call it `stut`) to manage the stutter. `stut` normally has a null value. In this new system we arrange that each occurrence of the store-on-empty rule must be preceded by a transition which merely sets `stut` to a non-null value; and store-on-empty itself is altered so that it also resets `stut` to null. Some of the details are shown in Figure 11. We then prove, using a TLA theorem provided for this purpose, that

```
Crf2 = EXISTSV(stut, Crf2s)
```

Next we define an abstraction function from the `Crf2s` state to the `Crf` state; it maps any state in which `stut` is non-null to the intermediate state in the two-transition equivalent of `store_on_empty`, and otherwise makes no change (see Figure 12). Using this function we can prove

```
ASSERT( EXISTSV(mem_cache_stut, Crf2s)
                   => EXISTSV(mem_cache_stut, Crf) )
```

Then, on the left-hand side, we can argue that

```
EXISTSV(mem_cache_stut, Crf2s) =
  EXISTSV(mem_cache, EXISTSV(stut, Crf2s))
```

and hence (using a previous result)

```
EXISTSV(mem_cache_stut, Crf2s) =
  EXISTSV(mem_cache, Crf2)
```

Similarly for the right-hand side,

```
EXISTSV(mem_cache_stut, Crf) =
  EXISTSV(mem_cache, EXISTSV(stut, Crf))
```

and hence

```
EXISTSV(mem_cache_stut, Crf) = EXISTSV(mem_cache, Crf)
```

since `Crf` is independent of `stut`. This gives us the equality of `CRF2` and `CRF`, as required.

4 The Machine-Assisted Proof of WP

4.1 Soundness

Our main example is the WP micro-protocol for Cachet. This follows the outlines we have described, but it also involves a version of the "derived rule" example. The main optimization in this protocol is that it allows reconcile-on-clean as a single operation (without reference to the main memory, and thus avoiding all the overhead of queued messages and responses). In CRF this single operation becomes the triple <Purge, Reconcile, Cache>. So, as in the previous example, we must use stuttering variables to arrange that the single operation is preceded and followed by a stutter. Since there are different arrangements depending on whether the stutter precedes or follows the externally visible transition, we do this in two stages, using two stuttering variables; but the methodology for each stage is exactly as described above.

Our next task is to define the soundness invariant. Its principal clause asserts that a clean cache value is always equal to the value in the main memory—this is what is required to justify the reconcile-on-clean optimization. Other clauses (for example, that the messages in the queues are reasonable) are needed to guarantee that the abstraction function will behave as expected. Yet more clauses (for example, that various conditions are mutually exclusive) were added during early attempts at the proof—strengthening the hypothesis in order to prove the induction.

We prove that this invariant is preserved by any of the permissible transitions. (This proof is too big to be done monolithically, so it must be split into smaller sections. Since the clauses in this invariant are inter-related, it is best to split the proof by operation, so that each lemma says that the complete invariant is preserved by a particular operation. These lemmas are then used to prove that the invariant is preserved by any transition. This result, together with a straightforward proof that any state satisfying the initial predicate also satisfies the invariant, is then used to prove that the invariant always holds for any behavior satisfying the WP specification.

Next we define the abstraction function. This, as in the example above, provides a substitution for `mem` and `cache`; in fact, also as above, it never changes `mem`. As in the manual proof, the mapping is trivial when the queues are empty; when there are queue entries, it is necessary to decide whether it is preferable to treat a particular entry as not having been issued or as having arrived at its destination. The abstraction function also has to handle the stuttering variables. So it does three things:

1. If either of the stuttering variables is non-null, it provides the appropriate intermediate state for the operation sequence;
2. it treats any WbAck or FlushAck command in the site queue as having arrived (but note that it ignores any Cache command in the site queue);
3. it translates the various cache states (Clean etc.) to their CRF equivalents.

The next stage is to prove, again operation by operation, that under this substitution each WP operation either simulates the appropriate CRF operation or is a CRF stutter (that is, a no-op). Finally, an argument manipulating the `EXISTSV` quantifiers, similar to that shown for the previous example, is required to complete the soundness proof.

