Proceedings of the Sixth International Conference on
Railway Technology: Research, Development and Maintenance
Edited by: J. Pombo
Civil-Comp Conferences, Volume 7, Paper 10.7
Civil-Comp Press, Edinburgh, United Kingdom, 2024
ISSN: 2753-3239, doi: 10.4203/ccc.7.10.7
©Civil-Comp Ltd, Edinburgh, UK, 2024

Model Refactoring for Spin-Lock Based Resource Sharing: A Case Study for Train Control and Management Software

B. E. Beygo[1], G. Dayı[1] and K. İmre[2]

[1]UGES, ASELSAN
Ankara, Türkiye
[2]Computer Engineering Department, Hacettepe University
Ankara, Türkiye

Abstract

Model-driven development is a widely used approach in various domains, and its popularity has been increasing in certain industries. Refactoring software models is essential to maintain and enhance the quality of software models. This paper focuses on refactoring an existing software model to achieve resource-sharing problems with the help of a spin-lock mechanism. Threads accessing the same resource might create problems if they read and write simultaneously. Locking is unnecessary if the shared resource is only read by all threads. Most of the time, the reading operation does not exist alone in the software projects, and the writing operation comes along with the reading operation. Problems arise when none of the synchronization mechanisms are used. In this work, we have worked on Unified Modeling Language diagrams to inject a spin-lock synchronization mechanism. For this purpose, we have identified refactoring opportunities, also known as model smells, and applied refactoring techniques. We discussed the results of applied refactoring techniques to demonstrate the effects of the smells.

Keywords: model-driven development, spin-lock, UML, refactoring, threads, multithreading.

1 Introduction

Model-driven development is applied across various domains and is increasingly being adopted in specific industries, particularly those involved in managing complex and extensive systems. Refactoring software models can help address various challenges, such as improving model clarity, adapting to changing requirements, and enhancing the structure of the models. Model refactoring can be applied in many subjects to ensure the overall maintainability and quality of the models. Model refactoring spans various disciplines, ensuring comprehensive maintainability and quality of the models.

A spin-lock serves as a synchronization method primarily employed within operating systems to safeguard shared resources from simultaneous access by multiple threads or processes. When a thread or process has acquired the lock, it relinquishes it upon completion of the critical section, thus enabling another thread or process to obtain and utilize the resource. Unlike mutexes and semaphores that allow threads to sleep while waiting for a resource, a spin-lock works differently. Instead of sleeping, a thread continuously checks the lock's status in a loop until it becomes available. Spin-locks can be advantageous when threads contending for the lock have similar priorities and are running on different cores with affinity set accordingly.

Existing software models sometimes need to be refactored to obtain new capabilities or to fix the errors discovered after development. Code refactoring can also be applied to fix the errors, but code refactoring might not be efficient. In such cases, requisite model refactoring can be applied.

Resource sharing between threads refers to the practice of allowing multiple threads within a process to access and utilize shared resources concurrently. In multi-threaded programming, threads often need to access common resources such as memory, files, network connections, or hardware devices. Efficient and safe sharing of these resources is crucial to ensure correct program behaviour and performance. Resource sharing between threads typically involves synchronization mechanisms to coordinate access to shared resources and prevent data corruption or race conditions. Some common synchronization techniques include mutexes, semaphores, condition variables, read-write locks, and atomic operations. Properly managing resource sharing between threads is essential for writing correct and efficient concurrent programs. Careful design and use of synchronization primitives are necessary to avoid common issues such as deadlocks. Additionally, excessive synchronization can lead to performance bottlenecks, so balancing safety and performance is important. Since our architecture has efficient cache coherence protocols, the inter-core communication latency is low, so spin-lock is more effective for this work.

Our work focuses on refining an existing software model aimed at resolving resource-sharing issues through implementing a spin-lock mechanism. The model belongs to a train control and management software that runs on the central control unit of the system. The simultaneous reading and writing by multiple threads accessing the same resource can lead to complications. While locking is unnecessary if all threads merely read from the shared resource, in practice, reading operations often coincide with writing operations in software projects. Challenges arise when synchronization mechanisms are absent. In this study, we utilized Unified Modeling Language (UML) diagrams of train control and management software to integrate a spin-lock synchronization mechanism. We identified refactoring opportunities, commonly referred to as model smells, and implemented appropriate refactoring techniques. Through our analysis of the effects of these smells, we present the outcomes of the applied refactoring techniques. In short, this work's contribution is to fix the resource-sharing problem on existing software models designed with UML diagrams.

We preferred spin-lock over mutex because of the lower overhead. Spin-locks typically have lower overhead compared to mutexes. In a spin-lock, a thread actively waits in a loop until the lock becomes available, whereas a mutex puts the waiting thread to sleep until the lock is available. This sleeping and waking process in mutexes involves more overhead due to context switching. Spin-locks can also have lower latency compared to mutexes, especially in scenarios where contention for the lock is infrequent or short-lived. In such cases, spinning may be more efficient than putting a thread to sleep and later waking it up. In our case, we only lock the source for the copying process. Each thread has its source, which they use for internal operations. Data is copied, according to their read or write labels, from threads' sources to the central source at the start and end of each cycle. The locks are free during times other than copying. As a result, our locks live in a very short time. In this way, we avoided context switching and we achieved more deterministic behaviour.

After the refactoring, there are data structure collections for each thread and a single central data structure collection. Our first approach was to lock data structures as a whole but we knew that spin-locks are more efficient if the lock is infrequent or short-lived. So, we decided to group the data structure collection by their update periods. We applied a distribution on the timeline. Data structure groups that have greater periods are split over other cycles of execution. This distribution operation is done to some of the threads. The threads that needed a few data structure groups did not need a distribution since they ran their cycle already in a short time. With this distribution operation, we achieved more deterministic behaviour.

