Composable Concurrency Models

Dan Stelljes
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
stell124@morris.umn.edu

ABSTRACT

The need to manage concurrent operations in applications has led to the development of a variety of concurrency models. Modern programming languages generally provide several concurrency models to serve different requirements, and programmers benefit from being able to use them in tandem. We discuss challenges surrounding concurrent programming and examine situations in which conflicts between models can occur. Additionally, we describe attempts to identify features of common concurrency models and develop lower-level abstractions capable of supporting a variety of models.

1. INTRODUCTION

Most interactive computer programs depend on concurrency, the ability to perform different tasks at the same time. A web browser, for instance, might at any point be rendering documents in multiple tabs, transferring files, and handling user interaction. On a lower level, the operating system might be running several other applications, juggling background processes, and responding to events. If every long-running process blocked other processes from proceeding, the system would be effectively unusable.

Processes themselves are often composed of multiple concurrent threads of execution that each work on a distinct task. A processor can only execute one thread at a time, so multitasking is accomplished by rapidly switching between threads [8]. Although concurrent threads may appear to be executed simultaneously, truly parallel execution can only take place across multiple processors.

Concurrency models enable programmers to reason about concurrent tasks instead of low-level thread management. The web browser is a good example of how different models might be chosen to represent specific types of tasks: The user interface layer might rely on an event loop, the rendering process might operate in shared memory, and suggestions from browsing history or a search engine might require parallel collection operations to achieve acceptable performance [9].

UMM CSci Senior Seminar Conference, December 2016 Morris, MN.

Given that an application is likely to make use of more than one concurrency model, programmers would prefer that different types of models could safely interact. However, different models do not necessarily work well together, nor are they designed to. Recent work has attempted to identify common "building blocks" that could be used to compose a variety of models, possibly eliminating subtle problems when different models interact and allowing models to be represented at lower levels without resorting to rough approximations [9, 11, 12].

2. BACKGROUND

In a concurrent program, the history of operations may not be the same for every execution. An entirely sequential program could be proved to be correct by showing that its history (that is, the sequence in which its operations are performed) always yields a correct result. For a concurrent program to be proved correct, though, it must be clear that all possible histories always yield a correct result. Furthermore, concurrent programming frequently involves the use of shared resources that require coordinated access and manipulation.

Consistency models, restrictions on possible execution histories, can guarantee that a program will produce a correct result [12]. If an execution of a program follows an allowed history, that execution is said to be consistent; if not, it is said to be inconsistent. If every possible execution of the program is guaranteed to follow a history allowed by a consistency model, that program conforms to that consistency model [5].

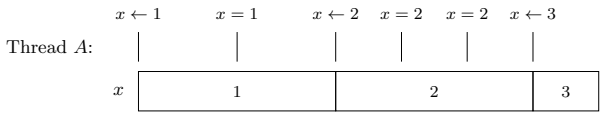

Figure 1: A single thread reads from and writes to a variable.

Figure 1 illustrates a simple program in which a single thread A reads ($=$) and writes ($\leftarrow$) values on a variable x, denoted by a segmented bar containing the value of x over time. The program satisfies an intuitive model of how variables should behave—each time x is read, the value returned is equal to the value most recently written and the consistency model holds. If, however, a read on x failed to return the most recently written value, the consistency model would be violated.

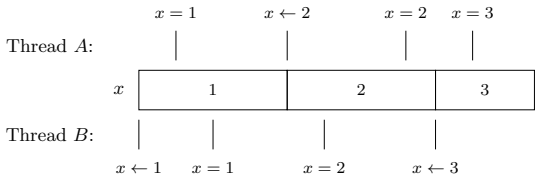

Figure 2: Two threads concurrently read from and write to a variable.

In Figure 2, two threads A and B concurrently read and write on x. Because the operations of the threads are interleaved, a read operation on x may not yield a value that matches the value of that thread's most recent write. The value may even be inconsistent between consecutive reads. If a single thread assumed that it had exclusive control of x, incorrectly applying the single variable consistency model, this behavior would appear to be inconsistent. Similar errors that arise due to an unintentional dependency on execution order are commonly referred to as race conditions.

2.1 Linearizable consistency

The examples above assume that all operations complete instantaneously. In real systems, though, operations take time. Even writing to a location in cache memory, an operation measured in nanoseconds, is not truly instantaneous. The fact that operations are completed at some time after they are invoked introduces uncertainty into a history of operations—the sequence of execution may be influenced by the time it takes for messages to travel [5].

