Chocola: Integrating Futures, Actors, and Transactions

Janwillem Swalens
Software Languages Lab
Vrije Universiteit Brussel
Belgium
jswalens@vub.be

Joeri De Koster
Software Languages Lab
Vrije Universiteit Brussel
Belgium
jdekoste@vub.be

Wolfgang De Meuter
Software Languages Lab
Vrije Universiteit Brussel
Belgium
wdmeuter@vub.be

Abstract

Developers often combine different concurrency models in a single program, in each part of the program using the model that fits best. Many programming languages, such as Clojure, Scala, and Haskell, cater to this need by supporting different concurrency models. However, they are often combined in an ad hoc way and the semantics of the combination is not always well defined.

This paper studies the combination of three concurrency models: futures, actors, and transactions. We show that a naive combination of these models invalidates the guarantees they normally provide, thereby breaking the assumptions of developers. Hence, we present **Chocola**: a unified framework of futures, actors, and transactions that maintains the guarantees of all models wherever possible, even when they are combined. We present the semantics of this model and its implementation in Clojure, and have evaluated its performance and expressivity using three benchmark applications.

CCS Concepts • **Software and its engineering → Concurrent programming languages**; **Parallel programming languages**; **Concurrent programming structures**; **Multiparadigm languages**;

Keywords concurrency, parallelism, futures, actors, transactions, Software Transactional Memory

ACM Reference Format:
Janwillem Swalens, Joeri De Koster, and Wolfgang De Meuter. 2018. Chocola: Integrating Futures, Actors, and Transactions. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE '18), November 5, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3281366.3281373

AGERE '18, November 5, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6066-1/18/11…$15.00
https://doi.org/10.1145/3281366.3281373

1 Introduction

Since the introduction of multicore processors, concurrency and parallelism have become crucial aspects of software development. Over the past decade, many researchers have revisited old and invented new concurrency models. A **concurrency model** provides constructs to introduce parallelism into a program. At the same time, it imposes restrictions on the program, in order to provide **guarantees** to the programmer that prevent common errors. For example, the actor model introduces actors that carry out computations in parallel, but it requires that data is shared using messages, thereby preventing low-leves data races.

Today, several of these concurrency models have found their way into modern mainstream programming languages. These languages and frameworks often support many different models. For instance, Clojure has constructs for no less than six concurrency models: futures, promises, atomic variables, transactional memory, channels, and agents; Java supports futures, promises, Fork/Join, parallel collections, threads, locks, and atomic variables; and Haskell supports threads, locks, atomic variables, transactions, and channels.

Tasharofi et al. [21] have shown that developers effectively combine multiple models in a single program: in a sample of 15 GitHub projects in Scala that use the actor model, 12 combined it with another model (illustrated in Figure 1). 8 out of 15 programs used actors and futures and 10 out of 15 programs actors and threads, including 6 which used all three models. Only 3 out of the 15 programs used only actors.

Unfortunately, concurrency models are often integrated in an ad hoc way and the semantics of their combination is not always well defined. We observe that *when the language constructs of different concurrency models are combined, their original guarantees are often invalidated.* In a case study of Clojure, we found several such cases [20]: for instance, when

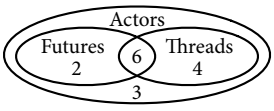

Figure 1. Results of a study by Tasharofi et al. [21]: out of 15 Scala projects that use actors, 12 combine it with futures and/or threads.

a message is sent over a channel in a transaction, and the transaction rolls back, the message is not retracted. Similarly, when communication over channels is combined with another model, deadlocks become possible.

We argue that the combination and integration of multiple concurrency models must be carefully considered. Most concurrency models fall into one of three categories: deterministic, message-passing, and shared-memory models [22]. In this paper, we present Chocola (for **co**mposable **co**ncurrency **la**nguage), a programming language that integrates three concurrency models, one from each category: futures, actors, and transactions. The goal of Chocola is to combine these three models and maintain the guarantees of each model wherever possible.

In this paper, we first describe the three concurrency models we studied (Section 2). Next, we discuss the problems that occur when these models are combined naively (Section 3). Then, we look at each pairwise combination in detail and define a semantics that satisfies our requirements, thereby defining transactional actors (Section 4), transactional futures (Section 5), and futures for intra-actor parallelism (Section 6). Afterwards, we describe how these pairwise combinations are integrated into Chocola (Section 7). We describe Chocola's semantics and implementation (Section 8) and evaluate its performance and expressivity using three benchmark applications (Section 9).

In previous work we studied two pairwise combinations in detail – transactional futures [18] and transactional actors [19]. The contribution of this paper is their integration into a single framework that unites all three models in a coherent manner. Sections 4 and 5 thus correspond to previous work, while the subsequent sections describe new work.

2 Futures, Actors, and Transactions

In this section, we present the three models we studied: futures (Section 2.1), actors (Section 2.2), and transactions (Section 2.3). For each model, we give a brief description and list its constructs. We illustrate each model using the same running example: a holiday reservation system that books flights and hotels. We also describe the guarantees provided by each model.

Chocola is built as a fork of Clojure, a Lisp-like language that runs on top of the Java Virtual Machine.[1] Hence, Chocola's syntax and its built-in functions are the same as Clojure's. Clojure supports futures and transactions, which Chocola reuses and extends to support actors.

2.1 Futures

A parallel **task** (or thread) is a fragment of the program that can be executed in parallel with the rest of the program. A parallel task can be created using the expression `(fork e)`. This begins the evaluation of the expression `e` in a new task, and immediately returns a future. A **future** is a placeholder variable that represents the result of a concurrent computation [2, 9]. Initially, the future is **unresolved**. Once the parallel evaluation of `e` yields a value `v`, the future is said to be **resolved** to `v`. This result can be retrieved by other tasks by calling `(join f)`. If the future is resolved, `join` returns its value immediately; if the future is still unresolved, this call will block until it is resolved and then return its value.