The remainder of this paper is organized as follows; in Section 2, related works in the literature are listed. In Section 3, we detailed the refactoring opportunities and techniques. Refactoring results are presented in Section 4, and finally, Section 5 presents the conclusion.

2 Related Works

In [1], the authors proposed an integrated approach to model-driven refactoring by selecting different UML diagrams to represent various views. The views are combined and integrated to detect more bad smells. The research [2] explores the model smells, and highlights commonly encountered smells. In addition, a list of behaviour-preserving techniques is introduced. In [3], the authors introduced functional decomposition as the dominant cause of design smells, and they applied a machine-learning technique to UML models to recognize functional decomposition. Analysis of challenges in model refactoring has been done in [4].

In [5], the authors presented a new lock-based resource-sharing protocol called preemptable waiting locking protocol that can be applied in partitioned and global scheduling scenarios. In [6], to resolve the resource allocation problem, a FIFO (first-in, first-out) multi-resource lock algorithm for shared memory multiprocessors is proposed. The investigated techniques in [7] are mainly based on assigning and changing priorities of tasks in multiprocessor platforms. There are also two protocols proposed in [8] for handling resource sharing under semi-partition scheduling in multiprocessor platforms. In [9], a detailed investigation has been done for spin-based global resource-sharing protocols for multi-core systems based on partitioned fixed-priority scheduling.

3 Refactoring Opportunities and Techniques

Opportunities for model refactoring include optimizing the relationships between components, improving the granularity of modules, reorganizing the data schema for better efficiency or flexibility, or adopting a different architectural pattern to better suit evolving requirements. Model refactoring aims to enhance system maintainability, scalability, and adaptability while preserving the functionality and interfaces that users interact with. While refactoring, next to preserving the functionality, we focused on resolving the resource-sharing problem. Thus, the selection criteria are special for our perspective of uncovering this problem.

3.1 Inaccurate Activity

Textual labels typically accompany activities within a software model. However, labels are sometimes not enough to meet the needs. Activities may be renamed, new activities may be added, and misnamed activities may be removed entirely to inform readers about the use cases. In Figure 1, you can see the use case diagrams before the refactoring on the left, and after the refactoring on the right. Considering BusContoller, BusinessController, and ActivationController as our threads, simply reading from, and writing to data structure collection creates a data corruption problem which occurs when multiple threads read from and write to the same memory location.

To prevent data corruption and achieve synchronized data sharing, first, we needed to use more than one data structure collection. New data structure collections are created and assigned to the threads while we keep the central data structure collection as you can see in Figure 2. After deciding the number of collections, we renamed the labels of read and write activities with a single copy activity. Now, since we ensure the threads are processing their memory locations, each thread has to copy processed data to the central collection, and from the central collection. We needed to lock the sources for copying operations to prevent data corruption, so we added lock and release activities as you can see on the right side of Figure 1.

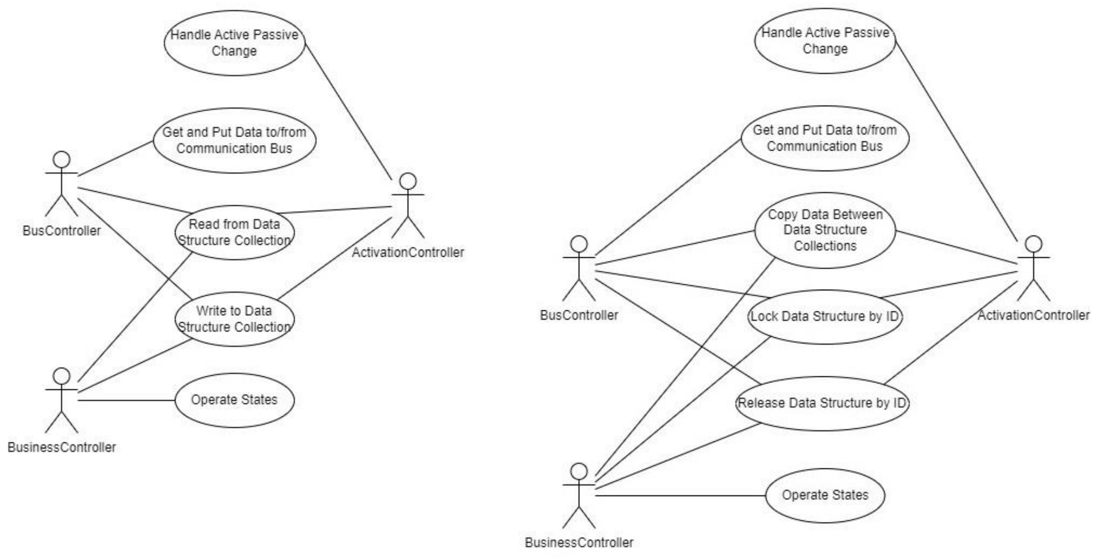
Handle Active Passive Change
Get and Put Data to/from Communication Bus
BusController
Read from Data Structure Collection
ActivationController
Write to Data Structure Collection
BusinessController
Operate States
Handle Active Passive Change
Get and Put Data to/from Communication Bus
Copy Data Between Data Structure Collections
BusController
Lock Data Structure by ID
ActivationController
Release Data Structure by ID
BusinessController
Operate States

Figure 1: Use case diagrams before and after the refactoring.

