Technical report: Object-centric Access Control Mechanisms in Dynamic Languages

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

► **To cite this version:**

Théo Rogliano, Guillermo Polito, Pablo Tesone, Luc Fabresse, Stéphane Ducasse. Technical report: Object-centric Access Control Mechanisms in Dynamic Languages. [Research Report] Inria Lille Nord Europe - Laboratoire CRIStAL - Université de Lille. 2022. hal-03784027

HAL Id: hal-03784027
https://hal.science/hal-03784027

Submitted on 26 Sep 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

Contents

1 Introduction **7**
- 1.1 Context: Concurrency . . . 7
- 1.2 Concurrent Program Specificities . . . 7
  - 1.2.1 Data races . . . 8
  - 1.2.2 Bad message interleavings . . . 8
  - 1.2.3 Message order violation . . . 9
  - 1.2.4 Synchronization . . . 10
  - 1.2.5 Real World Example of Synchronization . . . 11
- 1.3 Limitations of Planified Synchronization . . . 11
  - 1.3.1 Concurrency is Anticipated . . . 12
  - 1.3.2 Challenges of Unanticipated Synchronization . . . 12
- 1.4 Object Graph Access-control Challenges . . . 13
  - 1.4.1 Object Graph Transfer by Example . . . 13
  - 1.4.2 Challenges of Object Graph Transfer . . . 13
- 1.5 Research Question . . . 16
- 1.6 Contributions . . . 16
- 1.7 Structure of the Thesis . . . 17

2 State of the Art: synchronization **18**
- 2.1 Desired Properties . . . 18
- 2.2 Synchronization models . . . 19
  - 2.2.1 Locks family . . . 19
  - 2.2.2 Atomic Operations . . . 20
  - 2.2.3 Message Passing . . . 20
  - 2.2.4 Transactions . . . 21
  - 2.2.5 Summary . . . 22
- 2.3 Conclusion . . . 23

3 State of the Art: Intercession **24**
- 3.1 Introduction . . . 24
- 3.2 Barriers . . . 25
- 3.3 Object swapping . . . 26
- 3.4 Proxy . . . 26
  - 3.4.1 Java proxies . . . 27

3.4.2 Javascript Proxies 27
3.4.3 Racket/Scheme Proxies 28
3.4.4 Smalltalk Proxies 28
3.5 Conclusion 28

4 Classification of Object-Access Control Permissions 29
4.1 Introduction 29
4.2 Object-Access Rights 29
4.3 Identifying Permission Transfer Semantics 29
4.3.1 Copy Value Graph Transfer (CVGT) 31
4.3.2 Full Ownership Graph Transfer (FOGT) 32
4.3.3 Exclusive Write Object Transfer (EWOT) 33
4.3.4 Read-only Object Transfer (ROOT) 35
4.3.5 Channel Limits: Transactionality and Inconsistent Reads 35
4.4 Object-Access Control 36
4.5 Conclusion 36

5 Object-based Right Transfer Channels 37
5.1 Introduction 37
5.2 Efficient and Correct Object Graph Transfer in Dynamically-Typed Languages 37
5.2.1 Context: Pharo's Concurrency Model 37
5.3 Canal: An Extensible Channel Framework 37
5.3.1 Extensible Channels and Hooks 38
5.3.2 Channel Transfer by Example 39
5.3.3 In other languages 39
5.3.4 Shared Memory 40
5.3.5 Per-Process Ownership Model 42
5.4 Permission Transfer Channels 42
5.4.1 Copy Value Graph Transfer (CVGT) 42
5.4.2 Full Ownership Graph Transfer (FOGT) 42
5.4.3 Exclusive Write Object Transfer (EWOT) 42
5.4.4 Read-only Object Transfer (ROOT) 42
5.5 Conclusion 42

6 Evaluation of Permission Transfer Channels 43
6.1 Introduction 43
6.2 Comparing the Different Channels 43
6.2.1 Scenario 1: Single Object Transfer Speed 43
6.2.2 Scenario 2: Object Graph Transfer Speed 44
6.2.3 Scenario 3: Single object-access Speed 46
6.2.4 Discussion 46
6.3 Classification 47
6.4 Conclusion 47

7 Atomic Samurai for Unanticipated Object synchronization **48**
7.1 Introduction 48
7.2 Solution Overview 48
7.3 Atomic Samurai by Example 49
7.4 Conlusion 50

8 Implemention of Atomic Samurai in Pharo **51**
8.1 Introduction 51
8.2 Pharo 51
8.3 Take implementation 51
8.4 Super Message Sends 53
8.5 Special Objects Protection 54
8.6 Lazy Proxification 54
8.7 Conclusion 54

9 Evaluation of Atomic Samurai **55**
9.1 Introduction 55
9.2 Examples from literrature 55
9.3 Real world example in Pharo 55
9.4 Performance 56
9.5 Scalability 57
9.6 Conclusion 57

10 Conclusion **61**
10.1 Summary 61
10.2 Contributions 61

Bibliography **62**

List of Figures

1.1 Sequence leading to a data race. Methods foo and bar modify the object simultaneously. 8
1.2 Sequence leading to a bad message interleaving. Thread1 executing foo modifies the object.Thread2 executing foo' also modifies the object. Thread1 executing bar reads the object. 9
1.3 Sequence expected by the program. Thread1 excutes the method setup. Thread2 executes the method test. Thread1 executes the method tearDown. . . 9

1.4 Sequence leading to a message order violation. Thread1 executes the method setup then executes the method tearDown. Thread2 executes the method test. 10
1.5 Alice communicates the car object to Bob. 14
1.6 Alice communicates the car object to Bob but Alice also has a reference on the key. 15

5.1 Overview of Object Transfer Through a Channel. 38

6.1 Result of data transfer speed for simple objects. The more times per second it is executed the fastest it is.
CVGT = Copy Value Graph Transfer (red).
FOGT = Full Ownership Graph Transfer (blue).
EWOT = Exclusive Write Object Transfer (green).
ROOT = Read-Only Object Transfer (purple). 44
6.2 Data transfer speed result for object graphs of different sizes. It is on number of times executed per seconds, the higher the value is, the fastest the implmentation is.
CVGT = Copy Value Graph Transfer (red).
FOGT = Full Ownership Graph Transfer (blue).
EWOT = Exclusive Write Object Transfer (green). 45
6.3 In blue, the non-instrumented accesses. In red, the accesses instrumented with the write barrier 46

7.1 Example of Atomic Samurai with two processes.
1. Initial state.
2. State after take message, object is proxified.
3. Taker process sends a message, the proxy forwards it.
4. Non taker process send a message, the proxy blocks it. 50

9.1 Overhead induced by synchronization mechanism:
-Atomic Samurai take/release without stack traversal.
-Semaphore signal/wait.
-Atomic Samurai with stack traversal.
The lower the faster the implementation is. 58
9.2 Time to access an object:
-Going through accessor
-Baseline.
-With proxy
The lower the faster the implementation is. 59
9.3 Overhead of synchronization with semaphores depending on the number of processes. Higher = more overhead. 60
9.4 Overhead of synchronization with atomic samurai depending on the number of processes. Higher = more overhead. 60

List of Tables

1.1 Categories and descriptions of concurrency bugs. 10

2.1 Existing solutions and their strong points to make collections concurrent. The more stars, the better. Stars between parenthesis are conditionnal. . . . 23

3.1 Categories and descriptions of concurrency bugs. 25
3.2 Existing solutions for making collections concurrent. More *** the better. Less * the worst. ** In the middle.. 28

4.1 Four permission transfer semantics based on the evolution of the sender and receiver processes' permissions on the transferred object A. The letters W and R represent respectively the write and read permissions of a process on A. Having a '_' instead of a permission means that a process does not have this permission on the object. $\emptyset$ means that a process does not hold a reference on the object because it never had it or lost it. A' is a copy of object A. 30
4.2 Summary of the Permission Transfer Channel's Properties. 34

5.1 Channels permission transfer semantics offered in other languages. 39

List of Listings

1.1 Critical section for freeTypeFont. 11
4.1 Redefinition of send: for Copy Value Graph Transfer Channel. 32
4.2 Redefinition of receive for Copy Value Graph Transfer Channel 32
4.3 Redefinition of send: for Full Ownership Object Transfer Channel. 33
4.4 Redefinition of receive for Full Ownership Object Transfer Channel. 33
4.5 Redefinition of send: for Exclusive Write Object Transfer Channel. 34
4.6 Redefinition of receive for Exclusive Object Transfer Channel. 34
4.7 Usage Example of a Read-Only Object Transfer Channel. 35

5.1 Definition of send: and receive methods of the Channel base class. 41
5.2 Usage Example of a Channel. 41
6.1 Code example for benchmarks. 43
8.1 Take implementation 52
8.2 Example of code wrapping using Atomic Samurai 52
8.3 Example of method using the proxy and a self message send 53
8.4 Example of method using the proxy and a super message send 53
9.1 Example of method wrapping 56

Chapter 1

Introduction

1.1 Context: Concurrency

With Moore's law reaching its limits, programs rely more and more on parallelism and concurrency to continue to improve the execution speed.

Concurrency is the ability of a program to have parts that are executed out of order or in partial order without affecting the outcome. For example, an I/O operation such as a write operation to a file is a long one where a traditional program waits for it to finish. A concurrent program will execute another part while the I/O operation terminate thus not losing time.

A thread is a part of a program that executes independently, potentially out of order or in partial order. A scheduler orchestrates the order of thread execution. Processes and native threads are threads orchestrated by a scheduler provided by the operating system (OS). Lightweight or green-threads are threads orchestrated by a scheduler provided by a runtime-library or a virtual machine (VM) of a programming language. There are two strategies to schedule threads that are not exclusive:

- Cooperative threads are those that voluntarily yield control back to the scheduler. All threads must cooperate for the scheduling scheme to work (*i.e.,* each one of them managed to have time for execution).
- Preemptive threads are those that rely on the scheduler to determine which thread to execute.

While threads are independent execution units, they still interact with each other to compute the result of the program. Specific interactions produce bugs that, de facto, do not exist in sequential programs.

1.2 Concurrent Program Specificities

Some of the interactions produce bugs when threads share and manage the memory. In an object-oriented language, a shared memory location is a shared object. These interactions

are called race conditions. There are 3 kinds of race conditions that have been categorized by Lopez et al. [2018]: data races, bad message interleaving, and message order violations.

1.2.1 Data races

Data races are bugs that happen when two threads access the same data almost simultaneously and at least one of them modifies the data. Figure 1.1 shows a sequence leading to a data race, Thread1, on the left, executes method foo and Thread2, on the right, executes method bar at the same time. Method foo modifies the object and method bar reads it. In the worst case, the read operation from method bar returns a mix between the old value and the modified one of method foo and does not even represent an object, thus breaking the program.

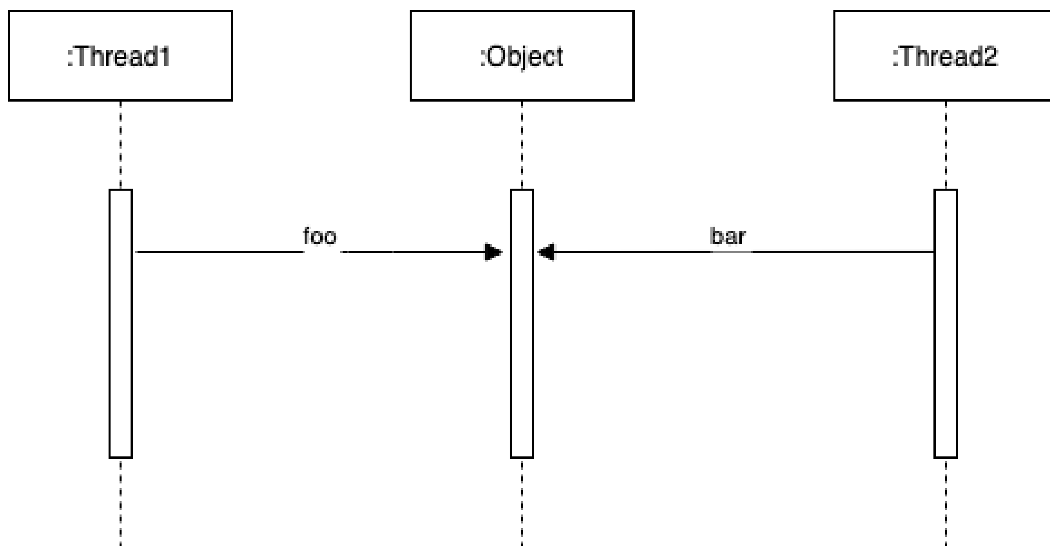

Figure 1.1: Sequence leading to a data race. Methods foo and bar modify the object simultaneously.

1.2.2 Bad message interleavings

A bad message interleaving or high-level data race is a bug that happens when a program exposes an inconsistent intermediate state due to the overlapping of two instructions. Figure 1.2 shows a sequence leading to a bad message interleaving. Thread1, on the left, executes a method foo that modifies the object. Then, Thread2, on the right, executes a method foo' that also modifies the object. Thread1 executes method bar that reads the object with Thread2 modifying while only Thread1's modification is expected . In the worst case, Thread1 has an unexpected behavior because the object is not being formed as expected.

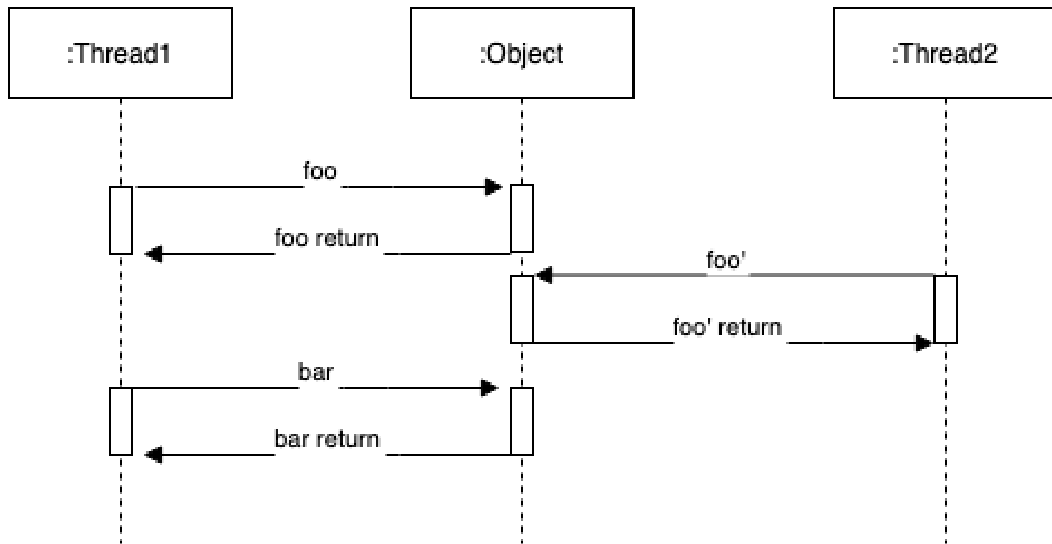

Figure 1.2: Sequence leading to a bad message interleaving. Thread1 executing foo modifies the object.Thread2 executing foo' also modifies the object. Thread1 executing bar reads the object.

