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

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

[1]CWI, Software Engineering Department
Kruislaan 413, 1098 SJ
Amsterdam, The Netherlands
{pangjun,wan}@cwi.nl

[2]Vrije Universiteit, Department of Computer Science
De Boelelaan 1081A, 1081 HV
Amsterdam, The Netherlands
{wanf,rutger,rveldema}@cs.vu.nl

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.

1. Introduction

Multithreading is a programming paradigm for implementing parallel applications on shared memory multiprocessors. The Java memory model (JMM) [9] prescribes certain abstract rules that any implementation of Java multithreading must follow.

Shared memory is an attractive programming model for interprocess communication and synchronization in multiprocess computations. In the past decade, a popular research topic has been the design of systems to provide a shared memory abstraction on 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).

Jackal [25] 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 (which is either an object or a fixed-size partition of an array) created on another processor (i.e., the region's home). All threads on one process share one copy of a cached region. The home node 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 [24, 25] improve both sequential and parallel application performance. Among them, the *automatic home node migration* reduces the amount of synchronization, by automatically appointing the processor that is likely to access a region most often as the region's home.

μCRL [10] 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 toolset [3] can be used in combination with CADP [7] to generate, visualize and analyze this LTS. For example, one can detect deadlocks and livelocks, or check the validity of temporal logic formulas.

In this paper, we present our formal analysis of a cache coherence protocol for Jackal using the μCRL toolset 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 toolset together with CADP. Our analysis revealed many inconsistencies between the description and the implementation. Two errors were found in the implementation. Both errors can happen when a thread is writing a region from remote (i.e., the thread does not run on the home of the region). During the thread's waiting for a proper protocol lock or an up-to-date copy of the region, the home node may migrate to the thread's processor, so

*This research is partly supported by the Dutch Technology Foundation STW under the project CES5008: Improving the quality of embedded systems using formal design and systematic testing.

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 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. Section 5 focuses on our formal analysis in μCRL. The μCRL specifications for each component of the protocol and the verification results are given. Discussions and future work are mentioned in Section 6.

2 Related Work

Using formal methods to analyze the JMM is an active research topic. In [21], the authors developed an equivalent formal executable specification of the JMM [9]. Their specification is operational and uses guarded commands. This model can be used to verify popular software construction idioms for multithreaded Java. In [26], the Murϕ verification system was applied to study the CRF memory mode [16]. 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 [17] by the same authors [27]. Two proofs of the correctness for Cachet [22], an adaptive cache coherence protocol, were presented in [23]. 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 [26, 27], 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 [21, 23, 26, 27]. 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 automatic verification of cache coherence on sequentially consistent systems [15], e.g. [4, 5, 12]. Coherence in shared memory multiprocessors is much more difficult to verify. Recently, Pong and Dubois [20] used their state-based tool for the verification of a delayed protocol, [6], which is an aggressive protocol for relaxed memory models. We encountered the same difficulties as [20], 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 [20] 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 current JMM as given in [9].

The JMM allows each thread to cache variables in its working memory, which keeps its own working copy of the variables. A thread can only manipulate the values in its working memory, which is inaccessible to other threads. The working memories are caches of a single main memory, which is shared by all threads. Main memory keeps the main copy of every variable. A thread's working memory must be flushed to main memory at each synchronization point. A synchronization point is a lock or unlock operation corresponding to the entry or exit of a synchronized block.

The JMM defines a set of actions that threads may use to interact with memory. A thread invokes four actions: use, assign, lock and unlock. The other actions; read, load, store and write, are invoked by a multithreaded implementation following the temporal ordering constraints in the current JMM ([9, Chapter 17]). The meaning of each action can be found in [9].

There are many problems in the current JMM [9], such as that semantics for *final* variable operations is omitted and that *Volatile* variable operations do not have synchronization effects for normal variable operations. In view of these problems, the Java Specification Request (JSR) 133 is under development.

4 Jackal DSM System

Jackal [25] is a fine-grained DSM implementation of the Java programming language. 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 (i.e., the region's home). All threads on one processor share one copy of a cached region. The home node 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 [28] and

TreadMarks [14]. As in HLRC, modifications are flushed to a home node; as in TreadMarks, twinning and diffing are 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.