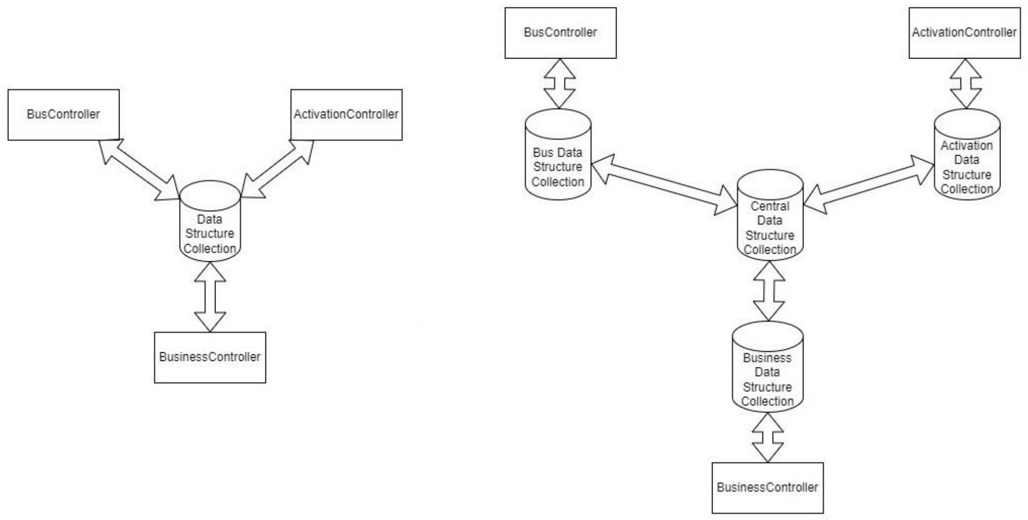
BusController
ActivationController
Data Structure Collection
BusinessController
BusController
ActivationController
Bus Data Structure Collection
Activation Data Structure Collection
Central Data Structure Collection
Business Data Structure Collection
BusinessController

Figure 2: Data structure collection usage before and after the refactoring.

3.2 Inaccurate Naming

After we added a data structure collection for each thread, we looked for some refactoring opportunities on the class diagram. CollectionManager as a class name was not efficient enough. As you can see on the left side of Figure 3, this class has control over only one data structure collection before the refactoring.

To reveal the intended purpose to readers, CollectionManager class is renamed to Proxy, and ICollectionManager interface is renamed to IProxy. The purpose of this new name is to form a slight demonstration because the refactored data structure collections created an ecosystem that resembles proxy systems. Functions to get the data by ID from collections, and functions to lock and release are also added to the class and interface.

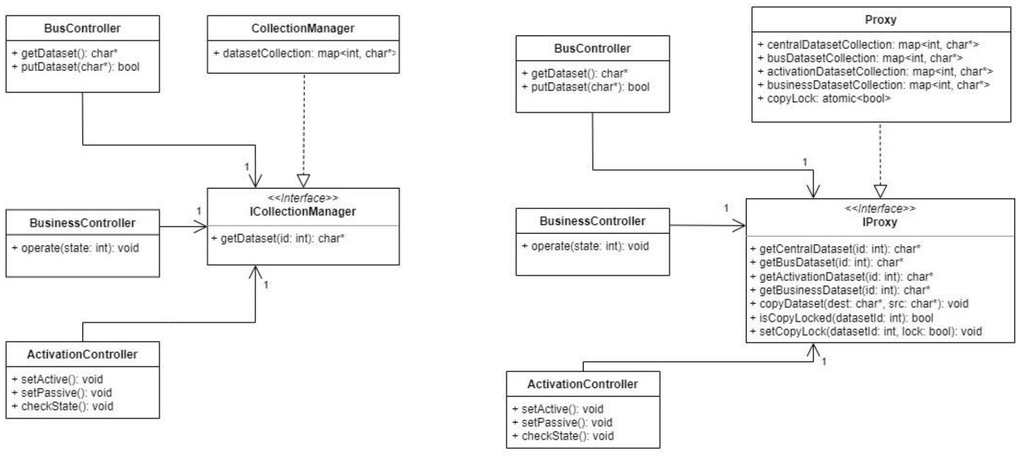
BusController
+ getDataset(): char*
+ putDataset(char*): bool
CollectionManager
+ datasetCollection: map<int, char*>
<<Interface>>
ICollectionManager
+ getDataset(id: int): char*
BusinessController
+ operate(state: int): void
ActivationController
+ setActive(): void
+ setPassive(): void
+ checkState(): void
1
1
1
Proxy
+ centralDatasetCollection: map<int, char*>
+ busDatasetCollection: map<int, char*>
+ activationDatasetCollection: map<int, char*>
+ businessDatasetCollection: map<int, char*>
+ copyLock: atomic<bool>
<<Interface>>
IProxy
+ getCentralDataset(id: int): char*
+ getBusDataset(id: int): char*
+ getActivationDataset(id: int): char*
+ getBusinessDataset(id: int): char*
+ copyDataset(dest: char*, src: char*): void
+ isCopyLocked(datasetId: int): bool
+ setCopyLock(datasetId: int, lock: bool): void
BusController
+ getDataset(): char*
+ putDataset(char*): bool
BusinessController
+ operate(state: int): void
ActivationController
+ setActive(): void
+ setPassive(): void
+ checkState(): void
1
1
1

Figure 3: Class diagrams before and after the refactoring.

3.3 Insufficient Processes

A flowchart is a visual representation of a process or system, to illustrate the flow of steps or activities from start to finish. There are three major threads for our software: activation, bus, and business threads. For those threads, start points exist but finish points do not because those threads are cyclic forever. They run to do some operations in their determined periods. Processes for preventing data corruption are missing. In our case, there must be processes to lock and release sources serving spin-lock operations. Also, decision points to display when to handle spin-lock operations are missing.