1.2.3 Message order violation

A message order violation is a bug that happens when an expected order of execution of two methods is not respected due to the nature of threads whose order is not predefined. Figure 1.3 shows the expected sequence. Thread1 executes the method setup, then Thread2 executes the method test and afterward Thread2 executes the method tearDown.

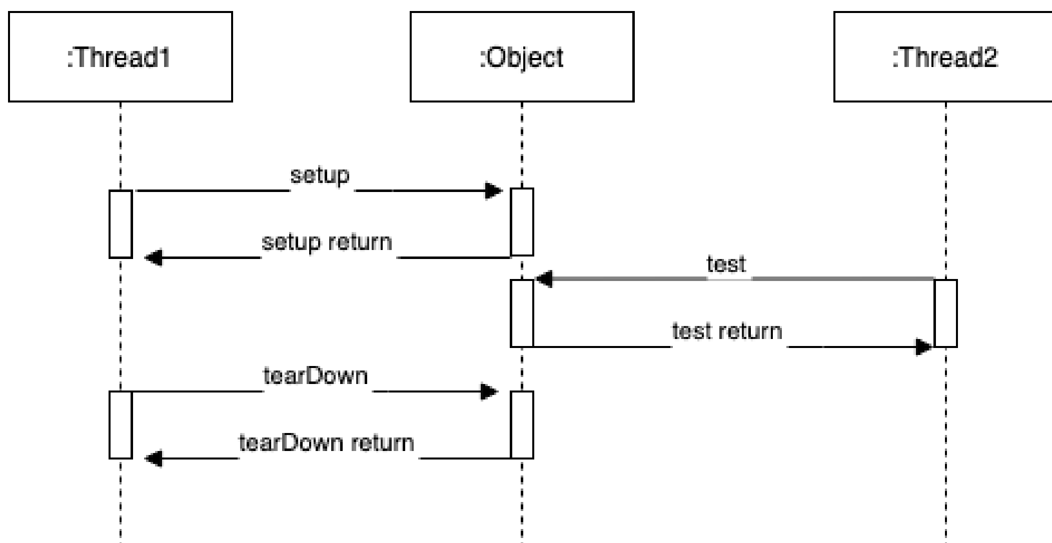

Figure 1.3: Sequence expected by the program. Thread1 excutes the method setup. Thread2 executes the method test. Thread1 executes the method tearDown.

Table 1.1: 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	An expected order of execution of at least two memory access is not respected.

Figure 1.4 shows the sequence leading to a message order violation. Thread1 executes the method setup then executes the method tearDown. Thread2 executes the method test that fails because the environment is not setup properly.

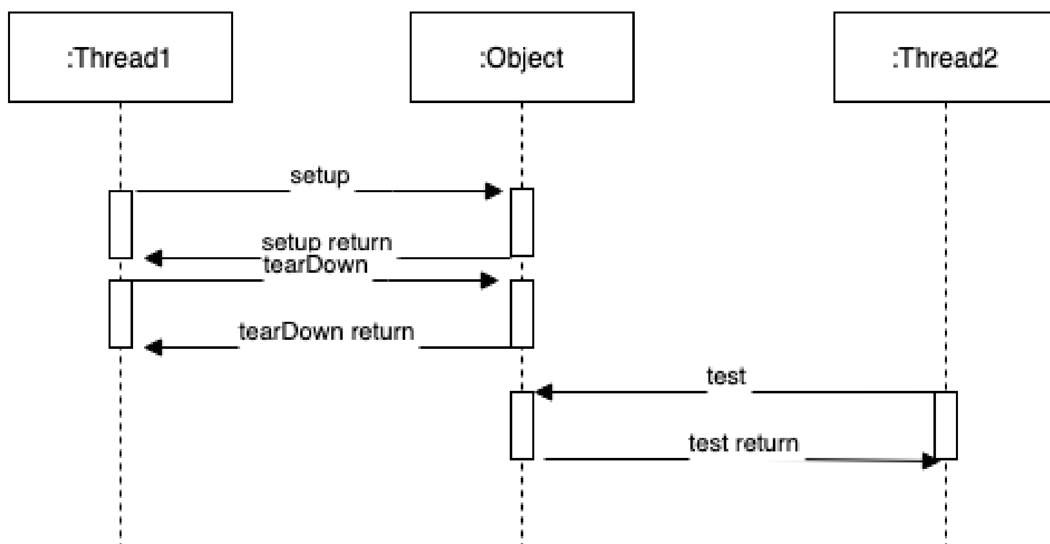

Figure 1.4: Sequence leading to a message order violation. Thread1 executes the method setup then executes the method tearDown. Thread2 executes the method test.

Three kinds of race conditions are summarized in Table 1.1.

Race conditions are constraints for a developer writing a concurrent program. In the worst case, they break the execution of the program. Therefore, developers want to prevent them from occurring in their program.

1.2.4 Synchronization

To avoid race conditions, a concurrent program needs to control which thread has an exclusive access to an object at a given time. Synchronization is the coordination of threads

to object-access. Data races are avoided by synchronizing the access to objects. Bad message interleaving is solved by synchronizing the execution of methods. Message order violations are solved by synchronizing the order of execution of methods. However, synchronizing program is known to be a difficult task with several challenges. To illustrate those challenges let's look at a real world example.

1.2.5 Real World Example of Synchronization

We present a case of synchronization for a non-thread-safe library in Pharo Black et al. [2009], a dynamically-typed language inspired from Smalltalk. Pharo integrated development environment (IDE) is written in Pharo and uses FreeType, an external C library, to render glyphs and fonts. This library is not thread-safe and accesses to it in a concurrent environment need to be synchronized. The low level synchronization mechanisms in Pharo are semaphores.

The Pharo IDE synchronizes accesses to a font with a semaphore in the method FreetypeCache»atFont: aFreeTypeFont charCode: charCodeInteger type: typeFlag ifAbsentPut: aBlock (Listing 1.1).

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

Listing 1.1: Critical section for freeTypeFont.

All concurrent accesses to a font are inside the critical section and must take the semaphore to ensure the synchronization. This design anticipates the three kinds of race conditions. However, a race condition exists in the current implementation [1] where two fonts are rendered glued together. This race condition is due to a concurrent access to an object referenced by the font, an attribute of the font: its `face`.

This race condition happens for two reasons:

Planification of Synchronization. Synchronization, even planned early in the development of a program, is difficult. In this case, at least one access to the library is missing inside the critical section.

Object Graphs Synchronization. Synchronizing an object is not only synchronizing the access to the reference of an object but all references reachable from this object.

Those two phenonmena represente greater challenges that will be explained in the following section.

1.3 Limitations of Planified Synchronization

In this section, we present the limititations of planification of concurrency and the challenges for unplanified synchronization.

[1] https://github.com/pharo-project/pharo/issues/8323

1.3.1 Concurrency is Anticipated

Concurrency is an early part of the design of a program development. The developer has to define which parts of the program should become threads and anticipate synchronization accordingly. The developer encounters several constraints such as using only concurrent external libraries, protecting all library calls with mutex or using a specific architectural style to ensure safety. If the program is not synchronized properly, finding and fixing the synchronization issues turn out to be difficult tasks.

In the example above, to achieve those tasks the developer needs to know all execution paths where the object's missing synchronization is accessed. To synchronize accesses, the developper then needs to change the architecture of the program so that those accesses are executed inside the critical section. In the worst case, it is required to rewrite the whole program. Rather than rewriting the program, the developper may try to add other critical sections. However, it does not prevent bad message interleaving between the newly created critical sections. Moreover, defensively synchronizing every access to the object hinders performance. As anticipating synchronization is not sufficient, the developer needs tools to deal with unanticipated synchronization.

1.3.2 Challenges of Unanticipated Synchronization

We identified 3 challenges in performing unanticipated synchronization for objects.

Dynamic intercession. The first challenge is that objects are not prepared for synchronization. The synchronization, then, must be added at runtime. It requires a mechanism that allows one to intercede for an object. This mechanism must handle the synchronization 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. In the example, the face of the font is the object that needs to be synchronized. If the addition of the synchronization mechanism fails, it is not possible to access to the face anymore and the PharoIDE cannot render this font.

Complete intercession. The second challenge is that the intercession mechanism must always intercede. If a complete intercession is not achieved, the synchronization will be bypassed, and the program will still be subject to race conditions. In dynamically-typed languages, late binding allows easy interception for most message sends except for static message sends such as super ones 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 example, if a thread sends a message to the face while another does a super message send, this latter will not be intercepted and will bypass the synchronization mechanism. As a result, the program will still be subject to race conditions bugs.

Unicity of Object Reference. The 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 synchronization mechanism. In the example, if a thread has its message intercede but another thread directly manipulates the object, the program is still subject to race conditions bugs.

An ideal unanticipated synchronization mechanism for objects needs to be able to achieve a not only complete but also dynamic intercession, and ensures the unicity of the synchronized object's reference. Synchronizing only a shared reference of the object is not sufficient. Objects form complex graphs of references. Transitively, a thread is able to reach any of these references to other objects thus all accesses to this complex graphs must be synchronized.

1.4 Object Graph Access-control Challenges

In this section, we show the challenges of object graph access control with an example.

1.4.1 Object Graph Transfer by Example

To introduce the problems of object graph transfer, let's consider the example illustrated in Figure 1.5. The example presents two processes and many objects shared between them. In this example, one process has a reference to the Alice object, and the other process has a reference to the Bob object. Alice has a car, which contains a disc, a key and some gas, and Bob has no reference to it: Bob cannot read, write or send messages to any of these objects.

If at some point during the execution Bob needs to use the car, we need to send a reference to the car from Alice to Bob, for example, by executing `bob car: alice car`, leading to the situation in Figure 1.6. As soon as Bob has a reference to the car, he obtains a complete access to it *i.e.,* reading, writing, and sending messages to it and all objects are reachable from the car.

Handling how objects are shared in a concurrent environment needs special attention. Such a model, in which object sharing relies on just sharing references, *i.e.,* an unrestricted sharing policy, introduces potential data-races. Indeed, if both Alice and Bob have regular references to the car, both may access and modify the entire object graph at the same time, which produces conflicting messages as side effects.

Even if we revoke Alice's reference to the car (*e.g.,* nilling it), there is still a possibility of data-races when there are shared objects, as it happens in the example with the key object which is directly referenced by Alice and also reachable by Bob from the car. Likewise, if the key has a reference to the car, the car is still reachable by Alice through the key.

1.4.2 Challenges of Object Graph Transfer

From the example above, we observe that sharing objects in a concurrent environment presents the following challenges:

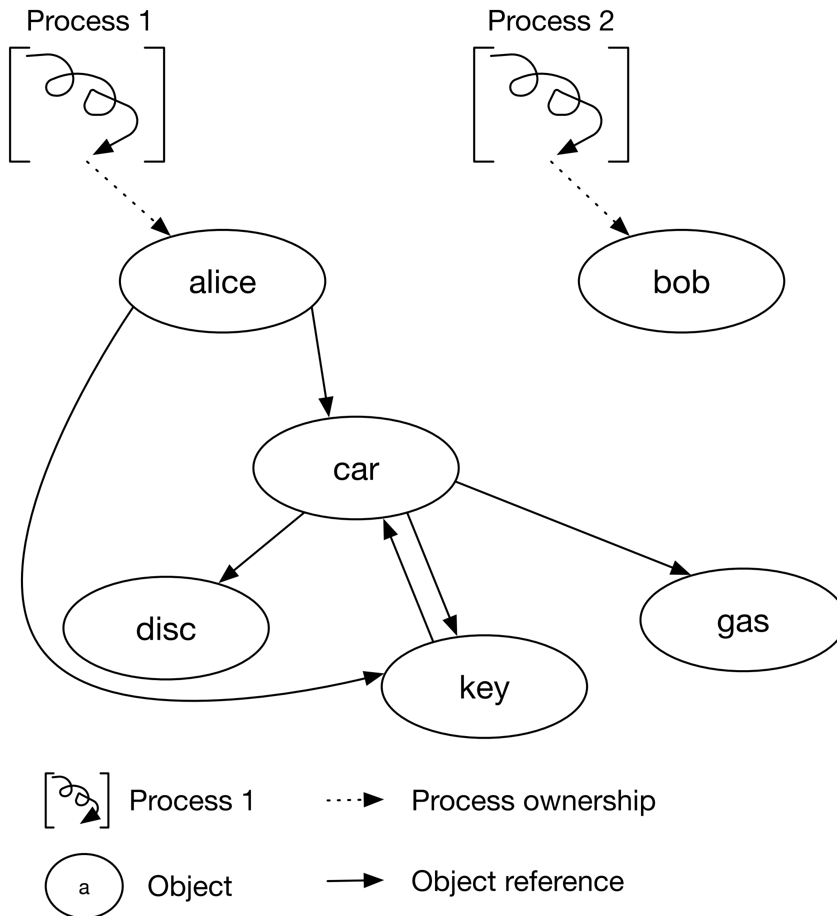

Figure 1.5: Alice communicates the car object to Bob.

Permission Transfer. Unrestricted object reference transfers provide full permissions to the referee on the referred object. To solve this problem we need to control the permissions on shared objects and how those permissions are granted and revoked. As shown in the example above, references give different types of permissions such as read, write, and execution (in the form of message sends). In addition, we need to define a permission model that allows a proper scoping of the sharing.

Object Graph Delimitation. Objects do not exist in isolation but in complex object graphs. When sharing an object, implicit access to its reachable object graph is granted too. We need to control how objects shared between the different graphs behave and how permissions are granted and revoked on an entire object graph. In our example, it would be desirable to grant Bob access permissions to the car without access permissions to the key.

