Technical Report: Unanticipated Object Synchronisation for Dynamically-Typed Languages

Théo Rogliano, Guillermo Polito, Pablo Tesone, Luc Fabresse, Stéphane Ducasse

Abstract

Developing concurrent programs requires the use of thread-safe abstractions to avoid race conditions. Nevertheless, many libraries are not thread-safe either because it was not a concern for the developer or the developer purposely traded thread-safety for performance. All calls to such a library require to be synchronised in the client program.

While different synchronisation mechanisms exist, only lock-based solutions and transaction solutions allow one to add synchronisation without anticipation. But they are both about reasoning in terms of execution. It forces developers to explore the whole execution to identify when synchronisation is required. A developer accidentally missing a synchronisation point circumvents the mechanism. Instead by ensuring exclusive access to an object for a thread prevents other thread to access it and thus allows developers to miss synchronisation points without circumventing the mechanism.

In this paper, we propose the Atomic Samurai, a synchronisation model that is object-based, unanticipated, and while the object is synchronised the rest of the program normally runs concurrently. It relies on 3 mechanisms: pointer swapping to ensure reference unicity, proxies to control object accesses and late binding to intercept and redirect messages sent. We validate our approach by showing how Atomic Samurai helps both solving examples extracted from the literature and a real-world scenario from the Pharo community on font rendering. We also measure the impact in performance of our solution by comparing it with a semaphore-based solution. In term of performance, our solution is 5 orders of magnitude slower than the semaphore-based solution, but scales better with the number of processes.

Technical report: Unanticipated Object Synchronisation for Dynamically-Typed Languages

Théo Rogliano, Guillermo Polito, Pablo Tesone, Luc Fabresse, Stéphane Ducasse

Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRIStAL -
Centre de Recherche en Informatique Signal et Automatique de Lille,
F-59000 Lille, France

0.1 Introduction

Developing concurrent programs requires to use thread-safe libraries to avoid data races. Nevertheless, many libraries are not thread-safe either because it was not a concern for the developer or this latter purposely traded thread-safety for performance. A client developer using such libraries has limited solutions when writing a safe concurrent program.

Developing a new library, or adding thread-safety to the current one nullifies the benefit of using a library in the first place There is also no guarantee to find an equivalent thread-safe library. All calls to this library latter requires to be synchronised in the client program to avoid other types of race conditions such as bad message interleaving (See Section 0.2).

Programming languages offer several synchronisation mechanisms to write thread-safe code such as semaphores, message passing and transactions [23, 21, 4, 11]. We identified 4 properties that are desired: safety, performance, scalability, and applicability. Each mechanism has its own advantages and weaknesses regarding those properties but only two allows unanticipated synchronisation: locks and transactions. The two force the developer to reason in term of execution while an oop developer is used to think in terms of objects.

In this paper, we propose the Atomic Samurai model that allows programmers to use non-thread-safe libraries in concurrent programs. It allows unanticipated synchronisation with an object granularity. The programmer synchronises the object accesses with message sends. Once the object is synchronised the developer uses the object as in a sequential program (See Section 0.3). It relies on 3 underlying mechanisms, pointer swapping, proxies, and late binding. Pointer swapping ensure the unicity of a reference to the object. Proxies act as read and write barriers to the object and thus control process accesses to the object. Late binding intercepts and redirects any message sends at runtime.

In Section 0.5, we demonstrate that Atomic Samurai solves examples from the literature. It also solves a real world concurrency bug encountered in Pharo [6], an open source industrially used system, with a wrapper to C library that is not thread-safe. We also show the performance overhead incurred by Atomic Samurai.

0.2 The Need for an Unanticipated Synchronisation Mechanism

In this Section, we present the need for an unanticipated synchronisation mechanism. We first present the categories of concurrency bugs that are solved thanks to synchronisation mechanisms. Then, we show an example of wrapping a non-thread-safe library and extracted the challenges of

Table 1: Categories and descriptions of concurrency bugs.

Category of Concurrency Bugs	Bug Definition
Data race	Two threads access the same data and at least one of them modifies the data.
Bad message interleaving	Program exposes an inconsistent intermediate state due to the overlapping execution of two threads.
Message order violation	Expected order of execution of at least two memory access is not respected.

unanticipated synchronisation.

0.2.1 Categories of Concurrency Bugs

Concurrent programs are prone to different type of bugs as categorized in [15]. A safe program is a program without race condition bugs. There are three categories of race condition bugs: data races, bad message interleaving and message order violations. We summarized them in Figure 1.

Let's consider the code example shown in Listing 1 extracted from [3]. Listing 1 shows the code of a client method of a library. The library provides a registry for processes by associating each process with a name. It offers the whereIs: aName message that answers if a process is already registered by that name and the register: aProcess withName: aName that registers the process aProcess under the specified name aName. This client method first checks if a process is registered under a specific name (line 1). If there are no processes registered under that name (line 3/4) it spawns a new process (line 5) and registers it under the specified name (line 6). If the name was already taken, this method throws an error (line 7/8).

```
1   registered := registry whereIs: aName.
2
3   registered
4     ifNil: [
5       spawnedProcess := Process spawn.
6       registry register: spawnedProcess
7         withName: aName.
8       ↑ spawnedProcess.]
9     ifNotNil: [
10      self error: 'Already have a process
11        registered with the name ', aName].
```

Listing 1: Example of non thread-safe object and bad message interleaving

Data races. A data race bug happens when two threads access the same data and at least one of them modifies the data resulting in an incoherent

read or modification. In our example, registerName: aName for: aProcess is a write operation. It requires that no other access operation occurs during it. Sending any other message concurrently during this operation will end up in a corrupted data read, in the worst case a data not even representing an object. For example, in the register at the name specified instead of a reference to a process object, the register has a reference to a random location in memory.

Bad message interleavings. A bad message interleaving happens when a program exposes an inconsistent intermediate state due to the overlapping of two instructions. In the presence of a thread-safe library, synchronisation is required to avoid inconsistent reads. With the order of message presented in Figure 1, a first process, process1, sends wherels: message with 'MyProcess' as argument that returns that there is no process registered by this name. It spawns a new process then get interrupted before the registration is executed. A second process, process2, sends the wherels: message with the same argument 'MyProcess'. Process1 did not register yet, the name 'MyProcess' is still available. Then process2 gets interrupted and process1 continues its execution. Process1 register a process with the name 'MyProcess'. Process1 gets interrupted and Process2 resumes. Process2 replaces the registered process instead of throwing an error due to an already registered process.