We have inserted lock, release, and copy processes to all threads as you can see in Figure 4, Figure 5, and Figure 6. In the figures, repository means data structure collection. Primary processes are preserved while inserting new processes. Each thread locks the source, copies data for reading from the central repository to its own repository, and releases the source. In addition, data structures are split into multiple lists for bus and business controller threads to distribute CPU load between thread cycles. Because of this, an additional process to determine the timed list of data structures exists for bus and business controller threads.

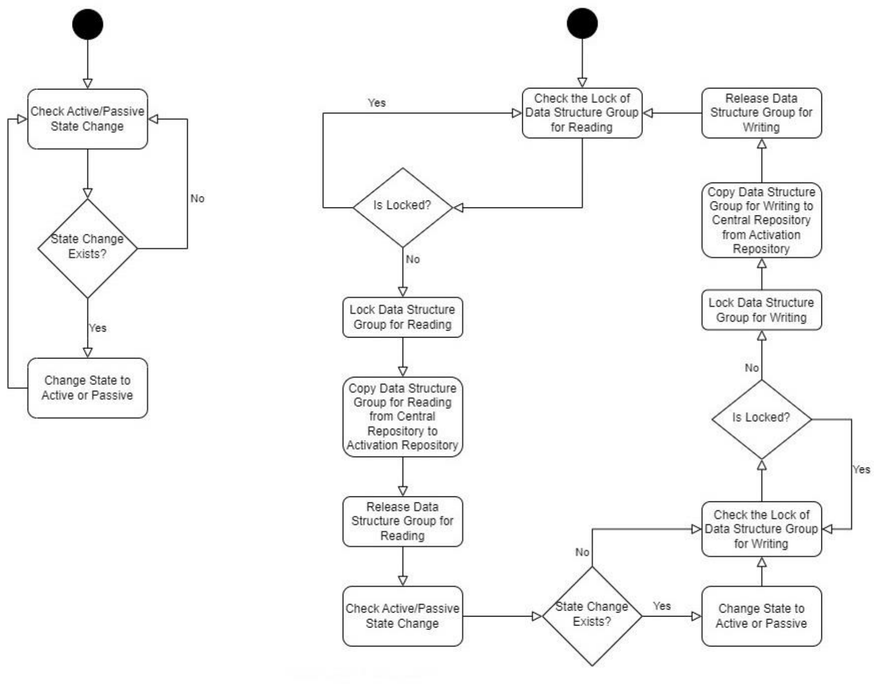
Check Active/Passive State Change
State Change Exists?
No
Yes
Change State to Active or Passive
Check the Lock of Data Structure Group for Reading
Yes
Is Locked?
No
Lock Data Structure Group for Reading
Copy Data Structure Group for Reading from Central Repository to Activation Repository
Release Data Structure Group for Reading
Check Active/Passive State Change
State Change Exists?
Yes
No
Change State to Active or Passive
Check the Lock of Data Structure Group for Writing
Is Locked?
Yes
No
Lock Data Structure Group for Writing
Copy Data Structure Group for Writing to Central Repository from Activation Repository
Release Data Structure Group for Writing

Figure 4: Flowcharts for activation controller thread before and after refactoring.

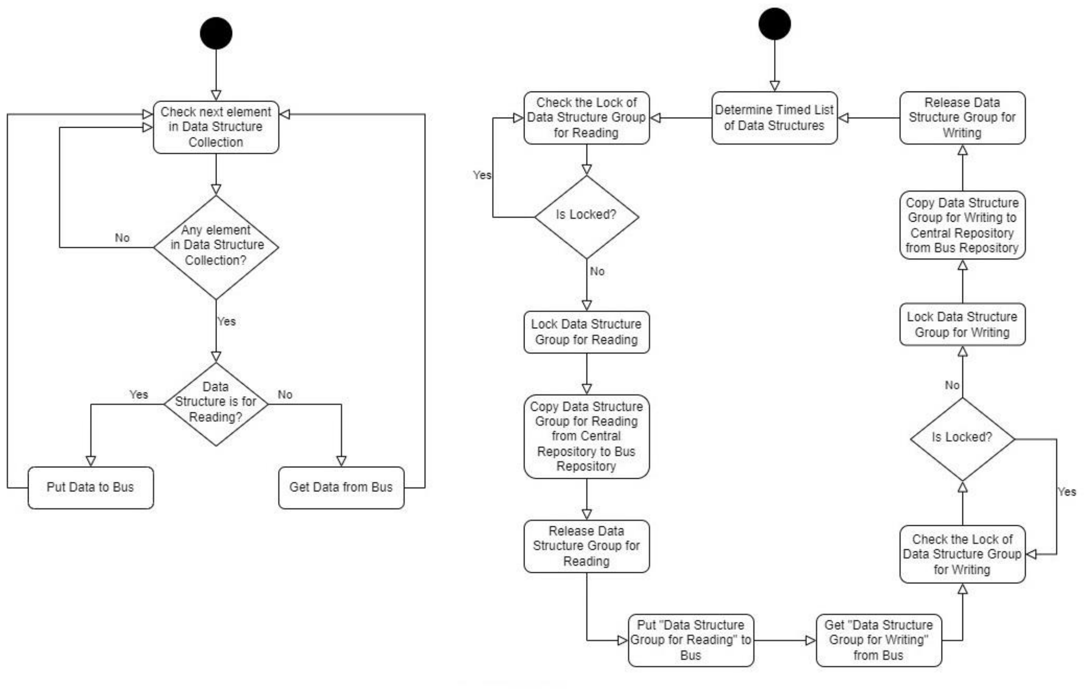
Check next element in Data Structure Collection
Any element in Data Structure Collection?
No
Yes
Data Structure is for Reading?
Yes
No
Put Data to Bus
Get Data from Bus
Determine Timed List of Data Structures
Check the Lock of Data Structure Group for Reading
Yes
Is Locked?
No
Lock Data Structure Group for Reading
Copy Data Structure Group for Reading from Central Repository to Bus Repository
Release Data Structure Group for Reading
Put "Data Structure Group for Reading" to Bus
Get "Data Structure Group for Writing" from Bus
Check the Lock of Data Structure Group for Writing
Is Locked?
Yes
No
Lock Data Structure Group for Writing
Copy Data Structure Group for Writing to Central Repository from Bus Repository
Release Data Structure Group for Writing