In other words, we need a sharing model preventing shared objects by construction or a model in which we can delimit within an object graph how access is transferred. These models may be left as the developer's pure responsibility or provide (semi-)automatic ways

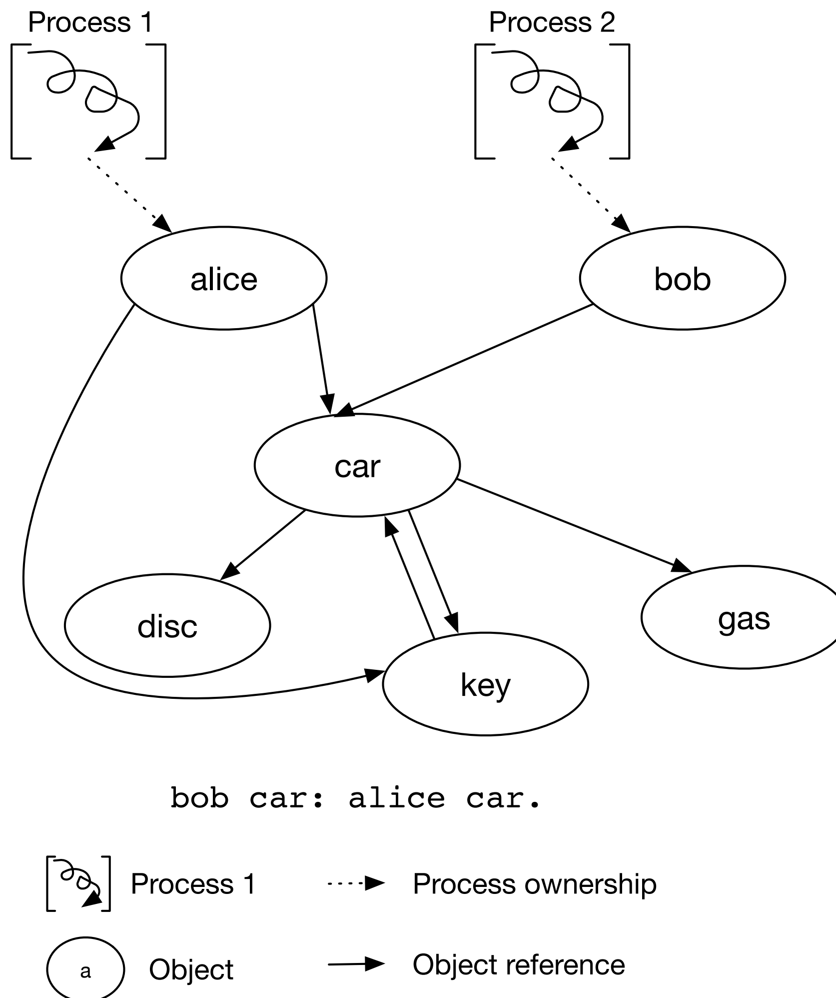

Figure 1.6: Alice communicates the car object to Bob but Alice also has a reference on the key.

to do such a delimitation.

Permission Check. Transferring object (and graph) permissions may incur excessive performance overheads either when the permissions are transferred or when the objects are accessed. For example, solutions that copy the object graph require the costs of allocating and copy memory at the moment of the transfer. Solutions using instrumentation to check object-access will generate performance overheads for each access where the cumulative performance overheads are directly related to the number of object accesses within the program. An optimal solution will minimize both data transfer overheads and data access overheads.

1.5 Research Question

Synchronizing access to an object is challenging for two reasons: (1) planning the synchronization is not sufficient and (2) synchronizing an object also involves synchronizing the graph of reachable references from this object. To address the challenges we face the two following questions:

RQ1: What are the mechanisms that allow one to synchronize object-accesses in an unplanified manner ?

While it exists a lot of synchronization mechanisms (locks, transactions, process calculi, actors...), only locks and transactions allow unplanified synchronization. While allowing unplanified synchronization, they were not designed for it. On the one hand, Locks are fast but it is a tedious task to add synchronization to a not-ready object. On the other hand, transactions are slow but adding synchronization to a not-ready object is easily achieved. A solution that is fast and easy to apply to an unprepared object would be ideal.

RQ2: What are the abstractions that allow managing object-access permissions in an object's graphs ?

Synchronization of objects is different than synchronization of blocks of executions or plain data. Objects are graphs of references that are transitively reachable. To synchronize an object, it is also required to synchronize parts of the subgraph. Recursively synchronizing the whole subgraph reduces the concurrency potential by creating unnecessary synchronizations. A solution that allows determining how the subgraph is accessed would be ideal.

The contributions of this thesis aim to anwser these questions.

1.6 Contributions

The contributions of this thesis are summarized below:

Atomic Samurai for Unplanified Object Synchronization Atomic Smaurai is a model for unplanified synchronization of object-access.

A Classification of Object-Access Control Permissions The identification and classification of the four relevant permissions for object-access.

Canal: Permission Tranfer Channel Framework Canal is a framework offering channels that help change object-access permissions.

1.7 Structure of the Thesis

We organized the thesis as follows:

Chapter 2 `State of the Art Synchronization` describes the different synchronization models encountered in other languages.

Chapter 3 `State of the Art Intercession` focuses on the techniques allowing intercession for unplanified synchronizations.

Chapter 4 `Classification od Object-Access Control Permissions` highlights the identification and classification of the four relavant permissions for object-access.

Chapter 5 `Object-based Right Transfer Channels` delineates the Canal framework that proposes channels to transfer object-access permissions.

Chapter 6 `Evaluation of Permission Transfer CHannels` presents a comparison between the four different channels.

Chapter 7 `Atomic Samurai for Unplanified Synchronization` introduces a model for object-centric unplanified object-access synchronization and its internal mechanisms.

Chapter 8 `Evalution of Atomic Samurai` presents a comparison of Atomic Samurai and semaphores.

Finally, `Conclusion` summarizes and concludes the work presented in this thesis and proposes future work.

Chapter 2

State of the Art: synchronization

In the chapter 1, we identified that synchronization is key to develop concurrent programs. In this chapter, we will review different synchronization models to assess the state of the art. First, we will present the properties desired for a concurrency model. Then, we will study these models to guide us to a model with the desired properties.

2.1 Desired Properties

By assessing the State of the Art, we identified three properties that are valuable in synchronization models: safety, performance, and scalability.

Safety is the guarantee that the model handles race conditions. While all models aim to avoid race conditions, they are subject to misuses in different ways. For example, synchronizing a variable access with a critical section requires that all accesses are actually inside the critical section, otherwise any access outside it circumvents the synchronization.

Performance is the overhead of the synchronization and access operations for non-concurrent executions. Adding synchronization brings an overhead compared to a program that does not require it in two different ways. First, the overhead comes directly from the addition of the synchronization mechanism, for example acquiring a lock, sending an object through a channel, etc. Second, some models ensure invariants at runtime by checking instead of ensuring it from the moment of construction.

Scalability is the overhead of the synchronization and access operations when submitted to multiple concurrent executions. In this case, the overhead sources remain the same but pose new problems such as an execution having to wait for another one to finish its synchronization or invariant checks.

Besides these three properties, we would like to add two other properties as a result of our problematics for a total of 5 properties. These two properties are unanticipation and object-graph access control.

Unanticipation is the ability of the model to add synchronization to an object whose synchronization has not been planned. Some models require to change the program structure or to know all paths of executions to add unplanified synchronization.

Object graph Access-Control is the ability of the model to control and decide the level of access within an object graph. For example, critical sections protect sections of code instead of objects and thus do not take into account object graphs.

2.2 Synchronization models

With all those properties in mind, we will now present a non-exhaustive list of synchronization models.

2.2.1 Locks family

These solutions seek to prevent 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 of waking up or signaling waiting executions that the lock is free.

Safety These solutions are often known to be misused Lee [2006]. The developer needs to locate every data access to add the synchronization mechanism. A lack of one access circumvents the whole mechanism. With multiple locks, lack of progress issues appear such as deadlocks where two executions await the release of their respective acquired locks.

Performance These solutions add an overhead on all data accesses or modifications. It is possible to reduce the overheads by locking only parts of the data (partial locking Michael and Scott [1996], javaConcurrentLinkedQeue) or by centralizing overheads on modification operations that are less frequent and reducing those on access operations that are more frequent.

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

Unanticipation Locating all execution paths leading to an access to the data is required. Then, there is two possibilities. The first possibility is to add a lock for each access located. This will undoubtedly produce lack of progress issues Zöbel [1983]. It is possible to avoid these lack of progress issues by using locking strategies that will require small changes in the program structure. The second possibility is to change the structure of the program directly so that all accesses are unified under one lock. This is a considerable change that significantly reduces the unplanified aspect of the synchronization.

Object Graph Access-Control These solutions emphasize more on controling flows than objects. To be more object-centric, some models tie a lock to an object such as in ADA or the synchronized keyword in java. A class with 'synchronized' methods has its instances with an intrinsic lock that is automatically taken and released. However, these solutions does not take into account the object graph that must be manually protected.

2.2.2 Atomic Operations

These solutions take adavantage of uninterruptible operations called atomic operations. By definition they are not subject to data races. Those operations are uninterruptible because either the processor ensures it or thanks to the scheduler that interrupts processes at known point. Atomic operations are used for building atomic data access Harris [2001], Michael [2004], Newton et al. [2015].

Safety Contrary to locks, these solutions are guaranteed to make progress, they are not subject to livelocks or deadlocks. They eliminate data races but they cannot solve other race conditions such as bad message interleaving or order message violations.

Performance In term of performance, they induce a minimal overhead as they do not introduce a synchronization mechanism. They also scale well as Guerraoui et Al **?** reports in their implementations.

Unanticipation These solutions are not applicable for unplanified synchronizations because they are dependent on the processor or the scheduler. They are unable to synchronize at specific points thus fail to solve bad message interleavings and message order violations.

Oject Graph Access-Control Similarly to locks, those solutions are control flow centric and do not take into account the graph of objects.

2.2.3 Message Passing

These solutions perform synchronizations by passing data through a first in first out (FIFO) data structure called channels for Communicating Sequential Process Hoare [1985], pipe for pipeline or mailboxes for actors Hewitt et al. [1973]. It is heavily used on distributed oriented languages such as Erlang, Go, Scala, Singularity OS Fahndrich et al. [2006]. When one process finishes processing data, it signals that it finishes by adding the data to the queue. Processes wait for their turn to access the data through the queue and consume it. Programmatically, the advantage of these solutions is to facilitate the communication between the threads as a means of synchronization.

Safety These solutions help in orchestrating concurrent processes but do not solve *data races* Fava and Steffen [2020]. They are coupled with an ownership system where ownership is transferred on object transfer via a channel.

Performance Most of these solutions are for distributed computing with separated memory so they rely on copying the data and transferring it to the distributed nodes. Making a copy induces a linear-time overhead in the number of bytes copied. Graphs of objects are often large and in the worst case be the whole program Martinez Peck [2012], thus the overheads are heavy. Some solutions still use shared memory on non-distributed programs. There are two categories, run-time checking approaches and compile-time checking approaches. For run-time cheking approaches, invariants ensuring thread-safety such as immutable or frozen data Leino et al. [2008] are checked at runtime. If the property is broken they return a run-time exception Van Cutsem and Miller [2013]. Checking such invariants produces overheads. For compile-time checking approaches, they rely on type annotations Boyland et al. [2001] and static analysis to guarantee invariants. The runtime has no overhead as it has been pushed into the compilation. Despite the overheads from copy or run-time checking invariants, these solutions scale well.

Unanticipation These solutions have the same problems as locks. Re-using the ownership transfer points offered by the existing FIFO queues requires changes in the structure of this latter to have the proper ownership at the right timing. Adding ownership transfer points by adding queues requires structural modifications that greatly reduce the unanticipation. These solutions do not combine well with unplanified synchronizations. It is required to redesign the program to add ownership transfer points particularly for the object missing synchronization thus locate all accesses to the object.

Object Graph Access-Control On the one hand, run-time checking approaches allow one only to modify the ownerhsip of the object transferred and it is the only one which has its invariants checked. For example, in the case of a frozen array only the array is immutable but all the elements inside are mutable, it is the developer responsibility to froze all the elements inside the array when needed.

On the other hand, compile time approaches allow one to finely express ownership transfer thanks to the type annotations and the checker that point out which objects violate the invariants such as in Pony Clebsch et al. [2015]. The downside is that it imposes a discipline on the developer to produce code in accordance with the permission rules Crichton [2020].

2.2.4 Transactions

These solutions freely modify the data and record every operation in a log Renggli and Nierstrasz [2007], Dice and Shavit [2010]. Eventually consistent data structures Sarnak [1986], Vogels [2008], Shapiro et al. [2011] also keep an history of the data thanks to copies of the object. Those are optimistic solutions in the sense they let the execution run and correct only if necessary. To synchronize they offer two operations: `commit` and `rollback`. A `commit` is the operation that verifies in the log that other transactions did not make concurrent changes to the data. If a concurrent change happens then the transaction

aborts and the rollback operation is in charge to undo all changes logged. Else, the changes are adopted and the execution continues.

Safety To not record all the executions of a program, transactions are only defined on part of the execution. At the begining of a transaction, the recording starts and then ends at the commit. While data races are free to occur, it does not matter as the object will be rollbacked to its state prior to the transaction. Nevertheless, special care must be taken on operations that cannot be undone. For bad message interleavings and message order violations, it is the responsability of the developer to define the transaction such as they do not happen. Small transactions raises the chance of those bugs.

Performance There are two sources of overhead. The first one is to maintain the log of operations. With copies the overhead is similar than the one of message passing solutions using copies. With a dedicated log, it reduces data accesses speed since each access must register itself in the log. The second one is the commit operation where the log is scanned for finding concurrent accesses. The bigger the transaction is, the longer it takes to scan the log and check for concurrent accesses.

Scalability Transactions scale well when there are no conflicting changes in the commit operation as the execution simply proceeds. In case of conflict, it depends on the strategy adopted to solve the conflict. For example, a strategy to retry until the transaction works creates contention points, thus reducing the scalability.

Unanticipation Transactions are not suited for unplanified synchronization. While it only requires to define a transaction on the sensible operations, it will also synchronize all the data inside the transaction that may not be necessary. The transaction must also be defined properly to avoid bad message interleavings and message order violations. Like other models, transactions are not synchronizing object accesses but control flow.

Object Graph Access-Control Since these solutions are more centered on control flow than object they do not take into account the object graph.