The implementation of the Jackal memory model contains three components: address space management, access checks and synchronization. Several optimizations were made to improve both sequential and parallel application performance [24, 25].

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 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 the *flush list*. At synchronization points, all regions on 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.

4.2 Access check

Jackal's compiler generates 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 current JMM [9].

4.3 Synchronization

Logically, each Java object contains an object lock and a condition variable. Since threads can access objects from different processors, Jackal provides distributed synchronization protocols. Briefly, an object's home node acts as the object's object lock manager. *lock*, *unlock*, *wait* and *notify* calls are implemented as control messages to the lock's home node. To acquire an object lock, a thread sends a lock request message to the object lock manager and waits. When the lock is available, the manager replies with a grant message; otherwise, the thread needs to wait for the lock to be released. To unlock, the lock holder sends an unlock message to the home node. We did not model object locks, since they are not relevant to the requirements that we formulated for the protocol (see Section 5.3).

4.4 Automatic home node migration

Java programs do not indicate which object locks protect which data items. This make it difficult to combine date and synchronization traffic. Jackal may have to communicate multiple times to acquire an object lock, to access the data protected by the lock and to release the lock. Recall that the home of a region acts as the manager of the object lock. To decrease synchronization traffic, automatic home node migration has been implemented in Jackal. It means that Jackal may automatically appoint the processor that is likely to access a region most often as the region's home. 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 in runtime, and thus reduce synchronization traffic. Automatic home node migration complicates meeting the requirements in Section 5.3.

4.5 Other features

To improve performance, a source-level global optimization *object-graph aggregation*, and runtime optimization *adaptive lazy flushing*, are implemented in Jackal.

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 since 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 to the requirements that we formulated.

5 Specification and Analysis in μCRL

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 [2] extended with equational *abstract data types*. A μCRL specification consists of two parts: one part specifies the data types, the other part specifies the processes. 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. There are two predefined actions in μCRL: δ represents deadlock, and τ represents a hidden action. These two actions never carry data parameters. $p \cdot q$ denotes sequential composition, it first executes p and then q. $p + q$ denotes non-deterministic choice, meaning that it can behave as p or q. Summation $\sum_{d:D} p(d)$ provides the possibly infinite choice over a data type D. The conditional construct $p \triangleleft b \triangleright q$ with b a boolean data term behaves as p if b and as q if not b. Parallel composition $p \parallel q$ 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. 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 $\partial_H(p)$, which renames all occurrences in p of actions from the set H into δ. Finally, hiding $\tau_I(p)$ renames all occurrences in p of actions from the set I into τ. 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. The syntax and semantics of μCRL are given in [10].

The μCRL toolset [3] is a collection of tools for analyzing and manipulating μCRL specifications, based on term rewriting and linearization techniques.The μCRL toolset, together with the CADP toolset [7], which acts as a backend for the μCRL toolset, features visualization, simulation, LTS generation, model checking, theorem proving and statebit hashing capabilities. μCRL and its toolset have been successfully used to analyze a wide range of protocols and distributed systems (e.g., [1, 8, 11, 13, 19]).

5.2 Specification of the Protocol

The starting point of verifying a protocol 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 on each component, if needed. In the following discussion, we present the μCRL specification for each component, together with the abstractions we made. Due to space limitation, 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 at `<http://www.cwi.nl/~pangjun/ccp/>`.

Our model of the cache coherence protocol is a parallel composition of threads, processors, regions, protocol lock managers and message queues upon a set of communication actions. By means of these communications, data can be transferred between two processes. The complete μCRL specification of this protocol consists of around 1800 lines.