Message order violation. A message order violation happens when the expected order of execution of at least two memory access is not respected.

All those problems come from a lack of a synchronisation mechanism. A solution to remove the data races is to give exclusive access to the registry to a process during the write operation. In the same manner, giving exclusive access to the registry during the method removes the bad message interleaving for this particular sequence of messages.

0.2.2 Real Example of Wrapping a Non Thread-safe Library

To further illustrate, we present an example of synchronisation for a non-thread-safe library in Pharo. Pharo uses FreeType, an external C library, to render glyphs and fonts. however, this library is not thread-safe and accesses to it in a concurrent environment needs to be synchronised to avoid race condition bugs presented in Section 0.2.1. The default synchronisation mechanisms in Pharo are semaphores. Every access to the library must be synchronised with a semaphore. It requires that the developer knows all the path of executions leading to an access to the library. Missing one synchronisation point means that the whole program is still subject to race conditions. In our example, the client application has been specifically designed

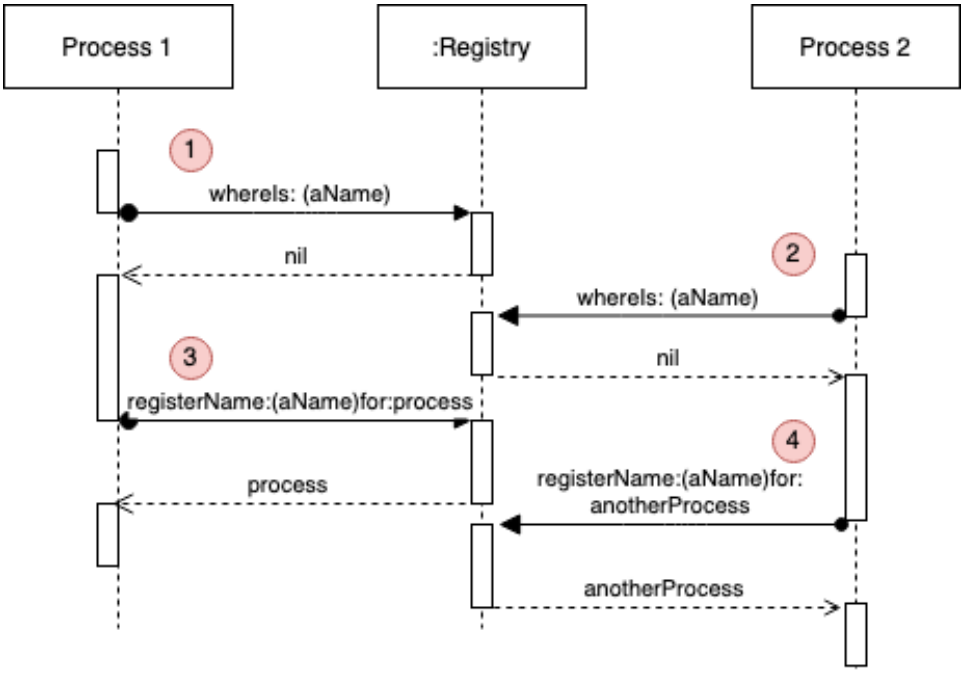

Figure 1: Order for bad message interleavings. 1. Process 1 checks first if the name is registered and spawns a new process. 2. Process 2 does the same. 3. Process 1 registers a process with the name. 4. Process 2 also spawns and tries to register a process with the same name instead of throwing an error.

with this matter in mind. The application groups the sensitive operations in a unique critical section in the method FreetypeCache¿¿atFont: aFreeTypeFont charCode: charCodeInteger type: typeFlag ifAbsentPut: aBlock such as Listing 2.

```
1
2     aFreeTypeFont mutex criticalReleasingOnError: [
3       self allMyOperationsOn: aFreeTypeFont.
4     ]
```

Listing 2: Critical section for freeTypeFont

Despite this effort, the client developer missed some operations that needs to be included in the critical section. A bug https://github.com/pharo-project/pharo/issues/8323 where two fonts are rendered glued together persists. Atomic Samurai solves it (See Section 0.5.2). Semaphores are about reasoning in execution paths for synchronisation. But developers need a mechanism that allow one to reason in terms of objects. We talk about unanticipated synchronisation because objects or libraries are not prepared beforehand for synchronisation.

0.2.3 Challenges of Unanticipated Synchronisation

We identified 3 challenges for doing unanticipated object synchronisation.

Dynamic intercession. The first challenge is that objects are not prepared for synchronisation. The synchronisation, then, must be added at runtime. It requires a mechanism that allows one to intercede for an object. This mechanism must handle the synchronisation in place of the object. Adding such a mechanism at runtime is a dangerous operation because, if not done properly, the runtime ends up corrupted. On the register example, the register is the object that need to be synchronised. If the addition of the synchronisation mechanism failed, it is not possible to access the registry anymore and the execution crashes.

Complete intercession. A second challenge is that the intercession mechanism must always intercede. If complete intercession is not achieved, the synchronisation will be bypassed, and the program will still be subject to race conditions. In dynamically-typed languages, late binding allows one to easily intercept most message sends but static message sends and potentially instance variable accesses are statically bound (*i.e.,* defined at compile-time). Super message sends also have a statically bound part. Such elements need to be identified and rebound dynamically. In the registry example, if process1 registers a process while process2 does a super message sends, this latter will not be intercepted and will bypass the synchronisation mechanism and the program will still be subject to race conditions bugs.

Unicity of Object Reference A third challenge is to ensure that there is a unique reference to the object and that this latter is owned by the intercession mechanism. Similarly, to the complete intercession, if a process uses directly the object, it bypasses the intercession mechanism and by extension the synchronisation mechanism. In the registry example, if process1 has its message intercede but process2 directly manipulates the object, the program is still subject to race conditions bugs.

An ideal unanticipated synchronisation mechanism for objects needs to be able to achieve dynamic intercession, complete intercession, and ensures the unicity of the synchronised object reference.

0.3 Atomic Samurai: an Unanticipated Synchronisation Mechanism

In this Section, we present Atomic Samurai, a model for unanticipated synchronisation of object-access.

0.3.1 Solution Overview

Atomic Samurai is an object-based synchronisation achieved with two message sends: take and release. Between those two messages, a developer is able to reason about the object in a sequential manner. The object does not require to be prepared for synchronisation such as in Java with the synchronized keyword. While the object is synchronised the rest of the program normally runs normally and concurrently. To ensure safe usage of an object, Atomic Samurai relies on 3 mechanisms, late binding, object proxification, and pointer swapping.

