Towards Composable Concurrency Abstractions[*]

Janwillem Swalens, Stefan Marr, Joeri De Koster and Tom Van Cutsem

Software Languages Lab, Vrije Universiteit Brussel, Belgium
{jswalens,smarr,jdekoste,tvcutsem}@vub.ac.be

Abstract

In the past decades, many different programming models for managing concurrency in applications have been proposed, such as the actor model, Communicating Sequential Processes, and Software Transactional Memory. The ubiquity of multi-core processors has made harnessing concurrency even more important. We observe that modern languages, such as Scala, Clojure, or F#, provide not one, but *multiple* concurrency models that help developers manage concurrency. Large end-user applications are rarely built using just a single concurrency model. Programmers need to manage a responsive UI, deal with file or network I/O, asynchronous workflows, and shared resources. Different concurrency models facilitate different requirements. This raises the issue of how these concurrency models interact, and whether they are *composable*. After all, combining different concurrency models may lead to subtle bugs or inconsistencies.

In this paper, we perform an in-depth study of the concurrency abstractions provided by the Clojure language. We study all pairwise combinations of the abstractions, noting which ones compose without issues, and which do not. We make an attempt to abstract from the specifics of Clojure, identifying the general properties of concurrency models that facilitate or hinder composition.

1 Introduction

A typical interactive computer program with a graphical user interface needs *concurrency*: a number of activities need to be executed simultaneously. For example, a web browser fetches many files over the network, renders multiple documents in separate tabs, runs plug-ins in the background, and needs to keep the user interface responsive. Since the last decade multi-core processors have become ubiquitous: servers, laptops, and smartphones contain multi-core processors. This has made it possible to perform these concurrent activities simultaneously.

To express the interactions between these activities, a number of concurrency models have been developed. For example, the Actor Model [1] introduces actors to represent components of a system that communicate using asynchronous messages, while Software Transactional Memory (STM) [7, 4] coordinates the access of concurrent activities to shared memory using transactions.

Because of the varying and extensive requirements of interactive end-user programs, developers choose to combine different models [8]. For example, a web browser might use actors to run separate tabs in parallel, and manipulate the DOM (Document Object Model) of a web page using STM. We also see that many programming languages support a number of these models, either through built-in language support or using libraries. For example, the Akka library[1] for Java and Scala provides STM, futures, actors, and agents. Similarly, Clojure[2] has built-in support for atoms, STM, futures, and agents.

[*]An appendix with a complete discussion of results is available at http://soft.vub.ac.be/~jswalens/PLACES2014.pdf

[1]http://akka.io/

[2]http://clojure.org/

However, the concurrency models are not necessarily well-integrated: programmers may experience subtle problems and inconsistencies when multiple concurrency models interact. In this paper, we analyze some problems that arise when combining different concurrency models and outline potential directions for *composable concurrency abstractions*.

2 Concurrency Models

In this paper, we study the combination of atomics, STM, futures and promises, as well as approaches for *communicating threads* (actors and CSP). To illustrate the usefulness of combining these approaches, we consider how they could be used in a modern email application.

Atomics Atomics are variables that support a number of low-level atomic operations, e. g., compare-and-swap. Compare-and-swap compares the value of an atomic variable with a given value and, only if they are the same, replaces it with a new value. This is a single atomic operation. Operations affecting multiple atomic variables are not coordinated, consequently when modifying two atomic variables race conditions can occur. Atomics are typically used to share independent data fragments that do not require coordinated updates. For example, a mail client might use an atomic variable to represent the number of unread mails.

In Clojure, `atoms` are atomic references. Their value can be read using `deref`. Atoms are modified using `swap!`, which takes a function that is evaluated with the current value of the atom, and replaces it with the result only if it has not changed concurrently, using compare-and-swap to compare to the original value. If it did change, the `swap!` is automatically retried.

Software Transactional Memory (STM) STM [7, 4] is a concurrency model that allows many concurrent tasks to write to shared memory. Each task accesses the shared memory within a transaction, to manage conflicting operations. If a conflict is detected, a transaction can be retried. A mail client can use STM to keep information about mails consistent while updates from the server, different devices, and the user are processed at the same time.

Clojure's STM provides `refs`, which can only be modified within a transaction. They are read using `deref` and modified using `ref-set`. Transactions are represented by a `dosync` block. Outside a `dosync` block, refs can only be read.

