Course Project Report for CS3211: A Verification-Oriented Concurrent Web Crawler

Kheng Yau Dong, Kok Jian Yu, Chan Kin Hung Dickson,
Jake Liam Briscoe, Lague Benjamin Paul
*Equal Contribution

Abstract

In this report, we will go into details of our implementation of the Java Web Crawler as well as our use of PAT and CSP# to test its correctness. We will run you through the design of our crawlers threads, builder threads, buffers, URL storage tree, statistics and result reporters, and the support functions to connect them all. We maximized our Web Crawler's efficiency by parallelizing our pings to web servers, separating crawling tasks and disk writing tasks into different threads, and using multiple disk writers concurrently. We discerned that increasing the number of URL/HTML temporary storage buffers resulted in our largest increases in scraper efficiency. Through experimentation, we determined that six buffers provided the maximum boost to HTML pages scraped, while still maintaining our test server's stability. Our change from one buffer to six buffers resulted in a 10 times increase in URLs scraped!

1. Introduction

In this project, a Java web crawler was created to scrape URLs present in the Internet using a set of seed URLs. It was designed with the goal of maximising the throughput (number of new URLs scraped) by utilising multithreading to investigate the effect of concurrency on the efficiency of a program.

To verify the correctness of the program, a CSP# model was created. Important properties such as liveliness and safety properties of the model were then verified using Process Analysis Toolkit (PAT), an enhanced simulator, model checker and refinement checker for concurrent and real-time systems.

The CSP# model is also designed in a way such that a data race bug could be replayed using PAT when synchronisation is not used. The bug is then replayed in the real Java program to better understand the effect of data race.

2. Approach

The project is divided into 3 parts to better manage the workflow. The different parts of the project are as follow:

Part 1: Implementation of Java concurrent web crawler

Part 2: Modelling of URL crawler using CSP#

Part 3: Data race replay

2.1 Implementation of Java Concurrent Web Crawler
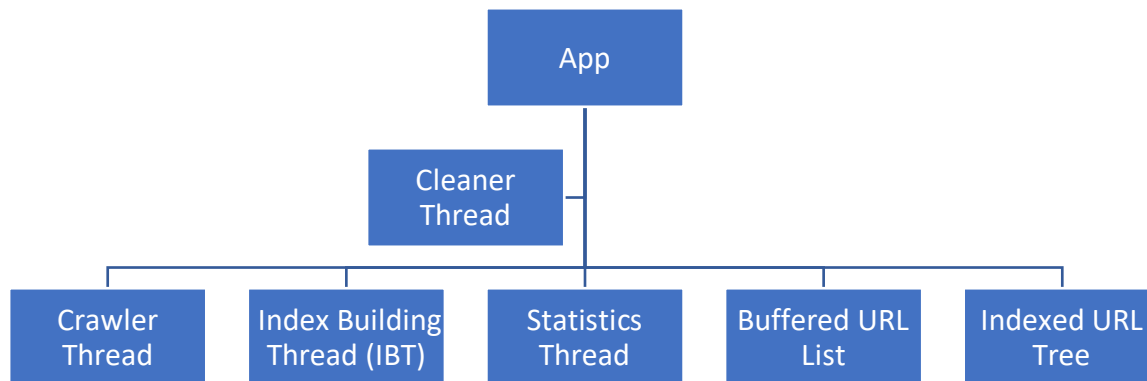
2.1.1 Overall Architecture

Figure 1: Overall Architecture of Web Crawler

2.1.2 App Class

The driving class of the application is App Class. It is responsible for the initialisation of the program and spawning off worker threads, namely Crawler Thread and Index Building Thread. The main thread spawns the App thread (the thread which runs the App object) and sets a time limit for the App Thread to run. Once the time limit is up, the App thread will be interrupted and the termination of the program will begin by joining all the worker threads.

2.1.3 Crawler Thread

The Crawler Threads each have a queue of URLs to be scrapped, which is initially each thread's portion of the seed URLs. At every iteration, the Crawler Thread will remove a URL from the queue (called "searchURL") and make an HTTP GET request to it to fetch

the HTML page back to scrape the new URLs in the page. Before adding the new URLs to its queue, the Crawler Thread will check if the URL is already contained in the IUT or its queue and will add the URL into its queue if none of the conditions mentioned is true. After that, the Crawler Thread will add the searchURL to its buffer which is shared with another Crawler thread. The Crawler Thread then proceeds to the next iteration until there is no more URL left in its queue to be crawled.

A limit on the buffer size is set such that when the buffer size reaches the limit, the Crawler Threads which share the same buffer will be blocked so that the Index Builder Thread can transfer the URLs from it to the Index URL Tree. The buffer size is limited using Java semaphores. Below is an illustration on the synchronisation used to limit the buffer size.

Crawler	**Builder**
`crawlerSemaphore.acquire()`	`builderSemaphore.acquire(BUFFER_SIZE)`
`// add url to buffer …`	`// transfer url from buffer to IUT`
`builderSemaphore.release()`	`crawlerSemaphore.release(BUFFER_SIZE)`

Figure 2: Synchronisation of access to the buffer between Crawler Thread and Index Builder Thread

The access to the buffer by Crawler Thread and Index Builder Thread is set as a critical section protected by two semaphores, namely crawlerSemaphore and builderSemaphore. Let N be the size of the buffer. The crawlerSemaphore is initialised with **N permits** whereas the builderSemaphore is initialised with **0 permit**.

When a Crawler Thread is trying to add a URL into the buffer, it will first try to acquire a permit from the crawlerSemaphore by calling **acquire()** on the **crawlerSemaphore**. Upon successful attempt, the Crawler Thread will add the URL into the buffer and leave the critical section by releasing a permit of the builderSemaphore through calling **release()** on the **builderSemaphore**.

If N URLs were added into the buffer, the number of permits in the crawlerSemaphore would have decreased by N. Thus, the number of permits in the crawlerSemaphore will be 0 and the Crawler Threads will be blocked when they try to acquire a permit from the **crawlerSemaphore**.

On the other hand, the number of permits in the builderSemaphore would have increased by N. Since the builderSemaphore now have N permits, the Index Builder Thread which tried to acquire N permits from the **builderSemaphore** (by calling **acquire(N)** on the builderSemaphore) will be woken up and acquire N permits from the semaphore. The Index Builder Thread will then transfer all the URLs from the buffer to the Index URL Tree.

Once the Index Builder Thread finishes transferring all the URLs, the Index Builder Thread will release N permits of the **crawlerSemaphore** by calling **release(N)** to signal the Crawler Threads that the buffer is now emptied and they can continue to add N URLs into the buffer. The Index Builder Thread will then proceed to the next iteration and try to acquire N permits from the builderSemaphore again.

2.1.4 Index Builder Thread

Each Index Builder Thread(IBT) is attached to a single buffer (explained in 2.1.5), and the job of the IBT is simply to move the items from the buffer, into the Indexed URL Tree(IUT). As mentioned in 2.1.3, IBT is synchronized with the Crawler Thread through the use of semaphores.