Dynamic intercession To address the challenge of dynamic intercession Atomic Samurai uses two mechanisms:

- Late binding is the resolution of the message send by looking up the method by name at runtime. Such resolution allows message interception and message rebinding.
- Proxies are special objects that intercept and handle messages in place of an object called the target. For example, it allows one to log, forward or even modify message sends. In Atomic Samurai, proxy act as barriers to determine if a process is allowed to manipulate the targeted object. The object is synchronised by proxification and unproxification.

Complete intercession To address the challenge of complete intercession Atomic Samurai uses safe synchronisation point. In some dynamically-typed languages, instance variable accesses are statically bound(*i.e.,* determined at compile-time). To ensure complete intercession, instance variable accesses must also be intercepted by the proxy. Once Atomic Samurai synchronises an object, the proxy will intercept the method accessing an instance variable and will rebind those accesses. But during the setup of the synchronisation, another process is free to access an instance variable. One solution is to verify that the synchronisation is achieved in a safe point. Before synchronisation, Atomic Samurai checks that the synchronised object is used by only one process. If it is not the case, Atomic Samurai retries the synchronisation later.

An alternative solution is to enforce that instance variable accesses are late bound in the language (*i.e.,* instance variable are accessed only through accessors). Both solutions have a different impact in performance, always going through accessors slightly impacts each instance variable accesses while checking all the stack of executions highly impacts the synchronisation. We measure this trade off in Section 0.5.

The class where super message sends start is also statically bound. We solved the case of super message sends in a specific way for our language

that we discussed in Section 0.4.3.

Unicity of Reference To address the challenge of unicity of reference, Atomic Samurai enforces the invariant that the proxy is always the unique owner of a reference to the synchronised object. It is achieved thanks to:

- Pointer swapping is mechanism that exchange the reference of two objects. Newly created object are only referenced by their creator. Atomic Samurai uses pointer swapping between a newly created proxy and the synchronised object. After swapping pointer between the proxy and the object, all references to the object points to the proxy and the reference previously to the proxy now points to the object. The proxy, then, stores the unique reference to the object.
- Delegation proxies [25] are reentrant proxies where the messages dispatched from the delegation proxy will also be catch by itself. Delegation proxies ensure that a message send does not leak a reference of the synchronised object outside the proxy.

Our Atomic Samurai library offers two messages:

take. The receiver of this message becomes exclusive to the process that executes it. This process is then called the taker process. To acquire exclusive access to the object, the taker process wait that no other process is actively using the object. The take message is reentrant for the taker process (*i.e.,* the taker process is free to send the take message multiple times).

release. The taker process removes its exclusive access on the receiver of this message. Other processes are free to access the object again. By default, blocked processes are unblocked. Sending this message to an object that has not be taken beforehand returns an error. It is not possible to release a taken object from a process other than the taker process. In case the object has been taken many times it needs as many release messages to release properly the object.

0.3.2 Atomic Samurai by Example

Figure 2 shows Atomic Samurai on the previous register example. The first part of the figure shows the initial state. There are two processes: process1 on the left and process2 on the right. Process1 sends the take message to the register to gain exclusive access on it.

The second part of the figure shows the state after the take message occurred. The take message checked process2 stack if the register is already in use which is not the case. It proceeds by creating a proxy (trap + handler)

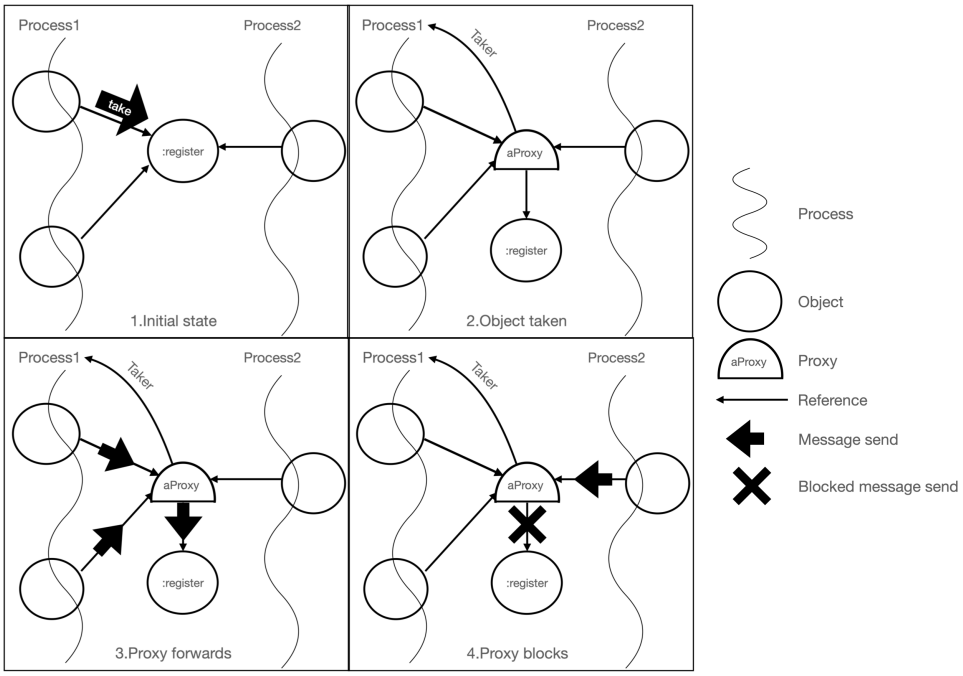

Figure 2: Example of Atomic Samurai with two processes.
1. Initial state.
2. State after take message, object is proxified.
3. Taker process, process 1, sends a message, the proxy forwards it.
4. Non taker process sends a message, the proxy blocks it.

and swaps the reference to the proxy with the ones of the register. The proxy is the only object with a reference to the original object, all other references point to the proxy. The proxy also keeps that process1 acquired the register. The third part of the figure shows that the messages are successfully sent from the taker process while the fourth part of the figure shows that the proxy acts as a barrier and blocks the message sends from the non-taker process.

It resolves the potential data race since only one process is able to access the registry between take and release. It is also possible to add thread-safety to a library by wrapping every call with a take and release on the receiver. It also removes the potential bad message interleaving because the second whereIs message will be blocked until the first process releases the registry. Take and release act as synchronisation on the registry.