4.2 Liveness

The liveness component of the WP protocol was, like its counterpart in CRF, simply

```
Cwp_fair : temporal_formula = LAMBDA b : FORALL i :
 ( []<>(op_is_ready(i)) )(b)
```

We now define a new version of this protocol `Cwp_fair`, in which this component is replaced by a formula asserting that various subsets of the transition rules are fair; the other components remain unchanged. The liveness proof consists of showing that this version is sufficient to imply the other; that is to say, that the fairness constraints are sufficient to guarantee that the original liveness criterion is satisfied.

The fairness constraints in `Cwp_fair` may be spelled out as in Figure 13. The various clauses sprung partly from intuition arising during the design of the protocol, and partly from the formal requirements of the TLA theorems used in the liveness proof. Note that the `Cwp_fair` specification is itself at only an intermediate stage in an implementation. Its terms may be regrouped into separate specifications of the various subcomponents of the system (the cache engines, the queue-processing engines and so on), and these subcomponents then further

– For each site, the set of rules which complete processor operations (Load, Store, Reconcile, Commit) is strongly fair.
– For each site, the set which deals with stalled processor operations is weakly fair.
– For each site and address, the set which services the site's incoming queue is weakly fair.
– For each address, the set which services the memory's incoming queue is weakly fair.
– For each address in addition, the rule which services a CacheReq request when the memory is in its `C` state is strongly fair.
– For each address, the set of rules which deal with the memory's `T` state is weakly fair.

Fig. 13. The Fairness Constraints for WP

– The only commands in site queues are Cache, WbAck, FlushAck and PurgeReq.
– The only commands in home queues are CacheReq, Wb or Purged.
– The only non-empty cache states found are Clean, Dirty, CachePending and WbPending.
– If both a Cache and a WbAck or FlushAck command are in a site queue, the Cache command is later.
– If a stalled Wb command is in the `sm` component of a `mem` directory, the site concerned is not entered in the `dir` component.
– If a site is entered in the `dir` component, and `mem` is in a transient state, and the corresponding site queue is not empty, then its last entry is a PurgeReq command.
– If a site is entered in the `dir` component, and `mem` is in a transient state, then there is either a PurgeReq command in its site queue or a Wb or Purged command for that site in the home queue.
– If a site is in the CachePending state (for a given address), then any CacheReq command will be the latest entry for that site in the home queue.
– If a site is in the CachePending state (for a given address), then there is either a CacheReq message for that site in the home queue or in the queue for stalled CacheReq messages in `mem`, or a Cache command in its site queue.

Fig. 14. The Extra Invariants for the Liveness Proof

refined. At this stage the fairness requirements of the separate subcomponents may be realized in various ways: using dedicated hardware, or scheduling resources in a way which guarantees service, or using queueing theory to show that the probability of denial of service tends to zero over time.

The liveness proof itself requires more invariants, in addition to those already required for the soundness proof. Unlike the latter, these involve directive messages as well as imperative ones. The extra clauses are shown in Figure 14. We must, as before, show that this invariant is preserved by the operations. (In this case, each new clause is independent of most of the others—sometimes they go in pairs. It is therefore possible to structure the proof differently from before, and to have each lemma prove a single clause across all the operations.)

After the invariant is shown always to hold, we must prove that (for each site) op is always eventually ready. For this it is enough to show that $\texttt{op} \neq \texttt{Ready} \leadsto \texttt{op} = \texttt{Ready}$. Since this transition is made by processor completion operations, which are strongly fair, this reduces to showing that such operations are continually being enabled. But these operations are enabled unless one of the following conditions holds:

1. The cache is in a transitory state (CachePending or WbPending).
2. The cache is empty, and a Load operation is requested.
3. The cache is dirty, and a Commit operation is requested.