When the buffer is full, IBT will be woken up by the semaphore and start running a loop to push every item in the buffer into the IUT. When the buffer is empty, it will then release the crawler semaphore, allowing the Crawler thread to continue populating the buffer.

2.1.5 Buffered URL List

Each buffer is a thread-safe synchronised list of Data objects, which contains URLs (and their source URLs) that have been added by a Crawler Thread and are to be added to the tree. Each buffer is shared by 2 Crawler Threads adding URLs to it and a single Index Builder Thread taking URLs out adding them to the tree. Each buffer can hold a maximum of 1000 URLs at a time.

As mentioned in 2.1.3 access to each buffer is controlled using two semaphore objects. To summarise, the crawler threads initially have N permits while the index builder thread has 0. Every time a crawler thread adds a URL to the buffer a permit is passed from the crawlers to the builder, once the crawlers have passed all N permits to the builder it takes control (as the crawlers have 0 permits and can no longer access this section) and moves all URLs from the buffer to the tree, then passes all N permits back to the crawlers.

2.1.6 Indexed URL Tree

We have implemented Indexed URL Tree (IUT) as a directory tree, where we split up the protocol, domain and directory of an URL into sections and create nested folders which can be traversed through to reach the HTML file.

To identify if the folder name represents a protocol, domain or directory, we also added a prefix to each folder name to identify it. Folders representing a protocol have the prefix 'pr-', domain has the prefix 'do-' and directory have the prefix 'dr-'

The following is an example of how a URL is split up into sections.

- URL: https://en.wikipedia.org/wiki/Concurrency_(computer_science)
    - Protocol → https
    - domain[0] → en
    - domain[1] → wikipedia
    - domain[2] → org
    - directory[0] → wiki
    - directory[1] → Concurrency_(computer_science)
- Stored in folder '/pr-https/do-en/do-wikipedia/do-org/dr-wiki/dr-Concurrency_(computer_science)'
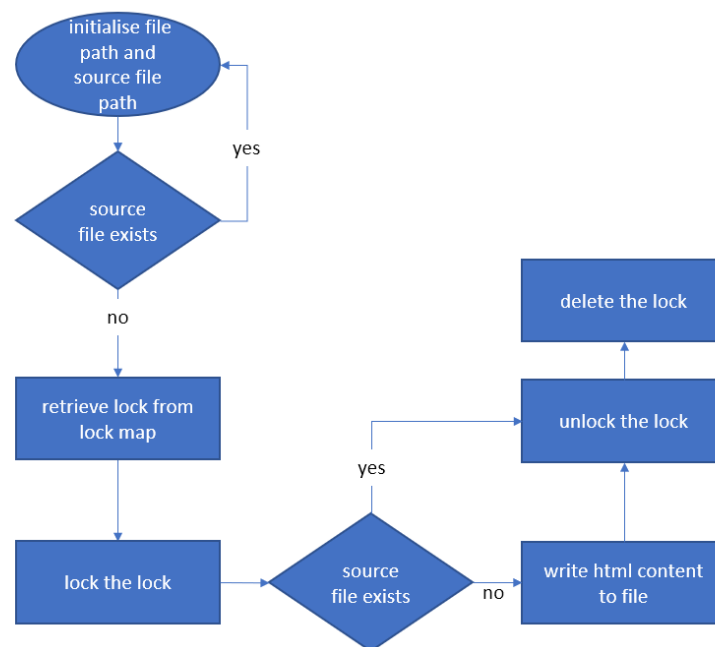
Figure 3: Process Diagram of Adding URL and HTML Content to File

The possibility of data race comes when we write data into the Index URL Tree.

To ensure no data race, we utilize a lock whenever we add something into the IUT. However, rather than locking the IUT object whenever we add something, we decided to only lock the file that we are writing to. This is because it is possible for multiple IBT threads to write to different files at the same time. As such, locking the IUT object can possibly result in a meaningless drop in performance.

To lock by file, we used ReentrantReadWriteLock that is provided by java.

Before writing a URL into the IUT, we first check if a lock for this particular URL has been created by other threads. This was done with a ConcurrentHashMap, where the key is the URL path, and value is the lock.

If a lock does not exist, the IUT creates the lock, put it in the ConcurrentHashMap.

If a lock exists, it will retrieve the lock.

Before writing, into the IUT, the thread will lock itself using the lock that was either created or retrieved. This ensures that the critical section of the code is mutually exclusive when writing to the same URL path.

After writing, the thread will unlock itself using the lock, and delete the lock from the ConcurrentHashMap if unused.

2.1.7 Statistics Thread

The statistics thread wakes up once an hour and calculates how many URLs have been added to the tree in the past hour. It also finds how many URLs were added versus how many were removed overall for all of the queues and for all of the buffers.

2.1.8 Cleaner Thread

When the main thread exits and Java Virtual Machine (JVM) begins its shutdown, the Cleaner thread which is attached as a shutdown hook to the program during the initialisation of the program will begin to clean up the application. The cleaning process is as follow:

1. Write all remaining URLs in the buffers to the Indexed URLs Tree.
2. Write all URLs in Indexed URLs Tree into a text file.
3. Write all remaining new URLs that has not been crawled in the Crawler Threads' queue into a text file named "**res2.txt**"
4. Write statistics of the result of the application to a text file named "statistics.txt"

The application terminates completely once the Cleaner Thread finishes the cleaning.

2.2 Concurrency Design Modelling

A CSP# model was created to verify the correctness of the web crawler's concurrency design. The model mainly focuses on the interaction between the Crawler Thread and Index Builder Thread, specifically the synchronisation, because it is the primary part of the program.

2.2.1 Absence of Deadlock

To model the crawling of URLs, an array of integers is created to model the finite set of all URLs on the Internet. Each index of the array represents an URL and each integer in the array represents the next index of the array to be accessed (crawled). If the integer is -1, it means that the current URL does not lead to any new URL. Such a URL is called "terminating URL". The array will be named as "Pool" in this section.

The main process in the CSP# model is called System(). It represents the interactions (modelled as CrawlersAndBuilders()) between Crawler Threads and Index Builder Threads on 3 buffers. Each buffer is shared by 2 crawlers and 1 index URL builder. This interaction is modelled by the process CrawlersBuilder(i).

The system begins with the initialisation of the Crawler Threads. Each Crawler Thread (modelled as Crawler(i)) will remove a URL (integer) from their own queue. The URL retrieved is named as "searchURL". Using the searchURL, the Crawler Thread will then access a particular index of the Pool array corresponding to the integer value of the search URL. For example, if the search URL is 3, then the Crawler Thread will access index 3 of the Pool array. The Crawler Thread then stores the integer value (new URL) at the index into another array which serves as temporary storage, where the new URL is stored at the index corresponding to that of Crawler Thread. For instance, if the Crawler Thread's index is 1, then the Crawler Thread will store the new URL at index 1 in the array which serves as temporary storage.

