Task Scheduling using Effects in Joelle

Stephan Brandauer
Abstract

Task Scheduling using Effects in Joelle

Stephan Brandauer

This thesis presents the design and implementation of a library for scheduling messages in parallel at runtime. This library is the future backend of Joelle, an extension of Java for parallel programming.

Joelle uses this library for implementing active objects. Active objects execute in parallel and communicate asynchronously through message passing. They convert a method call to messages which they store internally. They execute those messages as soon as possible.

Joelle allows a programmer to partition active objects in disjoint memory regions and to annotate methods with which regions they read or write, their effects. Joelle uses effects to allow messages with disjoint effects to run in parallel while avoiding data races.

This thesis has three key contributions: first, it derives requirements for the library from the available body of research; second, it attempts to summarize this research in a single document, thereby making it useful as an entry point for readers interested in Joelle; third, it develops a novel data structure that guarantees safe, efficient parallelism. In order to check the solution's feasibility, it compares the implementation's performance to the message passing frameworks Erlang and Akka. The thesis concludes that Joelle performs well overall.
## Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Introduction</strong></td>
<td></td>
</tr>
<tr>
<td>1.1 The Problems of Lock-Based Code</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Immutability and Thread Locality</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Joelle’s Components</td>
<td>7</td>
</tr>
<tr>
<td>1.4 Joelle – a Safe Multicore Programming Language</td>
<td>18</td>
</tr>
<tr>
<td>1.5 Design with Ownership</td>
<td>29</td>
</tr>
<tr>
<td>1.6 Active Object’s Internals</td>
<td>31</td>
</tr>
<tr>
<td><strong>2 Design and Implementation</strong></td>
<td>37</td>
</tr>
<tr>
<td>2.1 Building Blocks: the Java API</td>
<td>37</td>
</tr>
<tr>
<td>2.2 Problem Analysis</td>
<td>49</td>
</tr>
<tr>
<td>2.3 Scheduling</td>
<td>55</td>
</tr>
<tr>
<td><strong>3 Performance Evaluation</strong></td>
<td>61</td>
</tr>
<tr>
<td>3.1 Architectures</td>
<td>62</td>
</tr>
<tr>
<td>3.2 The Compared Tools</td>
<td>63</td>
</tr>
<tr>
<td>3.3 The Tests</td>
<td>70</td>
</tr>
</tbody>
</table>
It is impossible to sharpen a Pencil with a blunt axe. It is equally vain to try to do it with ten blunt axes instead.

– Edsger Dijkstra
Acknowledgements

First and foremost, I’d like to thank my supervisors Tobias Wrigstad and Johan Östlund for their help and patience. I remember that at the beginning of this thesis, far too long ago, I knew little about programming languages and nothing about the theories behind it. Through countless discussions with the two, I have learned a great deal about programming language design – the parts where it is a strict discipline but also about the black art: where a “better” or a “worse” is a matter of taste, of intended audience or of “typical future usage”. Knowing this now, I see programming language design as a field in computing science that combines beautiful math, a great practical importance and, very important to me, the need for creativity. Even though I am aware that much of this beauty is not contained in this document (for lack of space, scope and ability), Tobias’s and Johan’s guidance made me want to be one of those who strive to create it.

I also would like to thank my parents who (almost) never asked why my last term in Uppsala seems to be taking a year – and who have always put up with my way of studying, characterized by ignoring the stuff that was boring. Probably that is not always
easy for a parent.

Then, there are Elias Castegren and Andre Hilsendeger, Andre is a very keen observer who, in discussions over a glass of whisky, never seems to fail to spot a wrong argument I make. Elias, likely my future partner in crime, helped by proofreading, figuring out stuff together and sanity-checking random ideas I was having. I’m looking forward to working with you!

Last, but certainly not least, I want to thank Kim for not only proofreading this document, but also for understanding when I was stressed, tired and annoyed without ever losing patience. I hope that I can do the same for you when you’re working on your thesis project!
Recent years have shown that the amount of work a CPU core can do per time unit is not going to improve significantly unless energy efficiency is sacrificed [20]. Instead, the number of CPU cores will grow with multicore CPUs even in the smallest devices. In order to write well-performing software, it will be necessary to write code that uses all available cores. On the Java platform, an important paradigm is still lock based code. Locks, however, come with problems.

This chapter sets the stage for the thesis by explaining Joelle, the multicore-programming language that will use this thesis project’s outcome.

Opening with a section about the problems of locks that highlights the need for multicore-programming languages, we continue to introduce several necessary concepts. In order to put things in perspective, we are going to analyse the basic traits of safe data access, thread locality and immutability that lead to absence of data races\(^1\) the class of bugs we want to rule out. The final part of this chapter will put all the aforementioned concepts together and

\(^1\) *data races*: situations where the output of a program is dependent on the operating system’s scheduler. This could be because two threads accessing the same memory locations in parallel, destroying each other’s writes.
explain how they interact to achieve race freedom without explicit locks and their associated problems.

**Contribution**  In short, the goal of the practical part of this project is to *design* and *implement* a library for scheduling that uses effects. As Joelle at this stage is under development, the progress of the compiler depends on this project.

The solution will need to satisfy the following requirements:

- together with Joelle’s language semantics, scheduling guarantees race freedom while parallelising method calls where this is safe
- achievable performance is in the same ballpark range as with competing languages

Additionally, this document aims to be a single point of entry for readers who try to understand Joelle and therefore covers much of the reasoning behind the language and its constructs. The existing literature on the language is still rather scarce and a comprehensive single source is overdue.
1.1 The Problems of Lock-Based Code

Locks\(^2\) are a simple mechanism that allow to create atomic operations (operations that other threads perceive as either occurred or not occurred, but never partially occurred). Even though they have been used for a long time, parallel programming with locks has remained a domain for experts [10, 20]. Some reasons for this are:

Complex Debugging The fact that lock-based code is hard to debug is one of the reasons: some bugs might not appear on one computer architecture but will on another due to different memory models: Modern architectures have some freedom in reordering instructions on each core for performance. Locks constrain this freedom through so-called memory barriers. Omitting a necessary lock can therefore lead to no (visible) bug on some architectures because the cores do not reorder their instructions so that a bug shows.

Non-Composability Another shortcoming of lock based code is its non-composability: If we consider a hypothetical thread-safe implementation of a hash table, it is impossible to use this implementation for realising the atomic move of a value from one hash table instance to another without either a) breaking abstraction or b) changing the original implementation. Changing the original implementation would allow to implement a specialised method

\(^2\) Locks: objects that can be temporarily owned by a thread. Subsequent requests of other threads to take ownership block the requesting thread’s execution until the owning thread gives up ownership.
that does just that. If that is not possible, however, there is no other way to implement this feature satisfyingly: Providing methods for locking (owning) and unlocking the table (which would essentially give the table lock-semantics) and requiring clients to call them before and after interacting with the table object might seem like a fix and allows a correct solution but this also breaks abstraction as the table now exports the interface of a table and a lock. This solution also shifts the task of ensuring thread safety to the user of the library. This leads to users forgetting to do proper locking or provoking deadlocks [10, 9].

If we think further, the locking and unlocking of tables like this would introduce the possibilities of deadlocks:

**Deadlocks** If a user of the last table’s hash table implementation chooses to use the hash table class’ locking/unlocking facilities, the code is prone to deadlocks. Deadlocks are situations when two threads need the same two locks and each one successfully acquired one and is now waiting for the other. This leads to infinite stalling.

Consider the method in Listing 1. It works for most cases, but if two threads call the method with the same two hash tables in different orders, it can produce a deadlock:

Thread 1: moveValue(A,B, someKey);
Thread 2: moveValue(B,A, someKey);
void moveValue(HashTable From, HashTable To, Key key) {
    From.lock(); To.lock();
    To.insert(key, From.get(key));
    From.remove(key);
    From.unlock(); To.unlock();
}

Listing 1: Pseudo code: A method to move values from one hash table to the other atomically. For simplicity, the method assumes that the key exists in “From”.

Assume that thread 1 manages to lock hash table A in Listing 1 before the operating system’s scheduler decides that thread 2 should continue to run. Thread 2 will now lock object B before waiting for A. Now both threads are waiting for the object the other thread owns, the program is stuck.

Although there are solutions to this problem (ordered access for instance: every time a number of objects have to be locked at the same time, they have to be locked in the same order. This requires even more complex usage of the class.), it is the user of the library who has to write code that is far removed from the task at hand: moving a value from one hash table to the other.

Now that problems of lock-based code are introduced, we will take a step back and look at the fundamental root of races, the absence of immutability and thread locality at the same time.

1.2 Immutability and Thread Locality

Thread Locality means that data can be accessed by no more than
one thread at a time. *Immutability* means that data can not change.

If a program chooses to put every memory location into at least one of these two categories, it is free of races:

<table>
<thead>
<tr>
<th>immutability</th>
<th>thread locality</th>
<th>→</th>
<th>race freedom</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>→</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>□</td>
<td>→</td>
<td>✓</td>
</tr>
<tr>
<td>□</td>
<td>✓</td>
<td>→</td>
<td>✓</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>→</td>
<td>□</td>
</tr>
</tbody>
</table>

Table 1.1: Every memory location has to satisfy one of the first three alternatives of this table in order to ensure race freedom.

In other terms: each memory location, at any given time, can enjoy not more than two out of the three benefits *parallel access*, *mutability* and *race freedom*. Since race freedom is a fixed requirement, no memory location can provide parallel access and mutability at the same time. Versions of the graphic in the margin will be used in Section 1.4 to signify which choice a keyword represents – mutable but only accessible by one thread or immutable but accessible by many threads.