So we must show that each of these conditions gets resolved. Since the stalling operations are weakly fair, conditions 2 and 3 eventually become 1; so we are left to show that the CachePending and WbPending states lead to the clean state or (for WbPending only) the empty state.

For WbPending, the invariant shows that there must be a message in the home queue or the site queue, or a stalled entry in the central memory's transient state. We show that each queue entry makes progress in its queue: this is a proof by well-founded induction to show that the entry eventually reaches the top of the queue, relying on the relevant fairness condition to show that any non-empty queue eventually receives attention. If the site is entered in the set of stalled sites at the main memory, we must show that it eventually leaves that state. This is another well-founded induction, to show that the cardinality of that set eventually decreases to zero; but a pre-condition for this is that the `dir` component of the `mem` state is empty. Showing that this eventually happens requires yet another well-founded induction: the invariant shows that for each element in this set there is a PurgeReq in the site queue or a Purged in the home queue, so two further inductions are required to show that these entries eventually have their effect.

The CachePending argument is similar, with the added complication of the possibility of stalling a CacheReq message and subsequently reinstating it in the original queue. This requires greater subtlety in the inductions, as the movement of the queue is no longer strictly FIFO.

It will be seen that this liveness proof itself is a complicated nest of cycles. While proving that there is progress in each cycle, it is always necessary to allow for the possibility of abandoning it because some outer cycle has been completed some other way (for example, a voluntary cache action may obviate further need to make progress with a CacheReq message). When dealing with an imperative message, we are usually concerned with the earliest occurrence in a queue; but in the case of a directive it is the latest one which is important—this leads to certain technical differences in treatment.

This proof is very complicated, and the corresponding proof for the complete, integrated Cachet protocol is still more complex. It is necessary to be very systematic in structuring the proof to avoid losing track: Lamport discusses this issue [Lam93] in the context of a manual proof. With Akhiani et al. [ADH+99] he has employed a hierarchical proof technique in a manual verification of sophisticated cache-coherence protocols for the Alpha memory model. The protocols are specified in TLA+ [Lam96,Lam97], a formal specification language based on TLA.

Plakal et al. [CHPS99,PSCH98] has also proposed a technique based on Lamport's logical clocks that can be used to reason about cache-coherence protocols. The method associates a counter with each host and provides a time-stamping scheme that totally orders all protocol events. The total order can then be used to verify that the requirements of specific memory models are satisfied.

5 Discussion

Our experience with the proof in the PVS system has convinced us that, at least in a context this complicated, the proof-assistant program needs detailed steering. It is naïve to think one can simply point a theorem prover at the problem and press “start”. Choosing an appropriate invariant requires insight into why the implementation works as it does; and proving the various theorems requires insight into what are efficient strategies for the proof checker.

5.1 Model Checking

A more widely used approach to formal verification is model-checking [CGP99], which uses state enumeration [ID93a,ID93b], sometimes with symbolic techniques [CES86,McM92], to check the correctness of assertions by exhaustively exploring all reachable states of the system. For example, Stern and Dill [SD95] used the Murφ system to check that all reachable states satisfied certain properties attached to protocol specifications. Generally speaking, the major difference among these techniques is the representation of protocol states and the pruning method adopted in the state expansion process. Exponential state explosion has been a serious concern for model checking approaches, although various techniques have been proposed to reduce the state space. For example, Pong and Dubois [PD95] exploited the symmetry and homogeneity of the system states by keeping track of whether zero, one or multiple copies had been cached (this can reduce the state space and also makes the verification independent of the number of processors). Delzanno [Del00] extends this work, keeping a count of the number of processors in each state, and using integer-real relaxation techniques to handle the resulting model using real arithmetic.

The model-checking approach is attractive, since in principle it requires less detailed knowledge of the application being verified, and is more akin to testing. In particular, it can be used for initial sanity checking on small scale examples. Nevertheless, a theorem prover (or “proof assistant”) is likely to be more successful for the verification of sophisticated protocols.

