Model Checking a Cache Coherence Protocol of a Java DSM Implementation*

Jun Pang[1], Wan Fokkink[2,3], Rutger Hofman[2], Ronald Veldema[4]

[1] *Carl von Ossietzky Universität Oldenburg, Department für Informatik, Ammerländer Heerstraße 114-118, 26111 Oldenburg, Germany*
email: jun.pang@informatik.uni-oldenburg.de
[2] *Vrije Universiteit Amsterdam, Afdeling Informatica, Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands*
email: {wanf,rutger}@cs.vu.nl
[3] *CWI, Software Engineering Department, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands*
[4] *Friedrich-Alexander-Universität Erlangen-Nürnberg, Institut für Informatik, Martensstr. 3, 91058 Erlangen, Germany*
email: veldema@cs.fau.de

Abstract

Jackal is a fine-grained distributed shared memory implementation of the Java programming language. It aims to implement Java's memory model and allows multithreaded Java programs to run unmodified on a distributed memory system. It employs a multiple-writer cache coherence protocol. In this paper, we report on our analysis of this protocol. We present its formal specification in μCRL, and discuss the abstractions that were made to avoid state explosion. Requirements were formulated and model checked with respect to several configurations. Our analysis revealed two errors in the implementation.

Key words: formal specification, model checking, cache coherence protocols, Java memory model, μCRL

* This is the full version of an extended abstract that appeared in the Proceedings of the 8th Workshop on Formal Methods for Parallel Programming: Theory and Applications, IEEE Computer Society Press, 2003. The research is partly supported by the Dutch Technology Foundation STW under the project CES5008.

1 Introduction

Shared memory is an attractive programming model for interprocess communication and synchronization in multiprocessor computations. In the past decade, a popular research topic has been the design of systems to provide a shared memory abstraction of physically distributed memory machines. This abstraction, known as Distributed Shared Memory (DSM), has been implemented both in software (e.g., to provide the shared memory programming model on networks of workstations) and in hardware (e.g., using cache coherence protocols to support shared memory across physically distributed main memories).

Multithreading is a programming paradigm for implementing parallel applications on shared memory multiprocessors. It is widely used as a program structuring mechanism and to support efficient parallel computations. It can improve efficiency and performance in an application program by introducing concurrency or parallelism. Java is one of the few programming languages supporting multithreaded programming at the language level.

The Java memory model (JMM) [11] prescribes certain abstract rules that any implementation of Java multithreading must follow. Jackal [30] is a fine-grained DSM implementation of the Java programming language. It aims to implement the JMM and allows multithreaded Java programs to run unmodified on DSM. It employs a self-invalidation based, multiple-writer cache coherence protocol, which allows processors to cache a region (i.e., a contiguous block of memory) created on another processor (i.e., the region's home). All threads on one processor share one copy of a cached region. The region's home and the caching processors store this copy at the same virtual address. A cached region copy remains valid for a particular thread until that thread reaches a synchronization point. In Jackal, several optimizations [29,30] improve both sequential and parallel application performance. Among them, *automatic home node migration* reduces the amount of synchronization, by automatically appointing as the region's home a processor that is likely to access this region often.

μCRL [13] is a formal language for specifying protocols and distributed systems in an algebraic style. To each μCRL specification there belongs a *labeled transition system* (LTS), in which the edges between states are labeled with actions. The μCRL tool set [3] can be used in combination with the Construction and Analysis of Distributed Processes toolbox (CADP) [10] to generate, visualize and analyze this LTS. For example, one can detect deadlocks and livelocks, or check the validity of temporal logic formulas [7].

In this paper, we present our formal analysis of a cache coherence protocol

for Jackal using the μCRL tool set and CADP. A μCRL specification of the protocol (including automatic home node migration) was extracted from an informal (C language-like) description of the protocol. To avoid state explosion, we made certain abstractions with respect to the protocol's implementation. Requirements were verified by the μCRL tool set together with CADP. Our analysis revealed many inconsistencies between the description and the implementation. We found two errors in the description (see Section 6.2). The developers of the protocol checked the two errors and found their way in the implementation. Both errors can happen when a thread is writing to a region from remote (i.e., the thread does not run on the home of the region). During the thread's waiting for a lock or an up-to-date copy of the region, the home node may migrate to the thread's processor, so that the thread actually accesses the region at home. The first error resulted into a deadlock. The second error was found when model checking the property of only one home for each region. After updating our formal specification, the requirements were successfully checked on several configurations. Our solutions to the errors were adapted in the implementation of the protocol. The interested readers can find the Jackal distribution (version Beta 1.0) at `http://www2.informatik.uni-erlangen.de/Personen/veldema/privat/jackal_distribution.html`.

We summarize our contributions as follows:

- We developed a formal specification of a cache coherence protocol for a Java DSM implementation.
- We found errors both in the description and the implementation, which helped to improve the design and implementation of this protocol.
- This is the most complicated cache coherence protocol to date that has been formally specified and analyzed using model checking.

Outline of the paper. The remainder of this paper is structured as follows. In Section 2, we discuss related work on analyzing the JMM or its replacement proposal and verifying cache coherence protocols using formal techniques. An informal description of the JMM is given in Section 3. Section 4 presents the Jackal system and its cache coherence protocol. In Section 5, μCRL specifications for each component of the protocol are given. Section 6 focuses on our model checking analysis in μCRL. Conclusions are presented in Section 7.

2 Related Work

The use of formal methods to analyze the JMM is an active research topic. In [26], the authors developed a formal executable specification of the JMM [11]. Their specification is operational and uses guarded commands. This model

can be used to verify popular software construction idioms for multithreaded Java. In [31], the Murϕ verification system was applied to study the CRF memory mode [20]. A suite of test programs was designed to reveal pivotal properties of the model. This approach was also applied to Manson and Pugh's proposal [21] by the same authors [32]. Two proofs of the correctness for Cachet [27], an adaptive cache coherence protocol, were presented in [28]. Each proof demonstrates soundness (conformance to the CRF memory model) 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 the theorem prover PVS. Similar to [31,32], we use formal specification and model checking techniques. A major difference is that we analyzed a cache coherence protocol within a Java DSM system that is already implemented and far more complicated than the abstract memory models analyzed in [26,28,31,32]. Our analysis helped to improve the actual design and implementation of the protocol.

Our work is also related to the verification of cache coherence protocols. Formal methods have been successfully applied in the automated verification of cache coherence on sequentially consistent systems [18], e.g. [6,8,16]. Coherence in shared memory multiprocessors is much more difficult to verify. Recently, Pong and Dubois [24] used their state-based tool for the verification of a delayed protocol [9], which is an aggressive protocol for relaxed memory models. We encountered the same difficulties as [24], such as that the hardware to model is complex, and that the properties of the protocol are hard to formulate. Differences between our work and [24] are: we analyzed a protocol designed for *distributed* shared memory machines; and the protocol supports *multithreaded* Java programs, which makes matters more complicated.

3 Java Memory Model

The Java language supports multithreaded programming, where threads can interact among themselves via read/write of shared data. The JMM prescribes certain abstract rules that any implementation of Java multithreading must follow. We briefly present the JMM as given in [11].

We assume a multiprocessor setting, where each processor owns a collection of regions, which are contiguous blocks of memory that contain either a single object or a fixed-size partition of an array. Each thread runs on exactly one processor, and can only access the regions that reside at its processor.

The JMM allows each thread to cache regions in its working memory, which keeps its own working copy of the regions. A thread can only manipulate the regions in its working memory, which is inaccessible to other threads. The

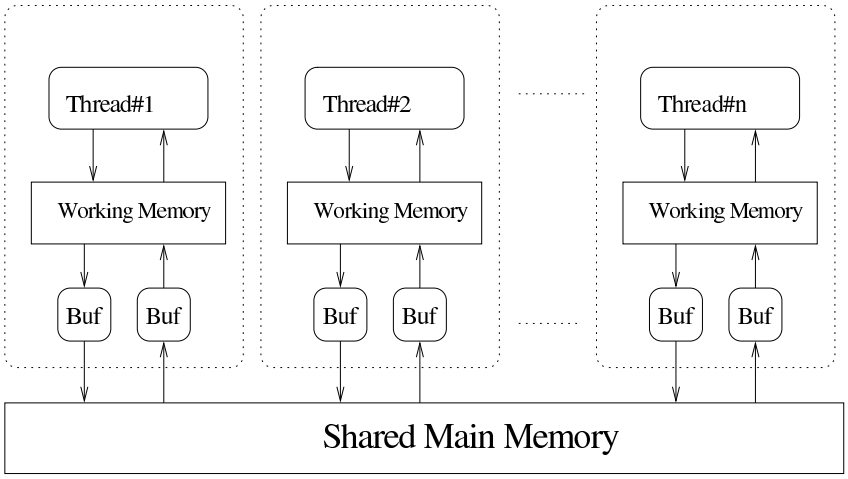

Fig. 1. JMM memory system

working memories are caches of a single main memory, which is shared by all threads. Main memory keeps the main copy of every region. This memory structure is depicted in Fig. 1. A thread's working memory must be flushed to main memory at each synchronization point, which is a lock (unlock) operation that corresponds to the entry (exit) of a synchronized block of code.

The JMM defines a set of actions that a thread may use to interact with memory. Each thread invokes four actions: use, assign, lock and unlock. Four more actions, read, load, store and write, are invoked in case of a multithreaded implementation, following the temporal ordering constraints in the JMM ([11, Chapter 17]). The meaning of each action is as follows:

(1) *use*: Read from the working memory of a region.
(2) *assign*: Write into the working memory of a region.
(3) *read*: Initiate reading from the main memory of a region.
(4) *load*: Complete reading from the main memory of a region.
(5) *store*: Initiate writing the working memory into the main memory of a region.
(6) *write*: Complete writing the working memory into the main memory of a region.
(7) *lock*: Get the values in the main memory transferred to a thread's working memory through *read* and *load* actions.
(8) *unlock*: Put the values a thread holds in its working memory back to the main memory through *store* and *write* actions.

Since threads can access regions from different processors, a region's home acts as the region's lock manager. To acquire a lock, a thread sends a lock request message to the lock manager and waits. If the lock is available, the lock manager replies with a notify message; otherwise, the thread needs to wait for the lock to be released. To unlock, the lock holder sends an unlock message to the lock manager.

There were several problems in the original JMM [11]. A detailed discussion of the various problems in the original JMM can be found at `http://www.cs.umd.edu/~pugh/java/memoryModel/`. Two replacement semantics for the JMM have been proposed, by Manson and Pugh [21] and by Maessen, Arvind and Shen [20]. A revision of the JMM, called JSR 133, was released in September 2004. Jackal, which will be described in the next section, implements the memory model in JSR 133.

4 Jackal DSM System

Jackal [30] is a fine-grained DSM implementation of the Java programming language. It allows multithreaded Java programs to run unmodified on a distributed memory system. Its runtime system implements a self-invalidation based, multiple-writer cache coherence protocol for regions.

The Jackal memory model allows processors to cache a region created on another processor. All threads on one processor share one copy of a cached region. The region's home and the caching processors all store this copy at the same virtual address. The protocol is based on self-invalidation, which means the cached copy of a region remains valid until the thread itself invalidates the copy, which occurs whenever it reaches a synchronization point. Jackal combines features of HLRC [33] and TreadMarks [17]. As in HLRC, modifications are flushed to a home node; as in TreadMarks, twinning and diffing is used to allow concurrent writes to shared data. Unlike TreadMarks, Jackal uses software access checks inserted before each object usage to detect non-local or stable data. Several optimizations were made to improve both sequential and parallel application performance [29,30].

Fig. 2 shows the various components and their interactions in Jackal's cache coherence protocol. P1,P2 are identities of processors, and T1,T2,T3,T4 identities of threads. This picture will be explained in the remainder of this section.

4.1 Address space management

Jackal stores all regions in a single, shared virtual address space. Each region occupies the same virtual address range on all processors that store a copy of the region. Regions are named and accessed through their virtual address. Each processor owns part of the physical memory and creates objects and arrays in its own part. In this way, each processor can allocate objects without synchronizing with other processors. When a thread wishes to access a region created by another processor, it must potentially allocate physical memory

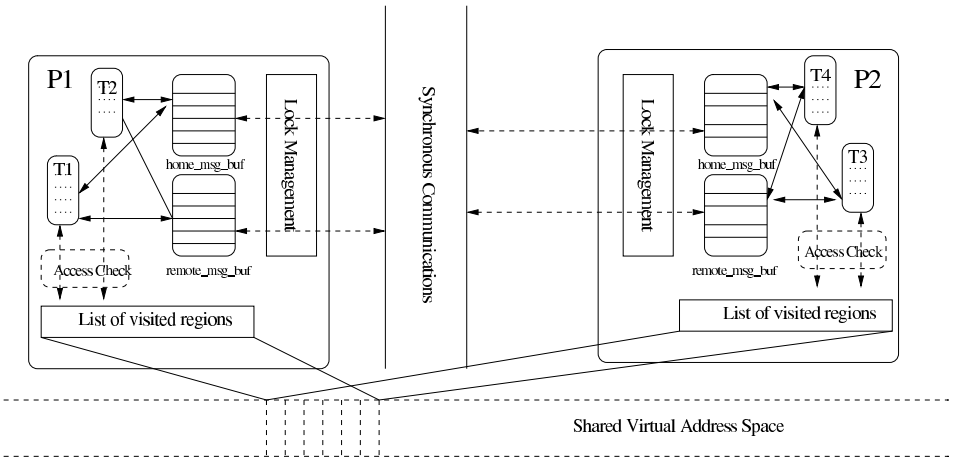

Fig. 2. Components in the Jackal architecture

for the virtual memory pages in which the object is stored, and retrieve an up-to-date copy of the region from its home node. If a processor runs out of free physical memory, it initiates a global garbage collection that frees both Java objects and physical memory pages.

To implement self-invalidation, each thread keeps track of the regions it accessed and cached since its last synchronization point. The data structure storing this information is called a *flush list*. At a synchronization point, all regions in the thread's flush list are invalidated for that thread, by writing *diffs* back to their home nodes. A diff contains the difference between a region's object data and its twin data.

Jackal performs a software access check for every use of a region. The access check determines whether the region referenced by a given pointer contains a valid local copy. Whenever an access check detects an invalid local copy, the runtime system contacts the region's home. It asks the home node for a copy of the region and stores this copy at the same virtual address as at the home node. The thread requesting the region receives a pointer to that region and adds it to its flush list. This flush list is similar to the working memory in the JMM.