5.2.1 Threads

Each thread runs on a processor, and can perform a number of actions: *read, write* and *invalidate*. It maintains two lists: *ReadList* contains the identifiers of regions that it is reading or recently read from, and *WriteList* contains the identifiers of regions that it is writing or recently wrote to. When a thread starts reading from or writing to a region, the corresponding access check determines whether there is a valid local copy of this region at the thread's processor. The *server_lock* is needed if the thread runs on the region's home (i.e., if the thread reads or writes at home); otherwise,

```
% Thread writes from remote, requires a fault lock,
% and asks for a fresh copy of the region.
WriteRemote(tid:ThreadId,pid:ProcessId,
  FlushList:RegionIdSet) =
  s_require_faultlock(pid).
  (r_no_faultwait(pid)+r_signal_faultwait(pid)).
  (∑_{r:Region} r_sendback(tid,pid,r).
% Ask for a fresh copy of the region.
   s_data_requiremsg(tid,pid,gethome(r)).
   s_norefresh(tid,pid).
% Copy arrives, the thread is notified.
   (∑_{pid':ProcessId} r_signal(tid,pid').
    (∑_{newr:Region} r_sendback(tid,pid,newr).
    s_refresh(tid,pid,setlocalthreads(newr,
      S(getlocalthreads(newr)))).
    s_free_faultlock(pid).Thread(tid,pid,wl))))
```

Table 1. Specification of a thread writing at a region from remote

the *fault_lock* of the thread's processor is acquired (i.e., if the thread reads or writes from remote). When a fault_lock is granted, the thread retrieves an up-to-date copy of the region from its home node. The thread continues reading from or writing to the region and finally releases the lock by sending an unlock message to the protocol lock manager. When a thread invalidates, it empties both its ReadList and WriteList, and sends a *Flush* message to the home of each region in these lists. If the thread invalidates a region in its WriteList from remote, the Flush message also contains a diff with the difference between the region's object and twin data. The *flush_lock* of the home of each region is acquired before invalidating, and released after invalidating.

In the μCRL specification, each thread is modeled as a separate process with a unique identifier; see Table 1. It contains one parameter *pid* to indicate on which processor the corresponding thread executes. Since the behavior of reading from a region is part of the behavior of writing to a region, and since writing is far more critical for the correctness of the protocol than reading, we abstracted away from the read action of threads. As a result, a thread only maintains a *FlushList* and flushes the regions in this FlushList.

5.2.2 Regions

Jackal uses a single shared virtual address space. Each region occupies the same virtual address range on all processors that store a copy of it. When a region is created on one processor, a copy of this region is also created on every other processor. A region contains these information:

1. Location: A processor's identifier, denoting at which node the region (or a copy) is.

```
% Synchronization between actions:
  s_refresh | r_refresh = c_refresh
  s_norefresh | r_norefresh = c_norefresh
  s_sendback | r_sendback = c_sendback
% r contains the region's information.
Region(pid:ProcessId, r:Region) =
% Communication with threads
  ∑_{tid:ThreadId} s_sendback(tid,pid,r).
   (r_norefresh(tid,pid).Region(pid,r) +
   ∑_{r':Region} r_refresh(tid,pid,r').Region(pid,r'))+
% Communication with processors
  s_sendback(pid,r).
  (r_norefresh(pid).Region(pid,r) +
   ∑_{r':Region} r_refresh(pid,r').Region(pid,r')))
```

Table 2. Specification of a region

2. Home: A processor's identifier, 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*; if all accesses of a region are read actions, the state of this region is *Readonly*; in all other cases, the state of a region is *Shared*.

4. ReaderList: A list of processors' identifiers containing threads that are reading or recently read from this region. It is only maintained at the home node.

5. WriterList: A list of processors' identifiers containing threads that are writing or recently wrote to 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.

In μCRL, each region is modeled as a separate component. As a result of our abstraction on the behavior of threads, we made some corresponding abstractions for regions. Each region has only two kinds of states; we kept the Unused state, while the other three states are mapped to a state *Used*. The region only needs to maintain the *WriterList*. Furthermore, we did not model object and twin data, since it they are not relevant to our requirements for the protocol. So in our model a thread cannot write a real value on 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 unlock its fault_lock.

```
Processor(pid:ProcessId) =
% Processor gets a Region_Sponmigrate message.
% It becomes the region's home node by refreshing
% the region's parameters.
 ∑tid:ThreadId ∑pid':ProcessId ∑r':Region
    r_region_sponmigrate(tid,pid',pid,r').
    (∑r:Region r_sendback(pid,r).
% Set the home by itself; maintain the state and writerlist.
      s_refresh(pid,sethome(setstate(
        setwriterlist(r,getwriterlist(r')),USED),pid)).
      s_free_homequeuelock(pid).Processor(pid))
```