0.4 Implementing Atomic Samurai in Pharo

In this Section, we first present the Pharo concurrency model. Then we present, the implementation of the take message. We follow up with discussions about corner cases of the implementation such as special objects, super message sends, over-proxification and deadlocks.

0.4.1 Pharo

The Pharo programming language implements concurrency with so-called *processes*: lightweight green-threads scheduled by the virtual machine. The process scheduler schedules processes depending on a priority. Processes are cooperative amongst the same priority and preemptive amongst different priorities. That is, a process can *yield* to give priority to another process in the same priority, and a process is suspended as soon as a higher priority process is ready [7]. Process switches happen on a timely basis but only at safe execution points: message sends and back jumps.

0.4.2 Take implementation

We present the code of the take message in Listing 3. The code between lines 3 and 8 is an uninterruptible block (code between square bracket) thanks to the message BlockClosure¿¿valueUnpreemptively. It allows the take message to be an atomic operation preventing concurrency bugs. In this block, the first operation line 4, checks that the object is not the receiver of another process. Otherwise, the intercession is not complete, and the take operation is not thread-safe. The check is done by comparing the object to the receiver of all contexts of all processes. In case the object is in use the process spinlocks and retries later to take the object, lines 10 and 11. If the object is not used, we proxify it with a delegation proxy, lines 5 to 8. During the proxification, all references to an object are swapped with the reference of a proxy thanks to the Object¿¿become: message. The only object that is left with a reference to the original object is the proxy. The proxy registers the taker process and acts as a barrier.

The proxy handles the messages from the taker process and, by default, spinlocks on the messages of other processes. The proxy also has a count for the number of times the taker process takes the object and releases the object only with the same number of release messages.

```
1  tryTake
2    | safe |
3    [
4    safe := self isNotUsed.
5    safe ifTrue: [
6      ↑ ProxyForAtomic
7           becomeTarget: self
8           withProcess: Processor activeProcess ] ]
```

```
9             valueUnpreemptively.
10
11      Processor yield.
12      self tryTake
```

Listing 3: Take implementation

Back to the example, we wrap the previous code Listing with registry take and registry release as in listing 4.

```
1  registry take.
2
3  registered := registry whereIs: aName.
4
5  registered
6    ifNil: [
7      spawnedProcess := Process spawn.
8      registry registerName: aName for: spawnedProcess ]
9    ifNotNil: [
10     self error: 'Already have a process
11       registered with the name ', aName ].
12
13 registry release.
```

Listing 4: Example of code wrapping using Atomic Samurai

0.4.3 Super Message Sends

Dynamically swapping two references during execution, with primitive become: produces execution bugs with super message sends such as a method wrongly applied to an object.

To better this show let's make an example. Let's take a method in class B whose superclass is A.

```
1  setter: anObject
2
3      self proxify.
4      self alternativeSetter: anObject.
5      self unproxify.
```

Listing 5: Example of method using the proxy and a self message send.

The setter: method from Listing 5 proxifies the object then sends the alternativeSetter: message and unproxifies the object. The proxification relies on pointer swapping with primitive become:. The pointer swapping replaces self with the proxy thus a message send to self is a message send to the proxy. The method lookup starts in the proxy class and the message is trapped by the proxy mechanism.

Let's consider this slightly different setter: method as in Listing 6. Instead of sending alternativeSetter: anObject to self, it sends it to super.

```
1  setter: anObject
2
```

```
3     self proxify.
4     super setter: anObject.
5     self unproxify.
```

Listing 6: Example of method using the proxy and a super message send.

The receiver is still the proxy, but with a super send the method lookup starts in the superclass where the method is defined, here class A. Class A is not in the hierarchy of the proxy. It avoids the proxy mechanism and, in this case, executes the found method on the proxy and corrupts it by changing the values inside the proxy. Swapping the proxy and the receiver object is not enough for super message sends because super message send lookup does not involve the receiver. A solution that we implemented in the virtual machine of Pharo is to modify the super message send lookup to check if the receiver is valid. A valid receiver is a receiver that has in his class hierarchy the class where the super message send lookup starts. If the receiver is not valid, the virtual machine creates an exception that the proxy catches and uses for interception.

0.4.4 Special Object Protection

Our prototype has some limitations. The take message is redefined for instances of Process to rise an exception. It is not possible to synchronise a process object because Atomic Samurai compares process's identity and taking a process disables the possibility to compare it with another process.

It is also not possible to take classes because the current delegation proxy library that Atomic Samurai uses does not handle these.

0.4.5 Lazy Proxification

Atomic Samurai always proxifies the object. Both the acquiring process and the other processes pay the cost of the proxy mechanism. With a capability system, the taker is able to use directly the object while other processes pass through the proxy.

Another case is when an object cannot be used in other processes. Atomic Samurai will create a proxy that is then costly in space and performance. If Atomic Samurai knows an object is unable to escape the scope of a process, it could let the acquiring process uses the object directly.

0.5 Evaluation

In this Section, we show the evaluation of our solution. We show how it solves problems from the literature and a real-world example from a concurrency bug in Pharo. Then, we compare the performance and the scalability of our solution to an equivalent semaphore solution.

Table 2: Example of bugs resolved.

N°	Bug type	Source	Description
1	Bad message interleaving	Fif 1 in [3]	
2	Data races	Fig 2 in [3]	
3	Data races	Fig 2 in [3]	
4	Data races	Sec 5 of [13]	
5	Bad message interleaving	Fig 1 in [2]	
6	Data races	Subsec 8.4.2 of [19]	
7	Data races	Subsec 8.4.2 of [19]	
8	Bad message interleaving	Subsec 8.4.1 of [19]	
9	Bad message interleaving	Sec 1.2 of [1]	
10	Bad message interleaving	Fig 2 in [12]	

0.5.1 Examples from literature

We validate our solution by implementing examples from the literature. Lopez et Al [15] provides a survey and a classification of all possible concurrency bugs for the Actor model. It also gives pointers to similar bugs for the thread model. Table 2 presents 10 of this bugs that we reproduced with failing tests to highlight them. We then used Atomic Samurai to solve them and make the tests pass without modifying the internal code to validate that we were able to solve the problem only at the caller side.

Data races. Data race solutions all consist in wrapping every access to the shared object with take and release. Inside such a section, only one thread is able to manipulate the state of the object thus removing all data races.

Bad message interleaving. Bad message interleaving solutions all consists in having a larger section that cover the messages that should not interleave. The difficult part is identifying which messages produce a bad interleaving. The fact that those examples come from the literature helps in knowing which messages produce such interleaving.