4.2 Automatic home node migration

Java programs do not indicate which locks protect which data items. This makes it difficult to combine data and synchronization traffic. Jackal may have to communicate multiple times to acquire a lock, to access the data protected by the lock and to release the lock. The home of a region acts as the manager of the lock (see Section 4.5). To decrease synchronization traffic, automatic home node migration has been implemented in Jackal. It means that Jackal

may automatically appoint as the region's home a processor that is likely to access this region often. This optimization is triggered during the following two cases.

(1) A thread writes to a region, and an access check detects an invalid local copy; the runtime system contacts the region's home, and finds that the thread's processor is the only one from which threads are writing to this region. Then the home of this region migrates to the thread's processor.
(2) A thread flushes at a synchronization point, and there is only one processor left from which threads are writing to some region. Then the home of this region migrates to this processor.

Jackal can detect these situations at runtime, and thus reduce synchronization traffic. Automatic home node migration complicates meeting the requirements in Section 6.1.

4.3 Regions

In Jackal, a region contains the following information.

(1) Location: A processor's identity, denoting at which node the region (or a copy) is.
(2) Home: A processor's identity, denoting the home node for this region.
(3) State: A region can evolve into four kinds of states. When no thread uses this region, the state of the region is *Unused*; if a region is only used by threads on its home node, its state is *Homeonly*; when this region is only read by threads, its state is *Readonly*; in all other cases, the state of a region is *Shared*.
(4) WriterList: A list of processors' identities containing threads that are writing or recently wrote to this region. It is only maintained at the home node.
(5) ReaderList: A list of processors' identities containing threads that are reading or recently read this region. It is only maintained at the home node.
(6) Object data: An array of bytes.
(7) Twin data: An array of bytes. It is a copy of the object data for diffing at non-home nodes; initially it is null.
(8) Localthreads: A natural number, the number of threads accessing this region at the location of the region.

4.4 Messages

Four types of messages can be delivered to a processor.

(1) *Data Request*: This message is sent when a thread starts writing to a region from remote. When a processor gets this message, and it is the home of the region, it adds the thread's processor into the WriterList of the region and sends back an up-to-date copy of the region to the thread's processor by a *Data Return* message. If it is not the home of the region (meaning that the region migrated its home in the meantime), it forwards the *Data Request* message to the region's new home.
(2) *Data Return*: This message is received by a processor when an up-to-date copy of a region has arrived. The processor updates the object and twin data of the region. Moreover, this message can also be a home node migration message. If this is the case, then the processor becomes the home of this region, and starts maintaining the WriterList and the state of the region.
(3) *Flush Request*: This message is sent when a thread flushes from remote. When a processor gets this request, and it is the home of the region, it mat remove the thread's processor from the WriterList of the region, if the thread was the only one on its processor that was writing to this region. Moreover, it may send a home node migration message to a new home of this region (by a *Region Sponmigrate* message); this happens when there is only one processor left in the region's WriterList. When it is not the home of the region, it simply forwards the *Flush Request* message to the region's new home.
(4) *Region Sponmigrate*: When a processor gets this message, it becomes the home of the region in question.

Each processor maintains two message queues to store incoming messages. The *HomeQueue* is designed to buffer messages containing a request, while the *RemoteQueue* buffers messages containing a reply.

4.5 Locks

Locks guarantee exclusivity when threads write to or flush a region. A processor acts as the lock manager of its regions and region copies. There are five types of locks for each processor: *homequeue*, *remotequeue*, *server*, *fault* and *flush*.

The homequeue lock and remotequeue lock are needed to make sure that the handling of a popped message from a HomeQueue or a RemoteQueue by its processor is completed before the next message is popped from the queue.

Jackal's cache coherence protocol allows writes to a region at home and from remote to happen concurrently. The server lock, fault lock and flush lock ensure exclusivity between threads at a processor. The server lock and flush lock must be mutually exclusive for the home of a region, to protect the integrity of region data values and other region information; likewise, the fault lock and flush lock must be mutually exclusive for non-home nodes of a region. When a thread writes at home or from remote, the server lock or fault lock of the thread's processor is needed, respectively. When a thread flushes, the flush lock of its processor is needed. When a lock is released, the lock manager notifies a thread according to the following rules. They are applied in the given order.

- If both the flush and the homequeue lock are available, and there are threads waiting for the homequeue lock or the server lock, one of those threads is notified.
- If both the flush and the remotequeue lock are available, and there are threads waiting for the remotequeue lock, one of those threads is notified.
- If the flush, homequeue and remotequeue lock are available, no threads waiting for either homequeue lock or remotequeue lock, and there are threads waiting for the flush lock, one of those threads is notified.
- If both the flush and the homequeue lock are available, and no threads are waiting for either the homequeue or the remotequeue lock, and there are threads waiting for the fault lock, one of those threads is notified.
- In all other cases, no waiting thread is notified.

4.6 Other features

To improve performance, a source-level global optimization *object-graph aggregation*, and runtime optimization *adaptive lazy flushing*, are implemented in Jackal. These features are not included in our μCRL specification of Jackal's cache coherence protocol, which will be described in Section 5.

The Jackal compiler can detect situations where an access to some object (called root object) is always followed by accesses to subobjects. In that case, the system views the root object and the subobjects as an object graph. Jackal attempts to aggregate all access checks on objects in such a graph into a single access check on the graph's root object. If this check fails, the entire object graph is fetched, which can reduce the number of network round-trips. We did not model object-graph aggregation, because we modeled memory at a rather abstract level.

The Jackal cache coherence protocol invalidates all data in a thread's working memory at each synchronization point. That is, the protocol exactly follows the specification of the JMM, which potentially leads to much interprocessor

communication. Due to adaptive lazy flushing, it is not necessary to invalidate and flush a region that is accessed by only a single processor or that is only read by its accessing threads. We did not model adaptive lazy flushing, since it is not relevant for the requirements that we formulated.

5 μCRL Specification of the Protocol

In this section, we present a formal specification of Jackal's cache coherence protocol in μCRL and verify some general requirements at the behavioral level.

5.1 μCRL

μCRL is a language for specifying distributed systems and protocols in an algebraic style. It is based on the *process algebra* ACP [1] extended with equational abstract data types [19]. The syntax and semantics of μCRL are given in [13,15]. A μCRL specification consists of two parts: one part specifies the data types, the other part specifies the processes.

The data part contains equational specifications; one can declare sorts and functions working upon these sorts, and describe the meaning of these functions by equations. Since booleans are used in the conditional construct of process descriptions, the sort *Bool* must be included in every μCRL specification. Besides the declaration of the sort *Bool*, it is also obligatory that T (true) and F (false) are declared in every specification and that $\mathsf{T} \neq \mathsf{F}$.

Processes are represented by process terms. Process terms consist of action names and recursion variables with zero or more data parameters, combined with process-algebraic operators. Actions and recursion variables carry zero or more data parameters. Intuitively, an action can execute itself, after which it terminates successfully. There are two predefined actions: δ represents deadlock, τ the internal action. $\mathsf{p.q}$ denotes sequential composition, it first executes p and then q. $\mathsf{p{+}q}$ denotes non-deterministic choice, meaning that it can behave as p or q. Summation $\sum_{\mathsf{d:D}} \mathsf{p(d)}$ provides the possibly infinite choice over a data type D. The conditional construct $\mathsf{p} \lhd \mathsf{b} \rhd \mathsf{q}$, with b a boolean data term, behaves as p if b and as q if not b.

For example, let $\mathsf{S:Natural} \rightarrow \mathsf{Natural}$ be the successor function on natural numbers. Given the recursive equation $\mathsf{X(n:Natural)=a(n).X(S(n))}$, the process $\mathsf{X(0)}$ performs the sequence of actions $\mathsf{a(0).a(S(0)).a(S(S(0))).}\cdots$. And given the recursive equation $\mathsf{Y}=\sum_{\mathsf{n:Natural}} \mathsf{a(n).Y}$, the process Y can perform any action $\mathsf{a(n)}$ and return to the process Y.

Parallel composition p||q interleaves the actions of p and q; moreover, actions from p and q may synchronize into a communication action, when this is explicitly allowed by a predefined communication function. In this paper we take as naming convention that for each send action s_name there is a receive action r_name, and that they communicate to the action c_name. Two actions can only synchronize if their data parameters are the same, which means that communication can be used to capture data transfer from one process to another. If two actions are able to synchronize, then in general we only want these actions to occur in communication with each other, and not on their own. This can be enforced by the encapsulation operator $\partial_H(\mathsf{p})$, which renames all occurrences in p of actions from the set H into δ. Additionally, the hiding operator $\tau_I(\mathsf{p})$ turns all occurrences in p of actions from the set I into τ.

The μCRL tool set [3] is a collection of tools for analyzing and manipulating μCRL specifications, based on term rewriting and linearization techniques [14]. The μCRL tool set, together with the CADP tool set [10], which acts as a back-end for the μCRL tool set, features visualization, simulation, LTS generation, model checking, theorem proving and statebit hashing capabilities. μCRL and its tool set have been successfully used to analyze a wide range of protocols and distributed systems (e.g., [2,12,23]).

5.2 Specification of the protocol

The starting point of verifying a system with μCRL is to give an algebraic specification. This generally involves identifying the key behaviors of the protocol components and understanding the way how each component communicates with others.

The cache coherence protocol in Jackal is more complex than an interleaved execution of the threads, where each thread executes in program order. The permitted set of execution traces is a superset of the simple interleaved execution of the individual threads. Furthermore, the μCRL specification is an exhaustive nondeterministic description of the cache coherence protocol. This may lead to state explosion. To deal with this problem, we made some abstractions of each component. In the following discussion, we present the μCRL specification of each component, together with the abstractions we made. For the sake of presentation, we only give parts of the specification to illuminate the crucial points, and omit the specification of data types. The complete specification can be found in Appendix A, which already includes our solutions to the found problems in Section 6 and the additional actions for checking the requirements.

Our model of the cache coherence protocol is a parallel composition of the threads, processors, regions, lock managers and message queues. The complete μCRL specification of this protocol consists of around 1000 lines.

5.2.1 Assertions from the developers

The developers added many assertions into the description and required that the protocol should not violate any of them. The assertions are modeled as a part of the μCRL specification. They can be divided into two classes: *order assertions* and *preconditions*.

- Order assertions: This class of assertions imposes a certain order on the usage of the system resources. For example, when a thread performs an action on a region, the corresponding lock should already be held by the thread. Order assertions are modeled in the μCRL specification by imposing a certain order on the execution of actions. For example, the behavior of a thread is modeled like this: only after a thread has taken the server lock of the thread's processor (either immediately by an action r_nodelay_serverwait, or after a delay by an action r_delay_serverwait), the thread can access a region at home.
- Preconditions: This class of assertions requires that only when a certain precondition is satisfied, the description after it can be executed. For example, only under certain conditions (see Section 4.2) the home of the region automatically migrates. Preconditions are modeled in the μCRL specification as boolean terms in conditional expressions.

5.2.2 Regions

In μCRL, each region is modeled as a separate component. It consists of an identity rid, a processor identity pid indicating where the region is, and its information r meaning its home, state, WriterList, and the number of local threads that are writing to this region.

We did not model object and twin data, since they are not relevant to our requirements for the protocol (see Section 6.1). So in our model a thread cannot write any value to a region. Still, when a thread flushes a region from remote, a message (without a diff) is sent back to the home of this region to release the obtained lock (see Section 5.2.3).

The behavior of reading from a region is part of the behavior of writing to a region, in the sense that if needed an up-to-date copy of the region has to be obtained. On top of this, in case of writing, coherence of the region's data is at stake, and at a synchronization point the adapted region must be flushed back to main memory. Thus writing is far more critical for the correctness of the

Table 1
Specification of a region

```
% pid indicates where the region is; rid is the region's identity;
% r contains the region's information.
Region(pid:ProcessorId, rid: RegionId, r:RegionInfo) =
% Communication with threads.
∑tid:ThreadId r_threadrequestinfo(tid,pid,rid,r).
        (r_threadnorefresh(tid,pid,rid).Region(pid,rid,r)
        +∑r':RegionInfo r_threadrefresh(tid,pid,rid,r').Region(pid,rid,r'))
% Communication with processors.
+ r_processorrequestinfo(pid,rid,r).
(r_processornorefresh(pid,rid).Region(pid,rid,r)
+∑r':RegionInfo r_processorrefresh(pid,rid,r').Region(pid,rid,r'))
```

protocol than reading. Therefore we abstracted away from the read action of threads. So a region has only two states; we kept the *Unused* state, while the other three states are covered by a single state *Used*. Furthermore, a region only needs to maintain the WriterList.

The μCRL specification of a region is presented in Table 1. We use synchronized actions to ensure that during an access of a thread to a region, no other threads can change the information of this region. This can ensure that a thread or a processor gets the latest status of the region. For instance, in Table 3, a thread gets the information of a region by performing a send action s_threadrequestinfo, and no access to this region by another thread is allowed until the thread executes a send action s_threadnorefresh (if it changed nothing) or s_threadrefresh (if it updated some information of the region). The corresponding actions, which synchronize with the three aforementioned actions (i.e., r_threadrequestinfo, r_threadnorefresh and r_threadrefresh) occur in Table 1. Likewise for processors; see Tables 7 and 8 for occurrences of the actions that synchronize with r_processorrequestinfo, r_processornorefresh and r_processorrefresh in Table 1.

5.2.3 Threads

In the μCRL specification, each thread is modeled as a separate process with a unique identity tid (see Table 2). It contains a parameter pid to indicate on which processor the thread executes. Each thread maintains a *FlushList* of identities of regions that it is writing or recently wrote to, to remember that they need to be flushed in the future. It can perform actions write(tid,rid) to start writing to a region with identity rid, and flush(tid) to start invalidating all the regions in its FlushList, if its FlushList is not empty.