After that, the Crawler Thread proceeds to check if the new URL retrieved is a terminating URL or present in the Index URL Tree or the queue. If none of the conditions aforementioned is true, the Crawler Thread will store the new URL into its own queue to be crawled at the next iteration. Else, it will proceed to the next step.

The next step is storing the search URL into the buffer to be transferred to the Index URL Tree when the buffer is "full". Before the Crawler Thread accesses the buffer, it will first try to grab a semaphore called Crawler Semaphore and after that a mutex called Buffer Mutex. In the CSP# model, the grab and release of the semaphore are modelled as an atomic decrement and increment of an integer variable respectively.

The Crawler Semaphore synchronises the access of the buffer between the Crawler Thread and the Index Builder Thread such that the Crawler Thread is blocked when the buffer size limit is reached. The Index URL Builder Thread (modelled as Builder(i)), which previously waits at a semaphore called Builder Semaphore, will then wake up and transfer all the URLs in the buffer into the Index URL Tree and release the Crawler Semaphore once the transfer is finished to wake up the blocked Crawler Threads. As mentioned in Section 2.1, the Index URL Builder will try to grab a total number of permits X of the Builder Semaphore which is equal to the specified buffer size (e.g. 1000) to set a limit on the buffer size. Once the Index Builder Thread finishes its current iteration, it will release the same total number of permits X of the Crawler Semaphore to signal that the buffer is

now empty. The process of transferring the URL will be explained in more details in the next section.

Moving on, the Buffer Mutexes are used to synchronise the access of buffers between Crawler Threads that share the same buffer such that at any time there can be only one Crawler Thread accessing the buffer.

Once the Crawler Thread manages to grab the two semaphores mentioned, it will check if the Search URL is present in the buffer or if it is a seed URL. If neither of the conditions mentioned is true, the Crawler Thread will store the search URL into the buffer. Else, the Crawler Thread will proceed to the next step.

After the Search URL is stored, the Crawler Thread will release the Buffer Mutex followed by the Builder Semaphore and proceed to the next iteration. Note that unlike Index Builder Thread, every grab and release on semaphore done by Crawler Thread will decrease and increase the number of permits in the semaphore by 1 respectively.

The Crawler Thread will continue to run until there is no more URL in its queue. Once this happens, the Crawler Thread will then proceed to the next process which is CrawlerJoin(i). It models the joining of a Crawler Thread with the application thread when the Crawler Thread has no more URL to be crawled.

When all Crawler Threads have joined, the Index Builder Threads will be interrupted after they have finished their current ongoing iteration (if any) of transferring the URLs from buffer to the Index URL Tree. The Index Builder Thread will then proceed to the next process BuilderJoin(i) which models the joining of a Builder Thread with the application thread.

Upon joining of all Index URL Builders, the application will terminate and the Cleaner Thread (modelled as Cleaner()) will begin cleaning up the resources. Here, the Cleaner Thread will transfer the remaining URL in the buffers if any into the Index URL Tree. Once the Cleaner Thread has finished the cleaning, the entire model will terminate.

To verify if there is any deadlock in the web crawler program, the CSP# model is tested against the assertion **deadlockfree** provided by PAT.

2.2.2 Absence of Data Race

To show the absence of data race, we have to ensure that there is no duplicate file creation, no duplicate data write and also ensure correctness at the end of the process.

To do this, we modelled our IUT implementation into PAT to verify if there is any data race.

We created an IndexURLTree class in C# to model our Index URL Tree. It contains a fileList to represent file creation, and a dataList to represent data writing into the file.

If a URL exists in fileList, it means it has been created. If a URL exists in dataList, it means that its HTML content has been written into the file. There are also 4 different methods built into the class, FileExists, FileListDuplicateExists, DataListDuplicateExists and CheckCorrectness.

FileExists allow us to check if a URL already exists in the IUT.

FileListDuplicateExists and DataListDuplicateExists do as the name states. They check if duplicates exist in the fileList and dataList. If a duplicate exists, it means that data race has occurred.

CheckCorrectness takes in an array that is what the fileList and dataList should look like after adding every URL, and compares it to the actual fileList and dataList to see if it is the same.

Using the model we created, we then modelled the IUT addUrlAndContent implementation into PAT.
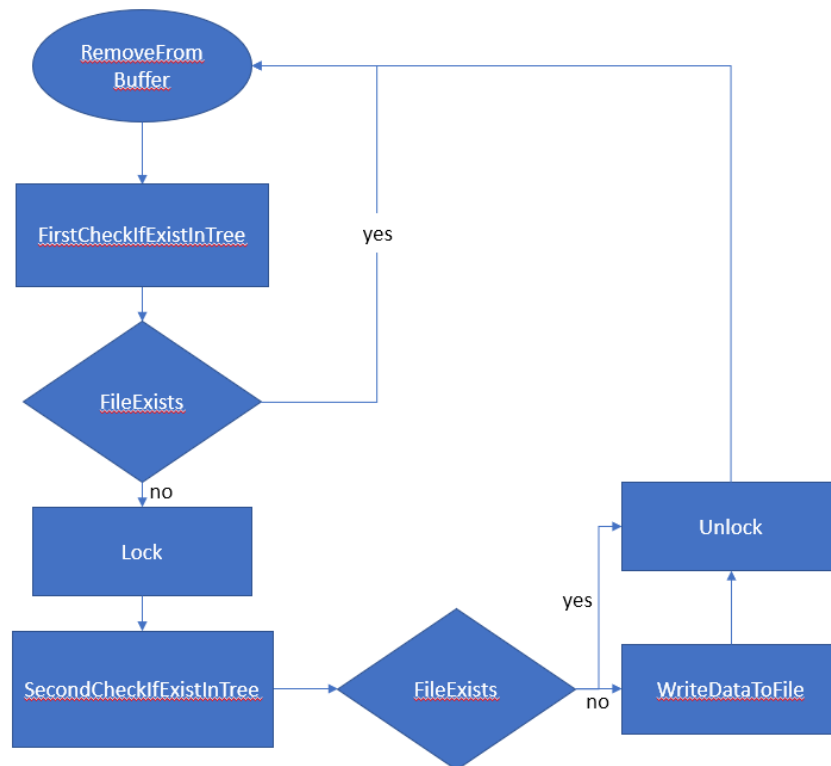
Figure 4: Process Diagram of PAT Model for IUT addUrlAndContent Implementation

We first retrieved a URL from the buffer as described in section 2.2.1 and checked if it exists in the IUT. If yes, we get another URL from the buffer. If not, we acquire a lock for the file. The lock here is represented with a channel of size 1.

After acquiring the lock, we proceed on to check if the file exists in the tree again. If yes, we write data to file, this includes both creation of a file, and writing of data into the file, then unlock the lock. If not, we immediately unlock the lock without writing data to file.

Then we return back to the very first step and retrieve another URL from the buffer. This will continue until the buffer is empty.

We also create 3 different conditions. noDuplicateFileCreation, noDuplicateDataWrite and correct. noDuplicateFileCreation ensures that there is no duplicate file creation, noDuplicateDataWrite ensures that there are no duplicate data writing and correct ensures that the data in the IUT is correct.

