# Chapter 1

# Introduction

## 1.1 Context

### The Protocol Concern

Since the late 1950s, hardware manufacturers double the number of transistors on chips roughly every two years, as Moore predicted already in 1965 [Moo98]. Until the early 2000s, hardware manufacturers used this exponential increase in transistors for speeding up *unicore processors*, capable of processing exactly one instruction stream. As a result, software engineers enjoyed a "free lunch" during the second half of the twentieth century: without effort from their side, every new generation of unicore processors executed existing programs twice as fast as those of the generation before. Unfortunately, the free lunch ended in 2005 [Sut05]. Although *Moore's Law* continued to hold, hardware manufacturers ran into three major obstacles—"walls"—that prevented them from directing the still exponential increase in transistors further toward faster unicore processors [ABC+06].

First, energy consumption grows disproportionately with clock frequency: a linear increase in clock frequency requires a quadratic (or worse) increase in energy consumption [Pos14]. Beside environmental and financial repercussions, this *Power Wall* causes higher-frequency unicore processors to generate more heat than conventional cooling technology can dissipate, ultimately leading to hardware failure. Second, memory accesses generally take substantially more time than do instructions [WM95]. This *Memory Wall* makes increasing the clock frequency of unicore processors beyond a certain threshold ineffective. Third, techniques for finding implicit *instruction-level parallelism* (ILP) in single instruction streams (e.g., branch prediction, out-of-order execution), useful for keeping unicore processors busy during delays (e.g., memory accesses), seem to have reached their limits [HP11a]. This ILP *Wall* calls for more explicit means of expressing, controlling, and exploiting parallelism.

To mitigate the previous three walls, hardware manufacturers switched from unicore processors to *multicore processors* around 2005: instead of equip-

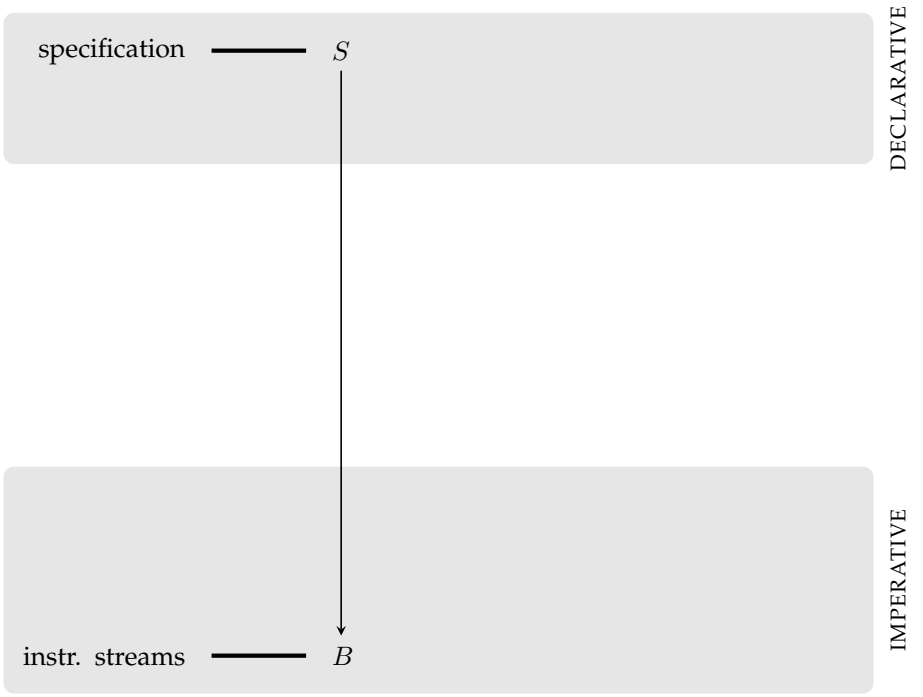


Figure 1.1: From a declarative specification $S$ to its imperative implementation as instruction streams $B$.

ping processors with one fast computational core (say, 5–7 GHz), hardware manufacturers started using the still exponential increase in transistors for constructing processors with multiple slower cores (say, 2–3 GHz). Although this move to multicore technology has enabled hardware manufacturers to keep up with Moore's Law, software engineers of *sequential programs* no longer enjoy free exponential speedups: a multicore processor with a single core executes a sequential program just as fast as one with two, four, or any other number of cores. To harness the power of *today's* multicore processors, therefore, software engineers must write *parallel programs* capable of dividing instructions evenly over, for instance, all twelve cores of Intel's modern E5-2690V3 processor. More importantly, however, to harness the power of *tomorrow's* multicore processors, software engineers must write parallel programs also capable of dividing instructions evenly over 32 or 1024 cores in the future—today's parallel programs must *scale*. Only if their parallel programs exhibit scalability can software engineers again enjoy a free lunch.

Conceptually, every parallel "program-in-execution" consists of (i) a number of *workers*, which perform the actual *computation*, and (ii) a number of *protocols*, which state the rules of *interaction* that the workers must abide by. Interaction covers both *communication* (e.g., a worker sends a value to another worker)

and *synchronization* (e.g., a worker waits for another worker). Software engineers of parallel programs essentially bridge a gap between:

- high-level *declarative specifications* of workers and protocols—some vague idea in their minds, a number of semiformal UML charts, or maybe even temporal logic formulas!—which abstractly define *what* must happen;
- low-level *imperative implementations* as instruction streams, which concretely define *how* things happen.

Figure 1.1 shows this gap. Instead of writing instruction streams directly, however, software engineers usually write their parallel programs as higher-level code from which a compiler later derives lower-level instruction streams. Typically, such parallel programs consist of (i) *computation code* for *worker subprograms* and (ii) *interaction code* for *protocol subprograms*. Note that sequential programs constitute the special class of parallel programs consisting of one worker subprogram and zero protocol subprograms. Henceforth, I therefore no longer distinguish between sequential programs and parallel program, simply writing "program" to refer to any kind of parallel program.

Ideally, twice as many cores execute scalable programs twice as fast. Such scalability, however, does not come easily. In fact, already in the late 1960s, Amdahl discovered that many programs—except those of the "embarrassingly parallel" kind, which require very few to no interaction—cannot indefinitely benefit from more parallel processing [Amd67]. In particular, Amdahl argued that on $n > 1$ cores, the execution time of a program, denoted by $time(n)$, depends on its truly parallel fraction, denoted by $0 \leq par \leq 1$, and its execution time on a single core, denoted by $time(1)$, as follows:

$$time(n) = \left((1 - par) + \frac{par}{n}\right) \cdot time(1) \tag{1.1}$$

Defining the speedup of a program—a measure for its scalability—as $time(1)$ divided by $time(n)$, Amdahl subsequently derives the following equation:

$$speedup_{\text{A}}(n) = \frac{1}{(1 - par) + \frac{par}{n}} \tag{1.2}$$

This equation reveals that even with infinitely many cores (i.e., if $n \to \infty$), the sequential fraction $1 - par$ of a program bounds its scalability. For instance, according to *Amdahl's Law*, a program with $par = 0.9$ can achieve only a 10-fold speedup at best. More concretely, on Intel's previously mentioned E5-2690V3 processor, a program with $par = 0.9$ achieves only a 5.7-fold speedup, despite this processor having as much as twelve cores.

