J. Parallel Distrib. Comput. 74 (2014) 2266–2285

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A survey of support for structured communication in concurrency control models

Alexandre Skyrme *, Noemi Rodriguez, Roberto Ierusalimschy

Informatics Department, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rua Marquês de São Vicente, 225 - RDC, CEP 22.451-900, Rio de Janeiro, RJ, Brazil

HIGHLIGHTS

- We survey communication among execution flows in concurrency control models.
- We evaluate each model's support for structured communication.
- Most models do not guarantee that the communication among execution flows occurs only in syntactically restricted code regions.
- The ease of reasoning about communication among execution flows is often neglected.
- Structured communication could reduce the complexity of concurrent programming.

ARTICLE INFO

Article history:
Received 9 July 2012
Received in revised form
19 September 2013
Accepted 18 November 2013
Available online 6 December 2013

Keywords:
Concurrency
Communication
Survey
Model
Structured

ABSTRACT

The two standard models used for communication in concurrent programs, shared memory and message passing, have been the focus of much debate for a long time. Still, we believe the main issue at stake should not be the choice between these models, but rather how to ensure that communication is structured, i.e., it occurs only in syntactically restricted code regions. In this survey, we explore concurrency control models and evaluate how their characteristics contribute positively or negatively to the support for structured communication. We focus the evaluation on three properties: reasonability, which is the main property we are interested in and determines how easily programmers can reason about a concurrent program's execution; performance, which determines whether there are any distinct features which can prevent or facilitate efficient implementations; and composability, which determines whether a model offers constructs that can be used as building blocks for coarser-grained, or higher-level, concurrency abstractions.

1. Introduction

In concurrent programs, communication among execution flows can be carried out according to two standard communication models: *shared memory* and *message passing* [5]. Both models are commonly employed as building blocks for the implementation of concurrent programming languages and libraries. Neither of them is explicitly associated with a programming model, but shared memory is commonly used with multithreading programming, while message passing is often used with event-based (or event-driven) programming. Despite much discussion comparing these models [86,87,97,113,116,125], no consensus has been reached about which of them is the best. Generally, shared memory is regarded as having better performance [97], but also as being more complex and error prone [87,97,113,125], while message passing is regarded as being less error prone [97] and easier to debug [53], but also as having lower performance and less flexibility [116,125].

In all the controversy surrounding shared memory and message passing, an issue that is often neglected is that of how to ensure that communication does not produce unpredictable results. The interactions among execution flows should be localized and explicit in the code, making it easier for programmers to reason about a program's execution. Specifically, concurrency models should guarantee that the communication among execution flows occurs only in syntactically restricted code regions. We will say that models that satisfy this requirement provide *structured communication*. Other intuitive requirements are that concurrency control models should be amenable to implementations with adequate performance and to composition, that is, it should be possible to build coarser-grained protected interactions from finer-grained ones. We believe this change of focus, from *selecting a communication model* to *ensuring structured communication*, is a necessary step to reduce the complexity of concurrent programming, a problem recurrently cited in the literature [87,97,113,125].

* Corresponding author.
E-mail address: askyrme@inf.puc-rio.br (A. Skyrme).

0743-7315/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jpdc.2013.11.005

As a first step in achieving a better understanding of this concept, we devoted ourselves to studying concurrency models to analyze how their characteristics contributed positively or negatively to the support for structured communication. A concurrency control model defines how multiple execution flows can interact. In the selection of the models we analyzed, we tried to pick representatives from different approaches, focusing on models that have been used in practice and that have been used in more than one system or language.

In this paper, we report the results of this investigation. For each model, we analyze what communication constructs or patterns are available and whether any constraints are enforced regarding their employment. Instead of strictly analyzing and comparing shared memory and message passing, we look for concurrency control models that build on these communication models, or disregard them, to provide higher-level communication abstractions. We evaluate each model according to three properties: *reasonability*, *performance* and *composability*.

The main property we are interested in evaluating in each surveyed model is *reasonability*. This property determines how easy it is for programmers to reason about the execution of a concurrent program. It considers whether a model can prevent unpredictable results (such as *out-of-thin-air values*), as well as whether it can keep non-determinism localized and explicit, i.e., whether it allows programmers to clearly identify, within the source code, what calls or operations can lead to non-deterministic results. When evaluating this property, we are mostly concerned with the ability to avoid race conditions, as we believe this is the issue that most complicates reasoning and leads to unpredictable results. We are also concerned with deadlocks, although to a lesser extent. In most models with blocking operations, deadlocks are possible. However, explicit localization of interactions between concurrent threads of execution also facilitates reasoning about deadlocks, and will thus help avoiding them.

We also evaluate two additional properties in each surveyed model: *performance* and *composability*. The discussion about performance evaluates whether a model has any distinct features which can by themselves prevent or facilitate efficient implementations. We do not include any performance measures, as it would not make sense to compare mechanisms that have been implemented in different languages and contexts. *Composability* refers to the ease with which coarser-grained, or higher-level, concurrency abstractions can be built from finer-grained, lower-level ones.

Both message passing and shared memory can be used as implementation techniques and/or as communication models. Just as it is possible to offer a shared-memory abstraction in a distributed environment which relies on an underlying message-passing scheme for communication, it is also possible to offer a message-passing abstraction implemented with shared-memory synchronization primitives in a multiprocessed environment. Our concern is with the model that is offered to the programmer and with the guarantees that it will give him. We only delve into implementation details when discussing the impact of a given feature on performance.

The focus of this survey is on *local concurrency*, that is, the concurrency among execution flows running on the same machine. We avoid distributed environments as we understand they limit the underlying implementation and involve many other factors which are outside the scope of the survey. This scope restriction does not prevent us from considering concurrency models which can also be used in distributed environments.

The remainder of this survey is organized as follows. In Section 2 we present an overview of the two conventional communication models used in concurrent programming, namely shared memory and message passing. In Section 3, we explain our evaluation criteria, analyze concurrency control models and evaluate their support for structured communication among multiple execution flows accordingly. In Section 4, we present an overall discussion of the surveyed models. Finally, in Section 5 we present some concluding remarks.

2. Conventional communication models

The two standard models used to allow for communication among multiple execution flows are shared memory and message passing. We choose to evade the controversy surrounding which model is better. Instead, we show in this section that both shared memory and message passing have specific issues regarding the complexity and performance. Besides, there is a duality between the two communication models: it is possible to offer a shared memory abstraction to programmers implemented with message passing, likewise it is possible to offer a message passing abstraction implemented with shared memory.

2.1. Shared memory

Shared memory communication typically refers to a setting in which preemptive threads can read or write data in a global shared space. Since there is no way for the processor to automatically identify correct and incorrect access sequences, it is up to the programmer to properly use available synchronization primitives to ensure correctness. Mutexes and semaphores [39] are the most widely used primitives.

We can summarize the criticism that has been leveled at shared memory [6,21,88,97,111] in the following points:

1. when shared memory is used with preemptive multithreading, it is difficult to determine whether an arbitrary operation accesses a shared variable and can lead to race conditions; this prevents programmers from confidently reasoning about a program's execution. Also, the possible data races encountered by programmers are in themselves hard to understand;
2. the synchronization primitives that must be used by programmers to provide mutual exclusion are complex and error-prone, thus increasing the potential for incorrect or incautious use, which can lead to race conditions or deadlocks. It is difficult for the programmer to pinpoint the proper placement of these primitives;
3. ensuring correctness while exploiting concurrency presents a dilemma to programmers. As more concurrency opportunities are explored in a program, potentially more synchronization primitives and shared variables are used and therefore reasoning about the program's execution and ensuring its correctness become more complex.

Non-blocking synchronization, which was first proposed by Herlihy [72] partly relieves the dilemma between ensuring correctness and exploiting concurrency opportunities. Non-blocking synchronization aims to ensure that no execution flow will have to wait indefinitely to access a shared resource and, for all practical purposes, to allow safe access to shared data without the use of synchronization primitives. Non-blocking operations are usually implemented by means of atomic read–write–modify primitives supported by hardware, such as the compare-and-swap (CAS) instruction, and are commonly classified as *wait-free*, *lock-free* and *obstruction-free* [73]. A wait-free operation guarantees that all execution flows calling it will eventually succeed, bounding the number of steps needed to complete its execution. A lock-free operation guarantees that at least one execution flow calling it will eventually succeed, and thus concerns overall progress as opposed to individual execution flow progress. An obstruction-free operation also guarantees that at least one execution flow calling it will eventually succeed, but as long as there

are no conflicting execution flows running at the same time. Both Java, in its java.util.concurrent package, and the .NET framework, in its System. Threading namespace, support non-blocking operations and data structures.

Despite relieving some of the drawbacks of shared memory by removing the use of locks, non-blocking synchronization also introduces its own subtleties. Many commonly used data structures still lack a non-blocking implementation and each new non-blocking data structure requires a new implementation which is typically so complex that results in a published article [112]. Besides, writing correct non-blocking code that performs well can be a difficult task, even for experts. One particular concern with non-blocking synchronization is memory management, as discussed by Herlihy et al. [74,73] and Michael [94].

2.2. Message passing

Message passing can refer either to an architecture model for parallel computers or to a communication model [125] — in this survey we are interested in the latter. When a message-passing communication model is used, multiple execution flows, processed independently, can communicate with each other by sending and receiving messages. A number of communication abstractions and programming models have been built based on message passing. Among the most well-known of them are Remote Procedure Calls (RPC) [14], Publish/Subscribe [13] and the Actor Model [2].

Message addressing usually relies either on unique execution flow (or thread or process) identification or on decoupled communication channels (or mailboxes). The two main operations associated with message passing are *send* and *receive*, which normally operate as one-to-one operations, although it is also possible to have collective operations [51], that is, operations that simultaneously send to or receive from multiple execution flows.

Send and receive operations can be either synchronous or asynchronous. A synchronous send returns only after the message has been received by the target execution flow, i.e., it waits until the message has been received, while an asynchronous send returns immediately. Analogously, a synchronous receive returns only after a message has been received, while an asynchronous receive returns immediately (perhaps after checking a channel or polling a mailbox for stored messages). This suggests that message passing requires some degree of coordination among execution flows, as communication can only happen if at least one execution flow is willing to send data and another execution flow is willing to receive it. The degree of coordination required is dependent on the chosen message-passing model, or more specifically, on its degree of *coupling*.

The coupling of message-passing models can be evaluated according to two dimensions: *space* and *time*. The space dimension determines how much information about the "location" of a receiver is needed to send a message. A more space-coupled model, for instance, would use point-to-point messages and an addressing scheme based on unique execution flow identifiers, all belonging to a single global namespace. A more space-decoupled model, on the other hand, would use broadcast messages or a blackboard communication paradigm, where senders do not interact directly with receivers. The time dimension determines the degree of synchronization required to send a message. A more time-coupled model, for instance, would require both sender and receiver(s) to exist at the same time to communicate. A more time-decoupled model, on the other hand, would use persistent channels or mailboxes to store messages, allowing execution flows to receive messages asynchronously. The degree of coupling determines the needs for coordination and, consequently, the extent of fault tolerance and of delays caused by waiting for other execution flows.

A commonly used standard to implement message passing is the Message Passing Interface (MPI) [93]. MPI includes a number of variations of the send and receive operations (such as blocking, non-blocking, buffered, synchronous, standard and ready). At first glance it might seem as if some of these variations do not fit the synchronous and asynchronous definitions we presented. However, in this survey we consider communication to be asynchronous whenever a sender has no guarantee that sent messages were delivered to a receiver after the send operation returns. Therefore, according to the definitions we presented, synchronous communication in MPI only happens when both send and receive operations are blocking and use the synchronous mode, i.e., when both `MPI_Ssend` and `MPI_Recv` functions are used. Non-blocking operations and operations that work in standard or buffered modes are considered asynchronous by the definitions we presented, since the send call may return before the target execution flow has received the message. Non-blocking operations in MPI require careful buffer management, as we discuss in Section 2.3.

An example of a programming model relying on message passing is event-based programming. In a typical event-based programming, execution flows register their interest in certain events and are notified when these events occur, usually by the invocation of event handlers. Event notification, or the execution of event handlers, can be understood as an asynchronous receive operation; the receiver does not make an explicit receive call, but rather registers its interest in an event and a corresponding handler. An *event dispatcher* places the received events in an event queue and sequentially invokes the registered handlers for the queued events.

Message passing has at least two prominent advantages: it can easily be used as an abstraction for both local and distributed communication and, as long as explicit operations are used, it makes communication among execution flows explicit in the code. The latter makes reasoning about a program's execution easier, as the points in code where non-deterministic behavior can occur are clear. Despite its advantages, message passing also has its drawbacks. Message passing involves copying data between different address spaces, leading to large overheads as the transmitted data grow in size. The semantics of message transmission, which is related to this impact on performance, are often unclear. For instance, how are complex objects or data structures transmitted? Event-based programming has also been criticized for demanding an *inversion of control* [59]: a program's execution flow is not determined by a sequence of operation calls, but rather by the occurrence of events, which are dispatched to the appropriate event handlers. This implies flow and logic fragmentation among event handlers and thus throughout a program. A possible solution to this problem is the use of *coroutines* [37].