Table 3. Specification of a processor dealing with a message

```
proc HomeQueue(pid:ProcessId) =
% Home queue gets a Region_Sponmigrate message.
% To deal with it, the homequeue_lock is needed.
 ∑tid:ThreadId ∑pid':ProcessId ∑r:Region
% Put a message into the queue.
      r_i_region_sponmigrate(tid,pid',pid,r).
      s_require_homequeuelock(pid).
      (r_no_homequeuewait(pid)
      +r_signal_homequeuewait(pid)).
% The processor takes this message.
      s_i_region_sponmigrate(tid,pid',pid,r).
      HomeQueue(pid)
```

Table 4. Part of specification for a home queue

We use a set of synchronized actions to ensure that during an access to a region, no other processes can change the information of this region. For example, a thread gets the information of a region by performing a synchronized action *r_sendback*, and the accesses to this region are blocked until this thread executes another synchronized action *s_norefresh* (if it has changed nothing) or *s_fresh* (if it has changed something). The synchronized actions on a region are presented in Table 2, together with the specification for regions in μCRL. To avoid state explosion, we only analyzed configurations containing one region.

5.2.3 Messages

Four kinds 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, if the message is a home node migration message, then the processor becomes the home of this region, and starts maintaining the WriterList and the state of the region.

3. *Flush*: This message is sent when a thread flushes from remote. When a processor gets this message, and it is the home of the region, it removes the thread's processor from the WriterList of the region; moreover, it may send a home node migration message to a new home of this region (by a *Region_Sponmigrate* message). When it is not the home of the region, it forwards the Flush 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.

In μCRL, each processor is modeled as a separate component (with a unique identifier). How a processor deals with a Region_Sponmigrate messages is specified in Table 3.

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. 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 node's HomeQueue. When a Data_Return message arrives, it is put into the RemoteQueue of the thread's processor. The μCRL process for a message queue contains one parameter *pid* to indicate which processor this message queue belongs to (see Tables 4 and 5). To avoid state explosion, we only modeled queues that can contain one message.

5.2.4 Protocol locks

As already explained in Section 5.2.1, *protocol locks* guarantee exclusivity when threads write to or flush a region. Each processor acts as the protocol lock manager of its regions and region copies. To acquire a protocol lock, a protocol lock request message is sent to the region's home. If the lock is available, the manager replies with a grant message. Otherwise, the requester needs to wait for the lock to be released, and the protocol lock manager adds the requester into the lock's waiting list. To unlock, the current lock owner sends an unlock message to the protocol lock manager. When the manager gets an unlock message, it

```
proc RemoteQueue(pid:ProcessId) =
% Remote queue gets a Data_Return message.
% To deal with it, the remotequeue_lock is needed.
  Σ_{tid:ThreadId} Σ_{pid':ProcessId} Σ_{r:Region} Σ_{b:Bool}
% Put a message into the queue.
      r_o_data_returnmsg(tid,pid',pid,r,b).
      s_require_remotequeuelock(pid).
      (r_no_remotequeuewait(pid)
      +r_signal_remotequeuewait(pid)).
% The processor takes this message.
      s_o_data_returnmsg(tid,pid',pid,r,b).
      RemoteQueue(pid)
```

Table 5. Specification for a remote queue

checks whether a thread waiting for this lock can be notified, under some constraints. For instance, a fault_lock can be granted only if this fault_lock and the flush_lock are not held by other threads.

There are five protocol locks for each processor: *homequeue_lock*, *remotequeue_lock*, *server_lock*, *fault_lock* and *flush_lock*. 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. The 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's 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 the fault_lock of the thread's processor is needed, respectively. When a thread flushes, the flush_lock of its processor is needed.