In the following example, futures are used to filter the list `xs` in parallel, so that only the elements for which `(f x)` returns true are returned:

```
(defn parallel-filter [f xs]
  (let [parts   ; Divide xs into 8 parts
          (partition 8 xs)
        futures ; Fork a future for each part, in which the built-in
                ; (sequential) filter is applied
          (map (fn [part] (fork (filter f part))) parts)
        results ; Join futures
          (map (fn [fut] (join fut)) futures)]
    (apply concat results))) ; Concatenate results of each part
```

Futures guarantee **determinacy** (Det): for a given input, a program always produces the same output. This means that a program always has the same result, no matter in which order its tasks are interleaved. Futures are commonly used to parallelize homogenous operations over lists, e.g. searching and sorting [9], as in the example above. In these cases, determinacy is often desired as the end result should not depend on how tasks are scheduled.

2.2 Actors

The actor model is a message-passing model that was originally introduced by Hewitt et al. [12] and later revised by Agha [1]. Actors run concurrently and can receive messages. In response to a message, an actor can send messages to other actors, spawn new actors, and change its own state. In the actor model, messages are sent asynchronously.

In Chocola, we use a "classic actors" model [6], in which an **actor** consists of three elements: an address, an inbox, and a behavior. Each actor has a unique and immutable **address**, used to send it messages. Its **inbox** is a queue of messages. In our model, a message is simply a tuple of values. Finally, a **behavior** specifies how an actor responds to a message. It is parameterized by two types of parameters: the internal state of the actor and the values of the received message.

In Chocola, a behavior is defined as follows:

```
(def travel-agent-behavior
  (behavior [flights hotels] [orig dest n]
    (let [; Search outbound and return flights
          outbound (search-flight flights orig dest)
          return   (search-flight flights dest orig)
          ; Search for a hotel
          hotel    (search-hotel  hotels dest)
          ; Reserve n seats on these flights
          flights1 (reserve-seats flights outbound n)
          flights2 (reserve-seats flights1 return n)
          ; Reserve a room for n people in the hotel
          hotels1  (reserve-room hotels hotel n)]
      (become travel-agent-behavior flights2 hotels1))))
```

[1] https://clojure.org

This behavior specifies an actor that represents a travel agent. Messages can be sent to this actor to reserve a holiday, consisting of an outbound and a return flight and a hotel room. The behavior of an actor defines how it responds to an incoming message. A behavior is parameterized by two types of parameters: first, the internal state of the actor (here, `flights` and `hotels`, maps containing the flights and hotels), second, the values of the received message (here, the details of a reservation: its origin `orig`, destination `dest`, and the number `n` of seats/beds to reserve).

An actor can be spawned using `spawn`:

```
14 (def flights
15   {"BA212" {:from "BOS" :to "LHR" :price 499 :seats 243}
16    "BA213" {:from "LHR" :to "BOS" :price 499 …}})
17 (def hotels {"Hilton" {:in "BOS" :price 100 :rooms 300}})
18 (def agent (spawn travel-agent-behavior flights hotels))
```

This creates a new actor with `travel-agent-behavior` as initial behavior and the maps of flights and hotels as internal state. `spawn` returns the address of the new actor.

`(send agent "LHR" "BOS" 3)` sends a message to this actor: it puts a message containing the values `"LHR"`, `"BOS"`, and `3` in the inbox of the actor with address `agent`. When the receiving actor processes the message, it executes the code in the behavior defined above (lines 3–13), with `flights` and `hotels` bound to the values given when the actor was spawned and `orig`, `dest`, and `n` bound to the message's values.

An actor can change its behavior and internal state using `become`. On line 13 in the example, `become` updates the `agent` actor, keeping its behavior identical but updating its internal state to the new maps of flights and hotels, in which three seats on a flight and a room with three beds in a hotel were reserved.

An actor alternates between two states: ready to accept a message, or busy processing a message. A **turn** is the processing of a single message by an actor, that is, the process of an actor taking a message from its inbox and processing that message to completion [6].

The actor model provides two useful guarantees. First, it guarantees that programs are free from races within turns: this is called the **isolated turn principle** (ITP). Hence, developers do not need to care how individual instructions within a turn are interleaved with those from other actors; instead, they can reason about their program at the level of turns. Second, the actor model guarantees **deadlock freedom** (DLF): as there are no blocking operations, an actor can never deadlock. These two guarantees make the program easier to understand, reason about, and debug.

2.3 Transactions

Software Transactional Memory (STM) is a concurrency model that allows multiple parallel tasks to access shared memory locations [11, 17]. To use STM, each shared memory location is encapsulated in a **transactional variable**. Access to shared memory can only occur in a **transaction**: a block of code in which transactional variables can be read and modified. In a transaction, the developer has a consistent view of the shared memory: reading a transactional variable multiple times in the same transaction always yields the same result, even if another task modified it concurrently. Furthermore, all changes made to shared memory in a transaction are made visible to other tasks atomically: it is not possible for other tasks to observe intermediate states.

In contrast to mechanisms based on locking, which are said to be pessimistic, STM is optimistic [10]: the code in a transaction is immediately executed, without taking locks. When different transactions attempt to access the same transactional variable(s), they **conflict**, which causes the transaction to **abort**. An aborted transaction is retried, which means that its changes are discarded or rolled back and its contents are reexecuted. When no conflicts occur, a transaction can **commit** successfully.

A transactional variable is created using `ref v`, containing the initial value `v`. A transaction is a block `(atomic e)` that encapsulates an expression, which can contain reads `(deref r)`, abbreviated to `@r`, and writes `(ref-set r v)` on the shared memory locations.