Functional programming languages achieve race free parallelism (the opposite of thread locality) by making data immutable: instead of changing data, data is copied while some parts are updated. This, however, typically results in high rates of object allocations. Locks, as seen in the last section try to limit parallelism where it is necessary. While this can be more memory-efficient,
locks can hurt *scalability* (the ability of a program to decrease runtime when the number of cores is increased), especially if they cover large sections of code or are *highly contended* (many threads try to take a lock at the same time).

We believe that a language can benefit from allowing mutability, even when this means restricting parallelism. Mutable data can allow high performance that immutable data doesn’t – in some cases. In other cases, mutable data is an intuitive concept that does not require programmers to learn functional programming. Explicit locks, however, are not our way to achieve this due to the problems introduced in the last section.

Joelle’s main goal is to provide a type system that forces data to be in one of these categories at all times thereby creating a programming language that is checkable for race-freedom at compile time.

1.3 Joelle’s Components

This section will introduce the concepts – *active objects*, *effects* and *ownership types* – that are necessary to understand Joelle as a whole. Joelle achieves efficient race-freedom through a combination of these, as Section 1.4 will continue to explain.
Active Objects

An active object is an object that decouples method invocation and method execution [12, 13]. Instead of calling a method and retrieving a result synchronously, a client calls a method that adds a message to an Active-Object-internal mailbox and returns a result handle (also known as a “future” or “promise”). As soon as the active object is able to, the message is executed. Having a future allows a client to continue executing until it is impossible to continue working without the future’s value. Then the client can block on the future.

Conceptually, an active object is encapsulating a single thread of control, no other thread is allowed to operate on the active object’s internal mutable state.

Active objects use an internal scheduler to manage the mailbox; a scheduler essentially takes messages from the mailbox in an infinite loop and executes them while guaranteeing the absence of data races. This guarantee, however, depends on the fact that no aliasing\(^4\) of mutable state between active objects is present.

Figure 1.1 shows an active object.

Since active objects process all of their messages (their method calls) sequentially, they are easier to reason about than mutable state that is protected by locks.
Effects

An effect is to read or to write mutable state. Joelle uses an effects system\textsuperscript{5} for Object-oriented languages [8]. The internal state of every active object is divided in regions by annotating the declarations of object fields. Every method of an active object contains its effects on those regions in its signature; there is no pure WRITE effect, READ is implied when declaring the effect WRITE. In short, for any given method $X$, writes($X$) is the set of regions $X$ writes to (with reading implied); likewise, reads($X$) is the set of regions $X$ reads from. Any method called by $X$ may only read memory locations which are contained in reads($X$) or writes($X$) respectively, thereby guaranteeing that no accidental

\textsuperscript{5} effects system: annotations of methods’ read- and or writes. They allow effective, static reasoning about the safety of running two given methods at the same time.
side effects can occur. This is enforced at compile time.

By analysing the effects, a conflict between two messages can be avoided statically: a message of method A can not run at the same time as messages of method B iff A’s execution could affect B’s execution or vice versa. This means that if, for instance, public void A(...) writes region_1 and public void B(...) reads region_1, then those two methods can never be executed at the same time. The same would apply for public void B(...) writes region_1.

Active objects that use an effects system allow many messages to be processed simultaneously. The constraint to be honoured is simply that no two messages with conflicting effects are allowed to be scheduled at the same time. This maintains the guarantee that during the lifetime of a message, the message can not access internal state which is concurrently written to by another internal message and can not write internal state that is concurrently read by another internal message. In this sense, internal data races are guaranteed to be impossible and active objects therefore do not have to employ any further internal synchronisation mechanisms.

The ability to run non-conflicting methods in parallel potentially gives Joelle a big advantage: as long as READ-WRITE effects are few, high parallelism can is achieved.

Figure 1.2 shows an active object with only one region and exclusively Read-Write accesses. In this scenario each and every method call would need to be processed sequentially. Splitting data into more regions and trying to get by with less READ-WRITE effects could allow parallelism.
Figure 1.2: All methods have a Read-Write effect (double-sided arrow) on one single region. This results in purely sequential processing. If even one of the messages would change to a Read effect, this message could run in parallel with itself. If the region would be split into more than one region, chances are that the effects of the methods would not be all Read-Writes any more. Internal parallelisation would likely improve. This active object contains a thread, drawn in the top left corner. This and other symbols are explained in Appendix A.

However, external races (races that involve threads inside two distinct actors) can still happen:

1. Two active objects have access to the same data\textsuperscript{7} – either through presence of global data, through passing as method arguments or through returning a reference to active object internal data. Since there is no mechanism to prevent the mutation of the data by both active objects, data races can be re-introduced [3].

2. An active object exposes internal state that can be modified.

These problems shall be resolved in the following sections.
Ownership Types

The last of the concepts that must be covered in order to treat Joelle is ownership types.

In Joelle, ownership types enforce encapsulation of mutable state in active objects.

In an ownership type system, objects belong to one and only one other object, their owner. This tree-shaped ownership structure with an abstract owner “world” as root divides the heap into a hierarchy, see Figure 1.3. It is possible to think of the memory organisation as boxes: every object that is created is a “box” and every mutable member in that box is strictly encapsulated in it. Only the object itself is allowed to access its internal members. Therefore, if one wants to manipulate internal state of an object, one needs to use that object’s public interface. A difference to conventional encapsulation by use of private is that the compiler also enforces the requirement that internal data never leak to the outside as in Listing 2 and Listing 3. In other words: any direct reference into an object’s state is prohibited by the compiler. The exact guarantee given is the owners-as-dominators rule: references can never cross ownership-domain-borders to the inside. If an object needs to be accessed from outside of its ownership context, the public interface of its owner has to be used.

Ownership types are a relatively simple type system: all types are annotated with an additional owner.

The owners and their meanings are:
(a) Ownership drawn as boxes. The links are allowed to hold references to outside contexts but not vice-versa (example: the reference from Elem-1 to Link-1 is forbidden since it does circumvent List’s public interface/violate the owners-as-dominators rule/point inside an ownership context).

(b) The Ownership hierarchy as a tree. The links would have the type `this Link` in List.

Figure 1.3: Ownership hierarchy of a linked list. Both representations are equivalent. References (←→) are legal, iff they follow the ownership hierarchy (→) upwards for zero or more steps, followed by zero or one downward steps.
- **this**: variables whose types are owned by this point to objects that are strictly encapsulated inside the declaring object. They can never be returned to the outside of the containing object. This is different from private in traditional object-oriented languages, instances of class A can access private members of other instances of the same class. The method `startOtherCar` in Listing 3 and Listing 5 shows the difference in regard to per-object safety. In traditional object-oriented languages, it would be possible to simply return references to encapsulated data that should remain encapsulated.

- **owner**: the object has the same owner as the containing owner. Intuitively, this declares a sibling relationship in the ownership hierarchy. An object is allowed to access the public interface of its siblings as well as of all objects directly nested in parent contexts [4], see Figure 1.3.

- **world**: this object has no owner, it is globally situated in “world”. Every location is allowed to own a reference to it and to call the object’s public interface.

In Listing 2 to Listing 5, a simple example is shown that highlights the possibilities of strict encapsulation via the this owner. In this case, ownership types allow to guarantee the invariant `engine.isStarted() → (driver != null)`.
class Car {
    private final Engine engine = new Engine();
    private Driver driver = null;
    public void setDriver(Driver drv) { ... }
    /* engine leaks! */
    public Engine getEngine() {
        return this.engine;
    }
    public void startOtherCar(Car other) {
        other.engine.start();
    }
    /* starts only if a driver is present */
    public void start() {
        if (driver != null) {
            getEngine().start();
        } else {
            throw new IllegalStateException();
        }
    }
}

Listing 2: The Car class in Java.

// in some other object/method:
// instantiating and starting a car:
Car car = new Car();
Driver driver = new Driver();
car.setDriver(driver);
// compiles and is correct:
car.start();
// compiles but is incorrect:
(new Car()).getEngine().start();
// compiles but is incorrect:
(new Car()).startOtherCar(new Car());

Listing 3: Broken Encapsulation in Java, example adapted from [4]. In the last two calls, cars are started without a driver. This circumvents the original intention of how the class Car is supposed to be used.
class Car {
  // Engine is strictly contained:
  this Engine engine = new Engine();
  // the Car can not mutate driver:
  immutable Driver driver = null;
  // the car can not mutate drv:
  public void setDriver(safe Driver drv) {
      ...
  }
  this Engine getEngine() {
      ...
  }
  void startOtherCar(this Car otherCar) {
      ...
  }
  // starts only if a driver is present
  public void start() {
      ...
  }
}

Listing 4: The Car class in Joelle.
// in some other object/method:
// instantiating and starting a car:
this Car car = new this Car();
this Driver driver = new this Driver();
car.setDriver(driver);
// compiles and is correct:
car.start();
// doesn’t compile because incorrect: no reference
// to the inside of an ownership context allowed:
(new Car()).getEngine().start();
// doesn’t compile because incorrect: owners do not match:
(new Car()).startOtherCar(new this Car);

Listing 5: Working encapsulation in Joelle: getEngine can not be called from outside of Car because it returns a reference to Car's internals. Additionally, the call to startOtherCar would only work if the call-receiving object would be identical to the parameter. The object driver could also be of type this driver or be owned by a completely different object since the setDriver-parameter is of type safe Driver.
1.4 Joelle – a Safe Multicore Programming Language

Joelle is an extension of the Java programming language, thereby enabling easy refactoring of existing sequential code to efficient, race-free, parallel (where appropriate) execution that shields programmers from the complexities of writing explicitly parallel code. Joelle’s way of dealing with the challenges of multicore programming is, unlike that of functional programming languages like Haskell or Erlang, not to simply outlaw mutability but to allow mutability only in contexts where it is guaranteed to be race-free. Joelle does so by providing language constructs that support ownership types, active objects, unique references and immutable data [3].