Protocol lock management of a processor is modeled in μCRL as a separate component; see Table 6. Each protocol lock is modeled as a boolean variable, since a protocol lock can be held by at most one thread at a time. The waiting list of a lock is modeled as a natural number, representing the number of threads in the waiting list, to enable checking for emptiness; waiting lists do not need to contain thread identifiers, since waiting and notification are specified by means of a pair of synchronized actions, carrying the identifier of the waiting thread as a parameter. When a protocol lock is available, the protocol lock manager randomly selects a waiting thread to notify.

```
% To save space, we only present those parameters
% whose values are changed.
proc Locker(pid:ProcessId,faulters:Bool,flushers:Bool,
  homequeue:Bool,remotequeue:Bool,wait_faulters:Natural,
  wait_flushers:Natural,wait_homequeue:Natural,
  wait_remotequeue:Natural)=
% Get a request for the fault lock. If this lock can be granted,
% send a no-wait message. Otherwise, increase the number
% of threads waiting for this lock.
  r_require_faultlock(pid).
    (s_no_faultwait(pid).Locker(T/faulters)
    ◁and(faulters,flushers)▷
    Locker(S(wait_faulters)/wait_faulters))
% The fault lock is released, if a thread can be notified,
% send a signal_wait message, and decrease the
% waiting number.
  +r_free_faultlock(pid).
    ((s_signal_homequeuewait(pid).
      Locker(F/faulters,T/homequeue,
        sub1(wait_homequeue)/wait_homequeue)
      ◁and(not(eq(wait_homequeue,0)),homequeue)▷...)
    ◁and(not(and(eq(wait_homequeue,0),
      eq(wait_remotequeue,0))),flushers)▷...)
```

Table 6. Part of specification for a protocol lock management

5.3 Requirements

We formulated four types of requirements for the cache coherence protocol.

1. Deadlock freeness: The protocol never ends up in a state where it cannot perform any action.
2. Assertion checking: The protocol violates none of the assertions written in the informal description.
3. Relaxed cache coherence: For each region, at any time there exists one home node.
4. Liveness: Requests for writing to or flushing a region are not be bounced around the network forever.

5.4 Validation of the Requirements

The μCRL toolset 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 for various configurations of processors and threads. Next, we validated the four types of requirements with respect to these configurations.

5.4.1 Requirement 1

We used the μCRL toolset 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 the analyzers 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 300 transitions) and difficult to comprehend. Whenever a mistake was found, the μCRL specification was adapted and checked for deadlocks again.

One deadlock found by the analyzers, 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 node 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 to the home node. 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. Then in fact the thread writes to the region at home, 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. If this is not the case, it sends an unlock message to release the held fault_lock, and then sends a message to acquire the server_lock. After fixing this problem as proposed, no more deadlocks were found.

5.4.2 Requirement 2

The developers added many assertions into the description and required that the protocol should not violate any of them. The assertions 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's resource. For example, when a thread performs an action on a region, the corresponding protocol lock should be already held by the thread. Order assertions are modeled in μCRL by imposing a certain order on the execution of actions. In the forementioned example, in the μCRL specification, the behavior of a thread is modeled like this: only after execution of the action *r_no_serverwait* or *r_signal_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 4.4) the home of the region automatically migrates. Preconditions are modeled in the μCRL specification as boolean terms in conditional expressions.

```
% Synchronization between actions:
  s_home | r_home = c_home
  s_copy | r_copy = c_copy
Region(pid:ProcessId, r:Region)=
% s_home, r_home indicate pid is the home.
  r_home.Region(pid,r)◁eq(pid,gethome(r))▷δ+
  s_home.Region(pid,r)◁eq(pid,gethome(r))▷δ+
% s_copy, r_copy indicate pid has a copy.
  δ◁eq(pid,gethome(r))▷r_copy.Region(pid,r)+
  δ◁eq(pid,gethome(r))▷s_copy.Region(pid,r)
```

Table 7. Modified specification of a region.

5.4.3 Requirement 3

Due to automatic home node migration, it needs to be checked that at any time there exists at most one home node for each region. We divided this requirement into two parts.

3.1 Each region has at most one home node.

3.2 If the system is stable, each region has no more than $n-1$ copies, where n is the number of processors.

To verify these two parts, 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; *s_copy* and *r_copy* were added, when a region finds that its location does not equal its home node. We synchronized s_home and r_home into *c_home*, s_copy and r_copy into *c_copy*; see Table 7. Furthermore, we encapsulated s_home, r_home, s_copy and r_copy, so that these actions are forced to synchronize.