The code below implements a flight reservation system using transactions. Lines 1–3 define a map of all flights, in which each flight is encapsulated in a transactional variable. Lines 11–13 contain a transaction that reserves three seats on two flights. This consists of reading the flight and updating its number of available seats. By encapsulating both reservations in the same transaction, we ensure they either both succeed or both fail.

```
1 (def flights
2   {"BA212" (ref {:from "BOS" :to "LHR" … :seats 243})
3    "BA213" (ref {:from "LHR" :to "BOS" … :seats 243})})
4
5 (defn reserve-seats [flight n]
6   (let [new-seats  (- (get @flight :seats) n)
7         ; In flight map, associate key :seats with new value
8         new-flight (assoc @flight :seats new-seats)]
9     (ref-set flight new-flight)))
10
11 (atomic
12   (reserve-seats (get flights "BA213") 3)
13   (reserve-seats (get flights "BA212") 3))
```

Transactional systems provide two useful guarantees.

First, **isolation** (Iso) ensures one transaction can never see the changes of another until the latter has committed. Different isolation 'levels' have been defined, here, we focus on *serializability*. Serializability requires that the result of a transactional program must always be equal to the result of *a* serial execution of the program. In the example, this entails that the values of the transactional variable on lines 6 and 7 must be equal, even if another thread modified them concurrently. Thanks to isolation, the developer can reason about the program at the level of transactions: when transactions execute in parallel, it does not matter in which order their instructions are interleaved, only in which order the

transactions are committed. This makes the program easier to understand and debug.

Second, transactional systems guarantee **progress** (Pro). While traditional locking systems are prone to issues such as deadlocks, livelocks, and starvation, transactional systems aim to free the programmer from worrying about these issues. Similar to isolation levels, different STMs provide one of a range of different 'progress guarantees'. Most systems guarantee *deadlock freedom* [10]: when two transactions conflict, progress is guaranteed by a contention manager, a mechanism that decides which transaction(s) to delay so that another can make progress.

3 Combinations of Models

Chocola combines futures, actors, and transactions into a unified model. The goal is to find a suitable semantics for the unified model, even when concurrency models are combined. We define two requirements:

1. First, **the semantics of the separate models should remain unchanged**, so that programs that do not use combinations work unchanged.
2. Second, **the guarantees of all models should be maintained even when they are combined when possible**. In some cases it is impossible to combine the guarantees of all models because they inherently conflict. For instance, when a non-deterministic model is used in a deterministic one, it is impossible to maintain determinism. In this case, we will need to relinquish one of the original guarantees and define a modified, less restrictive guarantee that is provided by our combination.

Figure 2 tabulates the 9 (3×3) pairwise combinations of the three models, visualizing the discussion in the rest of the paper. In each cell, we study how one model can be nested in another: we list the guarantees of the two models as described in the previous section, and using the colors we indicate which guarantees are valid in a naive combination and in Chocola:

- Guarantees in blue are valid, even in a naive combination. No changes to the semantics are needed.
- Guarantees in green are broken in a naive combination, but maintained in Chocola.
- Guarantees in ~~red~~ are inevitably broken, in a naive combination as well as in Chocola. This happens in two cases and is the result of embedding a non-deterministic model in a deterministic one.
- Using ~~Det~~ → ITD, we indicate that a guarantee (here determinacy) is broken in a naive combination and cannot be maintained by Chocola. Instead, we defined a less restrictive guarantee that can be upheld (here intratransaction determinacy).

We consider the *dynamic extent* of each construct: if one model is used in another at execution time, we say they are (dynamically) nested. This does *not* necessarily require their constructs to be nested *lexically*. For instance, if a library function that uses futures is called in a transaction, the construct `fork` will not appear in the `atomic` block in the code (lexically), but at execution time a future will be created while a transaction is running (dynamically). In the rest of the paper, whenever we say that two constructs are nested, we refer to this type of *dynamic* nesting.

Note that there is a sort of 'anti-symmetry' in the table. The diagonal contains models nested in themselves. All other cells have an opposite across the diagonal, e.g. the top-right cell represents actors in futures while the bottom-left cell represents futures in actors.

We discuss the cells on the diagonal, in which each model is nested in itself, in the following section. These 'trivial' combinations all maintain the guarantees. In the other cells, when *different* models are combined naively, the guarantees are broken. These cases are discussed in the subsequent sections. Each of these discussions will first describe the problems that arise when the models are combined and then offer a modified semantics that satisfies our requirements.

3.1 Trivial Combinations

We briefly discuss the combinations on the diagonal of Figure 2, in which each model is nested in itself. These combinations have been studied in existing literature and maintain the model's guarantees.

Nested Futures Forking one parallel task in another is common and expected in programs that use futures. Nesting futures does not break the determinacy of the program: no matter where futures are introduced, the program remains equivalent to the same program without futures.

Nested Actors 'Nesting' actors – creating one actor in another – is a standard part of the actor model. In fact, any actor program consists of only actors running concurrently, and therefore all actors except the initial one are nested actors. The guarantees of actors are maintained.

Nested Transactions When a transaction is started in a task in which another transaction is already running, this is a *nested transaction*. The nesting of transactions is a well-studied problem [10]. Moss and Hosking [16] distinguish two types of nesting: open and closed nesting. While open nesting enables better performance, it is complex to use and breaks the isolation of the outer transaction. Closed nesting is simpler, and in practice it is the norm: Clojure, Haskell, and ScalaSTM all implement closed nesting. Chocola therefore does so too.