2.3. Hybrid implementations

Some programming languages offer a hybrid model, where both message passing and shared memory are supported. Examples of such languages include Scala [96] and the D programming language [3]. Scala supports the Actor Model to allow for communication among execution flows. It is implemented on the Java VM and allows for interoperability with the Java programming language. Therefore, Scala also supports Java's standard shared memory concurrency facilities. The D programming language approach is to use isolated threads that communicate via message passing. It also supports shared memory and mutual exclusion. However, by default, only immutable data can be shared among threads; sharing mutable data requires the use of type modifier which restricts how data is accessed — we discuss immutability in Section 3.4.

In addition, sometimes despite relying on a single communication model programmers still have to be wary of implementation subtleties to avoid a false sense of security. The MPI standard, for instance, includes non-blocking operations (such as `MPI_Ibsend`, `MPI_Irsend`, `MPI_Issend`, `MPI_Isend` and `MPI_Irecv`). Non-blocking operations allocate a communication request object and return a handle that can be used to query the status of the communication or wait until it is finished. When a non-blocking call returns, the program may start to copy data out of a send buffer or write data into a receive buffer, even if the communication has not finished. Hence, in multithreaded implementations of MPI data races could occur when non-blocking operations are used and, according to the MPI standard, it is up to the programmer to ensure that buffers are not modified until communication finishes.

3. Concurrency control models

Over the last four decades, many concurrency control mechanisms have been proposed and built into programming languages [5,25]. These mechanisms include busy waiting, semaphores [39], conditional critical regions [60,61,77], monitors [62,63], guards [40] and path expressions [29], among others. However, many of them are either not concerned with enforcing the characteristics in which we are interested or are connected to specific niches in programming languages. In this section, we explore four models that have been widely discussed and implemented and that present features that are important for structured communication: monitors, transactional memory, tuple spaces, and data immutability. We believe these models are representative of different approaches to concurrency. Monitors were the first structured-programming concurrency construct to be widely discussed, and retain practical importance today. They impose a static view of the chunks of code in which accesses to shared memory can occur. Transactional memory extends to memory accesses the atomic properties of database transactions and reflects a more flexible approach to guaranteeing mutual exclusion, in which arbitrary chunks of code may be protected by enclosing constructs. Tuple spaces can be regarded either as global shared memory spaces or as sets of communication channels, and were important in emphasizing the separation between computation and coordination. Finally, immutability is in a slightly different category, as by itself it is not a concurrency control mechanism, but analyzing immutability is an important step in understanding how type systems can be used to guarantee protection from unwanted interference.

In the following subsections we present, for each of the surveyed models, an overview of the model, remarks about its practical use, a brief discussion about limitations, an evaluation of structured communication support (based on the reasonability, performance and composability properties), as well as a summary of this evaluation. Evaluations are based on the literature review as well as on our own experience with each of the surveyed models. When summarizing the evaluation of each model we use the following criteria:

Reasonability : when a model prevents unpredictable behaviors throughout a program's execution, we rate its reasonability as *predictable*. When a model prevents unpredictable behaviors in finer-grained constructs (or code blocks) but cannot prevent unpredictable behaviors in coarser-grained constructs (or code blocks) we rate its reasonability as *fine-grained predictable*. Finally, when a model cannot prevent unpredictable behaviors even in finer-grained constructs (or code blocks), we rate its reasonability as *unpredictable*;

Performance : when a model can be implemented to allow the performance of all communication to be comparable to the communication among execution flows within a single process (a best-case scenario), we rate its performance as *optimizable*. When a model can be implemented to allow the performance of some, but not all, communication to be comparable to the communication among execution flows within a single process, or when it has distinct features which can reduce the performance of its implementations, we rate its performance as *restrained*;

Composability : when a model is explicitly concerned with allowing its concurrency abstractions to be used to build coarser-grained, or higher-level abstractions we rate its composability as *modeled*. When a model does not have such an explicit concern but has implementations specifically designed to allow for the composition of its abstractions, we rate its composability as *implemented*. Finally, when a model is not explicitly concerned with composability and has no implementations specifically designed to support it, we rate its composability as *ad hoc*.

3.1. Monitors

Monitors [62,63] were perhaps one of the earliest attempts to guarantee controlled access to data shared among execution flows. A monitor, as defined by Brinch Hansen [64], is a combination of shared variables and procedures which provide the only means for accessing these variables. Monitor procedures are executed one at a time, i.e., with mutual exclusion, and may delay calling execution flows waiting for an arbitrary condition. A common use for a monitor's shared variables is to represent shared hardware or software resources, such as disk drives or files. A general monitor diagram is presented in Fig. 1.

The original monitor concept by Brinch Hansen was further developed by Hoare [78], who explicitly stated that procedures within a monitor should not access any variables external to the monitor and, in an analogous manner, a monitor's variables should not be accessible from outside the monitor. This ensures that monitors' procedures are the only way to access shared variables. It is an essential condition to avoid time-dependent errors, since it allows them to be detected during program compilation, a requirement stated both by Hoare and Brinch Hansen.

A clear analogy can be made between monitors and Remote Procedure Calls (RPC) [4]. Calling a monitor procedure is similar to making a call to a remote procedure; in both cases there is an entity (a monitor or a central server) which is responsible for keeping local state, as well as centralizing and orderly fulfilling requests that involve access to shared data or resources. This evidences that monitors could be implemented both with shared memory, as usually implied, and with message passing, analogously to Remote Procedure Calls. Furthermore, it shows that the choice between the shared memory and message passing communication models can be just an implementation matter.

3.1.1. In practice

The first programming language to implement monitors was Concurrent Pascal [65], an extension of the sequential programming language Pascal intended to support concurrency. Concurrent Pascal implements explicit *access rights*, which are lists, that can be checked at compile time, of shared resources that a monitor can operate on. The ability to check access rights at compile time allows the compiler to make sure that all shared data access is protected by monitors and that monitor procedures are executed in mutual exclusion, as well as to examine the graph of monitor calls to detect possible deadlocks due to cyclic calls, thus avoiding

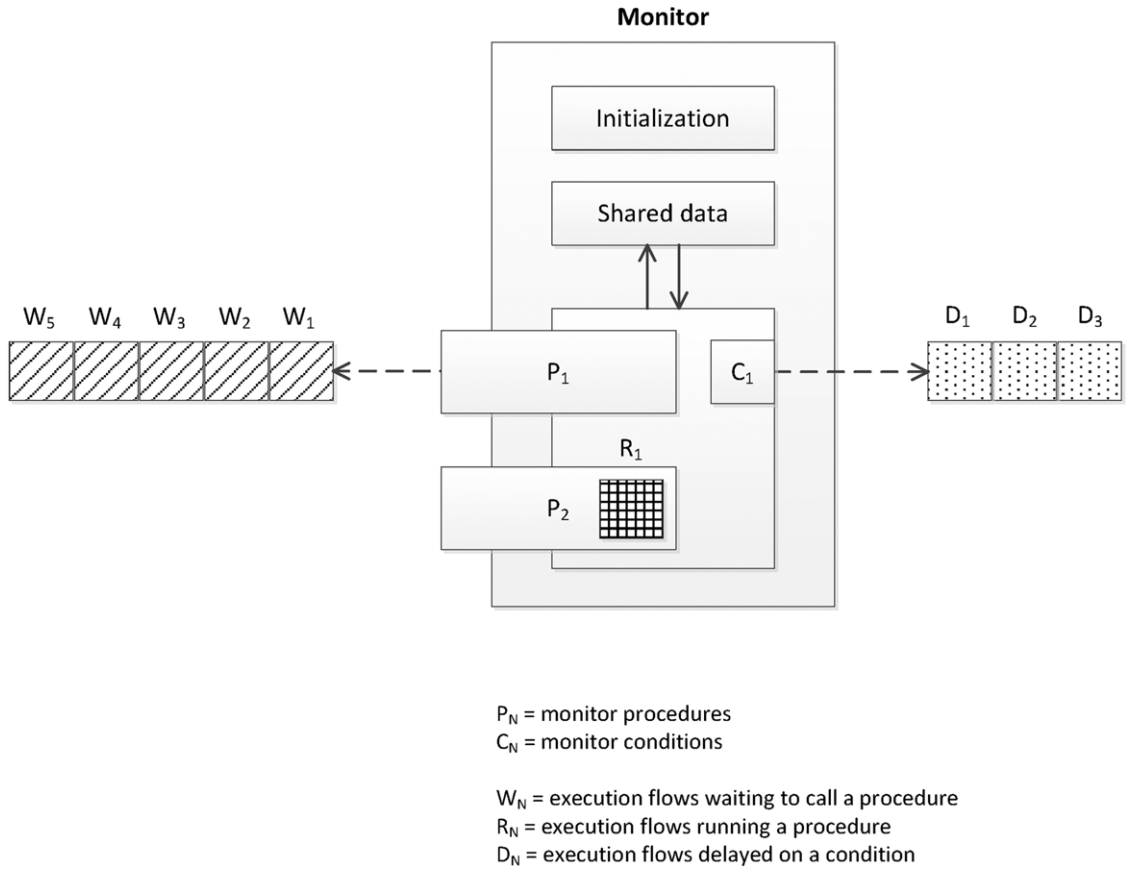


Fig. 1. A general monitor diagram.

time-dependent errors. Here, once again, we observe a clear concern with enforcing a well defined pattern for concurrency, as the compiler is used to ensure shared data can only be accessed with monitors' procedures.

An example, proposed by Brinch Hansen [64], of a monitor implemented in Concurrent Pascal and is presented in Fig. 2. The `linebuffer` type represents a bounded buffer that can hold a single line, stored in the `contents` variable. It uses a boolean variable (`full`) to indicate whether the buffer is full and two queue variables (`sender` and `receiver`) to hold delayed execution flows. The monitor includes two procedures: `receive`, which retrieves the contents of the buffer, and `send`, which stores a text line in the buffer. If `send` is called and the buffer is full, the execution flow calling it is delayed on the `sender` queue until another execution flow calls `receive` and performs a `continue` operation on the `sender` queue. Analogously, if `receive` is called and the buffer is empty, the calling execution flow is delayed on the `receiver` queue.

Concurrent Pascal enforces that the only means to access a monitor's shared variables is to use monitor procedures. Therefore, programmers cannot access the `contents` variable directly when using the `linebuffer` type; they must use the `receive` and `send` procedures to do so.

Apart from Concurrent Pascal, many modern programming languages support, or claim to support, monitors or synchronization features that are based on monitors. Examples of such programming languages include, but are not limited to: .NET-supported programming languages such as C#, C++ and Visual Basic (Monitor Class), Python (threading.Condition objects), Ruby (Monitor Class) and Java (synchronized methods, wait and notify methods in the Object class).

An example of a bounded buffer class implemented with Java monitors is presented in Fig. 3. Similar to the example in Concurrent Pascal (see Fig. 2), the buffer can hold a single string (`message`) and has a condition (`full`) to indicate whether it is full. It also includes two methods, `take` and `put`, equivalent to the `receive` and `send` procedures in the Concurrent Pascal example. Analogously, if `put` is called on a full buffer or `take` is called on an empty buffer, the calling execution flow is delayed. The `wait` and `notifyAll` methods, used to delay an execution flow and to resume delayed execution flows, are included in the public Object class, which is the root of the class hierarchy; as a result, all objects implement these methods.

The main difference between the Java and the Concurrent Pascal examples is that in Java there is no enforcement that shared variables can only be accessed with monitor procedures. Consequently, in Java, a programmer can directly access the `message` variable without using the `take` or `put` methods. The use of access level modifiers, such as `private`, can inhibit such accesses. Still, this is a fundamentally different approach from Concurrent Pascal, as it relies on programmers discipline, and not on the programming language itself, to ensure correctness. Clearly, Java monitors do not comply with the original monitor concept [66].

Therefore, despite the apparent wide support for monitors in modern programming languages, a closer inspection reveals that implementations are not faithful to the original monitor concept. Most do not enforce that shared data is accessed only inside a monitor. This leaves the burden of controlling shared data access to programmers and defeats detection, in compile time, of possible race conditions.