Many model-checking investigations of cache-coherence protocols are confined to verifying that the invariants hold. Some tools, however, are geared towards checking that an implementation is a faithful refinement of a specification. We have used one of these, FDR [Ros97,For], earlier in the present work, to verify a simpler protocol (see [Sto]), showing not only that the implementation was faithful to the specification, but also that it was free from deadlock or livelock. But we had to limit ourselves to considering one particular very simple configuration of caches, with an address space of size 1, and a storable-value space of size 2. Any increase in size caused a “state explosion” rendering the check infeasibly time-consuming.

The present protocol is much more complicated than this earlier one, and the dangers of a state explosion correspondingly greater. Moreover, some of the restrictions to small cases cause problems. The restriction to an address space of unit size is tolerable: we can show that each address is treated independently by both specification and implementation, so that the behavior of each can be viewed as an interleaving of the behaviors for each address considered separately. Thus if model-checking can show that a single-address implementation is faithful to a single-address specification, we can infer that the interleaved multi-address versions will be similarly faithful.

The restriction to a storable-value space of size two is also tolerable. Lazic [Laz99] has shown that in certain circumstances (satisfied in this case) a successful check for a small finite value space is sufficient to imply the correctness of the system for countably infinite spaces.

There remains the restriction to a simple configuration of just one or two caches. Lazic's current research suggests that it might be possible to prove the correctness of a system with arbitrarily many caches by means of an inductive argument, using model-checking to verify the inductive step. We await the outcome of this work with interest.

5.2 Model-Checking or Theorem-Proving

A problem with theorem-proving work in this area is that it is sometimes hard to convince practitioners of its importance. This may be partly because the mathematical nature of its techniques are far removed from the kind of testing more familiar to hardware engineers. This suggests that model-checking, more closely related to testing, has more intuitive appeal. But, as it stands, model-checking is applicable only to comparatively simple systems; and showing that a model-checking investigation suffices to show the correctness of an infinite class of systems is at present a task requiring considerable mathematical subtlety. It is to be hoped that this situation will improve, so that this mathematics can be to a large extent taken for granted. Meanwhile, however, from the point of view of a system designer, the mathematics of the theorem-proving approach may be more closely related to the design task itself, and therefore more likely to shed light on any design inadequacies.

5.3 The Structure of the Proof

The WP micro-protocol was originally designed in several stages [She00]. Although we have constructed the PVS proof for the final design only, it would have been possible to produce a proof of soundness (though not of liveness) structured in accordance with the design stages, as was done in the manual proof. Thus we could have considered a system containing only the imperative rules, with multi-set queues, and proved that it was faithful to the CRF specification. Then we could have made another simulation proof to show that the complete micro-protocol was faithful to that intermediate version.

In fact, however, the motivation for proceeding in this way is not as strong for the machine-assisted proof. The extra layer of simulation needs a good deal of new structure in the proof. It is easier to construct a single soundness proof for the complete protocol. We do this, however, by considering the imperative subset first (thus exploiting the modularity of the design methodology); then we add the directive messages and their operations. The presence of the new messages does not affect the validity of the soundness invariant (which does not refer to them). It is easy to adapt the soundness proof to accommodate these—it is principally a matter of showing that updating the queue by adding or removing a directive message does not affect the invariant. Similarly, specializing the non-deterministic choice in message retrieval to be FIFO has no effect on the soundness argument. The new operations are either equivalent to CRF no-ops (in the case of operations which issue directives) or are special cases of existing voluntary operations. In this latter situation the proofs for the two operations are very similar, and may be transferred by little more than cut-and-paste.

It should be emphasized that no such choice of approach is available for the liveness proof. The intermediate stages of the design do not satisfy the liveness criterion, and any proof of liveness has to be focussed on the final version of the protocol.