To verify if there is any data race, we assert that the System always has noDuplicateFileCreation and noDuplicateDataWrite. We also assert that the System will eventually be correct.

2.2.3 Occurrence of Data Race When Synchronisation is Not Used

To show the occurrence of data race when synchronization is not used, we created a boolean value, SHOW_DATARACE. If set to false, the System will use a lock that is represented with a channel as explained in 2.2.2. This will give us a result of having no data race if you run the 3 assertions to show data race.

If set to true, the System will no longer use the lock. This will give us a result of having data race if you run the 3 assertions to show data race.

2.3 Replay of Concurrency Bug

For the replay of concurrency bug (Part 3 of the project), we have written up 2 different approaches to do this.
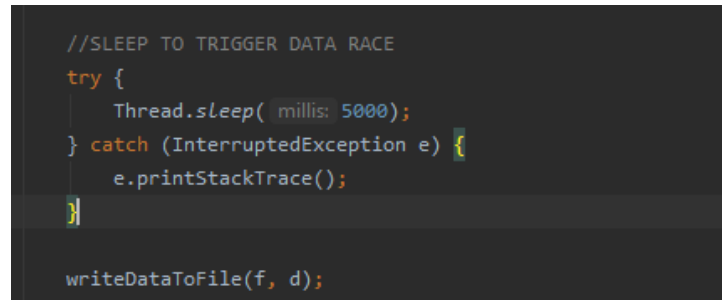
One way is through Thread.sleep() method, the other using a lock together with wait and notifyAll methods.

To set up the concurrency bug, we first scaled down the problem to use 4 crawlers, 2 buffers and 2 Index Builder Thread(IBT). We also changed the seed file to only contain 4 identical URL. This ensures that all of the 4 crawlers' queues will be identical, as such they will all be passing in the exact same output into their respective buffers. We also reduced buffer size to 1 so that all outputs will be immediately added into the Index URL Tree(IUT).

In addition to this, we also disabled the locking mechanism of our critical section in the IUT that was explained in section 2.1.6 to allow data race to occur in our code.

For the Thread.sleep() approach, we added Thread.sleep(5000) right before writing the data to file. This ensures that both IBTs will be in the critical sections of the code at the same time. After the Thread.sleep expires, they will both run the write data to file code at the same time. Since both buffers must have the same initial input due to our seed file, this results in a data race.

The code for this is provided below:

```
//SLEEP TO TRIGGER DATA RACE
try {
    Thread.sleep( millis: 5000);
} catch (InterruptedException e) {
    e.printStackTrace();
}

writeDataToFile(f, d);
```

Figure 5: Code snippet of replaying data race using Thread.sleep()

For the approach using a lock, wait and notifyAll, we added some logic to make it such that both threads will have to be adding the exact same URL in order for the code to continue and write the data to file. If not, the thread will be blocked.

In our code, we will first acquire a lock for the specific URL we are writing into. Afterwards, we will try to lock the lock. If successful, the thread will now perform the wait() method with the lock as the object. This blocks the current thread from proceeding. If this thread is ever unblocked through a notifyAll() or notify() method, it will proceed to unlock the lock.

If the thread fails to acquire the lock, it will then perform the notifyAll() method with the lock as the object. This unblocks any other threads that were being blocked.

Therefore, it can be seen that for the thread to be able to write the data to file, there must be 2 threads writing to the exact same URL, where the first thread will successfully acquire the lock, and be blocked with the wait method, and the second thread will fail to acquire the lock since it was acquired by the first thread, and unblock the first thread with the notifyAll() method.

The code of this section is provided below.

```
// 2 Threads writing to same file must be at this point at the same time to pass.
if(lock.writeLock().tryLock()) {
    try {
        // Block yourself if u can get lock
        synchronized (lock) {
            // Lock is reentrant lock, therefore wait used here instead.
            lock.wait();
            lock.writeLock().unlock();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
} else {
    // Unblock other thread if you are unable to get lock
    synchronized (lock) {
        lock.notifyAll();
    }
}


writeDataToFile(f, d);
```

Figure 6: Code snippet of replaying data race using a lock

It is to be noted that it might be possible for this code to reach a deadlock if both buffers provide different URLs. However, the possibility of this occurring is relatively small, only happening if there was some error that results in the crawler thread having to skip a URL.

There are 3 ways for us to verify if a data race has occurred. The first way will be to check the content.html file for the URL. If a data race occurs, the content.html file will be written into multiple times, as such duplicates or errors in data should occur.

The second way will be to check the source.txt file. This file contains information of the URL being stored in the current directory and its source URL. If data race occurs, this file will contain multiple lines as each URL should only be added once into the IUT.

The third way will be to check the res.txt file. This file is created by combining all source.txt file together. As such, if a data race occurs, you will be able to see there being multiple entries that lead to the same URL.

3. Implementation

The web crawler application is built using Gradle. Gradle is used to manage the dependencies of the application and package the application together with its dependencies into a runnable .jar file using a Gradle plugin known as ShadowJar. The dependency that is used in this application is called HTMLUnit which is a headless browser. HTMLUnit is used in Crawler class to facilitate the process of making HTTP GET requests to the URLs given to retrieve the web page and scrape the URLs in it.

The web crawler application is initiated using the command below

java -jar cawler.jar -time <hours>**h -input** <input file name> **-output** <output file name>

e.g. java -jar cawler.jar -time 24h -input seed.txt -output res.txt

- <hours> is the time limit (in hours) for the application to run
- <input file name> is the name of the seed URLs file. The seed URLs file contains the initial URLs given to the web crawler program, where each URL is separated on a new line.
- <output file name> is the name of the result file. The result file contains all the new URLs that were scraped by the program.

After the termination of the program, a few files will be created, namely:

- res.txt: contains all the new URLs that have been found and crawled.
  - The name of the file follows <output file name> as described above
- res2.txt: contains all the new URLs that have been found but not crawled by crawler yet.
- statistics.txt: contains statistics of the result of the program run.
- houry_statistics.txt: contains statistics of the program run recorded at every hour.

Thus, the **total number of new URLs found** (crawled or not crawled) are equal to the total number of URLs in res.txt **and** res2.txt, which is equivalent to the total number of lines in res.txt and res2.txt. Note that for **res2.txt**, the total number of lines should be subtracted by 1 to exclude the title of res2.txt. The summary of the calculation of the total number of new URLs found is as follow:

Total number of new URLs (crawled or not crawled)

= Total number of lines in res.txt + Total number of lines in res2.txt - 1

Another runnable jar called *merger.jar* is also provided. It serves as a back-up to generate the result file from the Indexed URL Tree (as a directory structure) in the case of an unexpected crash of the program during its execution.

The methods to create *cawler.jar* and *merger.jar* are described in README.md attached in the source code directory. Specifically, a shell file called *create_jars.sh* was written to facilitate the creation of the jar files using Gradle.