This section will finally cover the missing pieces – the annotations that we need to support active objects, effects and ownership types, as well as a short introduction to unique references.
Special Annotations

In Joelle, several additional annotations exist in addition to \texttt{this}, \texttt{owner} in order to safely realise its parallelism and create a programming language that let users easily express their intentionss. To make up for this these new annotations allow to create data that can be accessed by several active objects in a safe way.

The additional annotations are:

- \texttt{immutable/safe}
- \texttt{unique/borrow}
- \texttt{active}
- \texttt{region}
- \texttt{bridge/aggregate}

The owner \texttt{world} is never used explicitly as it would allow external data races.

Immutability and Safe References

Data with the owner \texttt{immutable} is considered to be on the highest level of the ownership hierarchy. Immutability in Joelle comes in several varieties:

1. \textit{per-class immutability} – every instance of a class is immutable\footnote{Examples for class immutability in other languages are Java strings, nearly all of Erlang’s data, or many of Scala’s collection-classes.}.
2. **object immutability** – only one specific instance is immutable.

3. **safe references** – references through which only immutable members can be accessed and non-mutating methods can be called.

Joelle does not allow *observational exposure*: read-only-references that allow reading mutable members would have the problem that a thread can read while another thread (using a reference that allows writing) writes to an object. This would lead to a race. Therefore, read-only references are not a part of Joelle [18].

**Per-class immutability**  A class is immutable when it is declared with an immutable modifier:

```java
immutable class ImmutablePoint extends Point { ... }
```

This means that all calls to the constructor of this class will return immutable objects. Creating a point like this:

```java
this Point p = new ImmutablePoint();
```

would be illegal since that would cast the right hand side’s type (`immutable Point`) to the left hand side’s type (`this Point`).

**Per-object immutability**  An object is immutable if it has the annotation `immutable`, an instantiation looks like this:

```java
immutable Point x = new immutable Point(0,0);
```
Safe references  Through a safe reference, a method is neither allowed to cause any mutation to an object nor to rely on mutable data. This includes calling methods that access mutable data. An example for a safe reference is contained in Listing 4 and Listing 5: the Car can only access those fields of safe Driver driver that are guaranteed to not expose effects of any other thread.

The keywords introduced here allow parallel access to data by guaranteeing that objects can not change – or that only non-changing parts of objects are accessed.

Unique References and Borrowing

The Kinds of Uniqueness  A reference is unique if it is guaranteed to be either unusable or the only usable reference to the object it is pointing to [15].

A reference is externally unique if it is guaranteed to be either unusable or the only usable reference to the object which is outside of the object it is pointing to [2]. The internal reference this is therefore allowed and unconstrained in its use.

Ownership types allow the concept of external uniqueness: even if there are several internal aliases of the externally unique reference (like the reference this), the externally unique reference is the only path into that object from the outside. This is guaranteed by the strict encapsulation guarantees of ownership types. This allows to use all classes in an ownership type system to be used in combination with unique references.

Externally unique references allow mutable data in an elegant way:
code can be written as it would be otherwise, no constraints on internal aliasing are necessary. Unique objects are free of races without locking since only one thread can hold a reference to them at a time.

Usage How unique references will be used in the final language is yet to be decided.

For example, one way to use a unique reference is by destructive reads: To destructively read a unique reference means also to nul- 

lify it. The destructive nature of the read is made explicit by adding the -- operator to the reference: uniqueRef--. Ownership types allow a simple, yet safe way to pass data around while maintaining the uniqueness invariant [2].

Non-destructive reads of unique references are not allowed, see Listing 6.

```java
// a newly created object is always unique:
unique Money myMoney = new unique Money();
//using a destructive read, pass myMoney
//to the pay method:
employee.pay(myMoney--);

println(myMoney);
// output --> null
```

Listing 6: Unique references: destructive reads.

If unique references are used in a read-only fashion however, de-structive reads can get in the way.

Imagine that the user wants to print the content of myMoney before
moving it to the `pay` method: in Joelle, this task would be impossible to accomplish since `myMoney` can only be read destructively and `println` does not return `myMoney` back.

Joelle’s solution is the introduction of *borrowing blocks*: these blocks introduce a new, temporary owner which did not exist before and allow a unique reference to be made non-unique for the scope of the block. An example of borrowing is shown in Listing 7.

This block allows the holder of a unique reference to use it as argument to non-consuming methods or as the receiver of messages (in the case of a unique reference to an active object).

Using borrowing blocks, method implementations do not need to declare their behaviour (consumption/non-consumption) explicitly.

```java
// a newly created object is unique:
unique Money myMoney = new unique Money(1000, Currency.EUR);
borrow myMoney as <tempOwner> borrowedMoney {
    println(borrowedMoney);
    // output --> 1000eur
}
// using a destructive read, pass myMoney
// to the pay method:
employee.pay(myMoney--);
println(myMoney);
// output --> null
```

Listing 7: Unique references: borrowing blocks. `borrowedMoney` is owned by `tempOwner` and can therefore not escape the borrowing block.
By using (externally) unique references, it is guaranteed that only one thread can access data at any point in time. Therefore, unique references are a way to guarantee race-freedom for mutable data without using locking.

Active Classes

Active objects, the objects instantiated from Active Classes, are Joelle’s fundamental unit of parallelism but also a way to allow race-free mutable state: their internals are protected from outside threads through ownership types providing encapsulation of mutable internal state, their schedulers respect conflicts of effects and they only expose asynchronous methods through their interface.

Making a class active means that every public method of that class is implicitly converted to a method that schedules a message and returns a Future instead of obtaining the result synchronously.

Since naively nested active objects would break race freedom (see Figure 1.5), active objects are for now constrained to exist only on the top level of ownership without nesting. This is a significant constraint that leads to the consequence that Joelle will be used differently than other actor-frameworks or -languages – specifically, we expect active objects to be amongst the central objects in a software design.

There is a limitation regarding borrowing and active objects: borrow blocks are safely breaking owners-as-dominators because they are only valid for a limited scope and the effects system guarantees that no two threads with conflicting effects pass through a borrow
Figure 1.5: Nested active objects would introduce races: the internal active object could access the enclosing active object’s mutable state as the state is its sibling through the crossed out reference. Since this would lead to unsynchronised accesses from two threads, this situation has to be avoided.

block at the same time. There is, however, a problem when a borrowed object reference is passed to an active object via method call: the callee could access the data at the same time as the caller, resulting in a data race. We solve this by prohibiting the passing of borrowed references to active objects[3]. We might turn back to this issue in the future if it should prove to be overly prohibitive in practice.

Regions

Regions are declared and used as shown in the following listing:

class Point {
    region geometry;
    region meta;

    this Double X in geometry;
    this Double Y in geometry;
}
final immutable String name in meta;

public Point(immutable String name) {
    this.name = name;
}

public immutable Double getX() reads geometry {
    return new immutable Double(X);
}

public immutable Double getY() reads geometry {
    return new immutable Double(Y);
}

public double set(immutable Double x, immutable Double y) writes geometry {
    this.X = x;
    this.Y = y;
}

public immutable String getName() reads meta {
    return this.name;
}
}

As was already explained, regions are crucial for internal parallelisation through methods annotating their effects on the regions.
Aggregates and Ombudsmen

Ownership types are effectively enforcing encapsulation but that also comes with certain problems. For instance, it is not possible to implement iterators on linked lists in a satisfying way [1]. An iterator is a data structure that allows iteration over the list’s elements. Typically, it is required of an iterator for linked lists to retrieve the next element with $O(1)$ complexity [16]. In order to implement iteration this efficient, it would be necessary for the iterator to have access to the list’s internal link objects. This would mean that references exist that point to the inside of the linked list – a violation of the ownership type system.

A solution are ombudsmen that allow aggregates. An aggregate is a context that is equally owned by several objects, the ombudsmen. Owners-as-dominators is relaxed: the type system guarantees that access to the aggregate context can happen through any of the ombudsmen. Figure 1.6 shows a linked list with ombudsmen: List and all iterator objects are owning the list with equal rights. Access can happen through any of them. This allows the iterator to be implemented while still retaining strict encapsulation of the aggregate context [21].

In order to support ombudsmen, Joelle uses the keywords bridge and aggregate. Their usage is straightforward: if a class declares objects with the keyword bridge, it is legal to export those objects to the outside. Those objects are the ombudsmen. Objects that are owned by aggregate are in the aggregate context and can be accessed by any of the bridge objects/ombudsmen.

In Figure 1.6 and Listing 8, the iterators are the ombudsmen.
The application of ombudsmen to active objects is still a field of research; open questions are, amongst others:

- is it possible to use aggregates for active objects? How can scheduling keep track of a potentially unbounded number of dynamically generated ombudsmen?
- will it be possible to share access to aggregates between active objects or will they have to be constrained to be unique references?
- aggregate objects break uniqueness – how will Joelle handle this case?
class Link {
    owner Link next;
    ...
}

class LinkedList {
    aggregate Link head;
    
    bridge Iterator makeIterator() {
        return new bridge LinkedListIterator(head);
    }
    ...
}

Listing 8: A sketch of a linked list implementation. head is part of the aggregate context and all other links (since next is of type owner Link) are its siblings.

1.5 Design with Ownership

In this section, we will outline a bigger example of Joelle's usage, repeating and highlighting some of its features and how they map to real world relationships.

The goal of the design is a model of persons that allows to store a basic social graph:

- Persons can possess things but can hand those things off to other persons if they choose to.
- Persons have other persons as friends.
- Persons keep track of a list of tasks they want to accomplish.
Persons have favourite places. Some of these places might be accessible by anyone, like parks, some of them might be private like the home of friends.

Possessions can be modelled with unique references. If a person is in possession of an object, only that person is allowed to access the object. It is, however, possible to hand off the object to a second person – via a destructive read, thereby eliminating the first person’s access. Race free, unique access is guaranteed.