A recent study by Yavitz et al. makes the previous analysis even more serious [YMG14]. By explicitly accounting for the execution time of protocol subprograms, denoted by $protocols(n)$, Yavitz et al. derive the following equation:

$$speedup_{\text{Y}}(n) = \frac{1}{(1 - par) + \frac{par}{n} + protocols(n)} \tag{1.3}$$

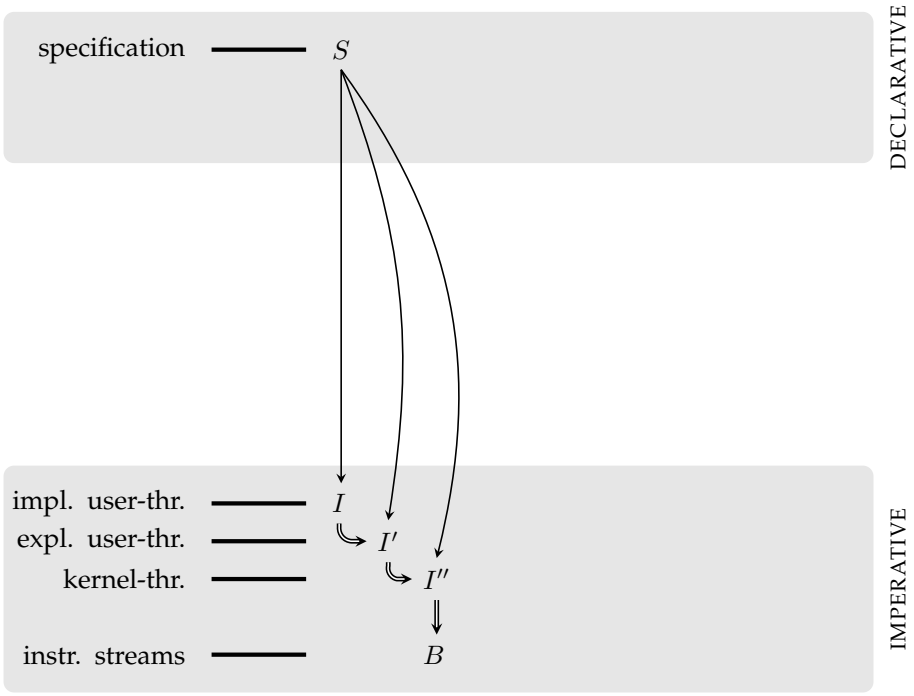


Figure 1.2: From a declarative specification $S$ to its imperative implementation as instruction streams $B$, possibly past several intermediate imperative implementations $I$, $I'$, and $I''$ at increasingly low levels of abstraction: implicit user-threads, explicit user-threads, and kernel-threads. Single arrows represent manual labor by a software engineer; double arrows represent automatic labor by a compiler.

(As this equation suggests, the execution time of protocol subprograms depends on the number of cores. The exact shape of this dependency differs between programs: as $n$ approaches infinity, at least $protocols(n) > 0$ and for some programs even $protocols(n) \to \infty$.) In the analysis of Yavitz et al., thus, not only its sequential fraction but also its protocol subprograms bound the scalability of a program. Indeed, maximizing $par$—suggested by Amdahl's Law—improves scalability only to some extent, while reducing $protocols(n)$ becomes increasingly important as $par$ increases.

The work of Yavitz et al. implies that to enjoy a free lunch in this multicore era, software engineers better write efficient and scalable protocol subprograms. I call this *The Protocol Concern*.

## Today's Abstractions

Fortunately, software engineers do not need to bridge the gap in Figure 1.1 all by themselves: several *levels of abstraction* on top of the hardware hide increasingly more details that software engineers rather not concern themselves with. Figure 1.2 shows three major such levels. Each of these levels supports some form of concurrent "subprograms-in-execution" called *threads* that together constitute a full "program-in-execution". All threads belonging to the same program-in-execution have access to the same piece of memory by default. Threads can use this *shared memory* for interacting with each other.

The levels of abstraction in Figure 1.2 differ in the extent to which software engineers manually manage threads in their programs.

- Operating systems provide the first level of abstraction: *kernel-threads* in kernel space, where an operating system *schedules* kernel-threads on cores (i.e., an operating system decides about which kernel-thread runs on which core at which time). Software engineers can manage kernel-threads directly with *system calls* to the operating system. Most software engineers, however, use higher-level *application programming interfaces* (API). I therefore ignore this level of abstraction as a serious alternative for general-purpose software engineering and mention it only as the foundation of higher-level APIs.

- Programming languages provide the second and third level of abstraction: *explicit* or *implicit user-threads* in user space, where the implementation of a user-threading API schedules user-threads on kernel-threads.

  Explicit user-threading APIs hide all system calls required for directly managing kernel-threads from software engineers. Examples include POSIX threads, Windows threads, and Java threads. Using these APIs, software engineers implement their worker specifications explicitly as user-threads: to write a program using an explicit user-threading API, software engineers organize that program into a number of worker subprograms, each of which explicitly defines a user-thread, and protocol subprograms. Afterward, the implementation of the API takes care of concurrently executing those worker subprograms, as user-threads.

  Implicit user-threading APIs, in contrast, hide from software engineers not only kernel-thread management but also to a large extent user-thread management. Examples include *thread pools* (e.g., OpenMP [DM98], Intel Cilk Plus [Rob13], Intel TBB [Rei07], Microsoft TPL [LSB09], Apple GCD, Java executors [GPB+06]) and *actors* (e.g., ERLANG processes [AVWW96], Scala/Akka actors [HO09, Hal12]).

  – Using thread pool APIs, software engineers implement their worker specifications as *tasks*, subprograms typically smaller than subprograms for explicitly defined user-threads. At run-time, tasks get submitted to a queue monitored by a pool of user-threads. Whenever this queue becomes nonempty, a dormant user-thread from the

pool awakes and starts working on the new task. If every user-thread in the pool already has work to do, the new task remains pending until one of those user-threads runs out of work.

To write a program using a thread pool API, software engineers organize that program into a number of worker subprograms, each of which defines a task, and protocol subprograms. Afterward, the implementation of the API takes care of managing both the pool, the queue, and a limited form of interaction among tasks. Some thread pool APIs also provide user-friendly templates for common task submission patterns (e.g., parallel loop iterations).

– Using actor APIs, software engineers implement their worker specifications as *actors*, subprograms typically smaller than subprograms for explicitly defined user-threads. At run-time, actors mix performing computation with exchanging asynchronous messages in an event-driven fashion. Whenever an actor sends a message to another actor, that message first arrives in the receiving actor's *mailbox*. Once an actor has finished processing a message, it selects a next message from its mailbox, should one exist. While processing messages, actors perform computation and may send messages to other actors. The *pure* actor abstraction hides the underlying user-threads' shared memory: pure actors communicate with each other only through asynchronous messaging.

  To write a program using an actor API, software engineers organize that program into a number of worker subprograms, each of which defines an actor, and protocol subprograms. Afterward, the implementation of the API takes care of buffering sent messages in mailboxes, selecting the next buffered message for processing, and scheduling actors on user-threads.

User-threading APIs usually provide a number of special *concurrency constructs* for implementing protocol specifications. For instance, explicit user-threading APIs typically provide concurrency constructs such as *atomic registers* [Lam86], *semaphores* [Dij02], or *monitors* [Hoa74]. These constructs guarantee *mutual exclusion*, to avoid *data races* among threads that interact with each other through shared memory, by protecting that memory from hazardous concurrent accesses. Essentially, software engineers use concurrency constructs to constrain the ways in which nondeterministic schedulers may schedule user-threads on kernel-threads and kernel-threads on cores. Lee calls this an exercise in "pruning nondeterminism" [Lee06]. For instance, while some thread performs an operation on an atomic register, no scheduler may schedule another thread to simultaneously perform an operation on the same register. Similarly, as long as a semaphore has no permits to release, no scheduler may schedule a thread to run past so-far failed attempts to acquire one. (Attempts to acquire a permit from an empty semaphore fail until another thread releases one.)