The basis for monitor implementations, and in particular to the monitor implementation in Concurrent Pascal, was presented by Brinch Hansen [61]: he proposed using a combination of queue variables and critical regions to control access to shared data. In his proposal, execution flows that need to enter a critical region

```
type linebuffer =
monitor
var contents: line; full: Boolean;
    sender, receiver: queue;

procedure entry receive(var text: line);
begin
  if not full then delay(receiver);
  text := contents; full := false;
  continue(sender)
end;

procedure entry send(text: line);
begin
  if full then delay(sender);
  contents := text; full := true;
  continue(receiver)
end;

begin full := false end
```

Fig. 2. A bounded (single line) buffer implemented with a monitor in Concurrent Pascal.

```
public class BoundedBuffer {

    String message;
    boolean full = false;

    public synchronized String take() {
      while (!full) {
        try {
          wait();
        } catch (InterruptedException e) {}
      }
      full = false;
      notifyAll();
      return message;
  }

  public synchronized void put(String message) {
    while (full) {
      try {
        wait();
      }
      catch (InterruptedException e) {
        (...)
      }
    }
    full = true;
    this.message = message;
    notifyAll();
  }
}
```

Fig. 3. A bounded buffer implemented with Java monitors.

are placed in a *main queue* which is associated with a queue variable. When an execution flow enters the critical region, it may check whether the queue variable satisfies a certain condition; if the condition is satisfied, it proceeds to executing its critical region, otherwise it is placed in a scheduling queue, called an *event queue*. After an execution flow completes executing the critical region, all other execution flows are moved (one at a time) to the main queue to be resumed, since the condition they are waiting for might now be satisfied. Despite enforcing mutual exclusion, this means execution flows could be transferred many times, unnecessarily, between the event queue and the main queue. The two basic operations on queue variables are `delay`, to defer an execution flow until a certain condition is satisfied, and `continue`, to indicate a certain condition is satisfied and resume an execution flow.

A variation of Brinch Hansen's proposal, presented by Hoare [78], proposes using *condition variables*, a variation of queues. Condition variables, like queues, represent different reasons why an execution flow might be delayed when trying to enter a critical region, i.e., when trying to call a monitor procedure. Each condition variable is associated with two operations: `wait`, the equivalent to `delay`, and `signal`, the equivalent to `continue`. A delayed execution flow waiting for a condition to be true does not occupy a monitor. So, unlike a busy wait, an execution flow can call a monitor procedure even if another execution flow is delayed on a condition that belongs to that monitor. This allows execution flows to eventually be able to satisfy the condition and send a signal that allows a delayed execution flow to resume.

Hoare also proposed a stricter resumption policy to delayed execution flows. He defined that once a signal operation was executed, it should be immediately followed by the resumption of a delayed execution flow, a restriction which is not present in Brinch Hansen's original proposal. Moreover, he defined that the execution flow that had been waiting the longest for a condition variable should be resumed first [67]. This policy contributes to ensure delayed execution flows will be granted access to a monitor, since it prevents non-delayed execution flows from using the monitor right after a signal and right before a delayed execution flow has resumed.

The strategy of giving immediate monitor access to a signaled execution flow and enqueuing the signaler execution flow, as proposed by Hoare, constitutes the signaling discipline called *Signal and Wait* (SW); monitors that employ this strategy are often referred to as *Hoare-style* monitors. There are, however, other signaling disciplines [79]. In *Signal and Urgent Wait* (SU), a slight variation of the previous discipline, signalers are assumed to have a higher priority to access monitors and are placed at an urgent queue which is checked before new execution flows are allowed to use a monitor. In *Signal and Return* (SR), signalers must immediately return after signaling, since often this will be the last operation before a procedure returns and so it would make no sense to queue execution flows which have nothing else to do other than return; monitors that employ this strategy, and thus signal as the last operation before exiting, are often referred to as *Brinch-Hansen-style* monitors. Finally, in *Signal and Continue* (SC), the signaler is allowed to continue its execution and the signal serves just as a reminder (sometimes called a hint or a notification) that a queued execution flow should be resumed once the current procedure waits or returns; monitors that employ this strategy are often referred to as *Mesa-style* monitors. The SC strategy is used both in the POSIX threads library (pthreads) and in the Java programming language.

An alternative to explicit condition queues (such as SW, SU, SR and SC) is *implicit signaling* [26], sometimes also called *Automatic Signaling* (AS). With explicit condition queues, while one or more execution flows wait (blocked) for a condition variable, another (active) execution flow eventually detects the condition variable is satisfied and signals awaiting execution flows. With implicit signaling, on the other hand, logical predicates are used instead of condition variables to eliminate the need for explicit signal operations. If the predicate is false, then the execution flow is blocked; otherwise, if the predicate is true, a waiting execution flow is implicitly resumed. Implicit signaling was first proposed by Kessels [84] and is thoroughly explored by Buhr and Harji [27].

3.1.2. Discussion

The main criticisms to monitors are the complexities associated with nested monitor calls and with execution flow scheduling. Nested monitor calls are considered a feature [68] in the original monitor implementation. If an execution flow is delayed in a nested monitor call, it releases access only to the most recently called monitor. Nevertheless, this behavior can cause deadlocks or performance degradation [90], a matter which is further discussed elsewhere [58,83,122,85]. Essentially, four strategies are proposed, none of which is unanimous: releasing access only to the most recently called monitor, releasing access to all monitors called so far, releasing access to a monitor whenever it calls another monitor, and forbidding nested monitor calls. Brinch Hansen argues that he used nested monitor calls without problems and that the potential problems lack experimental evidence. The perception that nested monitor calls are not a problem is also supported by Parnas [98].

The complexity of execution flow scheduling comes down to the choice among signaling discipline when implementing monitors. Although signaling disciplines can appear equally complicated and their differences in practice might appear subtle [64], the main dilemma lies in choosing whether to use implicit signaling. Explicit signaling strategies (SW, SU, SR and SC) can be easier to implement with good performance, as a single execution flow can be signaled and resumed, but are more difficult to use and require some reasoning by programmers; the latter, on the other hand, is simpler to use but can be harder to implement with good performance, as in a worst case scenario all blocked execution flows will need to have their predicates reevaluated to determine whether they should be resumed. Moreover, a program that performs well with a signaling discipline might not do so if another discipline is used, i.e., different signaling disciplines can be more or less suitable for different programs.

Also, as we mentioned earlier in this section, many so-called monitor implementations, like Java monitors, do not enforce that shared data access can only be performed by monitors. The existence of unsynchronized access to shared data makes programs vulnerable to race conditions and makes reasoning about programs' execution more difficult.

An approach to adapt Java monitors to enforce structured access to shared data, presented by Bacon et al. [10], describes Guava, a modified version of the Java programming language. Guava does not allow unsynchronized shared data access and mandates that classes whose instances can be shared must be explicitly distinguished from classes that cannot be shared. Classes in Guava must fall under three different categories: monitors, whose instances can be accessed concurrently and whose methods are always synchronized; objects, whose instances cannot be accessed concurrently, although they can be freely referenced within the same thread; and values, that unlike standard Java include user-defined classes as well as primitive types. Values cannot be referenced and thus cannot be shared. Furthermore, Guava uses program annotations to allow for static checking of concurrent programs; single threaded programs do not require annotations and may be implemented by using just objects and values.

Table 1
Summary of structured communication support in monitors.

	Monitors
Reasonability	*Predictable*, as long as shared data can only be accessed by using monitor procedures and it is not possible to use references to directly access shared variables. Signaling disciplines and nested monitor calls can make reasoning more complex.
Performance	*Restrained*, considering the need to execute entry and exit protocols in monitors. Different signaling strategies might influence performance. Implementing implicit signaling with good performance, in particular, can be difficult.
Composability	*Ad hoc*, since nested monitor calls are not explicitly considered in the model and, to our knowledge, there is no reference implementation that uses nested calls. Moreover, nested monitor calls can lead to deadlocks and reasoning about nested calls can be complex.

3.1.3. Structured communication support

Overall, monitors represent a proposal which is aligned with our concern in ensuring structured communication since it considers the enforcement of a well defined pattern for execution flows to communicate by means of shared data access. They are well suited for mutual exclusion, and seem specially useful for lower level synchronization needs, such as to control access to hardware resources.

Unfortunately, apart from Concurrent Pascal, other implementations ignored the fundamental aspects which could ensure structured access and thus deprived programmers from the main benefit of monitors. A possible explanation for the lack of proper implementations and more widespread employment of monitors lies in the difficulty to offer guarantees such as those offered by Concurrent Pascal in programming languages that were designed to allow references. When programming in a language with referential semantics, it is up to programmers to ensure that a shared variable is not directly accessed by its reference instead of by using the proper monitor methods, as we showed with the example of the bounded buffer implemented with Java monitors. The difference is that while Concurrent Pascal was designed to support the development of concurrent programs and the enforcement of monitors to access to shared variables, languages like Java were designed to be more multi-purpose and were not concerned with enforcing concurrency patterns for shared data access. Simply adding monitor support to a programming language with referential semantics is not enough to ensure structured access to shared variables.

Reasonability

Monitors, as formulated by Brinch Hansen and Hoare and implemented in Concurrent Pascal, allow programmers to reason about a program's execution with certainty, as access to shared data can only be performed by procedures within monitors. While signaling disciplines and nested calls can add some complexity to reasoning, monitors and monitor procedure calls are explicit in source code and guarantee mutual exclusion. Monitors do not prevent deadlocks, as they are blocking and allow nested calls. Later monitor implementations failed to comply with the fundamental concept of allowing shared data to be accessed only from within monitors; this, in effect, voids the benefits of monitors, as it means shared data can be accessed both from within and from outside monitors, thus allowing race conditions and preventing programmers from reasoning about a program's execution without worrying about arbitrary operations accessing shared variables.

Performance

To access shared data with monitors, programmers must call a monitor's procedure, which in turn entails the execution of an entry protocol to the monitor before the procedure's body can be executed, then the execution of the procedure's body and, finally, the execution of an exit protocol. This can create a performance overhead, particularly as monitors rely on mechanisms like semaphores or condition variables to implement their entry and exit protocols. Additionally, choosing whether to use implicit signaling presents a dilemma between performance and usability. Implicit signaling is simpler to use, but it is harder to implement with good performance, since when a monitor is signaled it might have to reevaluate the predicates of all waiting execution flows before determining which, if any, of them should be resumed. Other (explicit) signaling disciplines, on the other hand, can be more complex to use and reason about, but can be easier to implement with good performance, since when a monitor is signaled it can resume a specific execution flow without having to reevaluate predicates of waiting execution flows.

Composability

Using monitors as a building block to compose more coarse-grained concurrency control is possible, but it makes reasoning about a program's execution more difficult. When reasoning about a program with nested monitor calls, a programmer must consider the strategy used to retain and release exclusion on monitors; a programmer must also consider the order in which nested calls are made, since deadlocks will happen if calls in the same order are required to hold and to release exclusion on a monitor. Other practical aspects of composing monitors are unclear, since the literature is surprisingly lacking regarding this matter.

3.1.4. Summary

A summary of structured communication support in monitors is presented in Table 1.

3.2. Transactional memory

The concept of *transaction* [52] has been widely explored, specially regarding concurrency and database management systems. A transaction is a sequence of operations that executes in an all-or-nothing fashion: it can either succeed, if all its operations succeed, or fail, if one of its operations fails. The partial effects caused by a transaction while it is executing are not visible outside the transaction, i.e., a transaction executes in *isolation*. When a transaction succeeds, we usually say it *committed*, since any changes it made become visible outside the transaction. When a transaction fails, we usually say it *rolled back*, since any partial changes it made while executing, although not visible outside the transaction, are discarded.

The idea of using transactions as means to control in-memory concurrency was first suggested by Herlihy and Moss [76]. They describe *transactional memory* as a multiprocessor architecture with a lock-free implementation of transactions, which are defined as finite sequences of machine instructions that satisfy properties equivalent to isolation.

Conceptually, a transaction comprises a sequence of tentative memory operations executed by a single execution flow; any changes carried out by the memory operations are only effective, and thus visible to other execution flows, if the transaction commits, and are otherwise discarded. Transactions can only commit if there are no read/write *conflicts*. A read/write conflict

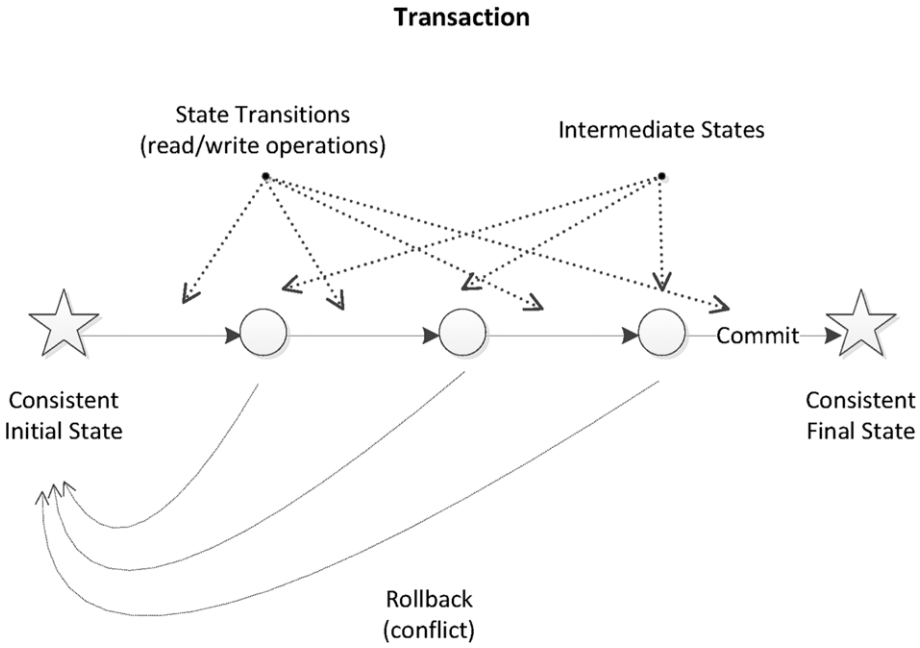


Fig. 4. A conceptual transaction diagram.

happens when two transactions are executing at the same time and one of them writes to a memory location which is read by the other one. When a conflict occurs, all transactions involved in the conflict are aborted. Other than committing, the original transactional memory proposal allows a transaction to abort, which discards all changes carried out, and to test (validate) its status in order to determine whether it has aborted. Transactions, for that matter, are supposed to be short lived and to operate on small data sets, as long-lasting transactions or transactions operating on large data sets could increase the likelihood of abortions due to conflicts. A transaction should not include operations that can cause irreversible effects (such as I/O operations) since it may need to be aborted and have its partial effects discarded. A conceptual transaction diagram is shown in Fig. 4.