With regards to the CSP# model of the program, two C# library files were created to abstract away some complex logic of the modelling and provide various methods to facilitate the modelling of the program. The two C# library files are **PAT.Lib.Lists.cs** and **PAT.Lib.IndexURLTree.cs**. **PAT.Lib.Lists.cs** model a two-dimensional list and is used to simulate the lists of buffers and lists of queues for Crawler Threads. On the other hand**,**

PAT.Lib.IndexURLTree.cs models the Index URL Tree of the program. Both C# library files should be placed in the **Lib** folder of PAT.

4. Evaluation

4.1 Efficiency

The runtime efficiency improvement of the web crawler is expected to be proportional to the number of Crawler Threads. For example, the program is expected to perform 2 times faster if 2 Crawlers Threads are used. The actual runtime performance is subjected to several factors such as the network speed which may differ in different runs.

To test the efficiency of the web crawler, the web crawler is run on a Ubuntu server with a 24-core CPU and 64 GB of RAM. Different configurations (e.g. number of Crawler Threads and Index Builder Threads) were tested to find the best configuration which maximises the efficiency of the web crawler on this server.

The main configuration that was changed was the number of buffers, which affected the total number of Crawler Threads and Index Builder Threads. The number of buffers used was 1, 3, and 6. For each different number of buffers, the web crawler is run against 2, 6, and 10 hours and the total number of new URLs written to the IUT was recorded.

Below are the data collected during different test runs.

Number of Buffers	1	*1 crawler only
Date	Duration (hours)	Number of New URLs Written to IUT
24.4.20	2	925
24.4.20	6	9714
25.4.20	10	16933

Number of Buffers	3	*6 crawlers
Date	Duration (hours)	Number of New URLs Written to IUT
20.3.20	2	26927
23.3.20	6	68956
6.4.20	10	101900

Number of Buffers	6	*12 crawlers

Date	Duration (hours)	Number of New URLs Written to IUT
23.4.20	2	51238
20.4.20	6	100847
23.4.20	10	171341

Figure 7: Number of new URLs scraped across different number of buffers
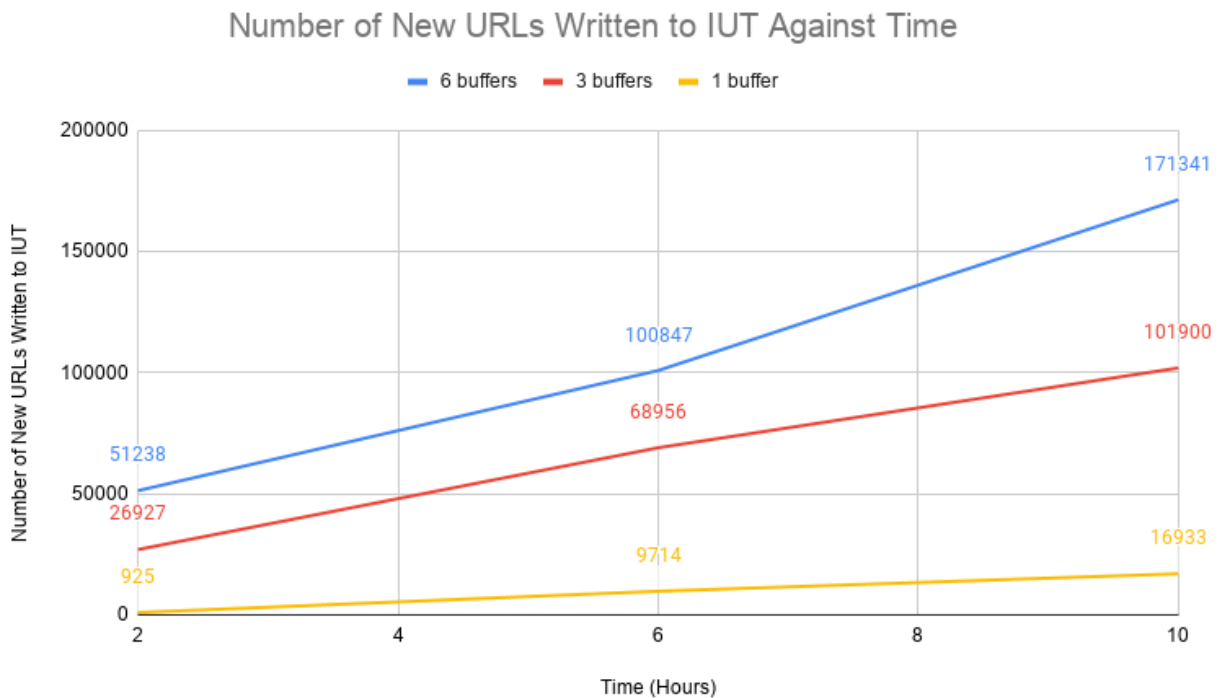
Figure 8: Graph of the number of new URLs written to IUT against time

According to the graph above, the throughput of 3 buffers was around 6 times greater than that of 1 buffer (based on the number of URLs collected after 10 hours). This meets the theoretical expectation of the increase in efficiency where the number of URLs crawled is proportional to the number of crawlers. In this case, there are a total of 6 crawlers used for 3 buffers whereas only 1 crawler is used for 1 buffer.

On the other hand, the throughput of 6 buffers was around 10 times greater than that of 1 buffer. This is lower than the theoretical expectation of the increase in efficiency, which is greater by 12 times. A plausible reason behind the lower increase may be the heavy overhead on the system when 6 buffers are used since the total number of threads in this

configuration is 21 threads (12 Crawler Threads + 6 Index Builder Threads + 1 Statistics Thread + 1 App Thread + 1 Main Thread).

Therefore, based on the data collected, 6 buffers will be used in the final web crawler program to be submitted so that the number of URLs crawled can be maximised while maintaining a supportable overhead on the system.

4.2 Correctness

The testing of the CSP# model against multiple assertions using PAT showed that the web crawler program satisfies the liveliness property as well as the desired safety properties, namely deadlock-free and no duplicate URLs in the Index URL Tree (i.e. no data race). The snapshots of the verification are attached in Appendix A. Also, the testing of CSP# model against the assertions when synchronisation was not used showed the existence of data race which can be found in Appendix B.