Without scheduling constraints imposed by concurrency constructs for mutual exclusion, interaction through shared memory may occur in *unsafe* ways.

```
1  public class UnsafeProducersConsumerProgram {
2    private volatile Object buffer;
3
4    public UnsafeProducersConsumerProgram() {
5      (new Producer()).start();
6      (new Producer()).start();
7      (new Consumer()).start();
8    }
9
10   private class Producer extends Thread {
11     public void run() {
12       while (true) {
13         Object datum = Thread.currentThread().getId();
14         while (buffer != null);
15         buffer = datum;
16   } } }
17
18   private class Consumer extends Thread {
19     public void run() {
20       while (true) {
21         while (buffer == null);
22         Object datum = buffer;
23         System.out.println(datum);
24         buffer = null;
25 } } } }
```

Figure 1.3: Unsafe producers/consumer program for LateAsyncMerger2 in Java

To illustrate this point, suppose that I must write a program that consists of three workers: two *producers*, which repeatedly produce data (e.g., fetch their own id) and send data, and one *consumer*, which repeatedly receives data and consumes data (e.g., print the received ids to the console). My protocol specification states that the producers communicate their data to the consumer:

- asynchronously: a producer proceeds after its send before the consumer has completed a corresponding receive;
- reliably: the consumer receives all data sent unchanged;
- unordered: the producers send their data in any order;
- transactionally: a send and its corresponding receive occur atomically (i.e., after a send by a producer, no producer can send until the consumer has completed a corresponding receive)

I call this protocol LateAsyncMerger2. Figure 1.3 shows an unsafe Java program for LateAsyncMerger2: although Java's shared memory straightforwardly supports asynchronous and unordered communication, the program in Figure 1.3 guarantees neither transactionality nor reliability, because it lacks concurrency constructs for mutual exclusion. For instance, both producers may, at the same time, evaluate shared variable `buffer` to **null** (line 14), after which both of

```
1  public class SafeProducersConsumerProgram {
2    private volatile Object buffer;
3    private Semaphore notEmpty;
4    private Semaphore notFull;
5
6    public SafeProducersConsumerProgram() {
7      notEmpty = new Semaphore(0);
8      notFull = new Semaphore(1);
9      (new Producer()).start();
10     (new Producer()).start();
11     (new Consumer()).start();
12   }
13
14   private class Producer extends Thread {
15     public void run() {
16       while (true) {
17         Object datum = Thread.currentThread().getId();
18         notFull.acquire();
19         buffer = datum;
20         notEmpty.release();
21   } } }
22
23   private class Consumer extends Thread {
24     public void run() {
25       while (true) {
26         notEmpty.acquire();
27         Object datum = buffer;
28         notFull.release();
29         System.out.println(datum);
30 } } } }
```

Figure 1.4: Safe producers/consumer program for LateAsyncMerger2 in Java

them write the value of their local variable `datum` to `buffer` (line 15). If the second producer to perform this write does so before the consumer reads `buffer` (line 22), the datum sent by the first producer gets lost, thereby violating both transactionality and reliability. In contrast, Figure 1.4 shows a *safe* Java program that does satisfy all four protocol requirements. It uses two semaphores. These semaphores guarantee that between a write to `buffer` by a producer and a read of `buffer` by the consumer, no producer overwrites `buffer`.

In this subsection, I mentioned several abstractions that today's software engineers use to write programs for multicore processors. I tried to focus on "leading" thread-based technology, based on recent literature: Poss discusses POSIX threads, Java threads, OpenMP, and Intel TBB [Pos14], Silberschatz et al. discuss POSIX threads, Windows threads, Java threads, OpenMP, Apple GCD, Java executors, and Intel TBB [SGG13], while Vajda et al. discuss OpenMP, Intel Cilk Plus, Intel TBB, Microsoft TPL, Apple GCD, and ERLANG actors [Vaj11]. (Poss remarks, however, that given the relatively short time since the introduction of the first multicore processors, much of the new technology developed has neither matured nor stabilized yet, and he expects the landscape to change in the coming decade.) Despite all these abstractions, many of today's software

engineers still need to manually address The Protocol Concern with *classical* concurrency constructs for mutual exclusion, invented many decades ago. In fact, even software engineers who use implicit user-threading APIs often need to resort to such constructs from underlying explicit user-threading APIs to implement their protocol specifications. For instance, pure actor APIs hide the underlying shared memory and provide only asynchronous messaging constructs to implement protocol specifications. While this works well for simple asynchronous communication protocols, it may complicate controlling other kinds of interaction. Some actor APIs therefore allow software engineers to mix asynchronous messaging with shared memory interaction, thereby breaking the pure agent abstraction and necessitating the exposure and usage of concurrency constructs for mutual exclusion (provided by the underlying explicit user-threading API). For instance, Tasharofi et al. discovered that software engineers often mix Scala/Akka actors with Java threads [TDJ13]. The same holds for implementing protocol specifications beyond those implemented inside thread pool APIs. On all threading levels of abstraction, thus, concurrency constructs for mutual exclusion play a crucial role in contemporary software engineering for multicore processors.

## 1.2 Problem

### Three Major Issues

Although explicit and implicit user-threading APIs narrow the conceptual specification/implementation gap in Figure 1.1, a rather wide gap still remains in Figure 1.2. One may concretely measure the size of this gap by comparing the textual length of a specification to the number of lines of code of its implementation. Alternatively, one may more abstractly measure the size of this gap in terms of the kinds of *resources*, physical or virtual, that software engineers need to manually manage in their programs. (Generally, the more such resources software engineers needs to manage, the more details they need to concern themselves with, and the more lines of code their programs consist of.) For instance, at the lowest level of abstraction, software engineers manually manage physical processors. One level above, software engineers no longer manage physical processors but only their virtual representations as kernel-threads in kernel space. One level above, software engineers no longer manage kernel-threads but only their representations as user-threads in user space. One level above, software engineers no longer manage user-threads but higher-level abstractions on top of those user-threads (e.g., thread pools, actors). Even at this highest level in Figure 1.2, however, software engineers still manually manage one particularly significant resource: shared memory, to lesser or greater degree (e.g., with or without automated garbage collection).

As explained in the previous subsection, to manually manage shared memory, software engineers require concurrency constructs for mutual exclusion. Such constructs give rise to three major issues that software engineers face

when addressing The Protocol Concern.

- *Issue 1: Writing correct protocol subprograms*

  Although widely used, concurrency constructs for mutual exclusion provoke controversy: their use inflicts unreasonable demands on the reasoning capabilities of software engineers, notably because of the unpredictable ways in which threads interact with each other [CL00]. After all, software engineers often cannot predict every way in which schedulers may seemingly nondeterministically schedule threads. Consequently, hazardous executions may arise out of unforeseen schedules. Examples of resulting bugs include data races, exemplified already in Figure 1.3, and *deadlocks*, where interdependent threads fail to make progress.

  Lee wrote a seminal essay on the problems of controlling seemingly nondeterministic schedulers with concurrency constructs for mutual exclusion [Lee06]. Lee argues that although explicit user-threading APIs comprise only a minor syntactic change to conventional sequential languages, their use has profound—and hard to manage—semantic repercussions: suddenly, also schedulers affect programs' functional semantics. As a solution, Lee proposes to discard interaction through shared memory, thereby eliminating the influence of nondeterministic schedulers on interaction. Instead, workers should interact with each other only via well-defined interfaces to protocols, where software engineers can judiciously introduce nondeterminism just whenever necessary.

  Arbab provides a different perspective on the same issue [Arb11]. Generally, programs without concurrency constructs for mutual exclusion allow every possible instance of interaction to occur at any time. By imposing mutual exclusion to (parts of) such programs, software engineers essentially constrain which of those instances actually may occur. In this approach to constraining schedulers, however, neither interaction nor protocols comprise first-class entities: as Arbab observes, *inter*action becomes only an "implicit, nebulous and intangible" byproduct of *action* (i.e., sequences of reads/writes to shared memory, mixed with operations of concurrency constructs). Arbab argues that this implicitness seriously complicates implementing protocol specifications. After all, one can hardly reason about something that one cannot easily see. Although the *idea* of constraining which instances of interaction may occur has no fundamental shortcomings, actually imposing such constraints through hand-written code in an action-based programming model—one without first-class constructs for interaction/protocols—demands prohibitively great intellectual effort and ingenuity from software engineers.