The semantics of transactional memory are often compared to that of a single global lock, as transactions in a program could be understood as critical regions protected by the same lock which ensures atomic execution. This analogy, sometimes referred to as *single global lock atomicity*, is praised for its simplicity and for its semantic similarity to locks [70], with which most programmers are already familiar with. Using single global lock semantics for transactions has the added benefit of allowing coarse-grained atomic execution to be provided while hiding transactions' specifics (such as explicit rollback primitives, for instance) [19]. This highlights the fact that transactional memory can be both perceived as a programming model and as an implementation technique to provide mutual exclusion [19,69].

An important aspect of transactional memory semantics lies in isolation. It is clear that isolation is ensured among transactional code, but the interaction between transactional and non-transactional codes is not so obvious. Two models are used to define how transactional and non-transactional codes interact: *weak atomicity* and *strong atomicity*. With strong atomicity, transactional and non-transactional codes execute isolated, i.e., each operation in non-transactional code behaves as a single instruction transaction. With weak atomicity, only transactional code executes isolated, i.e., non-transactional code may read values derived from incomplete transactions or may write values while a transaction is executing. The same program can behave differently depending on the chosen semantics; a program which runs fine with weak atomicity might deadlock with strong atomicity [91].

The transactional memory model does not explicitly determine that transactions should be enforced as the only means to access shared data. Hence, programmers could disregard transactions and access shared data directly, outside transactions. This, however, is done at programmers' own risk and could lead to race conditions that could cause programs to exhibit unexpected behaviors and produce incorrect results. The lack of enforcement is the most prominent disadvantage regarding reasoning about transactional memory, as well as to its potential to promote structured shared data access.

Fig. 5 presents a simple example, based on the code provided with the TinySTM software transactional memory implementation, to illustrate the use of transactions. The example is implemented in C and uses the transactional memory support provided with the latest stable version of the GNU C Compiler Collection (GCC). It shows a function (`add`) that uses a transaction statement (defined by the `__transaction_atomic` keyword) to add a node to an integer linked list structure. Neither the list structure nor any variables that instantiate it need to include type modifiers due to the fact that they are used in transactions. The function to create a new node (`new_node`), however, must be prefixed with the `transaction_safe` keyword, to allow it to be called from within an atomic transaction.

3.2.1. In practice

As opposed to conventional concurrency control models that work by avoiding potential conflicts, transactional memory implementations can employ different strategies, depending on design choices [70], to deal with the concurrent data access. *Pessimistic* concurrency control assumes conflicts will likely occur; thus, before a transaction starts, it must claim the ownership of all data it will access throughout its execution in order to secure exclusive access to it and prevent conflicts from happening. It works similarly to locking mechanisms, except that the necessary steps to secure exclusive access to data are transparent to the programmer and thus less error-prone. *Optimistic* concurrency control assumes conflicts will likely not occur; thus, a transaction

```
1  int add(int value)
2  {
3    int result;
4    node_t *prev, *next;
5
6    __transaction_atomic {
7      prev = set->head;
8      next = prev->next;
9      while (next->val < val) {
10       prev = next;
11       next = prev->next;
12     }
13     result = (next->val != val);
14     if (result) {
15       prev->next = new_node(val, next);
16     }
17   }
18   return result;
19 }
20
21 static __attribute__((transaction_safe))
22 node_t *new_node(val_t val, node_t *next)
23 {
24   (...)
25 }
```

Fig. 5. An implementation of a function to add a node to a linked list structure.

can start executing immediately, but will only be able to commit if there are no conflicts during its execution. While locking restricts concurrent access to shared data to avoid inconsistencies, transactional memory with optimistic concurrency control allows concurrent access to shared data and deals with potential inconsistencies as they become imminent. Both concurrency control strategies require careful implementation though; pessimistic control must prevent deadlocks when transactions are securing exclusive access to data and optimistic control must prevent livelocks when conflicting transactions continuously cause each other to abort.

Transactional memory can be implemented either in hardware or in software. *Hardware Transactional Memory* (HTM) uses special hardware instructions to access memory and to manipulate a transaction's state, like the ones proposed by Herlihy and Moss, and cache-coherent protocols. However, such hardware support is not widely available, as popular processors still do not include it [24,105], leading to low portability [103] (i.e., dependence on platform specific extensions). Hardware implementations rely on adaptive backoff in order to decrease abort rates and promote forward progress, which can also be considered a drawback. Additionally, as pointed out by Lev and Maessen [89], depending on hardware can also impose size limitations on transactions.

Software Transactional Memory (STM) [110] uses software instructions to support *static transactions*, which are transactions that access a pre-defined set of memory locations. Because transactions must monitor read and write operations on shared data in order to avoid inconsistencies, a software implementation must incur a large overhead for each load or store instruction, often resulting in poor performance [32,107].[1] Also, transaction conflicts incorrectly detected due to the coarse granularity of conflict detection mechanisms, or simply *false conflicts*, can result in transaction aborts and thus degrade performance [124]. Other limitations introduced by some software transactional memory implementations include the enforcement that once a memory location is accessed transactionally, it must continue to be accessed in that way, as well as the requirement that the reclamation of memory locations accessed transactionally is handled differently from other memory locations.

Although software implementations cannot match the performance of hardware-based implementations, they offer better portability (i.e., less dependence on specific platform extensions). In fact, a software transactional memory can be considered simply a shared object that supports multiple changes by means of transactions. Many other software implementations exist, like the one proposed by Herlihy et al. [75]. In contrast with the earlier hardware-based implementation proposal, this software-based implementation is obstruction-free instead of lock-free and is able to overcome size limitations. In fact, it is the first *Dynamic Software Transactional Memory* (DSTM) proposal, as it allows transactions and transactional objects to be created dynamically; transactions can also determine which objects to access based on values observed in the same transaction. Block-freedom was a key design choice for DSTM, specially regarding how it reportedly simplified the implementation when compared to lock-freedom. However, there are also claims that block-freedom is not an important property and, moreover, that it reduces transactional memory performance [42] when a comparative performance evaluation is carried out with a non-obstruction-free implementation of STM. Empirical findings reported by Dice and Shavit [38] also suggest that lock-based implementations of transactional memory have better performance than non-blocking implementations. Since it does not require hardware support, STM has greater present applicability, as evidenced by the experimental support included in

[1] Dragojević and others [41] believe it is incorrect to compare software and hardware implementations of transactional memory, and that the correct question is whether software transactional memory outperforms sequential code.

the GNU C Compiler Collection (GCC) and in an Intel C++ STM Compiler.

Apart from transactional memory implementations strictly in hardware or strictly in software, there are also hybrid implementations. *Virtual Transactional Memory* (VTM) [104] proposes a combined hardware and software architecture which seeks to improve hardware transactional memory by providing an abstraction layer that hides hardware specifics from the programmer, similar to the virtual memory with regard to physical memory, without incurring in a significant performance impact. *Hybrid Transactional Memory* (HyTM) [36,115] combines hardware and software architectures. HyTM improves portability by trying to exploit hardware transactional memory, if it is available, and falling back to software transactional memory otherwise.

3.2.2. Discussion

The main issue with transactional memory is the lack of enforcement to ensure that shared data can only be accessed from within transactions. This issue affects both hardware and software implementations, as in both of them it is up to programmers to use transactions to access shared data. Reasoning about transactional memory could be simplified by enforcing that all shared data access is carried out from within transactions, i.e., by disallowing non-transactional code to access shared data. One way to do this would be to use a type system which enforces that shared variables can only be used inside transactions, such as in the Haskell programming language [81]. However, we are unsure about the feasibility of implementing such an approach in imperative programming languages. This enforcement would contribute to ensure communication is carried out in a structured way, as it would take away from programmers the flexibility, which we consider harmful, to choose between transactional and non-transactional accesses to shared data.

Nesting transactions, like nesting monitors, requires caution. Nested transactions may seem appealing, for instance, to allow for partial commits or aborts within large transactions. However, implementing nesting can be complex and follow different nesting models [95]. For instance, when a child transaction aborts, does the parent transaction aborts as well or is it just notified so it can take some action? And can a child transaction conflict with a parent transaction? Another issue with nesting is that the sequential composition of transactions can lead to livelocks [91].

Harris et al. [71] have proposed two operators to compose transactions in Haskell: *retry* and *orElse*. The retry operator aborts the transaction and restarts it when at least one of the variables that were read by the aborted transaction is updated by another transaction. It can be called from within a transaction to indicate a condition which prevents the transaction from running to completion. A programmer might call the retry operator, for instance, from a transaction that removes an item from a buffer in case the buffer is empty. Although similar to implicit signaling in monitors, the retry operator does not specify either the transactions it coordinates with or the shared variables which are read or updated. The orElse operator can be used to compose transactions as *alternatives*. It can be called between two inner transactions within an outer transaction; if the first transaction commits, then the second transaction is not executed, otherwise, if the first transaction executes a retry, then the second transaction is executed. If the second transaction executes a retry, then the whole orElse statement executes a retry.

The performance of transactional memory, and specially of software implementations is also an open issue. One of the few works that report practical experience with transactional memory applied to real-world applications is that of Gajinov and others [47]. It reports an experiment to rewrite Quake, a complex multiplayer game, as a parallel application using software transactional memory for concurrency control. This empirical experience showed that indeed there were significant performance overheads when using STM and taking a coarse-grained parallelization approach. The performance overhead was mostly due to the high transaction abort rate. It also concluded that shared data accessed by transactions in some cases grew to sizes that could restrict or prevent hardware transactional memory from being used.

3.2.3. Structured communication support

Transactional memory does not qualify as structured communication because it does not enforce that programmers use transactions to access shared data. This lack of enforcement compels programmers to reason about the interaction between transactional and non-transactional codes.

Reasonability

Transactional memory supports two atomicity models: weak atomicity and strong atomicity. With weak atomicity, transactional and non-transactional codes might interact in unintended ways, thus allowing data races and complicating reasoning, as arbitrary operations could access shared data. With strong atomicity, transactional and non-transactional codes execute atomically and thus there are no data races. Since there is no enforcement to use transactions, arbitrary operations can access shared variables and thus reasoning about a program's execution is complex. Even when strong semantics are used, some implementations require programmers to declare which functions are safe to be called from within transactions, thus correctness still depends on programmers' discipline and not on language enforced mechanisms.

Performance

Transactional memory can be implemented either in the hardware or in the software. Hardware implementations offer superior performance at the cost of limited portability (i.e., higher dependence on platform specific extensions), since widespread hardware support for transactional memory is still sparse. Software implementations, on the other hand, exhibit performance bottlenecks but offer better portability. Besides, performance tuning with transactional memory involves many subtleties such as transaction duration, data set sizes, concurrency control strategy (optimistic or pessimistic) and conflict resolution.

Composability

Implementing nested transactions is not trivial, as different approaches can be taken regarding how parent and child transactions interact. However, unlike monitors, transactions do not need to specify the shared resources that they must access or synchronize with, as synchronization is implicit; this simplifies composability. Functions like retry and orElse, can improve composability, as they allow for transactions to be composed as sequences or alternatives.

3.2.4. Summary

A summary of structured communication support in transactional memory is presented in Table 2.

3.3. Tuple spaces

The *tuple space* concept was first proposed by Gelernter and Carriero in the context of the *Linda* framework [30,48]. A *tuple space* is a shared associative memory which can be used by multiple execution flows to communicate. As the name suggests, a tuple space comprises tuples – sequences of values, possibly of different types – that can be accessed concurrently. Tuples have no addresses or unique identifiers, and are retrieved through pattern

Table 2
Summary of structured communication support in transactional memory.