Table 2
Specification of a thread starting to write or flush

```
Thread(tid:ThreadId, pid:ProcessorId, FlushList:RegionIdSet) =
∑rid:RegionIdwrite(tid,rid).ThreadWrite(tid,pid,rid,FlushList)
+
flush(tid).ThreadInvalidate(tid,pid,FlushList) ◁ not(empty(FlushList)) ▷ δ
```

Table 3
Specification of a thread starting to write to a region

```
ThreadWrite(tid:ThreadId, pid:ProcessorId, rid:RegionId, FlushList:RegionIdSet) =
% The thread is already writing to the region.
% writeover(tid, rid) will be added here for our verification purpose.
Thread(tid,pid,FlushList)
◁ test(rid,FlushList) ▷
% The thread must obtain an up-to-date copy of the region.
∑r:RegionInfo s_threadrequestinfo(tid,pid,rid,r).
% Write to the region at home if pid is the home of the region.
        (s_threadnorefresh(tid,pid,rid).WriteHome(tid,pid,rid,insert(rid,FlushList))
        ◁ eq(gethome(r),pid) ▷
% Otherwise, write to the region from remote.
        s_threadnorefresh(tid,pid,rid).WriteRemote(tid,pid,rid,insert(rid,FlushList)))
```

When a thread starts writing to a region (see Table 3), the corresponding access check determines whether the thread is already writing to the region (test(rid,FlushList)). If not, then first an up-to-date copy of the region must be obtained from the region's home. This access check will also determine whether the thread writes to this region at home or from remote, depending on whether the region's home is the thread's processor (eq(gethome(r),pid)). In the first case the server lock is needed if the thread runs on the region's home; in the second case the fault lock of the thread's processor must be acquired.

Table 4 specifies a thread starting to write to a region from remote. The fault lock is acquired (s_require_faultlock) from the thread's processor. When the fault lock is granted, either immediately (r_nodelay_faultwait) or after the lock has been released (r_delay_faultwait) by some other thread, the thread sends a *Data Request* message (s_thread_datarequest) to the home of the region (by s_threadrequestinfo it gets to know the home). The thread waits until it receives a message (r_signal), which means that the region (local copy, located at the thread's processor) has become consistent with the region at the region's home. Then the thread gets the information of the up-to-date copy of the region (s_threadrequestinfo), it continues writing to/updating the region (s_threadrefresh), increases the local thread number by one, and finally releases

Table 4
Specification of a thread writing to a region from remote

```
WriteRemote(tid:ThreadId, pid:ProcessorId, rid:RegionId, FlushList:RegionIdSet) =
% Thread writes from remote, requires a fault lock,
% and asks for a fresh copy of the region.
s_require_faultlock(pid).
(r_nodelay_faultwait(pid)+r_delay_faultwait(pid)).
% Ask for a fresh copy of the region.
∑r:RegionInfo s_threadrequestinfo(tid,pid,rid,r).
s_thread_datarequest(tid,pid,gethome(r),rid).
s_threadnorefresh(tid,pid,rid).
% Copy arrives, the thread is notified.
∑pid':ProcessorId r_signal(tid,pid',rid).
        ∑r':RegionInfo s_threadrequestinfo(tid,pid,rid,r').
        s_threadrefresh(tid,pid,rid,increaselocalthreads(r')).
        s_free_faultlock(pid).
% writeover(tid, rid) will be added here for our verification purpose.
        Thread(tid,pid,FlushList)
```

the lock by sending an unlock message (s_free_faultlock) to the lock manager (see Table 11).

The specification of a thread writing to a region at home, which is omitted here, is similar to the one for a thread writing to a region from remote. Instead of a fault lock, the thread needs to acquire a server lock (s_require_serverlock). Once a server lock is granted, either immediately (r_nodelay_serverwait) or after the lock has been released (r_delay_serverwait), the thread gets the information of the region, and updates the region. Finally it releases the server lock by sending an unlock message (s_free_serverlock) to the lock manager.

When a thread invalidates (see Table 5), it empties its FlushList by flushing and removing each region's identity in its FlushList. Similar to the case when a thread writes to a region, a thread can flush a region at home or from remote, depending on whether the region's home is the thread's processor. The flush lock of the thread's processor is acquired before invalidating (s_require_flushlock). If the thread invalidates a region from remote (see Table 6), it sends a *Flush Request* message to the home of the region (s_thread_flushrequest). If the thread is the only local thread which was accessing the region, the home of the region will need to remove the thread's processor from the region's WriterList. This information is forwarded to the home of the region by setting the last boolean variable in the *Flush Request* message to true. Otherwise, the boolean variable is set to false. The thread updates the information of the local copy of the region by decreasing the parameter *Localthreads* for this region by one. The thread releases the flush lock

Table 5
Specification of a thread invalidating

```
ThreadInvalidate(tid:ThreadId, pid:ProcessorId, FlushList:RegionIdSet) =
% If FlushList is empty, do nothing.
% flushover(tid) will be added here for our verification purpose.
Thread(tid,pid,FlushList)
◁ empty(FlushList) ▷
% Thread requires a flush lock.
s_require_flushlock(pid).
(r_nodelay_flushwait(pid)+r_delay_flushwait(pid)).
% The thread gets the status of the first region in the FlushList.
∑r:RegionInfo s_threadrequestinfo(tid,pid,head(FlushList),r).
% Invalidate at home.
        (FlushHome(tid,pid,head(FlushList),tail(FlushList),r)
        ◁ eq(gethome(r),pid) ▷
% Otherwise, invalidate from remote.
        FlushRemote(tid,pid,head(FlushList),tail(FlushList),r))
```

(s_free_flushlock), and waits until it gets a message (r_signal) indicating that the home of the region has finished with the *Flush Request* message. Then the thread continues to flush other regions in its FlushList, if this list is not yet empty. In the actual protocol, the *Flush Request* message also contains a diff with the difference between the region's object and twin data; we recall that the μCRL specification abstracts away from object and twin data.

The specification of a thread flushing a region at home, which is omitted here, is similar to the one for a thread flushing a region from remote. The difference is that the home of the region also takes charge of automatic home node migration (see Section 4.2). The thread updates the information of the region by decreasing *Localthreads* for this region by one. If the thread is the only local thread which was accessing the region (i.e., if *Localthreads* for this region becomes zero), then the thread's processor is removed from the region's WriterList. If there is only one other processor left in the region's WriterList, the home of the region will migrate to that processor.

Note that the corresponding parts of ThreadWrite, WriteRemote and ThreadInvalidate in the appendix have extra actions writeover(tid,rid) and flushover(tid). These two actions were added to indicate that a thread has completed its action, in order to verify some interested properties. Also note that the corresponding part of WriteRemote in the appendix has already contained a solution to a deadlock found during our analysis of the protocol. The solution requires the thread to perform one extra access check after it receives a notification message r_signal. More explanation can be found in Section 6.2.

Table 6
Specification of a thread flushing a region from remote

```
FlushRemote(tid:ThreadId,pid:ProcessorId,rid:RegionId,
            FlushList:RegionIdSet,r:RegionInfo) =
% Decrease Localthreads for rid. If no other thread is using this region,
% this is remembered by setting the last boolean parameter to true,
% the region state is set to Unused.
      (s_thread_flushrequest(tid,pid,gethome(r),rid,r,T).
      s_threadrefresh(tid,pid,rid,setstate(decreaselocalthreads(r),Unused))
      ◁ eq(getlocalthreads(r),1) ▷
      s_thread_flushrequest(tid,pid,gethome(r),rid,r,F).
      s_threadrefresh(tid,pid,rid,decreaselocalthreads(r))).
% Thread releases the flush lock.
s_free_flushlock(pid).
% This invalidation is finished, the thread is notified.
∑pid':ProcessorId r_signal(tid,pid',rid).
ThreadInvalidate(tid,pid,FlushList)
```

5.2.4 Processors

Each processor is modeled as a separate component (with a unique identity pid). Processors get and update the information of a region in a similar way as threads by using a set of send actions: s_processorrequestinfo, s_processornorefresh and s_processorrefresh. How a processor reacts when it receives a *Data Return* message (modeled by r_queue_datareturn) is specified in Table 7. It first checks the last boolean parameter b in the message to find out whether this message is also a home node migration message. If that is the case, it will set itself as the home of the region. Otherwise, it updates the region's information according to the information contained in the message. How a processor reacts when it receives a a *Data Request* message (modeled by r_queue_datarequest) or a *Region Sponmigrate* message (modeled by r_queue_regionsponmigrate) is specified in Table 8. Processors deal with the *Flush Request* messages in a similar way.

Note the specification in Table 7 is slightly different from its corresponding part in the appendix, which has already contained a solution to a problem found during our analysis of the protocol. More explanation can be found in Section 6.2.

We recall that each processor maintains two message queues to store incoming messages. The HomeQueue is designed to buffer messages containing a request, while the RemoteQueue buffers messages containing a reply. To put a message into a queue, a *homequeue lock* or a *remotequeue lock* has to be obtained. The specifications of a HomeQueue and of a RemoteQueue are presented in Tables

Table 7
Specification of a processor dealing with a Data Return message

```
Processor(pid:ProcessorId) =
% The processor gets a Data Request message.
% If it is not a home node migration message by checking not(b),
% then update the information of the region according to r'
% and set its home by the home of r'.
∑tid:ThreadId ∑pid':ProcessorId ∑rid:RegionId ∑r':RegionInfo ∑b:Bool
r_queue_datareturn(tid,pid,pid',rid,r',b).
        (∑r:RegionInfo s_processorrequestinfo(pid,rid,r).
        s_signal(tid,pid,rid).
        s_processorrefresh(pid,rid,sethome(setstate(r,getstate(r')),gethome(r'))).
        s_free_remotequeuelock(pid).Processor(pid)
        ◁ not(b) ▷
% Otherwise, update the writerlist of the region according to r',
% set the its state as USED, and set its home by pid.
        ∑r:RegionInfo s_processorrequestinfo(pid,rid,r).
        s_signal(tid,pid,rid).
        s_processorrefresh(pid,rid,
          sethome(setstate(setwriterlist(r,getwriterlist(r')),USED),pid)).
        s_free_remotequeuelock(pid).Processor(pid))
```

9 and 10, respectively.

For example, when a thread tries to get an up-to-date data copy from a region's home, first a *Data Request* message is put into the home's HomeQueue (r_thread_datarequest). This HomeQueue acquires a homequeue lock (s_require_homequeuelock, see Table 9). The homequeue lock is released afterwards by the processor (s_free_homequeuelock, see Table 8). When the *Data Return* message with the fresh copy of the region arrives at the thread's processor, it is put into its RemoteQueue (r_thread_datareturn). This RemoteQueue acquires a remotequeue lock (s_require_remotequeuelock, see Table 10). The remotequeue lock is released afterwards by the processor (s_free_remotequeuelock, see Table 7). A processor receives messages from its queues by actions like r_queue_datarequest and r_queue_datareturn, which synchronize with the actions s_queue_datarequest and s_queue_datareturn.

5.2.5 Lock management

To acquire a lock, a lock request message should be sent to the region's home (s_require_locktype, where the locktype is homequeue, remotequeue, server, fault or flush). If the lock is available, the manager replies with a grant message (s_nodelay_locktypewait). Otherwise, the requester needs to wait for the lock to be released, and the lock manager adds the requester into the lock's waiting list. To unlock, the current lock owner sends an unlock message to the

Table 8
Specification of a processor dealing with a Data Request and a Region Sponmigrate message

```
Processor(pid:ProcessorId) =
% The processor gets a Data Request message.
∑tid:ThreadId ∑pid':ProcessorId ∑rid:RegionId r_queue_datarequest(tid,pid',pid,rid).
% If the processor is not the home of the region,
% then the message is forwarded to the real home.
∑r:RegionInfo s_processorrequestinfo(pid,rid,r).
(s_thread_datarequest(tid,pid',gethome(r),rid).
s_processornorefresh(pid).s_free_homequeuelock(pid).Processor(pid)
◁ not(eq(gethome(r),pid)) ▷
% Refresh the region's information, and send the region back.
% If the region is unused, then the Data Return message is also
% a home node migration message, so the last parameter b is set to true.
% *new-information-of-the-region* denotes the update of the region's information.
        ( (s_thread_datareturn(tid,pid',pid,rid,
        sethome(setstate(setwriterlist(r,insert(pid',getwriterlist(r))),Used),pid'),T).
        s_processorrefresh(pid,rid,*new-information-of-the-region*).
        s_free_homequeuelock(pid).Processor(pid)
        ◁ eq(getstate(r),Unused) ▷
% It is not a home node migration message. Set the last parameter to false.
        s_thread_datareturn(tid,pid',pid,rid,
        sethome(setstate(setwriterlist(r,insert(pid',getwriterlist(r))),Used),pid'),F).
        s_processorrefresh(pid,rid,*new-information-of-the-region*).
        s_free_homequeuelock(pid).Processor(pid))
% If the processor gets a request forwarded from itself,
% then there is no need to send a Data Return message back.
    ◁ not(eq(pid,pid')) ▷
    s_signal(tid,pid,rid).
    s_processorrefresh(pid,rid,*new-information-of-the-region*).
    s_free_homequeuelock(pid).Processor(pid) )
)
+
% The processor gets a Region Sponmigrate message.
% It becomes the region's home by refreshing the region's parameters.
∑tid:ThreadId ∑pid':ProcessorId ∑rid:RegionId ∑r':RegionInfo
r_queue_regionsponmigrate(tid,pid',pid,rid,r').
    (∑r:RegionInfo s_processorrequestinfo(pid,rid,r).
% Set the home by itself; maintain the state and WriterList.
    s_processorrefresh(pid,rid,*new-information-of-the-region*).
    s_free_homequeuelock(pid).Processor(pid))
+
% The processor gets a Flush Request message.
% ..... denotes a part of the specification that is excluded here.
∑tid:ThreadId ∑pid':ProcessorId ∑rid:RegionId ∑r':RegionInfo ∑b:Bool
r_queue_flushrequest(tid,pid',pid,rid,r',b). .....
```

Table 9
Specification of a HomeQueue

HomeQueue(pid:ProcessorId) =
% HomeQueue gets a Data Request message.
% To deal with it, the homequeue lock is needed.
$\sum_{\text{tid:ThreadId}} \sum_{\text{pid':ProcessorId}} \sum_{\text{rid:RegionId}}$
% Put a message into the queue.
r_thread_datarequest(tid,pid',pid,rid).s_require_homequeuelock(pid).
(r_nodelay_homequeuewait(pid)+r_delay_homequeuewait(pid)).
% The processor takes this message.
s_queue_datarequest(tid,pid',pid,rid).HomeQueue(pid)
+
% HomeQueue gets a Region Sponmigrate message.
$\sum_{\text{tid:ThreadId}} \sum_{\text{pid':ProcessorId}} \sum_{\text{rid:RegionId}} \sum_{\text{r:RegionInfo}}$
% Put a message into the queue.
r_thread_regionsponmigrate(tid,pid',pid,rid,r).s_require_homequeuelock(pid).
(r_nodelay_homequeuewait(pid)+r_delay_homequeuewait(pid)).
% The processor takes this message.
s_queue_regionsponmigrate(tid,pid',pid,rid,r).HomeQueue(pid)
+
% HomeQueue gets a Flush Request message.
$\sum_{\text{tid:ThreadId}} \sum_{\text{pid':ProcessorId}} \sum_{\text{rid:RegionId}} \sum_{\text{r:RegionInfo}} \sum_{\text{b:Bool}}$
% Put a message into the queue.
r_thread_flushrequest(tid,pid',pid,rid,r,b).s_require_homequeuelock(pid).
(r_nodelay_homequeuewait(pid)+r_delay_homequeuewait(pid)).
% The processor takes this message.
s_queue_flushrequest(tid,pid',pid,rid,r,b).HomeQueue(pid)

Table 10
Specification of a RemoteQueue

RemoteQueue(pid:ProcessorId) =
% RemoteQueue gets a Data Return message.
% To deal with it, the remotequeue lock is needed.
$\sum_{\text{tid:ThreadId}} \sum_{\text{pid':ProcessorId}} \sum_{\text{rid:RegionId}} \sum_{\text{r:RegionInfo}} \sum_{\text{b:Bool}}$
% Put a message into the queue.
r_thread_datareturn(tid,pid',pid,rid,r,b).s_require_remotequeuelock(pid).
(r_nodelay_remotequeuewait(pid)+r_delay_remotequeuewait(pid)).
% The processor takes this message.
s_queue_datareturn(tid,pid',pid,rid,r,b).RemoteQueue(pid)

lock manager (s_free_locktype). When the manager gets an unlock message, it checks whether a thread waiting for some lock can be notified following some rules, and sends the thread a notification (s_delay_locktypewait).