2.2.5 Summary

Table 2 shows an evaluation of related work by using the properties. For safety all the solutions allow one to handle data races but Atomic Operations does not allow one to deal with message order violation and bad message interleavings. For performance, only Atomic operations and Message Passing using compile-time approaches to ensure invariant induces almost no overhead. For scalability, Locks and Transactions are subject to contention issues reducing their potential scalability whereas Atomic Operations and Message Passing are known for their scalability. On the contrary, for unanticipation Locks and Transactions require the least change in the program. Since Atomic Operations only allow one to deal with data races, synchronization may be impossible in some cases. Finally, only Message

Table 2.1: Existing solutions and their strong points to make collections concurrent. The more stars, the better. Stars between parenthesis are conditionnal.

Approaches	Locks	Atomic Operations	Message Passing	Transaction
Safety	**	*	*	**
Performance	*	**	*(*)	*
Scalability	*	**	**	*
Unanticipation	**	(*)	*	**
Object-Graph control	*	*	*(*)	*

Passing tied to an ownership system with compile-time cheking invariant allow one to express ownership for an object graph. At best all other solutions manage to be object centric and not control flow based.

2.3 Conclusion

No model offers all the desired properties. It appears that there is a trade off between performance and scalability on one hand and safety and unanticipation on the other hand. Message passing appears to be the best solution for performance and scalability but it is due to compile-time checking invariant approaches nullifying overhead at runtime. However those techniques are not appliable in dynamically-typed languages.

A good solution would still take advantage of the Message Passing scalability while also having Locks or Transactions ability to be safe and unanticipated.

Chapter 3

State of the Art: Intercession

In the previous chapter, we make the difference between runtime-checking approaches and compile-time checking approaches for synchronization. However, compile-time checking approaches are not fit for dynamically-typed languages. In this chapter, we present the properties that we find valuable for our synchronization model. Then, we present the runtime-checking approaches used by the different synchronization models to find the best suited to our constraints.

3.1 Introduction

All the models presented in Chapter **??** aim to ensure invariants that guarantee thread-safety. While compile-time checking approaches ensure invariants without inducing runtime overhead, they are not fit for dynamically-typed languages because informations necessary for their analysis such as borrow checker, variable life cycle, type informations are not statically available. Instead, dynamically-typed languages rely on runtime checking approaches to ensure invariants. It require to intercept messages to check before and after a message send that the invariant is still respected. We identified 4 properties that are valuable in interception mechanisms: uniformity, invariant variety, transparency, and performance.

Uniformity is the ability of the mechanism to intercede for any objects, including special or meta objects such as closures, contexts, classes. If the mechanism does not handle some objects then it cannot ensure invariants on those objects. Transitively the synchronization model cannot ensure thread-safety for those objects.

Invariant variety is the ability of the mechanism to ensure different kinds of invariants. Having a different mechanism for each invariant complexify the overall solution. In the worst case, two mechanisms do not interact well together and cannot ensure the invariants anymore.

Transparency is the capacity of the mechanism to be transparent for the user. If the mechanism is transparent, it becomes more difficult for the user to bypass it. Transitively,

Table 3.1: Categories and descriptions of concurrency bugs.

Properties	Properties Definitions
Uniformity	The mechanism ensure invariants on all objects.
Invariant variety	The proxy is able to chain behavior.
Transparency	The proxy is transparent depending on context.
Stratification	The proxy have clear seperation of responsability.
Performance	The proxy induces low to no-overhead.

it becomes more difficult for the user to bypass the synchronization model.

Performance cost is the overhead induced by the mechanism compared to a program without it. The faster the mechanism is, the faster the synchronization model will be.

3.2 Barriers

A barrier is a low level mechanism that ensure memory access invariants. It is supplied by the CPU, the compiler or the virtual machine for managed languages. A barrier that prevent write operations is a write barrier. In the same manner, a barrier that prevents read operation is a read barrier. Read Only Objects Béra [2016] are objects that cannot be modified thanks to a write barrier. As there is no modification allowed on those objects, they are not subject to race conditions. Basic message passing models coupled with shared memory only allow one to share immediates and read only objects. An object has a flag marker that tells if it is read only. Each write operations checks for this flag and prevents the operations by returning an error if the flag is raised.

Uniformity While barriers works for most object there are few exceptions. Objects related to process scheduling cannot become read-only as the scheduler mutates those objects to manage the execution of the program. In the same way, objects representing an execution, such as contexts cannot, become read-only as they are mutated all the time at least to increase the program counter.

Invariant variety Barriers cannot ensure many invariants but only ensure the invariant for which they were designed. In the case read and write opretaions need to be synchronized, both read and write barrier are required.

Transparency Barriers are not fully transparent as the user needs to handle the error that will be return. The user is also free to activate or deactivate the barrier which will bypass the invariant cheking required by the synchronization model.

Performance cost The downside of barriers is that they slows down every mutating operations Béra et al. [2016] because the flag needs to be checked for every objects. With memory optimizations such as read only objects space, the overhead is limited to read only objects instead of all objects.

3.3 Object swapping

Object swapping is a mechanism that swap the pointers of two objects. After swapping object all variables pointing to the first object point to the second object, and vice-versa Miranda and Béra [2015]. It is a useful mechanism for interception as it allows one to replace an object that we desire to intercept its messages by a trap object that will intercept those messages, thus allow one to dynamically create proxies. It is also useful for ensuring the uniqueness of a reference to an object. Newly created objects are only referenced by their creattor. Swapping an object with a newly created object make the swapped object only referenced by the creator of the newly created object.

Uniformity Object swapping does not works with objects that are used by the virtual machine such as specific objects like the scheduler or specific classes like Array.

Invariant variety Object swapping cannot ensure invariant alone aside from the uniqueness of a reference. By doing so, one is able to create a partial read barrier since only one alias has access to the object.

Transparency Object swapping is completely transparent for the user. The user cannot know and writes the program like the object has not been swapped.

Performance cost Object swapping is a slow operation. Historically the implementation of become has required a scan of all objects, a very slow operation on large heaps Goldberg and Robson [1983]. Currently the implementation requires an extra level of indirection between an object's address and its body, slowing down slot access, and while faster still relatively slow compare to other operations.

3.4 Proxy

A proxy is an object taking the place of another object, the target object. The message sends to the target object are intercepted by the proxy. The proxy is free to add, replace or prevent behaviors. While having many usages such as remote invocations **?**, future objects Yonezawa and Tokoro [1987], the most interesting one for this thesis is invariant checking Van Cutsem and Miller [2010]. Proxies checks invariants by adding invariant checks and preventing the method from executing on the target object if the invariant is not respected.

The proxy design pattern Gamma et al. [1995] is the most well know manner to make proxy but there are a lot of different proxy designs. They are categorized in two group static and dynamic proxies.

Static proxies are alike the design pattern: the target object is proxified during the whole execution of the program. Static proxies only work for the type of object they were designed, they are not uniform. Finally, there is no distinction between the part that intercept message sends, called proxy, and the part that handle the message sends, called handler.

Dynamic proxies are proxies created and attached to an object at run-time.

3.4.1 Java proxies

Since version 1.3, Java offered dynamic proxies **?** for interfaces. Since Java propose uniform dynamic proxies Eugster [2006]. This proxy always forwards the message to the real object. Proxy have non side effect extra behavior (logging). Proxy is fully transparent. Real object escapes in specific situations.

Uniformity Java proxies are able to porixify all objects whose static types is an interface or Object but cannot proxify primitives types or methods.

Invariant variety

Transparency Java proxies are not fully transparent as they reveal themselves when using the == operator. Aside from operators that leak the identity of the proxy the user writes the rest of his program without knowing if the objects is the real one or the proxy.

Performance cost

3.4.2 Javascript Proxies

Uniformity Java script proxies do not support uniform intercession since they are able to proxify objects or functions but cannot proxify primtive values.

Invariant variety

Transparency Javascript proxies were at first revealed by identity checks van Cutsem et al. [2009] but then a VM extension changing the === operator by GetIdentityMethod() method permit to catch this latter method and make the proxy completly transparent to the user. The user writes the program with knowing if the objects is the real one or the proxy.

Performance cost

Table 3.2: Existing solutions for making collections concurrent. More *** the better. Less * the worst. ** In the middle..

Proxy	Uniformity	Stratification	Transparency	Behavior Compo	Perf
Barriers	**	**	**	***	**
Pointer swapping	**	**	**	***	*
Java Proxies	**	**	**	***	*
Javascript Proxies	**	***	**	*	*
Racket Proxies	*	***	***	*	*
Smalltalk proxies	**	*	***	***	*

3.4.3 Racket/Scheme Proxies

Racket proposes two kind of proxies: chaperones and impersonators.

Uniformity

Invariant variety

Transparency

Performance cost Microbenchmarks indicates a factor of 5 to 10 over uninlinded function call. Real applications experience a much lower slow down, around a factor of 2 and correpsond to the difference between Racket original contract system and using chaperones and impersonators for contracts.

3.4.4 Smalltalk Proxies

Uniformity Proxies proposed in Martinez Peck et al. [2011] could not proxify specific object such as blocks and classes. Latter, delegetion proxies Teruel et al. [2013] allow one to proxify even blocks and classes. Both of this proxy implementations rely on object swapping and suffer from the same restrictions, they cannot proxify objects known by the virtual machine such as the scheduler or the class Array.

Invariant variety

Transparency

Performance cost

3.5 Conclusion

Chapter 4

Classification of Object-Access Control Permissions

4.1 Introduction

In this section, we first report on our identification of four relevant permission transfer semantics. Then, each following subsection describes a Pharo implementation of each of these semantics by extending our Channel framework presented in Section 5.3.

4.2 Object-Access Rights

Capabilities as presented by Mark Miller Miller [2006] is an association between an object reference and the access permissions on this object. It can be a proxy or a handle on this association directly exchanged by processes. Capabilities evolve only by restricting further the permissions and not necessarily during a capability transfer. In our model, we exchange direct references and permissions evolve during the transfer.

Object ownership was originally introduced to control the effects of object aliasing in the context of Flexible Alias Protection. It was first embodied as a type system with ownership types Clarke et al. [1998]. Gordon et al. Gordon and Noble [2007] provides ownership for dynamically-typed language for encapsulation. The ownership is by object and forms ownership trees. It encapsulates the object graph but does not handle which process is able to use this object graph. The ownership model proposed is also restrictive, the owner has all permissions while the others have none. Other models exist with more relaxed permission models such as the one proposed by Wernli et all Wernli et al. [2013]. In our model, permission is also more fine-grained granting also write or read permission.

4.3 Identifying Permission Transfer Semantics

A Canal channel transfers references to objects along with permissions to those objects. We distinguish three kinds of permissions: write, read, and execute (sending a message).

Table 4.1: Four permission transfer semantics based on the evolution of the sender and receiver processes' permissions on the transferred object A. The letters W and R represent respectively the write and read permissions of a process on A. Having a '_' instead of a permission means that a process does not have this permission on the object. ∅ means that a process does not hold a reference on the object because it never had it or lost it. A' is a copy of object A.

Permissions Transfer	Sender Process (SP)		Receiver Process (RP)
	Pre-send:	**Post**-send:	**Post**-receive
(1) Copy value (owner process)	$A_{w,r}$	$A_{w,r}$	$A'_{w,r}$
(2) Copy value	$A_{-,r}$	$A_{-,r}$	$A'_{w,r}$
(3) Full transfer (owner process)	$A_{w,r}$	$\emptyset$	$A_{w,r}$
(4) Full transfer	$A_{-,r}$	$A_{-,r}$	$\emptyset$
(5) Exclusive Write (owner process)	$A_{w,r}$	$A_{-,r}$	$A_{w,r}$
(6) Exclusive Write	$A_{-,r}$	$A_{-,r}$	$\emptyset$
(7) Read-only (owner process)	$A_{w,r}$	$A_{w,r}$	$A_{-,r}$
(8) Read-only	$A_{-,r}$	$A_{-,r}$	$A_{-,r}$

A=Object, A'=Object A copy, W = write, R = read, ∅ = no references, - = not permitted.

As we explain in what follows, not all combinations of permissions are meaningful, hence it is not necessary to implement them. For example, a channel where both the receiver and the sender processes lose all permissions would result in the object being unusable. To constrain the field of what is possible, we followed two rules:

Write Implies Read Rule. Write permissions imply read permissions, read permissions imply execution permissions. The first part of this rule means that to write the fields of an object we require the permission to read the fields of that object. The second part of this rule implies that to read the field of an object, we need to be able to send it a message. This last part arises from the fact that object fields (instance variables) are encapsulated in Pharo and can only be accessed by the object itself.

Conservation of Permissions Rule. The set of permissions owned by the sender before the transfer must be equals to the set of permissions owned together by the sender and the receiver after the transfer. A first corollary of this rule is that a process cannot grant a permission that it did not have beforehand thus permissions cannot be forged on an object. A second corollary of this rule is that overall permissions over an object cannot be lost, preventing strange situations where an object reference exists but cannot be accessed by any other object.

Given these two rules we identified four permission transfer semantics (See Table 4.1) in languages based on the evolution of permissions of the sender and receiver processes

before and after the transfer. Since message sending to an object is never restricted in our semantics, we omit the execution permission from the rest of the paper. Note that writing to an object is sending a message but we do not prevent from sending the message and instead throw an error.

Table 4.1 reads as follow. A group of two rows represent a permission transfer semantics. The first row of the group represents a permission transfer when the sender process has the ownership of the transferred object. The second row of the group represents a transfer when the sender process does not have ownership of the transferred object. The first column is the name of the semantics. The second column shows the permissions the sender has before sending an object. The third column shows the permissions the sender has after sending an object. The last column shows the permissions the receiver has after receiving an object.

Taking as example the full transfer semantics represented by the third and forth row. The first column confirms that we are looking at the full transfer semantics.

Reading the third row. In the second column, $A_{w,r}$ means that the sender process will send an object A and has write (ownership) and read permissions on this object. In the third column, $\emptyset$ means that the sender process, after sending A, lost all references on A and all permissions on A. In the last column, $A_{w,r}$ means that the receiver process received a reference on object A and have all permissions on A.

Reading the forth row. In the second column, $A_{-,r}$ means that the sender process will send an object A and has only read permission on this object (no ownership). In the third column, $A_{-,r}$ means that the sender process, after sending A, kept a read-only reference on A. In the last column, $\emptyset$ means that the receiver process never received a reference on object A (in this case because we aborted the transfer).

In the following subsections, we present more in details these four permission transfer semantics: copy value, full ownership, exclusive write and read-only.

4.3.1 Copy Value Graph Transfer (CVGT)

A Copy Value Graph Transfer channel corresponds to the first and second rows of Table 4.1. When sending an object A through this channel, the sender process keeps a reference to A and sends a copy of object A graph to the receiver called A'. The receiver process has all permissions on A'.