0.5.2 Real world example in Pharo

To further validate the correctness of our solution, we deployed Atomic Samurai to solve one real world example from Pharo. Pharo relies on

FreeType, an external C library, to render glyphs and fonts. Each font has a different face, and the sensitive operations are grouped and executed inside a critical section. Two fonts that have the same name also share the same face which was unexpected behavior. The critical section is not enough anymore since it does not cover face operations. We had, at least, one failing test reproducing consistently this data race. Pharo is able to query the methods where an instance variable is used. We collected the ten methods using the face and wrapped them with AtomicSamurai take and release messages such as in Listing 7.

```
1    getLinearWidthOf: aCharacter
2
3      face take.
4      ...
5      self methodBody.
6      ...
7      face release.
```

Listing 7: Example of method wrapping

We found deadlock issues caused by the critical section and we replaced this latter with our solution applied on the font object. With our solution the test reproducing the issue passes and all other tests on the fonts were still passing.

0.5.3 Performance: Current Pharo Implementation

In this Section, we measure the overhead of our solution compared to semaphores, Pharo's default synchronisation mechanism. All measurements are microbenchmark executed on the same computer with a 2.4 Ghz Intel Core i5 quadcore processor and 16 Gio 2133 Mhz LPDDR3 ram with all other applications closed. Each microbenchmark is repeated 100 times. A violin plot shows the distribution of measurements and the average. We present the two sources of overhead measured, synchronisation overhead and object-access overhead.

Synchronisation overhead. The synchronisation overhead is the overhead that is induced by the synchronisation mechanism itself.

Figure 3 shows the overheads of each mechanism. As a baseline, we use the default synchronisation mechanism offered in Pharo: Semaphores. A semaphore uses the messages signal and wait to synchronise the execution. This overhead is represented by the left violin plot. The right violin plot represents the overhead induced by using Atomic Samurai. Atomic Samurai uses the messages take and release to synchronise the execution. Atomic Samurai usage is 10^5 times slower than a semaphore. Most of the overhead comes from the fact that the take message must proxify the object at a safe point in the execution. This is a requirement because the current Pharo implementation allows direct access of instance variables.

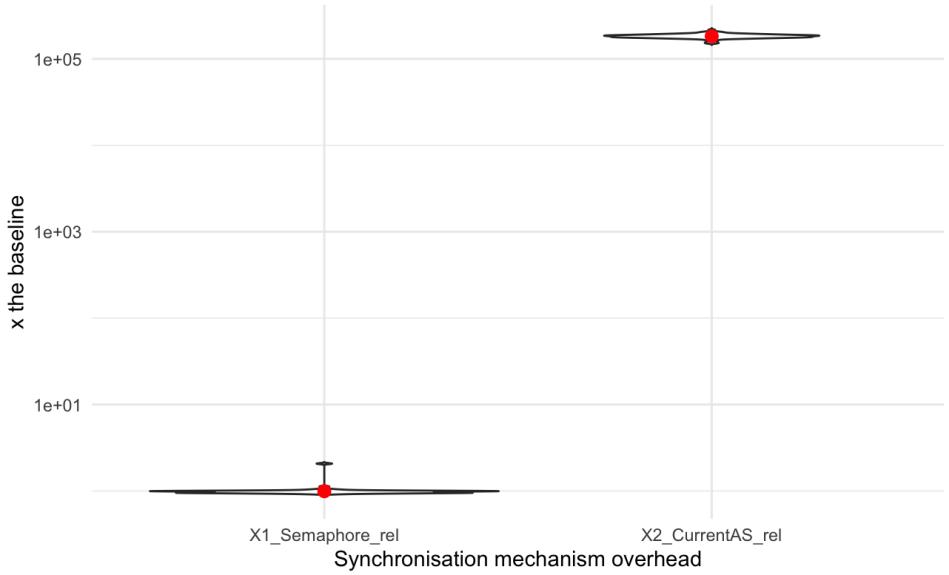

Figure 3: Overhead induced by synchronisation mechanisms in current Pharo implementation:
-Semaphore signal/wait.
-Atomic Samurai with safe point creation.
The lower the value the faster the implementation.

Object-access overhead. The object-access overhead is the overhead induced by accessing a synchronised object.

Figure 4 shows the overheads of each mechanism. As a baseline, using a semaphore allows the execution direct object-access. This is represented by The left violin plot. In Atomic Samurai case, object-access is intercepted and executed by the proxy thus producing an overhead. The right violin plot represents this overhead relative to the baseline. It is 10^5 times slower than a direct access. Most of the overhead comes from the delegation proxy internal mechanisms to ensure that a reference to the synchronised object is not leaked.

Atomic Samurai is slower in both sources of overhead. however, the current Pharo implementation allowing direct accesses disadvantages Atomic Samurai. In languages such as Python, object-access is always achieved via accessor methods (setter/ getter). It changes the distribution of overhead, object-access overhead is accentuated while for Atomic Samurai synchronisation overhead is alleviated.

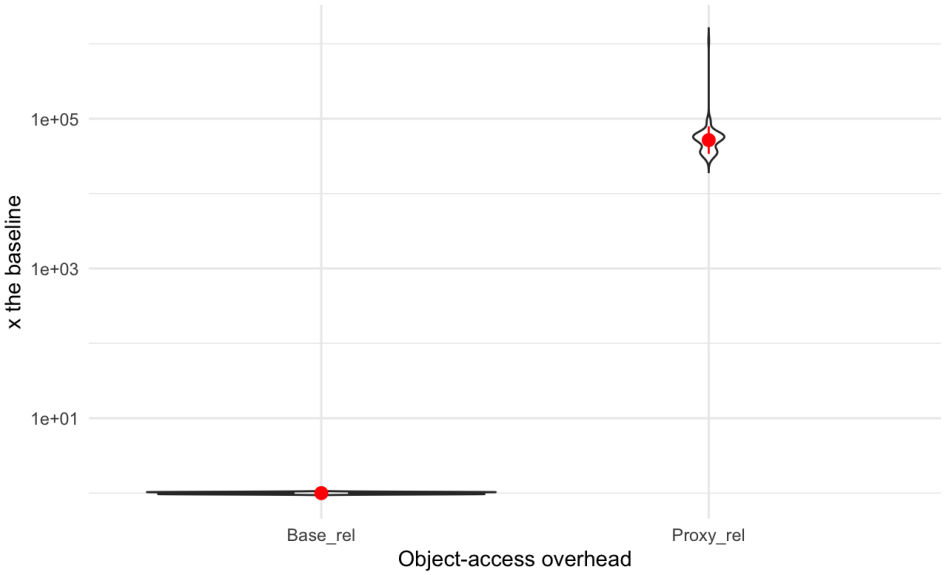

Figure 4: Time to access an object:
-Baseline.
-Through proxy
The lower the value the faster the implementation.