In the μCRL specification, lock management of a processor is modeled as a separate component. Each lock is modeled as a natural variable with value either 1 or 0, since a lock can be held by at most one thread at a time. The waiting list for each lock is modeled as a natural number, representing the number of threads in the waiting list. In the μCRL specification, waiting lists do not need to contain thread identities, since waiting and notification are specified by means of a pair of synchronized actions. When a lock is available, the lock manager selects a waiting thread to notify.

Table 11 describes the management of the fault lock of a processor pid. The other four types of locks are managed in a similar way. When a thread requires the fault lock (s_require_faultlock), it may get the lock immediately (s_nodelay_faultwait) if both the fault lock and the flush lock of the processor are not held by any other threads. Otherwise the thread waits for the locks to be freed by other threads. When the lock manager notices that a fault lock has been freed by a thread (r_free_faultlock), it notifies a thread waiting for a lock (s_delay_locktypewait) according to the rules given in Section 4.5. We present only the first rule, as the conditions at the bottom of Table 11.

5.2.6 Initial state

Table 12 contains the initial state of a configuration of the protocol with one region, two processors and three threads: one processor with one thread executing on itself, the other with two threads. Initially, each region's state is *Unused*, the WriterList of each region is empty, the FlushList of each thread is empty, all queues are empty, and all locks are available. The set H contains all send and receive actions, which are renamed into the deadlock δ by means of the encapsulation operator ∂_H. So send and receive actions can only occur in synchronization. The set I contains communication actions (see Appendix A), which are turned into the internal action τ by means of the hiding operator τ_I.

6 Model Checking the Protocol

In this section we present the results of our analysis of the μCRL specification of the cache coherence protocol using a model checker. We analyzed various configurations of processors and threads. The largest configuration we have checked consists of three processors, three threads and one region. In the

Table 11
Part of specification of management of fault locks

```
% We only present those parameters whose values are changed.
Locker(pid:ProcessorId, fault:Natural, flush:Natural, homequeue:Natural,
        remotequeue:Bool, wait_fault:Natural, wait_flush:Natural,
        wait_homequeue:Natural,wait_remotequeue:Natural) =
% Receiving a request for the fault lock.
% If this lock can be granted, send a nodelay message.
r_require_faultlock(pid).
s_nodelay_faultwait(pid).Locker(1/fault)
◁ and(eq(fault,0),eq(flush,0)) ▷
% Otherwise, increase the number of threads waiting for this lock.
% Later on, the thread waiting on fault lock will be notified.
r_require_faultlock(pid).
Locker(S(wait_fault)/wait_fault))
% The fault lock is released. If a thread can be notified,
% send a delay message, and decrease the waiting number.
+ r_free_faultlock(pid).
        (s_delay_serverwait(pid).
        Locker(0/fault,1/homequeue,sub1(wait_homequeue)/wait_homequeue)
        + s_delay_homequeuewait(pid).
        Locker(0/fault,1/homequeue,sub1(wait_homequeue)/wait_homequeue))
        ◁ and(not(eq(wait_homequeue,0)),eq(homequeue,0)) ▷ .....)
◁ and(not(and(eq(wait_homequeue,0),eq(wait_remotequeue,0))),eq(flush,0))▷ .....)
```

Table 12
Initialization of a protocol with two processors, three threads, one region.

```
τ_I ∂_H(
Thread(tid1,pid1,ridema) || Thread(tid2,pid2,ridema) || Thread(tid3,pid1,ridema) ||
Locker(pid1,0,0,0,0,0,0,0,0) || Locker(pid2,0,0,0,0,0,0,0,0) ||
HomeQueue(pid1) || HomeQueue(pid2) ||
RemoteQueue(pid1) || RemoteQueue(pid2) ||
Processor(pid1) || Processor(pid2) ||
Region(pid1,rid1,reg(pid1,Unused,ema,0)) ||
Region(pid2,rid1,reg(pid1,Unused,ema,0)) )
```

μCRL specification, the message queues of the processors can contain only one message.

6.1 Requirements

We formulated three requirements for the cache coherence protocol.

(1) *Deadlock absence*: The protocol never ends up in a state where it cannot perform any action.
(2) *Unique home*: For each region, at any time there only exists one home.
(3) *Bounded forwarding*: Requests for writing to or flushing a region cannot be forwarded forever.

We did not verify the order assertions and preconditions imposed on the implementation of the protocol by the developers (see Section 5.2.1). These order assertions and preconditions were taken into account while writing the μCRL specification, and are satisfied trivially.

6.2 Validation of the requirements

The μCRL tool set was used to check the syntax and the static semantics of the specification, and also to transform it into a linear form. The linear form was used to generate LTSs, against which we validated the three requirements.

The temporal logic used as input language for Evaluator, which is a model checker within CADP, is called the *regular alternation-free μ-calculus* [22]. It is an extension of the alternation-free fragment of the μ-calculus with action predicates and regular expressions over action sequences. The regular alternation-free μ-calculus is built from three types of formulas, according to the following BNF grammar:

(1) Action formulae $\alpha ::= \mathsf{T} \mid a \mid \neg\alpha \mid \alpha_1 \wedge \alpha_2$
(2) Regular formulae $\beta ::= \alpha \mid \beta_1 \cdot \beta_2 \mid \beta_1 | \beta_2 \mid \beta^*$
(3) State formulae $\varphi ::= \mathsf{F} \mid \mathsf{T} \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \langle a \rangle \varphi \mid [a]\varphi \mid Y \mid \mu Y.\varphi \mid \nu Y.\varphi$

Action formulas α represent a set of actions: T denotes all actions, a the set $\{a\}$, $\neg\alpha$ the complement of α, and $\alpha_1 \wedge \alpha_2$ the intersection of α_1 and α_2. Regular expressions β represent a set of traces: $\beta_1 \cdot \beta_2$ denotes the traces that can be obtained by concatenating a trace from β_1 and a trace from β_2, $\beta_1 | \beta_2$ the union of β_1 and β_2, and β^* the transitive-reflexive closure of β (i.e., the traces that can be obtained by concatenating finitely many traces from β). $\langle \beta \rangle \phi$ means that ϕ holds after some trace from β, and $[\beta]\phi$ means that ϕ holds after all traces from β. The boolean operators have the usual meaning: a state of an LTS always satisfies T; it never satisfies F; it satisfies $\varphi_1 \vee \varphi_2$ if it satisfies φ_1 or φ_2; it satisfies $\varphi_1 \wedge \varphi_2$ if it satisfies both φ_1 and φ_2. The formulas $\mu Y.\varphi$ and $\nu Y.\varphi$ represent minimal and maximal fixpoints, respectively. See [22] for more information on the regular alternation-free μ-calculus.

Table 13
Modified specification of a thread writing to a region from remote

```
% ... represents the parts shown earlier in the paper.
WriteRemote(tid:ThreadId, pid:ProcessorId, rid:RegionId, FlushList:RegionIdSet) =
s_require_faultlock(pid).
(r_nodelay_faultwait(pid)+r_delay_faultwait(pid)).
% Ask for a fresh copy of the region.
∑r:RegionInfo s_threadrequestinfo(tid,pid,rid,r).
        (...
        ◁ not(eq(gethome(r),pid)) ▷
        s_threadnorefresh(tid,pid,rid).
        s_free_faultlock(pid).
        WriteHome(tid,pid,rid,FlushList) )
```

6.2.1 Requirement 1

We used the μCRL tool set with respect to the linearized version of our μCRL specification to check for deadlocks. This deadlock checking exercise led to the detection of many mistakes both in the informal description and in the μCRL specification of the protocol. For the first case, when the developers extracted a C-like description of the protocol from its implementation, they abstracted away from certain implementation details; some of these details were actually crucial for the correctness of the μCRL specification. For the second case, at some points we understood the description differently from what the developers really meant. Whenever a deadlock trace was found, it was simulated to understand the reason for the deadlock. This analysis took us a lot of time, since many of the traces were quite long (typically more than 100 transitions) and difficult to comprehend. Whenever a mistake was found, the μCRL specification was adapted and checked for deadlocks again.

One deadlock, found on a configuration of two processors each containing one thread, was a real problem in the implementation. When a thread wants to write to a region from remote, it acquires the fault lock of its home by sending a lock message. If the lock is unavailable, the thread waits for the lock to be released. Whenever it is notified, it continues with its access to the region and holds the fault lock until it sends an unlock message. In the deadlock trace, we found that while a thread is waiting for a fault lock, the home of the region may migrate to the thread's processor. In fact the thread writes to the region at home, so that it needs to acquire the server lock instead of the fault lock. This error resulted in a deadlock in the implementation. The chosen solution is that after a thread obtains a fault lock, it checks whether it still writes from remote (see Table 13). If this is not the case, it sends an unlock message to release the held fault lock (s_free_faultlock), and then behaves as writing to the region at home (WriteHome). After fixing this problem, no more deadlocks were found.

Table 14
Modified specification of a region

```
% ... represents the parts shown earlier in the paper.
Region(pid:ProcessorId, rid:RegionId, r:RegionInfo) = ... +
% s_home, r_home indicate pid is the home of the region rid.
% s_copy, r_copy indicate pid has a copy of the region rid.
((s_home(rid)+r_home(rid)) ◁ eq(pid,gethome(r)) ▷ (s_copy(rid)+r_copy(rid))).
Region(pid,rid,r)
```

6.2.2 Requirement 2

In the cache coherence protocol, when a region is created on one processor, a copy of this region is also created on every other processor. Due to automatic home node migration, it needs to be checked that:

2.1 At any time, each region has at most one home node.
2.2 If no home node migration is taking place, each region has no more than $n-1$ copies, where n is the number of processors.

To verify requirement 2.1, actions s_home and r_home were added to the specification of a region, when a region finds that its location equals its home node (see Table 14). The idea is that if two different copies of a region rid find that their location is the region's home, then s_home(rid) of one of the copies can communicate with r_home(rid) of the other copy, resulting in an occurrence of c_home(rid).

We verified requirement 2.1 by checking the absence of c_home in the generated LTSs. This is formulated in the regular alternation-free μ-calculus as follows:

2.1 [T*·c_home(rid)] F

Here, we use rid to indicate an identity of a region. It says that if an execution sequence contains c_home, then in the resulting state false holds.

To verify requirement 2.2 in case of two processors, actions s_copy and r_copy were added to the specification of a region, when a region finds that its location does not equal its home node (see Table 14). The idea is that if there are two processors, then there are two different copies of a region rid if and only if c_copy(rid) occurs, because in that case s_copy(rid) of one of the copies can communicate with r_copy(rid) of the other copy.

For requirement 2.2, we needed to identify the states where no home node migration is taking place. We formulated a sufficient condition: when no lock is being held and the message queues are empty, there can be no home node migration. We added two actions homequeue_empty and remotequeue_empty

Table 15
Modified specification of HomeQueue, RemoteQueue, and Locker

```
% ... represents the parts shown earlier in the paper.
HomeQueue(pid:ProcessorId) = ... + homequeue_empty(pid).HomeQueue(pid)

RemoteQueue(pid:ProcessorId)= ... + remotequeue_empty(pid).RemoteQueue(pid)

Locker(pid:ProcessorId, fault:Natural, flush:Natural, homequeue:Natural,
       remotequeue:Bool, wait_fault:Natural, wait_flush:Natural,
       wait_homequeue:Natural,wait_remotequeue:Natural) = ... +
lock_empty(pid).Locker(*no update*)
◁ and(and(and(and(and(and(and(
   eq(fault,0),eq(flush,0)),eq(homequeue,0)),eq(remotequeue,0)),eq(wait_fault,0)),
   eq(wait_flush,0)),eq(wait_homequeue,0)),eq(wait_remotequeue,0)) ▷ δ
```

to the μCRL specification of queues to indicate that queues are empty, and added another action lock_empty to the specification of the lock manager to indicate that no lock is being held (see Table 15). Then we need to check whether the generated LTS does not contain a state which can perform the actions c_copy, lock_empty, homequeue_empty and remotequeue_empty. This requirement is presented in the regular alternation-free μ-calculus as follows:

$$2.2\ \neg(\langle T^* \rangle\ (\langle \mathsf{c\_copy(rid)} \rangle\ T \wedge \langle \mathsf{lock\_empty} \rangle\ T \wedge \langle \mathsf{homequeue\_empty} \rangle\ T \wedge \langle \mathsf{remotequeue\_empty} \rangle\ T))$$

A second error in the implementation of the protocol was found while model checking this property on a configuration of two processors, with two threads running on one processor and a third thread on the other processor. The error can happen when a thread is writing to a region from remote. During its waiting for an up-to-date copy of the region from the region's home (pid'), the home node may migrate (by a *Region Sponmigrate* message) to the processor (pid) where the thread resides. When the *Data Return* message with an up-to-date copy of the region arrives, the thread refreshes the region's home by the sender of this message (pid'). In the resulting state of the protocol, neither of the two processors is the home of the region. As a result c_copy may take place in a state where no lock is being held and the message queues are empty. The chosen solution is given in Table 16. When a processor gets a *Data Return* message containing region information r', and this message is not a home node migration message (the boolean variable b is false), the processor checks whether it is already the home of the region. If that is the case, the processor will not update the region's home by the home of r'. After fixing this problem as proposed, property 2.2 was successfully model checked.