Figure 5: Flowcharts for bus controller thread before and after refactoring

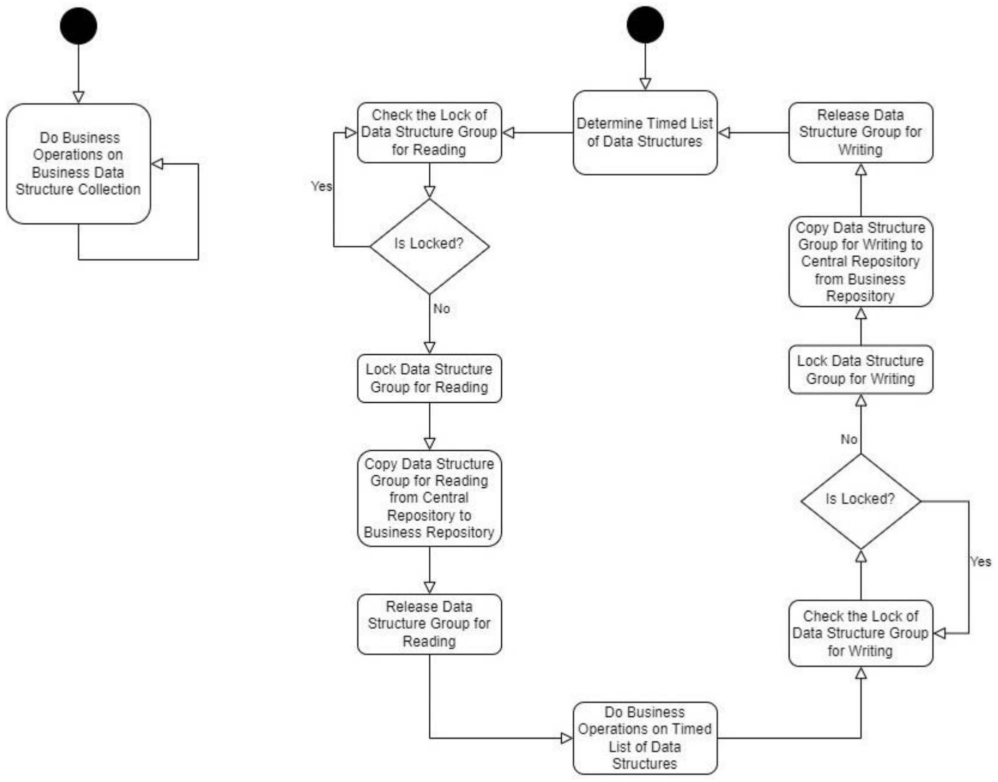
Do Business Operations on Business Data Structure Collection
Check the Lock of Data Structure Group for Reading
Determine Timed List of Data Structures
Release Data Structure Group for Writing
Yes
Is Locked?
No
Copy Data Structure Group for Writing to Central Repository from Business Repository
Lock Data Structure Group for Reading
Lock Data Structure Group for Writing
No
Copy Data Structure Group for Reading from Central Repository to Business Repository
Is Locked?
Yes
Release Data Structure Group for Reading
Check the Lock of Data Structure Group for Writing
Do Business Operations on Timed List of Data Structures

Figure 6: Flowcharts for business controller thread before and after refactoring.

3.4 Insufficient Dispatches

Sequence diagrams show the flow of messages or actions exchanged between objects or components in software over time. In our case, the sequence diagram depicts the interactions between the data structure collection manager and our three major threads. Each thread runs in separate loops by continuously requesting data by ID, and processing them as you can see in Figure 7. There are missing dispatches that handle spin-lock operations. Dispatches that take data by ID are also insufficient for spin-lock operations, they had to be renamed or replaced. Those dispatches are shown in Figure 7 as “getBuffer”.

Since we split data structures into multiple lists for bus and business controller threads, we needed to get data by group. Therefore, we added internal dispatches to determine the timed data group to the bus and business controller. For spin-lock operations, we added lock, copy, and unlock dispatches sequentially to all the threads. This dispatch group exists before and after the internal dispatches of threads. The newly added dispatch group before internal operations handles moving data from central data structure collection to the thread’s data structure collection. On the other hand, the newly added dispatch group after internal operations handles moving data from thread’s data structure collection to central data structure collection. Figure 8 shows the refactored sequence diagram.