0.5.4 Performance: Pharo with Access Encapsulation

In this Section, we measure the overhead of our solution by mimicking a Pharo implementation that does not allow direct object-access and we compare it to an implementation that allows it. The measurements are done in the same manner as in Section 0.5.3. We also compare the synchronisation and object-access overhead.

Synchronisation overhead. Always going through an accessor means that object accesses are not statically bound anymore. It is therefore unnecessary to proxify the object at a safe point anymore in the take method. The right violin plot in Figure 5 represents the overhead induced by the new take message implementation. First, we compare it to the baseline with semaphores. This implementation of Atomic Samurai is 10^3 times slower than the baseline implementation. however, Figure 6 shows that the new take implementation is 10^2 times faster than the current implementation. We conclude that object proxification is a slow operation. We are working toward a faster object proxification especially by creating a faster pointer swapping primitive.

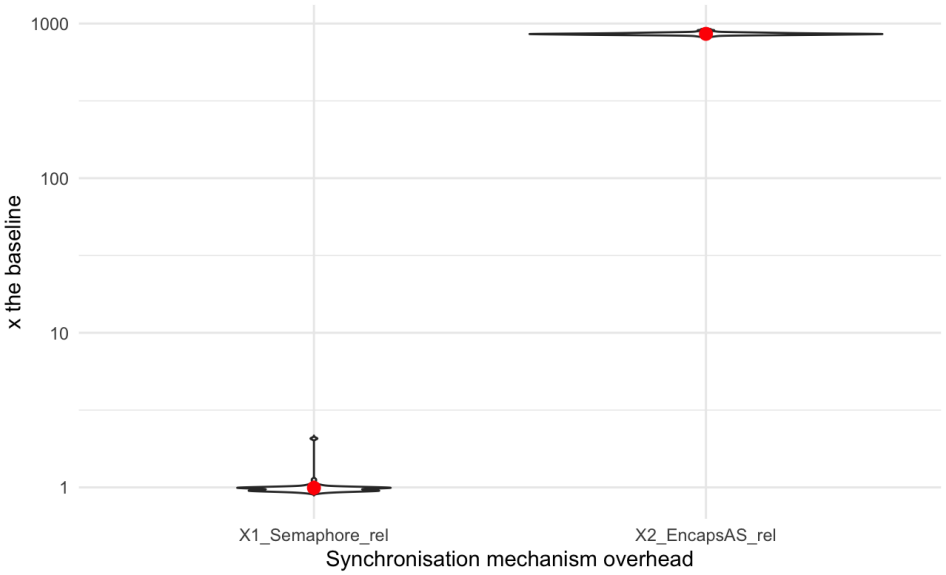

Figure 5: Overhead induced by synchronisation mechanism:
-Baseline with semaphore.
-Atomic Samurai with accessor implementation.
The lower the value the faster the implementation.

Object-access overhead. For object-access overhead, going through accessors produces an additional overhead compared to a direct access. The left violin plot in Figure 7 represents direct access as baseline. The right violin plot represents the overhead with access through accessors. Going through accessors take double the time than a direct access, but it is a minor source of overhead compared to the one induces by going through the proxy mechanism. It takes more than $1.7 * 10^6$ accesses through accessors to take more time than checking that the synchronisation is in a safe point.

To conclude, with a different language implementation our prototype of Atomic Samurai gains in speed. Forbidding direct object-access impacts slightly object-access and heavily speeds up object synchronisation. In the same manner, different pointer swapping and message interception would speed up our prototype. We are currently working toward optimizing our pointer swapping implementation. Also, escape analysis allows to use the object directly when a reference does not escape to another process. In those cases, the object-access overhead would be the same as with semaphores.

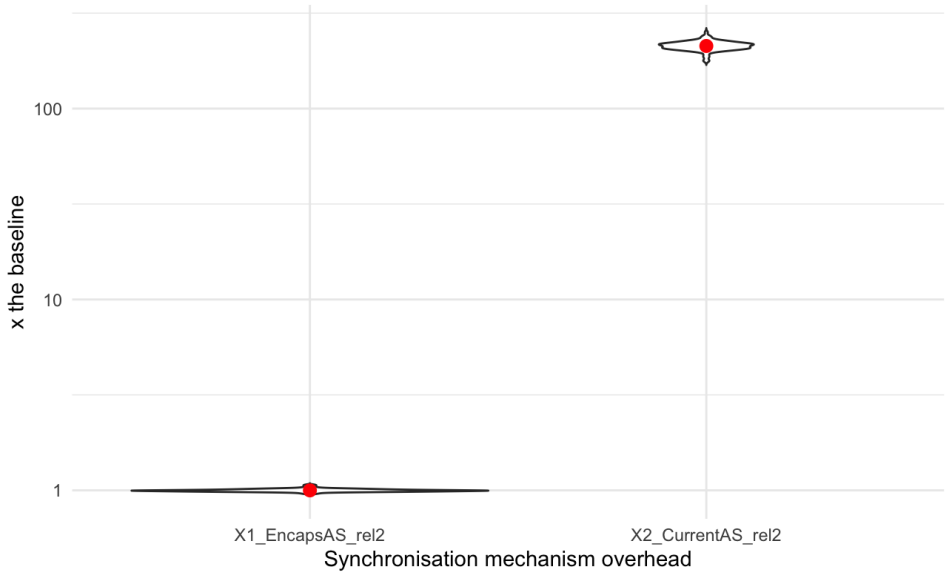

Figure 6: Overhead induced by synchronisation mechanism:
-Atomic Samurai with accessor implementation.
-Atomic Samurai with current implementation.
The lower the value the faster the implementation.

0.5.5 Scalability

In this Section, we measure the overhead induced by our solution in a concurrent environment. We compare our solution to a similar code synchronised with semaphores. Figure 8 shows the overhead depending on the number of threads. We measured for two, four, eight, and sixteen threads on the x axis and reported the value relative to the overhead for one thread. The x axis is in logarithmic scale and the y axis is on a normal scale. The semaphore implementation, in orange or on the top, scales linearly. It appears exponentially due to the x axis being logarithmic and y axis being normal. Atomic Samurai scales more in a logarithmic manner (*i.e.,* it appears linearly in the logarithmic scale).