Table 16
Modified specification of a processor dealing with a Data Return message

```
% ... represents the parts shown earlier in the paper.
Processor(pid:ProcessorId) =
∑tid:ThreadId ∑pid':ProcessorId ∑rid:RegionId ∑r':RegionInfo ∑b:Bool
r_queue_datareturn(tid,pid,pid',rid,r',b).
        (∑r:RegionInfo s_processorrequestinfo(pid,rid,r).
          (... ◁ not(eq(gethome(r),pid)) ▷
          s_signal(tid,pid,rid).
          s_processorrefresh(pid,rid,sethome(setstate(r,USED),pid)).
          s_free_remotequeuelock(pid).Processor(pid))
◁ not(b) ▷ ...)
```

6.2.3 Requirement 3

The third requirement expresses a liveness property of the protocol. It says that requests of writing to or flushing a region cannot be bounced around the network forever. This requirement can only be satisfied under some fairness condition. For example, consider a configuration with two threads. One thread can write and flush a region repeatedly forever, so that the other thread will have no chance to finish a write operation. An execution sequence is fair if it does not infinitely often enable the execution of a certain transition without executing it infinitely often (see, e.g., [25]). Actions writeover and flushover were added to the μCRL specification of a thread to indicate that a thread completed its pending actions. The following shows the code in the regular alternation-free μ-calculus for this requirement.

3.1 A thread eventually finishes writing to a region:

$[T^* \cdot \mathsf{write}(\mathsf{tid}, \mathsf{rid}) \cdot (\neg \mathsf{writeover}(\mathsf{tid}, \mathsf{rid}))^*]$
$\langle (\neg \mathsf{writeover}(\mathsf{tid}, \mathsf{rid}) \wedge \neg \mathsf{write}(\mathsf{tid}, \mathsf{rid}))^* \cdot \mathsf{writeover}(\mathsf{tid}, \mathsf{rid}) \rangle\, T$

3.2 A thread eventually finishes its flush of a region:

$[T^* \cdot \mathsf{flush}(\mathsf{tid}) \cdot (\neg \mathsf{flushover}(\mathsf{tid}))^*]$
$\langle (\neg \mathsf{flushover}(\mathsf{tid}) \wedge \neg \mathsf{flush}(\mathsf{tid}))^* \cdot \mathsf{flushover}(\mathsf{tid}) \rangle\, T$

We use tid to indicate an identity of a thread and rid to indicate an identity of a region. These two formulas express that after a thread initiates its action (write(tid,rid) or flush(tid)), the end of this action (writeover(tid,rid) or flushover(tid)) is inevitable under the fairness assumption. This requirement was successfully model checked.

We applied advanced techniques for generating LTSs on a cluster at CWI, consisting of eight nodes. Each node is a dual AMD Athlon MP 1600+ system, with 1.4Ghz processors 2GB RAM and 40GB disk. The nodes are connected by a private ethernet network (100baseT switch) and by a public fast ethernet network (1000baseT switch). Our case study benefited a lot from the μCRL distributed LTS generation tool [4], and also pushed forward its development.

The sizes of the generated LTSs and the verification results are summarized in Table 17. Due to the complexity of this protocol, the size of the LTS grows very rapidly with respect to the number of threads and processors. With the current μCRL tool set, we could generate LTSs for the following configurations:

(1) ccp111: one processor with one thread, one region;
(2) ccp112: one processor with one thread, two regions;
(3) ccp121: one processor with two threads, one region;
(4) ccp122: one processor with two threads, two regions;
(5) ccp221: two processors, each with one thread, one region;
(6) ccp222: two processors, each with one thread, two regions;
(7) ccp231: two processors, one with one thread, the other with two threads, one region;
(8) ccp331: three processors, each with one thread, one region.

For the last configuration, we could only check deadlock absence on the generated LTS, which was too large to serve as input to CADP to model check requirements 2 and 3. The (distributed) μCRL toolset also supports reduction of LTSs modulo branching bisimulation [5]. Requirements 2.1 and 3 were successfully checked on the reduced LTS (8,992,109 states and 88,374,686 transitions). The shortest error traces for the two flaws in the original implementation of the protocol (see Section 6.2) consisted of more than 300 transitions.

7 Conclusions

We used formal specification and model checking techniques to analyze a cache coherence protocol for a Java DSM implementation. We specified the protocol in the process algebraic language μCRL, and analyzed it using the CADP model checker. Three general requirements were formulated and verified. Our analysis uncovered many inconsistencies between the protocol description and its implementation. Two flaws were detected in the design of the protocol, which were then repaired in its implementation. Using the model checker we showed that the improved design does satisfy all requirements, at least for

Table 17
Verification results

Configuration	States	Transitions	Requirements Checked
ccp111	26	98	1, 2, 3
ccp112	97	375	1, 2, 3
ccp121	400	1,814	1, 2, 3
ccp122	5,368	25,278	1, 2, 3
ccp221	65,234	453,568	1, 2, 3
ccp222	2,227,404	16,443,768	1, 2, 3
ccp231	5,424,848	39,603,188	1, 2, 3
ccp331	76,893,921	823,448,619	1, 2.1, 3

configurations of up to three processors and three threads.

During the specification and analysis phase, we encountered quite a few difficulties. First, it took a relatively long time to obtain a μCRL specification of the protocol. During this period, the developers made important changes to the protocol, so that the μCRL specification had to be updated a number of times. Such gaps between an implementation and its formal model could be avoided if formal methods had been used at an earlier design phase. Second, both the developers and analyzers made mistakes in their work. In our analysis, many deadlocks were due to inconsistencies and misunderstandings. Third, advanced techniques for distributed state space generation, reduction, and model checking were essential for this verification effort.

Although model checking itself is fully automated, using model checking to prove the correctness of distributed protocols or algorithms from real life is hard labour. In order to achieve the final verification results, we went through the following steps:

- first, make a model from an informal description and/or its source code; understanding the communication relation among components is crucial in this presented case study;
- second, obtain requirements (from the developers) that can be formalized in modal logic;
- third, use abstraction to scale down the model to a feasible size for verification tools by ignoring the details irrelevant to those requirements;
- fourth, choose an appropriate tool to deal with the state explosion problem, in our case the distributed version of the μCRL tools;
- fifth, once counter-examples are found, they must be traced back to the model and then the real system, to reveal whether it is a bug in the system or a flaw during the modeling phase.

Acknowledgments

We thank Stefan Blom, Jan Friso Groote, Bert Lisser, Radu Mateescu, Jaco van de Pol and Judi Romijn for valuable discussions and technical support. We also thank the anonymous referees, whose remarks have helped us improve our presentation a lot.

References

[1] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[2] B. Badban, W.J. Fokkink, J.F. Groote, J. Pang, and J.C. van de Pol. Verification of a sliding window protocol in μCRL and PVS. *Formal Aspects of Computing*, 17(3):342–388, 2005.

[3] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A tool set for analysing algebraic specifications. In *Proc. 13th Conference on Computer Aided Verification*, LNCS 2102, pp. 250–254. Springer, 2001.

[4] S.C.C. Blom, I. van Langevelde, and B. Lisser. Compressed and distributed file formats for labeled transition systems. In *Proc. 2nd Workshop on Parallel and Distributed Model Checking*, ENTCS 89(1): 68-83. Elsevier, 2003.

[5] S.C.C. Blom and S.M. Orzan. Distributed branching bisimulation reduction of state spaces. In *Proc. 2nd Workshop on Parallel and Distributed Model Checking*, ENTCS 89(1): 99-113. Elsevier, 2003.

[6] M. Broy, S. Merz, and M. Spies, editors. *Formal Systems Specification: The RPC-Memory Specification Case Study*, LNCS 1169. Springer, 1996.

[7] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.

[8] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proc. 12th Conference on Computer Aided Verification*, LNCS 1855, pp. 53–68. Springer, 2000.

[9] M. Dubois, J.-C. Wang, L. Barroso, K. Lee, and Y.-S. Chen. Delayed consistency and its effects on the miss rate of parallel programs. In *Proc. 1991 ACM/IEEE Conference on Supercomputing*, pp. 197–206, 1991.

[10] H. Garavel, F. Lang and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology Newsletter*, 4:13-24, 2002.

[11] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[12] J.F. Groote, J. Pang, and A.G. Wouters. Analysis of a distributed system for lifting trucks. *Journal of Logic and Algebraic Programming*, 55(1-2):21–56, 2003.

[13] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Proc. 1st Workshop on the Algebra of Communicating Processes*, Workshops in Computing Series, pp. 26–62. Springer, 1995.

[14] J.F. Groote, A. Ponse, and Y.S. Usenko. Linearization in parallel pCRL. *Journal of Logic and Algebraic Programming*, 48(1/2):39–72, 2001.

[15] J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, eds., *Handbook of Process Algebra*, pp. 1151–1208. Elsevier, 2001.

[16] T.A. Henzinger, S. Qadeer, and S. Rajamani. Verifying sequential consistency on shared memory multiprocessor systems. In *Proc. 11th Conference on Computer-Aided Verification*, LNCS 1633, pp. 301–315. Springer, 1999.

[17] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: distributed shared memory on standard workstations and operating systems. In *Proc. USENIX Winter 1994 Conference*, pp. 115–132, 1994.

[18] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Transaction on Computers*, 28(9):690–691, 1979.

[19] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley/Teubner, 1996.

[20] J. Maessen, Arvind, and X. Shen. Improving the Java memory model using CRF. In *Proc. 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 1–12, 2000.

[21] J. Manson and W. Pugh. Core semantics of multithreaded Java. In *Proc. ACM 2001 Java Grande Conference*, pp. 29–38, 2001.

[22] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, 46(3):255-281, 2003.

[23] J.C. van de Pol and M. Valero Espada. Formal specification of JavaSpaces^{TM} architecture using μCRL. In *Proc. 5th Conference on Coordination Models and Languages*, LNCS 2315, pp. 274–290. Springer, 2002.

[24] F. Pong and M. Dubois. Formal automatic verification of cache coherence in multiprocessors with relaxed memory models. *IEEE Transaction on Parallel and Distributed Systems*, 11(9):989–1006, 2000.

[25] J-P. Queille and J. Sifakis. Fairness and related properties in transition systems - A temporal logic to deal with fairness. *Acta Informatica*, 19:195-220, 1983.

[26] A. Roychoudhury and T. Mitra. Specifying multithreaded Java semantics for program verification. In *Proc. ACM SIGSOFT Conference on Software Engineering*, pp. 192–201, 2002.

[27] X. Shen, Arvind, and L. Rodolph. Cachet: an adaptive cache coherence protocol of distributed shared memory systems. In *Proc. 13th ACM Conference on Supercomputing*, pp. 135–144, 1999.

[28] J. Stoy, X. Shen, and Arvind. Proofs of correctness of cache-coherence protocols. In *Proc. 11th Symposium of Formal Methods Europe*, LNCS 2021, pp. 43–71. Springer, 2001.

[29] R. Veldema, R.F.H. Hofman, R. Bhoedjang, and H. Bal. Runtime-optimizations for a Java DSM. In *Proc. ACM 2001 Java Grande Conference*, pp. 89–98, 2001.

[30] R. Veldema, R.F.H. Hofman, R. Bhoedjang, C. Jacobs, and H. Bal. Source-level global optimizations for fine-grain distributed shared memory systems. In *Proc. 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 83–92, 2001.

[31] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Analyzing the CRF Java memory model. In *Proc. 8th Asia-Pacific Software Engineering Conference*, pp. 21–28, 2001.

[32] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Specifying Java thread semantics using a uniform memory model. In *Proc. ACM 2002 Java Grande Conference*, pp. 192–201, 2002.

[33] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release-consistency protocols for shared virtual memory systems. In *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation*, pp. 75–88, 1996.