Friends – as all persons – are implemented as active objects. This way, many references to one person can co-exist safely, as explained in Section 1.3.

Favourite places here are assumed to be public or non-public. Public places are (here) not mutated after their creation, while private places are only mutated by their creator. This is modelled with a list of safe references: the person liking the place is not allowed to cause mutations or expose themselves to observational exposure (see Section 1.4). Person3 is allowed to mutate Home, though, since it is encapsulated in Person3. No one else is able to observe these changes, however.
We show the model of this scenario in Figure 1.7.

Figure 1.7: Ownership types: Ownership drawn as boxes. The object Person1 owns/contains/dominates several lists of references.

1.6 Active Object’s Internals

In this example, an active object has three regions (A, B, C) and four asynchronous methods (R, S, T, U). The effects of these methods are shown in Figure 1.8, where arrows pointing away from a region depict READ effects, and double-sided arrows stand for READ-WRITE effects.
This actor receives a sequence of asynchronous calls (for the sake of simplicity we will assume that their execution takes the same time):

$$[R_0, U_1, T_2, S_3, S_4, T_5, R_6, U_7, R_8, S_9, U_{10}, T_{11}, R_{12}]$$

**Static Regions and their Simplified Dependencies**

Regions and methods in Joelle’s classes are static: fully known at compile time – even though regions and methods can be created dynamically by simply instantiating active objects, the regions and method an active object has are always the same. This excludes use cases where a region would cover, for instance dynamic file
paths:

```java
public void appendToFile(immutable String text,
                        immutable String FilePath) writes FilePath {
...
}
```

This would have the benefit to allow `appendToFile` to run in parallel for calls with two different values for `FilePath` (please note that the parameter `FilePath` is also used as a region. Joelle does not support this kind of dynamic regions.

Here, the region (`FilePath`) is also a parameter and therefore differs per call – the semantics in such a case would be to create a dynamic region per value of `FilePath`, effectively enabling higher parallelism. An actor could write to several files in parallel, while IO on the same file would be serialised. For the sake of simplicity, however, this feature is not part of Joelle, it is up to the programmer to ensure safety (for instance by introducing a region `IO` or by representing files through `active` classes).

Since regions are static, message dependencies are static: the decision whether two messages have conflicts only requires information available at compile time.

For every actor, a conflict-graph can be drawn where message classes are nodes and message classes that are in conflict are adjacent. The graph for the example is shown in Figure 1.9. Since the “conflicts” property is commutative (A conflicts-with B → B conflicts-with A), the graph is undirected.
Considering the dependencies between messages, a schedule of the example is presented in Figure 1.10. This schedule is optimal in the sense that it is not possible to produce a schedule with a shorter total duration. However, in practice, “optimal” solutions are impossible to achieve: the durations of the tasks are not known, also a scheduler has to start execution while the tasks are being submitted – and without knowing all tasks that will arrive in the future, an optimal schedule can not be found. Scheduling of tasks therefore is always on a best effort basis.

Declarative Parallelism in Joelle

In Joelle, two different ways of parallelisation are possible and in practice, a combination of both is expected to be most effective[17]:

*Coarse-grained* parallelism is achieved by converting classic Java classes to *active* classes: an active class replaces all public methods with methods that return a *Future* object, according to the active object pattern in Section 1.3. This way, users of the class
can do additional work while a different CPU core is busy processing their messages.

A more fine-grained parallelism is available through the effects system: since the active object implementation is able to run non-conflicting tasks in parallel, a high potential for parallelisation is possible, in the best case only bounded by the number of CPU cores.

We call these two ways to parallelise external and internal parallelisation.

A difference between Joelle and competing solutions for parallel code is a certain transparency of the parallelism: note that the user declares rather basic and intuitive details about the code: 1) which classes should be active? 2) what are the effects of a method? Based from these declarations, parallelism emerges naturally. The user is not even concerned with threads. We hope that this implicit parallelisation will greatly reduce development com-
plexity by replacing constructs dealing with *threads and ordering* by constructs dealing with *data access*.
Design and Implementation

Now that Joelle is outlined, we continue by describing the practical part of this project.

We start by introducing parts of Java’s API that were used in the project.

After treating Java’s API, we analyse the scheduling problem we face and use that analysis to design data structures for active object mailboxes.

2.1 Building Blocks: the Java API

One of the key advantages of the Java platform is its wealth of well tested libraries and data structures that come with it. Most – but not all – of the relevant data structures are found in the java.util.concurrent package (here in short: j.u.c). In the following sections, we will introduce Java’s language facilities for concurrent programming – we use some of those to construct our implementation while others are basics that are important to know.
Mutual Exclusion through Synchronized

The guarantee that no two threads can access a memory location at the same time – mutual exclusion – can be achieved through several techniques. One of those is the keyword `synchronized`. It is not a part of Java’s libraries, but of the core language. The keyword can be used to annotate methods as well as to introduce so-called `synchronized sections`.

A synchronized method guarantees that no two threads will execute the same method at the same time. Additionally, should there be several synchronized methods, no two threads will execute any of them at the same time.

Listing 9 gives an example for synchronized methods: the keyword `synchronized` on two methods guarantees, for each object of type `SafeCounter`, that no two threads can execute any of `incrementAndGet` or `decrementAndGet` in parallel. There can neither be two parallel `cnt++` operations, nor two parallel `cnt--`, nor a `cnt++` in parallel with a `cnt--`.

A more flexible usage of `synchronized` is in form of `synchronized blocks`. Synchronized blocks allow more fine grained parallelism by synchronising on an object. The invariant is that as soon as a thread enters a synchronized block with a certain parameter, no other thread can enter any synchronized block with the same parameter. Therefore, if some synchronized blocks would use different objects as parameters, one thread would be allowed in each
class SafeCounter {
    private int cnt = 0;
    public synchronized int incrementAndGet() {
        cnt++;
        return cnt;
    }
    public synchronized int decrementAndGet() {
        cnt--;
        return cnt;
    }
}

Listing 9: Synchronized methods: only one thread at a time can be inside any synchronized method for a given object.

of them at the same time. This allows higher parallelism while it also increases the code’s complexity.

Listing 10 shows synchronized blocks that receive a parameter. The parameter can be any object but no primitive values. Only one thread can be in any of the synchronized blocks with the same parameter. Therefore, mutateMetaData and setCoordinates can run in parallel since their parameters differ. Note that access to the state is only safe as long as all accesses are under proper synchronisation; should a programmer forget one of the synchronized keywords, or add an access to data in setCoordinates without synchronising on data, the code would exhibit data races. Imagine, that a reference to the inside of geom is returned by a method of the class. Access to that reference could happen in parallel with an execution of setCoordinates. It would, in fact, be impossible to synchronise access to that reference, since geom is not known outside of the object. Synchronized does work if and only
class SafePoint {
    private MetaData data;
    private Geometry geom;
    public void mutateMetaData() {
        synchronized(data) {
            data.transmogrify();
        }
    }
    public void setCoordinates(double x, double y) {
        synchronized(geom) {
            geom.setX(x);
            geom.setY(y);
        }
    }
}

Listing 10: Synchronized blocks: Explicit synchronisation on different objects allows mutateMetaData() and setCoordinates() to run in parallel.

if it is clear which object protects which set of objects. Using the keyword synchronized requires strict discipline in order to work properly.

java.langRunnable, j.u.c.Callable<T>

A Runnable is an interface for objects meant to be executed by any thread. A Runnable implements a method public void run() to this end.

Callables are similar to Runnables with the difference being that they also return results by implementing public T call() instead of the above mentioned run. Note that call() has a generic
return type T as opposed to void run() in the Runnable interface.

In Joelle, we need both classes: Runnables are used to model method invocations of void-methods while Callables are used to model method invocations of methods with return values.

So far, it is neither clear how a Java program using Runnables or Callables can ensure that those objects have already been executed nor how the return value of Callables can be obtained after their execution. Both of these problems are solved by Futures.

**Future Values**

A future, as already described in Chapter 1, is an object that allows to wait for a task to finish and to retrieve the task’s return value. A future is tied to a task in form of a Runnable or Callable. In Joelle, this means that for each method call, a future will be generated from a Runnable or a Callable. The client can use this future to wait for the execution to finish and to retrieve the return object it finished with (if the method type returns non-void).

This functionality is grouped under Java’s j.u.c.Future<T> interface. Calls to an active object’s public method return an implementation of this interface. They allow a caller to block until a result is available using the public T get() method. Should the execution of a message result in throwing an exception, the exception is thrown when the client calls get().

**Unifying Runnable and Callable with j.u.c.FutureTask**  For historic reasons, Java has the two interfaces Runnable and Callable
that have similar responsibilities. This raises the questions: how can a client wait for a runnable to finish? If it does so using a future, what should the \texttt{get} method return? The class \texttt{FutureTask} implements the interfaces \texttt{Runnable} as well as \texttt{Future}. Objects of type \texttt{FutureTask} can contain a \texttt{Runnable} or a \texttt{Callable} as task, sometimes called \textit{payload}. They are useful since they can be used to block on the completion of their contained task: if the task is a \texttt{Callable}, the \texttt{get()} method will return the result of the call, if it is a \texttt{Runnable}, it will return a predefined value passed to the constructor \texttt{FutureTask(Runnable r, T result)}. In Joelle, the value that will be returned after waiting for void methods to run is simply \texttt{null}. The \texttt{FutureTask} class is important because it treats \texttt{Callables} and \texttt{Runnable} equally and thereby avoids any visible differences between methods of void and non-void return types to the outside.

```
<<interface>>
Runnable
+ run(): void

<<interface>>
Callable<T>
+ call(): T

<<interface>>
Future<T>
+ get(): T

FutureTask<T>
... + run(): void + get(): T
```

Figure 2.1: By either hiding a runnable or a callable, the \texttt{FutureTask} makes software design simpler. Note how \texttt{FutureTask} inherits from \textit{and} aggregates a runnable in order to unify \texttt{Runnable} and \texttt{Callable}. 
Threads

A `java.lang.Thread` is an independent thread of execution in a Java program.