0.6 Related Work

In this Section, we first present the four properties desired in unanticipated object synchronisation then we present a non-exhaustive list of object synchronisation models in other languages.

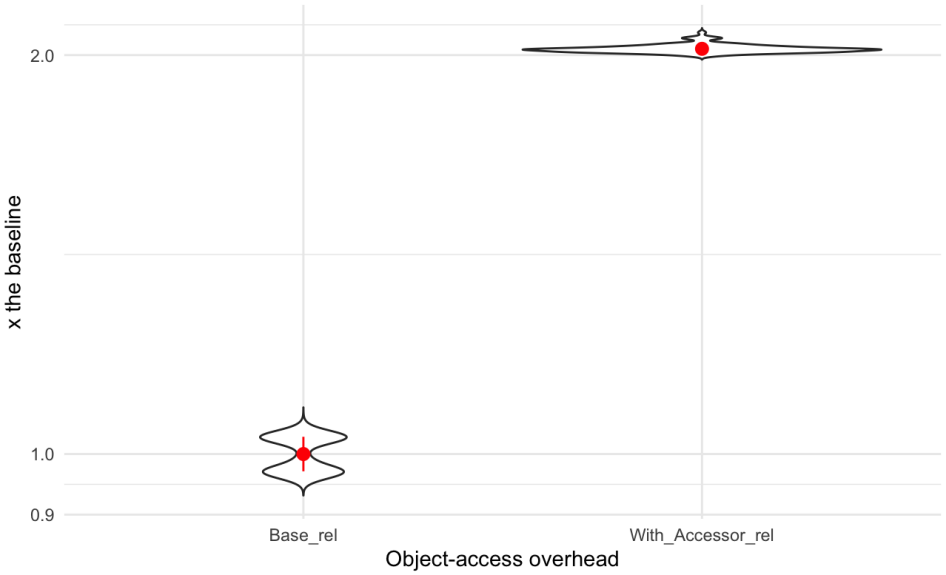

Figure 7: Time to access an object:
-Baseline.
-With accessor.
The lower the value the faster the implementation.

0.6.1 Properties

We first define the four properties that are safety, performance, scalability and applicability.

Safety. is the guarantee that the mechanism offers against race conditions. While all mechanisms aim to avoid race conditions they are subject to misuses in different ways. For example, synchronising a variable access with a critical section require that all accesses are actually inside the critical section, otherwise any access outside it circumvents the synchronisation.

Performance. is the performance of the synchronisation and access operations on the object for non concurrent execution. Adding synchronisation brings an overhead compared to a program that don't require it in two different way. First, the overhead directly coming from the addition of the synchronisation mechanism, for example acquiring a lock, sending an object through a channel. Second, some mechanisms ensure invariants at runtime by adding invariants checking instead of ensuring it by construction.

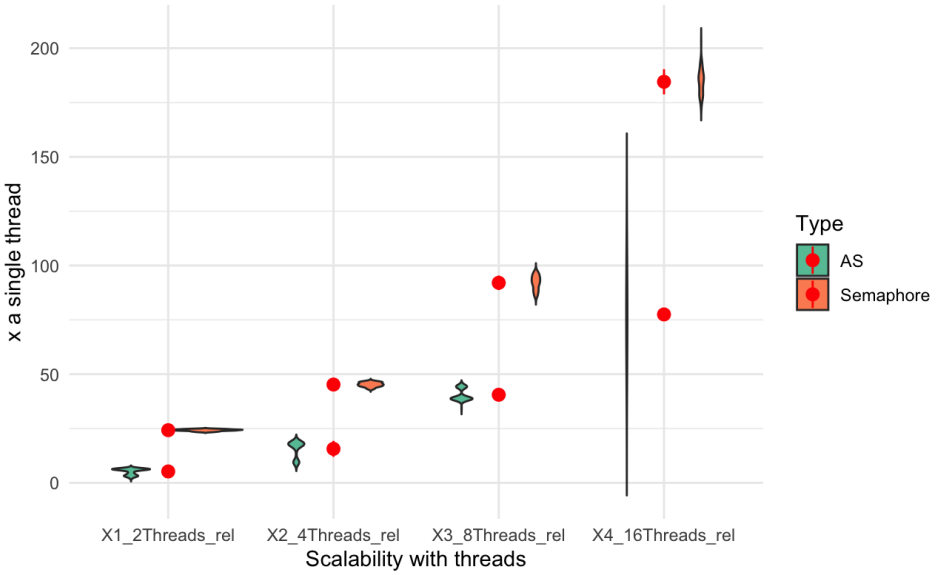

Figure 8: Overhead of synchronisation with semaphores depending on the number of processes. higher = more overhead.

Scalability. is the performance of the synchronisation and operations on the object when submitted to multiple concurrent executions. In this case, the overhead source remains the same but face new problem such as execution having to wait that another one finishes its synchronisation or invariants check.

Applicability. is the ability of the solution to add unanticipated synchronisation to a non-modifiable library. Some synchronisation solutions, such as channels, require modification on the library code.

Table 3: Existing solutions and their strong points to make collections concurrent. The more stars, the better

Property	**Safety**	**Performance**	**Scalability**	**Applicability**
Locks	**	**	**	***
Atomic Operations	**	***	**	*
Message passing	*	***	***	*
Transactions	**	*	***	***
AtomicSamurai	***	*	**	***

0.6.2 Approaches

Locks. These solutions consist in locking all accesses or modifications to the data with a lock (mutex, semaphore, etc...). All executions not holding the lock wait passively or actively for the holder to finish its operation. On lock release this information is broadcasted and the scheduler is in charge to wake up or signal waiting executions that the lock is free.

These solutions are known to be often misused[14]. The developer needs to locate every data access to add the synchronisation mechanism. Missing one access circumvents the whole mechanism. With multiple locks lack of progress issues appear such as deadlocks where two executions wait the release of their respective acquired locks.

In term of performance, these solutions add an overhead on all data accesses or modifications. It is possible to reduce the overhead by locking only part of the data (partial locking [17]) or by emphazing the overhead on modification operations that are less frequent and lightening the access operations that are more frequent.

A lock contention is an attempt to acquire an already acquired lock. The scalability is proportionnal to lock contentions. Making non-blocking acquire operations reduces the time to acquire a lock. Depending on the time it takes to acquire a lock, different acquiring strategies exist (spin lock, livelock, etc ...).

