

UMT Artificial Intelligence Review (UMT-AIR)

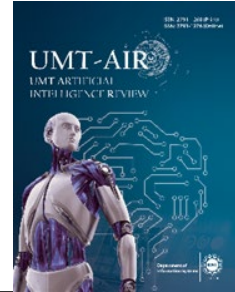
Volume 2 Issue 2, Fall 2022

ISSN_(P): 2791-1276 ISSN_(E): 2791-1268

Homepage: <https://journals.umt.edu.pk/index.php/UMT-AIR>



Article QR



Title: Formal Analysis of Distributed Shared Memory Algorithms

Author (s): Muhammad Atif¹, Mudasser Naseer², Ahmad Salman Khan³


Affiliation (s): ¹The University of Lahore, Pakistan
²Higher Colleges of Technology, UAE.
³The University of Lahore, Pakistan.

DOI: <https://doi.org/10.32350.umt-air.22.02>

History: Received: October 30, 2022, Revised: November 25, 2022, Accepted: December 12, 2022

Citation: M. Atif, M. Naseer, and A. S. Khan, "Formal analysis of distributed shared memory algorithms," *UMT Artif. Intell. Rev.*, vol. 2, no. 2, pp. 00–00, 2022, doi: <https://doi.org/10.32350.umt-air.22.02>

Copyright: © The Authors

Licensing:  This article is open access and is distributed under the terms of [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/)

Conflict of Interest: Author(s) declared no conflict of interest



A publication of

Department of Information System, Dr. Hasan Murad School of Management
University of Management and Technology, Lahore, Pakistan

Formal Analysis of Distributed Shared Memory Algorithm

Muhammad Atif^{1*}, Mudassar Naseer¹ and Ahmad Salman Khan²

¹Department of Computer Science and Information Technology, The University of Lahore, Pakistan

²Department of Software Engineering, The University of Lahore, Pakistan

Abstract—The memory coherence problem occurs while mapping shared virtual memory in a loosely coupled multiprocesses setup. Memory is considered coherent if a read operation provides the same data written in the last write operation. The problem has been addressed in the literature using different algorithms, although the correctness of a distributed algorithm remains questionable. Formal verification is the principal term for a group of techniques that routinely use an analysis established on mathematical transformations to conclude the rightness of the hardware or software behavior in divergence to dynamic verification techniques. The current study employed UPPAAL model checker to model the Dynamic Distributed algorithm for shared virtual memory given by K. Li and P. Hudak. The results showed that the Dynamic Distributed algorithm for shared virtual memory partially fulfils its functional requirements.

Index Terms—distributed algorithm, formal specification, verification, shared memory, virtual memory

I. Introduction

The idea of virtual memory becomes inevitable/indispensable when a system requires more memory than installed. Virtual memory comprises the usage of other than the main memory as the main memory. In shared virtual memory, physically separated memories (on the network) are shared among the processes connected through a loosely coupled fashion. Different processes may use shared virtual memory like the traditional virtual memory, as shown in Figure 1.

This paper investigates the Dynamic Distributed Memory Management algorithm given in [1], where other approaches, such as centralized manager, fixed, and broadcast are also given. Implementing the centralization algorithm becomes challenging when all of the traffic passes through a central manager for each type of page fault. An algorithm appears to have the best required results a namely the Dynamic Distributed Manager algorithm.

Corresponding Author: muhammad.atif@cs.uol.edu.pk

Department of Information Systems

Volume 2 Issue 2, Fall 2022



The Dynamic Distributed Manager algorithm is comparatively better than other algorithms, when there are a lot of page faults and network traffic needs to be managed in an efficient manner. The performance of this algorithm shows that it is probable/possible to implement it on a huge scale multiprocessor. However, the functional requirement of this algorithm needs to be verified by using formal methods [2], [3]. Formal Method is a standard word/term for system scheme [4], investigation, and application methods designated and used with scientific precision [5], [6].