- *Issue 2: Writing efficient/scalable protocol subprograms*

  Even if software engineers succeed in writing correct interaction code, writing interaction code that performs well constitutes a whole other challenge. This challenge has two components. On the one hand, soft-

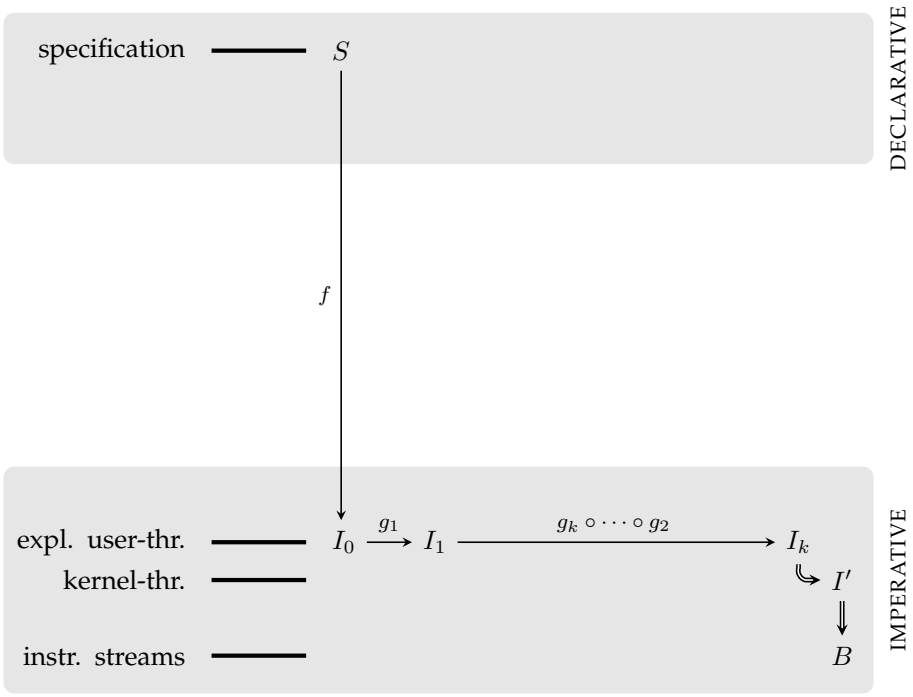


Figure 1.5: From a declarative specification $S$ to its imperative implementation as instruction streams $B$, including $k$ optimizations between intermediate imperative implementations $I_0, \ldots, I_k$ (at the explicit user-threads level). Single arrows represent manual labor by a software engineer; double arrows represent automatic labor by a compiler.

ware engineers should strive for *efficient* protocol subprograms, by minimizing the absolute resource consumption of their interaction code. On the other hand, software engineers should strive for *scalable* protocol subprograms, by minimizing the increase in resource consumption as parallelism increases. A highly efficient protocol subprogram may have poor scalability, while a highly scalable protocol subprogram may have poor efficiency. Software engineers should therefore strive for both.

Figure 1.5 exemplifies the process typically involved in writing efficient/scalable interaction code. First, software engineers transform specification $S$ into an initial implementation $I_0$ using shared memory and concurrency constructs for mutual exclusion (provided by an explicit user-threading API), denoted by arrow $f$. Subsequently, these software engineers incrementally improve the protocol subprograms in $I_0$ into implementations $I_1, \ldots, I_k$ by applying a number of *protocol optimizations* (e.g., introducing more fine-grained locking in a concurrent queue for asynchronous communication), denoted by arrows $g_1, \ldots, g_k$. Finally, a

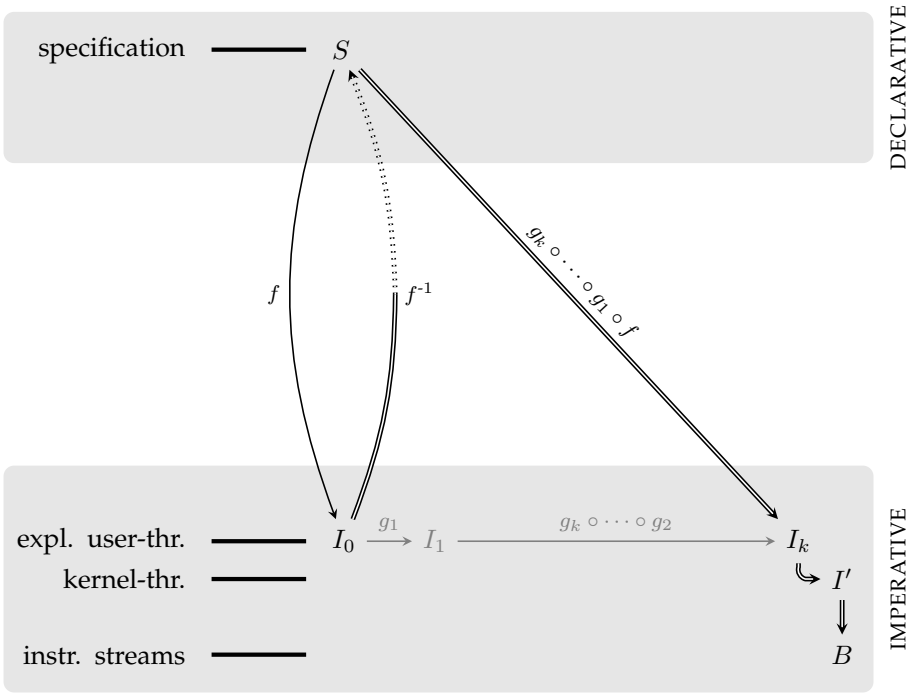


Figure 1.6: Irrecoverability of a specification $S$ from its implementation $I_0$. Single arrows represent manual labor by a software engineer; double arrows represent automatic labor by a compiler.

compiler derives the final instruction streams $B$.

The largely manual approach shown in Figure 1.5—and common in contemporary software engineering—forces software engineers to take responsibility for the laborious and error-prone activities of defining, selecting, and applying every $g_i$ and, moreover, for establishing that every $g_i$ preserves the semantics of implementation $I_{i-1}$. Ideally, of course, a compiler instead of software engineers should perform this work, in a provably correct way. But although decades of research has resulted in a battery of many important optimization techniques, current compilers typically cannot apply protocol optimizations.

To further illustrate this point, Figure 1.6 shows the problem that a modern compiler (e.g., `javac`, `gcc`) faces in applying protocol optimizations. For such a compiler to decide which optimization it can—and should—apply to which parts of implementation $I_0$, it essentially needs to reconstruct specification $S$. Only then, when a compiler knows the *intention* that software engineers had when they wrote $I_0$, can it decide which portions of the interaction code admit which protocol optimization. In other words, before a compiler can optimize anything, it first needs to apply

the *inverse* of $f$ to $f(S)$ to resurrect the lost "what", $S$. Generally, however, compilers cannot do this: in going from a declarative specification to one *specific* imperative implementation, certain information gets irretrievably lost or becomes practically impossible to extract from the resulting code. Indeed, exposing shared memory to software engineers forces those software engineers to implement their protocol specifications in excessive detail, without explicitly preserving their intention. Consider, for instance, the following C code:

```
for (int i = 0; i < 10; i++)
  a[i] = some_function(rand()); // without side effects
```

If I intended *just* to assign the output of `some_function` to every `a[i]`, for random inputs `x`, a compiler can parallelize the loop. However, if I *additionally* intended the resulting array to have the same content in executions with the same random seed (e.g., to reproduce bugs), a compiler cannot parallelize the loop: in that case, the order of generating random numbers matters. Just from this code, thus, neither a compiler *nor a human* can judiciously decide about loop parallelization.