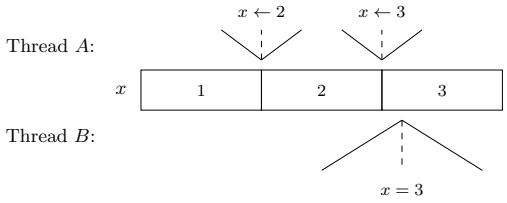

Figure 3: Two threads concurrently (and noninstantaneously) write to and read from a variable.

Consider a third example (Figure 3) in which two threads A and B interact with a variable x noninstantanously. Invocation is denoted by the leftmost point of an event and completion by the rightmost point. The dotted line indicates the instant at which the operation takes effect. Thread B invokes a read while the value of x is set to 2. Thread A invokes and completes a write between the time that the read is invoked and the value is actually read. The read operation then returns 3 even though x was equal to 2 at the time it was invoked.

While travel time may introduce ambiguity, there are still some restrictions on possible sequences of events. Specifically, an operation cannot take effect before its invocation, nor can it take effect after its completion; rather, it will appear to take effect atomically at some point after it is invoked and before it is completed.

This consistency model is referred to as linearizability (also referred to as atomicity or indivisibility), and it guarantees that the completion of a single operation on a single object will appear to the rest of the system to be instantaneous [3]. In other words, even though linearizable operations are executed concurrently and take time, they appear to happen in a simple linear order.

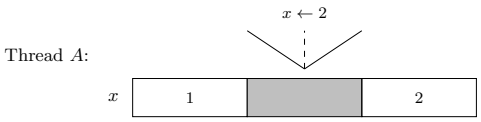

Figure 4: A single thread performs an atomic write operation on a variable.

Figure 4 demonstrates an atomic write on a variable x: Although the operation takes time, the entire operation (denoted by the shaded area) can be collapsed into one apparently instantaneous event. A useful consequence of linearizability is that the results of an operation must be visible as soon as the operation is complete. This can prevent issues such as stale reads (in which a read operation returns a value different from the most recently written value) and non-monotonic reads (in which a later read operation returns an older value than an earlier read operation). Linearizability is also a composable guarantee [3]—an operation made up of smaller linearizable operations is itself linearizable.

2.2 Sequential and causal consistency

For operations to be executed concurrently, they must be able to be executed out of order or in partial order. Lamport, in his foundational work on distributed systems [7], formalized this by defining a "happens before" relation ($\rightarrow$) and its negation "does not happen before" ($\nrightarrow$) on a set of operations: If A and B are operations in the same thread and A occurs before B, or if A is the sending of a message by one thread and B is the receipt of the same message by another thread, then $A \rightarrow B$. If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$. Two operations A and B are said to be concurrent if $A \nrightarrow B$ and $B \nrightarrow A$.

The first condition of the "happens before" relation, that A and B are ordered within the same thread, describes sequential consistency. Sequential consistency simply requires that operations in a thread take place in that thread's order. Causal consistency, described by the second condition of the "happens before" relation, enforces the intuitive requirement that an effect cannot take place before its cause.

2.3 Serializable consistency

A history of operations is said to be serializable if it is equivalent to a serial (i.e., non-interleaved) ordering of its operations [3]. Serializability is like linearizability in that it demands some linear order of execution. However, serializability describes multiple operations over multiple objects instead of single operations on single objects, and it does not impose any wall-clock time constraints on the history. This means that operations may occur in any order as long as some serial history exists.

Figure 5 illustrates a set of serializable read and write operations. Even though the executions of each operation occur out of order and even intersect, the events themselves can be arranged serially; no side effects are introduced by the fact that they are executed concurrently.

Serializability alone is a fairly weak consistency model because it does not place any restrictions on time or order [5]. Events may happen out of sequence, and, unlike linearizabil-

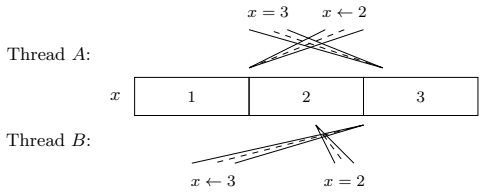

Figure 5: Two threads perform a set of serializable read and write operations on a variable.

ity, serializability is not a composable guarantee. Serializability is useful, though, as a guarantee of isolation: While a serializable set of operations is being executed, it appears to be the only set of operations being executed. For histories to be serializable, then, they must not overlap and cannot interfere with each other.