(Construction and Analysis of Distributed Processes) toolbox was used to design and implement the model. In [5], Johan. B et al. modeled the memory management system of virtual memory with MSVL tool. Memory management system is formalized via MSVL (Modeling Simulation and Verification Language) using the Model Checking (MC) approach. This approach is applied to verify the perfection, delay linked properties and regular repeated properties. Munez et al. presented the formal verification of a sequentially consistent memory model, where low level functions are considered as sequential [5]. In [5], Kim G. Larsen et al. performed model checking using UPPAAL and verified the audio protocol. Several researchers have described the importance of structures in UPPAAL for model checking [8]–[10]. Another integration of formal verification through Cyber-Physical Systems (CPS) design process is presented in the literature. It consists of executing the transformation of AADL (Architecture Analysis and Design Language) models and represents them in timed automata. This approach was analyzed through model checking [11].

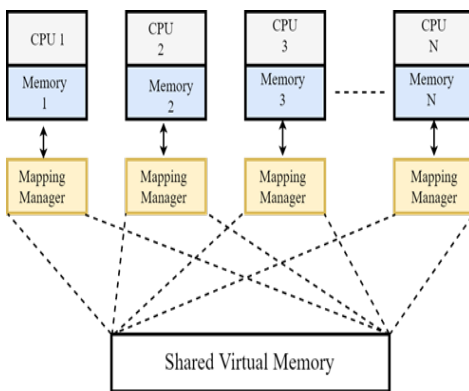


Fig. 1. Shared virtual memory [1]

II. Literature Review

In [7], Venkateswarlu Chennareddy et al. verified a weak consistency model of distributed shared memory. CADP

In [12], the authors introduce DSMC (Deep Statistical Model

Checking) to MECHATRONICUML, which is labelled as DSML (Domain Specific Model Checking) for the cyber physical systems.

III. Memory Coherence Problem

A single address space is shared by several processes in a shared virtual memory on the network, as shown in Figure 1. All processes are allowed to directly access any memory address in the address space. Memory Mapping Manager controls the implementation of mapping between shared virtual memory address space and local memories. Major responsibilities of a manager include to protect the system from the memory coherence problem. Its prevention is necessary to ensure that a read operation value on all processes remains the same as the most recent write operation.

Address spaces of shared virtual memory are divided into pages. Pages are a point to a memory block. Multiple copies of a page may exist over a network as read only however, for writing a page, it is assumed that there's only one copy and all other copies are invalidated.. Memory Mapping Manager scans the local memory as well as the address spaces of attached processes from the shared virtual memory cache. A page fault occurs due to memory reference

when the page memory location is not in the current physical memory of the process. So, in case of a page fault, memory manager rescues the page. It gets a page from the disk or via any other process. If another process has copies of the faulting memory page reference, then the manager needs to put in some effort to maintain memory coherence. The memory coherence problem might be encountered as these algorithms maintain the memory. A shared virtual memory on loosely coupled systems has no shared memory in physical form and the communication budget between the processes is non-trivial.

A. *Dynamic Distributed Memory Management*

Dynamic Distributed algorithm is a type of Distributed Manager algorithms in which tasks are divided among individual processes. In this algorithm, every process has its local table for maintaining the ownership of all pages, which is known as PTable. This PTable has five columns namely page ID, access field, copy set, probowner, and lock field [13].

- i. Page ID is the unique ID of page.
- ii. Access field shows the page accessibility roles, that is, either read or write.

- iii. Copy set contains the IDs of the processes having the copies of the page.
- iv. Probowner means a the possible owner of a page.
- v. Lock field is used to avoid the race condition between/among processes demanding the same page.

In this algorithm, probowner is set in a way that there remains no loop for pointing it out. For example, it is not possible that a node A says that probowner is the node B or the node B says that probowner is the node A.