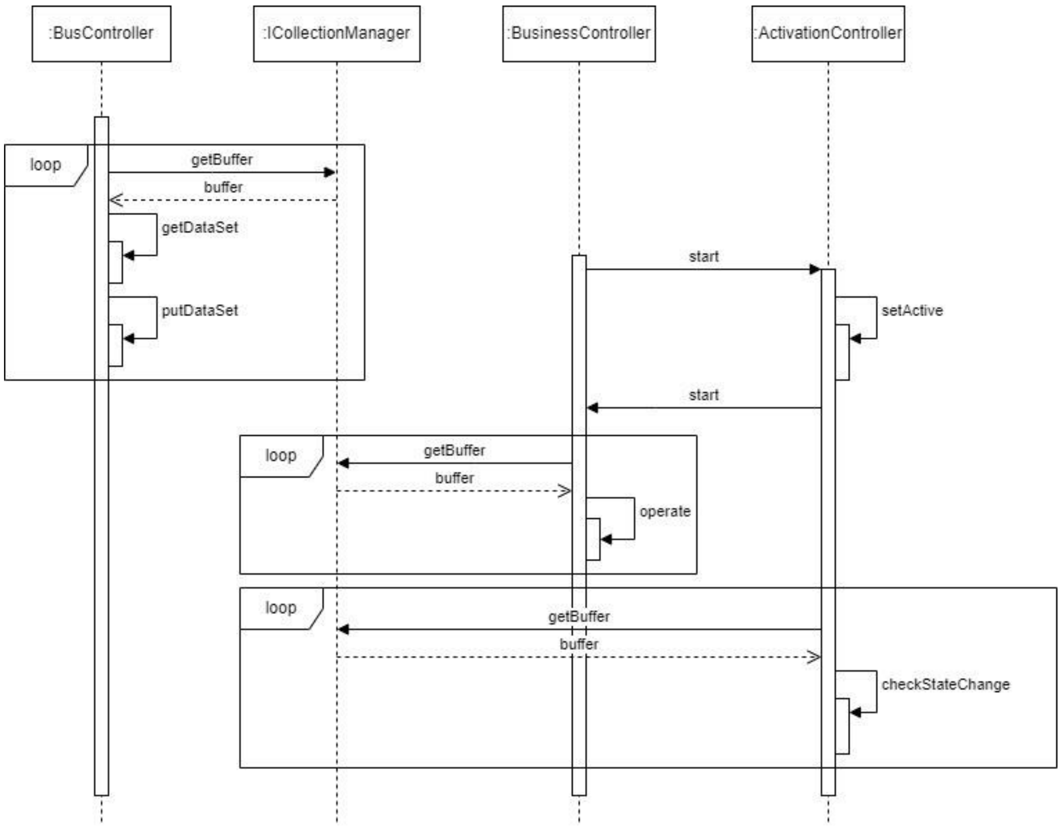

Figure 7: Sequence diagram before refactoring

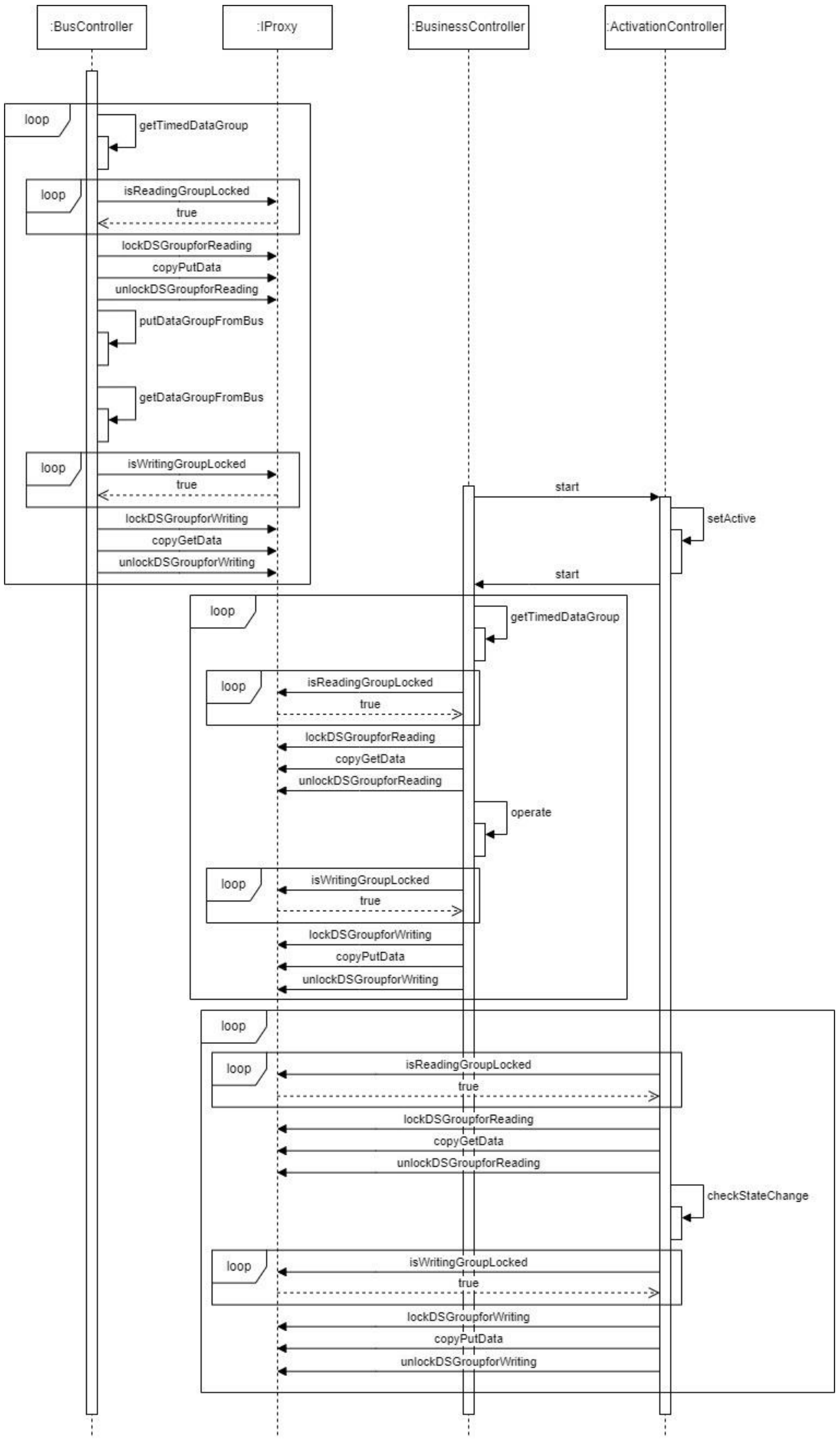

Figure 8: Sequence diagram after refactoring.

4 Refactoring Results

After the refactoring, the software's three major threads' external behaviour has not been changed while their internal structures have been improved. All of the three threads are operating their data structures the same as before the refactoring. The only differences are spin-lock steps, and data structure distribution steps between cycles. Data structure distribution operations are considered an improvement to reduce the number of spin times when the source is locked. Spin-lock steps include copying data from one data structure to another. Even if the copying looks like an extra step, the threads' cycle periods are long enough to include those copying operations, because threads are sleeping for their period after they complete their operations on the data structures before starting another cycle. The distribution operation made us ensure the threads do all their operations within their cycle times. The distribution operation is applied on the bus controller and business controller threads. We did not apply distribution on the activation controller thread because there are not so many data structures that the activation controller thread processes. The effect of distribution is not so perfect for business controller thread because nearly all data structures are processed in thread cycles. However, the major improvement is with the bus controller thread because the data structures can be grouped almost evenly between cycle periods. Data structures with ID numbers are put in a timeline in Figure 9 and Figure 10. Data structures with a period of 32 milliseconds are 100, 101, 102, 103; data structures with a period of 64 milliseconds are 200, 201, 202, 203; data structures with a period of 128 milliseconds are 301, 302, 303, 304, 305, 306; data structures with a period of 256 milliseconds are 400, 401; and 500 is a data structure with a period of 512 milliseconds.