4 Transactional Actors for Sharing Memory Between Actors

In this section, we study the combination of transactions and actors in Chocola. We first discuss the use of transactions in actors and next the use of actors in transactions. This

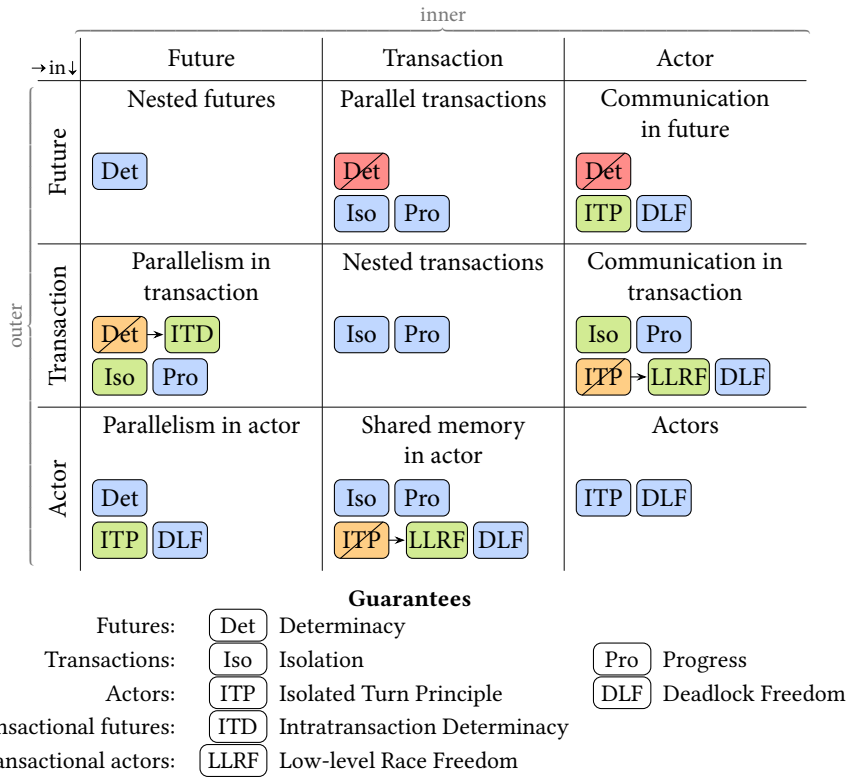

Figure 2. The pairwise combinations of the three models studied in this paper and their guarantees.

results in *transactional actors*, which were first introduced by Swalens et al. [19].

4.1 Using Transactions in an Actor to Safely Share Memory

In the travel agent example of Section 2.2, each agent has its own separate set of flights and hotels, which it searches and modifies to make a reservation. However, in a typical reservation system we would like multiple travel agents to access and modify the same flights and hotels. We would thus like to introduce shared memory in an actor system.

We discuss how this is currently achieved in two types of actor systems: pure and impure systems [5]. *Pure actor systems* enforce strict isolation between actors. Developers often introduce shared memory using several patterns [5]. While the guarantees of the actor model are maintained, the difficulty of preventing race conditions and deadlocks is pushed entirely to the developer. *Impure actor systems* do not enforce strict isolation, so developers can use the underlying shared-memory model of the language [21]. However, this breaks the isolated turn principle.

In both cases, representing shared state in an actor system is complex and error prone, and it is the responsibility of the developer to ensure correct access to the shared memory.

We introduced transactional actors, which maintain the expected guarantees by sharing memory between actors using transactions. STM guarantees the absence of low-level races by encapsulating atomic sections in a transaction. Moreover, in contrast to traditional locking, STM guarantees the absence of deadlocks. Thus, using STM, memory can safely be shared between actors.

We apply this to the travel agent example by sharing the flights and hotels between different actors using transactional memory. Accesses to shared memory are protected using a transaction:

```
(def flights
  {"BA212" (ref {:from "BOS" :to "LHR" :seats 243})
   "BA213" (ref {:from "LHR" :to "BOS" :seats 243})})
(def hotels …)

(def travel-agent-behavior
  (behavior [] [orig dest n]
    (atomic
      (let [outbound (search-flight flights orig dest)
            return   (search-flight flights dest orig)
            hotel    (search-hotel  hotels dest)]
        (reserve-seats outbound n)
        (reserve-seats return n)
        (reserve-room hotel n)))))
```

4.2 Using Actors in a Transaction to Distribute and Coordinate Work

The performance of the example of the previous section can be improved by processing the three reservations in parallel. The code snippet below implements this: it creates separate

actors to process flight and hotel reservations and the travel agent now sends messages to these actors to process each item in the reservation concurrently.

```
(def airline-behavior
  (behavior [] [orig dest n]
    (atomic
      (let [flight (search-flight flights orig dest)]
        (reserve-seats flight n)))))
(def airline (spawn airline-behavior))

(def travel-agent-behavior
  (behavior [] [orig dest n]
    (atomic
      (send airline orig dest n)
      (send airline dest orig n)
      (send hotel dest n))))
```

In a naive combination of transactions and actors, this program will not work correctly. The transaction in `travel-agent-behavior` sends three messages, but if the transaction aborts the messages are not rolled back. Messages can thus be sent multiple times, breaking the isolation of the transaction.

Transactional actors solve this issue by making any effects on actors that occur in a transaction part of the transaction. Actors provide four constructs. For each, we consider how it can safely be nested in a transaction:

`behavior` Defining a behavior in a transaction is no problem as this operation has no side effects. A behavior can refer to variables in its lexical scope – it is essentially a closure – but will run at a later time and thus does not have access to the encapsulating transaction.

`spawn` Spawning an actor is an effect that must be part of the transaction. As it is costly, we delay it until the transaction commits. This ensures that the transaction's isolation is maintained and the creation cost is only paid once.

`become` Become is a construct that is delayed by construction: its effect only takes place upon the start of a new turn. As a transaction cannot span multiple turns, the transaction will always be committed before the effect of `become` is made visible, maintaining isolation.

`send` As illustrated above, sending a message in a transaction has effects that must be rolled back when the transaction aborts. This implies that a message may now need to be retracted: when the transaction it was sent in aborts, the message and its effects need to be 'unsent'. We say that messages sent from within a transaction have a **dependency** on the transaction. There are now two types of messages: those sent when no transaction is active in the sender have no dependency and are **definitive**, those sent from within a transaction have a dependency and are **tentative**.

This has an impact on the receiver of a tentative message. When an actor takes a tentative message from its inbox, the turn that processes it also becomes tentative: the message is processed, but the effects it causes should not be persisted yet. Even though this turn is not a transaction, it executes in the same 'tentative' manner, as its effects can roll back. When a **tentative turn** ends, the actor waits until the transaction on which it depends has committed. After a successful commit of its dependency, the actor can continue to its next turn, and we say the turn was successful. If its dependency aborts, the tentative turn fails and its effects are discarded. The actor then processes the next message in its inbox as if nothing happened.