In this protocol, every node sends requests to its probowner. If probowner is the actual owner then it replies back. Otherwise, it forwards the request to probowner. Eventually, a page is served by the actual owner.

Read Operation: Two processes are involved in each read operation. One is read fault handler (which comprises the request for read access) and the other is read server (which is specified in the probowner field). For read access, fault handler requests a process mentioned by the probowner field. If read server is the true owner of the requested page, then it needs to do the following operations:

- i. Add itself to the copy set of the requested page.
- ii. Change access to “Read” in its PTable.
- iii. Send page and page copy set to faulting process.
- iv. Add faulting process into the probowner field of its PTable.

If read server is not the true owner of the requested page, then it forwards the request to the process mentioned in the probowner field of its PTable. It also updates its probowner field with the requested node. Every time a faulting process receives a page copy, it updates its PTable along with its probowner field with “self” and changes access to “read”.

Write Operation: Write operation also works the same as the read operation, except invalidating pages according to the copy set. Two processes are involved in each write operation. One is write fault handler (which requests for the write access) and the other is write server (which is specified in the probowner field). For write access, fault handler requests to a process mentioned by the probowner field. If write server is the true owner of the requested page, then it needs to do the following operations:

- 1) Change access to “nill” in its PTable.
- 2) Send page and page copy set to the faulting process.
- 3) Add requested process into the probowner field of its PTable.

If the write server is not the true owner of the requested page, it forwards the request to the process mentioned in the probowner field of its PTable. It also updates its probowner field with the requested node. When faulting process receives the page copy; firstly, it invalidates all copies from the copy set.

III. Formal Specification

In Dynamic Distributed Manager algorithm of shared virtual memory, there are synchronized processes that have been discussed previously. The current study cultivates models for every synchronized process. It practices the UPPAAL tool suit [13], [14] for modelling these processes. Modelling the Dynamic Distributed Manager algorithm of shared virtual memory turned out to be suitable in definite situations Structures of UPPAAL with broadcast frequencies and dedicated positions let broadcast communication being categorized as the atomic arrangement of identical process organizations. The foremost

apprehension investigated in this paper is to demonstrate the formal analysis of [1] through the use of UPPAAL. It delivers a complete examination of numerous protocol varieties in relation to the verification of complete functional requirements.

Let us discuss the summary of prescribed requirements in the toolset UPPAAL and the formalism which is castoff in the prescribed requirement of the Dynamic Distributed Manager algorithm. For demonstrating Dynamic Distributed Manager algorithm in UPPAAL, two local processes were generated along with a manager. These processes request as well as serve all pages and generate both the read and write requests. These processes perform the following tasks:

1. Generate a read fault.
2. Handle a read fault request.
3. Forward a read request to probowner.
4. Generate a write fault.
5. Handle a write fault request.
6. Forward a write request to probowner.
7. Invalidate pages upon giving up ownership.

A process named as invalidate-process was modeled to address all

the requests from all the processes when they want to invalidate the old copies of pages. Essentially, the invalidate-process behaves like a buffer to process requests one by one. A process invalidates pages according to the copy set, while transferring the ownership of a page. It shows that a page is going to be updated and previously used copies of that page are invalidated. A process gets the updated copy of a page by generating a read fault request.

IV. Results and Discussion

The prescribed analysis of the Dynamic Distributed Manager algorithm is presented in [14]. The specification of this distributed algorithm in an automaton theoretic formalism is formalized and functional requirements are verified.

A. Functional Requirements

The algorithm has the following functional requirements for formal analysis and formal verification.

[R1]: Deadlock freedom. No deadlock is supposed to be there when any process requests for the read or write page in the system. System does not hang while anyone requests for read, write, or broadcast invalidate request.

[R2]: Any process can get the read access of any page.

[R3]: Any process can get the write access of any page.

[R4]: When a process requests for the read page, then it must get the read access of the page. The true owner of the page must send the page copy to the requested process.