Linearizability and serializability together guarantee strict serializability, in which a history is equivalent to a serial execution order and that serial order corresponds to the execution order in real time. Given that, linearizability can actually be described as strict serializability restricted to single operations on single objects [3].

3. COMMON CONCURRENCY MODELS

Modern programming languages generally provide several different concurrency models as tools to reason about concurrent tasks. Swalens et. al, in a survey of concurrency models in the Clojure programming language, group commonly used concurrency models into four broad categories: atomic variables, software transactional memory, communicating threads, and proxies [11].

3.1 Atomic variables

Atomic variables are used to share independent objects that do not demand coordinated updates but need to be shared by multiple threads [11]. Atomic variables can only be read and mutated by operations that are guaranteed to be linearizable. For example, the atomic operation `compare-and-swap` compares the current value of a variable to a given value and only performs a write if the values are equal [11]. `compare-and-swap` ensures that a new value based on outdated information cannot be written: Suppose that a thread A reads a variable x and begins computing a new value. At the same time, another thread B modifies x. When A tries to set the value of x via `compare-and-swap`, the write will be denied.

Atomic operations are often implemented in hardware and guarantee linearizability at the lowest level possible. `fetch-and-add`, another atomic operation, is a common processor instruction that executes multiple hardware-level operations: The old value is copied from a location in memory into a temporary register, addition is performed on the value in the temorary register, and the new value is stored at the original location.

Atomic variables often serve as the basis for higher-level concurrency control mechanisms such as semaphores and locks, which in turn support the implementation of concurrent data structures. A counting semaphore is an atomic variable that serves as a record of how many units of a shared resource, such as a connection or a pooled thread, are available. By taking advantage of atomic operations, the semaphore can be safely updated and will reliably determine whether a shared resource can be used. A single thread can then test if a resource is available and acquire it before proceeding, thereby preventing race conditions [11]. A lock, also referred to as a mutex, guarantees mutual exclusion, which requires that two threads cannot operate on a shared object at the same time. Locks are commonly implemented by a binary semaphore (that is, a semaphore that simply indicates whether a single resource is available).

While lock-based programming is effective, dealing with multiple locks and complex interactions can be cumbersome. One common problem in lock-based programming is resource contention, in which multiple threads vie for control of multiple resources. Suppose, for example, that two threads A and B both require resources x and y. A might acquire x and B might acquire y. Both would then halt, waiting to acquire a lock on the second needed resource. This type of scenario is commonly referred to as deadlock.

3.2 Software transactional memory

Software transactional memory (STM) presents an alternative to lock-based programming by allowing multiple concurrent operations to write to a shared location in memory without securing any kind of lock. STM is an example of optimistic concurrency control: Each thread executes a series of read and write operations (called a transaction) on the shared memory without considering the activity of other threads. On its face, this approach seems disaster-prone—multiple threads writing to the same memory location could easily lead to corrupted data. STM solves this problem by recording all operations in a log. After the entire transaction is completed, the transaction manager verifies that other threads have not also made changes to the shared memory. If there are conflicting changes, the transaction is aborted and retried until it eventually succeeds [10].

Conceptually, STM simplifies concurrency because it allows a transaction to be thought of as a single strictly serializable operation. A thread cannot observe changes made to other threads while a transaction is in progress, nor can other threads observe modifications by that thread until the transaction has completed successfully. Wrapping a set of operations within a transaction eliminates the need to secure locks, leading to more simple programs. A transaction that modifies several shared variables, for example, would not have to secure locks for all of them. Practically, STM is useful in any situation in which multiple shared objects need to be accessed and modified by multiple threads [11].

STM also offers significant advantages over lock-based programming in terms of composability. Because a transaction can be viewed as a single linearizable operation, STM can allow any combination of linearizable operations to be composed into a larger linearizable operation. Transactions also enforce modularity by hiding implementation details of the operations within [2].

However, the requirement that transactions must be able to be aborted and retried places constraints on the set of allowed operations. Specifically, a transaction cannot execute any operation that cannot be undone, such as writing to disk or performing a network request. Operations that block indefinitely, like waiting to acquire a lock, may also cause a transaction to fail continuously. STM can also result in a decrease in performance caused by the overhead of maintaining the log and aborting and retrying transactions.