Although the previous example does not concern a protocol optimization, it well-illustrates a principle that applies also to such optimizations: typically, compilers cannot reconstruct all intention behind interaction code consisting of seemingly unrelated reads/writes to shared memory, mixed with operations of concurrency constructs for mutual exclusion. Consequently, no compiler that I know of supports protocol optimizations. Instead, software engineers have to take direct responsibility for such optimizations, thereby adding even more complexity to an already daunting task.

Incidentally, the annotations used in implicit user-threading APIs (e.g., OpenMP) serve to explicitly preserve some intention information that otherwise gets lost in translation, which the compiler leverages to produce more optimized instruction streams. For instance, with OpenMP, I can annotate the loop in my previous C code with the following pragma to inform the compiler that it may parallelize the loop:

```
#pragma omp parallel for
```

- *Issue 3: Writing modular protocol subprograms*

  Concurrency constructs such as atomic registers, semaphores, and monitors neither enforce nor encourage good practices for writing interaction code. Consequently, software engineers frequently succumb to the temptation of *not* separating interaction code from computation code. This issue differs from the previous two issues, because it does not complicate "writing code" directly. However, it does complicate many other aspects of software engineering, as also argued for by Arbab [Arb11].

Notions as "modularization" and "separation of concerns" have a long history in computer science [Dij82, Par72], and they have driven the development of software engineering practices for decades. In fact, already in the early 1970s, Parnas attributed three advantages to abiding by these principles:

> "(1) managerial—development time should be shortened because separate groups would work on each module with little need for communication; (2) product flexibility—it should be possible to make drastic changes to one module without a need to change others; (3) comprehensibility—it should be possible to study the system one module at a time." [Par72]

Also in the specific context of concurrency, researchers have studied separation of computation from interaction well before multicore processors became ubiquitous. Already in the early nineties, for instance, Arbab et al. investigated this principle in the context of the language Manifold [AHS93]. Later, separation of computation from interaction played a defining role both in the IWIM model [Arb96] and in *exogenous coordination* [Arb98]. More recently, Basu et al. advocated separation of computation from interaction in their work on the BIP component framework [BBS06].

Nevertheless, and despite Parnas' advantages, linguistic support for separation of computation from interaction has scarcely received due attention: neither concurrency constructs for mutual exclusion, nor the APIs that provide such constructs, nor the languages that support those APIs enforce modularization of interaction code. As a result, dispersing interaction code among computation code comprises a common practice for implementing protocol specifications. Such dispersal may even *seem* natural, as protocol subprograms typically consist not only of concurrency constructs but also of basic computation constructs (e.g. conditional statements). After all, concurrency constructs alone have too little expressive power to comprehensively implement nontrivial protocol specifications. The use of basic computation constructs in interaction code obscures the conceptual distinction between workers and protocols. But natural as the resulting dispersal of interaction among computation may seem, it does harm.

To illustrate such dispersal—and its deficiencies—reconsider the producers/consumer program in Figure 1.4. One cannot easily point to coherent segments of the code that actually implement the protocol specification. Indeed, only the combination of lines 2, 3, 4, 7, 8, 18, 19, 20, 26, 27, and 28 does so. In this example, thus, I have not isolated the interaction code in a distinct module; I have not separated my concerns. Therefore, the advantages of modularization identified by Parnas do not apply. In fact, the "monolithic program" in Figure 1.4 suffers from their opposites.

($\bar{1}$) Groups of software engineers cannot independently write computation code and interaction code of monolithic programs. Moreover, software engineers cannot straightforwardly reuse computation code or interaction code of monolithic programs.

($\bar{2}$) Small changes to a protocol specification require nontrivial changes throughout its monolithic implementation. For instance, suppose that I want to restrict the producers in Figure 1.4 such that they send data only in alternating order. Implementing such turn-taking requires significant changes.

($\bar{3}$) Software engineers cannot study entangled computation and interaction code in isolation: to reason about the correctness of either, they must analyze monolithic programs in their entirety.

The impact of these shortcomings only increases when programs grow larger, interaction among threads intensifies, and protocol complexity increases—a reasonable prospect in the current multicore era.

Importantly, I have not artificially fabricated the Java program in Figure 1.4 as a strawman just to make a point. Instead, I meticulously derived that program from pseudocode in Ben-Ari's textbook on concurrent and distributed programming [Ben06]. This shows that computer scientists and lecturers actually teach and encourage students to disperse interaction code among computation code.

To summarize, concurrency constructs for mutual exclusion give rise to three major issues: (i) they complicate writing correct protocol subprograms, because they complicate reasoning about programs' behavior, (ii) they complicate writing efficient/scalable protocol subprograms, because they fail to preserve important intention information, which makes it impossible for compilers to automatically perform protocol optimizations on behalf of software engineers, and (iii) they complicate writing modular protocol subprograms, because they neither enforce nor encourage syntactic separation of computation code from interaction code. Concurrency constructs for mutual exclusion, thus, seem to constitute an inadequate idiom for implementing protocol specifications. For instance, software engineers should write *directly* that a worker sends two floats and receives an array of reals for a response—not indirectly that a thread allocates shared memory and performs pointer arithmetic. Or, software engineers should write *directly* that communication between two workers inhibits interaction among other workers—not indirectly that threads acquire and release semaphore permits. Or, software engineers should write *directly* that workers exchange data synchronously—not indirectly that threads wait on a monitor until they get notified. Instead, a suitable level of abstraction should enable software engineers to write *directly* the intention behind their protocols—not indirectly the lower-level mechanics. Compilers should take care of the latter.

If only software engineers, language designers, and computer scientists could abolish concurrency constructs for mutual exclusion. Manual manage-

ment of shared memory, however, necessitates the use of such constructs to prevent data races. Therefore, to really abolish concurrency constructs for mutual exclusion, software engineers need a new level of abstraction that hides shared memory from them, far above implicit user-threading APIs. Effectively, such a new level of abstraction further narrows the remaining conceptual gap in Figure 1.2.

## Partial Solutions

*Transactional memory* provides a means of controlling concurrent accesses to shared memory as an alternative to concurrency constructs for mutual exclusion. Although originally described by Knight in the late 1980s and popularized by Herlihy and Moss in the early 1990s [Kni86, HM93], the advent of multicore processors caused a renewed interest in transactional memory from both academia and industry. Support for transactional memory can exist in hardware or in software. Below, I focus on the software variant, first described by Shivat and Touitou [ST97]. Primarily, transactional memory supports *transactions*: sequences of reads/writes to shared memory that occur atomically. Whenever two running transactions access the same memory location, one of these transactions aborts, rollbacks all the changes it has made so far, and reruns itself. The other transaction may proceed. Whenever a transaction runs to completion without conflicting memory accesses, it commits all the changes it has made. Because the implementation of a transactional memory API manages transactions transparent to software engineers, higher-level transactions should simplify programming compared to lower-level concurrency constructs for mutual exclusion. As such, transactional memory addresses the first issue in the previous subsection.

Although every single transaction corresponds to a single protocol, not every single protocol corresponds to a *single* transaction: generally, the implementation of a protocol may require multiple transactions. As far as I know, no existing transactional memory API provides constructs for composing full protocol subprograms out of transactions as first-class entities in a structural way. As such, transactional memory fails to address the third issue in the previous subsection. Moreover, transactions typically consist of low-level computation code. Thus, although transactional memory seems more high-level than concurrency constructs for mutual exclusion, it does not raise the level of abstraction high enough: by lack of structural ways to implement protocol specifications as first-class entities and because transactions consist of low-level code, software engineers leave still too much of their intention implicit, thereby inhibiting compilers from performing protocol optimizations. The fact that decades after its inception, and after a great proliferation of interest and research from the early 2000s onward, performance still remains a major issue with transactional memory seems to support this view [CBM+08, Her14]. Thus, transactional memory comprises only a partial solution to the issues in the previous subsection.