A Appendix: μCRL code of the protocol

```
%%%
```

```
%%%
```

```
     in:ProcessorId#ProcessorIdSet->ProcessorIdSet
map  remove:ProcessorId#ProcessorIdSet->ProcessorIdSet
     tail:ProcessorIdSet->ProcessorIdSet
     test:ProcessorId#ProcessorIdSet->Bool
     empty:ProcessorIdSet->Bool
     if:Bool#ProcessorIdSet#ProcessorIdSet->ProcessorIdSet
     eq:ProcessorIdSet#ProcessorIdSet->Bool
     count:ProcessorIdSet->Natural
     head:ProcessorIdSet->ProcessorId
     insert:ProcessorId#ProcessorIdSet->ProcessorIdSet
var  a,a':ProcessorId
     A,A':ProcessorIdSet
rew  remove(a,ema)=ema
     remove(a,in(a',A))=if(eq(a,a'),remove(a,A),in(a',remove(a,A)))
     tail(in(a,A))=A
     test(a,ema)=F test(a,in(a',A))=if(eq(a,a'),T,test(a,A))
     empty(ema)=T empty(in(a,A))=F
     if(T,A,A')=A if(F,A,A')=A'
     eq(ema,ema)=T eq(ema,in(a,A))=F eq(in(a,A),ema)=F
     eq(in(a,A),A')=and(test(a,A'),
                        eq(remove(a,in(a,A)),remove(a,A')))
     count(ema)=0
     count(in(a,A))=S(count(remove(a,in(a,A))))
     head(in(a,A))=a
     insert(a,ema)=in(a,ema)
     insert(a,in(a',A'))=if(eq(a,a'),in(a',A'),
                            if(le(a,a'),in(a,in(a',A')),
                                        in(a',insert(a,A')))
                         )

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% sort RegionIdSet
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort RegionIdSet
func ridema:->RegionIdSet
     in:RegionId#RegionIdSet->RegionIdSet
map  remove:RegionId#RegionIdSet->RegionIdSet
     tail:RegionIdSet->RegionIdSet
     test:RegionId#RegionIdSet->Bool
     empty:RegionIdSet->Bool
     if:Bool#RegionIdSet#RegionIdSet->RegionIdSet
     eq:RegionIdSet#RegionIdSet->Bool
     count:RegionIdSet->Natural
     head:RegionIdSet->RegionId
     insert:RegionId#RegionIdSet->RegionIdSet
var  a,a':RegionId
     A,A':RegionIdSet
```

```
rew  remove(a,ridema)=ridema
     remove(a,in(a',A))=if(eq(a,a'),remove(a,A),in(a',remove(a,A)))
     tail(in(a,A))=A
     test(a,ridema)=F test(a,in(a',A))=if(eq(a,a'),T,test(a,A))
     empty(ridema)=T empty(in(a,A))=F
     if(T,A,A')=A if(F,A,A')=A'
     eq(ridema,ridema)=T eq(ridema,in(a,A))=F eq(in(a,A),ridema)=F
     eq(in(a,A),A')=and(test(a,A'),
                        eq(remove(a,in(a,A)),remove(a,A')))
     count(ridema)=0
     count(in(a,A))=S(count(remove(a,in(a,A))))
     head(in(a,A))=a
     insert(a,A)=if(test(a,A),A,in(a,A))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% State of regions, initially, the region is UNUSED.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort State
func UNUSED,USED:->State
map  eq: State#State->Bool
     if:Bool#State#State->State
var  s1,s2:State
rew  eq(UNUSED, UNUSED)=T eq(UNUSED, USED)=F
     eq(USED, UNUSED)=F eq(USED, USED)=T
     if(T,s1,s2)=s1 if(F,s1,s2)=s2

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sort: RegionInfo
% Id, Home, State, writerlist, the number of local threads
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort RegionInfo
func reg:ProcessorId#State#ProcessorIdSet#Natural->RegionInfo
map  gethome:RegionInfo->ProcessorId
     getstate:RegionInfo->State
     getwriterlist:RegionInfo->ProcessorIdSet
     getlocalt:RegionInfo->Natural
     sethome:RegionInfo#ProcessorId->RegionInfo
     setstate:RegionInfo#State->RegionInfo
     setwriterlist:RegionInfo#ProcessorIdSet->RegionInfo
     setlocalt:RegionInfo#Natural->RegionInfo
     increaselocalt:RegionInfo->RegionInfo
     decreaselocalt:RegionInfo->RegionInfo
     eq:RegionInfo#RegionInfo->Bool
var  h,h':ProcessorId
     w,w':ProcessorIdSet
     s,s':State
     lt,lt':Natural
```

```
rew  gethome(reg(h,s,w,lt))=h
     getstate(reg(h,s,w,lt))=s
     getwriterlist(reg(h,s,w,lt))=w
     getlocalt(reg(h,s,w,lt))=lt
     sethome(reg(h,s,w,lt),h')=reg(h',s,w,lt)
     setstate(reg(h,s,w,lt),s')=reg(h,s',w,lt)
     setwriterlist(reg(h,s,w,lt),w')=reg(h,s,w',lt)
     setlocalt(reg(h,s,w,lt),lt')=reg(h,s,w,lt')
     increaselocalt(reg(h,s,w,lt))=reg(h,s,w,S(lt))
     decreaselocalt(reg(h,s,w,lt))=reg(h,s,w,sub1(lt))
     eq(reg(h,s,w,lt),reg(h',s',w',lt'))=
           and(and(and(eq(h,h'), eq(s,s')),eq(w,w')),eq(lt,lt'))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Actions: we synchronize s_* and r_* into action c_*.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
act
s_require_faultlock,r_require_faultlock,
c_require_faultlock: ProcessorId

s_require_flushlock,r_require_flushlock,
c_require_flushlock: ProcessorId

s_require_serverlock,r_require_serverlock,
c_require_serverlock: ProcessorId

s_require_homequeuelock,r_require_homequeuelock,
c_require_homequeuelock: ProcessorId

s_require_remotequeuelock,r_require_remotequeuelock,
c_require_remotequeuelock: ProcessorId

s_free_faultlock,r_free_faultlock,
c_free_faultlock: ProcessorId

s_free_flushlock,r_free_flushlock,
c_free_flushlock: ProcessorId

s_free_serverlock,r_free_serverlock,
c_free_serverlock: ProcessorId

s_free_homequeuelock,r_free_homequeuelock,
c_free_homequeuelock: ProcessorId

s_free_remotequeuelock,r_free_remotequeuelock,
c_free_remotequeuelock: ProcessorId
```

```
s_nodelay_faultwait,r_nodelay_faultwait,
c_nodelay_faultwait: ProcessorId

s_nodelay_flushwait,r_nodelay_flushwait,
c_nodelay_flushwait: ProcessorId

s_nodelay_serverwait,r_nodelay_serverwait,
c_nodelay_serverwait: ProcessorId

s_nodelay_homequeuewait,r_nodelay_homequeuewait,
c_nodelay_homequeuewait: ProcessorId

s_nodelay_remotequeuewait,r_nodelay_remotequeuewait,
c_nodelay_remotequeuewait: ProcessorId

s_delay_faultwait,r_delay_faultwait,
c_delay_faultwait: ProcessorId

s_delay_flushwait,r_delay_flushwait,
c_delay_flushwait: ProcessorId

s_delay_serverwait,r_delay_serverwait,
c_delay_serverwait: ProcessorId

s_delay_homequeuewait,r_delay_homequeuewait,
c_delay_homequeuewait: ProcessorId

s_delay_remotequeuewait,r_delay_remotequeuewait,
c_delay_remotequeuewait: ProcessorId

s_thread_datarequest,r_thread_datarequest,
c_thread_datarequest,
s_queue_datarequest,r_queue_datarequest,
c_queue_datarequest:
ThreadId#ProcessorId#ProcessorId#RegionId

s_thread_datareturn,r_thread_datareturn,
c_thread_datareturn,
s_queue_datareturn,r_queue_datareturn,
c_queue_datareturn:
ThreadId#ProcessorId#ProcessorId#RegionId#RegionInfo#Bool

s_thread_flushrequest,r_thread_flushrequest,
c_thread_flushrequest,
s_queue_flushrequest,r_queue_flushrequest,
c_queue_flushrequest:
ThreadId#ProcessorId#ProcessorId#RegionId#RegionInfo#Bool
```

```
s_thread_regionsponmigrate,r_thread_regionsponmigrate,
c_thread_regionsponmigrate,
s_queue_regionsponmigrate,r_queue_regionsponmigrate,
c_queue_regionsponmigrate:
ThreadId#ProcessorId#ProcessorId#RegionId#RegionInfo

s_threadrequestinfo,r_threadrequestinfo,
c_threadrequestinfo:ThreadId#ProcessorId#RegionId#RegionInfo

s_threadrefresh,r_threadrefresh,
c_threadrefresh:ThreadId#ProcessorId#RegionId#RegionInfo

s_threadnorefresh,r_threadnorefresh,
c_threadnorefresh:ThreadId#ProcessorId#RegionId

s_processorrequestinfo,r_processorrequestinfo,
c_processorrequestinfo:ProcessorId#RegionId#RegionInfo

s_processorrefresh,r_processorrefresh,
c_processorrefresh:ProcessorId#RegionId#RegionInfo

s_processornorefresh,r_processornorefresh,
c_processornorefresh:ProcessorId#RegionId

s_signal,r_signal,c_signal:ThreadId#ProcessorId#RegionId

write,writeover: ThreadId#RegionId
flush,flushover: ThreadId
r_home,s_home,c_home,r_copy,s_copy,c_copy: RegionId
lock_empty,homequeue_empty,remotequeue_empty: ProcessorId

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: Thread
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Thread(tid:ThreadId,pid:ProcessorId,FlushList:RegionIdSet)=
     sum(rid:RegionId, write(tid,rid).
         ThreadWrite(tid,pid,rid,FlushList)
     )
     +
     flush(tid).ThreadInvalidate(tid,pid,FlushList)
     <| not(empty(FlushList)) |>delta

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: ThreadWrite
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc ThreadWrite(tid:ThreadId,pid:ProcessorId,
```

```
                rid:RegionId,FlushList:RegionIdSet)=
     writeover(tid,rid).
     Thread(tid,pid,FlushList)
     <| test(rid, FlushList) |>
     sum(r:RegionInfo,s_threadrequestinfo(tid,pid,rid,r).
        (s_threadnorefresh(tid,pid,rid).
         WriteHome(tid,pid,rid,insert(rid,FlushList))
         <| eq(gethome(r), pid) |>
         s_threadnorefresh(tid,pid,rid).
         WriteRemote(tid,pid,rid,insert(rid,FlushList))
     )  )

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: WriteHome
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc WriteHome(tid:ThreadId,pid:ProcessorId,
               rid:RegionId,FlushList:RegionIdSet)=
     s_require_serverlock(pid).
     (r_nodelay_serverwait(pid)+r_delay_serverwait(pid)).
     sum(r:RegionInfo,s_threadrequestinfo(tid,pid,rid,r).
        (  (s_threadrefresh(tid,pid,rid,
              increaselocalt(setstate(setwriterlist(
              r,insert(pid,getwriterlist(r))),USED))).
            s_free_serverlock(pid).
            writeover(tid,rid).Thread(tid,pid,FlushList)
            <| eq(getstate(r), UNUSED) |>
            s_threadrefresh(tid,pid,rid,
               increaselocalt(setwriterlist(
               r,insert(pid,getwriterlist(r))))).
            s_free_serverlock(pid).
            writeover(tid,rid).Thread(tid,pid,FlushList)
            )
         <| eq(gethome(r), pid) |>
         s_threadnorefresh(tid,pid,rid).
         s_free_serverlock(pid).
         WriteRemote(tid,pid,rid,FlushList)
     )  )

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: WriteRemote
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc WriteRemote(tid:ThreadId,pid:ProcessorId,
                 rid:RegionId,FlushList:RegionIdSet)=
     s_require_faultlock(pid).
     (r_nodelay_faultwait(pid)+r_delay_faultwait(pid)).
     sum(r:RegionInfo,s_threadrequestinfo(tid,pid,rid,r).
        (s_thread_datarequest(tid,pid,gethome(r),rid).
```

```
          s_threadnorefresh(tid,pid,rid).
          sum(pid':ProcessorId,r_signal(tid,pid',rid).
             sum(newr:RegionInfo,
                 s_threadrequestinfo(tid,pid,rid,newr).
                 s_threadrefresh(tid,pid,rid,increaselocalt(newr)).
                 s_free_faultlock(pid).
                 writeover(tid,rid).Thread(tid,pid,FlushList)
          )  )
          <| not(eq(gethome(r),pid)) |>
          s_threadnorefresh(tid,pid,rid).
          s_free_faultlock(pid).
          WriteHome(tid,pid,rid,FlushList)
     )  )

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: ThreadInvalidate
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc ThreadInvalidate(tid:ThreadId,pid:ProcessorId,
                      FlushList:RegionIdSet)=
     flushover(tid).
     Thread(tid,pid,FlushList)
     <| empty(FlushList) |>
     s_require_flushlock(pid).
     (r_nodelay_flushwait(pid)+r_delay_flushwait(pid)).
     sum(r:RegionInfo,
        s_threadrequestinfo(tid,pid,head(FlushList),r).
        (FlushHome(tid,pid,head(FlushList),tail(FlushList),r)
         <| eq(gethome(r),pid) |>
         FlushRemote(tid,pid,head(FlushList),tail(FlushList),r)
     )  )

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: FlushHome
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc FlushHome(tid:ThreadId,pid:ProcessorId,rid:RegionId,
               FlushList:RegionIdSet,r:RegionInfo)=
     (  s_threadrefresh(tid,pid,rid,
          decreaselocalt(setstate(setwriterlist(
          r,remove(pid,getwriterlist(r))),UNUSED))).
        s_free_flushlock(pid).
        ThreadInvalidate(tid,pid,FlushList )
        <| empty(remove(pid,getwriterlist(r))) |>
        (  (  s_thread_regionsponmigrate(tid,pid,
                head(remove(pid,getwriterlist(r))),rid,
                setwriterlist(r,remove(pid,getwriterlist(r)))).
              s_threadrefresh(tid,pid,rid,
                sethome(decreaselocalt(setstate(
```

```
                setwriterlist(r,ema),UNUSED)),
                head(remove(pid,getwriterlist(r))))).
              s_free_flushlock(pid).
              ThreadInvalidate(tid,pid,FlushList)
              <| not(eq(head(remove(pid,getwriterlist(r))), pid))|>
              s_threadrefresh(tid,pid,rid,
                decreaselocalt(setwriterlist(
                r,remove(pid,getwriterlist(r))))).
              s_free_flushlock(pid).
              ThreadInvalidate(tid,pid,FlushList)
           )
           <| eq(count(remove(pid,getwriterlist(r))),S(0)) |>
              s_threadrefresh(tid,pid,rid,
                decreaselocalt(setwriterlist(
                r,remove(pid,getwriterlist(r))))).
              s_free_flushlock(pid).
              ThreadInvalidate(tid,pid,FlushList)
          )   )
     <| eq(getlocalt(r),S(0)) |>
     s_threadrefresh(tid,pid,rid,decreaselocalt(r)).
     s_free_flushlock(pid).
     ThreadInvalidate(tid,pid,FlushList)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: FlushRemote
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc FlushRemote(tid:ThreadId,pid:ProcessorId,rid:RegionId,
                 FlushList:RegionIdSet,r:RegionInfo)=
     s_thread_flushrequest(tid,pid,gethome(r),rid,r,T).
     s_threadrefresh(tid,pid,rid,
       decreaselocalt(setwriterlist(setstate(
       r,UNUSED),ema))).
     s_free_flushlock(pid).
     sum(pid':ProcessorId,r_signal(tid,pid',rid).
         ThreadInvalidate(tid,pid,FlushList)
     )
     <| eq(getlocalt(r),S(0)) |>
     s_thread_flushrequest(tid,pid,gethome(r),rid,r,F).
     s_threadrefresh(tid,pid,rid,
       decreaselocalt(setwriterlist(r,ema))).
     s_free_flushlock(pid).
     sum(pid':ProcessorId,
         r_signal(tid,pid',rid).
         ThreadInvalidate(tid,pid,FlushList)