3.3 Communicating threads

Other concurrently models avoid the use of shared memory by restricting threads to private memory. Threads communicate strictly by message passing, which avoids issues such as race conditions. Because of the higher level of isolation, communicating threads have long been a solution to describing concurrent operations. Limiting thread communication to message passing also enforces modularity and limits possible negative side effects. A wide variety of communicating thread models exist and are often used to implement event loops or handle communication with external systems [11].

Communicating sequential processes (CSP), one of the oldest communicating thread models, began as a method of formally describing actions in concurrent systems. CSP describes systems in terms of independent processes that communicate through predefined channels [4]. CSP is notable in that messages are passed synchronously; that is, a sending process will block until a complementary operation is executed on the receiving process [11]. CSP is still widely used in the specification, implementation, and testing of safety-critical systems.

Other models, most notably the actor model, rely on asynchronous message passing to specific entities (actors) rather than named channels [1]. Upon receiving a message, an actor can choose to perform an operation on a resource that it controls, send messages to other actors, create new actors, determine the behavior used for the next received message, or ignore the message entirely.

Shared state can also be represented by communicating threads. The agent model, for instance, works similarly to atomic references. A state wrapped by an agent can be modified by a message that sends an updated state. The value can be accessed by a dereferencing operation that reads the current value from the agent [11].

3.4 Proxies

Proxies are a general term used to describe placeholders for values that are the result of some concurrently executed operation. Unlike communicating thread models, which rely on message passing between dedicated threads, a proxy executes some task in a new thread and delivers the result upon completion [11].

Futures and promises are two of the most common proxy models. The terms are frequently used interchangeably, along with "delayed," "deferred," and "eventual." Generally, futures are resolved to the result of the completed operation. The result is then accessed implicitly; any use of the future will return its value. Promises differ in that they are created as independent objects and require the result to be accessed explicitly.

In practice, proxies are used to execute long-running operations such as rendering or network requests [11] without blocking other operations. For example, to eliminate the need for a program to block on a long network request, a promise could be used to complete the network request in another thread and specify an action that should take place once the result becomes available. Promise objects, which can be easily passed, composed, and chained, offer more flexibility than constructs such as asynchronous callbacks. In fact, many mainstream languages now directly support proxies.

4. COMPOSABILITY CHALLENGES

Even if a concurrency model guarantees correctness when used alone, issues may arise when it is used with other models. For example, an implementation of an STM that guarantees linearizability assumes that all shared resources are managed by the STM [10]. If such a resource is modified outside of the STM, an unexpected interleaving of operations could result.

Swalens et al. [11] attempted to identify issues that arise when different concurrency models are combined, hoping that future work could identify common "building blocks" that could be used to compose a variety of concurrency models. The study surveyed all of the concurrency models available in the Clojure programming language and used them within one another to see what types of correctness issues were encountered.

4.1 Correctness criteria

In the study, safety and liveness were used as the two criteria for evaluating the correctness of a combination of models. Together, the two have historically been used to prove the correctness of distributed systems [6].

Informally, safety guarantees that "nothing bad will happen." In other words, given a correct input, a program will not produce an incorrect result [11]. This property is also referred to as partial correctness. In the context of concurrent programming, safety is generally achieved by the management of shared resources. STM, for instance, only allows shared memory to be accessed through transactions; communicating threads only allow data to be shared through message passing.

Liveness guarantees that "something good will eventaully happen," or that a program will eventually terminate if its input is correct. Taken together, safety and liveness are referred to as total correctness—given a correct input, a program will terminate with the correct output [11]. Deadlocks (in which execution is blocked) and livelocks (in which execution continues indefinitely but never makes progress) are the primary obstacle to liveness and can arise when models are combined.

4.2 Possible conflicts

To discover the types of conflicts that might arise when combining different models, the study examined all pairwise combinations of Clojure's models. Using every model within every other model (for example, manipulating an atomic variable within an STM transaction) uncovered the following types of conflicts:

- A model might reexecute code containing another concurrency model that performs an irrevocable action. If a block of code such as a transaction were to contain an operation that sent a message on a channel, reexecutions of that block would cause the message to be sent repeatedly.
- A model might reexecute code that causes the reexecution to continually happen. The authors of the study observed that, especially when many transactions are being executed at once, a large STM transaction might consistently conflict with another and never succeed.
- A model that supports blocking operations might be used within a model that does not expect blocking operations. The authors of the study provided mutually

recursive futures as an example—if two futures each attempted to read the value of the other, a deadlock would occur.

- A model might not guarantee safety or liveness by design, making safe or live composability impossible.