*Algorithmic skeletons*, introduced by Cole in the late 1980s [Col88], provide

software engineers a means of writing programs by composing templates of common patterns of parallel computation and interaction. Algorithmic skeleton APIs conveniently hide all workers and protocols inside their implementation, thereby completely relieving software engineers from the task of implementing protocol specifications. While several standalone algorithmic skeleton APIs exist [GL10], also some of the thread pool APIs discussed in Section 1.1 provide simple algorithmic skeletons. Algorithmic skeleton APIs seem useful in cases where programs can indeed break down into the algorithmic skeletons provided by those APIs. In other cases, software engineers still need to manually address The Protocol Concern, resort to concurrency construct for mutual exclusion to implement their protocol specifications, and consequently suffer from the issues in the previous subsection. Thus, algorithmic skeleton APIs comprise a complete solution to the issues in the previous subsection if applicable, no solution otherwise, and therefore only a partial solution in general. However, the principles behind algorithmic skeletons—instead of their prepackaged implementations in APIs—seem generally useful. After all, algorithmic skeletons essentially constitute *parallel design patterns* that can help software architects and engineers in specifying and implementing their parallel programs [ABD+09, MSM05, MRR12], much in the same way as the classical software design patterns help in developing object-oriented programs.

## 1.3 Contribution & Organization

### Abstract Proposal

In the previous section, I argued that concurrency constructs for mutual exclusion cause three issues, each of which makes addressing The Protocol Concern problematic. To abolish such constructs, software engineers need a new level of abstraction that hides shared memory, far above implicit user-threading APIs. In this thesis, I present such a new level of abstraction.

Figure 1.7 shows the main idea. I aim to provide software engineers an *intention-expressing* level of abstraction with interaction-explicit constructs for writing protocol subprograms. In the resulting software engineering workflow, shown in Figure 1.8, software engineers still write their worker subprograms in a *general-purpose language* (GPL) for computation, such as Java or C. Almost orthogonally, however, software engineers write their protocol subprograms in a complementary *domain-specific language* (DSL) for interaction. In the words of Van Deursen et al. [vDKV00], such a DSL "is a programming language that offers, through appropriate notations and abstractions [for interaction], expressive power focused on, and usually restricted to, a particular problem domain [namely implementing protocol specifications]." Beside protocol subprograms, a DSL for interaction should also allow software engineers to write a simple *main subprogram* for establishing proper links between worker subprograms and protocol subprogram. To obtain instruction streams for a program so composed, a DSL compiler first derives code in the GPL from the protocol

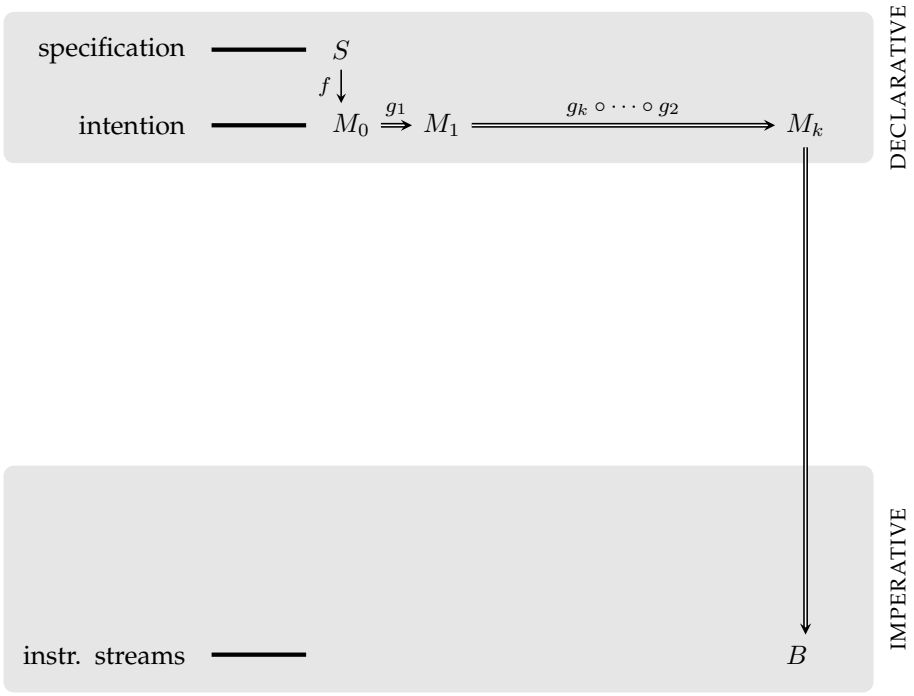


Figure 1.7: From a declarative specification $S$ to its imperative implementation as instruction streams $B$, including $k$ optimizations between intermediate declarative implementations $M_0, \ldots, M_k$ (at an intention-expressing level). Single arrows represent manual labor by a software engineer; double arrows represent automatic labor by a compiler.

subprograms and the main subprogram in the DSL. After this first compilation step, the full program exists as GPL code. In a second compilation step, a GPL compiler derives instruction streams from the GPL code, as usual.

Conceptually, every worker accesses only its own local memory. To interact with each other, then, workers perform blocking I/O *operations* on *ports*. Ports interface workers to their environment; they constitute the boundary between workers and protocols. Every worker has its own set of ports, and every port owned by a worker plays one of two *roles* relative to that worker: *output ports* allow workers to *offer* data to (the other workers in) their environment through `put` operations, while *input ports* allow workers to *accept* data from (the other workers in) their environment through `get` operations. Whenever a worker `puts` a datum to an output port, that worker does not know whereto that datum goes. Similarly, whenever a worker `gets` a datum from an input port, that worker does not know wherefrom that datum comes (cf. *exogenous coordination* [Arb04]). Only the protocol—seen as an active entity—"knows" and "decides" about how data *flow* between ports. Whenever a worker performs