	Transactional memory
Reasonability	*Unpredictable*, since there is no enforcement that shared data can only be accessed from within transactions.
Performance	*Restrained*, considering there is hardware support, but it is not widespread yet, and there is no consensus about the performance of software implementations.
Composability	*Implemented*, as implementing nested transactions is not trivial, yet composing operators have already been implemented in Haskell and transactions need not specify the shared resources they depend on.

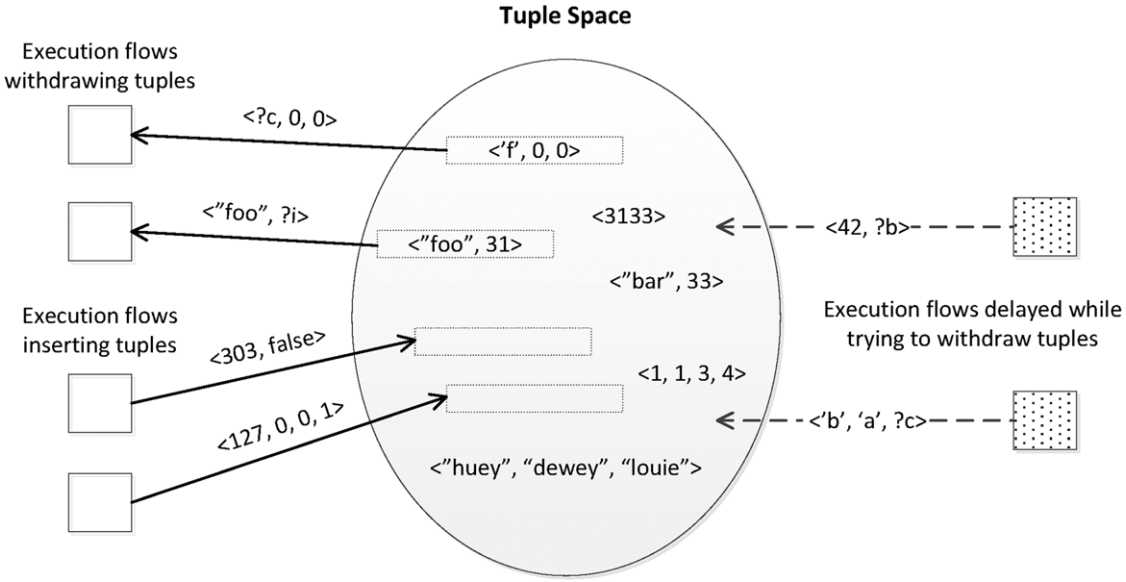


Fig. 6. A conceptual tuple space diagram.

matching on values or value types. Execution flows communicate with each other by atomically inserting and retrieving tuples in a tuple space. The tuple space communication model is both *space decoupled* and *time decoupled* Although commonly associated with distributed environments [11], conceptually there is no restriction that prevents tuple spaces from being used locally on a single machine (as implemented, for instance, in LuaLanes [82]). A conceptual diagram of a tuple space is shown in Fig. 6.

Because the literature on tuple spaces in general relates to communication among different operating system processes, it is not explicit about enforcing that access to shared data should be only through the tuple space. However, this is implicit in Carriero and Gelernter's discussion of *coordination languages* [49], which uses the Linda framework to emphasize the importance of explicitly separating computation from coordination. While the computation model – programmed in C or FORTRAN – is concerned with the activities each execution flow performs individually, the coordination model – programmed in Linda – is concerned with how execution flows are created and how they interact with each other. This promotes a clear distinction, in source code, of points where non-deterministic behaviors can occur, namely where interactions with tuple spaces happen, and directly relates to our concept of structured communication.

3.3.1. In practice

Tuple spaces represent a simple, yet flexible, communication model. They have been implemented in different programming languages, but the reference implementation is a commercial distribution [109] of the Linda programming language (TCP Linda). Although TCP Linda is supposedly used by a number of different applications in the industry, we were unable to find evidences in the literature to support that claim other than the case studies presented at its site [114]. Implementing tuple spaces holds a number of challenges, specially in terms of performance, as we will show in this section.

Linda provides a few basic operations that can be used to interact with a tuple space: *out*, *in*, *read* (sometimes referred to as *rd*), and *eval*. The `out` operation inserts a tuple into the tuple space and returns immediately. The `in` operation, conversely, withdraws a tuple from the tuple space based on pattern matching; if no match is found, it suspends execution until one becomes available, and if multiple matches are found, one is chosen at random. The `read` operation works in a similar fashion to the in operation, except that instead of withdrawing the matching tuple from the tuple space, it just copies its values. The `eval` operation triggers the execution of a new process. Both inserting and withdrawing tuples from the tuple space (`out` and `in`) are atomic operations; the order in which concurrent operations are actually executed, however, is not deterministic. Also, when more than one match is found for an `in` or `read` operation, any of the matching tuples can be returned. There are no operations to change a tuple's values in-place: in order to change values in a tuple it is necessary to first remove the tuple (`in`) and then insert a new tuple (`out`) with the updated values.

An example of an unbounded FIFO buffer implemented with a Linda-inspired syntax is presented in Fig. 7. The buffer is built using two tuple structures. The first structure includes a string label and an integer index; it is used to mark the buffer's head and the buffer's tail, both initially set to 0. The second structure includes a string label, an integer index and an integer value. The put and get functions are implemented entirely with the `in` and `out` operations.

Tuples are well-suited as a building block to implement other communication and synchronization abstractions. Lower-level abstractions can be directly mapped to Linda. The `out` and `in` operations, for instance, can be used to implement message passing, as they can clearly be mapped to the traditional send

```
1  init() {
2    out("head", 0)
3    out("tail", 0)
4  }
5
6  put(value) {
7    in("tail", ?index)
8    out("tail", index+1)
9    out("node", index, value)
10 }
11
12 take() {
13   in("head", ?index)
14   out("head", index+1)
15   in("node", index, ?value)
16 }
```

Fig. 7. An unbounded buffer implemented with a Linda-inspired syntax.

and receive operations. A tuple with a single element, on the other hand, can be used as a binary semaphore, where the `out` operation would work as the V (signal) operation and the `in` operation would work as the P (wait) operation. Building higher-level communication or synchronization abstractions with tuples, however, requires some implementation effort, as they only offer lower-level operations.

Linda's concepts and operations were implemented for many different programming languages. Besides the original extensions to C and Fortran [109], tuple spaces have been implemented for languages such as Python [123], Ruby [106] and Java [46,92]. Of particular interest is the eLinda [117] implementation in Java. eLinda includes some extensions to Linda, like a *Programmable Matching Engine* (PME) which improves the tuple matching functionality. The PME allows a program to retrieve a tuple based on criteria applied to some field or to summarize information on a subset of tuples. It allows a program, for example, to retrieve a tuple where some field has the lowest value or to calculate the number of fields of a certain type in a subset of tuples. This can improve the performance on operations which would normally require previous access to a subset of tuples before computing a result. A survey of Linda implementations in Java together with a comparison with eLinda and some performance benchmarks are presented by Wells et al. [120,118,121].

Linda also inspired the Concurrent Collections (CnC) model [28], which uses a coordination language to add parallelism to a host programming language. CnC uses a tuple space to allow the communication and creation of execution flows. Tuples in CnC are composed of values and tags, which are separate entities, unlike Linda, where tags can be expressed only by the use of wildcards to select tuples. Also, CnC does not allow values that are on the tuple space to be overwritten, i.e., in CnC once a value is placed in the tuple space, it becomes immutable (see Section 3.4). Finally, the standard operation to read tuples in CnC does not remove them from the tuple space.

An early paper about developing applications with Linda is presented by Linda's authors [31]; it includes three experiments in writing parallel code in Linda. A later and more elaborate report describes a number of (mostly scientific) applications where Linda was used, and presents execution benchmarks, mostly against similar message passing applications running in distributed environments [15]. As mentioned earlier, significant emphasis is placed on using Linda in distributed environments, which explains why comparisons with message passing are recurring. Linda, however, can be considered more expressive than pure message passing, as it is trivial to emulate message passing point-to-point style communication in Linda, but it is complex to emulate Linda's behavior by using pure message passing primitives; besides, as with other models, tuple spaces can also be implemented with shared memory. This flexibility evidences the benefits of using higher-level abstractions which focus on structured communication instead of specific communication models. Other, more recent, examples of applications which use Linda concepts include the LIME (Linda in a Mobile Environment) middleware [101] and its extensions, TinyLIME [35] and TeenyLIME [34], which allow for the development of mobile and sensing applications in wireless and wireless sensor networks using distributed shared tuples as means to support communication among mobile hosts.

3.3.2. Discussion

The performance of Linda programs relies heavily on compiler optimization [125], which in turn is very dependent on tuple space usage patterns; depending on how tuples are composed and accessed throughout a program, the Linda compiler might divide a tuple space in multiple subsets to improve the indexing performance or even choose different underlying implementation mechanisms. This, in turn, makes it difficult to predict the performance of Linda programs and thus makes Linda unsuitable for systems with strict performance requirements, such as real-time and embedded systems. Associative value matching adds complexity and has little apparent value, since the tuple structure and naming schemes must be previously known to allow communication among execution flows with disjoint name sets. Some Linda implementations require that a tuple's first value is a constant string. This requirement can simplify tuple indexing, as the ability to perform pattern matching on an arbitrary tuple value could impose performance and scalability implementation issues.

Linda semantics have also received their share of criticisms. Perhaps the most common of these is the fact that retrievals must be based on exact matches [119], disallowing searches based on ranges or other, more flexible, conditions. This highlights the fact that Linda only offers lower-level operations for communication and synchronization. Therefore, although Linda is well-suited as a building block for communication and synchronization abstractions, developing higher-level abstractions requires an implementation effort and reasoning about how to collectively coordinate operations with tuples.

Table 3
Summary of structured communication support in tuple spaces.

	Tuple spaces
Reasonability	*Fine-grained predictable*, as single operations in tuple spaces have predictable behaviors but collective operations may lead to unpredictable results, such as coarse-grained deadlocks.
Performance	*Optimizable*, as tuple indexing can be optimized when statically typed, compiled programming languages are used.
Composability	*Ad hoc*, since typical operations on tuple spaces are on individual tuples and composing them requires additional constructs to ensure atomicity. There are no explicit model concerns with composability, nor reference implementations.

Another frequent concern is the lack of support for bulk operations, which may impact the performance if the programmer must resort to a large number of individual atomic operations. Finally, the fact that there is no implicit order among tuples matching a given pattern can be an extra source of complexity and requires the programmer to explicitly use sequencing mechanisms if they are necessary.

3.3.3. Structured communication support

We believe the Linda framework has brought important research contributions, in particular regarding the coordination language concept. Its communication scheme based on tuple spaces continues to draw interest in the present, as evidenced by the many Linda implementations for different programming languages and citations in research papers. However, despite Linda's apparent popularity, its reference implementation is only available commercially and lacks evidences in the literature of its use in real-world applications; thus, the tuple space concept seems to be more significant in theory than in practice.

Reasonability

The tuple space concept is simple to understand. Reasoning about a program in Linda, for instance, is easy: the communication operations are clearly defined and a programmer can determine code segments where there is interaction with a tuple space. Because all operations on a tuple space are automatically serialized, fine-grained race conditions are eliminated. Operations in tuple spaces are very similar to simple message passing. Still, in message passing there is a clear distinction between messages and their contents, i.e., the structure of the data being communicated. With tuples this distinction is blurred, as the structure of the data being communicated is defined by the tuple itself, which also represents a message. Hence, the intuitive use of message passing can be easier to reason about than the intuitive use of tuple spaces. Moreover, since tuple spaces only provide lower-level operations, they impose a greater risk of coarse-grained deadlocks in programs.

Performance

Tuple spaces' performance relies heavily on how tuples are inserted, removed and read, i.e., it relies on how tuples are structured and on how they are indexed throughout a program. Implementing tuple spaces with statically typed, compiled programming languages, allows for performance optimizations by the compiler, since it can analyze tuple access patterns. Implementing features such as allowing a tuple to be indexed by any of its values (instead of having a pre-determined index value), for instance, only seems reasonable, in terms of performance, when using such programming languages. Thus, the tuple space performance is good as long as implementations allow for compiler optimizations.

Composability

Typical operations on tuple spaces target only individual tuples. Thus, atomicity can only be ensured for single operations on single tuples. Composing higher-level operations, like an operation which requires access to multiple tuples, demands that programmers create constructs to ensure atomicity for the whole operation.

3.3.4. Summary

A summary of structured communication support in tuple spaces is presented in Table 3.

3.4. Data immutability

Type systems [45,102] allow programming languages, by means of type checking, to automatically detect certain program misbehaviors and ensure some invariants are maintained throughout a program's execution. Moreover, type qualifiers can be used to increase the expressiveness of types in standard type systems and, thus, to allow additional invariants to be specified and checked.