With these changes to the semantics of actors' constructs, the example above now works as expected. The messages sent on lines 11–13 are tentative. When they are processed by the `airline` actor, the turn on lines 3–5 is tentative. The effects of this turn, i.e. the reservation of the seats on the flights, are only persisted if the original transaction succeeds. If the original transaction aborts, its effects as well as the effects of its dependent transactions are discarded.

5 Transactional Futures for Parallelism in Transactions

In this section, we study the combination of transactions and futures in Chocola. Creating transactions in futures is standard in languages with transactions: as the transactional model does not provide any constructs to create threads, it must rely on another model to do so. Hence, we focus on the opposite combination: the creation of futures in a transaction. Using an example, we show that the futures can be used in a transaction to introduce more fine-grained parallelism. This breaks the isolation of the transactions in a naive combination. Hence, we introduce *transactional futures*: futures created in a transaction with access to the encompassing transaction's context. Transactional futures maintain the isolation and progress guarantees of transactions. They were first introduced by Swalens et al. [18].

5.1 Motivation and Problems

To demonstrate the use of futures in a transaction, we revisit our running example. It contained a transaction that searches for a flight and reserves seats on that flight. In the code example below, the function `search-flight` is defined. It searches for a suitable flight by filtering the list of all flights based on their trajectory, and returning the first flight that matches.

```
(defn search-flight [flights orig dest]
  (first
    (parallel-filter
      (fn [flight] (and (= (get @flight :from) orig)
                        (= (get @flight :to) dest)))
      (vals flights))) ; vals returns all values of a map

(def airline-behavior ; Same as in previous code snippet
  (behavior [] [orig dest n]
    (atomic
      (let [flight (search-flight flights orig dest)]
        (reserve-seats flight n)))))
```

In this example, we improve the performance of the search process by parallelizing the filter operation. `parallel-filter`,

as defined in Section 2.3, divides the list of flights into partitions that are filtered in parallel, in separate tasks.

However, using a naive combination of futures and transactions this program will not work as expected! In a language like Clojure, a transaction is thread-local, i.e. it is bound to the task it was created in. The tasks created in `parallel-filter` do not have a transaction running when they execute the function on lines 4–5, hence, they will see inconsistent values for the flight. The problem is that futures created in a transaction are not part of that transaction.

5.2 Solution

We solve these issues by defining transactional futures. A transactional future is the future associated with a so-called **transactional task**: a task that is forked while a transaction is running. A transactional task operates within the context of its encapsulating transaction, so that it can access and modify the state of the transaction.

Conceptually, each transactional task creates a copy of the transactional memory, and will access and modify that private copy. This ensures that two tasks can run concurrently without interfering with each other. To this end, a transactional task contains two data structures: a (read-only) **snapshot** containing a conceptual copy of the state of the transactional memory when the task was spawned, and a **local store** containing the changes made to the transactional memory in the task.

Each transaction starts with one *root* task that evaluates the transaction's body. Its snapshot is a copy of the transactional heap; its local store starts empty. When a new task is spawned, its snapshot is the current state of the transactional memory, i.e. the snapshot of its parent task modified with the current local store of the parent. The local store of a newly spawned task is empty. While a task executes, it looks up values in its snapshot and modifies them by storing their new values in the local store. It only uses its own snapshot and local store, ensuring that each task runs in isolation.

When a task finishes its execution, its future is resolved to its final value. When a task is joined for the first time, its local store is merged into the task performing the join,[2] and the value of its future is returned. This way, changes propagate from child tasks to their parent. Subsequent joins of the same task will not repeat this, as their changes are already merged; they only return the final value of the future.

At the end of the transaction, the modifications of all transactional tasks in the transaction should have been merged into the root task, and these are committed atomically. All changes from all tasks are committed in a single step, so the transaction remains an indivisible step to the outside, maintaining its isolation. If a conflict occurs at commit time, the whole transaction is aborted and retried. If a conflict occurs in one of the tasks while the transaction is still running, *all* tasks are aborted and the whole transaction is retried. In other words, the tasks within a transaction are coordinated to either all succeed or all fail: they form one atomic group.

[2]When two tasks modify the same transactional variable, a conflict occurs. To solve this, we allow the developer to specify a *conflict resolution function*, which takes the conflicting values and returns the new value for the variable.

```
1 (def airline-behavior
2   (behavior [] [orig dest n]
3     (fork (book-flight orig dest n))
4     (fork (book-flight dest orig n))
5     (fork (book-hotel dest n))))
```

Figure 3. Illustration of the escaping task problem: the three forked tasks may continue executing after the turn has finished, interleaved with the next turn.

6 Futures in Actors for Intra-actor Parallelism

In this section, we study the combination of futures and actors. Combining actors and futures can be useful. On the one hand, futures can be introduced in an actor to process a turn in parallel: this is **intra-actor parallelism** (the lower-left cell of Figure 2). On the other hand, actors can be used in a program with futures to introduce communication between parallel tasks (the upper-right cell of Figure 2). Furthermore, when a program using one model includes a library that uses the other, the models are combined implicitly.

In a naive combination, two minor problems occur: determinacy and the isolated turn principle can be broken.

Determinacy (Futures) When the actor constructs `send` and `become` are used in futures, determinacy can be broken (the upper-right cell of Figure 2) as they may execute in any order. Breaking determinacy for this combination is inevitable. We argue that this is no problem because this only occurs where the programmer explicitly uses `send` and `become`, constructs of the non-deterministic model. As program using *only* actors can also have a non-deterministic result, the developer should expect non-determinism, whether futures are used or not.

Isolated Turn Principle (Actors) A naive combination of actors and futures breaks the isolated turn principle, because turns can be interleaved. We call this the **escaping task problem**.