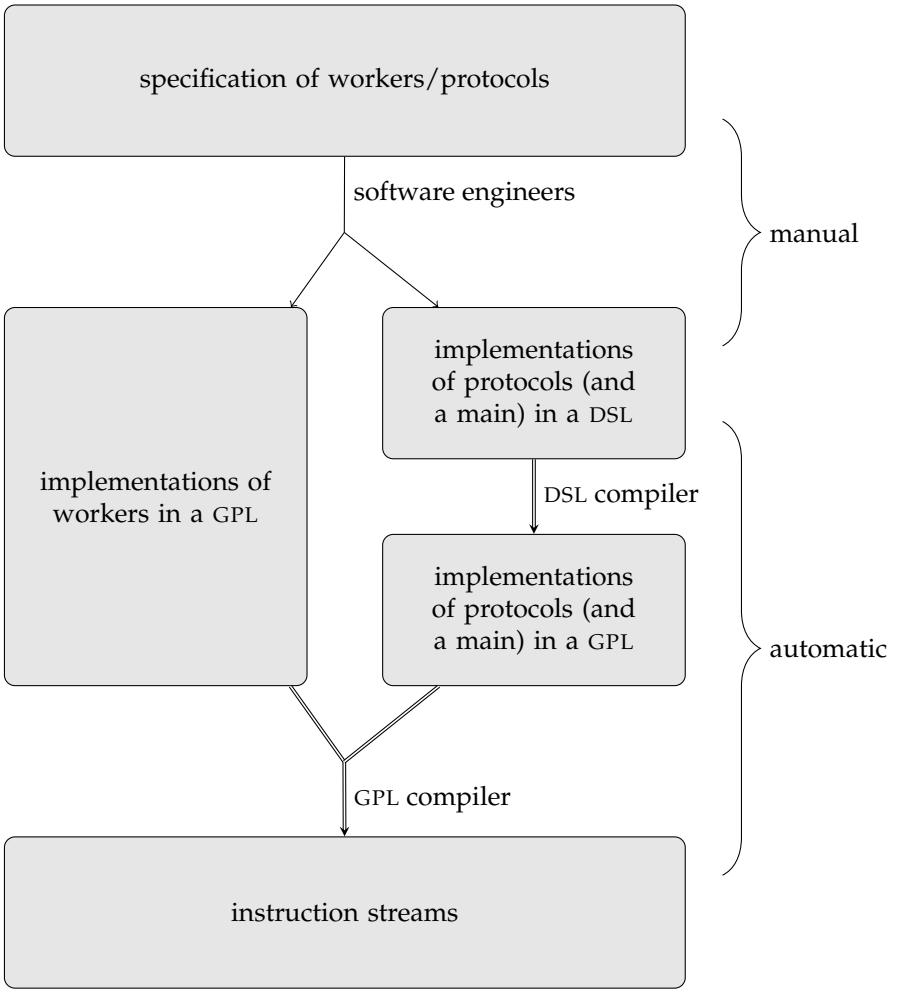


Figure 1.8: Software engineering with an interaction level of abstraction

an I/O operation on a port, that operation becomes *pending* on that port. At the same time, the worker becomes *suspended* and will not *resume* until the protocol on the other side of the port—again seen as an active entity—has *completed* the I/O operation. Whenever an I/O operation completes on a port, the worker that performed that I/O operation *exchanges* a datum *through* that port. Importantly, qualifiers "input" and "output" do not state an inherent property of a port but merely a role that a port plays *from a particular perspective*. For instance, I call the output port (from the perspective) of a worker an input port (from the perspective) of a protocol, and vice versa. Because protocols—not workers—constitute the primary subject of study in this thesis, henceforth, I qualify ports

```
1  public interface OutputPort {
2    public void put(Object datum) throws InterruptedException;
3    public void putUninterruptibly(Object datum);
4    public void resume() throws InterruptedException;
5  }

6  public interface InputPort {
7    public Object get() throws InterruptedException;
8    public Object getUninterruptibly();
9    public Object resume() throws InterruptedException;
10 }
```

Figure 1.9: Java API for ports

as "input" or "output" by default from a protocol perspective, unless explicitly stated otherwise.

As suggested above ("every worker accesses only its own local memory"), and crucially important for resolving the first issue on page 10, `put` and `get` have *value-passing* semantics instead of *reference-passing* semantics: whenever a worker exchanges a datum on a port, the run-time system that implements `put`/`get` should make a deep copy of that datum, no matter its size. Although value-passing semantics (i.e., conceptually private memory) makes software engineers' job of reasoning about their programs easier than reference-passing semantics (i.e., conceptually shared memory), the necessary run-time copying of data requires substantial resources. Fortunately, through static code analysis techniques for worker subprograms, compilers may—transparent to software engineers—determine when value-passing and reference-passing coincide and substitute the latter for the former. For instance, if a worker `put`s a variable to a port and never accesses/mutates that variable in the future, and if the datum in this variable flows to only one other worker, the run-time system does not need to copy that datum. Van de Nes studied compilation techniques for substituting reference-passing for value-passing in his MSc thesis [vdN15]. I do not discuss such techniques further in this thesis, because they involve analyses of computation code; such analyses lie beyond my current scope. Abstracting away this class of optimizations, for the examples in this thesis, I stipulate that software engineers judiciously substituted reference-passing for value-passing, whenever safe and necessary. After all, although discouraged for nonexperts to simplify programming, value-passing allows software engineers to emulate reference-passing by having workers exchange references *as values* and, symmetrically, by having workers interpret such values as references to a shared medium (e.g., memory, a file system, or online resources).

Figure 1.9 shows a Java API for ports, whose implementation I present in Chapter 4; Figure 1.10 shows another version of the producers/consumer program of before, which uses this API. (These figures constitute one of the rare exceptions in this thesis, where I qualify ports as "input" or "output" from the perspective of workers.) More precisely, Figure 1.10 shows only the worker subprograms of the full program. In this new version, the producers and the

```
1  public class Producer extends Thread {
2    private OutputPort port;
3
4    public Producer(OutputPort port) {
5      this.port = port;
6    }
7
8    public void run() {
9      while (true) {
10       Object datum = Thread.currentThread().getId();
11       this.port.putUninterruptibly(datum);
12 } } }

13 public class Consumer extends Thread {
14   private InputPort port;
15
16   public Consumer(InputPort port) {
17     this.port = port;
18   }
19
20   public void run() {
21     while (true) {
22       Object datum = this.port.getUninterruptibly();
23       System.out.println(datum);
24 } } }

25 public class ModularProducersConsumerProgram {
26   public ModularProducersConsumerProgram(
27       OutputPort A, OutputPort B, InputPort C) {
28
29     (new Producer(A)).start();
30     (new Producer(B)).start();
31     (new Consumer(C)).start();
32 } }
```

Figure 1.10: Modular producers/consumer program in Java

consumer interact with each other only via ports, passed to the program as actual parameters of the constructor: one producer thread has access to `OutputPort A`, the other producer has access to `OutputPort B`, and the consumer has access to `InputPort C`. To this program, Parnas' advantages of modularization apply. First, groups of software engineers can write the protocol subprogram in the DSL independently from the worker subprograms in Java. Moreover, software engineers can easily reuse the protocol subprogram. Second, software engineers can change the protocol subprogram without touching the worker subprograms. Third, software engineers can analyze the protocol subprogram separate from the worker subprograms.

By using a GPL for writing computation code and a DSL for writing interaction code, software engineers syntactically separate worker subprograms from protocol subprograms. The natural modularization of protocols resulting from this separation resolves Issue 3 on page 13 and promotes protocol reuse. More-

over, a *true* intention-expressing DSL for interaction preserves enough information for a compiler to perform protocol optimizations (without the need to reconstruct such intention information). As such, this compiler relieves software engineers from the responsibility of manually performing protocol optimizations, by automatically selecting and applying such optimizations itself. The compiler designer of such optimizations, instead of software engineers, becomes responsible for proving the correctness of those optimizations. Formally establishing such desirable properties, however, remains a one-shot activity (cf. ad-hoc reasoning about every manually protocol-optimized program with concurrency construct for mutual exclusion). Typically, because DSL code has a higher level of abstraction than GPL code, proving properties of protocol optimizations in this way becomes simpler and more mathematically elegant than reasoning about GPL code. As such, a true intention-expressing DSL resolves Issue 2 on page 10; I present such a DSL in this thesis and give concrete examples of protocol optimizations that its compiler supports. Finally, because workers no longer interact with each other through shared memory but through value-passing I/O operations and protocols, schedulers no longer affect the correctness of programs. And because interaction no longer occurs as a byproduct of seemingly unrelated reads/writes and concurrency constructs, but as first-class entities in DSL code, interaction and protocols become explicit artifacts. Both these improvements simplify reasoning about protocols and interaction among workers, thereby resolving Issue 1 on page 13.

## Concrete Instantiation

In this thesis, I present the theory and practice of a true intention-expressing DSL for interaction, called *FOCAML*, pronounced "foe camel", with emphasis on "foe". Despite its similar name—an acronym whose meaning I clarify in Chapter 3—FOCAML has no relation to the OCAML language.

As any DSL for interaction, FOCAML provides constructs for implementing protocol specifications. Essentially, such specifications define the *admissible* instances of interaction among workers during runs of programs. A protocol, thus, imposes *constraints* on which instances of interaction may occur when (i.e., which I/O operations may synchronously complete on which ports in which instant). In FOCAML, software engineers represent such constraints compositionally as *multiplication expressions* over a special kind of finite *automata*; I further motivate my choice for automata on page 32. By their definition, these automata capture the intention behind protocols as precisely as possible. Drawing strong inspiration from previous work by Baier et al. [BSAR06] and Arbab [Arb04], I first define FOCAML's semantics and syntax, in that order. My main contribution, then, consists of the theory and practice of a FOCAML compiler, including protocol optimizations. As such, this thesis essentially instantiates Figure 1.7, for FOCAML. Figure 1.11 shows this instantiation.

- In Chapter 2, I define the automata-based semantics of FOCAML.