Futures and Promises Futures and promises[3] are placeholders for values that are only known later. A future executes a given function in a new thread. Upon completion, the future is resolved to the function's result. Similarly, a promise is a value placeholder, but it can be created independently and its value is 'delivered' later by an explicit operation. Reading a future or a promise blocks the reading thread until the value is available. A mail client can use futures to render a preview of an attachment in a background thread, while the mail body is shown immediately. When the future completes, the preview can be added to the view.

In Clojure, futures are created using `future`. Promises are created using `promise` and resolved using `deliver`. Futures and promises are read using `deref`, which potentially blocks.

Communicating Threads We classify concurrency models that use structured communication in terms of messages, instead of relying on shared memory, as *communicating threads*. Often, each thread has only private memory, ensuring that all communication is done via messages. This, combined with having each thread process at most one message at a time, avoids race conditions. However, models and implementations vary the concrete properties to account for a wide range of trade-offs. We distinguish between models that use asynchronous messages, such as actors [1] or agents (as in Clojure and Akka), and models that communicate

[3]Literature uses various different definitions of futures and promises. We use those of amongst others Clojure and Scala here.

	atoms	agents	STM	futures	promises	core.async
create	atom	agent	ref	future	promise	chan
read	deref	deref	deref	deref ⊗	deref ⊗	<! ⊗
write	reset!		ref-set		deliver	>! ⊗
	swap! ↻	send	alter			
block			dosync ↻			go
other		await ⊗				<!! >!! ⊗
						take! put!

Figure 1: Operations supported by Clojure's concurrency models. ⊗ indicates a (potentially) blocking operation, ↻ an operation that might be re-executed automatically.

using synchronous messages, such as CSP [5]. In a mail client, typical use cases include the event loop of the user interface as well as the communication with external systems with strict communication protocols, such as mail servers.

Clojure `agents` represent state that can be changed via asynchronous messages. `send` sends a message that contains a function which takes the current state of the agent and returns a new state. The current state of an agent can be read using a non-blocking `deref`. The `await` function can be used to block until all messages sent to the agent so far have been processed. Clojure's core.async library implements the CSP model. A new thread is started using a `go` block, channels are created using `chan`. Inside a `go` block, values are put on a channel using `>!` and taken using `<!`. Outside `go` blocks, `>!!` and `<!!` can be used. These operations block until their complementary operation is executed on another thread.

All operations described above for Clojure are summarized in figure 1, highlighting blocking and re-execution.

3 Integration Problems of Concurrency Models

3.1 Criteria for composability

We study pairwise combinations of the five concurrency models described in the previous section. Two correctness properties are evaluated: safety and liveness [6]. For each combination we study whether additional safety or liveness issues can arise, emerging from the interactions between the two models. We consider two models *composable* if combining them cannot produce new safety or liveness issues.

Safety Safety means that, given a correct input, a program will not produce an incorrect result. In our context, many concurrency models are designed to avoid race conditions to achieve safety. They do this by managing shared resources: STM only allows shared memory to be accessed through transactions, while CSP or the actor model only allow threads to share data through explicit message passing.

When combining two models, new races could be introduced unexpectedly. For example, some implementations of STM have been proven linearizable [7]: every concurrent execution is equivalent to a legal sequential execution. However, this assumes that all shared resources are managed by the STM system, which is not true if a thread communicates with other threads, e. g., using CSP. This can cause unexpected interleavings that eventually lead to race conditions. This is not dissimilar to the problem of feature interaction [2], where several features that each function correctly separately, might behave incorrectly when combined.

Concretely, we study whether the combination of two concurrency models can introduce race conditions: incorrect results caused by unexpected interleavings.

	Safety					Liveness				
in \ used	atoms	agents	refs	futures promises	channels	atoms	agents	refs	futures promises	channels
atoms	✗	✗	✗	✗	✗	✓	✓	✓	✓	✗
agents	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
refs	✗	✓	✓	✗	✗	✓	✓	✓	✓	✗
futures promises	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗
channels	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗

Figure 2: This table shows when safety and liveness issues can arise by combining two models in Clojure. The model in the column is used in the model in the row.