Although type systems are not concurrency control models, they can be useful to support structured communication, in particular when used to provide data *immutability* and to ensure only immutable data can be used for communication among execution flows. Immutability is also not, per se, a concurrency control model; however, it is a proven building for communication among execution flows and hence we choose to include it in this survey.

Several authors discuss the benefits of immutability [16,44,50], some of which are directly related to concurrency. Immutable objects are inherently simpler than mutable objects. They are a suitable choice for elements in sets and keys in associative arrays, as well as for building blocks for other objects. Since an immutable object cannot be altered, it essentially admits only a single state. Thus, it is easier to ensure object consistency throughout a program's execution, as long as class invariants are upheld by constructors. Regarding concurrency, it is safe to share or cache references to immutable objects, so it is not necessary to implement cloning methods or constructors; likewise, it is not necessary to make defensive copies of immutable objects.

Immutability makes the choice between passing an object by value or by reference come down to an implementation matter. It allows message passing semantics to be implemented without copying data [80], i.e., it allows references to immutable data be passed as if they were copies of the same data. This reinforces the idea that the main issue at stake is not choosing between shared memory or message passing, but rather ensuring that the communication is performed in a structured fashion.

3.4.1. In practice

Plenty of programming languages support *constants*, or identifiers which are associated to values that are not meant to be changed during a program's execution. It is important to distinguish constants from immutability: while constants are associated with single values that cannot be changed, immutability is associated with data structures or objects that, as a whole, including all contained values, cannot be changed. Therefore, constants are essential building blocks to implement *immutable* objects and data structures.

Constants can be both static, like hard-coded literals, or dynamic [108], like run-time constants whose values cannot be defined at compile time. Purely functional programming languages, like Haskell, operate only with constants. Imperative languages,

like C, C++ and Java, on the other hand, usually include type qualifiers which can be used to denote constants. In C and C++ the keyword `const` is used; it can be applied to variables' values and references, to prevent them from being altered, as well as to methods, in order to indicate they do not change an object's data members. In Java, the keyword `final` is used; it can be applied to variables, to allow only for a single (value) initialization. The proper usage of constant type qualifiers to allow the compiler to check that constants are not modified throughout a program is called, in C and C++ parlance, *const-correctness*.

The Java programming language, in particular, has been widely used as a testbed for immutability research. However, Java's referential semantics and the arbitrary combination of mutable and immutable types make it a complex testbed. Defining and enforcing the semantics of immutability in an object-oriented language like Java might appear simple at first, but it is not. Some evidence of the complexities associated with immutability can be found in the Java Specification Request 133 (JSR-133), which defines the Java memory model. It specifically addresses, for instance, *initialization safety*, a property which ensures that as long as an object does not leak references during its construction, all execution flows that access that object will see the values of its final fields as set by the constructor, without the need for explicit synchronization. Also, although not specific to the language, since Java uses referential semantics to access objects, it is difficult to ensure that objects inside an immutable object (sub-objects) cannot be directly accessed by using their references.

Haack et al. [55–57] explore immutability in Java by proposing two notions of immutability: *observational*, if two instances of the same object cannot be told apart, at different points in time, by an external observer, and *state-based*, if an object's state does not change after initialization. They also present an extension to the core Java language which includes an immutable attribute, that can be used with classes which can only be instanced by immutable objects, and attributes to constrain objects and methods that immutable objects depend on. They also built up this extension to create a pluggable type system which can be used to statically check object immutability in Java-like languages.

Another indication that the semantics of immutability are not trivial is presented by Pechtchanski and Sarkar [99]. They argue that type modifiers such as `final` are not enough to express some immutability properties and are limited in scope. For instance, the lifetime (or duration) of the immutability cannot be changed for a `final` variable: it always begins after the constructor finished executing and only ends when a program finishes executing. Also, the `final` modifier only applies to the declared variable, not to any objects that may be referenced by the variable. Hence, they propose a framework, implemented by means of annotations, to define immutability with improved expressiveness and evaluate how it can be used for code optimization. Annotations are also used in other work to explore immutability: Boyland and others [23], for instance, observe that frequently annotations used to define and protect immutable references are defined individually and formalized independently, resulting in arbitrary semantics and an abundance of names with subtle significance differences. Their paper also summarizes some of the more popular annotations used to protect references and describes a system designed to control reference sharing which can model the annotations.

Zibin et al. [126] propose using Java generics and annotations to define and enforce both reference and object immutability. This proposal is partly inspired by Javari [12], a Java extension which includes a type system to express and enforce immutability, and presents Immutability Generic Java, an extension to the Java language and its type system. Neither of the aforementioned work focuses on concurrency and both lack the means to ensure shared data remains consistent, for instance by restricting that only immutable data can be shared.

An example regarding immutability in Java is presented by Boyapati et al. [22]. It proposes a type system to prevent data races and deadlocks in Java and uses ownership types [33] to associate objects with protection mechanisms. Each protection mechanism is defined as part of a variable's type and may refer to the lock that is used to protect access to the object pointed by the variable or indicate that the object may be freely accessed by multiple execution flows. The latter case implies the object is immutable, or it is only accessible by a single execution flow, or the variable holds a unique pointer to the object. Although rather focused on the ordered use of locks, this work is pertinent to our survey as it uses immutability as part of its protection mechanisms and it enforces that objects are associated with a protection mechanism, thus promoting communication in a structured fashion.

Immutable values are natural in functional environments. The Erlang programming language [7] is a good example of the use of immutability in concurrent programming. Erlang has built-in support for concurrency, distribution and fault-tolerance, and is used primarily for soft real-time systems, notoriously for controlling large telecommunication systems from Ericsson [8,9]. Because it follows a functional paradigm, Erlang only supports single assignment variables, which in effect result in immutable objects. It also supports and stimulates massive concurrency by means of multiple lightweight processes that communicate through message passing. Even when executing on a local (non-distributed) environment, all data in messages exchanged among Erlang processes is copied [43]; however, this is transparent to the programmer, and another implementation could be chosen with no impact on the communication semantics [80].

Concurrent Haskell [100] also explores the combination of immutability and concurrency. Concurrent Haskell is an extension to the Haskell programming language. It provides primitive types and operations that allow concurrent execution flows to be created, synchronized and to communicate. Since data is immutable by default in Haskell, execution flows can share data simply by using the same variables. An underlying synchronization mechanism ensures that lazy expressions are only evaluated by one execution flow; if other execution flows try to evaluate an expression which is already being evaluated, they are blocked until the first execution flow finishes evaluation and overwrites the expression with its value. Despite being based on a purely functional programming language, Concurrent Haskell provides a mutable state variable, by means of monads, to allow execution flows to share data and explicitly recognizes the need for such mechanism for a number of reasons outlined by Peyton Jones et al. [100].

Yet another example of immutability used as a building block to support concurrency is the D programming language [3]. It uses message passing to allow for communication among threads and its variables are, by default, local to each thread. Moreover, messages can only contain immutable values, a restriction which is enforced by the compiler. Fig. 8 presents a simple implementation, in D, of a program with two threads that uses immutability to share data. The first thread, defined in the `main` function, creates the second thread with the `spawn` function and the executes a read loop over chunks of unsigned bytes, each with a size equal to `bufferSize`. At each iteration, a new buffer of immutable unsigned bytes is allocated and its values are initialized with the chunk that was read. Then, the buffer is sent to the second thread, which in turn, executes a writer loop, receiving the array and writing its contents to an output. If we changed the buffer type from immutable unsigned bytes (`immutable(ubyte)[]`) to simply unsigned bytes (`ubyte[]`), the call to the `send` function would not compile.

```
1  import std.algorithm, std.concurrency, std.stdio;
2
3  void main() {
4      enum bufferSize = 1024 * 100;
5      auto tid = spawn(&fileWriter);
6      // Read loop
7      foreach (immutable(ubyte)[] buffer;
8               stdin.byChunk(bufferSize)) {
9          send(tid, buffer);
10     }
11 }
12
13 void fileWriter() {
14     // Write loop
15     for (;;) {
16         auto buffer = receiveOnly!(immutable(ubyte)[])();
17         tgt.write(buffer);
18     }
19 }
```

Fig. 8. A simple program in the D language that uses immutability to share data between two threads: a reader and a writer.

3.4.2. Discussion

A disadvantage of immutable objects is the need to create a new object for every distinct value (or set of values). This can represent a performance bottleneck for large objects or for methods where it is necessary to create multiple intermediate objects that will be discarded once a final result is reached. In this case, the performance bottleneck is not only caused by the time taken to create each new object, but also by the potential burden of intense garbage collection.

Another commonly discussed issue regarding immutability lies in the complexity of properly defining its semantics. Pechtchanski and Sarkar [99], for instance, argue that common type modifiers associated with immutability, like `const` and `final`, have limited significance as they cannot be used to express immutability patterns commonly observed in programs. Additional evidence of the complexity associated with the semantics of immutability are presented by Haack et al. in their work to enhance the Java type system to support immutability [57,55].

3.4.3. Structured communication support

Despite being a useful feature for communication, immutability by itself is not enough to ensure structured communication. Most obviously, because allowing both mutable and immutable data to be used in a program, as well as supporting referential semantics to access objects, makes it difficult to enforce that only immutable objects are shared and that they are not changed throughout a program's execution. The semantics of immutability are also not trivial to implement on programming languages that have not considered this model from start.

Working exclusively with immutable data requires a paradigm shift for programmers used to imperative programming languages, as programs must be structured considering that data cannot be modified in place. Even worse, modifying data in place is a fundamental functionality in object-oriented programming languages. This can explain why it is complex to implement and enforce immutability in languages like Java.

Reasonability

The semantics of immutability are simple to understand, in particular when it is used for fine-grained concurrency, yet hard to implement, specially on programming languages like Java, that allow both mutable and immutable objects, as well as supports referential semantics. Immutable data is simple to reason about, as long as its immutable status is explicit. Since no updates are possible, writes are disallowed and concurrent read operations can be safely executed without concerns about data races or unpredictable behaviors. The lack of mechanisms to enforce that data can only be shared as long as it is immutable, however, invalidates that benefit.

Performance

When using immutable data, implementations can benefit from performance optimizations, like passing references instead of copying values; this allows, for instance, the implementation of message-passing semantics without copying data and thus with performance similar to that of shared memory.

Composability

Immutability is well suited as a building block for other models, however it has limited applicability by itself to concurrency and offers no constructs that can be composed. Therefore, we understand the composability property is not applicable.

3.4.4. Summary

A summary of structured communication support with immutability is presented in Table 4.

4. Discussion

In this section we summarize the previous discussions of each surveyed concurrency control model, plus of data immutability. Then, we comment on some of ours findings after having analyzed them.

Monitors, as implemented in Concurrent Pascal, probably represent the concurrency control model which comes the closest to providing structured communication among execution flows. In particular, the enforcement that shared data access can only occur from within monitors is a fundamental aspect in that respect; it simplifies reasoning about the execution of a program that uses monitors and removes from programmers much of the burden of controlling mutual exclusion. Unfortunately, there are no recent implementations of the original monitor model—programming

Table 4
Summary of structured communication support with immutability.

	Immutability
Reasonability	*Predictable*, due to easy to understand semantics for fine-grained concurrency when accessing immutable objects.
Performance	*Restrained*, as implementations can benefit from optimizations such as passing references instead of copying values but must create new objects for distinct values.
Composability	*n/a*

Table 5
Summary of structured communication support in surveyed models.

	Monitors	Transactional memory	Tuple spaces
Reasonability	Predictable	Unpredictable	Fine-grained predictable
Performance	Restrained	Restrained	Optimizable
Composability	Ad hoc	Implemented	Ad hoc

languages such as Java permit access to shared data outside the scope of monitors. We believe that one of the reasons for this is that it is difficult to enforce that data sharing occurs only inside a given syntactic construct when working with memory references.

Transactional memory shares some similarity with monitors, such as controlling communication by defining operations that must be executed atomically. However, differently from monitors, it does not bind the atomic operations to the data accessed during their execution. There are no mechanisms to enforce that shared data is accessed only from within transactions, and therefore transactional memory cannot prevent low-level race conditions and all resulting problems. An exception is Concurrent Haskell, that uses monads and the Haskell type system to prevent free access to shared variables, but there are as yet no proposals for applying this technique to more conventional languages. Additionally, despite all the interest in transactional memory, it still lacks widespread hardware support and it is difficult to implement transactional memory efficiently in software.

Tuple spaces are extensively cited in the literature and implemented in different programming languages. The concept of separating computation from coordination, implemented in the Linda programming language, relates to our concept of structured communication as it allows for clear distinction, in source code, of points where non-deterministic behaviors may occur. However, coarse-grained (or collective) operations in tuple spaces can lead to unpredictable results and require additional constructs to ensure atomicity.