5. Discussion

The concurrency architecture of the program consists of several parts which work together to maximise the number of new URLs crawled by the program. One example would be the separation of the application workload into Crawler Thread and Index Builder Thread as well as the usage of a buffer to store the crawled URLs. It enables the Crawler Threads to continuously crawl for new URLs until the buffer is full and run again when the buffer is emptied. If crawling and storing of URL into the disk is combined into a single synchronous task, then a great portion of the runtime will be wasted as storing the URL into the disk is an IO process which can take time to complete. Hence, the separated approach enables the program to continuously crawl for more new URLs compared to the synchronous approach. As a result, the program can utilise a greater proportion of its given runtime to crawl for more new URLs. The disadvantage of the concurrent approach is it increases the complexity and overhead of the program as synchronisations need to be done between Crawler Threads and Index Builder Threads. However, the overhead is outweighed by the increase in efficiency when the program is run for a long time.
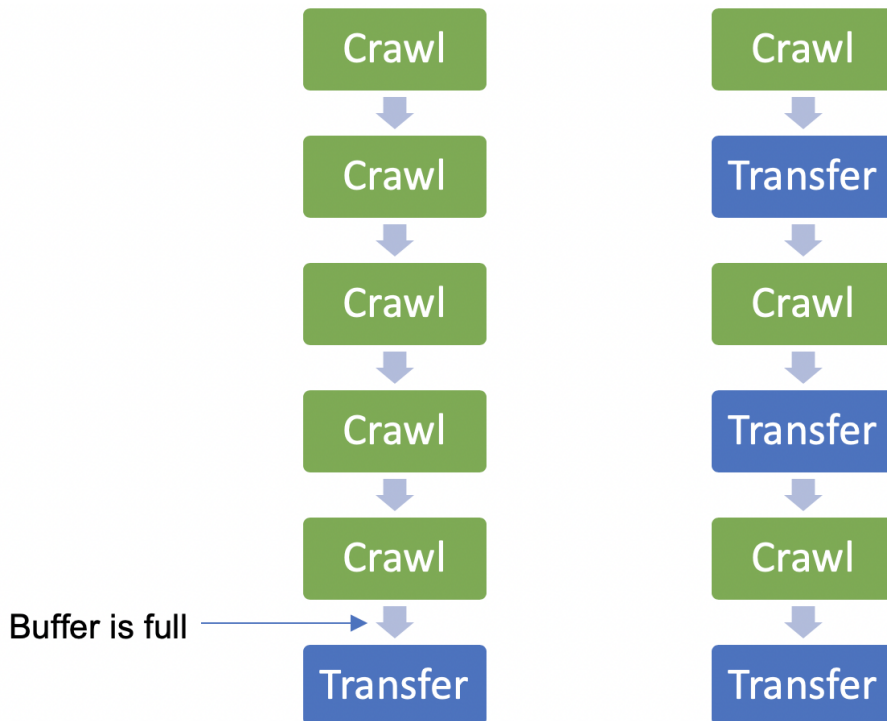
Figure 9: Comparison between crawling efficiency of concurrent and sequential approach (Left: concurrent approach; Right: sequential approach)

Moving on, multiple Crawler Threads are created to help the program to overcome the network speed bottleneck by firing multiple HTTP requests at the same time to different servers. Thus, more HTML pages can be retrieved in a given period of time. Multiple buffers are used because it only blocks the Crawler Threads which share the same buffer when the buffer is full. If only one buffer is used and shared between all Crawler Threads, all Crawler Threads will be blocked when the buffer is full. Therefore, a significant runtime which can be used to crawl for new URLs will be wasted. Also, configuring a fixed size for the buffer also helps to minimise the overhead of synchronisation of access to the buffer. It reduces the number of times synchronisation needs to be done between Crawler Threads and Index Builder Threads compared to the approach of signalling the Index Builder Threads to wake up once there is a crawled URL to be transferred.

However, the downside of the multiple buffers is duplicated URLs can be present in different buffers at the same time because the Crawler Thread only checks for duplicates in the Index URL Tree before adding the crawled URL to the buffer. This is resolved by checking if the URL is already present before the Index Builder Thread writes the crawled URL from the buffer into the Index URL Tree.

6. Conclusion

From utilizing multiple crawlers to access multiple websites at the same time, to increasing the number of buffers to reduce reading and writing wait times, we applied parallelization to many key aspects of our program. Building our Java Web Crawler gave us hands-on experience with threads and parallelization methods and revealed their significance in increasing program efficiency. Designing programs with parallelization in mind allow them to maximize their utilization of the available computing resources and, in many cases, out-produce asynchronous programs. We are excited to have learned many parallelization techniques and are excited to implement them in our future programs to increase their efficiency and provide them with a significant edge over competitors.

References:

[1] PAT: Process Analysis Toolkit. (n.d.). Retrieved April 29, 2020, from https://pat.comp.nus.edu.sg

[2] Java® Platform, Standard Edition & Java Development Kit Version 11 API Specification. (n.d.). Retrieved April 29, 2020, from https://docs.oracle.com/en/java/javase/11/docs/api/index.html

Appendix A: Snapshots of Verification of Program Correctness using PAT

A.1 Data Race Free

PAT - An Enhanced Simulation and Model Checking Tool (Version 3.5.1 running on Unix)

File Edit View Tools Examples Window Help

ProcessModel_original...

```
1   #import "PAT.Lib.IndexU
2   #import "PAT.Lib.Lists"
3
4   #define BUFFER_SIZE 2;
5   #define NUM_BUFFERS 2;
6   #define NUM_CRAWLERS 4;
7   #define NUM_SEEDS 4;
8   #define NUM_URLS 6;
9   #define SHOW_DATARACE f
10
11  // Model the pool of ur
12  // Each integer in the
13  var urls = [5, 4, 4, 5,
14  var result = [5, 4];
15
16  // queues: queue of eac
17  var tree = new IndexURL
18  var<Lists> queues = new
19  var<Lists> buffers = ne
20
21  // Initialise each queu
22  // a, b, c ... are just
23  var a = queues.Append(0
24  var b = queues.Append(1
25  var c = queues.Append(2
26  var d = queues.Append(3
27
28  // crawlerSem & builder
29  // bufferMutex : synchr
30  var crawlerSem = [BUFFE
31  var builderSem = [0(NUM
32  var bufferMutex = [1(NU
33  var builderFinishIterat
```

Verification - ProcessModel_original.csp

Assertions

1	System() deadlockfree
2	System() reaches allCrawlersJoined
3	System() reaches allBuildersJoined
4	System() reaches containsDuplicates
5	System() reaches bufferSize
6	System() |= [] noDuplicateFileCreation
7	System() |= [] noDuplicateDataWrite
8	System() |= <> correct

Selected Assertion

System() |= <> correct

[Verify] [View Büchi Automata] [Simulate Witness Trace]

Options

Admissible Behavior: All   Timed out after (minutes): [ ]
Verification Engine: Strongly Connected Component Based Search   Generate Witness Trace ☑

Output

********Verification Result********
The Assertion (System() |= <> correct) is VALID.

********Verification Setting********
Admissible Behavior: All
Search Engine: Loop Existence Checking - The negation of the LTL formula is a safety property!
System Abstraction: False

********Verification Statistics********
Visited States:6146183
Total Transitions:25855700
Time Used:1307.3238256s
Estimated Memory Used:227780.448KB

********Verification Result********
The Assertion (System() deadlockfree) is VALID.

********Verification Setting********
Admissible Behavior: All
Search Engine: First Witness Trace using Depth First Search
System Abstraction: False

Verification Completed

Output Window

Specification is parsed in 0.2761966s
//=============Global Constants======
#define BUFFER_SIZE 2;
#define NUM_BUFFERS 2;
#define NUM_CRAWLERS 4;
#define NUM_SEEDS 4;
#define NUM_URLS 6;
#define SHOW_DATARACE F;
//=============Global Variable Definitions=============
var urls = [5, 4, 4, 5, -1, -1];
var isNewUrlInQueue = [false, false, false, false];