Liveness Liveness guarantees a program will terminate if its input is correct. In our context, two problems can occur: deadlocks and livelocks. Deadlocks are introduced by operations that block, waiting until a certain condition is satisfied, but the condition is continually not satisfied. Livelocks appear when code is re-executed under a certain condition, and this condition is continually satisfied. Some concurrency models have proven liveness properties, for instance, STMs are usually non-blocking [7]. Others try to confine the problem by limiting blocking to a small set of operations. For example, futures only provide one blocking operation, reading, which waits until the future is resolved. As long as the future eventually resolves, no deadlocks will happen.

Again, when concurrency models are combined, unexpected deadlocks and livelocks might arise. An STM transaction that uses blocking operations of another model, such as CSP, is not guaranteed to be non-blocking anymore. Or, a future that contains blocking operations from another model might not ever be resolved, in other words, combining futures with another model can introduce unexpected deadlocks.

We study whether the combination of two concurrency models can introduce new deadlocks, by studying the blocking operations offered by a model, and/or new livelocks, by studying the operations that can cause re-execution.

3.2 Integration Problems in Clojure

We examine these combination issues specifically for Clojure. Each of the five concurrency models from section 2 is embedded in each of the models (including itself). The complete set of results is shown in the table in figure 2. For example, we embed a `send` to an agent (1) in an atom's `swap!` block (figure 3a), (2) in another agent action, (3) in an STM transaction (figure 3b), (4) in a future, and (5) in a `go` block (these results form the second column of figure 2). Even though some of these individual results are already known, we systematically study each pairwise combination of models, in an attempt to provide a comprehensive overview of safety and liveness issues. A discussion of the most interesting results is given below, a complete discussion of all results in the table is given in the online appendix.

Safety We first look at the possibility of race conditions (left side of table 2). Race conditions are caused by an incorrect interleaving between two models.

When using any concurrency model in the function given to an atom's `swap!` operation (first row of the table), race conditions are possible, because the function might be re-executed if the atom changed concurrently. For example, when this function sends a message to an agent, it could be sent twice (figure 3a). Moreover, because operations on multiple atoms are not coordinated, their updates are inherently racy.

```
(def notifications (agent '()))
(def unread-mails (atom 0))

(swap! unread-mails
  (fn [n]
    (send notifications
      (fn [msgs] (cons "New mail!" msgs)))
    (inc n)))
```

(a) Sending a message to an agent, in `swap!`. Send may happen more than once.

```
(def notifications (agent '()))
(def mail (ref {:subject "Hi" :archived false}))

(dosync
  (ref-set mail (assoc @mail :archived true))
  (send notifications
    (fn [msgs] (cons (str "Archived mail " (:subject
          @mail)) msgs))))
```

(b) Sending a message to an agent, in a `dosync`. Send is delayed until the transaction is committed.

Figure 3: Sending a message to an agent, in a block that might re-execute (`swap!` and `dosync`).

For STM, actions inside a `dosync` block are re-executed if the transaction is retried, and therefore no irrevocable operations should happen inside `dosync`. However, there are two safe combinations. Firstly, when a message is sent to an agent inside a `dosync` block (figure 3b), Clojure does not send this message immediately. Instead, it delays the send until the transaction is successfully committed. Secondly, embedding one `dosync` block in another means the inner transaction will be merged with the outer one, and as a result transactions are combined safely.

Based on these results we conclude that *if the 'outer' model might re-execute code it is given, and the 'inner' model might perform irrevocable actions, unexpected interleavings can happen and therefore safety is not guaranteed.*

Liveness Next, we look at the liveness property (right side of table 2): deadlocks and livelocks.

Deadlocks are introduced by blocking operations (indicated in figure 1). CSP relies heavily on blocking for communication, and as such deadlocks are possible when it is embedded in another model (see last column). This is particularly problematic when embedded in `swap!` (atoms) or `dosync` (STM): synchronous communication is irrevocable and should not be re-executed.

Reading a future or a promise blocks until its value is available. This can cause a deadlock when a promise is embedded in an agent (fourth column), because one thread might send an action to an agent, which reads a promise that is delivered in a later action sent to that same agent, possibly from another thread. Reading futures inside another future can also cause a deadlock when mutually recursive futures are allowed, as is the case in Clojure.