Figure 3 illustrates the problem. An actor creates three tasks which are never joined, 'escaping' the task they were created in. As a result, the root task finishes its work and proceeds to the next turn while the child tasks may still be running. The two turns overlap, interleaving the processing of two messages, thus violating the isolated turn principle. This can lead to two unexpected results: (1) if the child task sends a message, it can arrive *after* messages sent in the next turn, and (2) a `become` in an escaped task can still change the behavior of the actor *after* the next turn has already started, with the old behavior.

Fortunately, the isolated turn principle can be reintroduced using a simple requirement: any future created in an actor must be joined before the turn ends. This ensures that only one task (the root task) is running when the turn ends, and that the side effects of all futures created in the turn have occurred and all their effects have been merged into the root task. All tasks end when the turn ends, so no more effects can take place during the next turn. Thus, the isolated turn principle is maintained.

We believe this requirement is not overly restrictive: it only applies when a future is forked in a turn but its result is never used in that turn. Furthermore, a similar requirement existed for transactional futures: we required that all futures created in a transaction were joined before the transaction ends. Thus, both techniques provide a consistent model.

7 Chocola: an Integration of Futures, Actors, and Transactions

As said by Hoare [13], the task of the language designer is "consolidation, not innovation" of features. Accordingly, Chocola does *not* introduce new syntactical constructs, nor does it change the semantics of its constituent models when used separately. The novelty of Chocola is that it defines a semantics of the constructs of these models when they are combined with one another, that aims to maintain the guarantees expected by developers. The guarantees of Chocola were already shown in Figure 2. We discuss them in detail.

7.1 Determinacy and Intratransaction Determinacy (Futures)

When used separately, futures guarantee determinacy. Chocola sometimes breaks this guarantee. We distinguish two cases: when futures are the outer or inner model.

Futures as outer model When futures are the outer model, determinacy is no longer guaranteed (first row in the table). This is inevitable: non-deterministic models introduce non-determinism, even when used in a deterministic model. However, we argue that this is not unexpected because the developer must explicitly use a construct from a non-deterministic model to break determinacy, and it is only in those places that it no longer holds.

Futures as inner model In contrast, when using futures as the inner model (first column in the table), determinacy is expected. For instance, when a library that uses futures is embedded in a program that uses another model, the developer of the library still assumed determinacy.

Using futures in another future or in an actor maintains determinacy. However, in a naive combination of futures and transactions determinacy is broken: transactions are necessarily non-deterministic because the order in which transactions are committed is not deterministic. Instead, Chocola provides a weaker guarantee: determinism *inside* transactions, which we refer to as **intratransaction determinacy**.

Intratransaction determinacy states that, given the initial state of the transactional memory, a transaction must always have the same result, assuming that all conflict resolution functions are determinate.[3] A transaction has two kinds of results: its final value and its effects on transactional memory, both are determinate.

The fact that transactional futures do not introduce non-determinism inside transactions follows from two observations. First, it does not matter in which order the instructions of two tasks are interleaved, as they both work on their own copies of the data. Second, changes made in one task only become visible in another one after an explicit and deterministic `join` statement has been executed, and the join operation is deterministic as long as the conflict resolution function is. As a result, given the state of the transactional memory when a transaction started, it can only have one result.

Intratransaction determinacy makes the behavior in a transaction easier to predict, as developers can trace back the value of a variable by looking where tasks were joined.

7.2 Isolation (Transactions)

Transactions guarantee a form of isolation, such as serializability. When they are used in a future or actor, this guarantee is maintained (second column in the table). On the other hand, when a future is forked or a message is sent within a transaction, a naive combination may break isolation (second row in the table). Chocola maintains isolation even for these problematic combinations by incorporating any side effects into the transaction.

Transactional futures realize this guarantee by making `fork` and `join` a part of the transaction in which they run, instead of an independent side effect. Transactional tasks run within the context of the encapsulating transaction. We require that all tasks created in a transaction are joined before its end, thus all changes made by all tasks in the transaction have been applied to the local store of the root task before the transaction commits. Upon commit, they are applied to the transactional heap in a single atomic step, just as if no futures were created in the transaction.

Transactional actors maintain isolation by only making the effects of tentative messages visible if the transaction in which they were sent succeeds. Hence, upon a successful commit, all effects of a transaction are made visible. If a transaction aborts, all its effects are discarded, including the messages it sent and their effects.

[3] Compare this with the definition of determinacy from Section 2.1, which states that, given an input, a program must always have the same output.

7.3 Isolated Turn Principle and Low-Level Race Freedom (Actors)

The isolated turn principle guarantees that, once a turn started, it will always run to completion, in isolation. This allows developers to reason at the level of turns: it does not matter in which order the individual instructions of different turns are interleaved, only how the turns as blocks are interleaved. The isolated turn principle assumed no shared memory and no internal parallelism, which is obviously no longer true in Chocola; it is therefore impossible to maintain.

However, Chocola still prevents races, as all accesses to shared memory are protected by transactions. While the actor model guaranteed freedom from low-level races by prohibiting shared memory, Chocola allows shared memory but requires it to be encapsulated in a transaction. This extends the actor model with safe, shared memory.

Traditionally, actors guarantee a consistent view of the memory during a turn, as the only memory that can be accessed synchronously is the private memory of the current actor. Transactions guarantee a consistent view of the memory during a transaction. Chocola combines both: during a turn, the actor has a consistent view of its private memory, and during a transaction, it has a consistent view of the shared memory. We call this guarantee **Low-Level Race Freedom**: it is not possible to introduce a race on the private memory of an actor within a turn, or on shared memory within a transaction. At the 'level' of turns and transactions, races are impossible.

7.4 Progress (Transactions) and Deadlock Freedom (Actors)

Transactions guarantee progress and actors guarantee deadlock freedom. Even when these models are combined with others, these guarantees are maintained. This is a result of the fact that Chocola only contains one blocking operation, `join`, which always completes because cyclical dependencies between futures are impossible [18]. This is a result of the lexical scoping of the base language (Clojure): a future can only refer to futures that were created before it, resulting in an ordering of futures that cannot contain cycles.