     )

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Process: Region
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Region(pid:ProcessorId, rid: RegionId, r:RegionInfo)=
        sum(tid:ThreadId, r_threadrequestinfo(tid,pid,rid,r).
           (r_threadnorefresh(tid,pid,rid).Region(pid,rid,r)+
            sum(r':RegionInfo,
                r_threadrefresh(tid,pid,rid,r').
                Region(pid,rid,r'))
        )  )
        +
        r_processorrequestinfo(pid,rid,r).
           (r_processornorefresh(pid,rid).Region(pid,rid,r)+
            sum(r':RegionInfo,
                r_processorrefresh(pid,rid,r').
                Region(pid,rid,r'))
           )
        +
        r_home(rid).Region(pid,rid,r)
        <| eq(pid,gethome(r)) |>delta
        +
        s_home(rid).Region(pid,rid,r)
        <| eq(pid,gethome(r)) |>delta
        +
        r_copy(rid).Region(pid,rid,r)
        <| not(eq(pid,gethome(r))) |>delta
        +
        s_copy(rid).Region(pid,rid,r)
        <| not(eq(pid,gethome(r))) |>delta

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: Processor
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Processor(pid:ProcessorId)=
     sum(tid:ThreadId,sum(pid':ProcessorId,
     sum(rid:RegionId,sum(r':RegionInfo,sum(b:Bool,
         r_queue_datareturn(tid,pid,pid',rid,r',b).
         (  sum(r:RegionInfo,s_processorrequestinfo(pid,rid,r).
            ( s_signal(tid,pid,rid).
              s_processorrefresh(pid,rid,sethome(setstate(
                r,getstate(r')),gethome(r'))).
              s_free_remotequeuelock(pid).
              Processor(pid)
              <| not(eq(gethome(r),pid)) |>
              s_signal(tid,pid,rid).
              s_processorrefresh(pid,rid,sethome(setstate(
                r,USED),pid)).
              s_free_remotequeuelock(pid).
```

```
          Processor(pid)
        )  )
    <| not(b) |>
        sum(r:RegionInfo,s_processorrequestinfo(pid,rid,r).
            s_signal(tid,pid,rid).
            s_processorrefresh(pid,rid,
              sethome(setstate(setwriterlist(
              r,getwriterlist(r')),USED),pid)).
            s_free_remotequeuelock(pid).
            Processor(pid)
        )
)   )  )  )  )  )
+
sum(tid:ThreadId,sum(pid':ProcessorId,sum(rid:RegionId,
    r_queue_datarequest(tid,pid',pid,rid).
    sum(r:RegionInfo,
        s_processorrequestinfo(pid,rid,r).
        (  s_thread_datarequest(tid,pid',gethome(r),rid).
           s_processornorefresh(pid,rid).
           s_free_homequeuelock(pid).
           Processor(pid)
           <| not(eq(gethome(r),pid)) |>
           (  (   s_thread_datareturn(tid,pid',pid,rid,
                    sethome(setstate(
                    setwriterlist(r,
                    insert(pid',getwriterlist(r))),
                    USED),pid'),T).
                  s_processorrefresh(pid,rid,
                    sethome(setstate(setwriterlist(
                    r, ema),UNUSED),pid')).
                  s_free_homequeuelock(pid).
                  Processor(pid)
                  <| eq(getstate(r),UNUSED) |>
                  s_thread_datareturn(tid,pid',pid,rid,
                    setstate(setwriterlist(
                    r, insert(pid',getwriterlist(r))),USED),F).
                  s_processorrefresh(pid,rid,
                    setstate(setwriterlist(
                    r, insert(pid',getwriterlist(r))),USED)).
                  s_free_homequeuelock(pid).
                  Processor(pid)
              )
              <| not(eq(pid,pid')) |>
               s_signal(tid,pid,rid).
               s_processorrefresh(pid,rid,
                 setstate(setwriterlist(
                 r, insert(pid',getwriterlist(r))),USED)).
```

```
                s_free_homequeuelock(pid).
                Processor(pid)
            )
    )   )
)   )   )
+
sum(tid:ThreadId,sum(pid':ProcessorId,
sum(rid:RegionId,sum(r':RegionInfo,sum(b:Bool,
    r_queue_flushrequest(tid,pid',pid,rid,r',b).
    sum(r:RegionInfo,
        s_processorrequestinfo(pid,rid,r).
        (  s_thread_flushrequest(tid,pid',gethome(r),rid,r',b).
           s_processornorefresh(pid,rid).
           s_free_homequeuelock(pid).
           Processor(pid)
           <| not(eq(gethome(r), pid)) |>
           (  s_signal(tid,pid,rid).
              s_processorrefresh(pid,rid,r).
              s_free_homequeuelock(pid).
              Processor(pid)
              <| not(b) |>
              (  s_signal(tid,pid,rid).
                 s_processorrefresh(pid,rid,
                   setstate(setwriterlist(
                   r,remove(pid',getwriterlist(r))),UNUSED)).
                 s_free_homequeuelock(pid).
                 Processor(pid)
                 <| empty(remove(pid',getwriterlist(r))) |>
                 (  (   s_thread_regionsponmigrate(tid,pid,
                          head(remove(pid',getwriterlist(r))),rid,
                          setwriterlist(r,
                          remove(pid',getwriterlist(r)))).
                        s_signal(tid,pid,rid).
                        s_processorrefresh(pid,rid,sethome(
                          setstate(
                          setwriterlist(r,ema),UNUSED),
                          head(remove(pid',getwriterlist(r))))).
                        s_free_homequeuelock(pid).
                        Processor(pid)
                        <| not(eq(head(remove(pid',
                            getwriterlist(r))),gethome(r))) |>
                        s_signal(tid,pid,rid).
                        s_processorrefresh(pid,rid,
                          setstate(setwriterlist(
                          r,remove(pid',getwriterlist(r))),USED)).
                        s_free_homequeuelock(pid).
                        Processor(pid)
```

```
                        )
                        <| eq(count(remove(pid',getwriterlist(r))),
                              S(0)) |>
                        s_signal(tid,pid,rid).
                        s_processorrefresh(pid,rid,setwriterlist(
                          r,remove(pid',getwriterlist(r))) ).
                        s_free_homequeuelock(pid).
                        Processor(pid)
            )  )  )
     )  )  )  )  )  )  )
     +
     sum(tid:ThreadId,sum(pid':ProcessorId,
     sum(rid:RegionId,sum(r':RegionInfo,
         r_queue_regionsponmigrate(tid,pid',pid,rid,r').
         sum(r:RegionInfo,
             s_processorrequestinfo(pid,rid,r).
             s_processorrefresh(pid,rid,
               sethome(setstate(setwriterlist(
               r,getwriterlist(r')),USED),pid)).
             s_free_homequeuelock(pid).
             Processor(pid)
     )  )  )  )  )

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: HomeQueue
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc HomeQueue(pid: ProcessorId)=
     sum(tid:ThreadId,sum(pid':ProcessorId,sum(rid:RegionId,
         r_thread_datarequest(tid,pid',pid,rid).
         s_require_homequeuelock(pid).
         (r_nodelay_homequeuewait(pid)+r_delay_homequeuewait(pid)).
         s_queue_datarequest(tid,pid',pid,rid).HomeQueue(pid)
     )  )  )
     +
     sum(tid:ThreadId,sum(pid':ProcessorId,
     sum(rid:RegionId,sum(r:RegionInfo,
         r_thread_regionsponmigrate(tid,pid',pid,rid,r).
         s_require_homequeuelock(pid).
         (r_nodelay_homequeuewait(pid)+r_delay_homequeuewait(pid)).
         s_queue_regionsponmigrate(tid,pid',pid,rid,r).
         HomeQueue(pid)
     )  )  )  )
     +
     sum(tid:ThreadId,sum(pid':ProcessorId,
     sum(rid:RegionId,sum(r:RegionInfo,sum(b:Bool,
         r_thread_flushrequest(tid,pid',pid,rid,r,b).
         s_require_homequeuelock(pid).
```

```
          (r_nodelay_homequeuewait(pid)+r_delay_homequeuewait(pid)).
          s_queue_flushrequest(tid,pid',pid,rid,r,b).HomeQueue(pid)
     )  )  )  )  )
     + homequeue_empty(pid).HomeQueue(pid)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: RemoteQueue
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc RemoteQueue(pid: ProcessorId)=
     sum(tid:ThreadId,sum(pid':ProcessorId,
     sum(rid:RegionId,sum(r:RegionInfo,sum(b:Bool,
          r_thread_datareturn(tid,pid,pid',rid,r,b).
          s_require_remotequeuelock(pid).
          (r_nodelay_remotequeuewait(pid)+
           r_delay_remotequeuewait(pid)).
          s_queue_datareturn(tid,pid,pid',rid,r,b).RemoteQueue(pid)
     )  )  )  )  )
     + remotequeue_empty(pid).RemoteQueue(pid)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: Locker
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Locker(pid:ProcessorId,fault:Natural,flush:Natural,
            homequeue:Natural,remotequeue:Natural,
            wait_fault:Natural,wait_flush:Natural,
            wait_homequeue:Natural,wait_remotequeue:Natural)=
     lock_empty(pid).
     Locker(pid,fault,flush,homequeue,remotequeue,
            wait_fault,wait_flush,wait_homequeue,wait_remotequeue)
     <| and(and(and(and(and(and(and(
         eq(fault,0),eq(flush,0)),eq(homequeue,0)),
         eq(remotequeue,0)),eq(wait_fault,0)),eq(wait_flush,0)),
         eq(wait_homequeue,0)),eq(wait_remotequeue,0)) |>delta
     +
     r_require_faultlock(pid).
     s_nodelay_faultwait(pid).
     Locker(pid,S(fault),flush,homequeue,remotequeue,
            wait_fault,wait_flush,wait_homequeue,
            wait_remotequeue)
     <| and(eq(fault,0), eq(flush,0)) |>
     r_require_faultlock(pid).
     Locker(pid,fault,flush,homequeue,remotequeue,
            S(wait_fault),wait_flush,wait_homequeue,
            wait_remotequeue)
     +
     r_require_flushlock(pid).
     s_nodelay_flushwait(pid).
```

```
Locker(pid,fault,S(flush),homequeue,remotequeue,
       wait_fault,wait_flush,wait_homequeue,
       wait_remotequeue)
<| and(and(and(eq(fault,0),eq(flush,0)),
    eq(homequeue,0)),eq(remotequeue,0)) |>
r_require_flushlock(pid).
Locker(pid,fault,flush,homequeue,remotequeue,
       wait_fault,S(wait_flush),wait_homequeue,
       wait_remotequeue)
+
r_require_serverlock(pid).
s_nodelay_serverwait(pid).
Locker(pid,fault,flush,S(homequeue),remotequeue,
       wait_fault,wait_flush,wait_homequeue,
       wait_remotequeue)
<| and(eq(homequeue,0),eq(flush,0)) |>
r_require_serverlock(pid).
Locker(pid,fault,flush,homequeue,remotequeue,
       wait_fault,wait_flush,S(wait_homequeue),
       wait_remotequeue)
+
r_require_homequeuelock(pid).
s_nodelay_homequeuewait(pid).
Locker(pid,fault,flush,S(homequeue),remotequeue,
       wait_fault,wait_flush,wait_homequeue,
       wait_remotequeue)
<| and(eq(homequeue,0),eq(flush,0)) |>
r_require_homequeuelock(pid).
Locker(pid,fault,flush,homequeue,remotequeue,
       wait_fault,wait_flush,S(wait_homequeue),
       wait_remotequeue)
+
r_require_remotequeuelock(pid).
s_nodelay_remotequeuewait(pid).
Locker(pid,fault,flush,homequeue,S(remotequeue),
       wait_fault,wait_flush,wait_homequeue,
       wait_remotequeue)
<| and(eq(remotequeue,0),eq(flush,0)) |>
r_require_remotequeuelock(pid).
Locker(pid,fault,flush,homequeue,remotequeue,
       wait_fault,wait_flush,wait_homequeue,
       S(wait_remotequeue))
+
r_free_faultlock(pid).
(  (  (  s_delay_serverwait(pid).
          Locker(pid,sub1(fault),flush,S(homequeue),
                 remotequeue,wait_fault,wait_flush,
```

```
                sub1(wait_homequeue),wait_remotequeue)
          +
          s_delay_homequeuewait(pid).
          Locker(pid,sub1(fault),flush,S(homequeue),
                 remotequeue,wait_fault,wait_flush,
                 sub1(wait_homequeue),wait_remotequeue)
       )
       <| and(not(eq(wait_homequeue,0)),eq(homequeue,0)) |>
       (  (  s_delay_remotequeuewait(pid).
             Locker(pid,sub1(fault),flush,homequeue,
                    S(remotequeue),wait_fault,wait_flush,
                    wait_homequeue,sub1(wait_remotequeue))
             <| not(eq(wait_remotequeue,0)) |>
             Locker(pid,sub1(fault),flush,homequeue,
                    remotequeue,wait_fault,wait_flush,
                    wait_homequeue,wait_remotequeue)
          )
          <| eq(remotequeue,0) |>
          Locker(pid,sub1(fault),flush,homequeue,
                 remotequeue,wait_fault,wait_flush,
                 wait_homequeue,wait_remotequeue)
   ) )
   <| and(not(and(eq(wait_homequeue,0),
       eq(wait_remotequeue,0))),eq(flush,0)) |>
   (  s_delay_flushwait(pid).
      Locker(pid,sub1(fault),S(flush),homequeue,
             remotequeue,wait_fault,sub1(wait_flush),
             wait_homequeue,wait_remotequeue)
      <| and(and(and(and(
         not(eq(wait_flush,0)),eq(remotequeue,0)),
         eq(homequeue,0)),
         eq(flush,0)),
         eq(fault,S(0))) |>
      (  s_delay_faultwait(pid).
         Locker(pid,fault,flush,homequeue,
                remotequeue,sub1(wait_fault),wait_flush,
                wait_homequeue,wait_remotequeue)
         <| and(and(and(
            not(eq(wait_fault,0)),eq(homequeue,0)),
            eq(flush,0)),
            eq(fault,S(0))) |>
         Locker(pid,sub1(fault),flush,homequeue,
                remotequeue,wait_fault,wait_flush,
                wait_homequeue,wait_remotequeue)
) ) )
+
r_free_flushlock(pid).
```

```
(  (  (  s_delay_serverwait(pid).
         Locker(pid,fault,sub1(flush),S(homequeue),
                remotequeue,wait_fault,wait_flush,
                sub1(wait_homequeue),wait_remotequeue)
         +
         s_delay_homequeuewait(pid).
         Locker(pid,fault,sub1(flush),S(homequeue),
                remotequeue,wait_fault,wait_flush,
                sub1(wait_homequeue),wait_remotequeue)
      )
      <| and(not(eq(wait_homequeue,0)),eq(homequeue,0)) |>
      (  (  s_delay_remotequeuewait(pid).
            Locker(pid,fault,sub1(flush),homequeue,