While, in a first thought, it seems to break the conservation of permission rule, it does not. The sender process keeps exactly the same permission over object A and the receiver cannot access A (the original object).

During a transfer, the sender process makes the object A' a copy of A Copying object does not keep invariants such as read-only so the sender process has the unique reference on A' with all permissions. After a transfer, the sender process loses this unique reference to A' hence loses all permissions on it. The receiver process gains all permissions A'. This semantics is also found in the solutions of CSP models Hoare [1985] and inspired Go channels **?**.

One way to achieve this behaviour is by implementing a deep-copy of the object graph

reachable from the sent object. To implement this, we redefine the method send: to insert in the channel a deep-copy of the object at send-time.

```
1 CopyValueGraphTransferChannel>>send: anObject
2   | copiedObject |
3   copiedObject := anObject deepCopy.
4   copiedObject graphOwner: nil.
5   super send: copiedObject
```

Listing 4.1: Redefinition of send: for Copy Value Graph Transfer Channel.

We redefine the method receive to set the receiver as the new owner of each object copies inside the object graph, thanks to the method graphOwner:. Thus, the receiver process gains write permission on all objects of the graph.

```
1   CopyValueGraphTransferChannel>>receive
2   | receivedObject |
3   receivedObject := super receive.
4   receivedObject beWritableObject.
5   receivedObject graphOwner: Processor activeProcess.
6   receivedObject beReadOnlyObject.
7   ↑ receivedObject
8
```

Listing 4.2: Redefinition of receive for Copy Value Graph Transfer Channel

A CopyValueGraphTransferChannel guarantees that two reads or two writes cannot happen concurrently on the same object because two separate copies of the graph exist at the same time. Moreover, a deep-copy does not produce shared objects but this channel suffers the duplication problem: we can modify the two copies independently.

4.3.2 Full Ownership Graph Transfer (FOGT)

A Full Ownership Graph Transfer channel corresponds to the third and forth rows of table 4.1. The thirs row represents the case when the sender process has ownership of object A. After a transfer, the sender process loses all references on A thus all permissions are represented by $\emptyset$. The receiver process gains all permissions.

This behaviour respects the conversation of permissions rule since the sender permissions become the receiver permissions. If the sender process does not own object A as in the forth row then the channel throws an error and the transfer does not happen. The receiver has no references on object A. This behaviour also complies with the conservation of permissions rule since the permission did not change. This semantics is also found in the solutions of Rust channel Narayanan et al. [2020] or Kilim Srinivasan [2010].

This behaviour is achieved by revoking recursively all references in the object graph. Listing 4.3 shows the Pharo code of the redefined send: method for this channel. We implemented this channel using Pharo's atomic object reference swapping (*i.e.,* become: is used in the graphBecome: method). Using pointer-swapping, all original references to the

sent object are replaced by references to the argument object. After pointer-swapping, the channel object is the only one that has a reference to the object to send.

```
1 FullOwnershipObjectTransferChannel>>send: anObject
2   | objectToSend |
3   "Create placeholer object"
4   Processor activeProcess = anObjectOwner
5     ifTrue:[ anObject graphOwner: nil ]
6     ifFalse: [ self error: 'Cannot full transfer
7   an object not owned' ].
8   objectToSend := Object new.
9   "Swap references"
10  anObject graphBecome: objectToSend.
11  "At this point, objectToSend has
12  the sole reference to the sent object"
13  queue nextPut: objectToSend
```

Listing 4.3: Redefinition of send: for Full Ownership Object Transfer Channel.

Later on, when a process calls receive and consumes the reference from the channel, it will get the unique reference to that object. Moreover, the new owner of the object graph is assigned to the receiver process as shown by the redefinition of the receive method in Listing 4.4.

```
1   CopyValueGraphTransferChannel>>receive
2   | receivedObject |
3   receivedObject owner ifNil:["gain ownership"
4     receivedObject := super receive.
5     receivedObject beWritableObject.
6     receivedObject graphOwner: Processor activeProcess.
7     receivedObject beReadOnlyObject.
8   ]
9   ↑ receivedObject
10
```

Listing 4.4: Redefinition of receive for Full Ownership Object Transfer Channel.

4.3.3 Exclusive Write Object Transfer (EWOT)

An Exclusive Write Object Transfer channel corresponds to the fifth and sixth rows of Table 4.1. In the sixth row the sender process starts with all permissions on object A. During the transfer, the sender process loses the write permission but keeps at least one reference on object A. The receiver process gains all permissions over object A. The receiver process ends up being the only one with write permissions.

This behaviour complies with the conservation of permission rules because the permissions of the sender before the transfer are the permissions of the receiver after the transfer.

Table 4.2: Summary of the Permission Transfer Channel's Properties.

Property	CVGT	FOGT	EWOT	ROOT
Transfered permissions	Full	Copy	Exclusive Write	Read-only
Granularity	Graph	Graph	Object	Object
Sender Read	Inconsistent Reads	Revoked	Allowed	Allowed
Sender Write	Inconsistent Writes	Revoked	Revoked	Allowed

If a process does not possess the write permission on object A as in the sisth row then the channel throws an error and the transfer does not happen. It allows us to comply with the conservation of permissions rule. One way to achieve this behaviour is to instrument all writes to object fields and check if the writing is being done from the owner process. This semantics is also found in Haskell or Clojure channel implementation with persistent data Sarnak [1986].

Our current implementation makes use of pre-existing per-object low-overhead write barriers Béra [2016] in Pharo.

Listing 4.5 shows the code of the send: method for the EWOT Channel. Before adding the transferred object into the channel queue, its owner is reset (set to nil) thus preventing any further write access by the sender.

```
1 ExclusiveWriteObjectTransferChannel>>send: anObject
2   Processor activeProcess = anObject owner
3     ifTrue: [ anObject owner: nil.
4       queue nextPut: objectToSend ]
5     ifFalse: [ self error: 'Trying to send
6 a not owned object' ]
```

Listing 4.5: Redefinition of send: for Exclusive Write Object Transfer Channel.

In its receive method (See Listing 4.6), the channel sets the owner of the object to the current process before returning it.

```
1 ExclusiveWriteObjectTransferChannel>>receive
2   | receivedObject |
3   receivedObject := super receive.
4   receivedObject beWritableObject.
5   receivedObject owner: Processor activeProcess.
6   receivedObject beReadOnlyObject.
7   ↑ receivedObject
```

Listing 4.6: Redefinition of receive for Exclusive Object Transfer Channel.

It is important to note that thanks to the Pharo's concurrency model (See Section 5.2.1), writes are atomic thus a read cannot occur while a process is writing on the shared object such as modifying its owner. Also note, the write permission granting is on a per object

basis and not directly the whole object graph. It allows one to manually delimit the granting of the write permission on the object graph.

4.3.4 Read-only Object Transfer (ROOT)

A Read-Only Object Transfer channel corresponds to the penultimate and ultimate rows of Table 4.1. In both rows, the sender process is keeping the same permissions it had over object A. The receiver process gains a reference on object A and has only the read permission.

This behaviour complies with the conservation of permissions rules because the sender process does not change and the receiver process has only the read permission.

We implemented it with the same write barrier mechanism used in the exclusive write object transfer (EWOT) except that the object ownership remains unmodified. Since the object's owner does not change the receiver process is only able to read the object. Note that the sender may or may not have write permissions on the object.

In Listing 4.7, a person object is created and its owner is manually set to nil. This removes the write permission of the sender process on this object. Nevertheless, the sender is still able to send the object through a read-only object transfer channel. In this example, the receiver process does not gain the write permission but only the read permission on the object.

```
1 channel := ReadOnlyObjectTransferChannel new.
2 objectToTransfer := Person new.
3
4 "Change the ownership"
5 objectToTransfer owner: nil.
6 objectToTransfer name: 'Alice'. "Raise an exception"
7
8 "receiver process"
9 [ objectReceived := channel receive.
10   objectReceived name: 'Bob'. "Raise an exception"
11 ] fork.
12
13 channel send: objectToTransfer.
```

Listing 4.7: Usage Example of a Read-Only Object Transfer Channel.

4.3.5 Channel Limits: Transactionality and Inconsistent Reads

Table 4.2 summarizes the different semantics and characteristics of all channel semantics.

The EWOT channel granting the read permission to other processes induces inconsistent read. Back to the car example, let's say bob owns the car. Alice reads the title of the disc and process it. Now, Bob changes the disc and Alice reads the number of track of the disc. Alice will read the number of track of the new disc.

Inconsistent reads also occurs with the CVGT channel. Alice gives the car to Bob expecting that Bob does an action with the car. Since Bob has a copy, Alice cannot see the action effect. A new synchronization is necessary to avoid inconsistent reads. A callback re-transferring the copied object back, or a merging approach is then necessary for the sender process to access the modified object once the receiver is done.

We believe this issue is proper to transactional systems, and is orthogonal to the permission transfer that channels allow.

4.4 Object-Access Control

4.5 Conclusion

Chapter 5

Object-based Right Transfer Channels

5.1 Introduction

5.2 Efficient and Correct Object Graph Transfer in Dynamically-Typed Languages

5.2.1 Context: Pharo's Concurrency Model

The Pharo programming language implements concurrency with so-called *processes*: lightweight green-threads scheduled by the virtual machine. The process scheduler schedules processes given their 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 Ducasse and Polito [2020]. Process switches happen on a timely basis but only at safe execution points: message sends and back jumps.

5.3 Canal: An Extensible Channel Framework

In this section, we present an overview of our channel framework to experiment with different permission transfer semantics. We decided to use channels as a permission transfer mechanism because they allow a clear delimitation between the sender and the receiver processes while making object sharing explicit. Processes that receive objects from a channel gain some permissions on those objects and processes that sent them may lose some permissions on them. We also describe our per-process ownership model to control write permission on shared objects and thus prevent data races.

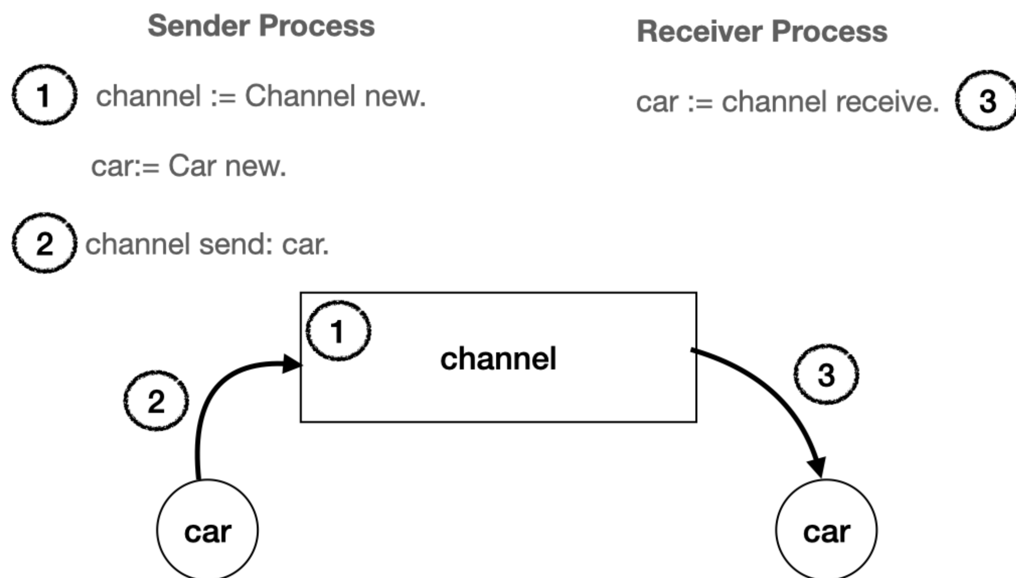

Figure 5.1: Overview of Object Transfer Through a Channel.

5.3.1 Extensible Channels and Hooks

Figure 5.1 depicts the general view of using a channel to transfer an object (the car object on the figure) between two processes. We distinguish two kind of roles a process can take regarding a channel: the sender process and the receiver process. A sender process sends object references into a channel when it does not use this object anymore or wants to share it with other processes. A receiver process acquires references to objects to process them by receiving a reference from a channel.

Channel overview. A channel is a shared data structure between processes that allows one to exchange references. Our channels are first class objects. Channels are unidirectional and can be shared between multiple senders and multiple receivers. Channels are in shared memory, any process having a reference of a channel is able to use it. Channel is the base class of the Channel hierarchy. To guarantee atomicity they are implemented using thread-safe atomic FIFO queues that allow one to transfer any type of objects. The public API is minimalistic with only new, send: and receive: messages. The send: and receive: messages are the ones responsible for the permission transfer and are the hooks to define tailored channel subclasses. The general API is composed of three main messages:

Channel creation. Creating a channel consists only in sending the new message to a specific Channel subclass. It is an extension point for specific initialisations.

Table 5.1: Channels permission transfer semantics offered in other languages.

Permissions Transfer	CVGT	FOGT	EWOT	ROOT
Rust	X	✓	X	X
Kilim	X	✓	X	X
Erlang	X	X	X	✓
Go	✓	X	X	X
C++	✓	X	X	X
Java	✓	X	X	X
Javascript	✓	X	X	X
Kotlin	✓	X	X	X
Lua	✓	X	X	X
Clojure	X	X	✓	X
Haskell	✓	X	✓	X
Pony	✓	✓	✓	✓

✓= the language offers a channel with the semantics, X = the language does not offer a channel with this semantics

Channel send. Sending an object consists in sending the message send: with the desired object as argument. To define channels with specific semantics, the send: message is redefined. The send operation is non-blocking. First specific policies are applied to the object such as revoking write permission then the object is enqueued in the channel.

Channel receive. Receiving an object from a channel consists in sending the receive message to a channel. This message is blocking for the receiver process in the case the channel queue is empty. We chose to make them blocking in this case because when sending a receive message a process expects an object to be returned. Returning nil or an unexpected object only defers possible cause of bugs. When an object is dequeued, its permission is updated according to the receiver process.

5.3.2 Channel Transfer by Example

The Channel abstract class is the base class of our framework. It implements the exchange of object references using a unique atomic thread-safe queue. Listing 5.1 shows the Pharo code of the send: and receive methods of this class. Those methods need to be redefined to add the permission transfer.

5.3.3 In other languages

The notion of permission is not explicit with those queues but their semantics is comparable to our Copy Value Graph Transfer channel.

5.3.4 Shared Memory

Run-time checking approaches. Older languages mostly rely on run-time checking approaches. Many models exist such as the thread/mutex model **?** and the Software Transactional Memory(STM) Shavit and Touitou [1995] model.