5.4 The Abstraction Function

In the manual proof the abstraction function was elegantly defined using the TRS mechanism, in terms of forward and reverse draining. In order to show that this produced a well-defined function, it was necessary to prove that the subsystem of draining rules always terminated and was strongly normalizing. This approach could also have been followed in the machine-assisted proof. The function could be defined[1] as

$$f(s_0) \; = \; \epsilon(\lambda s. \forall b. b_0 = s_0 \land \Box[D]_{wp\_fp} \land WF_{wp\_fp}(D) \; \Rightarrow \; \Diamond\Box b = s),$$

where D defined the transition system for the draining. To prove that f was well defined, it would be necessary to prove that there was a unique state satisfying $\Diamond\Box b = s$. The proof that the behavior eventually achieved a constant state would be a well-founded induction, using the $LATTICE$ rule of TLA and relying on the fairness premise; it would show that the total number of items in all the queues was decreasing, and would therefore need to invoke the finiteness of the system to show that the total number of queued items was itself well defined. To show that the state achieved was unique (and thus that f was a function, and not merely a relation), we might note that all its queues would be empty, and we might therefore define (and prove) an invariant of the state such that there was a unique empty-queue state satisfying it. When defining the invariant, we would have to resist any temptation to characterize the target state by means of an abstraction function, for this would beg the entire question.

The greater formality of the machine-assisted system, however, made all this more trouble than it was worth. It was much simpler to define the function explicitly, rather than in terms of draining operations, at the cost of the intuitive attraction to hardware people of the more operational approach.

[1] The definition of the function ϵ is that for any predicate P, $\epsilon(P)$ is a value satisfying P, provided any such value exists.

In one detail the two functions are actually different. The function defined by draining Cache messages towards the cache; the function defined for PVS is as though those messages suffered reverse draining. This somewhat simplifies the treatment of the reconcile-on-clean operation, by ensuring that it is always simulated by a triplet of CRF operations.

5.5 A Comparison of the Proofs

Compared with a human mathematician, the machine is unforgiving; so employing machine assistance forces the human prover to give systematic attention to every area of the proof. This has advantages and disadvantages. It requires the explicit proof of results which might be thought "obvious" but lead to excessive formal detail in spelling it all out (an example is discussed in the previous section). On the other hand, such a systematic examination can bring to light aspects which were unnoticed before. For example, in our machine-assisted proof of WP, there was a place where we noticed we were relying on the queues being of unbounded length. This did not break the proof—the specification had no boundedness constraint—but it did not accord with the designer's intention, and the protocol definition was revised to avoid it.

5.6 Summary

It is difficult to gain confidence in the correctness of a complex protocol without some formal reasoning. We think that the first step in designing a robust protocol is to follow a methodology that keeps the correctness issue in the center of the whole design process. The Imperative-&-Directive methodology is one way of achieving this goal: it separates soundness and liveness, and lets the designer refine a simpler protocol into a more complex one by introducing pragmatic concerns one step at a time. But even after using such a methodology, if the resulting protocol is large or complex one needs to go the extra mile of using automatic tools for verification. Model checkers are unlikely to eliminate all doubts about correctness, because to avoid the state-space explosion one is forced to apply the model checker to a simpler or smaller version of the system. We think semi-automatic verification of the final protocol is the most promising approach to gain confidence in the correctness of a complex protocol; and even semi-automatic verification is possible only after the user has considerable insight into how the protocol works.

References

[ABM93] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy Caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.

[ADH+99] Homayoon Akhiani, Damien Doligez, Paul Harter, Leslie Lamport, Joshua Scheid, Mark Tuttle, and Yuan Yu. Cache coherence verification with TLA+. In *World Congress on Formal Methods in the Development of Computing Systems, Industrial Panel*, Toulouse, France, September 1999.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, pages 66–76, December 1996.

[Arc87] James K. Archibald. The Cache Coherence Problem in Shared-Memory Multiprocessors. PhD Dissertation, Department of Computer Science, University of Washington, February 1987.

[Bro90] Geoffrey M. Brown. Asynchronous Multicaches. *Distributed Computing*, 4:31–36, 1990.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[CHPS99] Anne E. Condon, Mark D. Hill, Manoj Plakal, and Daniel J. Sorin. Using Lamport Clocks to Reason About Relaxed Memory Models. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, 1999.