These solutions are applicable to a non modifiable library by wrapping all the calls to this library with a lock.

Atomic Operations. These solutions take advantage of uninterruptible (atomic) operations that are by definition not subject to data race conditions. With those uninterruptible operations such as Compare and Swap, it is then possible to build non interruptible data access or modification operations [10, 16, 18].

These solutions are guaranteed to make progress, they are not subject to livelock or deadlock. They eliminate data races but they cannot solve bad message interleaving that combine those operations.

These solutions induce a minimal overhead on access and modification operations.

Guerraoui et Al [9] reports a good scalability for their implementations of these kind of solutions.

These solutions are not applicable to a non accessible library since it does not permit to handle bad message interleaving.

Messages Passing. These solutions synchronise by passing data through a first in first out data structure called channels for Communicating Sequential Process [11], pipe for pipeline or mailboxes for actors. When one process finishes processing a data, it signals it by adding the data to the

queue. Processes wait their turn to access the data through the queue and consume it. Programatically, the advantages are that the communication is easy for developers to reason about as a mean of synchronisation.

These solutions help in orchestrating concurrent processes but do not solve *data races* [8].

To transfer a data, it usually requires to copy the whole graph of data to be referenced thus it induces a linear-time overhead in the number of bytes copied. Apart from the copy, these solutions did not produce an overhead on data access or modification.

Transactions. In these solutions an execution freely modifies data and records every operations in a log [20, 5]. Eventually consistent data structures [24, 22] also keep an history of the data thanks to copies of the object. A commit is the operation that verifies the log that other transactions did not make concurrent changes to the data. A rollback is the operation undoing all changes logged. After the verification if a concurrent change has been made the transaction aborts and starts a roll back else the changes are adopted.

having an operation that is not wrapped by a transaction will not be logged and thus will bypass the validation mechanism. Also special care must be taken on operations that cannot be undone.

There are two sources of overhead. The one coming from maintaining the log and the one from committing transactions.

This solution scales well when there are no conflicting changes in the commit operation. Else it depends on the behavior adopted to solve the conflict. For example, re-executing the transaction from the beginning until it succeeds create contention point similar to those that happen when passively waiting for acquiring a lock. Similarly to locks, these solutions are applicable for a non accessible library by wrapping the calls to the library inside a transaction.

0.6.3 Summary

Table 2 shows an evaluation of related work by using the properties. Only Locks and Transactions are suitable to synchronise the calls to a non-modifiable library in a concurrent program. They both suffer from the fact that one non wrapped operation makes the system unsafe. Locks are midly performant and does not scale pretty well while Transactions are poorly performant and scale better.

0.7 Conclusion

A developer writing a concurrent program faces new categories of bugs that does not exist in sequential programs: race conditions. Atomic Samurai

allows to delimitate sections of code where an object is exclusive to a process. The developer then reason in a sequential manner for this object and avoid race conditions.

To evaluate Atomic Samurai we implemented this model in Pharo without any Virtual Machine support. We then demonstrate that we solved races conditions bugs from the literature and solved a race condition bug encountered in our system. Nevertheless, the developer must manually delimitate the section. A wrong delimitation is still subjects to race condition bugs. The other issue is that Atomic Samurai does not prevent lack of progress issue for now .

For the future, we want to tackle the first issue by providing semi-automatic or automatic delimitation of the section that needs atomicity. A lot of tools help finding race condition in the literature. Escape analysis, for example, give information of which objects are used in other processes. Such analysis also helps in preventing lack of progress issues by identifying escaping objects from the same object graph.

Bibliography

[1] C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13:207 – 227, 12 2003.

[2] M. Christakis, A. Gotovos, and K. Sagonas. Systematic testing for detecting concurrency errors in erlang programs. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 154–163, 2013.

[3] M. Christakis and K. Sagonas. Static detection of race conditions in erlang. In *Proceedings of the 12th International Conference on Practical Aspects of Declarative Languages*, PADL'10, page 119–133, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 1–12, 2015.

[5] D. Dice and N. Shavit. Tlrw: Return of the read-write lock. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 284–293, New York, NY, USA, 2010. Association for Computing Machinery.

[6] S. Ducasse. *Pharo with Style*. Square Bracket Associates, 2019.

[7] S. Ducasse and G. Polito. Concurrent programming in pharo, 2020.

[8] D. S. Fava and M. Steffen. Ready, set, go!: Data-race detection and the go language. *Science of Computer Programming*, 195:102473, 2020.

[9] R. Guerraoui, A. Kogan, V. J. Marathe, and I. Zablotchi. Efficient multi-word compare and swap. *CoRR*, abs/2008.02527, 2020.

[10] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, page 300–314, Berlin, Heidelberg, 2001. Springer-Verlag.

[11] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[12] S. Hong, Y. Park, and M. Kim. Detecting concurrency errors in client-side java script web applications. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 61–70, 2014.

[13] J. M. Hughes and H. Bolinder. Testing a database for race conditions with quickcheck: None. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, page 72–77, New York, NY, USA, 2011. Association for Computing Machinery.

[14] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[15] C. T. Lopez, S. Marr, H. Mössenböck, and E. G. Boix. A study of concurrency bugs and advanced development support for actor-based programs, 2018.

[16] M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[17] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, page 267–275, New York, NY, USA, 1996. Association for Computing Machinery.

[18] R. R. Newton, P. P. Fogg, and A. Varamesh. Adaptive lock-free maps: Purely-functional to scalable. *SIGPLAN Not.*, 50(9):218–229, aug 2015.

[19] S. K. Prasad, A. Gupta, A. L. Rosenberg, A. Sussman, and C. C. Weems. *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2015.

[20] L. Renggli and O. Nierstrasz. Transactional memory for Smalltalk. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 207–221. ACM Digital Library, 2007.

[21] N. I. Sarnak. *Persistent Data Structures*. PhD thesis, New York University, USA, 1986. AAI8706779.

[22] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. In X. Défago, F. Petit, and V. Villain, editors, *SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400, Grenoble, France, Oct. 2011. Springer.

[23] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, 1995.

[24] W. Vogels. Eventually consistent: Building reliable distributed systems at a worldwide scale demands trade-offs?between consistency and availability. *Queue*, 6(6):14–19, oct 2008.

[25] E. Wernli, O. Nierstrasz, C. Teruel, and S. Ducasse. Delegation proxies: The power of propagation. In *Proceedings of the 13th International Conference on Modularity*, pages 63–95, Lugano, Suisse, apr 2014.