We verified requirement 3.1 by checking the absence of c_home in the generated LTSs. This is formulated in the regular alternation-free μ-calculus [18] as follows:

3.1 [T*.c_home] F

It says that if an execution sequence contains c_home, then in the resulting state false holds. This formula was checked to be true by Evaluator, a model checker among CADP.

For requirement 3.2, a stable state of a system means that no protocol lock is held, and that the message queues are empty. We added actions *homequeue_empty* and *remotequeue_empty* to the μCRL specification of queues to indicate that queues are empty, and added an action *lock_empty* to the specification of the protocol lock manager to indicate that no lock is held. Then for a model with two processors, we checked that the generated LTS does not contain a state which can perform c_copy, lock_empty, homequeue_empty and remotequeue_empty. This requirement is presented in the regular alternation-free μ-calculus as follows:

3.2 $\neg$ <T*> (<c_copy>T $\wedge$ <lock_empty>T $\wedge$ <homequeue_empty>T $\wedge$ <remotequeue_empty>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 may 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, the home node may migrate (by a Region_Sponmigrate message) to the processor 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 the answer message. In the resulting state of the protocol, neither of the two processors is the home of the region. So c_copy may happen even in a stable state. The chosen solution is that when a processor gets a Region_Sponmigrate message, it informs those local threads that are writing to the region at the previous home node, so that these threads will behave as writing at home. After fixing this problem as proposed, property 3.2 was successfully model checked.

5.4.4 Requirement 4

The fourth requirement, that requests of writing to or flushing a region cannot be bounced around the network forever, is a liveness property. 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.

4.1 A thread eventually finishes writing to a region:

$$[T^*.\mathrm{write}(\star)]\ \mu X \cdot \langle T\rangle T \wedge [\neg \mathrm{writeover}(\star)]\ X$$

4.2 A thread eventually finishes its flush of a region:

$$[T^*.\mathrm{flush}(\star)]\ \mu X \cdot \langle T\rangle T \wedge [\neg \mathrm{flushover}(\star)]\ X$$

We use '$\star$' to indicate an identifier of a thread. These two formulas express that after a thread initiates its action (writer($\star$) or flush($\star$)), the end of this action (writeover($\star$) or flushover($\star$)) is inevitable. This requirement was successfully model checked on two configurations.

5.5 Verification Results

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 benefitted a lot from the μCRL distributed LTS generation tool, and also pushed forward its development.

Config.	states	transitions	Req. Checked
1	65,234	460,162	1, 2, 3, 4
2	5,424,848	40,476,069	1, 2, 3, 4
3	82,371,105	893,181,444	1, 2

Table 8. Verification results

The sizes of the generated LTSs and the verification results are summarized in Table 8. 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 toolset, we could generate LTSs for the following three configurations: 1) two processors, each with one thread; 2) two processors, one with one thread, the other with two threads; 3) three processors, each with one thread. For the third configuration, we could only check the first two requirements, because the generated LTS was too large to serve as input to the model checker. The shortest error traces for the two flaws in the original implementation of the protocol that were detected during the model checking phase (see Section 5.4) both consisted of more than 100 transitions.

6 Conclusions and Future Work

In this paper, we used formal specification and model checking techniques to analyze a cache coherence protocol for a Java DSM implementation. We specified the protocol in μCRL and analyzed it. Some general requirements were formulated and verified for several configurations. Our analysis uncovered a lot of inconsistencies between the description and the implementation of this protocol. Two errors were found and fixed in the implementation, which improved the design and implementation of this protocol.

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 were 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 the inconsistencies and misunderstandings. Third, some error traces were too long to be analyzed; a simulation tool that helps to automatically execute and interpret such long traces is needed. Fourth, more advanced tools are needed to generate, store and reduce LTSs. Our future work will mainly focus on verifying whether the cache coherence protocol implements the JMM in [9, Chapter 17], and checking the requirements on more configurations.