[R5]: When a process requests for the write page, then it must get the write access of the page. The true owner of the page must send the page write access to the requested process.

B. Formal Specification of the Requirements

The principal requirement is that system does not contain any deadlock. According to the requirement, there is no valid deadlock in the system. The model must be deadlock free. So, the query to verify this requirement is given below.

- A [] not deadlock

The query says that for all paths and states, there is no deadlock. In the query, 'A' represents all paths and '[]' represents all states. Any process in the system can get the read access of any page, indicating that any process can send read request for any page. Then, the true owner of the page sends the page access to the process page.

The query to verify R2 is given below.

- $E \langle \rangle \text{ forall } (i:\text{pro_id_t}) \text{ forall } (j:\text{page_id_t}) \text{ Process}(i).\text{PTable}[j][1]==1$

This query uses nested loop. The outer loop is for the process and the inner loop is for the page. The query checks the read access from the PTable. It verifies the read access for all pages under each process. As described earlier, Index 1 shows the page access value, where 0, 1, and 2 represent the nil, read, and write access, respectively.

Any process in the system can get the write access of any page. Similarly, any process can send the write request for any page. The true owner of the page sends the page for the write access to the requesting process. The query to verify R3 is given below.

- $E \langle \rangle \text{ forall } (i:\text{pro_id_t}) \text{ forall } (j:\text{page_id_t}) \text{ Process}(i).\text{PTable}[j][1]==2.$

In the query, ‘E’ represents ‘some path’ and ‘ $\langle \rangle$ ’ represents ‘some state’. This query is similar to the query in R2, except it checks the write access. For the write access, the value of Index 1 in PTable must be equal to 2. When the process requests to read a page, it must get the read access of that page. So, according to this

requirement, when a process requests to read a page, i.e., it is at the readFault state then it is supposed to reach the Ideal state, eventually.. The formula of R4 requirement is given below.

- $\text{Process } (1). \text{ReadFault} \text{ --> Process}(1). \text{Ideal}$

According to the R4 requirement, when Process (1) reaches the readFault state, it definitely goes back to the ideal state. When the process requests for a write page, it must get the write access of that page. So, according to this requirement, when a process requests to write a page, the process goes to the writeFault state and eventually reach to the Ideal state. The formula of R5 requirement is given below.

- $\text{Process } (1).\text{writeFault} \text{ --> Process}(1). \text{Ideal}$

According to R5, when the Process (1) reaches the writeFault state it definitely goes back to the ideal state.

C. Verification Results

The current model was verified with respect to the given functional requirements and the results are shown below in Table 1. The technique of verifying a model using timed automata is applied in the same way [15]. An algorithm

with 3 processes and 8 pages is modelled.

Table I
Verification Results

Requirements	Results	Time	Memory
R1	Satisfied	19h25s	2.16GB
R2	Satisfied	0.046	102MB
R3	Satisfied	0.001s	29.16MB
R4	Satisfied	11.032s	100MB
R5	Satisfied	13.172s	102MB

The current authors faced some serious challenges related to machine power in verifying and validating these requirements. Firstly, it was executed on a machine with the specification, Windows 10, 8GB RAM, and Core i5 7th generation. On this machine, the model was executed for 20 minutes and then it crashed due to the state space problem. This query was also executed on MacBook 2016 with 16 GB RAM, where it ran for around 4 hours and then crashed. So, its execution on a more powerful machine is needed to verify this requirement. The other four requirements were satisfied with 2 processes and 2 pages. The results are presented in Table I, where the time needed to verify a requirement and the memory used are shown.

Acknowledgement

We acknowledge the anonymous reviewers who helped us to improve this article. We are

also thankful to the University of Lahore for providing research facilities.