                   S(remotequeue),wait_fault,wait_flush,
                   wait_homequeue,sub1(wait_remotequeue))
            <| not(eq(wait_remotequeue,0)) |>
            Locker(pid,fault,sub1(flush),homequeue,
                   remotequeue,wait_fault,wait_flush,
                   wait_homequeue,wait_remotequeue)
         )
         <| eq(remotequeue,0) |>
         Locker(pid,fault,sub1(flush),homequeue,
                remotequeue,wait_fault,wait_flush,
                wait_homequeue,wait_remotequeue)
   ) )
   <| and(not(and(
      eq(wait_homequeue,0),
      eq(wait_remotequeue,0))),
      eq(flush,S(0))) |>
   (  s_delay_flushwait(pid).
      Locker(pid,fault,flush,homequeue,
             remotequeue,wait_fault,sub1(wait_flush),
             wait_homequeue,wait_remotequeue)
      <| and(and(and(and(
         not(eq(wait_flush,0)),
         eq(remotequeue,0)),
         eq(homequeue,0)),
         eq(sub1(flush),0)),
         eq(fault,0) )|>
      (  s_delay_faultwait(pid).
         Locker(pid,S(fault),sub1(flush),homequeue,
                remotequeue,sub1(wait_fault),wait_flush,
                wait_homequeue,wait_remotequeue)
         <| and(and(and(
            not(eq(wait_fault,0)),
            eq(homequeue,0)),
            eq(flush,S(0))),
```

```
              eq(fault,0)) |>
           Locker(pid,fault,sub1(flush),homequeue,
                  remotequeue,wait_fault,wait_flush,
                  wait_homequeue,wait_remotequeue)
) ) )
+
r_free_serverlock(pid).
(  ( (  s_delay_serverwait(pid).
        Locker(pid,fault,flush,homequeue,
               remotequeue,wait_fault,wait_flush,
               sub1(wait_homequeue),wait_remotequeue)
        +
        s_delay_homequeuewait(pid).
        Locker(pid,fault,flush,homequeue,
               remotequeue,wait_fault,wait_flush,
               sub1(wait_homequeue),wait_remotequeue)
        )
        <| and(not(eq(wait_homequeue,0)),
           eq(homequeue,S(0))) |>
        (  (  s_delay_remotequeuewait(pid).
              Locker(pid,fault,flush,sub1(homequeue),
                     S(remotequeue),wait_fault,wait_flush,
                     wait_homequeue,sub1(wait_remotequeue))
              <| not(eq(wait_remotequeue,0)) |>
              Locker(pid,fault,flush,sub1(homequeue),
                     remotequeue,wait_fault,wait_flush,
                     wait_homequeue,wait_remotequeue)
           )
           <| eq(remotequeue,0) |>
           Locker(pid,fault,flush,sub1(homequeue),
                  remotequeue,wait_fault,wait_flush,
                  wait_homequeue,wait_remotequeue)
      ) )
      <| and(not(and(eq(wait_homequeue,0),
         eq(wait_remotequeue,0))),eq(flush,0)) |>
      (  s_delay_flushwait(pid).
         Locker(pid,fault,S(flush),sub1(homequeue),
                remotequeue,wait_fault,sub1(wait_flush),
                wait_homequeue,wait_remotequeue)
         <| and(and(and(and(
            not(eq(wait_flush,0)),
            eq(remotequeue,0)),
            eq(homequeue,S(0))),
            eq(flush,0)),
            eq(fault,0)) |>
         (  s_delay_faultwait(pid).
            Locker(pid,S(fault),flush,sub1(homequeue),
```

```
                  remotequeue,sub1(wait_fault),wait_flush,
                  wait_homequeue,wait_remotequeue)
            <| and(and(and(
               not(eq(wait_fault,0)),
               eq(homequeue,S(0))),
               eq(flush,0)),
               eq(fault,0)) |>
            Locker(pid,fault,flush,sub1(homequeue),
                  remotequeue,wait_fault,wait_flush,
                  wait_homequeue,wait_remotequeue)
) ) )
+
r_free_homequeuelock(pid).
( ( ( s_delay_serverwait(pid).
        Locker(pid,fault,flush,homequeue,
               remotequeue,wait_fault,wait_flush,
               sub1(wait_homequeue),wait_remotequeue)
        +
        s_delay_homequeuewait(pid).
        Locker(pid,fault,flush,homequeue,
               remotequeue,wait_fault,wait_flush,
               sub1(wait_homequeue),wait_remotequeue)
      )
      <| and(eq(homequeue,S(0)),not(eq(wait_homequeue,0))) |>
      ( ( s_delay_remotequeuewait(pid).
          Locker(pid,fault,flush,sub1(homequeue),
                 S(remotequeue),wait_fault,wait_flush,
                 wait_homequeue,sub1(wait_remotequeue))
          <| not(eq(wait_remotequeue,0)) |>
          Locker(pid,fault,flush,sub1(homequeue),
                 remotequeue,wait_fault,wait_flush,
                 wait_homequeue,wait_remotequeue)
        )
        <| eq(remotequeue,0) |>
        Locker(pid,fault,flush,sub1(homequeue),
               remotequeue,wait_fault,wait_flush,
               wait_homequeue,wait_remotequeue)
    ) )
    <| and(not(and(eq(wait_homequeue,0),
       eq(wait_remotequeue,0))),eq(flush,0)) |>
    ( s_delay_flushwait(pid).
      Locker(pid,fault,S(flush),sub1(homequeue),
             remotequeue,wait_fault,sub1(wait_flush),
             wait_homequeue,wait_remotequeue)
      <| and(and(and(and(
         not(eq(wait_flush,0)),
         eq(remotequeue,0)),
```

```
            eq(homequeue,S(0))),
            eq(flush,0)),
            eq(fault,0)) |>
         (  s_delay_faultwait(pid).
            Locker(pid,S(fault),flush,sub1(homequeue),
                   remotequeue,sub1(wait_fault),wait_flush,
                   wait_homequeue,wait_remotequeue)
            <| and(and(and(
               not(eq(wait_fault,0)),
               eq(homequeue,S(0))),
               eq(flush,0)),
               eq(fault,0)) |>
            Locker(pid,fault,flush,sub1(homequeue),
                   remotequeue,wait_fault,wait_flush,
                   wait_homequeue,wait_remotequeue)
) ) )
+
r_free_remotequeuelock(pid).
(  (  (  s_delay_serverwait(pid).
         Locker(pid,fault,flush,S(homequeue),
                sub1(remotequeue),wait_fault,wait_flush,
                sub1(wait_homequeue),wait_remotequeue)
         +
         s_delay_homequeuewait(pid).
         Locker(pid,fault,flush,S(homequeue),
                sub1(remotequeue),wait_fault,wait_flush,
                sub1(wait_homequeue),wait_remotequeue)
       )
       <| and( eq(homequeue,0),not(eq(wait_homequeue,0))) |>
       (  (  s_delay_remotequeuewait(pid).
             Locker(pid,fault,flush,homequeue,
                    remotequeue,wait_fault,wait_flush,
                    wait_homequeue,sub1(wait_remotequeue))
             <| not(eq(wait_remotequeue,0)) |>
             Locker(pid,fault,flush,homequeue,
                    sub1(remotequeue),wait_fault,wait_flush,
                    wait_homequeue,wait_remotequeue)
          )
          <| eq(remotequeue,S(0)) |>
          Locker(pid,fault,flush,homequeue,
                 sub1(remotequeue),wait_fault,wait_flush,
                 wait_homequeue,wait_remotequeue)
    ) )
    <| and(not( and(eq(wait_homequeue,0),
        eq(wait_remotequeue,0))),eq(flush,0)) |>
    (  s_delay_flushwait(pid).
       Locker(pid,fault,S(flush),homequeue,
```

```
                sub1(remotequeue),wait_fault,sub1(wait_flush),
                wait_homequeue,wait_remotequeue)
          <| and(and(and(and(
             not(eq(wait_flush,0)),
             eq(remotequeue,S(0))),
             eq(homequeue,0)),
             eq(flush,0)),
             eq(fault,0)) |>
          (  s_delay_faultwait(pid).
             Locker(pid,S(fault),flush,homequeue,
                    sub1(remotequeue),sub1(wait_fault),wait_flush,
                    wait_homequeue,wait_remotequeue)
             <| and(and(and(
                not(eq(wait_fault,0)),
                eq(homequeue,0)),
                eq(flush,0)),
                eq(fault,0)) |>
             Locker(pid,fault,flush,homequeue,
                    sub1(remotequeue),wait_fault,wait_flush,
                    wait_homequeue,wait_remotequeue)
     )  )  )

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Communications
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
comm

s_require_faultlock | r_require_faultlock = c_require_faultlock
s_require_flushlock | r_require_flushlock = c_require_flushlock
s_require_serverlock | r_require_serverlock = c_require_serverlock
s_require_homequeuelock | r_require_homequeuelock
= c_require_homequeuelock
s_require_remotequeuelock | r_require_remotequeuelock
= c_require_remotequeuelock
s_free_faultlock | r_free_faultlock = c_free_faultlock
s_free_flushlock | r_free_flushlock = c_free_flushlock
s_free_serverlock | r_free_serverlock = c_free_serverlock
s_free_homequeuelock | r_free_homequeuelock = c_free_homequeuelock
s_free_remotequeuelock | r_free_remotequeuelock
= c_free_remotequeuelock
s_nodelay_faultwait | r_nodelay_faultwait = c_nodelay_faultwait
s_nodelay_flushwait | r_nodelay_flushwait = c_nodelay_flushwait
s_nodelay_serverwait | r_nodelay_serverwait = c_nodelay_serverwait
s_nodelay_homequeuewait | r_nodelay_homequeuewait
= c_nodelay_homequeuewait
s_nodelay_remotequeuewait | r_nodelay_remotequeuewait
= c_nodelay_remotequeuewait
```

```
s_delay_faultwait | r_delay_faultwait = c_delay_faultwait
s_delay_flushwait | r_delay_flushwait = c_delay_flushwait
s_delay_serverwait | r_delay_serverwait = c_delay_serverwait
s_delay_homequeuewait | r_delay_homequeuewait
= c_delay_homequeuewait
s_delay_remotequeuewait | r_delay_remotequeuewait
= c_delay_remotequeuewait
s_thread_datarequest | r_thread_datarequest = c_thread_datarequest
s_queue_datarequest | r_queue_datarequest = c_queue_datarequest
s_thread_datareturn | r_thread_datareturn = c_thread_datareturn
s_queue_datareturn | r_queue_datareturn = c_queue_datareturn
s_thread_flushrequest | r_thread_flushrequest
= c_thread_flushrequest
s_queue_flushrequest | r_queue_flushrequest = c_queue_flushrequest
s_thread_regionsponmigrate | r_thread_regionsponmigrate
= c_thread_regionsponmigrate
s_queue_regionsponmigrate | r_queue_regionsponmigrate
= c_queue_regionsponmigrate
s_threadrequestinfo | r_threadrequestinfo = c_threadrequestinfo
s_threadrefresh | r_threadrefresh = c_threadrefresh
s_threadnorefresh | r_threadnorefresh = c_threadnorefresh
s_processorrequestinfo | r_processorrequestinfo
= c_processorrequestinfo
s_processorrefresh | r_processorrefresh = c_processorrefresh
s_processornorefresh | r_processornorefresh
= c_processornorefresh
s_signal | r_signal = c_signal
s_home | r_home = c_home
s_copy | r_copy = c_copy

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The protocol with 2 processors, 3 threads and 1 region.
% (each processor has a copy of the region)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
init

hide({ c_require_faultlock,c_free_faultlock,
       c_require_flushlock,c_free_flushlock,
       c_require_serverlock,c_free_serverlock,
       c_require_homequeuelock,c_free_homequeuelock,
       c_require_remotequeuelock,c_free_remotequeuelock,
       c_nodelay_faultwait,c_nodelay_flushwait,c_nodelay_serverwait,
       c_nodelay_homequeuewait,c_nodelay_remotequeuewait,
       c_delay_faultwait,c_delay_flushwait,c_delay_serverwait,
       c_delay_homequeuewait,c_delay_remotequeuewait,
       c_thread_datarequest,c_queue_datarequest,
       c_thread_datareturn,c_queue_datareturn,
```

```
c_thread_flushrequest,c_queue_flushrequest,
c_thread_regionsponmigrate,c_queue_regionsponmigrate,
c_threadrequestinfo, c_processorrequestinfo,
c_threadrefresh, c_processorrefresh,
c_threadnorefresh, c_processornorefresh,
c_signal},
encap({ s_require_faultlock,r_require_faultlock,
        s_require_flushlock,r_require_flushlock,
        s_require_serverlock,r_require_serverlock,
        s_require_homequeuelock,r_require_homequeuelock,
        s_require_remotequeuelock,r_require_remotequeuelock,
        s_free_faultlock,r_free_faultlock,
        s_free_flushlock,r_free_flushlock,
        s_free_serverlock,r_free_serverlock,
        s_free_homequeuelock,r_free_homequeuelock,
        s_free_remotequeuelock,r_free_remotequeuelock,
        s_nodelay_faultwait,r_nodelay_faultwait,
        s_nodelay_flushwait,r_nodelay_flushwait,
        s_nodelay_serverwait,r_nodelay_serverwait,
        s_nodelay_homequeuewait,r_nodelay_homequeuewait,
        s_nodelay_remotequeuewait,r_nodelay_remotequeuewait,
        s_delay_faultwait,r_delay_faultwait,
        s_delay_flushwait,r_delay_flushwait,
        s_delay_serverwait,r_delay_serverwait,
        s_delay_homequeuewait,r_delay_homequeuewait,
        s_delay_remotequeuewait,r_delay_remotequeuewait,
        s_thread_datarequest,r_thread_datarequest,
        s_queue_datarequest,r_queue_datarequest,
        s_thread_datareturn,r_thread_datareturn,
        s_queue_datareturn,r_queue_datareturn,
        s_thread_flushrequest,r_thread_flushrequest,
        s_queue_flushrequest,r_queue_flushrequest,
        s_thread_regionsponmigrate,
        r_thread_regionsponmigrate,
        s_queue_regionsponmigrate,r_queue_regionsponmigrate,
        s_threadrequestinfo,r_threadrequestinfo,
        s_threadrefresh, r_threadrefresh,
        s_threadnorefresh, r_threadnorefresh,
        s_processorrequestinfo,r_processorrequestinfo,
        s_processorrefresh, r_processorrefresh,
        s_processornorefresh, r_processornorefresh,
        s_signal, r_signal,
        s_home, r_home,
        s_copy, r_copy},
    Thread(tid1,pid1,ridema) ||
    Thread(tid2,pid2,ridema) ||
    Thread(tid3,pid1,ridema) ||
```

```
        Locker(pid1,0,0,0,0,0,0,0,0) ||
        Locker(pid2,0,0,0,0,0,0,0,0) ||
        HomeQueue(pid1) ||
        HomeQueue(pid2) ||
        RemoteQueue(pid1) ||
        RemoteQueue(pid2) ||
        Processor(pid1) ||
        Processor(pid2) ||
        Region(pid1,rid1,reg(pid1,UNUSED,ema,0)) ||
        Region(pid2,rid1,reg(pid1,UNUSED,ema,0))
)       )
```