Model Saved ProcessModel original.csp Ln: 14 Col: 20 INS

PAT - An Enhanced Simulation and Model Checking Tool (Version 3.5.1 running on Unix)

File Edit View Tools Examples Window Help

ProcessModel_original... test

```
1   #import "PAT.Lib.IndexU
2   #import "PAT.Lib.Lists"
3
4   #define BUFFER_SIZE 2;
5   #define NUM_BUFFERS 2;
6   #define NUM_CRAWLERS 4;
7   #define NUM_SEEDS 4;
8   #define NUM_URLS 6;
9   #define SHOW_DATARACE f
10
11  // Model the pool of ur
12  // Each integer in the
13  var urls = [5, 4, 4, 5,
14  var result = [5, 4];
15
16  // queues: queue of eac
17  var tree = new IndexURL
18  var<Lists> queues = new
19  var<Lists> buffers = ne
20
21  // Initialise each queu
22  // a, b, c ... are just
23  var a = queues.Append(0
24  var b = queues.Append(1
25  var c = queues.Append(2
26  var d = queues.Append(3
27
28  // crawlerSem & builder
29  // bufferMutex : synchr
30  var crawlerSem = [BUFFE
31  var builderSem = [0(NUM
32  var bufferMutex = [1(NU
33  var builderFinishIterat
```

Verification - ProcessModel_original.csp

Assertions

1	System() deadlockfree
2	System() reaches allCrawlersJoined
3	System() reaches allBuildersJoined
4	System() reaches containsDuplicates
5	System() reaches bufferSize
6	System() |= [] noDuplicateFileCreation
7	System() |= [] noDuplicateDataWrite
8	System() |= <> correct

Selected Assertion

System() |= [] noDuplicateFileCreation

[Verify] [View Büchi Automata] [Simulate Witness Trace]

Options

Admissible Behavior: All   Timed out after (minutes): [ ]
Verification Engine: First Witness Trace using Depth First Search   Generate Witness Trace ☑

Output

********Verification Result********
The Assertion (System() |= [] noDuplicateFileCreation) is VALID.

********Verification Setting********
Admissible Behavior: All
Method: Refinement Based Safety Analysis using DFS - The LTL formula is a safety property!
System Abstraction: False

********Verification Statistics********
Visited States:8017611
Total Transitions:32939556
Time Used:1580.8832524s
Estimated Memory Used:352596.576KB

Verification Completed

Output Window

Specification is parsed in 0.0348915s
//=============Global Constants======
#define BUFFER_SIZE 2;
#define NUM_BUFFERS 2;
#define NUM_CRAWLERS 4;
#define NUM_SEEDS 4;
#define NUM_URLS 6;
#define SHOW_DATARACE F;
//=============Global Variable Definitions=============

Output Window Error List

Model Saved ProcessModel original.csp Ln: 9 Col: 27 INS

PAT - An Enhanced Simulation and Model Checking Tool (Version 3.5.1 running on Unix)

File   Edit   View   Tools   Examples   Window   Help

test...   ProcessModel_original...

```
1   #import "PAT.Lib.IndexU
2   #import "PAT.Lib.Lists"
3
4   #define BUFFER_SIZE 2;
5   #define NUM_BUFFERS 2;
6   #define NUM_CRAWLERS 4;
7   #define NUM_SEEDS 4;
8   #define NUM_URLS 6;
9   #define SHOW_DATARACE f
10
11  // Model the pool of ur
12  // Each integer in the
13  var urls = [5, 4, 4, 5,
14  var result = [5, 4];
15
16  // queues: queue of eac
17  var tree = new IndexURL
18  var<Lists> queues = new
19  var<Lists> buffers = ne
20
21  // Initialise each queu
22  // a, b, c ... are just
23  var a = queues.Append(0
24  var b = queues.Append(1
25  var c = queues.Append(2
26  var d = queues.Append(3
27
28  // crawlerSem & builder
29  // bufferMutex : synchr
30  var crawlerSem = [BUFFE
31  var builderSem = [0(NUM
32  var bufferMutex = [1(NU
33  var builderFinishIterat
```

Verification - ProcessModel_original.csp

Assertions

#	Assertion
1	System() deadlockfree
2	System() reaches allCrawlersJoined
3	System() reaches allBuildersJoined
4	System() reaches containsDuplicates
5	System() reaches bufferSize
6	System() |= [] noDuplicateFileCreation
7	System() |= [] noDuplicateDataWrite
8	System() |= <> correct

Selected Assertion

System() \|= [] noDuplicateDataWrite

[Verify]   [View Büchi Automata]   [Simulate Witness Trace]

Options

Admissible Behavior	All
Verification Engine	First Witness Trace using Depth First Search

Timed out after (minutes) [ ]
Generate Witness Trace ☑

Output

********Verification Result********
The Assertion (System() \|= [] noDuplicateDataWrite) is VALID.

********Verification Setting********
Admissible Behavior: All
Method: Refinement Based Safety Analysis using DFS - The LTL formula is a safety property!
System Abstraction: False

********Verification Statistics********
Visited States:8017611
Total Transitions:32939556
Time Used:1475.2457661s
Estimated Memory Used:262162.864KB

Verification Completed

Output Window

Specification is parsed in 0.2968128s
//=============Global Constants=======
#define BUFFER_SIZE 2;
#define NUM_BUFFERS 2;
#define NUM_CRAWLERS 4;
#define NUM_SEEDS 4;
#define NUM_URLS 6;
#define SHOW_DATARACE F;
//==============Global Variable Definitions====================
var urls = [5, 4, 4, 5, -1, -1];
var isNewUrlInQueue = [false, false, false, false];

Model Saved   ProcessModel_original.csp   Ln: 1   Col: 0   INS

A.2 With Data Race

PAT - An Enhanced Simulation and Model Checking Tool (Version 3.5.1 running on Unix)

File   Edit   View   Tools   Examples   Window   Help

100%

ProcessModel...