Runnables can be executed by `Thread` objects as shown in Listing 11. Note how the usage of the class `FutureTask` allows a uniform way of dealing with runnables and callables.

In Listing 11, the tasks are executed by three freshly spawned threads. This approach comes at a very high cost since spawning a thread is an expensive operation both in terms of time and memory. This cost is so high that it is only feasible if the number of tasks is guaranteed to be low (certainly in the range of less than 1000 would be a ballpark number) and the runtimes of the tasks is long enough that the parallelisation pays for its overhead.
FutureTask task1 = new FutureTask(new Runnable() {
    public void run() {
        doExpensiveOp1();
    }
}, null); // Future.get() will return null

FutureTask<Integer> task2 = new FutureTask(new Callable() {
    public Integer call() {
        return doExpensiveOp2();
    }
});

Runnable task3 = new Runnable() {
    public void run() {
        doExpensiveOp3();
    }
};

(new Thread(task1)).start();
(new Thread(task2)).start();
(new Thread(task3)).start();
doOtherStuff();
// returns null once doExpensiveOp1() has finished: task1.get();
// returns the return value of doExpensiveOp2(): Integer result = task2.get();

Listing 11: Using freshly spawned threads to execute task1/2/3 in parallel. The objects task1 and task2 allow to wait for them to be finished since they are FutureTasks, and therefore also implement Future. There is no way to wait for task3’s execution short of observing task3’s side effects.
Abstract Execution of Tasks:
\texttt{j.u.c.ExecutorService}

As explained in the previous section, using a thread for each task is too expensive for most applications. The interface \texttt{ExecutorService} provides an abstraction for executing tasks. Implementations of the interface take tasks (in form of runnables or callables; Joelle's scheduler uses \texttt{FutureTasks}) and execute them in an \textit{unconstrained} way, in other words: as soon as possible. The benefit of that abstraction is that the way of running threads is transparent to the client: Java comes with several implementations that distribute the tasks evenly to a set of thread objects, but there are also open source implementations available that execute all passed tasks immediately and synchronously in the client thread.\footnote{\texttt{MoreExecutors\_sameThread\_Executor()}} in Google's open source guava library does exactly that.

Implementations of \texttt{ExecutorService} export several submitting methods:

1. \texttt{public Future\textless T\textgreater submit(Runnable task, T result)}: submits a task and returns a future that will yield \texttt{result} once the task is completed.

2. \texttt{public Future\textless ?\textgreater submit(Runnable task)}: equivalent to the call \texttt{submit(task, null)};

3. \texttt{public \langle T\rangle Future\langle T\rangle submit(Callable\langle T\rangle task)}: submits a task and returns a future that will yield the task return-value.

To avoid confusion, Joelle's schedulers use the second method which receives a \texttt{FutureTask} (which implements \texttt{Runnable}), rep-
resenting a method invocation as a parameter, this FutureTask object also doubles as a future to be returned to the client. The schedulers discard the future returned by submit.

The mechanism of unifying the treatment of runnables and callables is similar as in the case of FutureTask: submit could be called either with a callable or with a runnable but will return a future in both cases.

**j.u.c.ThreadPoolExecutor**

The thread pool executor implements a thread pool: it contains a number of so-called worker threads which are executing incoming tasks as quickly as they can manage.

ThreadPoolExecutor is highly configurable and the j.u.c.Executors class provides some static factory methods to easily generate pre-configured instances, for instance thread pools that grow whenever a task is added but all worker threads are busy.

Even though the class is flexible and adaptable to many use cases, the implementation suffers from a severe performance bottleneck: as all tasks are managed by a single internal queue, adding to that queue suffers from high contention (as all clients can add concurrently) as well as taking from the queue (as all worker threads can take concurrently). The contention problem can be expected to get worse with a rising number of CPU cores as we show in Figure 2.2.
j.u.c.ForkJoinPool

In order to avoid the contention problems of the thread pool treated above, we explored using the fork-join pool. Even though the pool is part of a much larger family of classes, the fork join framework, the important aspect for this thesis is how the ForkJoinPool’s performance differs from that of ThreadPoolExecutor.

The ForkJoinPool avoids the bottlenecks of the ThreadPoolExecutor, by giving each worker thread its own queue of tasks. If a worker thread, during execution of a task, happens to submit a new task to the ForkJoinPool, that new task will be added to the worker thread’s queue. This means the add operation is almost free of contention. When a task is done, the worker thread will try to take a task from its own queue and only if the private queue is empty, it will try to dequeue a task from another worker thread’s queue. Taking tasks from another worker thread is commonly called work stealing. The contention when taking tasks from the queue is much lower because most of the time, workers will have non-empty work queues.

Using this class proved consistently faster than ThreadPoolExecutor in all measured benchmarks and on all used architectures, so for the rest of the document, all values reported are obtained using the ForkJoinPool.
Figure 2.2: A thread pool executor. The client threads on the left submit tasks and contend to add them while the worker threads inside the thread pool executor contend to retrieve tasks. This problem hurts performance of the thread pool, especially with high numbers of workers and/or clients.

Figure 2.3: A fork-join pool. Every worker has its own queue. The contention is much lower since messages from the clients are distributed across the work queues now. Should a worker thread run out of tasks, it tries to steal (not depicted) tasks from a randomly chosen other workers.
2.2 Problem Analysis

Messages as Partially Ordered Set

When an asynchronous method is called, the call is converted to a message object and a future object is returned immediately. The message object is stored by the scheduler. It is now the scheduler’s responsibility to execute the task as soon as possible.

However, the scheduler is bound by a constraint that ensures behaviour as intended by the programmer: Messages that depend on each other can not be re-ordered.

For a message $A$, $\text{reads}_A$ is the set of regions read during $A$’s execution, $\text{writes}_A$ is the set of regions written during $A$’s execution and $A$ conflicts-with $B$ means that during parallel execution of two messages $A$ and $B$, a data race could happen since there is at least one region which both access and at least one of the accesses is a write:

\[
A \text{ conflicts-with } B \iff (\text{writes}_A \cap \text{reads}_B) \cup (\text{reads}_A \cap \text{writes}_B) \cup (\text{writes}_A \cap \text{writes}_B) \neq \emptyset
\]

Additionally, $B$ received-before $A$ means that a message $B$ was submitted to the active object before a message $A$ was. If the two messages are submitted concurrently, meaning that the time ranges it takes for them to be submitted overlap (see Figure 2.4), each of them could legally end up first in the queue. Another way to say this: submitting of messages has to be linearizable [11].
Figure 2.4: Submitting messages. \( A \) and \( B \) are concurrent submissions – either \( A \) received-before \( B \) or \( B \) received-before \( A \) could hold, depending on the scheduler implementation – while \( C \) and \( D \) are ordered, \( C \) received-before \( D \) holds.

**Definition 2**  
\[ (A \text{ conflicts-with } B) \land (B \text{ received-before } A) \]  
\( A \) depends-on \( B \)

The messages an actor receives are partially ordered: when the effects of two messages are in conflict, the one that was received first has to be *finished* before the second one can start to run. Disallowing them to run at the same time would not be enough; in this case, a read to an actor member that was issued after a write to the same member could be scheduled before the write and therefore produce a logically incorrect result. When they are *not* in conflict, they can be scheduled in any order (including at the same time).

As some readers might be familiar with Erlang’s guarantee to deliver messages from any process \( A \) to any process \( B \) in sending order, Joelle’s schedulers do *not* give this guarantee. However, they guarantee execution in sending order for messages with conflicting effects. This follows from the definition of depends-on which rules out non-conflicting messages.
**Fairness**

Definitions for the term “fairness” in the literature vary: “Strict fairness” is scheduling adhering to a FIFO-order:

**Definition 3** *Strict fairness*: "Suppose, there are two requests $p_1$ and $p_2$ and two corresponding responses $q_1$ and $q_2$. We may want to impose strong FIFO discipline on the responding agent and state that if $p_1$ preceded $p_2$, then $q_1$ will precede $p_2$.” [7]

Providing strict fairness would negatively impact scalability: strict FIFO-ordering of messages leaves little room for internal parallelisation since only messages that have no dependency to the message that was submitted directly before them could run in parallel. Coming back to the example in Section 1.6, specifically Figure 1.10: if the messages $R_0$, $U_1$ and $T_2$ would be submitted in that order, no parallelisation could be exploited by a strictly fair scheduler – $U_1$ conflicts with $R_0$ and $T_2$ could not run before $U_1$. With a non-strictly fair scheduler however, $R_0$ and $T_2$ can run immediately; as soon as $R_0$ is done, $U_1$ can be executed. $T_2$ is “overtaking” $U_1$ because through analysis of their effects it is safe to say that $T_2$ can not observe the changes from $U_1$.

It is easy to see that the question of relaxing the FIFO requirement is central to achieving high parallelisation – examples exist where relaxation of the FIFO-requirement lead to higher throughput and lower latency [6].

**Definition 4** "Fairness means that every process$^{10}$ gets a chance..."
to make progress, regardless of what other processes do. Fairness is guaranteed by a truly concurrent system in which each process is run on its own processor: one process cannot halt the physical execution of a process running on a different processor.” [19]

Considering all these factors, the choice for Joelle is to guarantee non-strict fairness that most closely follows Definition 4. Some details that are not mentioned in this definition still need attention: blocking futures and starvation.

Blocking Futures

If a message contains a reference to a Future, it can block on that future. This is a violation of Definition 4 – for instance, a process that is stuck in an infinite loop would be blocking every thread waiting for the result.