Acknowledgments

We would like to thank Stefan Blom, Jan Friso Groote, Natalia Ioustinova, Izak van Langevelde, Jaco van de Pol and Judi Romijn for valuable discussions.

References

[1] T. Arts and I. v. Langevelde. Correct performance of transaction capabilities. In *Proceedings 2nd Conference on Application of Concurrency to System Design*, pages 35–42. IEEE Computer Society, 2001.

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

[3] S. Blom, W. Fokkink, J. Groote, I. v. Langevelde, B. Lisser, and J. van de Pol. μCRL: A toolset for analysing algebraic specifications. In *Proceedings 13th Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer-Verlag, 2001.

[4] M. Broy, S. Merz, and M. Spies, editors. *Formal Systems Specification: The RPC-Memory Specification Case Study*, volume 1169 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[5] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proceedings 12th Conference on Computer-Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68. Springer-Verlag, 2000.

[6] 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 *Proceedings 1991 ACM/IEEE Conference on Supercomputing*, pages 197–206, 1991.

[7] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP – a protocol validation and verification toolbox. In *Proceedings 8th Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer-Verlag, 1996.

[8] W. Fokkink, N. Ioustinova, E. Kesseler, J. van de Pol, Y. Usenko, and Y. Yushtein. Refinement and verification applied to an in-flight data acquisition unit. In *Proceedings of 13th Conference on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, 2002.

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

[10] J. Groote and A. Ponse. The syntax and semantics of μCRL. In *Proceedings 1st Workshop on the Algebra of Communicating Processes*, Workshops in Computing Series, pages 26–62. Springer-Verlag, 1995.

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

[12] T. Henzinger, S. Qadeer, and S. Rajamani. Verifying sequential consistency on shared memory multiprocessor systems. In *Proceedings 11th Conference on Computer-Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 301–315. Springer-Verlag, 1999.

[13] J. Hooman and J. v. d. Pol. Formal verification of replication on a distributed data space architecture. In *Proceedings ACM 2002 Symposium on Applied Computing, special track on Coordination Models, Languages and Applications*, 2002.

[14] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings USENIX Winter 1994 Conference*, pages 115–132, 1994.

[15] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Transaction on Computers*, 28, 1979.

[16] J. Maessen, Arvind, and X. Shen. Improving the Java memory model using CRF. In *Proceedings 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2000.

[17] J. Manson and W. Pugh. Core semantics of multithreaded Java. In *Proceedings ACM 2001 Java Grande Conference*, 2001.

[18] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. In *Proceedings 5th Workshop on Formal Methods for Industrial Critical Systems*, 2000.

[19] J. v. d. Pol and M. Valero Espada. Formal specification of JavaSpaces$^{\mathrm{TM}}$ architecture using μCRL. In *Proceedings 5th Conference on Coordination Models and Languages*, volume 2315 of *Lecture Notes in Computer Science*, pages 274–290. Springer-Verlag, 2002.

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

[21] A. Roychoudhury and T. Mitra. Specifying multithreaded Java semantics for program verification. In *Proceedings ACM SIGSOFT Conference on Software Engineering*, 2002.

[22] X. Shen, Arvind, and L. Rodolph. Cachet: an adaptive cache coherence protocol of distributed shared memory systems. In *Proceedings 13th ACM Conference on Supercomputing*, 1999.

[23] J. Stoy, X. Shen, and Arvind. Proofs of correctness of cache-coherence protocols. In *Formal Methods for Increasing Software Productivity: Proceedings Symposium of Formal Methods Europe*, volume 2021 of *Lecture Notes in Computer Science*, pages 43–71. Springer-Verlag, 2001.

[24] R. Veldema, R. Hofman, R. Bhoedjang, and H. Bal. Runtime-optimizations for a Java DSM. In *Proceedings ACM 2001 Java Grande Conference*, pages 89–98, 2001.

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

[26] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Analyzing the CRF Java memory model. In *Proceedings 8th*

Asia-Pacific Software Engineering Conference, pages 21–28, 2001.

[27] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Specifying Java thread semantics using a uniform memory model. In *Proceedings ACM 2002 Java Grande Conference*, 2002. To appear.

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