	Activation Controller Thread	Bus Controller Thread	Business Controller Thread
Number of runs with lock	75	165	38
Number of runs without lock	393	303	430
Minimum spin with lock	1	1	2
Maximum spin with lock	1640	8819	387
Average spin with lock	419	316	105

Table 1: Spin results within 15 seconds.

Spin results which are measured within 15 seconds are shown in Table 1. The table contains the counts of cycles with and without locks; minimum, maximum, and average spins for three major threads. The activation controller thread runs 75 out of 468 cycles with a lock hit which means 84% of cycles are run without waiting for the lock. The bus controller thread runs 165 out of 468 cycles with a lock hit which means 65% of cycles in 15 seconds are run without waiting for the lock. This percentage becomes 92% for the business controller thread. These percentages are high enough to consider an improvement for our software because the threads exchange data approximately these percentages. When we look at the minimum spins for threads, 1 or 2 is perfect for 32 milliseconds of cycles. 1640, 8819, and 387 may be considered high for spins because spin-locks consume CPU resources while they spin, and it could lead to high CPU utilization and increased context switching between threads, especially on systems with many threads. However, our software does not include so many threads, so the average spins of our threads which are 419, 316, and 105 are fine results for our software.

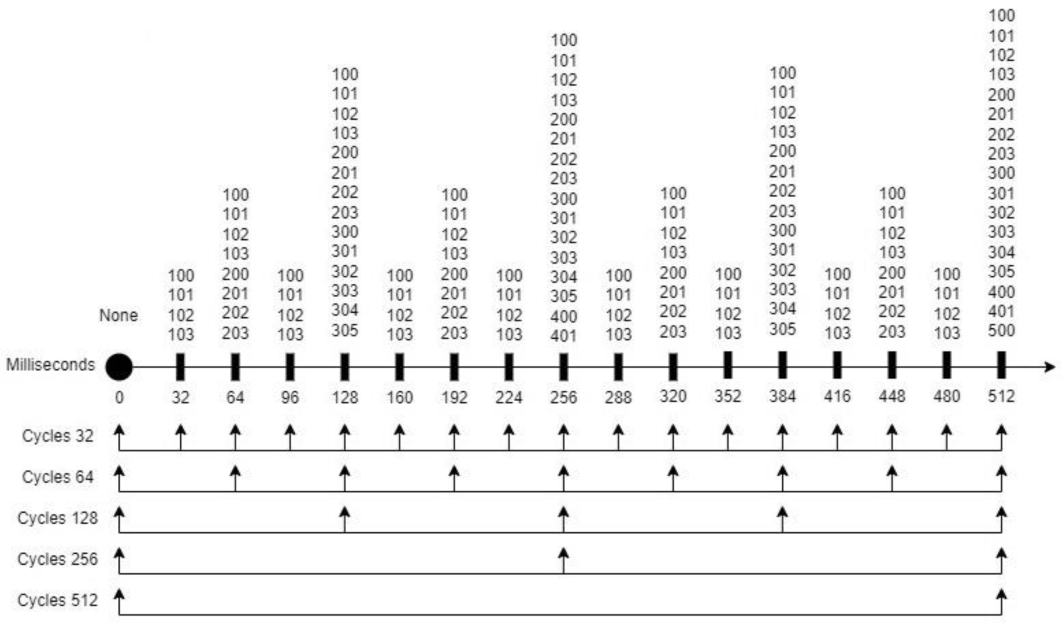

Figure 9: Timeline of bus controller thread before refactoring.