The study noted that some types of bad interactions are prevented by Clojure; as an example, sending a message to an agent from within a transaction is delayed until the transaction succeeds. The study also noted that similar safeguards could prevent other negative interactions. However, preventing every type of negative interaction between models might undermine the purpose of the models to begin with; channels, as mentioned previously, block by design and are inherently incompatible with other models. With that in mind, further research would likely seek to decompose concurrency models into common elements and provide a method to compose those elements safely and efficiently.

5. UNIFYING ABSTRACTIONS

Other research into composable concurrency models has focused on the development of underlying abstractions capable of supporting a wide variety of models. Instead of programmers using composable "building blocks" to construct models, these abstractions would make use of those common elements at a lower level to improve performance and interoperability.

5.1 Ownership-based meta-object protocol

Marr and D'Hondt [9] surveyed a variety of concurrent and parallel programming concepts such as immutability, critical sections, ownership, and common concurrency models to identify a unifying concurrency abstraction that could be implemented in a high-level language virtual machine. A unifying abstraction would enable each concurrency model in a high-level language to be represented in machine code without sacrificing performance or semantics. For instance, a concurrency model in Clojure could be accurately represented by an underlying abstraction in the Java Virtual Machine, and code written in the Java programming language could interact with that model correctly. To identify the concepts that would need to be supported at the virtual machine level, Marr and D'Hondt selected four questions:

1. Can the concept be reasonably implemented as a library? A library implementation may suffer from loss of performance or semantics to an extent that the concept would warrant virtual machine inclusion.

2. Does the concept require runtime support to guarantee its semantics? Semantic guarantees may be enforced by a compiler but not the virtual machine. For example, immutable objects in one language may not be immutable in machine code, and therefore may be mutated by another language.

3. Would runtime support benefit performance? Deeper knowledge of the concept may enable the virtual machine to better optimize. Some concepts, though, may require knowledge of the underlying hardware to realize any performance improvements; such concepts would not necessarily benefit from runtime support.

4. Is the concept already supported by a common virtual machine like the Java Virtual Machine or the Common Language Runtime?

From those questions, 26 concepts were identified that would benefit from runtime support. Of those, 18 suffered from loss of semantic guarantees when compiled to machine code; the survey mentioned transactions as an example. With those concepts in mind, Marr and D'Hondt derived the following requirements for a unifying abstraction:

- Managed mutation and execution: A model may impose rules on how objects can be modified; therefore, mutation must be handled in a way that allows those rules to be enforced. By way of example, a thread should not be permitted to mutate a variable that a model declares immutable. Similarly, a model may restrict the invocation of operations on objects.

- Ownership: All mutation and execution on an object is regulated relative to some "owning" entity, so ownership should be supported in a way that allows for adaptable mutation and execution rules.

- Leveled reflection: Reflection allows a program to examine its own structure and behavior at runtime. To guarantee safety, there needs to be a distinction between language-level reflection (reflection operations supported by the high-level language) and meta-level reflection (reflection operations on the low-level abstraction).

- Enforceability: All defined restrictions should be enforceable on different concurrency models. Additionally, the semantics of the language (immutability, for example) in which a model is defined must be enforced regardless of where the model is used.

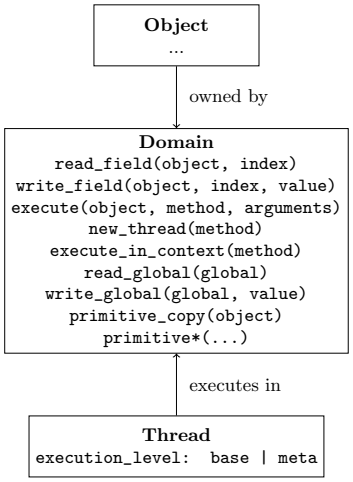

Figure 6: Marr and D'Hondt's owenership-based meta-object protocol [9].

Given those requirements, Marr and D'Hondt defined an ownership-based meta-object protocol (MOP) that could describe a low-level abstraction, illustrated in Figure 6. The owner of an object, referred to as the domain, manages operations on all of the objects that it owns. Specifically, the

domain handles all reading and writing of object fields and invocation of methods on objects. This satisfies the ownership requirement. Read and write operations are directed to the `read_field` and `write_field` operations on the domain, satisfying the managed mutation requirement. Similarly, all method invocations are directed to the `exec`, satisfying the managed execution requirement. Threads are executed in a domain and specify whether execution occurs at the language level with restricted reflection or the meta level with unrestricted reflection, satisfying the leveled reflection requirement. Globally shared resources are handled by the domain (through the `read_global` and `write_global` methods) if they might break semantics. Additionally, the `primitive*` operations allow callers to override the semantics of virtual machine primitives, completing the enforcability requirement.