Data immutability is not a concurrency model per se. It lacks the means, by itself, to provide or to enforce structured communication. However, it offers a way to solve one of the main obstacles in the way of structured communication mechanisms, namely using references to share data. Initiatives such as that of the D programming language show that immutability can be used by the compiler to provide the programmer with some guarantees, accomplishing a mechanism akin to message passing in terms of reasonability but with the performance of shared memory. Higher-level, composable constructs have yet to be explored. Moreover, we believe immutability has not yet been thoroughly researched as a building block to support structured communication in concurrency. Its potential suggests that future work could be carried out to improve type systems to better support concurrency and enforce its correctness.

Table 5 presents a summary of structured communication support in each of the surveyed models. The table purposely does not include immutability, as it does not really make sense to compare it side-by-side with the other models.

One result that was somewhat surprising is that we did not find that composability was *modeled* in any of the studied concurrency control mechanisms. In Linda, the lack of support for composability seems well integrated with the design of the model, as authors have explicitly expounded the advantages of allowing programmers to build their own data structures from the fine grained primitives [30], and the idea of nesting is not immediately applicable. In the case of monitors and transactional memory, however, nesting seems to be a rather intuitive occurrence but in neither case was the semantics for nesting thoroughly discussed in initial proposals.

A common pattern we observed when analyzing each model is that authors seem to expect that a single concurrency control model will be enough to address all communication requirements of a program. Examples provided in the literature are usually based on carefully chosen programs, which can be neatly rewritten according to a specific concurrency control model in order to improve the performance or simplify the communication among execution flows. Although fine from a didactic point of view, real-world programs present different communication requirements which can prevent a single concurrency control model from being used, or at least may force the programmer to use a model in unintended or sub-optimal ways just to maintain homogeneity. Using multiple concurrency control models in a single program, on the other hand, can quickly increase the complexity and make reasoning about a program's execution harder; how can a programmer evaluate, and reason about, the composition of communication constructs from different concurrency control models?

We found that the reasonability property is commonly neglected in many research papers on concurrency control models. More often than not, too much effort is placed in improving performance or exploring niche use cases, while too little effort is placed in defining and enforcing precise semantics, as well as in evaluating the practical implications for programmers of using a model in real-world programs. The lack of reasonability is a major cause of the complexity of concurrent programming; it is mostly related to the unstructured communication among multiple execution flows, which usually results in unexpected behaviors.

We understand that the enforcement of the use of constructs limits flexibility, as it is common for programmers to try to improve performance by circumventing standard communication patterns [107]. However, it is our belief that in the vast majority of situations the price for such flexibility is too high; concurrent programming is already inherently complex and very few programmers can (or want to) afford the responsibility of maintaining communication consistency in a concurrent program.

The choice between flexibility and simplicity is a common one. An interesting parallel is with memory management, regarding the choice between manual and automatic memory management [54]. For a long time, programmers despised automatic memory management as too slow for real applications; currently, many programmers and programming languages adopt it, making dangling pointers and memory leaks things of the past. We believe that concurrency must do a similar transition.

5. Conclusion

In this paper we analyzed three concurrency control models – monitors, transactional memory and tuple spaces – plus data immutability and evaluated their support for structured communication among execution flows. We concluded that the analyzed models mostly lack support for structured communication since they do not prevent the use of constructs that can lead to unpredictable results.

One of the major difficulties in enforcing well-behaved accesses to shared data seems to be related to memory references (pointers). The possibility of having complex data structures leak references to nested data seems to be a crucial difficulty for enforcing that data is shared only inside monitors or inside transactions. It is also the source of difficulties in guaranteeing that only immutable values are shared. Thus, it seems that any work on structured communication must from the start consider how to deal with references. The recent work with types and effects seems to be a promising direction [17,18].

Another conclusion is that the analyzed concurrency models have not given much thought to the issue of composability in their design. The composition of concurrent operations is a common requirement, and should be taken into consideration from the start.

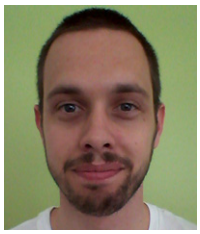
In fact, the design of a programming language must deal with concurrency from the start. As pointed out by Boehm and Adve [20,1], the language memory model itself can prevent the construction of predictable concurrent programs. In the interest of flexibility, languages should probably not have ingrained concurrency models, but it is adamant that they offer sufficient conditions for the implementation of structured models with the enforcement of conditions that guarantee predictability.

References

[1] S.V. Adve, H.-J. Boehm, Memory models: a case for rethinking parallel languages and hardware, Comm. ACM 53 (8) (2010) 90–101.
[2] G. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, MA, USA, 1986.
[3] A. Alexandrescu, The D Programming Language, Addison-Wesley Professional, Upper Saddle River, NJ, USA, 2010.
[4] G.R. Andrews, Paradigms for process interaction in distributed programs, ACM Comput. Surv. 23 (1991) 49–90.
[5] G.R. Andrews, F.B. Schneider, Concepts and notations for concurrent programming, ACM Comput. Surv. 15 (1983) 3–43.
[6] J. Armstrong, Why I don't like shared memory, personal Blog – Armstrong on Software (September 2006). URL http://armstrongonsoftware.blogspot.com/2006/09/why-i-dont-like-shared-memory.html.
[7] J. Armstrong, Programming Erlang: Software for a Concurrent World, Pragmatic Bookshelf, 2007.
[8] J. Armstrong, Erlang — a survey of the language and its industrial applications, in: INAP'96 — The 9th Exhibitions and Symposium on Industrial Applications of Prolog, Hino, Tokyo, Japan, 1996.
[9] J. Armstrong, The development of Erlang, in: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, ICFP'97, ACM, New York, NY, USA, 1997.
[10] D.F. Bacon, R.E. Strom, A. Tarafdar, Guava: a dialect of Java without data races, in: Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA'00, ACM, New York, NY, USA, 2000.
[11] H.E. Bal, J.G. Steiner, A.S. Tanenbaum, Programming languages for distributed computing systems, ACM Comput. Surv. 21 (1989) 261–322.
[12] A. Birka, M.D. Ernst, A practical type system and language for reference immutability, in: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA'04, ACM, New York, NY, USA, 2004.
[13] K.P. Birman, T.A. Joseph, Exploiting virtual synchrony in distributed systems, ACM SIGOPS Oper. Syst. Rev. 21 (1987) 123–138.
[14] A.D. Birrell, B.J. Nelson, Implementing remote procedure calls, ACM Trans. Comput. Syst. 2 (1984) 39–59.
[15] R. Bjornson, N. Carriero, D. Gelernter, T. Mattson, D. Kaminsky, A. Sherman, Experience with Linda, Tech. Rep., Yale University, YALE/DCS/TR866B, 1991.
[16] J. Bloch, Effective Java (The Java Series), second ed., Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.
[17] R.L. Bocchino Jr., V.S. Adve, S.V. Adve, M. Snir, Parallel programming must be deterministic by default, in: Proceedings of the 1st USENIX Conference on Hot Topics in Parallelism, HotPar'09, USENIX Association, Berkeley, CA, USA, 2009.
[18] R. Bocchino Jr., S. Heumann, N. Honarmand, S.V. Adve, V.S. Adve, A. Welc, T. Shpeisman, Safe nondeterminism in a deterministic-by-default parallel language, in: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'11, ACM, New York, NY, USA, 2011, URL http://doi.acm.org/10.1145/1926385.1926447.
[19] H.-J. Boehm, Transactional memory should be an implementation technique, not a programming interface, in: Proceedings of the First USENIX Conference on Hot Topics in Parallelism, HotPar'09, USENIX Association, Berkeley, CA, USA, 2009.
[20] H.-J. Boehm, Threads cannot be implemented as a library, in: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'05, ACM, New York, NY, USA, 2005, URL http://doi.acm.org/10.1145/1065010.1065042.
[21] H.-J. Boehm, S.V. Adve, You don't know jack about shared variables or memory models, Comm. ACM 55 (2) (2012) 48–54.
[22] C. Boyapati, R. Lee, M. Rinard, Ownership types for safe programming: preventing data races and deadlocks, in: Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA'02, ACM, New York, NY, USA, 2002.
[23] J. Boyland, J. Noble, W. Retert, Capabilities for sharing: A generalisation of uniqueness and read-only, in: Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP'01, Springer-Verlag, London, UK, UK, 2001.
[24] P. Bright, IBM's new transactional memory: make-or-break time for multi-threaded revolution, Website (August 2011). URL http://arstechnica.com/hardware/news/2011/08/ibms-new-transactional-memory-make-or-break-time-for-multithreaded-revolution.ars.
[25] J.-P. Briot, R. Guerraoui, K.-P. Lohr, Concurrency and distribution in object-oriented programming, ACM Comput. Surv. 30 (3) (1998) 291–329.
[26] P.A. Buhr, M. Fortier, M.H. Coffin, Monitor classification, ACM Comput. Surv. 27 (1995) 63–107.
[27] P.A. Buhr, A.S. Harji, Implicit-signal monitors, ACM Trans. Program. Lang. Syst. 27 (2005) 1270–1343.
[28] M.G. Burke, K. Knobe, R. Newton, V. Sarkar, Concurrent collections programming model, in: D. Padua (Ed.), Encyclopedia of Parallel Computing, Springer, US, 2011, pp. 364–371.
[29] R.H. Campbell, A.N. Habermann, The specification of process synchronization by path expressions, in: Operating Systems, Proceedings of an International Symposium, Springer-Verlag, London, UK, 1974.
[30] N. Carriero, D. Gelernter, Linda in context, Comm. ACM 32 (1989) 444–458.
[31] N. Carriero, D. Gelernter, Applications experience with Linda, in: Proceedings of the ACM/SIGPLAN Conference on Parallel Programming: Experience with Applications, Languages and Systems, PPEALS'88, ACM, New York, NY, USA, 1988.
[32] C. Cascaval, C. Blundell, M. Michael, H.W. Cain, P. Wu, S. Chiras, S. Chatterjee, Software transactional memory: why is it only a research toy?, ACM Queue 6 (5) (2008) 46–58.
[33] D.G. Clarke, J.M. Potter, J. Noble, Ownership types for flexible alias protection, in: Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA'98, ACM, New York, NY, USA, 1998.
[34] P. Costa, L. Mottola, A.L. Murphy, G.P. Picco, TeenyLIME: transiently shared tuple space middleware for wireless sensor networks, in: Proceedings of the International Workshop on Middleware for Sensor Networks, MidSens'06, ACM, New York, NY, USA, 2006.
[35] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A.L. Murphy, G.P. Picco, TinyLIME: bridging mobile and sensor networks through middleware, in: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications, IEEE Computer Society, Washington, DC, USA, 2005.
[36] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, D. Nussbaum, Hybrid transactional memory, in: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-XII, ACM, New York, NY, USA, 2006.
[37] A.L. de Moura, R. Ierusalimschy, Revisiting coroutines, ACM Trans. Program. Lang. Syst. 31 (2) (2009) 6:1–6:31.
[38] D. Dice, N. Shavit, Understanding tradeoffs in software transactional memory, in: Proceedings of the International Symposium on Code Generation and Optimization, CGO'07, IEEE Computer Society, Washington, DC, USA, 2007.
[39] E.W. Dijkstra, The structure of THE — multiprogramming system, Comm. ACM 11 (1968) 341–346.
[40] E.W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Comm. ACM 18 (8) (1975) 453–457.
[41] A. Dragojević, P. Felber, V. Gramoli, R. Guerraoui, Why STM can be more than a research toy, Comm. ACM 54 (4) (2011) 70–77.
[42] R. Ennals, Software Transactional Memory Should not be Obstruction-free, Tech. Rep. IRC-TR-06-052, Intel Research Cambridge Tech. Report (January 2006). URL http://www.cs.wisc.edu/trans-memory/misc-papers/052_Rob_Ennals.pdf.
[43] A.B. Ericsson, Erlang/OTP System Documentation, erlang/OTP System Documentation 5.8.5, October 2011.
[44] N. Ford, Functional thinking: immutability — make Java code more functional by changing less (July 2011). URL http://www.ibm.com/developerworks/java/library/j-ft4.