Checking during execution induces an overhead specifically on frequently accessed data. The STM model with the use of persistent data aims to reduce the number of checks. A persistent data structure Sarnak [1986] is a data structure that preserves one or multiple previous versions of itself when it is modified. Here one process writes on the to-be-modified version meanwhile other processes read on a preserved version. The check is delayed when the modified version needs to become the preserved version. Instead of having many little checks during execution, there is only one big check. Processes accessing the data only to read are then not penalized. With all those solutions an overhead still exists at least when writing onto the data but the data transfer cost is close to non-existant. The semantics is the one of our Exclusive Write Object Transfer channel in the fact that only one process has the write permission but has the side-effect of the Copy Value object Graph Transfer channel where inconsistent reads happen.

Compile-time checking approaches. The idea for type annotation in the objective of sharing Boyland et al. [2001] data has only be demonstrated in some recent languages such as Rust, Pony and Project Midori. To synchronize between processes, Rust offers a CSP with channels but with shared memory Narayanan et al. [2020]. It guarantees the uniqueness of a reference to a datum with a static analysis during compilation with a borrow-checker. The owner of this unique reference is simply the owner of the data. Rust channels are Full Ownership Object Transfer channels. Pony offers an actor model, it is one of the few actor model with shared memory. Pony Clebsch et al. [2015] guarantees the uniqueness of the writer with a static analysis during compilation with type annotations. Contrary to Rust, it is possible to have multiple references to a datum but with different capabilities. If there is already a reference with the write capabilities all further reference will not have this capability for the lifetime of the first reference with the write capability. Pony type annotations allow one to express the same transfer semantics than our channels. While compile-time checking does not suffer from overhead or bigger memory usage at run time, it imposes a discipline on the developer to produce code in accordance with the permission rules Crichton [2020]. However this technique is possible for statically typed language, it is not an easy feat for dynamically typed language where the control flow graph depends on the type of the receiver. They are a starting point to enhance the performance of our channels.

Table **??** summarizes the semantics tied to object transfer through channels in other languages. This list of languages is not exhaustive. Some languages are not represented because we could not determine exactly in which category they belong such as C# and Ruby. Rust and Kilim both offer FOGT semantics thanks to compile-time checks. Even though Erlang effectively copy messages, they only allow the sharing of immutable data thus having the same semantics as ROOT. Other languages that we did not list (notably

functional ones) take this approach. Most of the languages with CVGT semantics follow the Go trend. They deep-copy the data to send. Note that the notion of pointer exists in some of those languages and sending a pointer is not restricted causing data races. Clojure and Haskell propose channels coupled with STM or persistent data that allow one writer and many readers. This is the EWOT semantics. Finally, Pony with its type system allows for a fine grain of permission transfer and offers each of the semantics.

```
1 Channel>>send: anObject
2   queue nextPut: anObject
3
4 Channel>>receive
5   | result |
6   [ | keepWaiting |
7     keepWaiting := false.
8     self isClosed
9       ifTrue: [ChannelClosedException signal].
10    result := queue
11      nextIfNone: [ keepWaiting := true ].
12    keepWaiting ] whileTrue: [ queue waitForNewItems ].
13  ↑ result
```

Listing 5.1: Definition of send: and receive methods of the Channel base class.

Listing 5.2 shows a Ping Pong example where two processes exchange a ping and a pong object through two channels. The sender process first creates a new channel (line 1) to send a Ping object and another channel (line 2) to receive a Pong object. A receiver process is created using the fork message (line 6) sent to a block (lexical closure syntactically delimited by square brackets). This receiver process waits until it receives an object from the channel (line 7) and then sends a Pong object into the channel (line 6). The sender process sends a Ping object (line 8) and then waits until it receives Pong object (line 10).

```
1  pingChannel := ExampleChannel new.
2  pongChannel := ExampleChannel new.
3  "receiver process"
4
5  [ objectReceived := pingChannel receive.
6  pongChannel send: Pong new ] fork.
7
8  pingChannel send: Ping new.
9
10 pongChannel receive.
```

Listing 5.2: Usage Example of a Channel.

This example uses a Channel subclass that does not redefine send: and receive methods but it would be mostly unchanged using more specialized channels. In the following section, we will extend this minimal model to build specific channels by subclassing the

Channel class. By carefully choosing specialized channels, the developer prevents data races on the transferred objects.

5.3.5 Per-Process Ownership Model

To avoid data races, concurrency models typically impose a unique writer process at any time for a single object Fava and Steffen [2020]. Ownership models, using message passing, achieve this by attaching a unique owner to all objects. These models may be too restrictive because they prevent non-owner processes to access an object.

In our framework, each object has a unique owner process stored in its instance variable named owner. An object's owner is the only process that has the write permission on this object. Initially, the process that creates an object is its owner. Changing the ownership of an object only requires assigning another process in its owner instance variable. An attempt to write to an object from a process that doesn't own an object results in an error. Our ownership model allows multiple read-only references on an object while still guaranteeing the uniqueness of the writer. The process scheduler of Pharo ensures that a read operation does not happen during a write operation.

In the following section we will show how our framework models permission transfer at the level of channels.

5.4 Permission Transfer Channels

5.4.1 Copy Value Graph Transfer (CVGT)

5.4.2 Full Ownership Graph Transfer (FOGT)

5.4.3 Exclusive Write Object Transfer (EWOT)

5.4.4 Read-only Object Transfer (ROOT)

5.5 Conclusion

Chapter 6

Evaluation of Permission Transfer Channels

6.1 Introduction

6.2 Comparing the Different Channels

Concurrency mechanisms target first correctness to avoid inconsistencies and then performance. Most of the time, implementations are a trade-off between correctness and performance Henzinger et al. [2013]. In this section, we compare the performance of our different channels. In our case, solutions using copy or pointer-swapping suffer an overhead during the object transfer via a channel meanwhile solutions based on the write barrier do not. In contrast, solutions based on the write barrier suffer from overhead on object-access meanwhile the others do not.

In this section, we report on our results benchmarking different scenarios. The Pharo bench message measures the number of times a message is sent per second. The time taken to send a message is inversely proportional to the result of the bench message. In other words, the higher the result of the bench message is, the faster it is. Each channel of each scenario is benched 100 times. A box summarizes 100 benchmarks on a channel. The first and third quartile form the box, the lowest and maximum value form the whiskers. We run all measurements 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.

6.2.1 Scenario 1: Single Object Transfer Speed

In this scenario, we measure the cost of transferring only one object with our different channels. To achieve this, we reuse a modified version of our Pong example (See Listing 5.2) with the different channels. The transferred object has 3 instance variables: its owner process, a name and a potential collection of friends not initialized for this scenario.

```
1 channel := OwnershipGraphTransferPartialReadBarrier
2 Channel new.
```

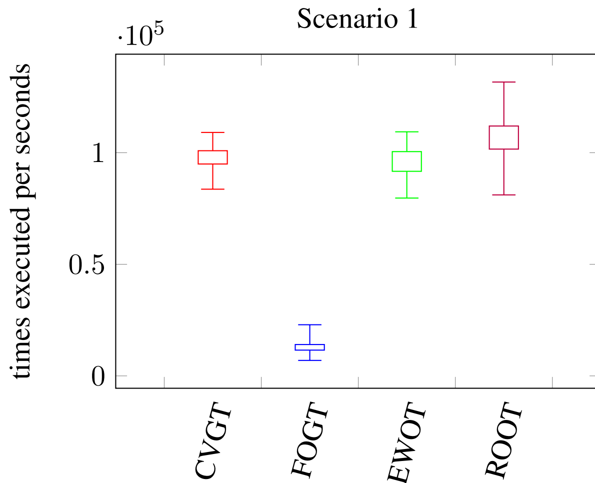

Figure 6.1: Result of data transfer speed for simple objects. The more times per second it is executed the fastest it is.
CVGT = Copy Value Graph Transfer (red).
FOGT = Full Ownership Graph Transfer (blue).
EWOT = Exclusive Write Object Transfer (green).
ROOT = Read-Only Object Transfer (purple).

```
3 objectToTransfer := OwnedPerson new name: 'Alice'.
4 [objectReceived := channel receive.
5   channel send: objectReceived
6   ] fork.
7 channel send: biggerObjectToTransfer.
8 channel receive
```

Listing 6.1: Code example for benchmarks.

Figure 9.1 shows that Copy Value Graph Transfer channel (red) is on par with the Exclusive Write Object Transfer channel (green). Copying a small object is almost as fast as sending a reference through a channel. Both are around 10% slower than the Read-Only Object Transfer channel (purple). The ROOT channel does not transfer ownership so it does not have to update the ownership status and does not need a graph traversal. It explains the better performance of this channel in transfer speed. The Full Ownership Graph Transfer channel is 8 times slower than the other ones. Pointer-swapping is slower than a field update for ownership transfer and also slower than copying small objects. The conclusion is that except for the Full Ownership Graph Transfer channel implementation using pointer-swapping, they are all in the same order.

6.2.2 Scenario 2: Object Graph Transfer Speed

In this scenario, we measure the cost of transferring an object but for different size of object graphs. The code is similar to scenario 6.2.1 but the transferred object now references a

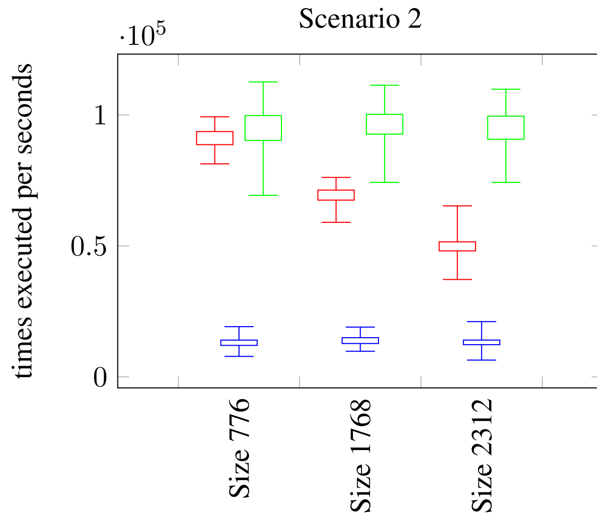

Figure 6.2: Data transfer speed result for object graphs of different sizes. It is on number of times executed per seconds, the higher the value is, the fastest the implmentation is.
CVGT = Copy Value Graph Transfer (red).
FOGT = Full Ownership Graph Transfer (blue).
EWOT = Exclusive Write Object Transfer (green).

list of friends in its variable objectToTransfer. In this scenario, we use three different sizes for the friends list. The EWOT channel and ROOT channel operate the transfer at an object granularity and not on a graph granularity. To be fair in the comparison, we adapted the EWOT channel to a graph granularity *i.e.,* we recursively apply a write barrier to all objects in the transferred graph. We omitted the ROOT channel in this comparison because transferring an object reference already gives access to the graph. There is no modification on this channel, hence the result is the one from the previous scenario 6.2.1.

Figure 6.2 shows that the Copy Value Graph Transfer channel linear-time in the size of the object graph. It is already 2 times slower by adding two friends in the object graph compared to no friends. It becomes slower than the Full Ownership Graph Transfer after adding five friends. The Full Ownership Object Graph Transfer based on the become: message and the adapted Exclusive Write Transfer channel does not vary much for this sample. Since they all perform the same graph traversal, we conclude that the creation of new objects is expensive. For the FOGT channel, it is quite surprising to repeat a seemingly costly operation and to not degrade performance. An alternative that we did not explore, is pointer-swapping all the objects in the object graph at once. Indeed, the become operation is based on a primitive that performs the pointer-swapping from elements inside an array to elements from another array. In the become's case, both arrays contain one element each, the two objects to swap. The alternative solution is then to collect all the object graphs inside an array and to swap with an array of filler objects by using directly the primitive. An inspection of the implementation of this primitive is necessary to potentially understand our result.

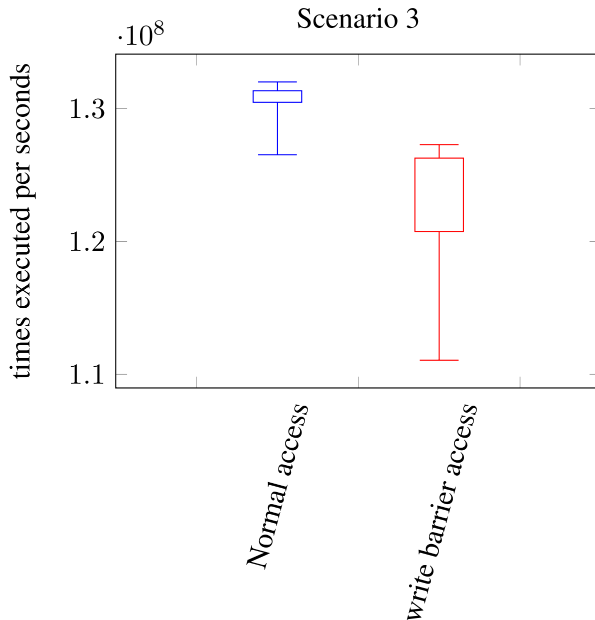

Figure 6.3: In blue, the non-instrumented accesses. In red, the accesses instrumented with the write barrier

6.2.3 Scenario 3: Single object-access Speed

In this scenario we measure the cost of accessing an object with the write barrier compared to accessing without it. Channels using copy or become are not penalized on data access, thus measuring accessing without the barrier is equivalent.

Figure 6.3 shows that accessing an object field with the write barrier is in average 6% slower than a regular access. For the transfer of an object graph of size 1768, it needs 7000 accesses to the object to have a bigger overhead than doing a copy of the object. For an object graph of size 2312, it requires more than 18000 accesses to the object to have a bigger overhead than doing a copy of the object. In conclusion, channels based on copy lose performance on data transfer depending on the size of the object graph to transfer. Those channels are more suitable for programs heavily accessing objects and doing few object transfers. Channel based on a write barrier lose performance on data access. Those channels are more suitable for programs exchanging a lot of objects and performing few accesses. Finally, channels based on a partial-read barrier mechanism are more appropriate for programs both transferring and accessing a lot of objects.

6.2.4 Discussion

Mechanism comparison in languages. Other languages having potentially more efficient write barriers do not change the conclusion brought by our results. Those write barriers will still introduce an overhead on object-access. It will only change at what point accessing object becomes more expensive than copying or vice versa. In the same way,

others language having potential better copying algorithms will still introduce an overhead on object transfer.