In the context of a programming language, the case of never-terminating messages can be seen as an exception in regards to fairness: calling an asynchronous method that does not end has similar consequences as calling such a method on a passive object.

The consequences of a message never terminating can prevent other messages from running indefinitely, namely all those messages that depend on the blocking message – the scheduler is not allowed to run them since that would violate the partial ordering from Section 2.2. In a short peek forward, if message \( U_6 \) in Figure 2.5 would not terminate, none of the messages \( U_{7..14} \) could run, ever.
Also, the choice of the backing thread pool does make a difference: should a thread pool with constant size $N$ be picked, $N$ messages in infinite loops would be enough to bring the whole system to a halt. There are ways around this issue: for instance, choosing a thread pool implementation that grows on receiving tasks when all worker threads are busy (like `Executors.newCachedThreadPool()`).

A language with cheaper threads (sometimes called “green threads”) than Java has the advantage that no thread pool is necessary: for each task, a cheap thread can be spawned. Therefore, no thread pool can ever “run out” of worker threads, thereby halting the whole program. However, if a green thread gets stuck in an infinite loop, this still can affect other threads: if three green threads A, B, C interact and the thread A sends a message to the thread B which forces B into an infinite loop, B will not be able to react to future requests from C. If C would now wait for an answer from B, C would block – indirectly because of A’s original message to B.

**Starvation**

The term “starvation” describes the situation when a task cannot be finished because it is waiting for a resource – in scheduling, that resource is the CPU core. A scheduler has to make sure that this situation does not arise: each message delivered to the actor needs to be guaranteed to be executed eventually.

To illustrate, a well-known example of starvation is the readers-writers-problem: there are two kinds of threads which all access the same data. The first kind tries to read the data, the second
kind tries to write the data.

The first way of implementation would be to give access to the data to only one thread at a time. This is safe and ensures that all threads eventually get access. Unfortunately, this is a suboptimal solution since two readers can not access the data at the same time.

Another way of implementation would let readers execute their task immediately when the data currently is held by another reader. It also is a suboptimal implementation since it allows starvation of writers: A writer might be waiting for a reader to finish, when another reader requests permission to read – and is granted that permission immediately. If enough readers request the data, the writer will never finish.

A similar implementation would give priority to writers: if several threads wait for the lock, writers have priority over readers. This implementation would allow starvation of readers.

In Joelle, this problem does not arise: since the readers and writers (implemented as methods) have conflicting effects (for instance: the example actor’s method \( R \) is a reader, the method \( U \) is a writer on region \( A \)), ordering is maintained: successive \( R \) calls are executed in parallel, but \( R \) calls can never be executed before a \( U \) call that was made before them. Figure 2.5 shows the pattern of dependencies of messages that arises from the readers-writers problem.

We believe that this inherent fairness might prove to be one of Joelle’s strengths by allowing users of the language to think about their task without worrying about scheduling.
Concluding Fairness

Fairness in Joelle depends on the implementation of the thread pool: constant sized thread pools do not allow for fairness since all workers can be blocked by infinite looping tasks which would prevent the program from making any further progress. Thread pools that grow with the demand for workers (as implemented in the Java API) are one solution that is directly applicable to Joelle.

2.3 Scheduling

Equipped with some knowledge about the admissible reorderings a scheduler can make in order to use the available computing power effectively, it is now time to cover data structures that implement
the rules derived in the previous section.

In the section on ExecutorServices, we explained how they will execute tasks passed to them in an unconstrained way, as soon as possible. In Joelle’s context, the problem is this lack of constraints: an implementation of ExecutorService can and will blissfully execute all tasks passed to it as soon as possible, not caring for high-minded concepts like read-write conflicts. This is why a scheduler in Joelle has to be able to represent an execution DAG (directed acyclic graph) like in Figure 2.5 and use this structure to hand off tasks to the ExecutorService in a concerted manner, respecting their dependencies. As it will turn out in this section, the naive approach is not the most efficient, but a data structure specialised to use knowledge about static dependencies in Chapter 1 performs well.

DAG-based Scheduling: the Naive Implementation

The developed DagScheduler class exploits the fact that partially ordered sets can be represented as DAGs as in Figure 2.5 [5]. When a task is submitted to the scheduler, it gets converted into a graph node and is wired into the execution DAG. The upheld invariant is that a node can be passed to the ExecutorService if there are no outgoing depends-on edges any more. In the execution DAG in Figure 2.5, the task $U_1$ would be free to run. After $U_1$ is done, the tasks $U_{2..5}$ could be passed to the ForkJoinPool for parallel execution. To this end, whenever a task $A$ is finished, the active object will remove it from the DAG and then check all tasks
that depended on $A$. If $A$ was their only (remaining) dependency, they can be passed to the fork join pool.

\[
\begin{align*}
R_0 & \rightarrow R_1 & \cdots & \rightarrow R_{N-1} & \rightarrow U_N
\end{align*}
\]

Figure 2.6: The resulting DAG after submitting the messages $[R_0 \ldots R_{N-1}, U_0]$. The message $U$ has $N$ dependencies which are expensive to maintain.

The problem of the DagScheduler is the fact that the number of dependencies of a message is unbounded – there can be an unlimited number of running messages at any time. Figure 2.6 shows such a case. During our tests, the DagScheduler performed so badly (in terms of runtime and memory consumption) that we decided to not even benchmark it in Chapter 3 for simplicity.

**Multi Queue Scheduling**

The scheduler presented here is, compared to the naive scheduler, a more efficient solution. The MultiQueueScheduler maintains one internal inbox for each type of message it can receive, so there is one queue for each asynchronous method the actor implements.

When a new task is submitted, it is added to the end of its class’s queue. Additionally, for each dependency class of that message, a Barrier object is appended to the list for that dependency class. This maintains the invariant that:

**Definition 5** Every task which has no barrier before it in its queue is free to run.
Adding barriers to the queues of dependent classes makes use of the fact that dependencies are static: all messages of a class $A$ will either depend on all messages of a class $B$ or not. There are no exceptions. Section 1.6 explains this in more details.

If the newly added task is found to be the first item on its queue, it is submitted to the ExecutorService immediately. Figure 2.7 shows a MultiQueueScheduler executing the standard example.

After a task finishes, all of its barriers are removed from the queues. For each removed barrier, the queue it was contained in is flushed: all the tasks from the beginning to the first barrier are handed to the fork join pool. In Figure 2.7 (c), $R_0$ and $T_2$ would be scheduled immediately, while $U_1$ and $S_3$ are blocked at first. As soon as $R_0$ is completed, the barrier blocking $U_1$ is removed and $U_1$ gets executed.

In Appendix B, we show the two schedulers as UML diagrams.
Figure 2.7: The MultiQueueScheduler’s internal queues. The black lines represent barriers that tied to the tasks. In this example, $R_0$ and $T_2$ can be handed off to the fork join pool since they have no barrier in front of them.
The goal of the evaluation attempt here is by no means to prove a relative performance difference between the compared languages but to roughly estimate Joelle’s performance and scalability compared to other, more mature and well established tools, the programming language Erlang and the library Akka, implemented in Scala.

In order to get a better impression of the actual performance, each benchmark for each tool is executed on two different computer architectures.

We try to compare straightforward code. This means: we will use defaults – no changes of command line switches\textsuperscript{11}, no optimisation for any specific architecture. This necessarily means that the maximum achievable performance for any of the tools or platforms is likely not reached. However, since we try to evaluate without giving our implementation an artificial advantage, we must rule out over-optimisation of the Joelle benchmarks. Our way to do that is to not optimise any of them. Although we acknowledge that this approach is not without problems, practical limitations

\textsuperscript{11} although we tried a changed heap size for Erlang, both larger and smaller. No change in performance was observed.
force us to take it, namely the fact that the author of this thesis has more experience in optimizing Java than in optimizing Scala or Erlang and also knows Joelle’s scheduler better than both, Erlang and Scala/Akka.

### 3.1 Architectures

The implemented code is compared on two different architectures, one representing the class of “typical” office PCs, the other one being an example of a larger server as it might be used in – for instance – scientific computing applications or application servers.

<table>
<thead>
<tr>
<th></th>
<th>PC</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor(s)</td>
<td>Intel Core i5 (2.676GHz)</td>
<td>4 × Intel Xeon E5-4650 (2.7GHz)</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256k</td>
<td>256k</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>8MB</td>
<td>20MB</td>
</tr>
<tr>
<td>Total #Cores</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>Total #Threads</td>
<td>4</td>
<td>64</td>
</tr>
<tr>
<td>Memory</td>
<td>1GB</td>
<td>128GB</td>
</tr>
</tbody>
</table>
3.2 | The Compared Tools

Erlang

Erlang (http://www.erlang.org/) is developed by Ericsson since the mid-1980s. The language uses mostly immutable data, mutability can be achieved using so called ets tables (http://www.erlang.org/doc/man/ets.html). There is, however, some overhead in their usage. This ensures in practise that programmers will not make uninformed decisions towards mutable data.

Listing 12 and Listing 13 contain a simple example of an erlang program where a process (a lightweight thread) receives tick messages and increases a counter upon receiving them.

The erlang version used in the evaluation was R15B03.

Erlang is untyped, data is largely passed in form of tuples and a flavor of tuples called records, which are basically tuples with named fields. There exist, tools to increase type safety. One example is dialyzer, developed at Uppsala University and nowadays included in the Erlang distribution [14].

Listing 14 shows the usage of Erlang records: a record is simply a tuple with its “class” name in the first field. Fields after that are open to be read by any thread. Records can not be mutated but any thread can make copies of a record with altered fields.

Single core performance is not one of Erlang’s strengths (on http://benchmarksgame.alioth.debian.org/, the Erlang implementations are on median, 17.04× slower than the fastest im-
-module(tick).
-export([start/0]).