[45] J.S. Foster, M. Fähndrich, A. Aiken, A theory of type qualifiers, in: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, PLDI'99, ACM, New York, NY, USA, 1999.
[46] E. Freeman, K. Arnold, S. Hupfer, JavaSpaces Principles, Patterns, and Practice, first ed., Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.
[47] V. Gajinov, F. Zyulkyarov, O.S. Unsal, A. Cristal, E. Ayguade, T. Harris, M. Valero, QuakeTM: parallelizing a complex sequential application using transactional memory, in: Proceedings of the 23rd International Conference on Supercomputing, ICS'09, ACM, New York, NY, USA, 2009.
[48] D. Gelernter, Generative communication in Linda, ACM Trans. Program. Lang. Syst. 7 (1985) 80–112.
[49] D. Gelernter, N. Carriero, Coordination languages and their significance, Comm. ACM 35 (1992) 97–107.
[50] B. Goetz, Java theory and practice: to mutate or not to mutate? – Immutable objects can greatly simplify your life (February 2003). URL http://www.ibm.com/developerworks/java/library/j-jtp02183.
[51] S. Gorlatch, Send-receive considered harmful: Myths and realities of message passing, ACM Trans. Program. Lang. Syst. 26 (2004) 47–56.
[52] J. Gray, The transaction concept: virtues and limitations (invited paper), in: Proceedings of the seventh International Conference on Very Large Data Bases, VLDB'1981, vol. 7, VLDB Endowment, 1981.
[53] W. Gropp, E.L. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message Passing Interface, second ed., MIT Press, 1999.
[54] D. Grossman, The transactional memory/garbage collection analogy, in: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA'07, ACM, New York, NY, USA, 2007.
[55] C. Haack, E. Poll, Type-based object immutability with flexible initialization, in: ECOOP 2009, in: LNCS, vol. 5653, Springer, 2009.
[56] C. Haack, E. Poll, J. Schäfer, A. Schubert, Immutable objects in Java, Dept. of Computer Science ICIS-R06010, Radboud University Nijmegen, 2006.
[57] C. Haack, E. Poll, J. Schäfer, A. Schubert, Immutable objects for a Java-like language, in: R.D. Nicola (Ed.), ESOP'07, in: LNCS, vol. 4421, Springer, 2007.
[58] B.K. Haddon, Nested monitor calls, ACM SIGOPS Oper. Syst. Rev. 11 (1977) 18–23.
[59] P. Haller, M. Odersky, Event-based programming without inversion of control, in: Proceedings of the Joint Modular Languages Conference, in: Springer LNCS, 2006.
[60] P.B. Hansen, Concurrent programming concepts, ACM Comput. Surv. 5 (1973) 223–245.
[61] P.B. Hansen, Structured multiprogramming, Comm. ACM 15 (1972) 574–578.
[62] P.B. Hansen, An outline of a course on operating system principles, in: C.A.R. Hoare, R.H. Perrott (Eds.), Operating Systems Techniques, in: A.P.I.C. Studies in Data Processing, vol. 9, Academic Press, London, 1972.
[63] P.B. Hansen, Operating System Principles, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973.
[64] P.B. Hansen, Monitors and Concurrent Pascal: a personal history, in: The second ACM SIGPLAN Conference on History of Programming Languages, HOPL-II, ACM, New York, NY, USA, 1993.
[65] P.B. Hansen, The programming language Concurrent Pascal, IEEE Trans. Softw. Eng. 1 (2) (1975) 199–207.
[66] P.B. Hansen, Java's insecure parallelism, ACM SIGPLAN Notices 34 (1999) 38–45.
[67] P.B. Hansen, The invention of concurrent programming, in: The Origin of Concurrent Programming, Springer-Verlag New York, Inc., New York, NY, USA, 2002, pp. 3–61.
[68] P.B. Hansen, A programming methodology for operating system design, in: IFIP Congress, 1974.
[69] T. Harris, Language constructs for transactional memory, in: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'09, ACM, New York, NY, USA, 2009.
[70] T. Harris, J. Larus, R. Rajwar, Transactional Memory, second ed., Morgan and Claypool Publishers, 2010.
[71] T. Harris, S. Marlow, S. Peyton-Jones, M. Herlihy, Composable memory transactions, in: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'05, ACM, New York, NY, USA, 2005.
[72] M. Herlihy, Wait-free synchronization, ACM Trans. Program. Lang. Syst. 13 (1991) 124–149.
[73] M. Herlihy, Technical perspective: Highly concurrent data structures, Comm. ACM 52 (2009) 99.
[74] M. Herlihy, V. Luchangco, P. Martin, M. Moir, Nonblocking memory management support for dynamic-sized data structures, ACM Trans. Program. Lang. Syst. 23 (2005) 146–196.
[75] M. Herlihy, V. Luchangco, M. Moir, W.N. Scherer III, Software transactional memory for dynamic-sized data structures, in: Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing, PODC'03, ACM, New York, NY, USA, 2003.
[76] M. Herlihy, J.E.B. Moss, Transactional memory: architectural support for lock-free data structures, in: Proceedings of the 20th Annual International Symposium on Computer Architecture, ISCA'93, ACM, New York, NY, USA, 1993.
[77] C.A.R. Hoare, Towards a theory of parallel programming, in: C.A.R. Hoare, R.H. Perrott (Eds.), Operating Systems Techniques, in: A.P.I.C. Studies in Data Processing, vol. 9, Academic Press, London, 1972.
[78] C.A.R. Hoare, Monitors: an operating system structuring concept, Comm. ACM 17 (1974) 549–557.
[79] J.H. Howard, Signaling in monitors, in: Proceedings of the 2nd International Conference on Software Engineering, ICSE'76, IEEE Computer Society Press, Los Alamitos, CA, USA, 1976.
[80] E. Johansson, K. Sagonas, J. Wilhelmsson, Heap architectures for concurrent languages using message passing, in: Proceedings of the 3rd International Symposium on Memory Management, ISMM'02, ACM, New York, NY, USA, 2002.
[81] S.P. Jones, Beautiful concurrency, in: G. Wilson, A. Oram (Eds.), Beautiful Code: Leading Programmers Explain How They Think, OReilly Media, Inc., 2007, pp. 385–406 (chapter 24).
[82] A. Kauppi, Lualanes — multithreading in Lua, Website (2009). URL http://kotisivu.dnainternet.net/askok/bin/lanes/.
[83] J.L. Keedy, On structuring operating systems with monitors, ACM SIGOPS Oper. Syst. Rev. 13 (1979) 5–9.
[84] J.L.W. Kessels, An alternative to event queues for synchronization in monitors, Comm. ACM 20 (1977) 500–503.
[85] L. Kotulski, About the semantic nested monitor calls, ACM SIGPLAN Notices 22 (1987) 80–82.
[86] H.C. Lauer, R.M. Needham, On the duality of operating system structures, ACM SIGOPS Oper. Syst. Rev. 13 (1979) 3–19.
[87] E.A. Lee, Disciplined Message Passing, Tech. Rep., EECS Dept. University of California Berkeley, January 2009.
[88] E.A. Lee, The problem with threads, IEEE Computer 39 (5) (2006) 33–42.
[89] Y. Lev, J.-W. Maessen, Toward a safer interaction with transactional memory by tracking object visibility, in: Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages, San Diego, CA, 2005.
[90] A. Lister, The problem of nested monitor calls, ACM SIGOPS Oper. Syst. Rev. 11 (1977) 5–7.
[91] M. Martin, C. Blundell, E. Lewis, Subtleties of transactional memory atomicity semantics, IEEE Comput. Architect. Lett. 5 (2) (2013).
[92] S.W. McLaughry, P. Wyckoff, T Spaces: the next wave, in: Proceedings of the Thirty-second Annual Hawaii International Conference on System Sciences, vol. 8, HICSS'99, IEEE Computer Society, Washington, DC, USA, 1999.
[93] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, version 3.0 (September 21, 2012).
[94] M.M. Michael, Hazard pointers: Safe memory reclamation for lock-free objects, IEEE Trans. Parallel Distrib. Syst. 15 (2004) 491–504.
[95] J.E.B. Moss, A.L. Hosking, Nested transactional memory: model and architecture sketches, Sci. Comput. Program. 63 (2) (2006) 186–201.
[96] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, M. Zenger, An Overview of the Scala Programming Language, Tech. Rep., EPFL Lausanne, Switzerland, 2004.
[97] J. Ousterhout, Why threads are a bad idea (for most purposes), Presentation given at the 1996 USENIX Annual Technical Conference.
[98] D.L. Parnas, The non-problem of nested monitor calls, ACM SIGOPS Oper. Syst. Rev. 12 (1978) 12–18.
[99] I. Pechtchanski, V. Sarkar, Immutability specification and its applications, in: Java Grande/ISCOPE 2002, Concurrency and Computation: Practice and Experience 17 (5–6) (2005) 639–662 (special issue).
[100] S. Peyton Jones, A. Gordon, S. Finne, Concurrent Haskell, in: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, ACM, 1996.
[101] G.P. Picco, A.L. Murphy, G.-C. Roman, LIME: Linda meets mobility, in: Proceedings of the 21st International Conference on Software Engineering, ICSE'99, ACM, New York, NY, USA, 1999.
[102] B.C. Pierce, Types and Programming Languages, MIT Press, Cambridge, MA, USA, 2002.
[103] R. Rajwar, J.R. Goodman, Transactional lock-free execution of lock-based programs, in: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-X, ACM, New York, NY, USA, 2002.
[104] R. Rajwar, M. Herlihy, K. Lai, Virtualizing transactional memory, in: Proceedings of the 32nd Annual International Symposium on Computer Architecture, ISCA'05, IEEE Computer Society, Washington, DC, USA, 2005.
[105] J. Reinders, Transactional synchronization in Haswell, Website (February 2012). URL http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/.
[106] Rinda — a module to implement the Linda distributed computing paradigm in Ruby, Website, 2011. URL http://www.ruby-doc.org/stdlib-1.9.3/libdoc/rinda/rdoc/Rinda.html.
[107] Z.B. Rui Zhang, W.N. Scherer III, Composability for application-specific transactional optimizations, in: Proceedings of the 5th ACM SIGPLAN Workshop on Transactional Computing, TRANSACT 2010, 2010.
[108] J.L. Schilling, Dynamically-valued constants: an underused language feature, ACM SIGPLAN Notices 30 (1995) 13–20.
[109] Scientific Computing Associates Inc., Linda User Guide (September 2005). URL http://www.lindaspaces.com/downloads/lindamanual.pdf.
[110] N. Shavit, D. Touitou, Software transactional memory, in: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, PODC'95, ACM, New York, NY, USA, 1995.
[111] H. Sutter, The trouble with locks, Dr. Dobb's Journal. (March) (2005).
[112] H. Sutter, Lock-free code: A false sense of security, Dr. Dobb's Journal. (September) (2008).
[113] H. Sutter, J. Larus, Software and the concurrency revolution, ACM Queue 3 (2005) 54–62.

[114] TCP Linda case studies, last accessed September 2013. URL http://www.lindaspaces.com/casestudies/index.html.
[115] E. Vallejo, T. Harris, A. Cristal, O. Unsal, M. Valero, Hybrid transactional memory to accelerate safe lock-based transactions, in: 3rd ACM SIGPLAN Workshop on Transactional Computing, TRANSACT 2008, Salt Lake City, UT, USA, 2008.
[116] R. von Behren, J. Condit, E. Brewer, Why events are a bad idea (for high-concurrency servers), in: Proceedings of the 9th Conference on Hot Topics in Operating Systems – Volume 9, USENIX Association, Berkeley, CA, USA, 2003.
[117] G.C. Wells, A programmable matching engine for application development in Linda, Ph.D. Thesis, University of Bristol, July 2001.
[118] G. Wells, Coordination languages: back to the future with Linda, in: Proceedings of WCAT'05, 2005.
[119] G.C. Wells, New and improved: Linda in Java, in: Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java, PPPJ'04, Trinity College Dublin, 2004.
[120] G.C. Wells, A.G. Chalmers, P.G. Clayton, Linda implementations in Java for concurrent systems: research articles, Concurrency and Computation: Practice and Experience 16 (2004) 1005–1022.
[121] G. Wells, P. Clayton, A.G. Chalmers, A comparison of Linda implementations in java, in: P.H. Welch, A.W.P. Bakkers (Eds.), Communicating Process Architectures 2000, 2000.
[122] H. Wettstein, The problem of nested monitor calls revisited, ACM SIGOPS Oper. Syst. Rev. 12 (1978) 19–23.
[123] A. Wilkinson, PyLinda – an implementation of the tuplespace based distributed computing system, Website (2011). URL http://pypi.python.org/pypi/linda/0.5.1.
[124] R.M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, H.-H.S. Lee, Kicking the tires of software transactional memory: why the going gets tough, in: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA'08, ACM, New York, NY, USA, 2008.
[125] S.E. Zenith, Process interaction models, Ph.D. Thesis, Ecole Nationale Supérieure des Mines de Paris, Centre de Recherche en Informatique – 35 rue Saint-Honore 77305 – Fontainebleau – France (1992).
[126] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kiezun, M.D. Ernst, Object and reference immutability using Java generics, in: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE'07, ACM, New York, NY, USA, 2007.



Alexandre Skyrme is a Ph.D. student at PUC-Rio (the Pontifical Catholic University of Rio de Janeiro). His research interests include concurrency, parallelism, distributed computing, computer networks and information security.

Noemi Rodriguez is an Associate Professor of Computer Science at PUC-Rio (the Pontifical Catholic University of Rio de Janeiro), where she works with concurrent and distributed programming.

Roberto Ierusalimschy is an Associate Professor of Computer Science at PUC-Rio (the Pontifical Catholic University of Rio de Janeiro), where he works with programming-language design and implementation.