To demonstrate the suitability of the MOP as a low-level abstraction, Marr and D'Hondt implemented several concurrency models. The MOP was able to support LRSTM (a STM for the Smalltalk language), Clojure's agents (which rely on communicating threads), active objects (a variation on proxies), and several other models. While the MOP was able to successfully enforce the semantics of each model, the resulting performance cost suggested that the MOP must be implemented at the virtual machine level to sufficiently reduce the overhead of abstraction. Marr and D'Hondt also note that regarding the owner of an object as the only entity able to restrict interaction is somewhat limiting. For instance, the behavior of the system when different semantics interact needs to be explicitly specified.

6. CONCLUSION

There are a rich variety of concurrency models that can be used to represent concurrent tasks, abstract over low-level details, and ensure that concurrent applications perform correctly. Recent research has attempted to identify ways in which these models can be safely used together, common features of these models, and underlying abstractions capable of representing many different models.

Future work will likely explore ways in which concurrency models can be decomposed and recomposed from common elements. This, in turn, would lead to the development of more generalized ways of describing concurrency models and perhaps additional unifying abstractions. As Marr and D'Hondt note, a "silver bullet" does not exist. However, there is certainly promise in searching for better ways to represent concurrent operations.

Acknowledgments

Thanks to Elena Machkasova, K. K. Lamberty, and Matthew Justin for their suggestions and feedback.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA: MIT Press, 1986. ISBN: 0-262-01092-5.

[2] Tim Harris et al. "Composable Memory Transactions." In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '05. Chicago, IL: ACM, 2005, pp. 48–60. ISBN: 1-59593-080-9. URL: http://doi.acm.org/10.1145/1065944.1065952.

[3] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects." In: *ACM Transactions on Programming Languages and Systems* 12.3 (July 1990), pp. 463–492. ISSN: 0164-0925. URL: http://doi.acm.org/10.1145/78969.78972.

[4] C. A. R. Hoare. "Communicating Sequential Processes." In: *Communications of the ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. URL: http://doi.acm.org/10.1145/359576.359585.

[5] Kyle Kingsbury. *Strong Consistency Models — Aphyr*. May 2014. URL: https://aphyr.com/posts/313-strong-consistency-models.

[6] Leslie Lamport. "Proving the Correctness of Multiprocess Programs." In: *IEEE Transactions on Software Engineering* 3.2 (Mar. 1977), pp. 125–143. ISSN: 0098-5589. URL: http://dx.doi.org/10.1109/TSE.1977.229904.

[7] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." In: *Communications of the ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. URL: http://doi.acm.org/10.1145/359545.359563.

[8] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." In: *J. ACM* 20.1 (Jan. 1973), pp. 46–61. ISSN: 0004-5411. URL: http://doi.acm.org/10.1145/321738.321743.

[9] Stefan Marr and Theo D'Hondt. "Identifying a Unifying Mechanism for the Implementation of Concurrency Abstractions on Multi-language Virtual Machines." In: *Objects, Models, Components, Patterns: 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings*. Ed. by Carlo A. Furia and Sebastian Nanz. Berlin, Heidelberg, Germany: Springer-Verlag, 2012, pp. 171–186. ISBN: 978-3-642-30561-0. URL: http://dx.doi.org/10.1007/978-3-642-30561-0_13.

[10] Nir Shavit and Dan Touitou. "Software Transactional Memory." In: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '95. Ottowa, Ontario, Canada: ACM, 1995, pp. 204–213. ISBN: 0-89791-710-3. URL: http://doi.acm.org/10.1145/224964.224987.

[11] Janwillem Swalens et al. "Towards Composable Concurrency Abstractions." In: *Proceedings of the Seventh Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*. PLACES '14. Grenoble, France: EPTCS 155, 2014, pp. 54–60. URL: http://dx.doi.org/10.4204/EPTCS.155.8.

[12] Ofri Ziv et al. "Composing Concurrency Control." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR: ACM, 2015, pp. 240–249. ISBN: 978-1-4503-3468-6. URL: http://doi.acm.org/10.1145/2737924.2737970.