References

- [1] P. Hudak and K. Li, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Sys.*, vol. 7, no. 4, pp. 321–359, Nov. 1989, doi: <https://doi.org/10.1145/75104.75105>
- [2] C. Baier and J.P. Katoen, *Principles of model checking*, MIT Press Cambridge, 2008.
- [3] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Comput. Sur.*, vol. 28, no. 4, pp. 626–643, Dec. 1996, doi: <https://doi.org/10.1145/242223.242257>
- [4] N. Ibrahim and I. Khalil, "Verifying web services compositions using Uppaal," in *Proc. 1st IEEE Int. Conf. Comput. Sys. Indust. Inform.*, Sharjah, UAE, Dec. 18–20, 2012, doi: <https://doi.org/10.1109/ICCSII.2012.6454365>
- [5] J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Verification of an audio protocol with bus collision

- using uppaal,” in *Proc. Comput. Aid. Verific. 8th Int. Conf.*, CAV'96 New Brunswick, USA, July 31–August 3, 1996, pp. 244–256.
- [6] A. Blanchard, N. Kosmatov, M. Lernerre and F. Loulergue, “A Case Study on Formal Verification of the Anaxagoras Hypervisor Paging System with Frama-C,” in *Proc. FMICS Formal Meth. Indus. Critic. Sys. 20th Int. Work.*, Oslo, Norway, June 22–23, 2015, pp.15–30.
- [7] V. Chennareddy and J. K. Deka, “Formally Verifying the Distributed Shared Memory Weak Consistency Models,” in *Proc. Int. Conf. Adv. Comput. Commun.*, Mangalore, India, Dec. 20–23, 2006, pp. 455–460, doi: <https://doi.org/10.1109/ADCO M.2006.4289935>
- [8] A. F. G. David, K. G. Larsen, A. Legay, M. Mikucionis, and D. B. Poulsen, “Uppaal smc tutorial,” *Int. J. Softw. Tools. Technol. Transfer.*, vol.17, pp. 397–415, 2014, doi: <https://doi.org/10.1007/s10009-014-0361-y>
- [9] D. Fabian and R. Marik, “Configuration dynamics verification using Uppaal,” in *Proc. 15th Int. Config. Works.*, Vienna, Austria, August 29–30, 2013, pp. 35–42.
- [10] Y. Fei, H. Zhu, and X. Li, “Modeling and verification of nlsr protocol using uppaal.” in *Proc. Int. Symp. Theor. Aspec. Soft. Eng.*, Guangzhou, China, Aug. 29–31, 2018, pp. 108–115, doi: <https://doi.org/10.1109/TASE.2018.00022>
- [11] F. S. Gonçalves, D. Pereira, E. Tovar, and L. B. Becker, “Formal Verification of AADL Models Using UPPAAL,” in *Proc. VII Braz. Symp. Comput. Sys. Eng.*, Curitiba, PR, Brazil, Nov. 6–10, 2017, pp. 117–124, doi: <https://doi.org/10.1109/SBESC.2017.22>
- [12] C. Gerking, S. Dziwok, C. Heinzemann, and W. Schäfer, “Domain-specific model checking for cyber-physical systems,” in *Proc. 12th Work. Model-Driven Eng., Verific. Valid.*, Ottawa, Canada, 2015, pp. 18–27.
- [13] P. Bulychev, et al., “Uppaal-smc: Statistical model checking for priced timed automata,” in *Proc. Workshop Quanti. Aspec. Program. Lang. Sys.*, 2012, pp. 1–16, doi:

- <https://doi.org/10.48550/arXiv.1207.1272>
- [14] K. G. Larsen, P. Pettersson, and W. Yi, “Uppaal in a nutshell.” *Int. J. Softw., Tools Technol. Trans.*, vol. 1, no. 1, pp. 134–152, 1997.
- [15] S. Wimmer and P. Lammich, “Verified model checking of timed automata,” in *Proc. Tools Algor. Construc. Anal. Syst.*, 2018, pp. 61–78, doi: https://doi.org/10.1007/978-3-319-89960-2_4