-spec start() -> pid().
start() ->
    spawn(fun() -> loop(0) end).

-spec loop(integer()) -> no_return().
loop(Count) ->
    receive
tick ->
        NewCount = Count + 1,
        io:format("Count: ^p^n", [NewCount]),
        loop(NewCount);
        {getCount, Sender} ->
            Sender ! Count,
            loop(Count);
        Other ->
            io:format("received unknown command: ^p^n", [Other]),
            loop(Count)
    end.

Listing 12: tick.erl: A usage example for Erlang. The -spec lines are used by dialyzer. The keyword spawn starts a process that executes to passed function and uses receive blocks to block until messages are sent to it.

Implementation of a benchmark, whereas Java 7 achieves 2.06× and Scala 2.07×). Erlang is a good choice for distributed systems, especially considering OTP (the Open Telecom Platform, a collection of libraries that allow rapid development of fault tolerant distributed systems).

A noteworthy feature of Erlang is location transparency – its abil-
Listing 13: Interaction with tick.erl.

ity to make communication between processes transparent with respect to the physical machine they are running on – sending a message to a process on a different server is working exactly the same way as sending a message to a local process.

**Soft Realtime**  Erlang uses its own process scheduler (that is backed by several OS threads) to execute its processes. Since garbage collection in Erlang is done separately for each process, there is no “stop-the-world” effect (see [http://www.erlang.org/...../faq/academic.html#id55790](http://www.erlang.org/...../faq/academic.html#id55790)) – this typically allows Erlang systems to deliver predictable performance. The term used is “soft real time” – Erlang tries to allow real-time guarantees like “a response will take less than 10ms in 97% of cases”.

```
% start a tick-process and store process-ID in P:
P = tick:start(),
% send 'tick' message to process P:
P ! tick,
% output --> Count: 1
P ! tick  % another message..
% output --> Count: 2
P ! tick.  % another message..
% output --> Count: 3

% ask process P for count:
P ! {getCount, self()},
% store the answer in 'Count':
Count = receive Answer -> Answer end,
io:format("received Count: "\p\n", [Count]).
% output --> 3
```
% declaring the record "user":
-record(user, {name :: string(),
    birthday :: string(),
    bio :: string()}).

%%% instantiating a user:
User = #user{name="Joe User",
    birthday="1980/1/1",
    bio="..."},

% accessing the user's fields:
io:format("name=~s~n", User#user.name),

% printing the whole user record:
io:format("user=~p~n", [User]),
% output --> user={user, "Joe User", "1980/1/1", "..."}

% creating a tuple that looks like a user:
Tuple = {user, "Joe User", "1980/1/1", "..."},
io:format("tuple=~p~n", [Tuple]),
% output --> tuple={user, "Jane User", "1980/1/2", "..."}

io:format("record equal to tuple?: ~p~n", [User =:= Tuple]).
% output --> record equal to tuple?: true

Listing 14: Records in Erlang are tuples with syntactic sugar like named instantiation (line 9) and named field access (line 12). The fields of a record can have associated data types (line 2), this allows dialyzer to statically discover bugs.
Scala

Scala (www.scala-lang.org) is a programming language on the JVM, combining object-orientation with functional programming. Amongst Scala’s strengths are good interoperability with Java code which can make it a good choice in presence of legacy Java code, a big set of existing Java libraries that are easily usable for developers and – subjectively – elegant syntax that allow the development of expressive libraries.

Scala is statically typed but uses type inference where possible in order to increase flexibility and legibility.

Since Scala is running on the JVM and directly compiles to Java bytecode, the language profits from the JVM’s single core performance.

Akka

Akka (http://akka.io/) is a concurrency framework, written in Scala but with APIs in Java and Scala. One of Akka’s main influences is Erlang’s OTP.

Akka implements actors with similar features as Erlang: location transparency and fault tolerance through supervision trees. In terms of supervision trees, Akka goes one step further than Erlang: every spawned actor is supervised.

The versions used were Scala 2.9.1 and Akka 2.0.
Possible Violations of Encapsulation in Akka

Akka does not provide race freedom.

If a method on an actor is called, it is executed synchronously, even if the actor is in the middle of processing a message. This leads to potential data races. Akka’s solution is the class `ActorRef`: When an actor is created using the `system.actorOf` call (see Listing 16 and Listing 17 in Appendix C), the value returned is not a reference to an object of the class `TickActor`, but to an object of the class `ActorRef[TickActor]`. Therefore, the client is never able to directly call methods on the actor. If the client sends a message to the `ActorRef`, the message is forwarded to the “real” actor.

However, since the user never sees a reference to the Actor directly, it is not possible to access this method.

This mechanism works as long as the implementation of the Actor ensures that the `this` pointer never gets exposed to the outside, adding a clause like in Listing 15 would violate this principle, another thread could obtain a reference to the Actor object and execute public methods illegally.

```scala
  case GetThis =>
    sender ! this
```

Listing 15: Breaking Akka’s encapsulation.

Joelle

Joelle has been described in sufficient detail already. This section will only cover some differences between Joelle and the other two
tools, Erlang and Scala/Akka.

Races

Joelle is intended to be a race free language from the start and gives (assuming absence of bugs) this guarantee. If a program in Joelle compiles, it is race free. Akka manages to uphold race freedom only under assumption of disciplined users, while Erlang guarantees race freedom through immutability.

Explicit / Implicit Parallelism

Akka and Erlang are based on message passing – they make parallelism explicit while Joelle’s approach is hiding it: the user never explicitly expresses when and how a message should be executed, the runtime system will figure out a safe way to run the code. This has obvious advantages, for instance that parallelism emerges with low development overhead, but this also might lead to some disadvantages: programmers might feel that it is appropriate to use mutable state as freely as it would be used in a single threaded program. Even though this is safe, it leads to poor internal parallelisation through the introduction of Read-Write effects.

Distributed Computing

Erlang and Akka are both proven tools for building reliable distributed systems – systems that can continue service “as good as possible” even in the presence of hardware failure and network
outage. Joelle, being a young research language with multicore programming on one machine in mind has nothing to offer in comparison.

3.3 The Tests

The benchmarks, introduced a bit lower, ran over a wide range of parameters, the visualisations will compare the tools in terms of scalability and absolute runtime. The scalability, or speedup, measures how much faster the solution is with $N$ threads than with 1 thread. The speedup $S_N = \frac{T_1}{T_N}$ is retrieved by dividing the runtime with one core $T_1$ by the runtime with $N$ cores $T_N$.

Data Processing Benchmark

This benchmark builds a chain of chain length active objects (or processes in Erlang/actors in Scala). The active objects each can receive a message containing a single list of double values. There are two kinds of active objects in the system, one kind sorts the list before passing it on to the next active object in the chain, the other kind reverses the list before passing it on to the next element in the list. These two kinds are in the chain in an alternating order.

The benchmark then creates 500 lists of random doubles, each of these lists is set size elements long.

As soon as the benchmark is initialised in this way, the timing
starts and the actual task is executed: all the data lists are passed
to the first active object, followed by a stop message. The timing
stops when all messages and the stop message are received.

Figure 3.1 shows an instance of this benchmark.

For sorting, the fastest available algorithm in the languages stan-
dard API was used, for Erlang, this was lists:sort, for Java, it’s
java.util.Collections.sort and for Scala, it is
scala.util.Sorting.quickSort.

The questions this benchmark intends to answer are:

1. How large do tasks have to be in order to achieve speedups
   in the different systems?

2. How does contention of the submit operation affect perfor-
   mance?

3. Is internal parallelisation giving Joelle benefits over purely
   sequential processing?

4. For this application, how do the absolute runtimes compare?

Benchmark Parameters

For all tools, the number of threads is varied between 1 and the
number of native threads (1..4 on the PC and 1..64 on the Server).

Also, we vary the chain length to check whether the latency (the
speed with which a message moves through the chain) is not out
of the ordinary in comparison with the other tools. After all, it
Parameter | Values
--- | ---
threads | \([1, 2, \ldots, 4] \quad \rightarrow \quad [1, 2, \ldots, 64]\)
chain length | \([2, 12, 102, 1002]\)
set size | \([0, 250, 500, 750, 1000]\)
set count | \([500]\)

Table 3.1: The used benchmarking parameters.

might be the case that memory effects delay the passing on of messages between objects.

The number of sets we send through the chain is fixed, this is mostly in order to save time on benchmarking runs.

A summary of the parameters is available in Table 3.1.

---

![Data Processing Benchmark](image)

**Figure 3.1:** The Data Processing Benchmark for chainLength=4, setSize=3, setSize=4, without a bottleneck.

---

**Data Processing Benchmark With Bottleneck**

This benchmark adds one additional actor and one additional message type to the picture: every time an actor receives a data-message, it will send a count-message (labelled with \(+1\) in Figure 3.2 to a central counter-actor: this actor will increase an internal counter by one upon execution of the \(+1\) message.
We expected this test to be especially hard for Joelle’s scheduler because all the actors in the chain submit to the same counter actor which will put the synchronised section in \texttt{submit} under high contention. Since Joelle’s submit mechanism is considerably more complex through the adding of barriers, we expected our implementation to suffer worse problems than Erlang and Akka.

![Diagram](image)

\textbf{Figure 3.2:} The \textit{Data Processing Benchmark} for chainLength=4, setSize=3, setCount=4, with a bottleneck.

\section*{Results}

This section will analyse the results obtained on both platforms when executing the benchmarks in Erlang, Akka, and Joelle with different numbers of worker threads.

\subsection*{How large do tasks have to be in order to achieve speedups?}

Looking at the counter=False scenarios, speedups are achieved for all sizes on the PC, and on all sizes from 250 on the Server. A
smaller task will have the same amount of overhead for scheduling as a larger task. This makes smaller tasks’ work-to-overhead ratio smaller. This hinders scalability for small tasks.