8 Semantics and Implementation

The semantics of Chocola has been defined in two ways: in a formalization and in a reference implementation.[4]

Due to space constraints, we will not describe the formal operational semantics here, but it can be found online. We also created an executable version of its essential parts using PLT Redex [7], to automatically verify its most significant properties. We plan a formal verification of Chocola's guarantees in future work.

Chocola's implementation is a fork of Clojure. Clojure has built-in support for futures and transactions. First, we extended it to support actors, using a standard implementation of actors. Next, we modified these implementations of the separate models to provide the semantics defined above when models are combined, by making the following changes:

- When a message is sent in a transaction, a dependency pointing to the transaction is attached.
- At the start of a turn, it is marked tentative if the processed message has a dependency.
- `become` and `spawn` in a tentative turn are stored instead of being executed immediately.
- At the end of a tentative turn, the actor waits until the dependency has finished. If it succeeded, its tentative `become` and `spawn` operations are executed; otherwise they are discarded.
- When a transaction is started, it contains one 'root' transactional future. The transaction's data structures (its snapshot, local store, etc.) moved to the transactional future.
- `become` and `spawn` in a transactional future are stored in that future.
- `fork` in a transactional future creates a new transactional future, with its data structures initialized as in Section 5.
- `join` in a transactional future merges its data structures with that of its parent.
- When a transaction is committed:
  - If the current turn is tentative, the transaction waits until the dependency finished. If the dependency succeeded, we can proceed; if it failed the whole turn is discarded.
  - The transaction commits the changes from the root future as before. (All tasks in the transaction and their changes have been merged into the root future already.)
  - Any `become` and `spawn` operations that occurred in the transaction are executed.

9 Evaluation: Performance and Expressivity

We evaluated Chocola by transforming programs that use one concurrency model to introduce another. Our evaluation focuses specifically on programs with transactions, because the combination of actors and futures, as discussed in Section 6, did not pose major issues. The aim of this evaluation is thus to demonstrate that in existing programs that use transactions, additional parallelism can be exploited by introducing futures and actors within transactions, without fundamentally changing the design of the program.

We use the STAMP benchmark suite as a basis [15]: it consists of eight applications that use transactions and is commonly used to compare the performance of transactional systems. These applications are based on real-world scenarios and exhibit a range of characteristics.

We are interested in two characteristics in particular. First, the proportion of the execution time spent in transactions: when most of the program's execution occurs in transactions,

[4] Both are available at http://soft.vub.ac.be/~jswalens/chocola.

Benchmark	Lines of code			Speed-up		Architecture
	original	added	removed	original	Chocola	
Labyrinth	682	78 (11%)	30 (4%)	1.3	2.3	8-core (with HyperThreading)
Bayes	1248	1 (<1%)	1 (<1%)	2.8	3.5	
Vacation2	320	25 (8%)	17 (5%)	2.6	33.2	64-core (no HyperThreading)
Yada	No further parallelization possible without domain expertise					

Figure 4. Expressivity and performance results of introducing Chocola in three benchmarks that contain transactions. We list the number of lines of code that were changed to introduce additional parallelism using futures or actors, and compare the maximal speed-up of the modified version that uses Chocola with the original version that only uses transactions.

to further parallelize these programs, it will be necessary to introduce parallelism *within* the transactions. Second, we look at the transaction length: long-running transactions may benefit from more fine-grained parallelism, as in these cases the benefits of introducing parallelism can outweigh its costs. Four (out of eight) programs in the benchmark suite match these criteria. They are listed in Figure 4.

We first ported the applications from C to Clojure, retaining the design and algorithms of the original. Afterwards, we introduce transactional futures and transactional actors where applicable, by performing the following steps:[5]

1. Using profiling tools, we search for the part of the program that takes the most execution time. In our cases, this is always a loop in a transaction.
2. We try to parallelize this loop. We examine whether there are dependencies between the iterations of the loop:
   - When the iterations are *independent*, we parallelize the loop by processing each iteration in parallel. This occurs in the Bayes and Vacation benchmarks.
   - When there are *dependencies* between the iterations, but the program follows a *standard algorithm* for which a parallel version exists in literature, we replace the sequential algorithm with its parallel equivalent. This occurs in Labyrinth.
   - When there are *dependencies* between the iterations and the program uses a *custom algorithm*, we reach a negative result and do not introduce futures or actors. This is the case for the Yada benchmark. Note that this does not necessarily mean that the loop cannot be parallelized, it may also mean that specific domain expertise is required to parallelize the custom algorithm.

We compare the original and transformed programs using two criteria:

Performance The end goal of introducing additional parallelism in transactions is to increase performance, hence, we compare the maximal speed-up achieved by the original version with that achieved by our transformed version.

Developer effort We assess the effort that is required from the developer to use our techniques by measuring the number of lines of code that were changed in the transformation.

[5]The code of both versions of all benchmarks is available on our website.

The results are shown in Figure 4. The performance results were already discussed in detail in Swalens et al. [18, 19], including more details about the experimental set-up. We summarize these results:

Labyrinth spends almost all of its time in transactions that execute a search algorithm, which can be replaced with a standard parallel equivalent that uses transactional futures. This leads to a speed-up thanks to faster, internally parallel transactions and fewer conflicts.

Bayes spends most of its time in a loop in a transaction that can be trivially parallelized. As there is only limited work available, at a certain point the number of transactions is lower than the number of cores in the machine. Transactional futures allow us to introduce more fine-grained parallelism. This is a matter of changing one line, replacing `for` by `parallel-for`, and increases the maximal speed-up.

Vacation implements an event-based vacation reservation system similar to the running example of this paper, to which actors can be applied naturally. We split the transaction of the original application into smaller transactions that are distributed over different actors. This improves performance by introducing more fine-grained parallelism and by lowering the chance and cost of conflicts.