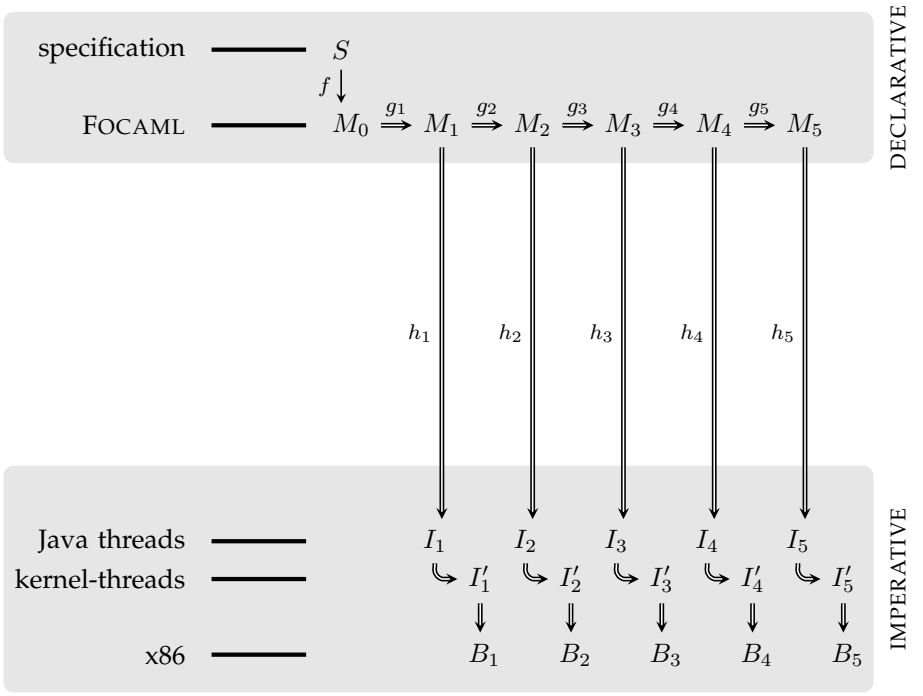


Figure 1.11: Instantiation of Figure 1.7

- Chapter 3 covers arrow $f$ in Figure 1.11. In this chapter, I define the syntax of FOCAML, which software engineers can use for writing their protocol subprograms; essentially, FOCAML allows software engineers to concisely and compositionally write multiplication expressions over automata.

  In this chapter, I also present example FOCAML code—including code for protocols in NASA's well-established NAS Parallel Benchmarks [BBB+91, BBB+94]—which I use throughout this thesis both as running examples and for experimentation with my FOCAML compiler.

- Chapter 4 covers arrows $g_1$ and $h_1$ in Figure 1.11. In this chapter, I discuss basic compilation approaches for FOCAML.

  In arguably the most natural compilation approach, a FOCAML compiler generates a thread for every automaton in a multiplication expression over automata. At run-time, the resulting threads concurrently run and use a consensus algorithm to synchronize their behavior (i.e., synchronize the firings of their transitions). Natural as this may seem, the process of reaching consensus inflicts significant overhead. Transformation $g_1$ improves this approach by serializing all parallelism among automata already at compile-time, by computing their full product, thereby avoiding

the need for a consensus algorithm at run-time.

In this chapter, I also present a basic FOCAML-to-Java compiler, which applies $g_1$ before generating Java code (i.e., transformation $h_1$), and discuss experiments performed with this compiler.

- Chapter 5 covers arrows $g_2$ and $h_2$ in Figure 1.11. In this chapter, I discuss an improved compilation approach for FOCAML, formalized and proven correct at the higher level of automata instead of at the lower level of GPL code.

  As the experimental results in Chapter 5 show, transformation $g_1$ can cause both compile-time and run-time performance problems related to (i) exponential growth of serialized automata and (ii) oversequentialization of serialized automata. Transformation $g_2$ compensates for those problems by applying more selective serialization, thereby recovering useful parallelism among automata.

  In this chapter, I also present an improved FOCAML-to-Java compiler, which applies $g_1$ and $g_2$ before generating Java code (i.e., transformation $h_2$), and discuss experiments performed with this compiler.

- Chapter 6 covers arrows $g_3$ and $h_3$ in Figure 1.11. In this chapter, I discuss an improved compilation approach for FOCAML, formalized and proven correct at the higher level of automata instead of at the lower level of GPL code.

  As the experimental results in Chapters 4 and 5 show, the neutral element for automaton product (modulo some behavioral congruence) does not behave neutrally with respect to performance: it degrades performance. This phenomenon does not stand by itself but rather symptomizes a more fundamental problem. Transformation $g_3$ solves this problem, thereby improving the performance of nearly every FOCAML program.

  In this chapter, I also present an improved FOCAML-to-Java compiler, which applies $g_1$, $g_2$, and $g_3$ before generating Java code (i.e., transformation $h_3$), and discuss experiments performed with this compiler.

- Chapter 7 covers arrows $g_4$ and $h_4$ in Figure 1.11. In this chapter, I discuss an improved compilation approach for FOCAML, formalized and proven correct at the higher level of automata instead of at the lower level of GPL code.

  As part of firing a transition at run-time, a thread for an automaton must solve one or more constraint satisfaction problems. However, the use of *general* solvers and algorithms inflicts significant overhead. Transformation $g_4$ reduces such overhead by computing a *dedicated* solver for constraint satisfaction problems at compile-time, thereby minimizing the time spent on solving at run-time.

In this chapter, I also present an improved FOCAML-to-Java compiler, which applies $g_1$, $g_2$, $g_3$, and $g_4$ before generating Java code (i.e., transformation $h_4$), and discuss experiments performed with this compiler.

- Chapter 8 covers arrows $g_5$ and $h_5$ in Figure 1.11. In this chapter, I discuss an improved compilation approach for FOCAML, formalized and proven correct at the higher level of automata instead of at the lower level of GPL code.

  As the experimental results in Chapter 7 show, the performance of compiler-generated code for certain protocols degrades as the number of workers increases, even though one may reasonably expect performance to stay constant. Here, problematically, threads defined by such compiler-generated code check the transitions in their corresponding automata for enabledness only one after the other, in linear time. As the number of transitions often increases with the number of workers, also the run time of such threads increases. Transformation $g_5$ resolves this issue by identifying cases as this and by subsequently injecting data structures that allow threads to check linearly many transitions in constant time.

  In this chapter, I also present an improved FOCAML-to-Java compiler, which applies $g_1$, $g_2$, $g_3$, $g_4$ and $g_5$ before generating Java code (i.e., transformation $h_5$), and discuss experiments performed with this compiler.

I conclude this thesis in Chapter 9 with a summary and future work. As in this chapter, throughout this thesis, I discuss related work whenever relevant—"by need"—instead of in separate chapters or sections.

The concrete syntax of FOCAML, additional definitions, and detailed proofs of all lemmas and theorems in this thesis appear in a separate technical report [**Jon**16].

With Arbab, Halle, and Santini, I previously published parts of this thesis in seven conference/workshop papers [**J**A13a, **J**A13b, **J**A14, **J**A15a, **J**A15b, **J**HA14a, **J**SA14] and in two journal papers [**J**A16, **J**SA15], all as first/lead author. This thesis, however, contains also a significant body of new material that I have not yet submitted for publication, notably my experimental results. At the beginning of every new section, I indicate the publication status of the material presented in that section. The other six conference/workshop papers [**J**A11, **J**CP12, **J**HA14b, **J**KA11, **J**KA16, **J**SS$^+$12] and the other three journal papers [**J**A12, **J**CP16, **J**SS$^+$14] that I published as first/lead author during my PhD project, with Afsarmanesh, Arbab, Clarke, Halle, Kappé, Krause, Proença, Santini, and Sargolzaei, do not fit the scope of this thesis; I cite some of them at the appropriate places in this thesis, though.