For the counter=True scenarios, there are two effects influencing scalability: additionally to the work-to-overhead ratio, a smaller task size means that more messages per unit of time will be sent to the counter. Therefore, smaller tasks have another performance hindering effect in this scenario. It is noteworthy that even though this affects all tools (and Joelle the worst), Erlang fares the best: it’s maximum speedup is almost the same as for the counter=False scenario. This should not be underestimated: it might be the case that submitting is not Erlang’s limiting factor and future improvement on the general scalability is possible. Accounting for the fact that Akka is a mature tool and still suffers from this network topology could suggest that this effect is here to stay. In practise, this would mean that designing network topologies to avoid this kind of problem is necessary. Further work on Joelle’s mailbox could lessen this effect but is unlikely to remove it.

With the counter, Joelle scales poorly or not on the Server and for set sizes of at least 250 on the PC. Without the counter, Joelle scales up with set sizes of at least 250 on the Server, in the PC, even empty messages lead to speedups.

How does contention of the submit operation affect performance? Looking at the counter=False speedups, we see distinct peaks on the Server, after those the speedup goes down again. We attribute these peaks to high contention when many messages are submitted at the same time: at one point, the addition of
threads does not increase the number of submits per second any more but it does increase the negative effects of the cache protocol\textsuperscript{12} – performance goes down. On the PC, we do not see a peak, the submit section seems small enough to not pose an issue there. On PC, even with the counter, Joelle compares well to Erlang and Akka, outscaling both with low chain lengths (due to internal parallelization!) and scaling almost as well on long chains. Contention on submit is, however, clearly an issue on the Server: Joelle performs well on the Server without the counter but loses a lot of its performance, once the counter is added. This is not surprising, since up to 64 threads can contend for submitting +1 messages there.

Joelle does not cope well with the counter enabled on the Server; it does well in all other cases.

Is internal parallelisation giving Joelle benefits over purely sequential processing? Assuming that internal parallelisation does work, we would see bigger speedups than Erlang and Akka for the short chains: sequential processing (as in Erlang and Akka) achieves speedups through the pipeline-like topology of the benchmark. If the chain is \( N \) elements long, under sequential processing there can be no more than \( N \) messages processed at any given time. Under internal parallelisation, we can process more than the number of actors. Once the chain length exceeds the number of cores, this effect should vanish quickly, since then sequential processing is able to use all cores as well.

This effect is confirmed by looking at the top row in Figure 3.3: Joelle achieves speedups for the small chain length while the other
platforms fall behind (not including set size 0).

*Internal parallelization is giving Joelle benefits over purely sequential processing.*

**For this limited application, how do the absolute runtimes compare?** For most scenarios, Joelle wins the crown in terms of absolute runtimes. An exception is the benchmark with the counter enabled on the Server in Figure 3.9, where Akka is faster on some of the runs.

Of course, not all of this can be attributed to our implementation, we want to remind the reader that the implementations of the sorting algorithms differ, for instance.

*For all benchmarks on PC and for most benchmarks on Server, Joelle is the fastest tool.*
Figure 3.3: Speedups on the Server with counter=False. Higher is better. Joelle shows good scalability on average but tends to be more sensitive to the task size than Erlang and Akka. The high scalability in the top row is due to successful internal parallelisation.
### 3.3. The Tests

#### Figure 3.4: Speedups on the PC with counter=False. Higher is better. All tools do a good job in terms of scalability. Contrary to the same test on the Server, Joelle deals the best with small set sizes. Interesting are some very high speedups for Akka and Joelle in the top row. We do not attempt to explain those.
Figure 3.5: Speedups on the Server with counter=True. Higher is better. The counter is a serious problem for all tools: even though not adding much actual work, scalability is suffering for all tools. Erlang loses the least performance, however.
3.3. The Tests

Figure 3.6: Speedups on the PC with counter=True. Higher is better
Figure 3.7: Runtimes on the Server with counter=False. Y-Axis in milliseconds. Lower is better.
Figure 3.8: Runtimes on the PC with counter=False. Y-Axis in milliseconds. Lower is better.
3. **Performance Evaluation**

Figure 3.9: Runtimes on the Server with counter=True. Y-Axis in milliseconds. Lower is better.
3.3. The Tests

Figure 3.10: Runtimes on the PC with counter= True. Y-Axis in milliseconds. Lower is better.
Conclusion

The performance evaluation suggests that we achieved the goal of being in the same ballpark range as Erlang and Scala. To some, our numbers might even suggest that we surpassed it. However, we explicitly reject any such claims: even though there was no conscious effort to optimize Joelle further than Erlang and Scala, further tests, carried out by developers not part of the Joelle team would be necessary to justify such claims.

In our benchmarking scenario, Joelle scales better than or roughly equal to Erlang and Scala on the quad core machine. On the server with 64 threads it scales better for short chains (that require internal parallelisation for achieving speedups) and worse than Erlang and Scala for longer chains. The fact that longer chains are a weakness is hardly a surprise, given the more complex submitting operation.

If we look at absolute runtimes, we see that for most test scenarios, Joelle is ahead of the pack. We attribute this to Java’s high performance. Figure 3.9 on page 84 shows a situation where the total runtime of the program is barely dependent on the task size.
4. Conclusion

This suggests that most of the time is spent with scheduling and synchronisation overhead. Even though this is – to some degree – also the case for Erlang and Scala, Joelle suffers the worst. In part, this is justifiable: the mailboxes in Joelle have a more complex task to accomplish than those in Erlang and Scala. Still, we believe that further improvement of our mailboxes, for instance by using more sophisticated locking techniques could prove beneficial.
Chapter 5

Future Work

It is not clear, what the “normal” application of a Joelle program is – in order to assess the urgency of optimisations of the multi queue scheduler, a usage study of Joelle would be helpful.

Smarter locking strategies for the mailboxes might improve performance for small task sizes and mailboxes under high contention (see the counter=True benchmarking scenarios).

The interplay of ombudsmen and active objects might introduce dynamically changing dependencies. This could force the mailboxes to change as well but depends on the further course of research.

During the course of this thesis and partly initiated by the author, talks with Typesafe (the main developer of Akka) happened, planning a possible cooperation between Uppsala University (the group under Tobias Wrigstad) and Typesafe, focusing on statically guaranteeing encapsulation for Actors. This might be the first step to combine Akka’s considerable wealth of features with guaranteed race freedom.
## Visual Glossary

<table>
<thead>
<tr>
<th>Description</th>
<th>Symbol</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>----→</td>
<td></td>
</tr>
<tr>
<td>Illegal Reference</td>
<td>--→</td>
<td></td>
</tr>
<tr>
<td>A reads B</td>
<td>A ←→ B</td>
<td></td>
</tr>
<tr>
<td>A <strong>READS-WRITES</strong> B</td>
<td>A ←→ B</td>
<td>the active side of the process is obvious from the context</td>
</tr>
<tr>
<td>Message flow (no future returned)</td>
<td>→→</td>
<td>messages sent to the right</td>
</tr>
<tr>
<td>Message flow (future returned)</td>
<td>←→</td>
<td></td>
</tr>
<tr>
<td>Object, contents omitted</td>
<td>● Name</td>
<td></td>
</tr>
<tr>
<td>Object with region “R” and method</td>
<td></td>
<td>an object can contain other objects</td>
</tr>
<tr>
<td>Active object</td>
<td></td>
<td>an object containing a thread</td>
</tr>
</tbody>
</table>
This section contains UML Diagrams for the scheduler implementations for further reference.
B. UML Diagrams

B.1 | DagScheduler

```
<<abstract>>
JoelleScheduler

DagScheduler
- activeDeps : Map<Class,
  Set<DagTask>>
+ submit(message : Message<T>) :
  Future<T>

DagTask<T>
- task : FutureTask<T>
- msg : Message<T>
- owner : DagScheduler
- successors : Collection<DagTask<T>>
- predecessors : AtomicInteger
+ getFuture() : Future<T>
+ isBlocked() : Boolean
+ directlyPrecedes() : Boolean
+ run() : void

<<interface>>
Runnable
```
B.2 | MultiQueueScheduler

<<abstract>>
JoelleScheduler

MultiQueueScheduler
- inboxes : Map<Class, Deque<FutureMessage>>
+ submit(message : Message<T>) :
  Future<T>
- flush(Deque<FutureMessage> key) : void

FutureMessage<T>
- task : FutureTask<T>
+ run() : void
+ getFuture() : Future<T>
+ getMessage() : Message<T>

Barrier

Input: Message message
Output: Future

synchronized this
  synchronized queue_class(message)
    append(message, queue_class(message))
  end
  foreach dep ∈ dependencies(msg) do
    append(Barrier, queue_dep)
  end
end
flush(class(message))
return future(message)
C.1 Scala

tick.scala

case object Tick       // declare the Tick message
case object GetCount   // declare the GetCount message
class TickActor extends Actor {
  var Count = 0
  def receive = {
    case Tick =>
      Count = Count + 1 // mutation of internal state!
    case GetCount =>
      sender ! Count     // reply to the sender of the message
  }
}

Listing 16: A usage example for Scala. Declaring an actor.
object TickApp extends App {
  val system = ActorSystem("TickSystem") //create an ActorSystem
  //start a TickActor:
  val tickActor = system.actorOf(Props[TickActor], name="MainActor")
  tickActor ! Tick //send a Tick messages via the '!' operator
  implicit val timeout = Timeout(5 seconds)
  val count = tickActor ? GetCount // the ask pattern '?' returns a future
  val result = Await.result(count, timeout.duration).asInstanceOf[Int]
  println("received Count: " + result)
  system.shutdown
}

Listing 17: A usage example for Scala. Using the actor.
Bibliography


C. Code