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995. Available, with specification files, at `http://www.csl.sri.com/wift-tutorial.html`.

[Del00] Giorgio Delzanno. Automatic Verification of Parameterized Cache Coherence Protocols. Technical Report DISI-TR-00-1, DISI, University of Genoa, January 2000. Available at `http://www.disi.unige.it/person/DelzannoG/papers`.

[For] Formal Systems (Europe) Limited. Fdr2. Web site. See `http://www.formal.demon.co.uk/FDR2.html`.

[HQR99] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Verifying Sequential Consistency on Shared-Memory Multiprocessor Systems. In *Proceedings of the 11th International Conference on Computer-aided Verification (CAV)*, pages 301–315. Springer-Verlag, 1999. Lecture Notes in Computer Science 1633.

[ID93a] C.N. Ip and D.L. Dill. Better Verification Through Symmetry. In *Proceedings of the 11th International Symposium on Computer Hardware Description Languages and Their Applications*, pages 87–100, April 1993.

[ID93b] C.N. Ip and D.L. Dill. Efficient Verification of Symmetric Concurrent Systems. In *International Conference on Computer Design: VLSI in Computers and Processors*, October 1993.

[KPS93] David R. Kaeli, Nancy K. Perugini, and Janice M. Stone. Literature Survey of Memory Consistency Models. Research Report 18843 (82385), IBM Research Devision, 1993.

[Lam93] Leslie Lamport. How to write a proof. In *Global Analysis in Modern Mathematics*, pages 311–321. Publish or Perish, Houston, Texas, U.S.A., February 1993. A symposium in honor of Richard Palais' sixtieth birthday.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam96] Leslie Lamport. The Module Structure of TLA+. Technical Note 1996-002a, Compaq Systems Research Center, September 1996.

[Lam97] Leslie Lamport. The Operators of TLA+. Technical Note 1997-006a, Compaq Systems Research Center, June 1997.

[Laz99] Ranko Lazic. *A Semantic Study of Data Independence with Applications to Model Checking.* PhD thesis, Oxford University Computing Laboratory, 1999.

[McM92] K.L. McMillan. Symbolic Model Checking: An Approach to the State Explosion Problem. PhD Dissertation, Carnegie Mellon University, May 1992.

[PD95] Fong Pong and Michel Dubois. A New Approach for the Verification of Cache Coherence Protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6, August 1995.

[PD96a] Seungjoon Park and David L. Dill. Protocol Verification by Aggregation of Distributed Transactions. In *International Conference on Computer-Aided Verification*, July 1996.

[PD96b] Seungjoon Park and David L. Dill. Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions. In *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[PD96c] Fong Pong and Michel Dubois. Formal Verification of Delayed Consistency Protocols. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[PNAD95] Fong Pong, Andreas Nowatzyk, Gunes Aybay, and Michel Dubois. Verifying Distributed Directory-based Cache Coherence Protocols: S3.mp, a Case Study. In *Proceedings of the European Conference on Parallel Computing*, 1995.

[PSCH98] Manoj Plakal, Daniel J. Sorin, Anne E. Condon, and Mark D. Hill. Lamport Clocks: Verifying a Directory Cache-Coherence Protocol. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[Ros97] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1997.

[SAR99a] Xiaowei Shen, Arvind, and Larry Rodolph. CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems. In *Proceedings of the 13th ACM International Conference on Supercomputing*, June 1999.

[SAR99b] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers. In *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[SD95] Ulrich Stern and David L. Dill. Automatic Verification of the SCI Cache Coherence Protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

[She00] Xiaowei Shen. *Design and Verification of Adaptive Cache Coherence Protocols.* PhD thesis, Massachusetts Institute of Technology, February 2000.

[Sto] Joseph E. Stoy. Web sites concerning Cachet, TLA in PVS, and cache protocol verification using FDR. See `http://web.comlab.ox.ac.uk/oucl/work/joe.stoy/`.