Finally, the agents' blocking `await` operation can cause deadlocks in a `go` block or a future (second column). In agent actions and STM transactions this situation is prevented by raising an exception. We conclude that *if the 'inner' model might block, and the 'outer' model does not expect this, a deadlock is possible.*

Livelocks appear when code is re-executed (operations that can cause re-execution are indicated in figure 1). An STM transaction is retried when it conflicts with another one, causing a livelock if the conflict would consistently occur. However, Clojure's STM prevents such deadlocks and livelocks dynamically [3]. In general, *a livelock can appear when a model that re-executes code is combined with a model that causes this re-execution to continually happen.* However, this occurs in none of the examined cases in Clojure.

4 Solutions and Open Questions

In the discussion of the previous section, we already pointed out some places where bad interactions between models are avoided by Clojure. Specifically: (1) Sending a message to an agent in an STM transaction is delayed until the transaction has been committed. (2) `await` is not allowed in STM transactions, nor in actions sent to agents. (3) A `dosync` block embedded in another will not start a new transaction: the inner transaction is merged into the outer one. These mechanisms could be replicated in some other cases: similar to sending a message to an agent in a transaction, delivering a promise could also be delayed until the transaction has been committed. The combination of futures and transactions could further be improved by canceling futures started in a transaction if the transaction is aborted.

A solution to deadlocks caused by the combination of agents and futures/promises is to disallow reading a future/promise in an agent action, an operation that potentially blocks the agent. Instead, the future/promise would need to be read before sending the message.

To remove the unsafe combinations of atoms with any other model, some of the mechanisms of combinations with STM could be replicated (e. g., delaying sends to agents). However, this can be considered contrary to the purpose of atoms: they are low-level and uncoordinated, and accordingly offer no safety guarantees. Similarly, the CSP model uses synchronous, blocking operations by design, and therefore liveness issues are inherent to this model. Trying to avoid these would be contradictory to the nature of the model.

In general, in future research we would like to decompose several concurrency models into their components, or “building blocks”. For example, we observe that some common elements are: (1) most models supply a way to create new threads (e. g., `agent`, `future`, `go`); (2) agents, actors, futures, promises and CSP provide message passing (asynchronous or synchronous); (3) agents, actors, and CSP have private memory per thread, while (4) atomics and STM provide a way to manage shared memory.

We want to extract such common elements and provide a way to compose them safely and efficiently. For example, different threads could have some private memory, communicate using message passing (as in the actor model and CSP), but also share a section of memory (e. g., using STM). Using these composable concurrency abstractions, it should be possible to express existing concurrency models as well as combinations of them correctly. In the end, it should be possible to write complex applications, such as the email client example of section 2, using a combination of concurrency models, without introducing new safety or liveness issues caused by interactions between the models.

5 Conclusion and Future Work

There exist various different concurrency models, and in many large-scale applications these are combined. However, subtle problems and inconsistencies can appear in the interactions between these models. In this paper, we studied the safety and liveness issues that can appear when the various concurrency models available for Clojure are combined.

We identified four reasons for conflicts between models. Firstly, when a model re-executes code, and this code uses another concurrency model that performs irrevocable actions, safety is not guaranteed. Clojure takes some special precautions in some of these cases. Secondly, when a model re-executes code, and this code can cause the re-execution to continually happen, a livelock is possible. In Clojure, this is prevented in the studied cases. Thirdly, when a model that supports blocking operations is embedded in a model that does not expect this, deadlocks become possible. Again, Clojure prevents this in some cases but not in others. Lastly, some models do not provide safety or liveness guarantees by design.

In future work, we aim to work towards composable concurrency abstractions: we will decompose existing concurrency models into more primitive “building blocks”, and provide a way to compose these safely and efficiently.

References

[1] Gul A. Agha. *Actors: a model of concurrent computation in distributed systems.* PhD thesis, MIT, 1985.

[2] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[3] Chas Emerick, Brian Carper, and Christophe Grand. *Clojure Programming.* O'Reilly, 2012.

[4] Tim Harris, Simon Marlow, Simon P. Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '05*, pages 48–60, New York, New York, USA, 2005. ACM Press.

[5] C. A. R. Hoare. Communicating sequential processes. *Comm. of the ACM*, 21(8):666–677, 1978.

[6] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[7] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC'95: Proc. of the fourteenth annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM, 1995.

[8] Samira Tasharofi, Peter Dinges, and Ralph Johnson. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models? In *Proc. of ECOOP'13*, Montpellier, France, 2013.