Partial-read barrier in the form of the become message does not relate as much in other languages. The logic behind the become message is hidden in the virtual machine supporting the Pharo language and an inspection of this latter could give us a better understanding.

Memory usage. We did not measure the memory consumption induced by our different channels. Nevertheless, our write barrier is implemented by marking and checking an unused bit in the header of the objects. Therefore, the memory size of the objects is not affected at all. In contrast, the partial-read barrier leaves a placeholder object and copying duplicates the object, which both increase memory usage. Some techniques, such as persistent data, diminish the number of copies but do not completely eliminate them. In constrained memory environments still offering concurrency, a programmer should opt for channels using the write barrier.

6.3 Classification

6.4 Conclusion

Chapter 7

Atomic Samurai for Unanticipated Object synchronization

7.1 Introduction

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

7.2 Solution Overview

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

Dynamic intercession In dynamically typed languages, every message send is late bound which allows dynamic intercession. It is possible to intercept a message and modify its behavior. For example it allows to add logging, forwarding or even modifying the message send. Atomic Samurai uses object proxification for message interception. The proxy acts as a barrier to determine if a process is allowed to manipulate its object.

Complete intercession One of the challenges that Atomic Samurai has to tackle is complete intercession. In some dynamically typed languages, instance variable accesses are statically bound. 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 synchronization, another process is free to access an instance variable. The synchronization must be added at safe points. One solution is to verify that

the synchronization is added in a safe point. That is the current solution used by Atomic Samurai. Atomic Samurai first checks that an object is used by only one process. If it is the case, Atomic Samurai retries to proxify the object later. An alternative solution is to enforce that instance variable accesses are message sends in the language, *i.e.,* instance variable are access 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 synchronization. We measure this trade off in section 9.4.

Super message sends are also statically bound. We solved the case of super message sends in a specific way for our language that we discussed in section 8.4.

Unicity of Reference Another challenge that Atomic Samurai has to tackle is to ensure the unicity of object reference. We enforce the invariant that the proxy is the unique owner of a reference to the synchronized object at all time. Newly created object are only referenced by their creator. Atomic Samurai uses pointer swapping between a newly created proxy and the synchronized object. After swapping pointer between the proxy and the object, all references to the object now points to the proxy and the reference previously to the proxy now points to the object. The proxy stores the unique reference to the object. Afterward to ensure that a message send does not leak a reference of the synchronized object, we use delegation proxies Wernli et al. [2014]. Delegation proxies are reentrant proxies where the messages dispatched from the delegation proxy will pass through itself.

Our Atomic Samurai library offers two messages.

take The receiver of this message becomes exclusive to the process that executes it. To acquire exclusive access to the object, the taker process wait that no other process is actively using the object. The take operation 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 the case, the object has been taken many times it needs as many release messages to release properly the object.

7.3 Atomic Samurai by Example

Figure 7.1 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 `register take` message to gain exclusive access on the register.

The second part of the figure shows the state after the take operation occured. The take operation checked process2 stack if the register is already in use which is not the case. It proceeds by creating a proxy (trap + handler) and swaps the reference to the proxy with

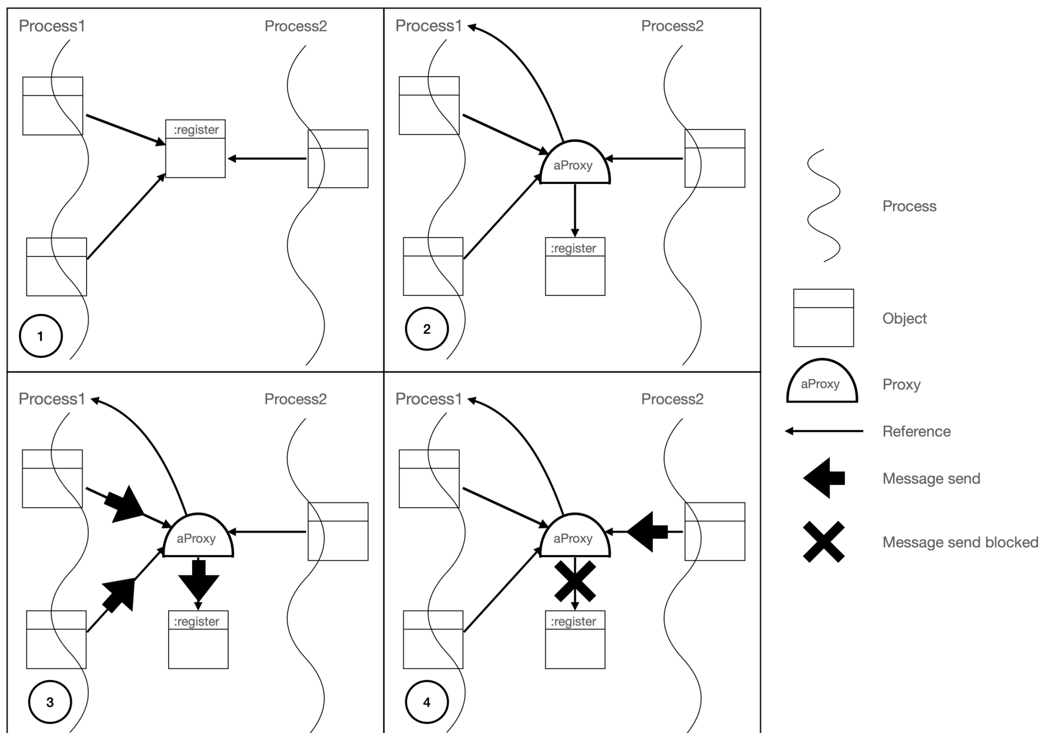

Figure 7.1: Example of Atomic Samurai with two processes.
1. Initial state.
2. State after take message, object is proxified.
3. Taker process sends a message, the proxy forwards it.
4. Non taker process send a message, the proxy blocks it.

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 forth part of the figure shows that the proxy acts as a barrier and blocks the message sends fron 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 potiental bad message interleaving because the second whereIs message will be blocked until the first process releases the registry. Take and release act as synchronization on the registry.

7.4 Conlusion

Chapter 8

Implemention of Atomic Samurai in Pharo

8.1 Introduction

In this chapter, 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.

8.2 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 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 Ducasse and Polito [2020]. Process switches happen on a timely basis but only at safe execution points: message sends and back jumps. The Pharo VM does not support parellelism either but it allows a broaden concurrency.

8.3 Take implementation

We present the code of the take message in Listing 8.1. 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. Elsewise, 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
13
```

Listing 8.1: Take implementation

Back to the example, we wrap the previous code Listing with registry take and registry release like in Listing 8.2.

```
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 8.2: Example of code wrapping using Atomic Samurai

8.4 Super Message Sends

Dynamically swapping two objects at runtime, with primitive become: for example, produces execution bugs with super message sends such as a method wrongly applied to an object.

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

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

Listing 8.3: Example of method using the proxy and a self message send

The setter method from Listing 8.3 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.

Now let's modify this method like in Listing 8.4. Instead of sending alternativeSetter: anobject to self, it sends it to super.

```
1
2   setter: anObject
3
4   self proxify.
5   super setter: anObject.
6   self unproxify.
7
```

Listing 8.4: 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 invokes 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 sends does not involve the receiver. A solution that we implemented in the virtual machine of Pharo is to modify the super message send look up 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 look up starts. If the receiver is not valid, the virtual machine creates an exception that the proxy catches and uses for interception.

8.5 Special Objects Protection

Our prototype has some limitations. The take message is redefined for process and rises an exception in Atomic Samurai. It is not possible to take a process because Atomic Samurai compares processes identity and taking a process disable 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 latters cases.

8.6 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.

8.7 Conclusion

Chapter 9

Evaluation of Atomic Samurai

9.1 Introduction

In this chapter, we show the evaluations of our solution. We show how it solves problems from the literrature and a real world example from a concurrency bug in Pharo.

9.2 Examples from literrature

Lopez and Al Lopez et al. [2018] provides a survey and a classification of all possible concurrency bugs for the Actor model. It also gives pointers to similar bugs in the thread model such as Artho et al. [2003]. We reproduced all bugs in the race condition category with failing tests highlighting 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 on the caller side.

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

Bad message interleavings Bad message interleavings solutions all consists in having larger section that cover the messages that should not interleave. With Atomic Samurai, the difficult part is identifying which messages produce a bad interleaving. The fact that those examples comes from the literrature helps in knowing which operation produce such interleavings.

9.3 Real world example in 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 such as in Listing 9.1.

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

Listing 9.1: 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 font were still passing.

9.4 Performance

In this Section, we measure the overhead induced by our solution for a non concurrent execution. We first present the measurement methodology. Then we present the two sources of overhead: synchronization installation overhead and proxy interception overhead. We also compare the scalability of our solution to a semaphore base solution.

Methodology The Pharo bench message is a microbenchmark that measures the number of times a message is sent per second. The time taken to send a message is inversely proportional to the result of the bench message. Each violin plots summarizes 100 microbenchmarks for the given scenario. The violin plots shows the same information as a box plots (average, quartile ...) but has the advantage to better show the data distribution. We run all measurements 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.

synchronization installation overhead. The first source of overhead comes from adding the synchronization. Figure 9.1 shows the overhead induced by adding the synchronization. For reference, the middle violin plot is the overhead induced by the default synchronization mechanism, a semaphore. It takes around 80 nanoseconds to signal or wait with a semaphore.

In Atomic Samurai, synchronization is handled by the take and release messages. As seen in the implementation Section 8.3, the take message brings the more overhead. It first checks that the object is not already in use and then proxyfies the object. On the right, we measure the overhead with the solution that does a stack traversal to insure the safety of the take operation. The violin plot shows that it takes around 16 milliseconds to execute our

take implementation. Atomic Samurai is five order of magnitude slower than a semaphore but it is possible to have a faster take implementation.

As discussed, in Chapter 7, the check is not necessary if there is no statically bound elements such as direct accesses. On the left, we measure the same code with a take implementation that only proxify the object therefore that has no check. The violin plot shows that it takes only around 75 microseconds to execute this take implementation. While this implementation is significantly faster, it is still 3 order of magnitude slower than a semaphore. There is also the tradeoff that every access goes through an accessor. We mesure this overhead alongside the second source of overhead, the proxy interception.

Proxy interception. Intercepting a message with a proxy is more expensive than a direct message send to the object. Figure9.2 demonstrates this. As baseline, we measure the time it takes to access message an object through its variable. The middle violin plot shows that it takes around 9 nanoseconds. The left violin plot shows that it takes around 18 nanoseconds to access the object by sending the accessor message. Then, we measure the same accessor message send that is intercepted and executed by a proxy. The right violin plot shows that it takes between 300 to 700 microseconds. It takes more than $1.7 * 10^6$ accesses through accessors to take more time than doing a stack traversal on an execution. Also, the overhead to access the object trough the accessor is negligeable compared to the time it takes going through the proxy.

To conclude, on object-accessed heavily it is preferable to have a slower synchronization installation than using accessor all the time. Our prototype also has room for improvement. Pointer swapping is a slow operation and a language with a faster pointer swapping primitive will not have as much overhead on synchronization installation. Similarly, a language with a faster proxy library will not have as much overhead on proxy interception.

9.5 Scalability

In this Section, we measure the overhead induced by our solution in a concurrent environment. We compare our solution to a similar code synchronized with semaphores which are the default synchronization mechanism available in Pharo. We started with 2 concurrent executions and increment by one execution at a time. Figure 9.3 shows the overhead depending on the number of processes with the semaphore implementation. Figure 9.4 shows the overhead depending on the number of processes with Atomic Samurai. The two Figures shows that Atomic Samurai is more expensive than a synchronization with semaphores but they both scales quite linearly with the number of processes.

9.6 Conclusion

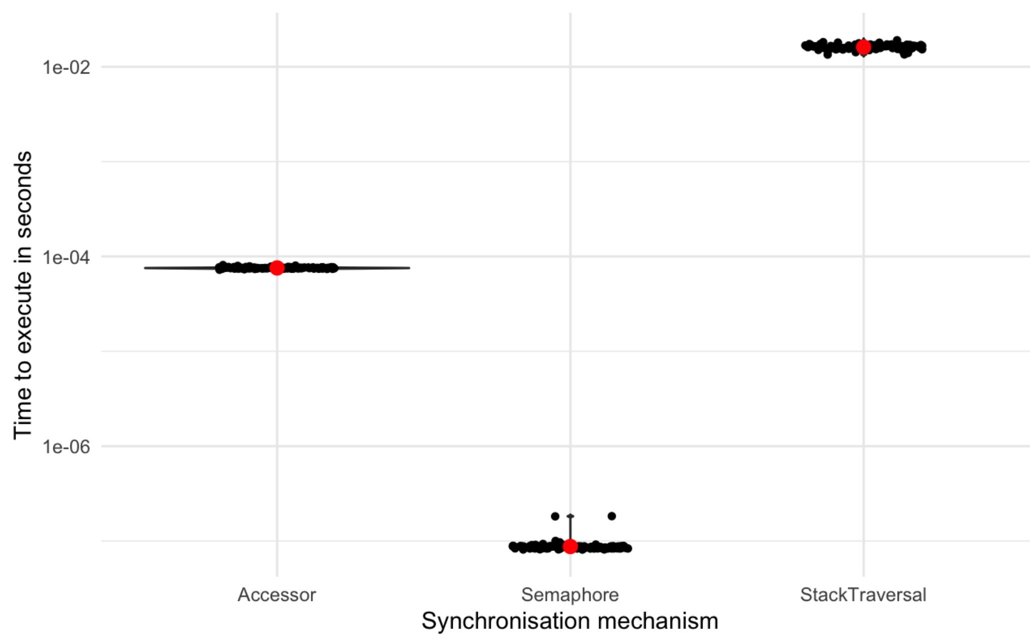

Figure 9.1: Overhead induced by synchronization mechanism:
-Atomic Samurai take/release without stack traversal.
-Semaphore signal/wait.
-Atomic Samurai with stack traversal.
The lower the faster the implementation is.