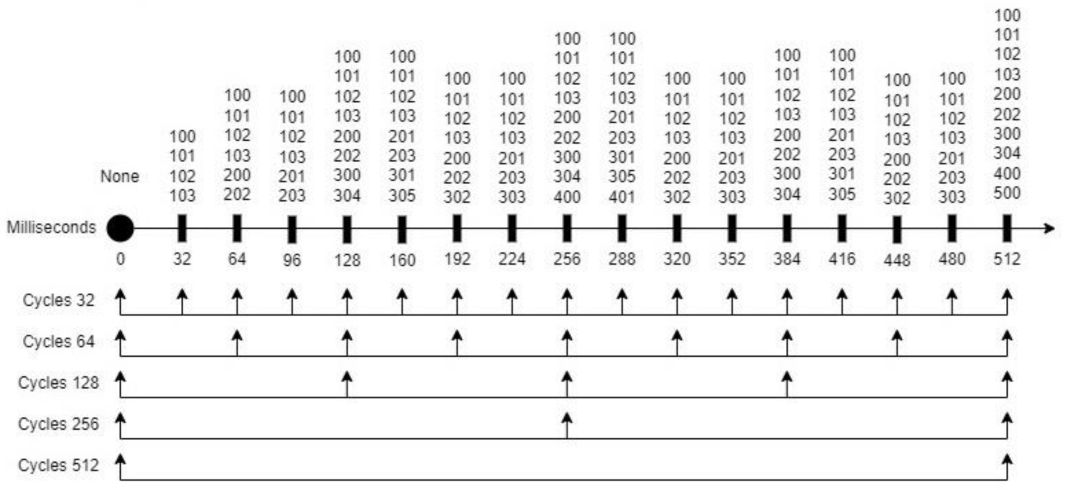

Figure 10: Timeline of bus controller thread after refactoring.

5 Conclusions and Contributions

We chose spin-lock over mutex for lower overhead. Spin-locks typically incur less overhead than mutexes as they keep threads active while waiting for the lock. This avoids the overhead of putting threads to sleep and waking them up, reducing latency, particularly in scenarios with infrequent contention. In our case, we lock the source only during the copying process, ensuring a brief lock lifespan and deterministic behaviour, avoiding context switching. Spin-locks prove beneficial when threads contending for the lock share similar priorities and execute on different cores with affinity configured accordingly. Given our architecture's efficient cache coherence protocols, inter-core communication latency remains minimal, rendering spin-locks notably effective for this task.

After refactoring, each thread now possesses its own set of data structure collections alongside a centralized one. Initially, we adopted a strategy of locking entire data structures, yet recognizing the efficiency of spin-locks in scenarios of infrequent or brief locking periods, we opted for a different approach. We organized the data structure collections based on their update frequencies, implementing a distribution across the timeline. Collections with longer update periods were divided across multiple execution cycles, a process selectively applied to certain threads. Threads requiring only a few data structure groups bypassed this distribution, completing their cycles swiftly. This distribution methodology facilitated a more deterministic behaviour in our system. This work can be improved with more refactoring aiming to reduce the number of spins for separate threads and thread cycles.

References

[1] M. Misbhauddin, M. Alshayeb, “An integrated metamodel-based approach to software model refactoring” in “Software & Systems Modeling 18”, 2013-2050, 2019, DOI:10.1007/s10270-017-0628-3

[2] B. Weber, M. Reichert, J. Mendling, H.A. Reijers, “Refactoring large process model repositories”, Computers in industry, 62(5), 467-486, 2011, DOI:10.1016/j.compind.2010.12.012

[3] B.K. Sidhu, K. Singh, N. Sharma, “A machine learning approach to software model refactoring”, International Journal of Computers and Applications, 44.2: 166-177, 2022, DOI:10.1080/1206212X.2020.1711616

[4] T. Mens, G. Taentzer, D. Müller, “Challenges in model refactoring”, in “Proc. 1st Workshop on Refactoring Tools”, University of Berlin, Volume 98, 1-5, 2007.

[5] M. Alfranseder, M. Deubzer, B. Justus, J. Mottok, C. Siemers, “An efficient spin-lock based multi-core resource sharing protocol”, in “2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC)”, Austin, TX, USA, 1-7, 2014, DOI:10.1109/PCCC.2014.7017090

[6] D. Zhang, B. Lynch, D. Dechev, “Fast and Scalable Queue-Based Resource Allocation Lock on Shared-Memory Multiprocessors”, in: “Baldoni, R., Nisse, N., van Steen, M. (eds) Principles of Distributed Systems. OPODIS 2013”, Lecture Notes in Computer Science, vol 8304. Springer, Cham., 2013, DOI:10.1007/978-3-319-03850-6_19

[7] S. Afshar, M. Behnam, R.J. Bril, T. Nolte, “Flexible spin-lock model for resource sharing in multiprocessor real-time systems”, in “Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)”, Pisa, Italy, 41-51, 2014, DOI:10.1109/SIES.2014.6871185

[8] S. Afshar, F. Nemati, T. Nolte, “Resource Sharing under Multiprocessor Semi-partitioned Scheduling”, in “2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications”, Seoul, Korea (South), 290-299, 2012, DOI:10.1109/RTCSA.2012.25

[9] S. Afshar, M. Behnam, R.J. Bril, T. Nolte, “Per processor spin-based protocols for multiprocessor real-time systems”, Leibniz Transactions on Embedded Systems, 4(2), Article 03, 2017, DOI:10.4230/LITES-v004-i002-a003