Verification - ProcessModel.csp

Assertions

1	System() deadlockfree
2	System() reaches allCrawlersJoined
3	System() reaches allBuildersJoined
4	System() reaches containsDuplicates
5	System() reaches bufferSize
6	System() |= [] noDuplicateFileCreation
7	System() |= [] noDuplicateDataWrite
8	System() |= <> correct

Selected Assertion

System() |= [] noDuplicateFileCreation

Verify   View Büchi Automata   Simulate Witness Trace

Options

Admissible Behavior: All   Timed out after (minutes)

Verification Engine: First Witness Trace using Depth First Search   Generate Witness Trace

Output

```
********Verification Result********
The Assertion (System() |= [] noDuplicateFileCreation) is NOT valid.
A counterexample is presented as follows.
<init -> dequeue.3 -> crawl.3 -> checkTree.3 -> checkQueue.3 -> addToQueue.3 -> grabCrawlerSem.3 -> grabBufferMutex.3 ->
releaseBufferMutex.3 -> releaseBuilderSem.3 -> dequeue.3 -> crawl.3 -> checkTree.3 -> checkQueue.3 -> grabCrawlerSem.3 ->
grabBufferMutex.3 -> addToBuffer.3 -> releaseBufferMutex.3 -> releaseBuilderSem.3 -> grabBuilderSem.1 -> setNotFinishIteration.1 ->
removeFromBuffer.1 -> check.1 -> [if!(!(SHOW_DATARACE))] -> check.1 -> createFile.1 -> writeData.1 -> [if!(!(SHOW_DATARACE))] ->
releaseCrawlerSem.1 -> setFinishIteration.1 -> updateCrawlerJoined.3 -> dequeue.2 -> crawl.2 -> checkTree.2 -> checkQueue.2 ->
addToQueue.2 -> grabCrawlerSem.2 -> grabBufferMutex.2 -> releaseBufferMutex.2 -> releaseBuilderSem.2 -> dequeue.2 -> crawl.2 ->
checkTree.2 -> checkQueue.2 -> grabCrawlerSem.2 -> grabBufferMutex.2 -> addToBuffer.2 -> releaseBufferMutex.2 -> releaseBuilderSem.2
-> grabBuilderSem.1 -> setNotFinishIteration.1 -> removeFromBuffer.1 -> check.1 -> [if!(!(SHOW_DATARACE))] -> check.1 ->
updateCrawlerJoined.2 -> dequeue.1 -> crawl.1 -> checkTree.1 -> checkQueue.1 -> addToQueue.1 -> grabCrawlerSem.1 ->
grabBufferMutex.1 -> releaseBufferMutex.1 -> releaseBuilderSem.1 -> dequeue.1 -> crawl.1 -> checkTree.1 -> checkQueue.1 ->
grabCrawlerSem.1 -> grabBufferMutex.1 -> addToBuffer.1 -> releaseBufferMutex.1 -> releaseBuilderSem.0 -> grabBuilderSem.0 ->
setNotFinishIteration.0 -> removeFromBuffer.0 -> check.0 -> [if!(!(SHOW_DATARACE))] -> check.0 -> createFile.1 -> writeData.1 ->
[if!(!(SHOW_DATARACE))] -> releaseCrawlerSem.1 -> setFinishIteration.1 -> createFile.0

********Verification Setting********
Admissible Behavior: All
Method: Refinement Based Safety Analysis using DFS - The LTL formula is a safety property!
System Abstraction: False
********Verification Statistics********
```

Verification Completed

Output Window

Specification is parsed in 0.0498092s
//=================Global Constants=======
#define BUFFER_SIZE 2;
#define NUM_BUFFERS 2;
#define NUM_CRAWLERS 4;
#define NUM_SEEDS 4;
#define NUM_URLS 6;
#define SHOW_DATARACE T;
//===========Global Variable Definitions===============
var urls = [5, 4, 4, 5, -1, -1];
var isNewUrlInQueue = [false, false, false, false];

Model Saved   ProcessModel.csp   Ln: 9   Col: 26   INS

Verification - ProcessModel.csp

Assertions

1	System() deadlockfree
2	System() reaches allCrawlersJoined
3	System() reaches allBuildersJoined
4	System() reaches containsDuplicates
5	System() reaches bufferSize
6	System() |= [] noDuplicateFileCreation
7	System() |= [] noDuplicateDataWrite
8	System() |= <> correct

Selected Assertion

System() |= [] noDuplicateDataWrite

Verify   View Büchi Automata   Simulate Witness Trace

Options

Admissible Behavior: All   Timed out after (minutes)

Verification Engine: First Witness Trace using Depth First Search   Generate Witness Trace

Output

```
********Verification Result********
The Assertion (System() |= [] noDuplicateDataWrite) is NOT valid.
A counterexample is presented as follows.
<init -> dequeue.3 -> crawl.3 -> checkTree.3 -> checkQueue.3 -> addToQueue.3 -> grabCrawlerSem.3 -> grabBufferMutex.3 ->
releaseBufferMutex.3 -> releaseBuilderSem.3 -> dequeue.3 -> crawl.3 -> checkTree.3 -> checkQueue.3 -> grabCrawlerSem.3 ->
grabBufferMutex.3 -> addToBuffer.3 -> releaseBufferMutex.3 -> releaseBuilderSem.3 -> grabBuilderSem.1 -> setNotFinishIteration.1 ->
removeFromBuffer.1 -> check.1 -> [if!(!(SHOW_DATARACE))] -> check.1 -> createFile.1 -> writeData.1 -> [if!(!(SHOW_DATARACE))] ->
releaseCrawlerSem.1 -> setFinishIteration.1 -> updateCrawlerJoined.3 -> dequeue.2 -> crawl.2 -> checkTree.2 -> checkQueue.2 ->
addToQueue.2 -> grabCrawlerSem.2 -> grabBufferMutex.2 -> releaseBufferMutex.2 -> releaseBuilderSem.2 -> dequeue.2 -> crawl.2 ->
checkTree.2 -> checkQueue.2 -> grabCrawlerSem.2 -> grabBufferMutex.2 -> addToBuffer.2 -> releaseBufferMutex.2 -> releaseBuilderSem.2
-> grabBuilderSem.1 -> setNotFinishIteration.1 -> removeFromBuffer.1 -> check.1 -> [if!(!(SHOW_DATARACE))] -> check.1 ->
updateCrawlerJoined.2 -> dequeue.1 -> crawl.1 -> checkTree.1 -> checkQueue.1 -> addToQueue.1 -> grabCrawlerSem.1 ->
grabBufferMutex.1 -> releaseBufferMutex.1 -> releaseBuilderSem.1 -> dequeue.1 -> crawl.1 -> checkTree.1 -> checkQueue.1 ->
grabCrawlerSem.1 -> grabBufferMutex.1 -> addToBuffer.1 -> releaseBufferMutex.1 -> releaseBuilderSem.0 -> grabBuilderSem.0 ->
setNotFinishIteration.0 -> removeFromBuffer.0 -> check.0 -> [if!(!(SHOW_DATARACE))] -> check.0 -> createFile.1 -> writeData.1 ->
[if!(!(SHOW_DATARACE))] -> releaseCrawlerSem.1 -> setFinishIteration.1 -> createFile.0 -> writeData.0>

********Verification Setting********
Admissible Behavior: All
Method: Refinement Based Safety Analysis using DFS - The LTL formula is a safety property!
System Abstraction: False
********Verification Statistics********
```

Verification Completed

Output Window

Specification is parsed in 0.0511404s
//=================Global Constants=======
#define BUFFER_SIZE 2;
#define NUM_BUFFERS 2;
#define NUM_CRAWLERS 4;
#define NUM_SEEDS 4;
#define NUM_URLS 6;
#define SHOW_DATARACE T;
//===========Global Variable Definitions===============
var urls = [5, 4, 4, 5, -1, -1];
var isNewUrlInQueue = [false, false, false, false];

Model Saved   ProcessModel.csp   Ln: 9   Col: 26   INS