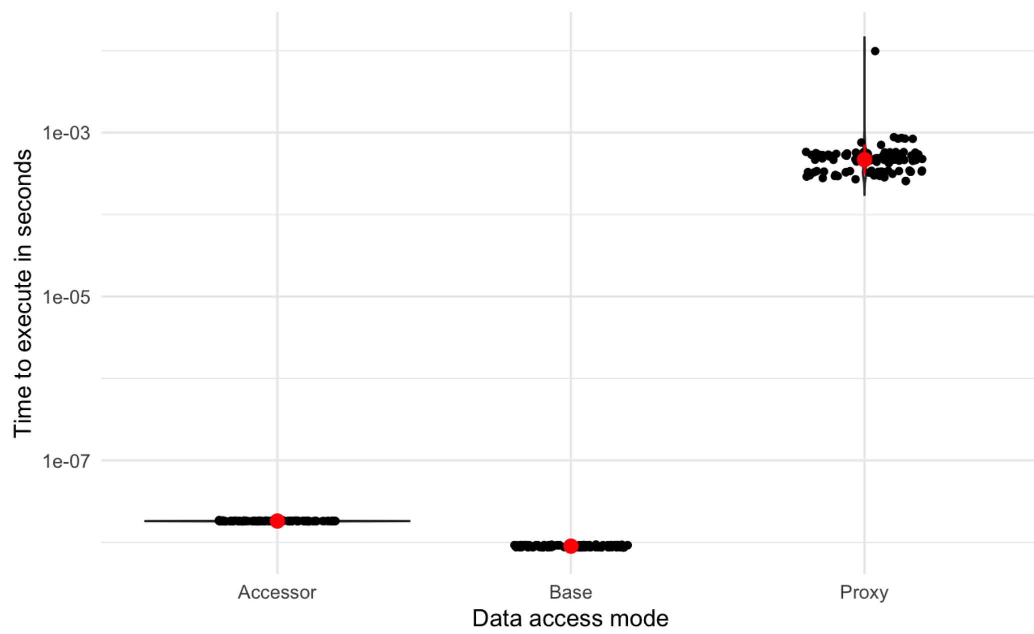

Figure 9.2: Time to access an object:
-Going through accessor
-Baseline.
-With proxy
The lower the faster the implementation is.

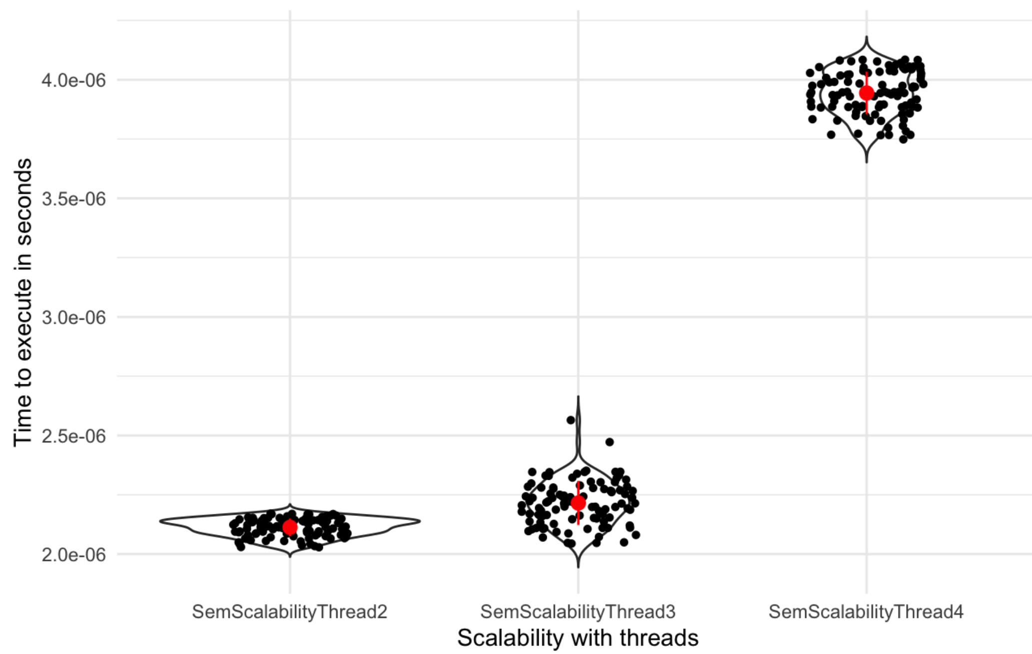

Figure 9.3: Overhead of synchronization with semaphores depending on the number of processes. Higher = more overhead.

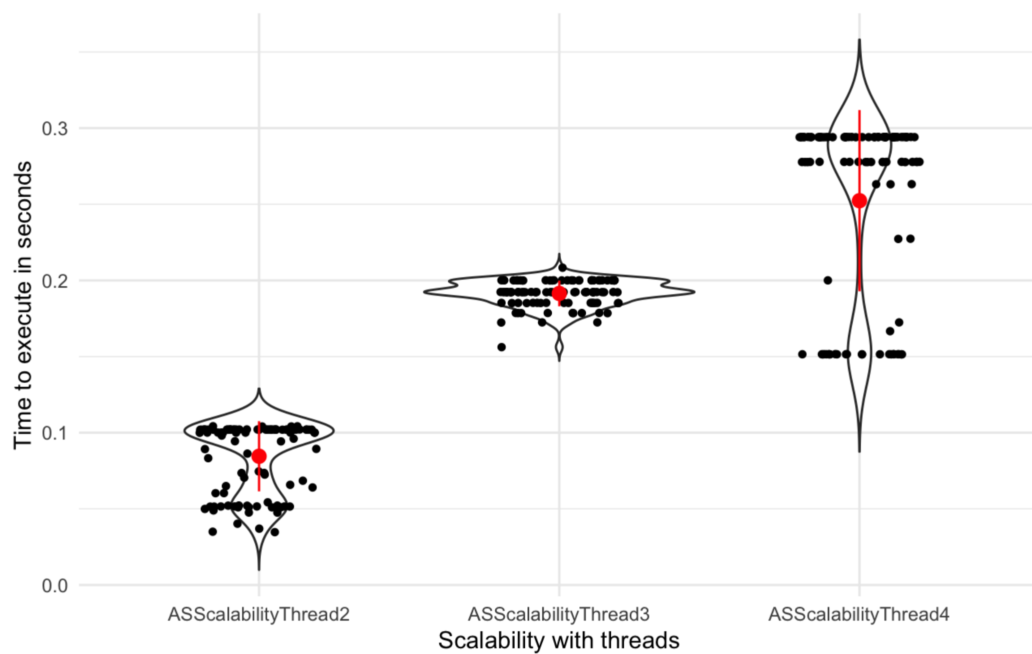

Figure 9.4: Overhead of synchronization with atomic samurai depending on the number of processes. Higher = more overhead.

Chapter 10

Conclusion

10.1 Summary

10.2 Contributions

Bibliography

Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Software Testing, Verification and Reliability*, 13:207 – 227, 12 2003. doi: 10.1002/stvr.281. 55

Clément Béra. A low overhead per object write barrier for the cog vm. In *International Workshop on Smalltalk Technologies IWST'16*, Prague, Czech Republic, August 2016. doi: 10.1145/2991041.2991063. 25, 34

Clément Béra, Eliot Miranda, Marcus Denker, and Stéphane Ducasse. Practical validation of bytecode to bytecode jit compiler dynamic deoptimization. *Journal of Object Technology*, 15(2):1:1–26, 2016. doi: 10.5381/jot.2016.15.2.a1. 26

Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009. ISBN 978-3-9523341-4-0. URL http://books.pharo.org. 11

John Boyland, James Noble, and William Retert. Capabilities for sharing, a generalisation of uniqueness and read-only. In *Proceedings ECOOP 2001*, number 2072 in LNCS, pages 2–27. Springer, June 2001. doi: 10.1007/3-540-45337-7_2. 21, 40

David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings OOPSLA '98*, pages 48–64. ACM Press, 1998. ISBN 1-58113-005-8. doi: 10.1145/286936.286947. 29

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy 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. doi: 10.1145/2824815.2824816. 21, 40

Will Crichton. The usability of ownership, 2020. 21, 40

Dave Dice and Nir 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. ISBN 9781450300797. doi: 10.1145/1810479.1810531. URL https://doi.org/10.1145/1810479.1810531. 21

Stéphane Ducasse and Guillermo Polito. Concurrent programming in pharo, 2020. 37, 51

Patrick Eugster. Uniform proxies for Java. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 139–152, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167485. URL http://doi.acm.org/10.1145/1167473.1167485. 27

M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity os. In *1st EuroSys Conference*. ACM, 2006. doi: 10.1145/1218063.1217953. 20

Daniel Schnetzer Fava and Martin Steffen. Ready, set, go!: Data-race detection and the go language. *Science of Computer Programming*, 195:102473, 2020. doi: 10.1016/j.scico.2020.102473. 20, 42

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 27

Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983. ISBN 0-201-13688-0. URL http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf. 26

Donald Gordon and James Noble. Dynamic ownership in a dynamic language. In Pascal Costanza and Robert Hirschfeld, editors, *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 41–52, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-868-8. doi: 10.1145/1297081.1297090. 29

Timothy 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. ISBN 3540426051. 20

Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. *SIGPLAN Notices*, 48(1):317 328, January 2013. ISSN 0362 1340. doi: 10.1145/2480359.2429109. URL https://doi.org/10.1145/2480359.2429109. 43

Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. doi: 10.5555/1624775.1624804. 20

C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. ISBN 0-13-153289-8. 20, 31

javaConcurrentLinkedQeue. Oracle. java dynamic conccurentlinkedqueue. the java platform 1.5 api specification. hhttp://download.oracle.com/javase/1.5.0/docs/api/java/lang/reflect/Proxy.html. URL https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html. 19

E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006. doi: 10.1109/MC.2006.180. 19

K Rustan M Leino, Peter Müller, and Angela Wallenburg. Flexible immutability with frozen objects. In *Flexible Immutability with Frozen Objects*, volume 5295, page 192 208, oct 2008. ISBN 978-3-540-87872-8. doi: 10.1007/978-3-540-87873-5_17. 21

Carmen Torres Lopez, Stefan Marr, Hanspeter Mössenböck, and Elisa Gonzalez Boix. A study of concurrency bugs and advanced development support for actor-based programs, 2018. 8, 55

Mariano Martinez Peck. *Application-Level Virtual Memory for Object-Oriented Systems*. PhD thesis, Ecole des Mines de Douai - France & Université Lille 1 - France, October 2012. 21

Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Efficient proxies in Smalltalk. In *Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST'11)*, Edinburgh, Scotland, 2011. doi: 10.1145/2166929.2166937. 28

Maged M. Michael and Michael 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. ISBN 0897918002. doi: 10.1145/248052.248106. URL https://doi.org/10.1145/248052.248106. 19

M.M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004. doi: 10.1109/TPDS.2004.8. 20

Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006. 29

Eliot Miranda and Clément Béra. A partial read barrier for efficient support of live object-oriented programming. In *International Symposium on Memory Management (ISMM '15)*, pages 93–104, Portland, United States, June 2015. doi: 10.1145/2754169.2754186. URL https://hal.inria.fr/hal-01152610. 26

Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. Redleaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram. 32, 40

Ryan R. Newton, Peter P. Fogg, and Ali Varamesh. Adaptive lock-free maps: Purely-functional to scalable. *SIGPLAN Not.*, 50(9):218–229, aug 2015. ISSN 0362-1340. doi: 10.1145/2858949.2784734. URL https://doi.org/10.1145/2858949.2784734. 20

Lukas Renggli and Oscar Nierstrasz. Transactional memory for Smalltalk. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 207–221. ACM Digital Library, 2007. ISBN 978-1-60558-084-5. doi: 10.1145/1352678.1352692. URL http://scg.unibe.ch/archive/papers/Reng07bTransMem.pdf. 21

Neil Ivor Sarnak. *Persistent Data Structures*. PhD thesis, New York University, USA, 1986. AAI8706779. 21, 34, 40

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In Xavier Défago, Franck Petit, and Vincent 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, October 2011. Springer. doi: 10.1007/978-3-642-24550-3_29. URL https://hal.inria.fr/hal-00932836. 21

Nir Shavit and Dan Touitou. Software transactional memory. In *Proc. of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, 1995. doi: 10.1145/224964.224987. 40

S. Srinivasan. *Kilim : a server framework with lightweight actors isolation types zero-copy messaging*. PhD thesis, University of Cambridge, King s College, 2010. 32

Camille Teruel, Damien Cassou, and Stéphane Ducasse. Object Graph Isolation with Proxies. In *DYLA - 7th Workshop on Dynamic Languages and Applications, Collocated with 26th European Conference on Object-Oriented Programming - 2013*, 2013. URL http://rmod-files.lille.inria.fr/Team/Texts/Papers/Teru13a-Dyla-ObjectGraph.pdf. 28

Tom Van Cutsem and Mark S. Miller. Proxies: design principles for robust object-oriented intercession APIs. In *Dynamic Language Symposium*, volume 45, pages 59–72. ACM, oct 2010. doi: 10.1145/1899661.1869638. URL http://doi.acm.org/10.1145/1899661.1869638. 26

Tom Van Cutsem and Mark S. Miller. Trustworthy proxies - virtualizing objects with invariants. In *ECOOP'13*, 2013. doi: 10.1007/978-3-642-39038-8_7. 21

Tom van Cutsem, Alexandre Bergel, Stéphane Ducasse, and Wolfgang De Meuter. Adding state and visibility control to traits using lexical nesting. In Sophia Drossopoulou, editor, *Proceedings of European Conference on Object-Oriented Programming (ECOOP'09)*, number 5653 in Lecture Notes in Computer Science, pages 220–243, London, UK, 2009. Springer. doi: 10.1007/978-3-642-03013-0. 27

Werner 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. ISSN 1542-7730. doi: 10.1145/1466443.1466448. URL https://doi.org/10.1145/1466443.1466448. 21

Erwann Wernli, Pascal Maerki, and Oscar Nierstrasz. Ownership, filters and crossing handlers: flexible ownership in dynamic languages. *ACM SIGPLAN Notices*, 48:83–94, jan 2013. doi: 10.1145/2480360.2384589. 29

Erwann Wernli, Oscar Nierstrasz, Camille Teruel, and Stéphane Ducasse. Delegation proxies: The power of propagation. In *Proceedings of the 13th International Conference on Modularity*, pages 63–95, Lugano, Suisse, apr 2014. URL http://rmod.inria.fr/archive/papers/Wern13a-DelegatingProxy-AOSD.pdf. 49

Akinori Yonezawa and Mario Tokoro. *Object-Oriented Concurrent Programming*. MIT Press, Cambridge, Mass., 1987. ISBN 0-262-24026-2. 26

Dieter Zöbel. The deadlock problem: A classifying bibliography. *SIGOPS Oper. Syst. Rev.*, 17(4):6 15, October 1983. ISSN 0163 5980. doi: 10.1145/850752.850753. URL https://doi.org/10.1145/850752.850753. 19