In future work, we would like to strengthen our evaluation by collecting real-world examples of programs that combine multiple concurrency models or might benefit from doing so. By comparing their ad-hoc implementation with an implementation in Chocola, we can demonstrate the benefits of the consistent semantics of Chocola.

10 Related Work

In previous work, we described literature that focuses on the combinations of transactions and futures [18] and transactions and actors [19]. In this section, we describe more recent work that has appeared since, and work that combines futures and actors.

Transactions and Actors *Pony* [4] allows memory to be shared between actors, using deny capabilities to statically guarantee there is only one writer to a shared memory location and thereby preventing races. Similarly, *Encore* [3] uses capabilities to allow memory to be shared between active

objects. Encore plans to support capabilities for both pessimistic and optimistic concurrency, although at the time of writing the exact semantics have not been defined.

Transactions and Futures Independently from our work, Zeng et al. [23] developed *Java Transactional Futures*. They provide intratransaction determinacy, but moreover also maintain the semantic transparency of futures. As a consequence, when a parent and child future conflict, the child must roll back and retry. This affects performance when these kinds of conflicts are frequent, e.g. in one of our benchmarks (Labyrinth). Because our transactional futures resolve these conflicts instead of rolling back, we expect performance for Labyrinth to be better using our transactional futures. Our transactional futures thus forsake semantic transparency to avoid rollbacks within the transaction.

Futures and Actors ParT [8] extends Encore and Clojure to combine actors and futures. It decomposes actors into tasks to which asynchronous messages are sent whose result is a future, thus not directly supporting actors. However, it does support speculative parallelism, which we may consider as a future addition of Chocola.

Imam and Sarkar [14] combine actors with the async–finish model (AFM), which is similar to futures. They moreover allow a task to escape the actor in which it is spawned (the escaping task problem of Section 6), preventing race conditions by prohibiting these tasks from modifying the actor's internal state. Furthermore, the explicit `finish` construct can be used to coordinate actors by encapsulating them. This is not possible in our system, but a similar coordination mechanism can be implemented by passing messages between the actors.

11 Conclusion

Many programming languages support a wide variety of concurrency models and these are often combined by developers. In this paper, we studied the combinations of futures, actors, and transactions. We demonstrated that a naive combination can invalidate the guarantees that these models normally provide, thereby breaking the assumptions of developers.

We presented Chocola, a framework that integrates futures, actors, and transactions into a unified model. Chocola defines a semantics for the combinations that maintains the guarantees of its constituent models wherever possible. We implemented Chocola as a fork of Clojure and formalized its semantics. Moreover, we transformed three benchmark applications to mix multiple models and demonstrate that this improves their performance by introducing more fine-grained parallelism. These transformations do not fundamentally change the design of the program and thus require only a relatively small effort from the developer. Hence, using Chocola, developers can freely pick and mix concurrency models in their program.

References

[1] G. A. Agha. 1985. *Actors: a model of concurrent computation in distributed systems.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[2] H. C. Baker and C. Hewitt. 1977. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages.* 55–59.

[3] S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. B. Johnsen, K. I. Pun, S. L. T. Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. 2015. Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In *Formal Methods for Multicore Programming: 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM 2015).* 1–56.

[4] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *Proceedings of AGERE! 2015.* 1–12.

[5] J. De Koster, S. Marr, T. Van Cutsem, and T. D'Hondt. 2016. Domains: Sharing state in the communicating event-loop actor model. *Computer Languages, Systems & Structures* 45 (2016), 132–160.

[6] J. De Koster, T. Van Cutsem, and W. De Meuter. 2016. 43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties. In *Proceedings of AGERE'16.* 31–40.

[7] M. Felleisen, R. B. Findler, and M. Flatt. 2009. *Semantics Engineering with PLT Redex.* The MIT Press.

[8] K. Fernandez-Reyes, D. Clarke, and D. S. McCain. 2016. ParT: An Asynchronous Parallel Abstraction for Speculative Pipeline Computations. In *Proceedings of COORDINATION 2016.* 101–120.

[9] R. H. Halstead. 1985. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems* 7, 4 (Oct. 1985), 501–538.

[10] T. Harris, J. R. Larus, and R. Rajwar. 2010. *Transactional Memory* (2nd ed.). Morgan & Claypool.

[11] M. Herlihy and J. E. B. Moss. 1993. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of ISCA'93.* 289–300.

[12] C. Hewitt, P. Bishop, and R. Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of IJCAI'73.* 235–245.

[13] C. A. R. Hoare. 1973. *Hints on Programming Language Design.* Technical Report.

[14] S. M. Imam and V. Sarkar. 2012. Integrating Task Parallelism with Actors. In *Proceedings of OOPSLA '12.* 753–772.

[15] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *2008 IEEE International Symposium on Workload Characterization.* 35–46.

[16] J. E. B. Moss and A. L. Hosking. 2006. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming* 63, 2 (2006), 186–201.

[17] N. Shavit and D. Touitou. 1997. Software transactional memory. *Distributed Computing* 10, 2 (Feb. 1997), 99–116.

[18] J. Swalens, J. De Koster, and W. De Meuter. 2016. Transactional Tasks: Parallelism in Software Transactions. In *Proceedings of ECOOP'16.* 23:1–23:28.

[19] J. Swalens, J. De Koster, and W. De Meuter. 2017. Transactional Actors: Communication in Transactions. In *Proceedings of SEPS'17.* 31–41.

[20] J. Swalens, S. Marr, J. De Koster, and T. Van Cutsem. 2014. Towards Composable Concurrency Abstractions. In *Proceedings of PLACES'14.*

[21] S. Tasharofi, P. Dinges, and R. E. Johnson. 2013. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models?. In *Proceedings of ECOOP'13.* 302–326.

[22] P. Van Roy and S. Haridi. 2004. *Concepts, techniques, and models of computer programming.* The MIT Press.

[23] J. Zeng, J. Barreto, S. Haridi, L. Rodrigues, and P. Romano. 2016. The Future(s) of Transactional Memory. In *Proceedings of ICPP '16.* 442–451.